Functional Verification

Prof. Wang Jiang Chau

Edgar Romero

Carlos Castro

Joel Quispe

Grupo de Projeto de Sistemas Eletrônicos e Software Aplicado (G-Seis)

Laboratório de Microeletrônica – LME

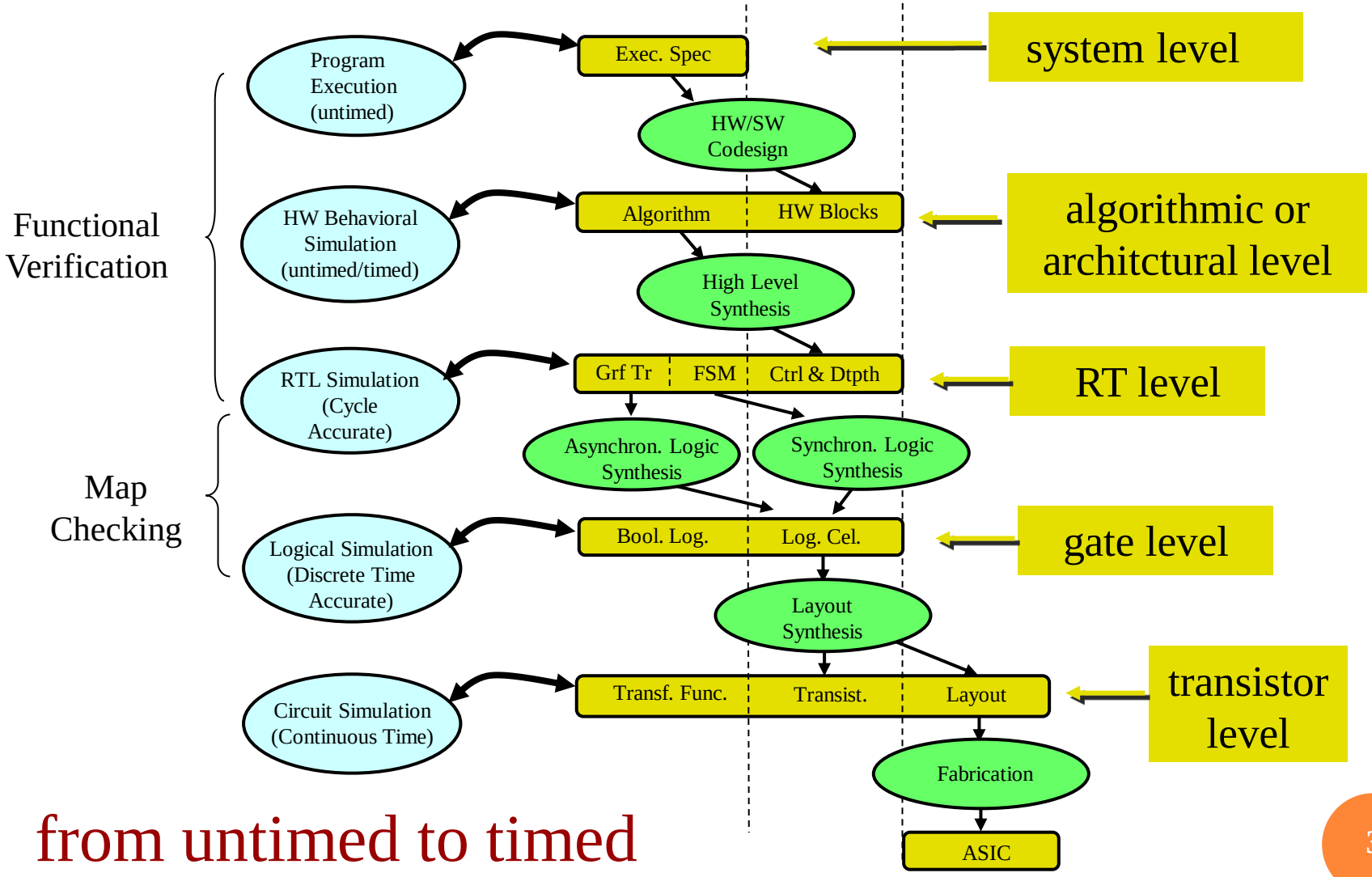
Depto. Sistemas Eletrônicos

Universidade de São Paulo

OUTLINE

- Fuctional Verification
  - What is it?
  - Why do we need it?
- Formal X simulation-based verification
- Aspects of simulation-based verification
  - Testbenches
  - (Random) stimuli generation
  - Coverage
- Coverage-driven verification
  - Static X dynamic approaches
  - Examples

THE SYNTHESIS/VERIFICATION FLOW



from untimed to timed

THE DESIGN'S CHALLENGES

Presently, the main challenge in the activity of designing multi-million-transistors ICs IS NOT to SYNTHESIZE them, but IT IS to BE CONFIDENT that the synthesized product is correct !!!

