What's between simulation and formal verification?

David L. Dill
Computer Systems Laboratory
Stanford University
dill@cs.stanford.edu

Why do we care?

- Verification is increasingly a bottleneck
  - Large verification teams
  - Huge costs
  - Increases time-to-market
  - Bugs are being shipped
- Simulation and emulation are not keeping up
- Formal verification is hard

We need alternatives to fill the gap.

Outline

- General observations
- Conventional answers
- Semi-formal methods
- Research issues
- Conclusion

Orientation

- **Focus of this talk: Late stage bugs in register transfer level descriptions (and above).**
- ***Late stage bugs* are hard to find**
  - **few bugs per simulation cycle, person-hour**
  - **delays time-to-market**
- ***Functional errors in RTL* are**
  - **not eliminated by synthesis or**
  - **discovered by equivalence checking.**

Where do bugs come from?

- Incorrect specifications
- Misinterpretation of specifications
- Misunderstandings between designers
- Missed cases
- Protocol non-conformance
- Resource conflicts
- Cycle-level timing errors
- ...

Design scales

- Single FSM: ~12 bits of state, ~30 states
- Individual designer subsystem: ~50K gates, 2K state bits, 10 FSMs
- Major subsystem: ~ 250K gates, 10K state bits, 50 FSMs
- ASIC: ~2M gates

Coverage is the key concept

Maximize the probability of

stimulating and detecting bugs,

at minimum cost

(in time, labor, and computation)

