



Systems and Technology Group

Experience with Formal Methods, Especially Sequential Equivalence Checking

Mark Firstenberg
firsten@us.ibm.com

From Bob Fredieu's Talk

Statements

'Current Trends: Boredom'

'Don't Waste Money'

'What has not helped:

Formal Verification –

Can't get people to do it'

Comments

What?!

Explore Something New!

Pursue Efficiency/

Improve Quality

Wrong Expectations?

Benefits Unknown?

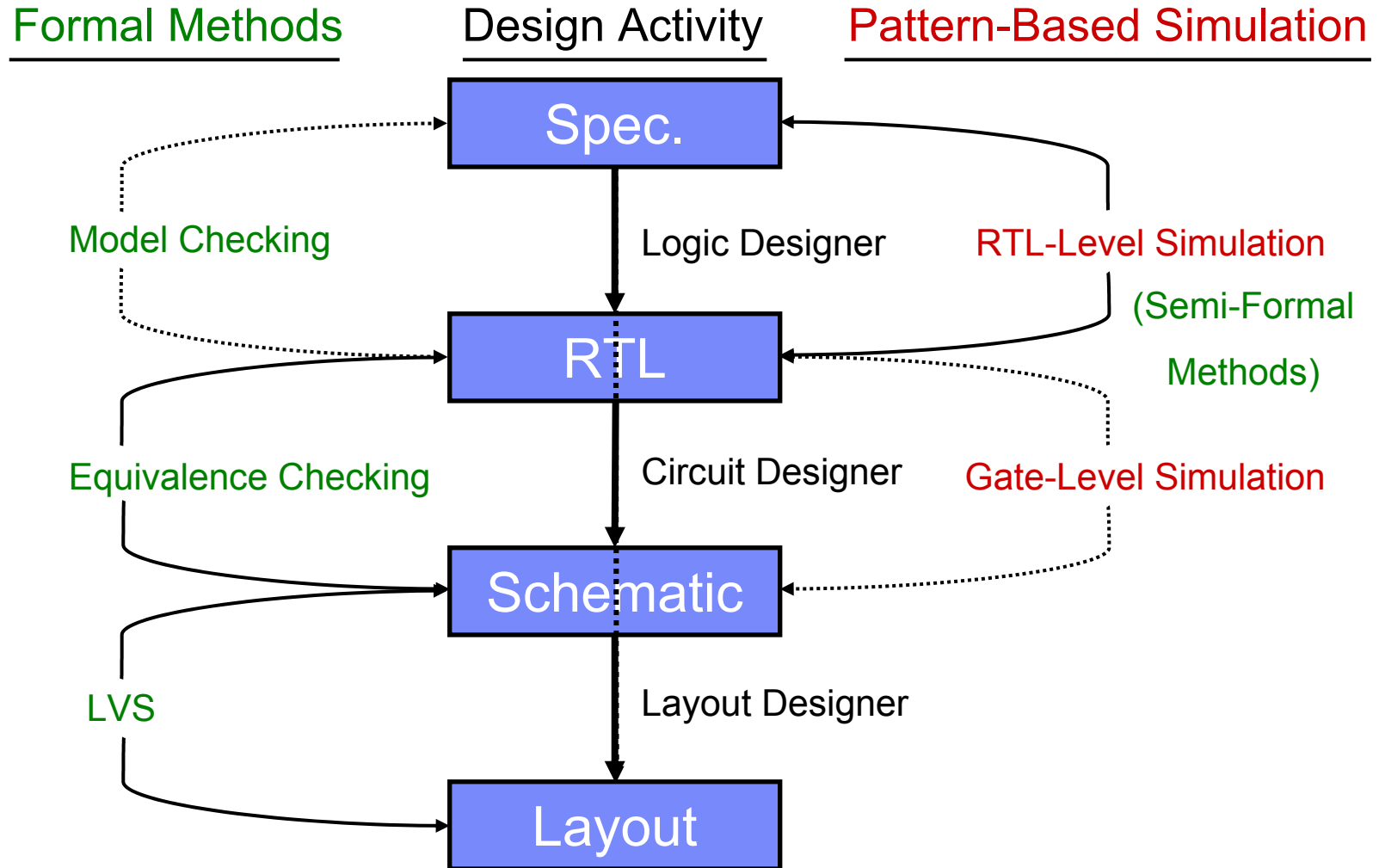
Purpose of this talk:

Provide a user's perspective of Formal Methods

Why Consider Formal Methods?

- **Formal methods are preferable to pattern based simulation since formal proofs:**
 - **Provide complete coverage** while pattern based simulation does not,
 - **Require a fraction of the computer resources** pattern based simulation does, and
 - **Require fewer human resources** than pattern based simulation does.
- **At least that is the theory . . .**

Verification Process: New Design



Verification Process: New Design (notes)

- **Overall task is to verify that the design specification and the final product (i.e. layout) are functionally equivalent**
- **Since the level of abstraction between specification and final product is too great, break the design task into activities which produce functionally equivalent representations:**
 - Logic designer translates the specification to RTL by hand
 - Circuit designer (or synthesizer) translates RTL to schematics
 - Layout designer (or layout tool) translates schematics to layout
- **Then verify that adjacent design representations are functionally equivalent**

Verification Process: New Design (notes)

■ **Pattern-Based Simulation Methods**

– **RTL-Level Simulation**

- Required if the specification is not machine readable (typically the case)

– **Gate-Level Simulation**

- Can be used for RTL to schematic comparison, but formal methods are preferable
- Typically used for POR sequence testing

Verification Process: New Design (notes)

▪ **Formal Methods**

– Model Checking

- Used to prove specific assertions about the design
- Inherently incomplete (in that all interesting assertions cannot be specified) so is used as a backup to RTL-Level Simulation