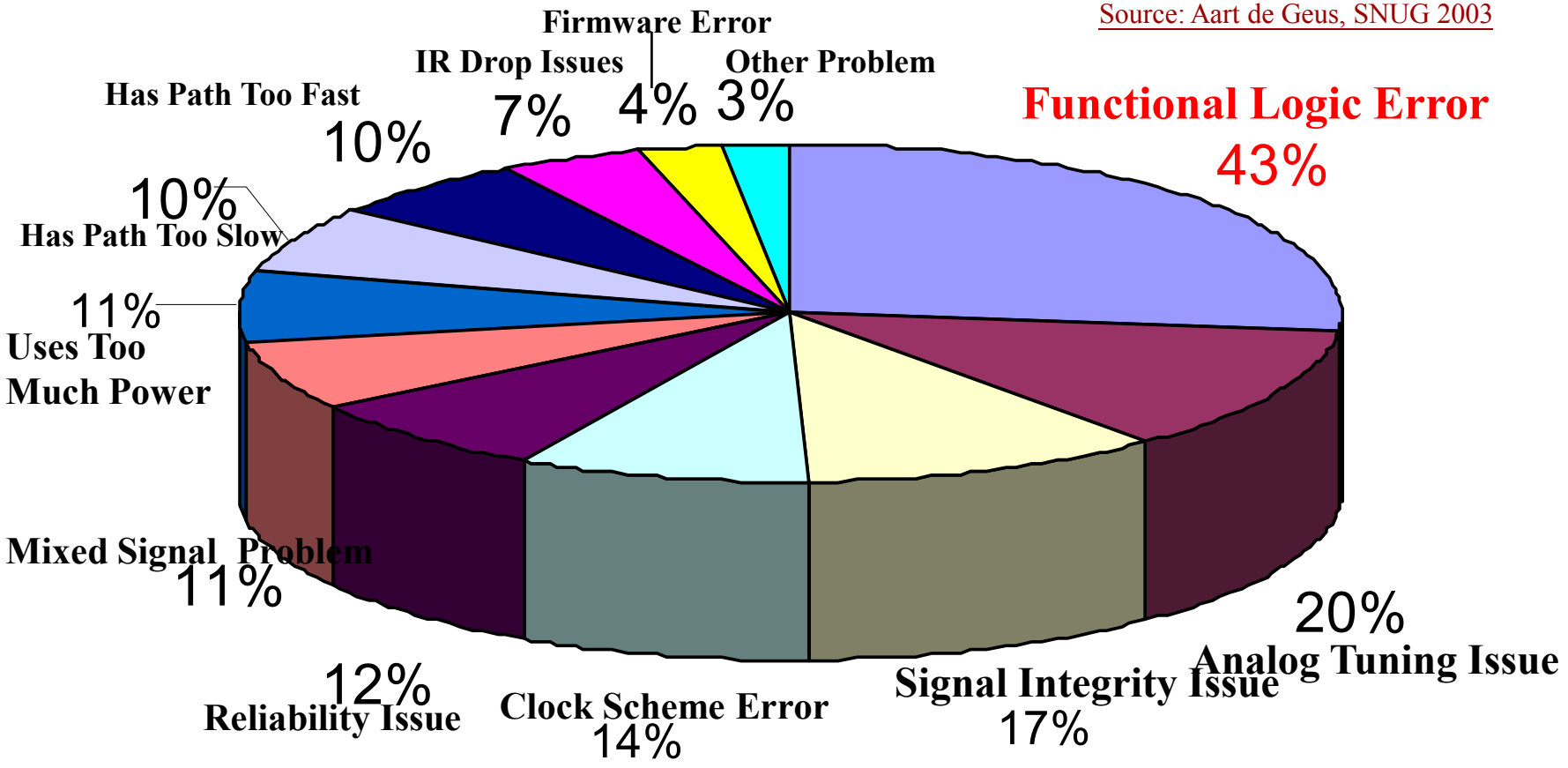
FUNCTIONAL VERIFICATION – DESIGNERS' VIEW

- Why does it matter?
  - Design is still an activity dependent on manual intervention
  - Too many (interpreted) specification errors!
  - Too many implementation errors!
  - It is common that complex chips go through multiple tape-outs before release- overall 61% of new ICs/ASICs require at least one re-spin
  - High cost of faulty designs (loss of life, product recall)