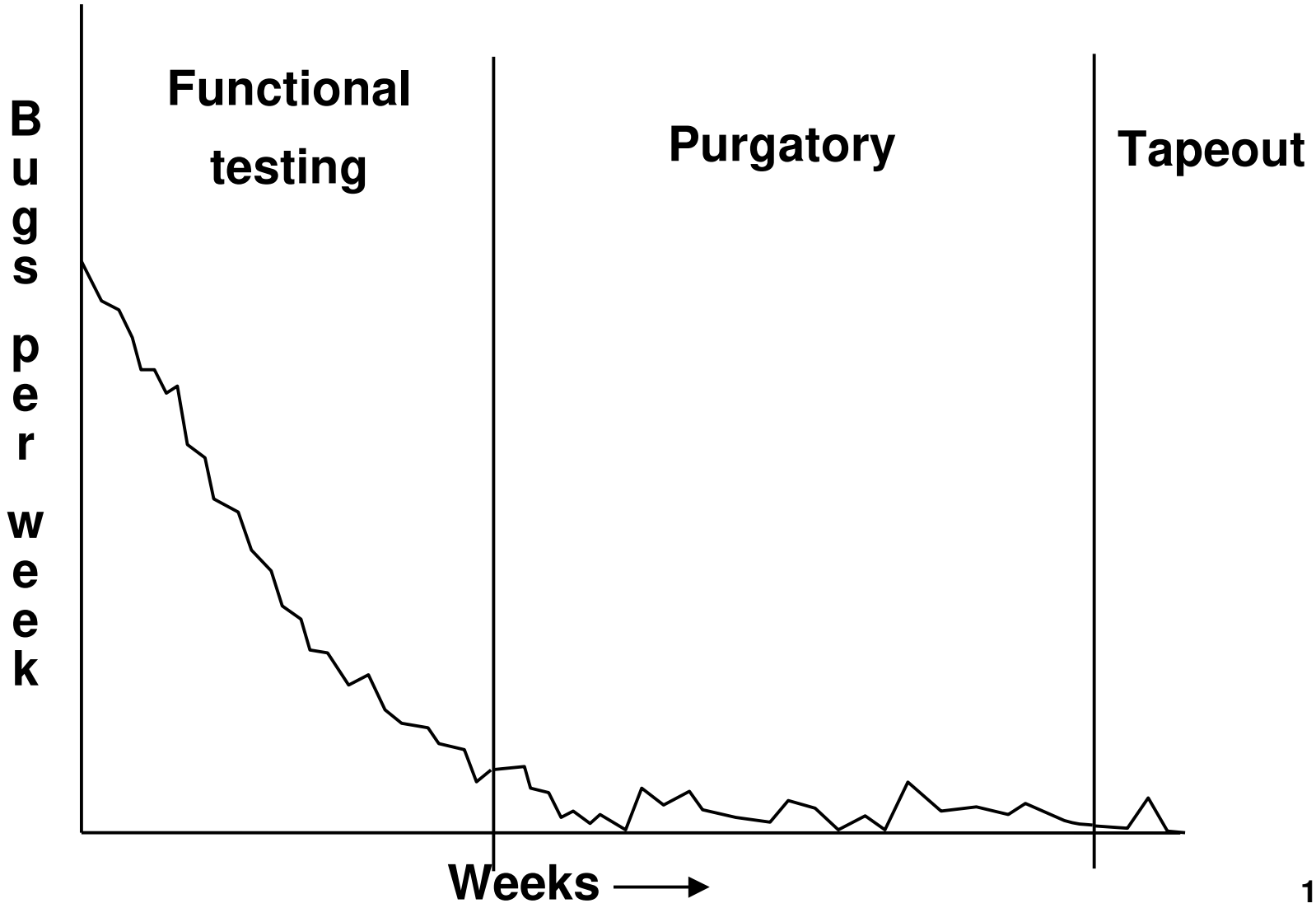
Outline

- General observations
- Conventional answers
- Semi-formal methods
- Research issues
- Conclusion

Simulation

- **Simulation is predominant verification method**
  - **Gate level or register transfer level (RTL)**
- **Test cases**
  - **manually defined, or**
  - **randomly generated**

Typical verification experience



Near-term improvements

- **Faster simulators**
  - **compiled code**
  - **cycle simulation**
  - **emulation**
- **Testbench tools**

  make pseudo-random better/easier

Incremental improvements won't be enough.

Formal verification

- **Ensures consistency with specification for *all* possible inputs (100% coverage).**
- **Methods**
  - **Equivalence checking**
  - **Model checking**
  - **Theorem proving**

Valuable, but not a general solution.

Equivalence checking

- **Compare high level (RTL) with gate level**
- **Gaining acceptance in practice**

  Abstract, Avant!, Cadence, Chrysalis, Synopsys, Verplex, Verysys, …
- **But the hard bugs are usually in both descriptions**

Targets *implementation errors,* not *design errors.*

Model checking

- **Enumerates all states in state machine.**
- **Gaining acceptance, but not yet widely used.**

  Abstract, Chrysalis, IBM, Lucent, Verysys, …

Barrier: Low capacity **(~100 register bits).**

- **Requires *extraction* (of FSM controllers) or *abstraction* (of the design).**
- **Both tend to cause costly *false errors.***

Theorem proving

- Theorem prover checks formal proof
  - Mostly check detailed manual proof.
  - Sometimes provides some automatic help.
- Useful for
  - verifying algorithms (floating point)
  - integrating verification results
    [Aagard, et al. DAC 98]

Not a general solution (too hard!)

Outline

- General observations
- Conventional answers
- Semi-formal methods
  - Coverage measurement
  - Test generation
  - Symbolic simulation
  - Model checking for bugs
- Research issues
- Conclusion

Semi-formal methods

- **Coverage measurement**
  - Code coverage
  - Design fault coverage
  - Tag coverage
  - State machine coverage

Coverage metrics

- **Philosophy: *Quantify coverage of tests.***
- **Helps user optimize tests or testbench.**
- **All metrics should approximate the ultimate: *bug coverage*.**
- **There are many metrics. Issues:**
  - **Ease of implementation, interpretation.**
  - **Predictive ability.**

Code coverage metrics

HDL source based:

- **Line coverage**
- **Statement coverage**
- **Branch and conditional coverage**
- **Path coverage**
- **Toggle coverage**
- **etc. (*More being invented as I speak*)**

Comments on code coverage

- **Valuable tool to guide test writers.**
- **Commercially supported tools for large-scale designs exist.**
  ATC, DAI, Summit, TransEDA, Veritools

Misses lots of bugs.

Design fault coverage

Analogous to manufacturing test:

Many papers, incl.

Abadir, Ferguson, & Kirkland, IEEE TCAD 1988
Kang & Szygenda, ICCD 92

- ***Design fault*** **is a local mutation in gate-level design.**
- **Various models (and names) considered**
  - **Gate omission/insertion/substitution**
  - **Gate input omission/insertion/substitution**

Using design fault models

- Measures coverage of test suite.
- Answers the question: *"How many of these hypothetical errors would my tests have detected?"*
- Methods similar to fault coverage.

  (Easy approach: use stuck-at model and standard fault simulator.)

Comments on design faults

- No empirical justification of fault models
- Relevant only for *gate-level* errors
  - Doesn't find many errors on my bug list
- Additional restrictions even harder to justify
  - e.g. single faults

Feature: faults must be observable at output.

Tag coverage

- **Evaluates error observability at HDL level**
  [Devadas,Ghosh, & Keutzer, ICCAD 96]
  (related papers in this DAC)
- **Error model: positive or negative mutation in assigned value: $+\Delta$ or $-\Delta$.**
- **Mutated values are symbolically propagated during simulation (similar to fault simulation)**

Model doesn't address most of my error classes.