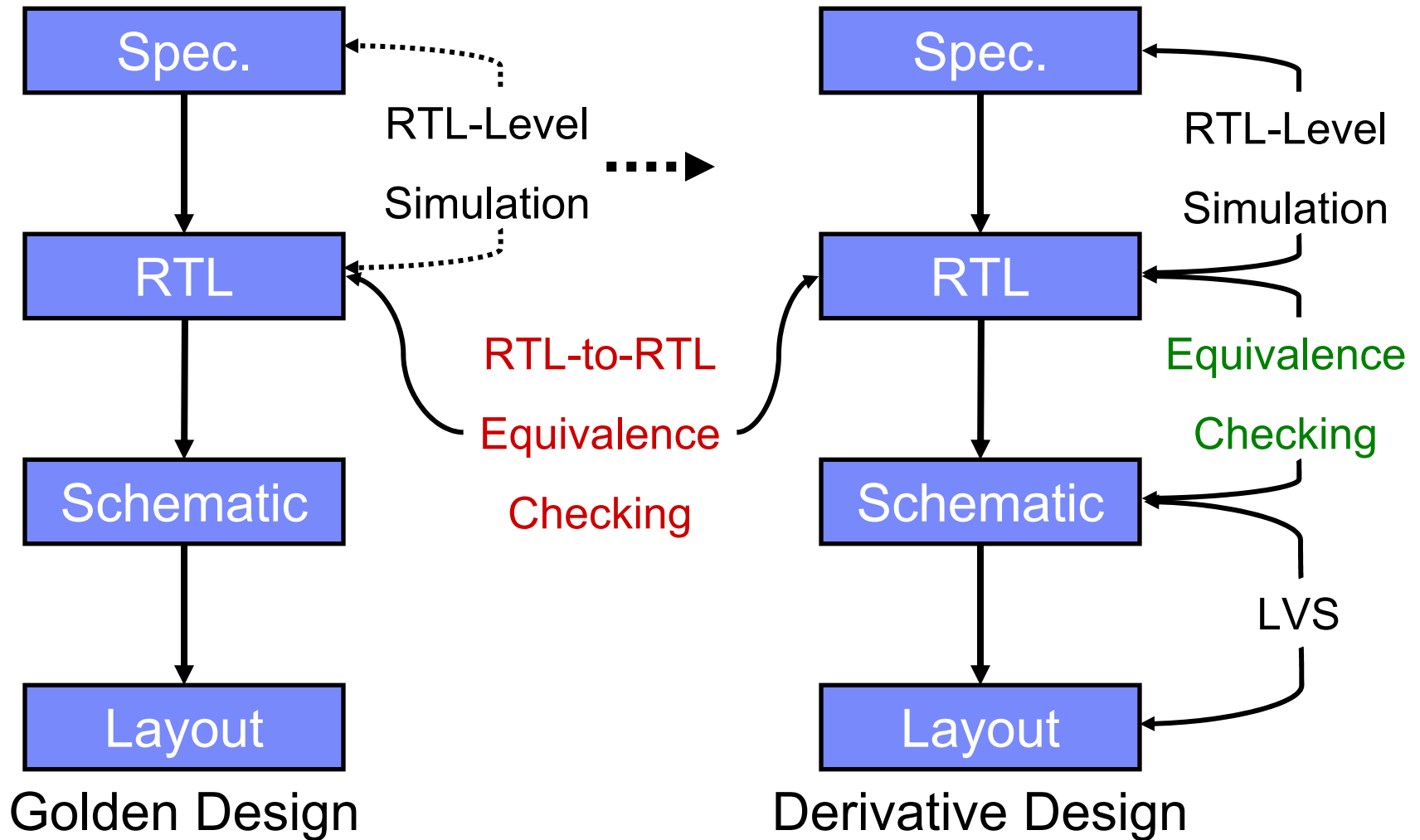
– Equivalence Checking for RTL to schematic comparison

– LVS for schematic to layout comparison

– Semi-Formal Methods

- Accelerates triggering RTL based assertions
- Is not a pure formal method since it uses RTL-Level Simulation to traverse portions of the state space

Verification Process: Derivative Design



Verification Process: Derivative Design (notes)

- **Previously verified ‘golden design’ used as a starting point**
- **Use primary verification methods for adjacent derivative design representations:**
 - RTL-Level Simulation for specification to RTL comparison
 - **Equivalence Checking** for RTL to schematic comparison
 - LVS for schematic to layout comparison
- **Bridging the gap between ‘golden’ and ‘derivative’ designs:**
 - Either migrate ‘golden model’ RTL-Level Simulation environment and tests to derivative design RTL-Level Simulation environment (effort may be significant depending on type of changes in the derivative design)
 - Or perform **RTL (golden) to RTL (derivative) Equivalence Checking** taking derivative design changes into account

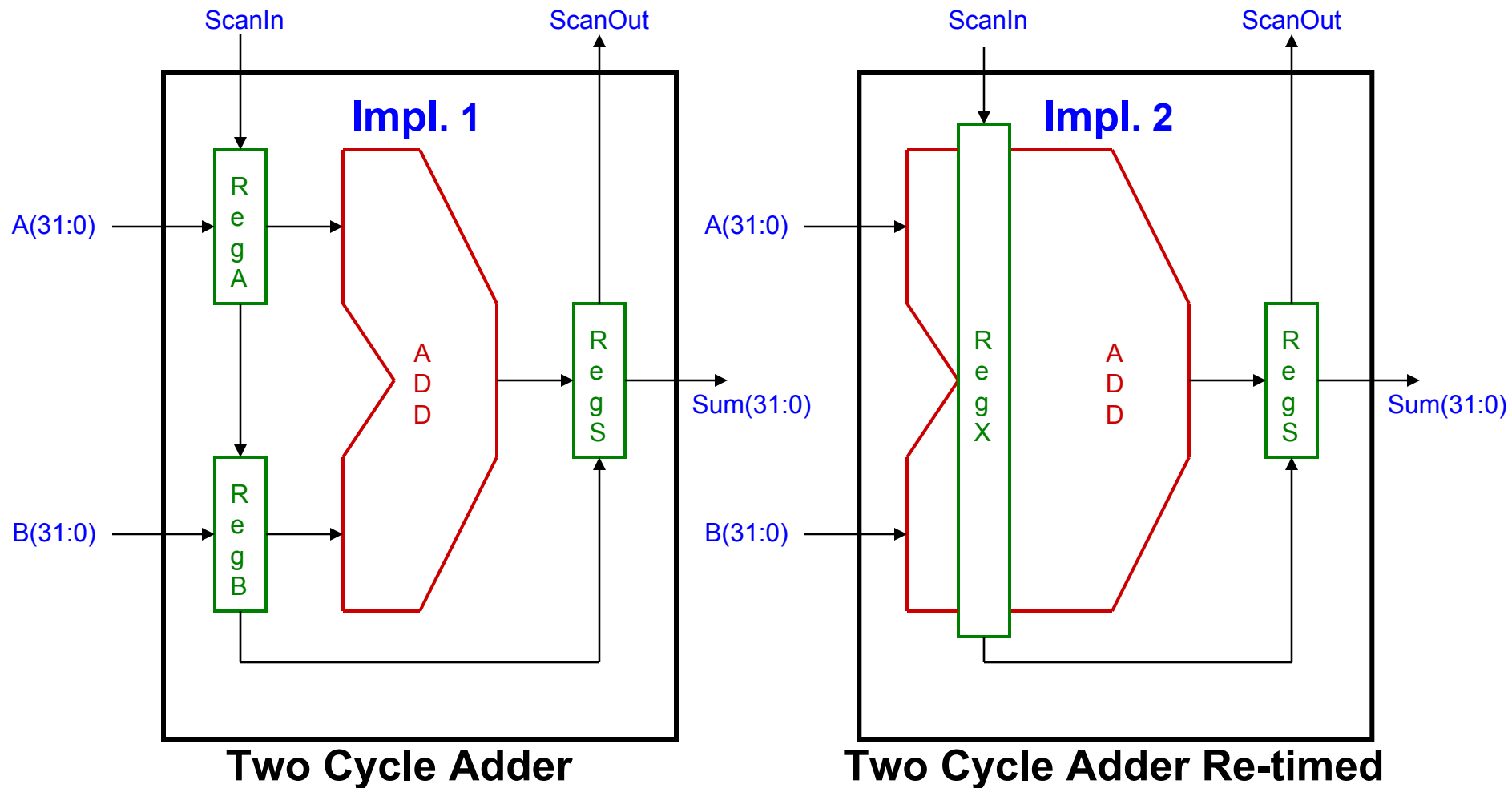
Equivalence Checking Base Definitions

- **Equivalence Checking:** The process of comparing two design representations to ensure that their digital function is the same. Note that equivalence checking says nothing about functional correctness!
- **Logic Cone:** The collection of non-state device logic which feeds a single state device within a design or a single primary output of a design. Logic cone inputs are state device outputs and/or design primary inputs.
- **Boolean Function:** The function of a single logic cone.
- **Sequential Behavior:** The digital behavior of a sequence of logic cones and any intervening state devices (i.e. multi-cycle behavior) as observed at the output of the sequential path.
- **Register Re-timing:** The movement of logic across state devices which preserves sequential behavior.