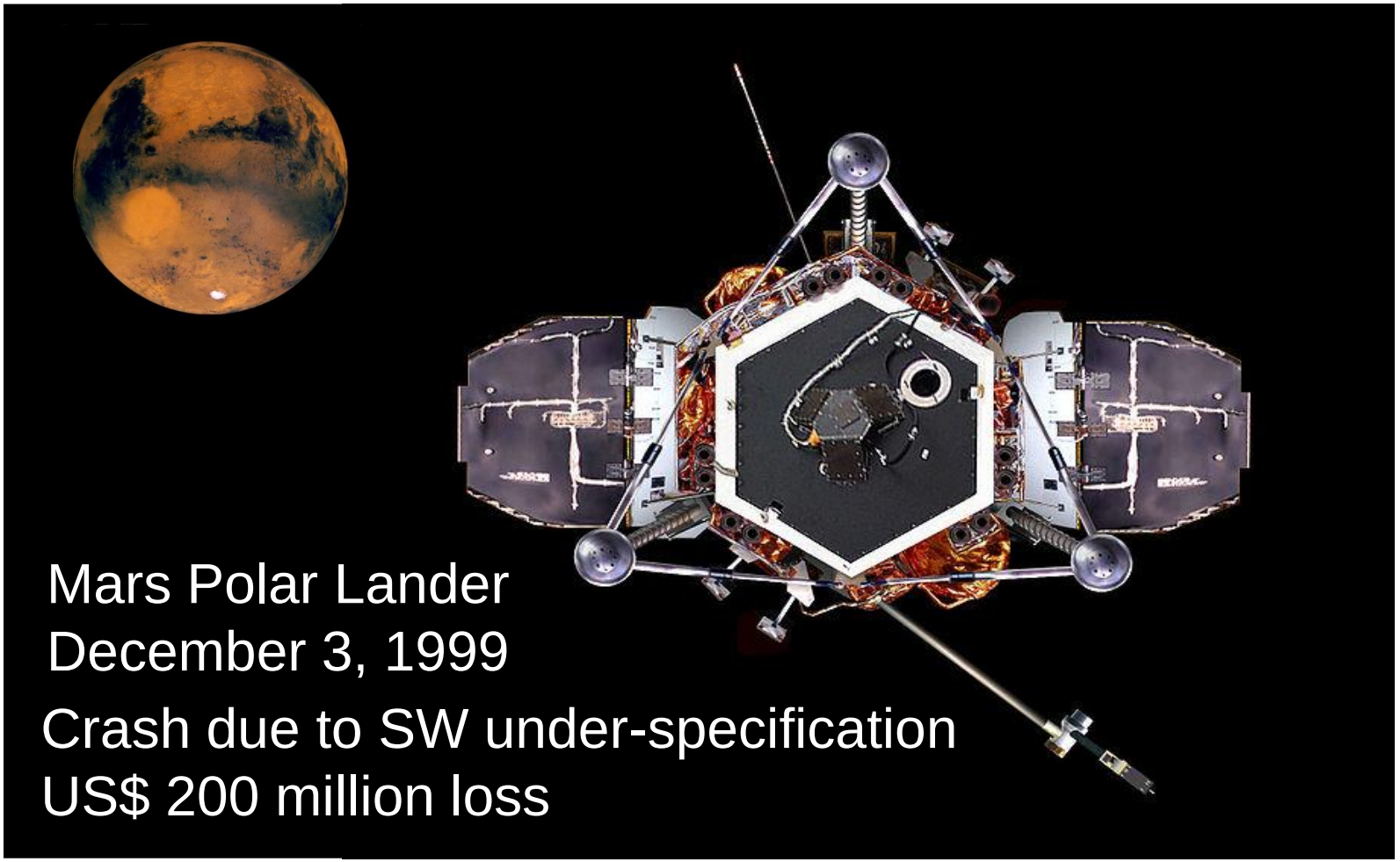
FUNCTIONAL VERIFICATION DESIGNERS' VIEW

Problems found on first spin of ICs/ASICs

Source: Aart de Geus, SNUG 2003



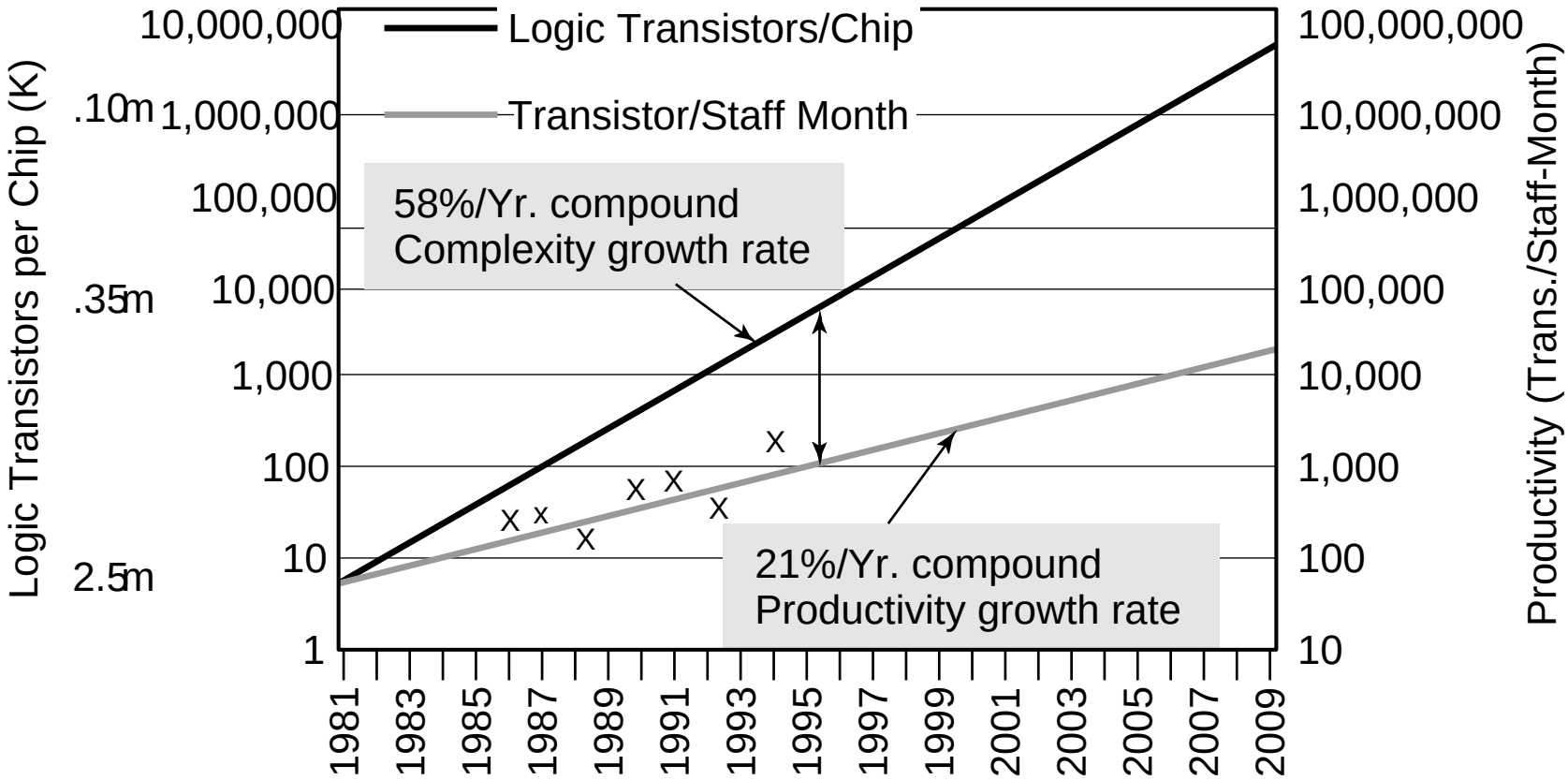
FUNCTIONAL VERIFICATION DESIGNERS' VIEW



Functional Verification Developers' View

- Why does it matter?
  - Verification is a bottleneck in the design process.
  - ~ 70% of project development cycle: design verification (time-to-market)
  - ~ 70% of project development resources: design verification (sometimes number of verification engineers = 2 * number of synthesis engineers)
  - At the end, doubts on robustness remains
  - Difficulties in Functional Verification add to the design productivity gap