Finite state machine (FSM) coverage

- Various concepts of coverage
  - State coverage
  - Transition coverage
    - old state/new state
    - old state/new state/output
- State machines can be controllers or abstract models.

FSM coverage measurement

- Get some FSMs
  - Extract control FSMs automatically from design
    - Cut out data path
  - or, have user make up an abstract FSM that models behavior.
- Record and report coverage from user tests
- Several tools available

  Basically, all the code coverage companies

Multiple FSM coverage

- **May be the *most relevant* metric**
  - **Many errors are problems in FSM *interaction***
    - **synchronization errrors**
    - **resource conflicts**
    - **etc.**
- **Difficulties**
  - **Results may be difficult to interpret.**
  - ***Proportional* coverage is difficult**

    Reachable states/transitions may depend on whole design.

Semi-formal methods

- Coverage measurement
- Test generation
- Symbolic simulation
- Model checking for bugs

Verification test generation

Philosophy: ***Generate tests automatically that maximize coverage per simulation cycle.***

- **Test *generation* is crucial for high productivity.**
- **Specific topics**
  - **ATPG methods**
  - **FSM-based methods**

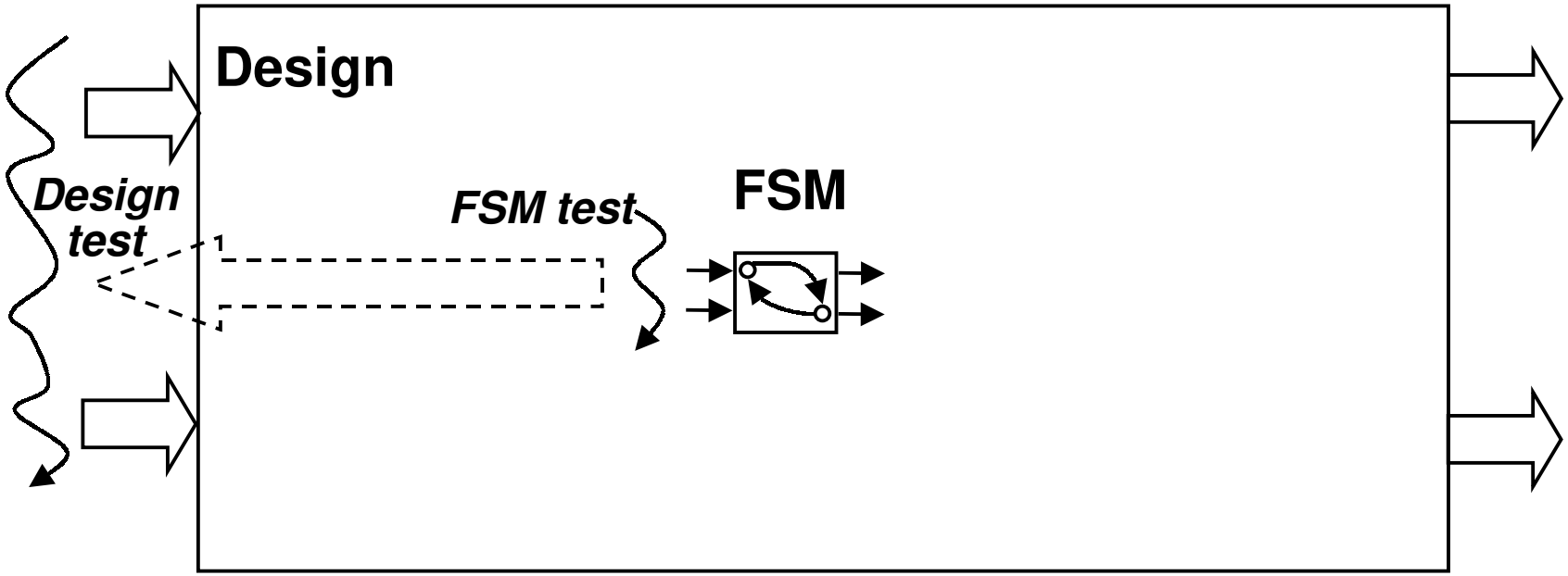
ATPG methods

- Use stuck-at fault models or design faults.
- Generate tests automatically using ATPG techniques
- *Takes into account "observability" of error.*
- Oriented towards combinational designs
  - General solution would need sequential ATPG [*hard*]

Relevance of metric is still a problem

FSM-based test generation

- Generate FSM tests using model checking techniques (e.g. BDD, explicit).
- Map FSM test to design test vector [*hard!*]



Test vector mapping

- **User defines mapping rules from FSM step to test subvectors.**

  [Ho PhD, Stanford 1996, Geist, *et al.,* FMCAD 96]

  Mapping must be relatively simple.