Formal Equivalence Checking Types

- **Boolean Equivalence Checking:** The process of **comparing corresponding logic cones** from two different design representations to ensure that their digital function is the same.
- **Sequential Equivalence Checking:** The process of **comparing corresponding sequential path outputs** from two different design representations to ensure that their digital function is the same.

Boolean vs Sequential Equivalence Checking Example



Boolean vs Sequential Equivalence Checking Example (notes)

- **Boolean equivalence will detect mismatches at all comparison points**
 - Impl. 1 RegA and RegB versus Impl. 2 RegX, because
 - The logic cones feeding the registers have changed.
 - If the size of RegA plus the size of RegB does not equal the size of RegX, there will be extra state devices in either Impl. 1 or Impl. 2 without corresponding state devices in the other design.
 - Impl. 1 and Impl. 2 RegS's (and thus their Sum's) will mismatch because of logic cone differences.
 - Impl. 1 and Impl. 2 ScanOut's will mismatch because of the RegS differences.

Boolean vs Sequential Equivalence Checking Example (notes)

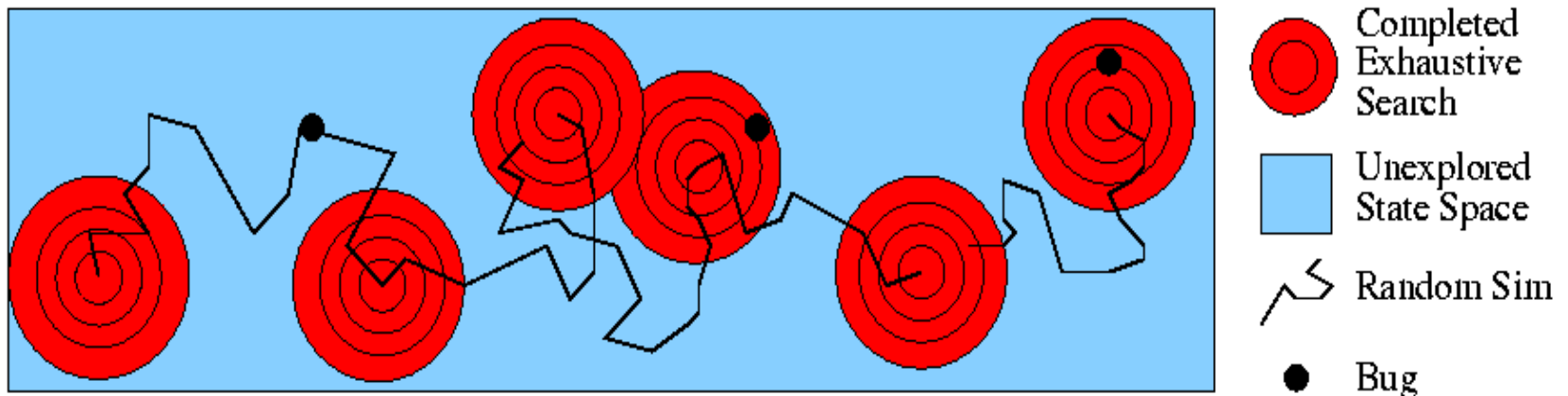
- **Sequential equivalence would find the two designs equivalent except for the ScanOut**
 - The two Sum's are functionally equivalent and
 - The two ScanOut's are different since the logic cone differences feeding the RegS's are visible during scan chain shifting.
 - Since only primary outputs are compared, a mismatch in state devices between the two designs does not necessarily matter.
 - Initial values in the 'Reg's could cause mismatches during the first two cycles, so initial state device values matter!
- **If scan operations were disabled, the two designs would be sequentially equivalent**
- **Note that this includes sequential path latency**

Boolean vs Sequential Equivalence Checking Comparison

- **Boolean equivalence checking is ‘stricter’ and more microscopic**
 - Register re-timing is reported as a mismatch.
 - State device correspondence is required.
 - Design constraints cannot propagate through state devices.
 - State device initialization does not need to be considered.
- **Sequential equivalence checking takes a more macroscopic view of the design**
 - Register re-timing is not reported as a mismatch.
 - State device correspondence is not required.
 - Design constraints applied at primary inputs propagate.
 - State device initialization does need to be considered.

Semi-Formal Methods

- **Combination of pattern-based simulation and formal methods**
 - Simulate to get deep into design's state space
 - Then use formal methods to reach design assertions
 - Accelerate determination if assertion can be triggered
- **Since simulation is involved, method is not purely formal**

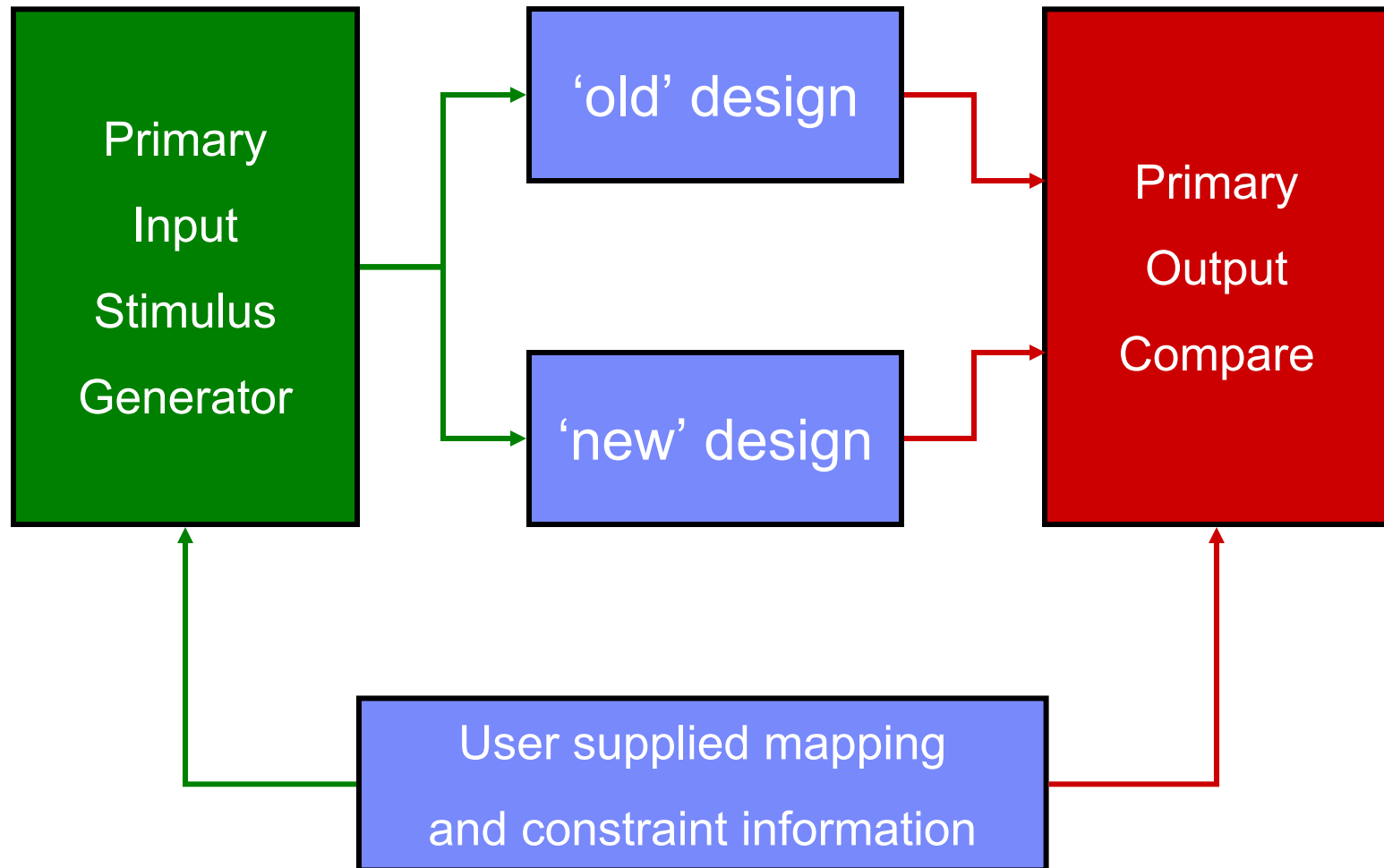


(diagram from 9/12/05 SixthSense presentation, Jason Baumgartner, et. al.)