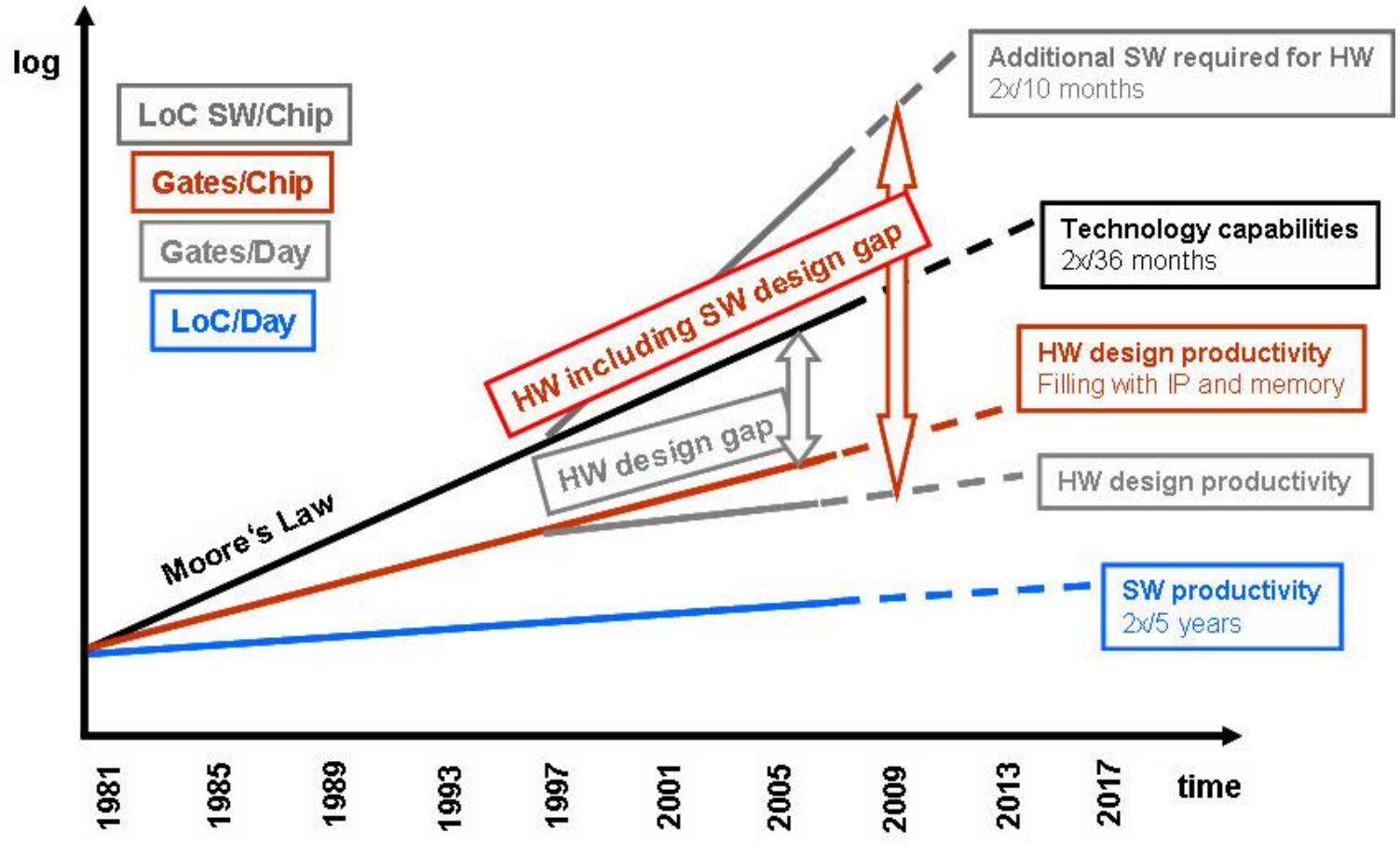
THE INCREASING DESIGN PRODUCTIVITY GAP



A growing gap between design complexity and design productivity

Source: ITRS 99

THE INCREASING DESIGN PRODUCTIVITY GAP



Source: ITRS 07

FUNCTIONAL VERIFICATION – BROAD INTERPRETATION

- Andrew Piziali, 2004
  - Functional Verification is demonstrating the design intent is preserved in its implementation
- Janick Bergeron, 2003
  - The main purpose of Functional Verification is to ensure that a design implements intended functionality
- And many others....

There is a difference between system specification and interpreted specification (intended design)