- **Map to test vectors using sequential ATPG techniques.**

  [Moundanos, *et al.,* IEEE TOC Jan. 1998]

- **Generate vectors symbolically, prune using multiple FSM coverage [Aziz *et al*. DAC 98]**

Published examples are small.

On-the-fly test generation

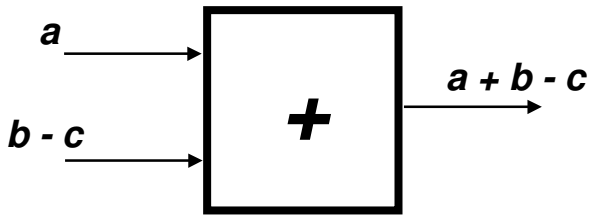
- *On-the-fly* generation is more efficient
  - Disk bandwidth is a bottleneck
  - Tests can be huge
- 0-In Search: multiple FSM coverage + on-the-fly generation.

Semi-formal methods

- Coverage measurement
- Test generation
- Symbolic simulation
- Model checking for bugs

Symbolic simulation

- **Philosophy: *Get a lot of coverage from a few simulations.***
- **Inputs are variables or expressions**
- **Operation may compute an expression instead of a value.**



- **Advantage: more coverage per simulation**

  one expression can cover a *huge* set of values.

Symbolic Trajectory Evaluation

- Symbolic values are BDDs.
- Special logic for specifying input/output tests.
- Used at MOS transistor or gate level.
- Systems: COSMOS [Bryant, DAC 90], Voss [Seger]

Higher-level symbolic simulation

- Symbolic simulation doesn't have to be bit-level.
- RTL simulation can have built-in datatypes for:
  - Bitvectors
  - Integers (linear inequalities)
  - Arrays

Especially useful if combined with automatic decision procedures for these constructs.

[Barrett *et al.* FMCAD 96, DAC 98]

Symbolic simulation as in-between method

- *Partial* symbolic simulation as semiformal method has not been studied systematically.
- Many ideas have not been explored
  - Heuristics for constant/symbolic inputs.
    - Symbolic data path.
  - On-the-fly conversion from symbolic to constant.
  - Etc.
- Related to tag coverage (propagation of tag = propagation of symbolic value)

Semi-formal methods

- Coverage measurement
- Test generation
- Symbolic simulation
- **Model checking for bugs**

Model checking for bugs

Philosophy: ***Optimize model checking to find bugs, not to check correctness.***

Forward model checking

- **BDD model checking is usually breadth-first**

  Can't find bugs if BDD blows up

  [Iwashita, *et al.* ICCAD 96, 97]

- **FMC mixes depth-first/breadth-first.**
  - **Limits memory use**
  - **May find bug before it blows up**
- **Uses special temporal logic that supports forward traversal.**

Prioritized model checking

Idea: use best-first search for bugs

- **Metric for "best" is unclear.**
- **May be used with BDDs, or explicit MC**
- **Metrics:**
  - **Hamming distance**
    [Yang, Dill HLDVT 96, Yuan *et al.* CAV 97]
  - **"Tracks" [Yang & Dill, DAC 98]**
- **Also expands error sets**

  Makes finding error more likely.

Comments on model-checking for bugs

- Topic is not mature.
- Published examples are very small.
- Big increases in capacity needed.

Many more ideas to be explored.

Outline

- General observations
- Conventional answers
- Semi-formal methods
- Research issues
- Conclusion

Requirements for verification

- ***Simulation takes short cuts, but formal and semi-formal require more information.***
- **Specification - provides a way to identify bugs. Could be**
  - **Logical spec**
  - **Operational spec: HDL, FSM, …**
  - **Assertions**
- **Environment model: What are the legal inputs to design?**

Error observability

- A bug may be stimulated, but still missed.
  - Wrong value is generated and stored in memory
    - Read 1M cycles later, but simulation doesn't run that long.
  - Wrong value is generated speculatively, but never used.
- False sense of security may result from good coverage but bad observability.