Building a Sequential Equivalence Checker Using Formal/Semi-Formal Methods

- **Semi-Formal Methods use formal methods to determine if design assertions can be triggered**
- **If the assertion is a checker comparing corresponding outputs of two design representations, such a tool could be used to formally prove design equivalence (once the dependence on pattern-based simulation is eliminated)**
- **Since the formal methods are designed to propagate sequentially from an initial state, the result would be a Sequential Equivalence Checker**

Sequential Equivalence Checking Test Bench



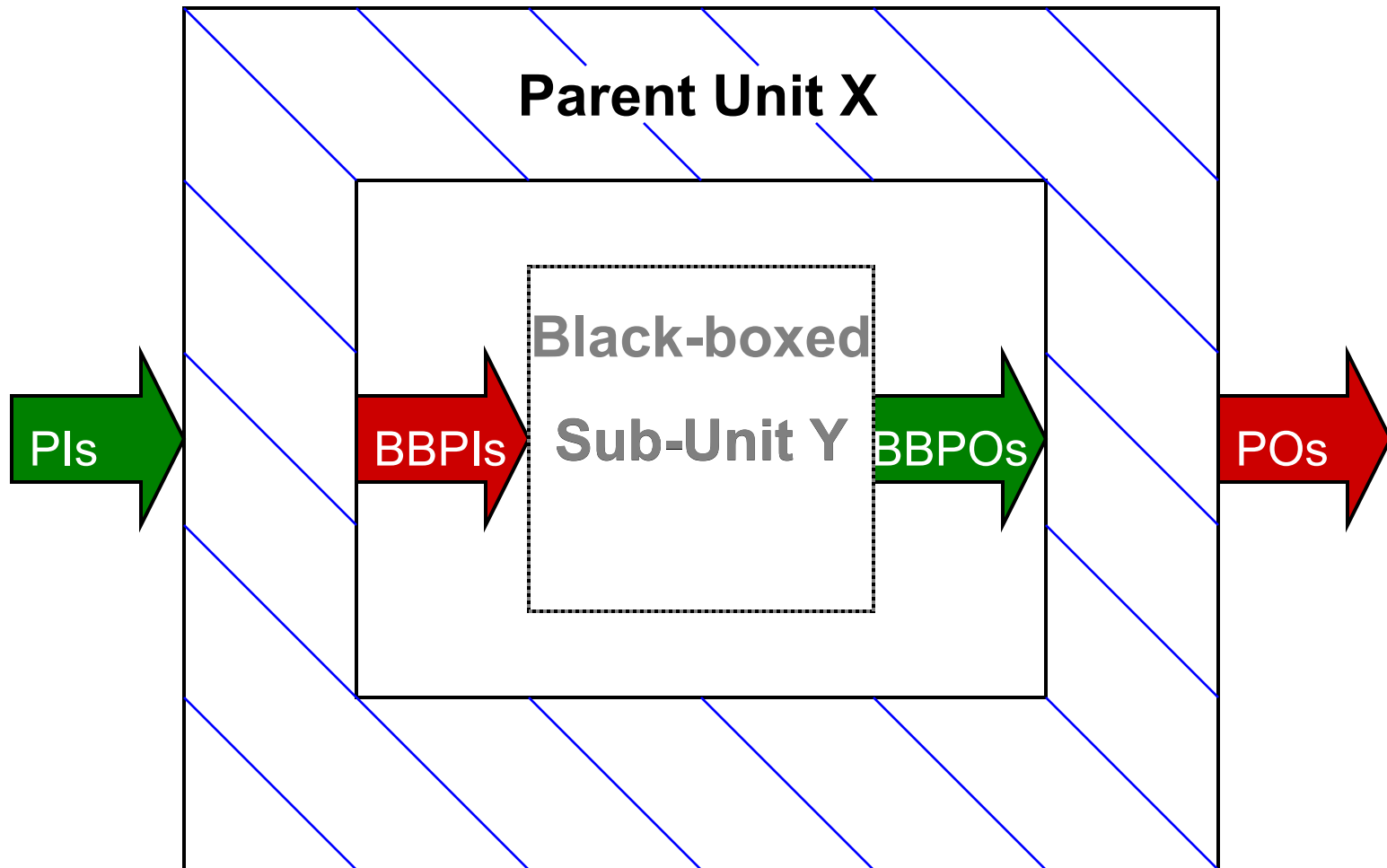
Sequential Equivalence Checking Test Bench (notes)

- **The user supplies:**
 - ‘old’ design
 - ‘new’ design
 - Mapping and constraint information
- **Sequential equivalence checking tool automatically generates the test bench**
 - Correlates output pins, connecting them to checkers
 - Correlates input pins, driven from a stimulus generator
 - Both **output checkers** and **stimulus generator** can be modified via user supplied signal mappings and input constraints

Input/Output Pin Correlation

- **By default, input/output pins from both designs are paired based on pin names**
- **For any uncorrelated input pins:**
 - Test bench allowed to stimulate them with different values
 - Could lead to ‘false mismatches’ (which are annoying!)
 - Fix/waive such warnings (via mapping file) before pursuing mismatches
- **For any uncorrelated output pins:**
 - Test bench eliminates the pins
 - Could lead to ‘false matches’ (which are not acceptable!)
 - Fix/waive such warnings (via mapping file) before declaring equivalence