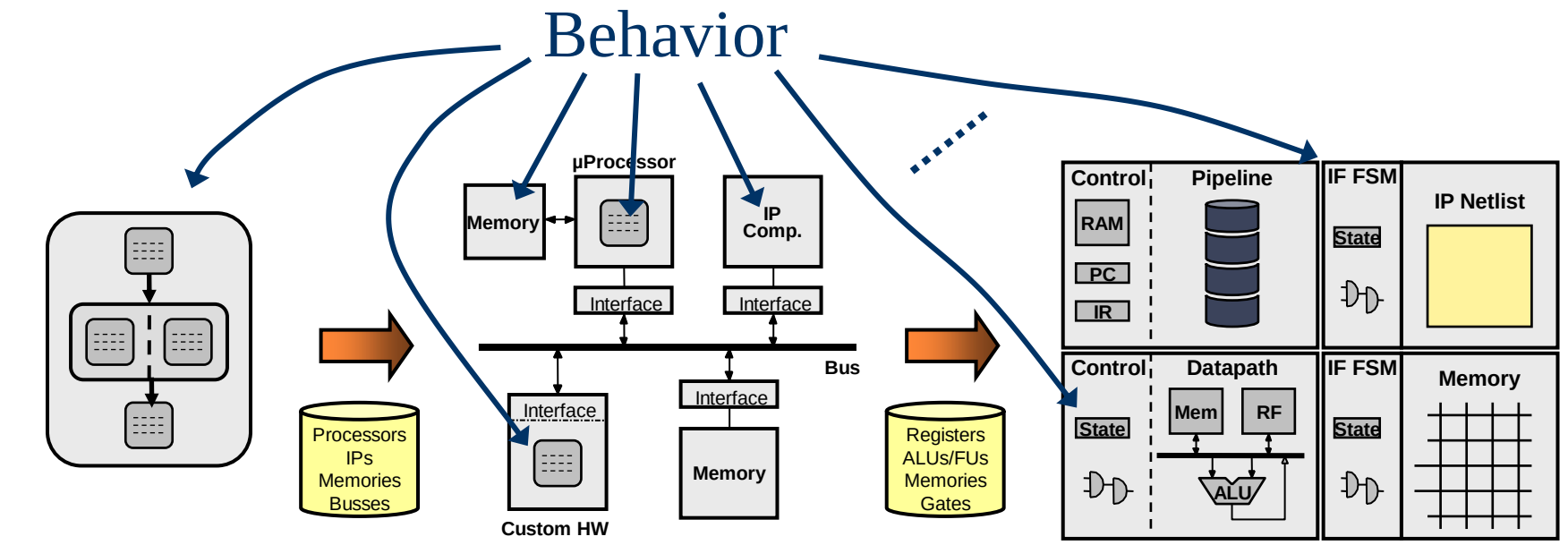
FUNCTIONAL VERIFICATION NARROW INTERPRETATION

- Ira Chayut, Verification Architect with NVIDIA, 2009
  - Functional verification is the task of checking that a design implements a specified architecture
- ACE Verification (www.aceverification.com)
  - Functional Verification is defined as the process of verifying that an register-transfer level , RTL, (Synthesizable Verilog, VHDL, SystemVerilog) design meets its specification from a functional perspectiv


DUV
assertion
Model Checking
Property

The Design Flow

- Executable Specification to architecture and down to implementation
- Behavior (functional) to structure



Specification + constraints

System architecture + estimates

RTL/IS Implementation + results

What is the Artifact to be Verified After All?

- In the broad interpretation for FV
  - All system and syb-system models are to be verified against specification
  - The model may be behavioral/structural at any abstraction level
- In the narrow interpretation
  - The design-under-verification (DUV) model is the register transfer level (RTL) one

Typical Algorithmic Code (untimed)

```
ARCHITECTURE behavior OF example_1 IS
BEGIN
        PROCESS (i1,i2,a,b)
        BEGIN
                o1 <= i1 or i2;
                IF i3 = '0' THEN
                        o1 <= '1;
                        o2 <= a + b;
                                ELSE
                        o1 <= '0';
                        END IF;
        END PROCESS;
END behavior ;
```

clock cycle boundary can be moved to design different register transfers

Typical RTL Code (clock accurate)

```
ARCHITECTURE behavior OF count2 IS
BEGIN
    count_up: PROCESS (clock)
       VARIABLE count_value: NATURAL := 0;
    BEGIN
       IF (clock'event AND clock='1') THEN
          count_value := (count_value+1) MOD 4;
          q0 <= bit'val(count_value MOD 2) AFTER prop_delay;
          q1 <= bit'val(count_value/2) AFTER prop_delay;
       END IF;
    END PROCESS count_up;
END behavior;
```

DUV OBSERVATION: Properties

- Property is an attribute of behavior that the implement-ation should always honor
- More formal: "property is a collection of *logical* and *temporal* relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behavior"
- In other words, properties make statements about functional aspects of a DUV.