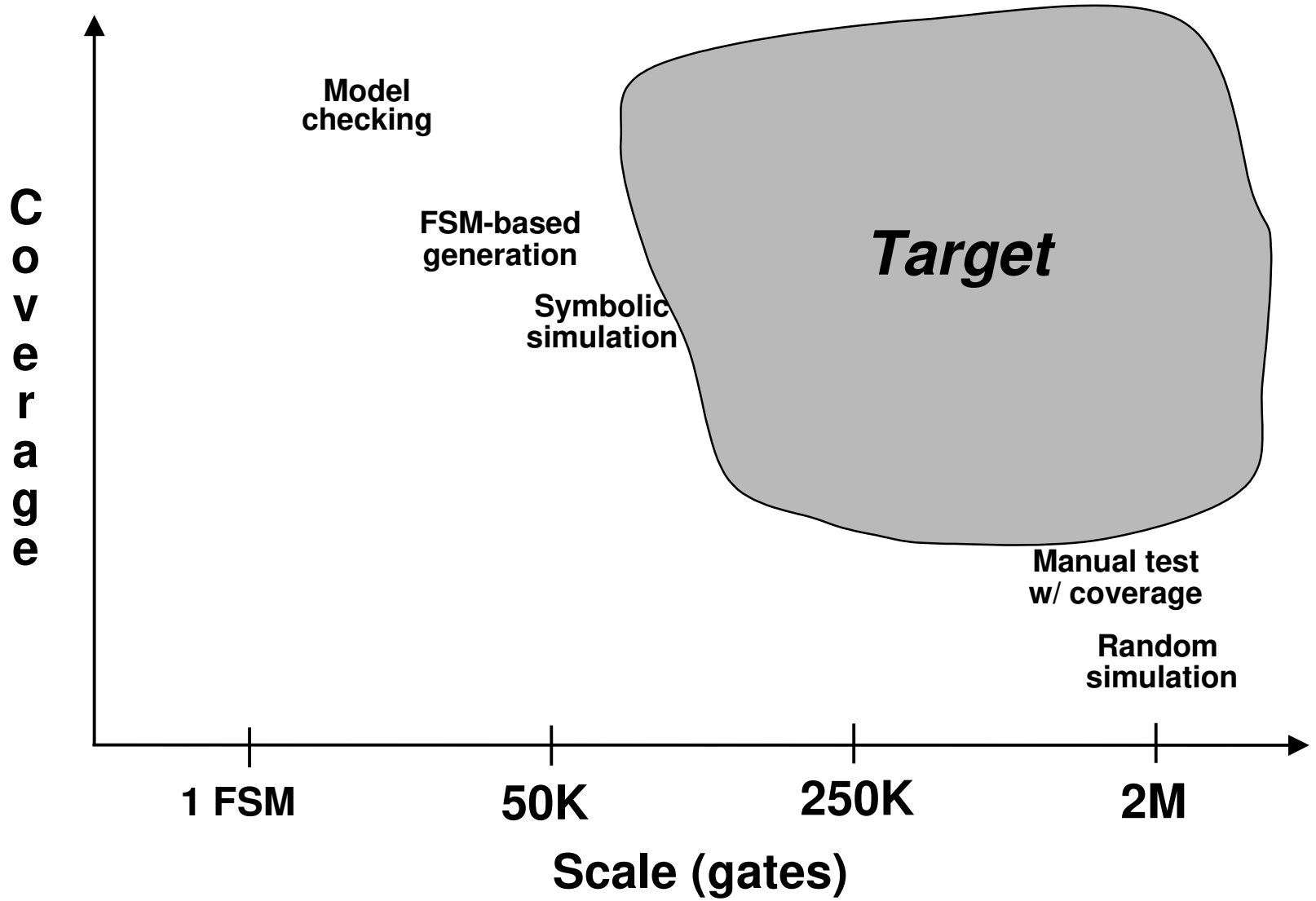
Solutions to error observability

- Explicitly check observability as part of coverage metric
  - ATPG, design faults
  - Tag coverage
- Use *internal* assertions (whitebox verification)
  - Assertions detect errors at the source
  - Assertion synthesis (0-In)

Research methodology

- Solid research results are going to be very difficult to document in this area.
- Apparent effectiveness depends on
  - Design methodology (language, processes)
  - Type of design
  - Designer style, training, and psychology
  - Size of design!
- Design examples need to be *large, realistic,* and *varied.*

Coverage vs. scale



Outline

- General observations
- Conventional answers
- Semi-formal methods
- Research issues
- Conclusion

Prediction

- This is going to be a hot area
  - Many papers
  - Verification products
- Simulation & emulation will continue to be heavily used.
- Formal verification will be crucial, *when applicable*
  - Special application domains: protocols, FSMs, floating point, etc.
  - *Design for verification* would increase scope

Prediction

Semi-formal verification will be the primary workhorse for eliminating late-stage bugs in RTL