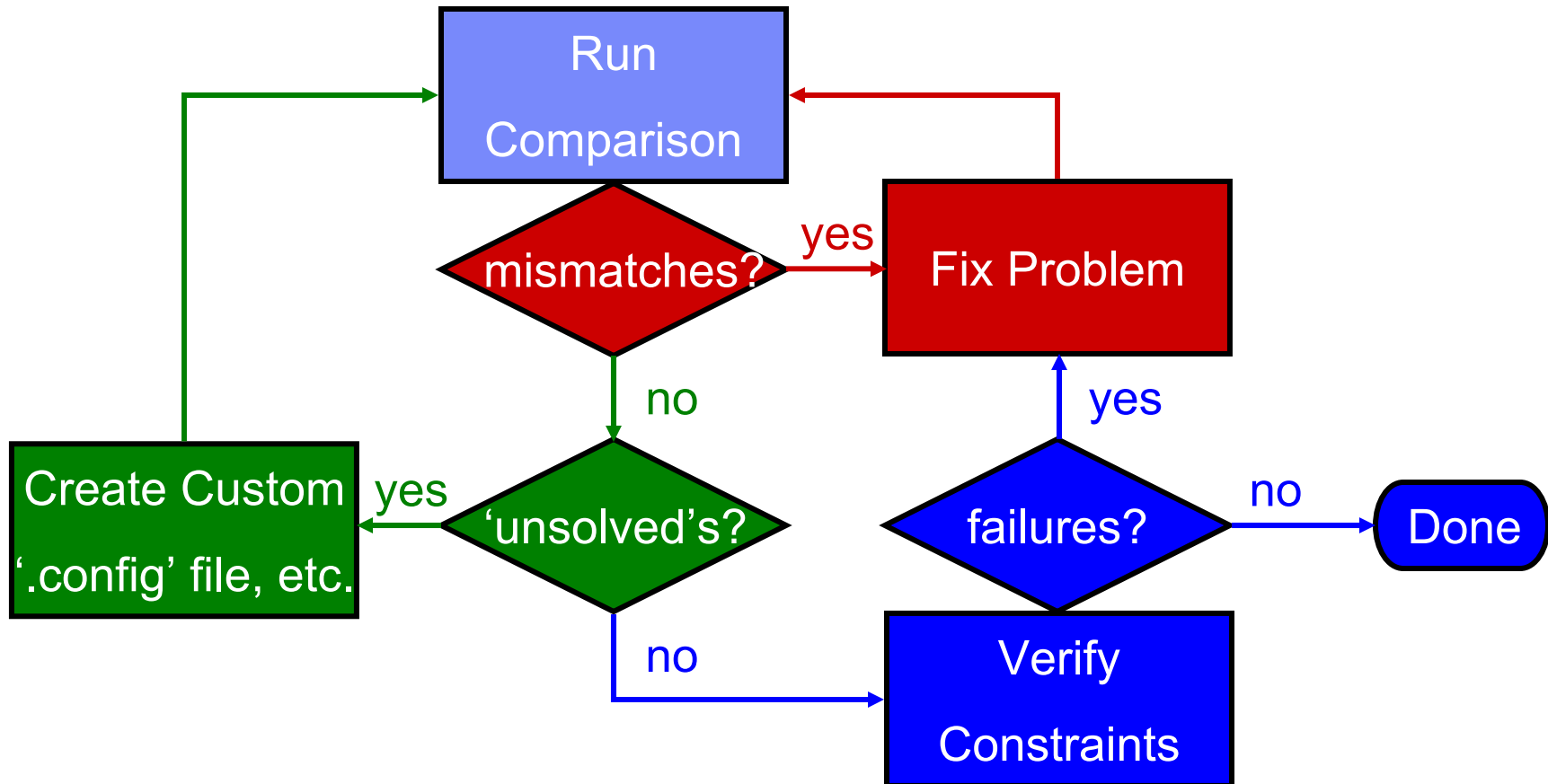
Black Boxing Design Sections



Black Boxing Design Sections (notes)

- **After ‘Sub-Unit Y’ has been ‘black boxed’, ‘Parent Unit X’s:**
 - Primary inputs will be the combination of ‘X’s Primary Inputs (PIs) and ‘Y’s Black Box Primary Outputs (BBPOs)
 - Primary outputs will be the combination of ‘X’s Primary Outputs (POs) and ‘Y’s Black Box Primary Inputs (BBPIs)
- **‘Black Boxing’ helps alleviate tool capacity issues**
 - Reduces amount of logic being checked
- **‘Black Boxing’ increases complexity in other ways**
 - Increases number of items to prove (primary outputs)
 - Increases number of items to control (primary inputs)
 - Requires separate verification of the ‘black boxed’ unit
 - Creates another equivalence checking boundary which needs to be maintained

Sequential Equivalence Checking Debug Phases

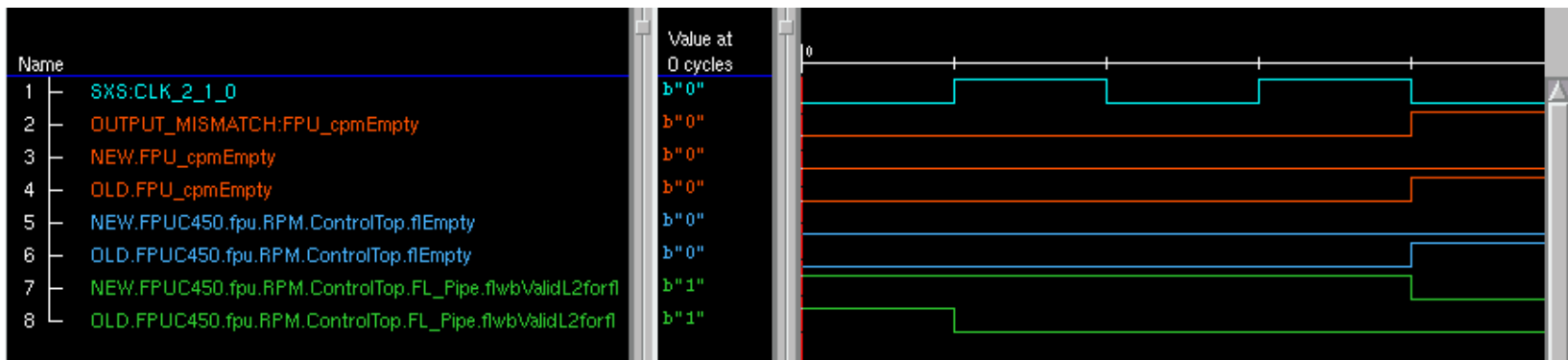


Sequential Equivalence Checking Debug Phases (notes)

- **Comparison can produce three results for each ‘property’**
 - ‘Mismatch’: designer action required
 - ‘Match’: equivalence proven
 - ‘Unsolved’: incomplete proof (neither ‘mismatch’ or ‘match’)
- **Debug Phases**
 1. Designer resolves mismatches (via design or constraints changes)
 2. Application Engineer resolves ‘unsolved’s (via ‘.config’ files, if not other methods)
 3. Designer resolves incorrect constraints
- **Debug activities can be intermixed . . .**

Debugging Sequential Equivalence Checking Mismatches

- **Trace (.aet file) is created for each mismatch:**
 - OUTPUT_MISMATCH shows when failure occurred
 - Corresponding NEW and OLD model signals available
 - Only as many cycles as needed are displayed . . .
 - . . . making it easy to trace with little or no design knowledge!