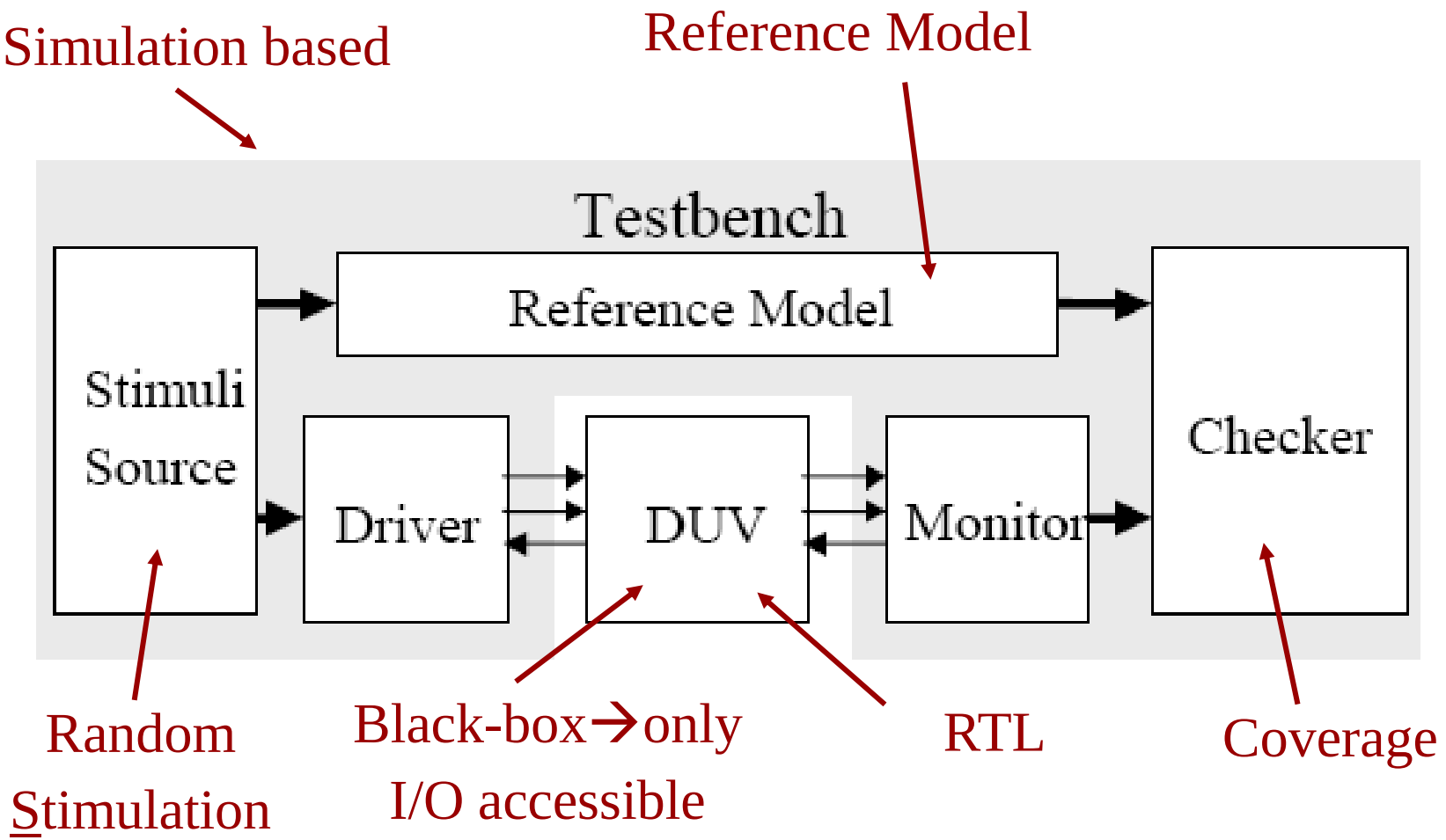
PROPERTIES

- Safety (invariant) property: it is the one that should evaluate to true for all specified sample of times (or that something wrong should *never* happen).
  - Ex. “No more than one bit per cycle should change on a gray-code control bus.”
- Liveness property: it specifies an eventuality that is unbounded in time (or that something right should happen *eventually*) .
  - Ex. “ if *req* is asserted, then *grant* is asserted sometime later”

DUV OBSERVATION: Events

- Event is a *user-specified* property that is satisfied at a specific point in time during the process of verification; sometimes it is associated to some change of the state in a system.
- An event may be Boolean (when a Boolean expression evaluates to True at some time) or Sequential (when a sequence of Boolean events occur).
  - Ex. “ read_a is active”
  - Ex. “sequence access_allowed is immediately followed by read_a is satisfied”