XBOX 360 Processor Follow-on: Description

- **Goals**

- Cost reduction of a multi-processor SOC
- Maintain 'mission' function and performance

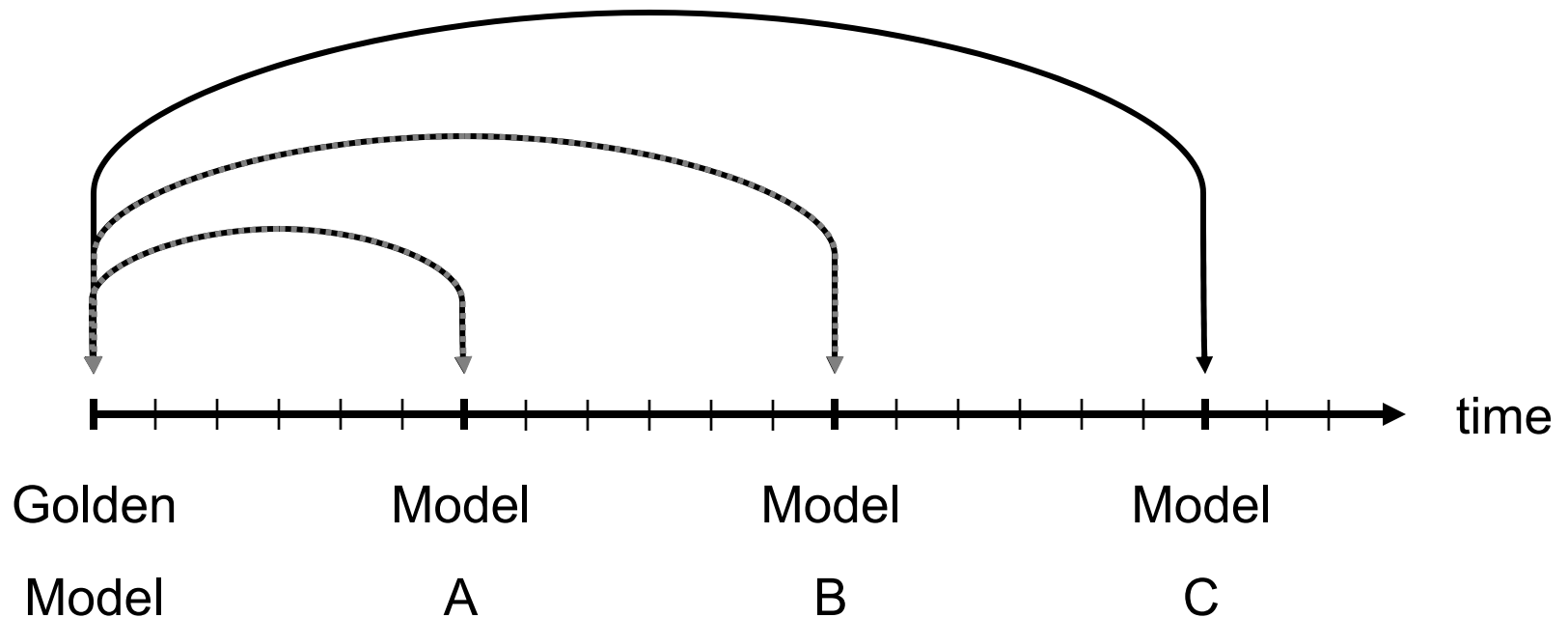
- **Design Methods**

- Process technology transfer
- 100+ changes to 'pervasive' (IBM term for 'POR/DFT/debug') function
- HDL: VHDL

- **Verification Methods**

- Pervasive function: pattern based simulation
- Mission function: combination of sequential equivalence checking and pattern based simulation (from previous design)
- Performance: sequential equivalence

XBOX 360 Processor Follow-on: 'Latest is Greatest' Comparisons



XBOX 360 Processor Follow-on: 'Latest is Greatest' Comparison (notes)

- **Always compare back to the 'golden model'**
 - Problems (and solutions) accumulate
 - A difficult problem can't be avoided (unless designed away)
 - Solutions must be carried forward (unless designed away)
 - Always a desire to get to latest model . . .
 - . . . more like traditional verification
- **Configuration management is less of an issue (mainly needed for 'golden model')**

XBOX 360 Processor Follow-on: Results

- **Sequential equivalence was used to compare the entire design**
 - ~25 design sections and ~25 arrays
 - Only the physical layer of the I/O unit (PHY macro) was not compared (since the old and new designs did not split the analog and digital functions in the same way, so comparison was not possible)
- **Design issues uncovered**
 - Removal of piping latches detected (i.e. latency difference)
 - Undocumented change discovered
- **Other discoveries**
 - Numerous signal constraint/documentation errors
 - Differences between 'pervasive' and 'mission' simulation environment's use of POR state . . .

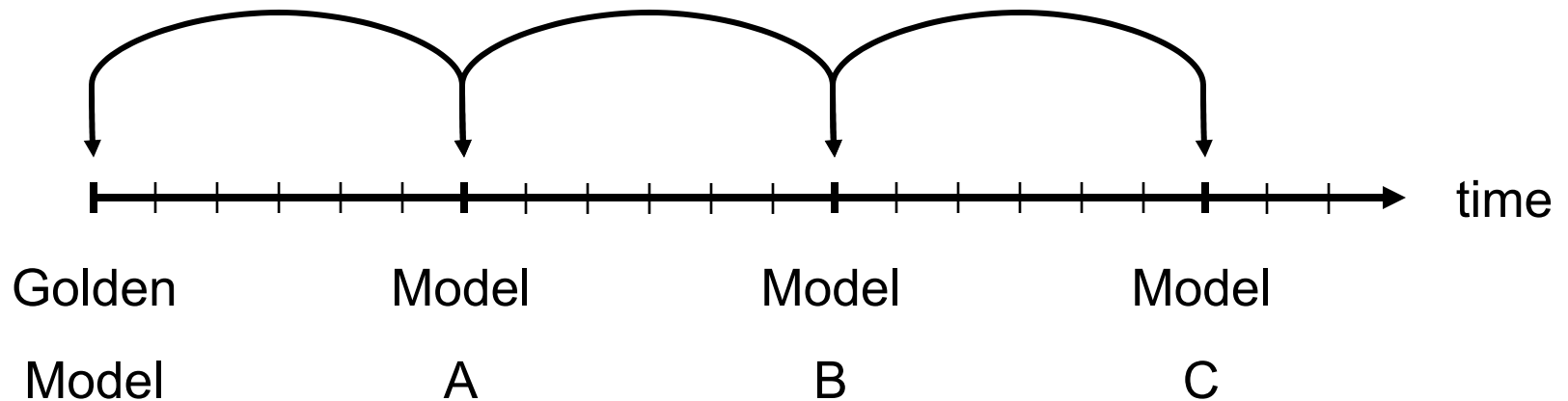
XBOX 360 Processor Follow-on: Lessons Learned

- **Only 'black box':**
 - At agreed upon equivalence checking boundaries
 - When forced to because of tool capacity issues
- **Discovering uncorrelated inputs via debugging is painful!**
- **The driver of a signal knows more about how to constrain that signal than the consumer does, but the consumer does the debugging (and thus proposes the constraint)**
- **Must verify all constraints (i.e. equivalence checking assumptions), using either:**
 - Formal Methods, or
 - Simulation assertions
- **Don't wait till the last minute to start using POR based state device initialization**
- **We needed POR results (for state device initialization) but POR is a pervasive function (which was subject to change)!**

PowerPC 464FP FPU: Description

- **Goals**
 - Cycle time improvement of an FPU
 - Use the same process technology
 - Maintain 'mission' function
- **Design Methods**
 - Significant register re-timing
 - Micro-architectural changes (while maintaining operation latency)
 - HDL: Verilog
- **Verification Methods**
 - Original design verified with many directed tests, with unknown coverage . . .
 - Sequential equivalence (to 'hold the line')
 - Pattern based simulation (concentrating on random testing)

PowerPC 464FP FPU: 'Step-Wise' Comparisons



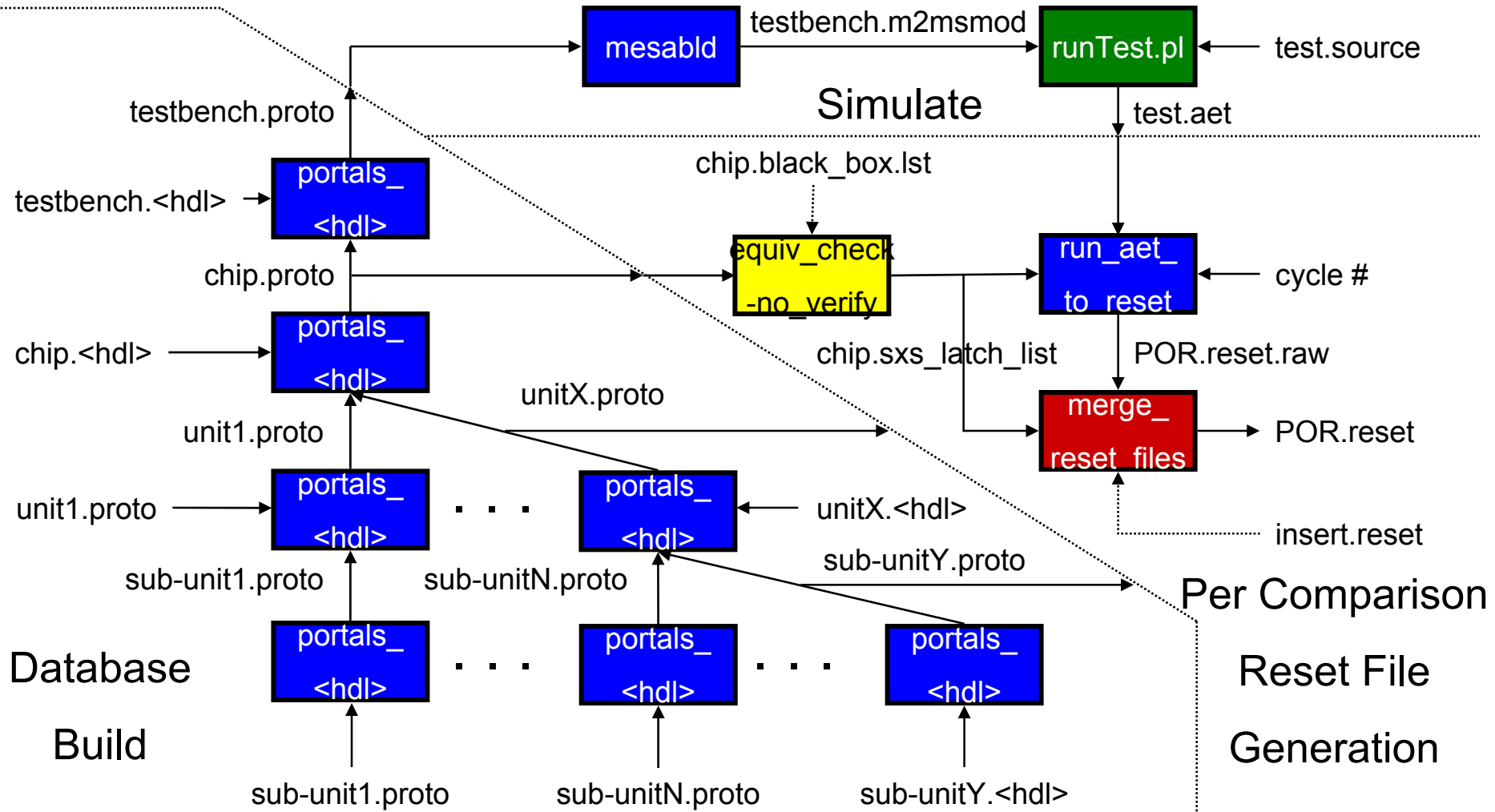
PowerPC 464FP FPU: 'Step-Wise' Comparisons (notes)

- **Based on 'A==B, B==C, therefore A==C' mentality**
 - Problems (and solutions) can be isolated
 - Usually trying to isolate design changes which will be difficult to verify (ex: register re-timings)
 - Assumes difficult design changes are staged independently!
 - Comparisons do not have to be resolved in order and debug can progress in parallel
 - Model patches may be required . . .
- **Strong configuration management required**
 - Constantly going back to old models (not just the 'golden model')!
 - Had better be versioning libraries and tools . . .

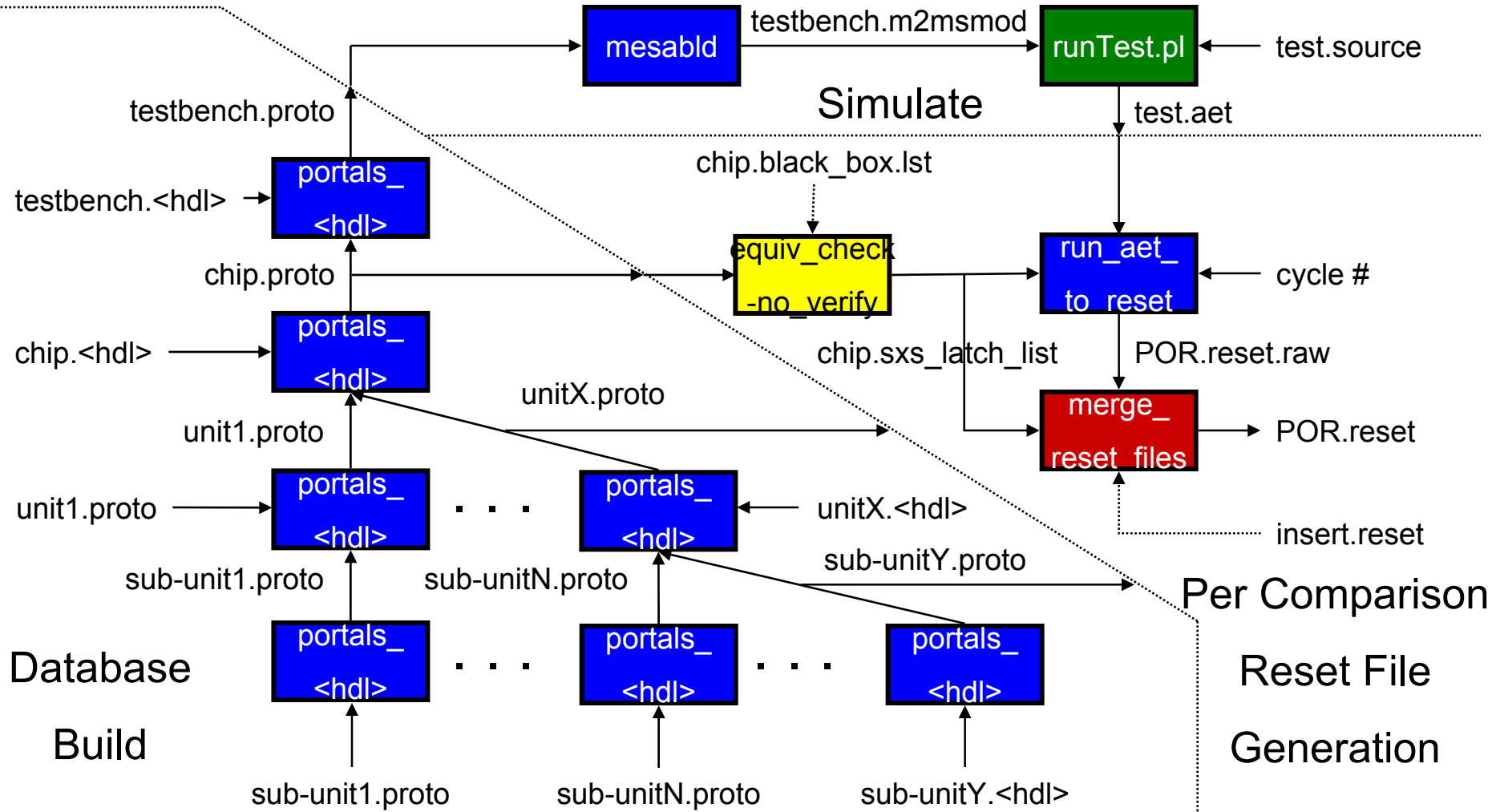
PowerPC 464FP FPU: Lessons Learned

- **Since there is less pressure to keep up, verification can quickly fall behind**
- **Accidentally disabling all clocks is a great way to get everything to match ;^)**
 - Little warning is given for this mistake (other than the comparison job runs very quickly), although a new option is being added to fix this
 - Leads back to the importance of verifying constraints!
- **Although state device correlation is not required, it does help . . .**
- **. . . constantly renaming state devices and/or using the same state device names when changing function can be very disruptive!**
- **Success with sequential equivalence checking is related more closely to the **nature of the design** (ex: arithmetic versus control) and the **changes being made** (ex: register re-timing versus DFT changes) than to **design size** (ex: multi-processor SOC versus FPU)**

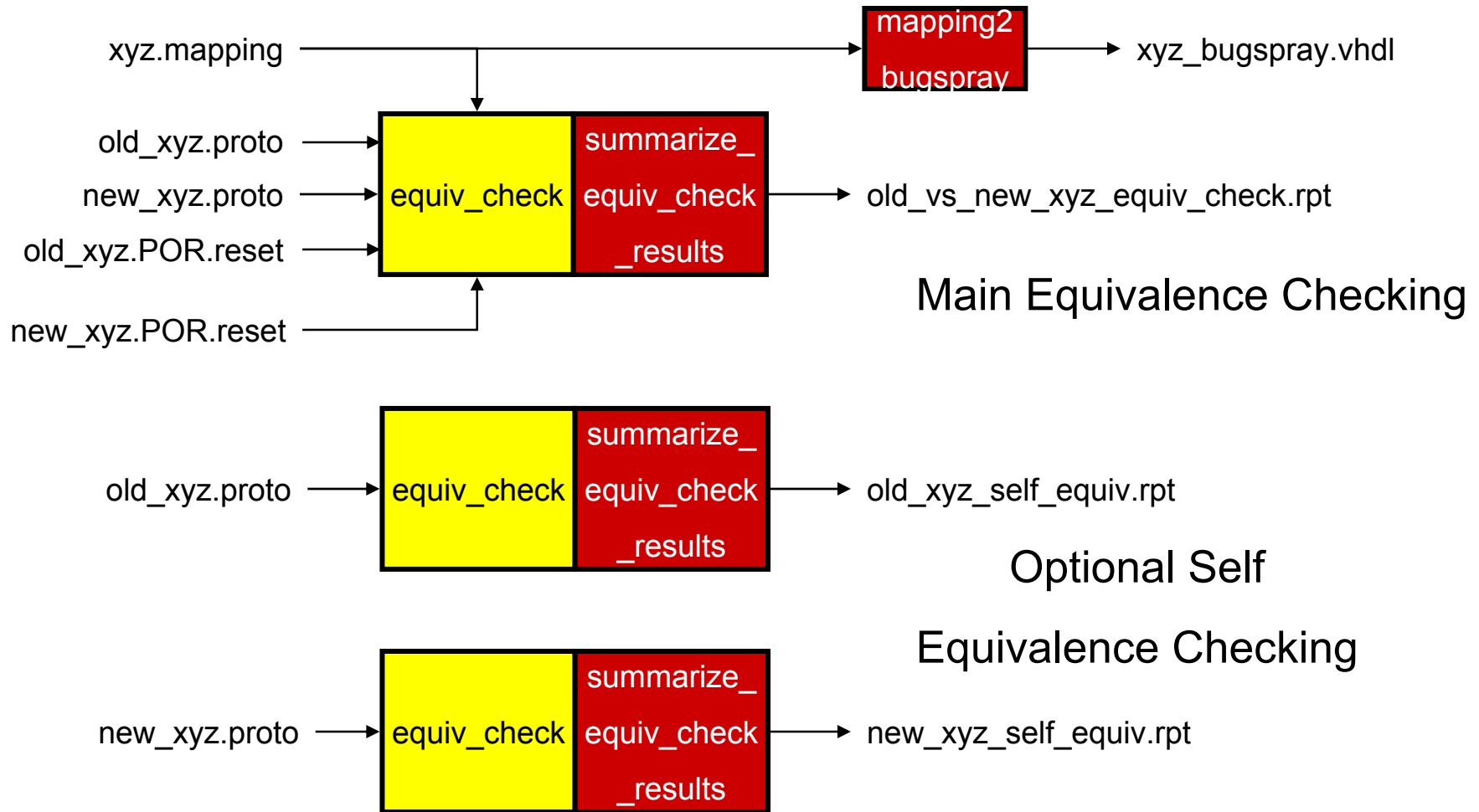
Per Process Diagram



Process Flow: Per Design Targets



Process Flow: Per Comparison Targets



Process Flow (notes)

- **Significant process steps to support main point tool (equiv_check) execution**
- **Generated 'makefile' based flow**
 - Converts a design 'hierarchical definition' (.hd) file into a 'makefile' containing targets which perform analysis tasks on that design
 - Reproducible and reliable (execute only required targets)
 - Proper sequencing and parallel execution of targets
 - Build avoidance (don't re-execute up to date targets)
 - Scalable and easily modified (via '.hd' file changes)
 - User can choose to know as much or as little about target specifics as he/she wants

Process Flow (notes)

▪ Tool catalog

– Standard IBM internal tools

- Support mixing VHDL and Verilog source files during database builds (portals_<hdl>)
- Cycle-Based Simulation build (mesabld)
- Waveform to reset file conversion (run_aet_to_reset)

– Project specific tools

- Running simulation test (runTest.pl)

– SixthSense flow tools

- Makefile (flow control) generator (hd2make – not depicted)
- '.reset' file merging/checking (merge_reset_files)
- 'equiv_check' results checking (summarize_equiv_check_results)
- Constraint to assertion converter (mapping2bugspray)

– SixthSense

- 'No verify' test bench compile (equiv_check –no_verify)
- Test bench compile and verify (equiv_check)

Summary

- **Pursue Formal Methods . . .**
- **. . . as long as you don't expect them to work perfectly right out of the box or without some flow development!**
- **Formal Methods require the thoroughness that verification engineers naturally possess**
- **Analyzing a design from a different perspective is an effective means of uncovering new problems**
- **Embrace Assertion Based Verification**
 - Encourage designers to document their RTL (what a concept!)
 - Enables formal methods
- **No excuse to be bored ;^)**

Tool References

- **SixthSense**
 - (Semi-)Formal Tool and Sequential Equivalence Checker
 - IBM internal tool (www.research.ibm.com/sixthsense)
- **Calypto SLEC**
 - Sequential Equivalence Checker
 - Commercial EDA tool (www.calypto.com)
- **0-In**
 - Semi-Formal Tool
 - Commercial EDA tool (www.mentor.com)
- **Magellan**
 - Semi-Formal Tool
 - Commercial EDA tool (www.synopsys.com)
- **Incisive Formal Verifier**
 - Assertion Based Verification
 - Commercial EDA tool (www.cadence.com)