Classifying Functional Verification

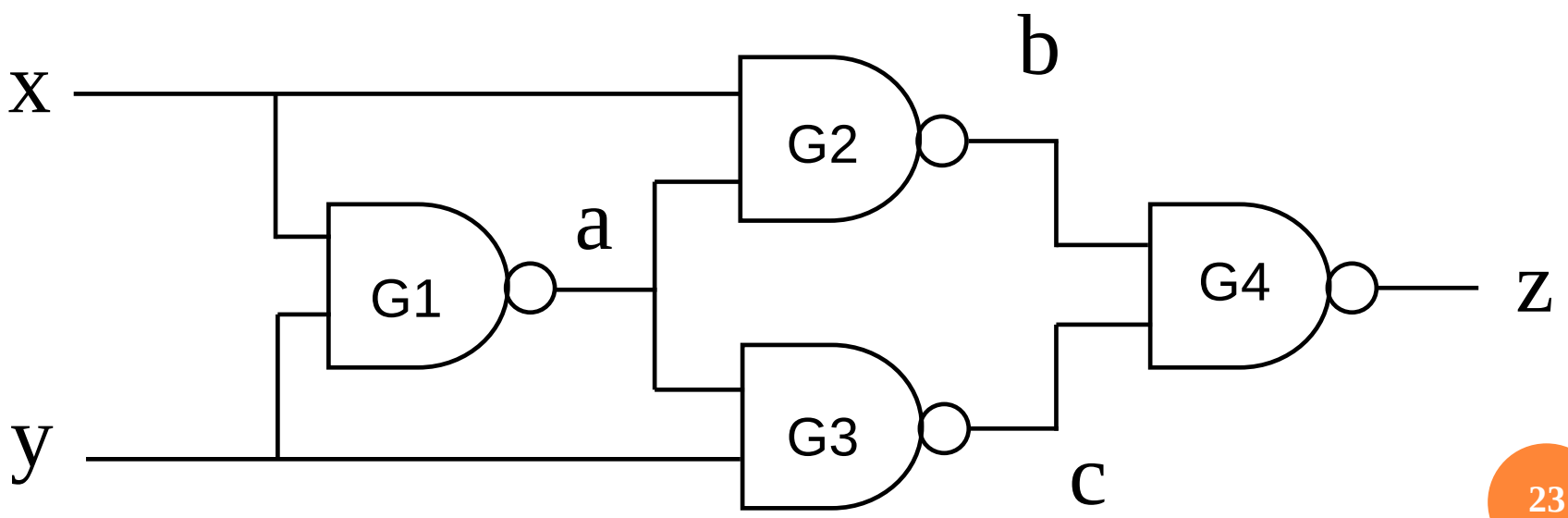


Functional Verification by Robustness

- Simulation – confidence depends on coverage
  - + The most widely used
  - + The only current solution for system verification
  - – Needs testbench and stimuli
  - – Hard to understand coverage/completeness
- Formal – high confidence
  - + Exhaustive
  - – Limited capacity/ limited applicability
- "Real life"- confidence depends on coverage
  - + FPGA, ICE, HW emulators and prototypes
  - + Realistic, little TB development effort, fast
  - – Low observability/debug, FPGA not for large systems

FORMAL vs SIMULATION BASED VERIFICATION

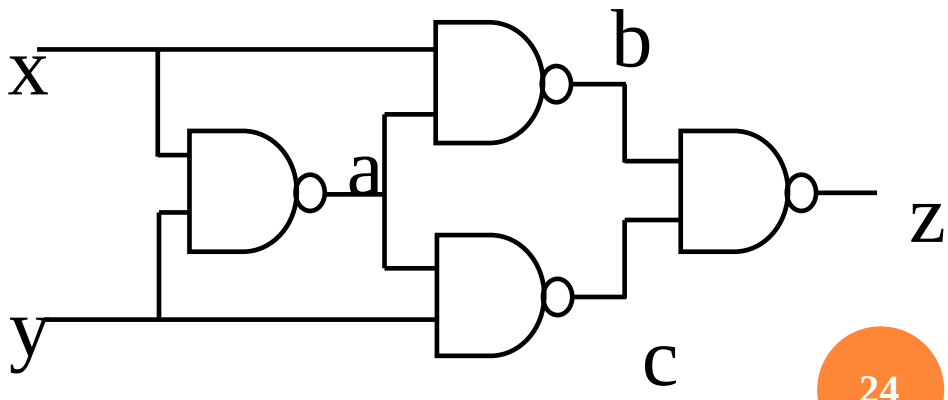
- Example:
  - Verification of Exclusive-OR circuits. Consider:
    - The scheme below is the implementation
    - z = ~x&y + x&~y ('~' means complement) is the spec



FORMAL vs SIMULATION BASED VERIFICATION

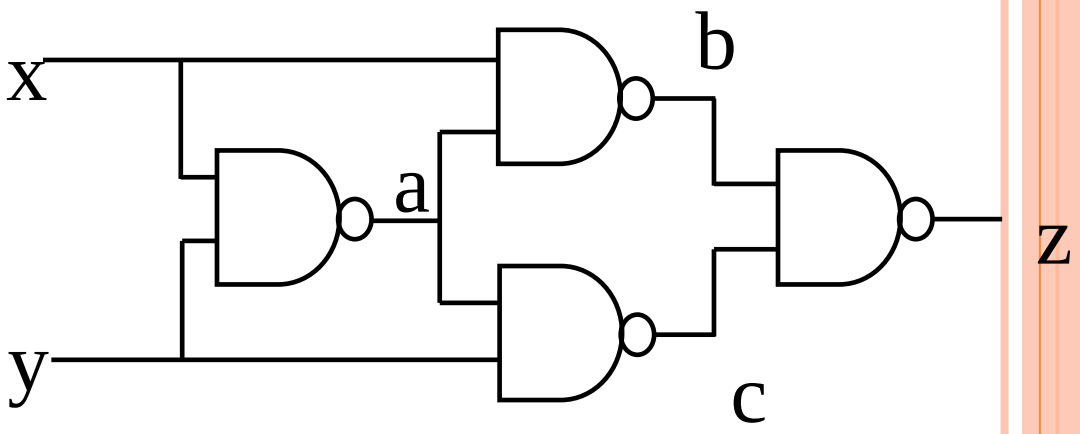
- Verification of Exclusive-OR circuit with "simulation"
  - Simulate the circuit to evaluate z
  - Compare z with specification ~x&y+x&~y in all four cases
  - Need to simulate 2**N cases where N is #ins

x	y	Simulated z
0	0	0
0	1	1
1	0	1
1	1	0



FORMAL vs SIMULATION BASED VERIFICATION

- Verification of Exclusive-OR circuit with "formal verification"



z = ~b + ~c (NAND)

b = ~x + ~a (NAND)

c = ~a + ~y (NAND)

a = ~x + ~y (NAND)

z = ~b + ~c

= ~(~x + ~a) + ~(~a + ~y)

= a&x + a&y (composition rule + Boolean algebra)

= (~x + ~y)&x + (~x + ~y)&y

= x&~y + ~x&y (composition rule + Boolean algebra)

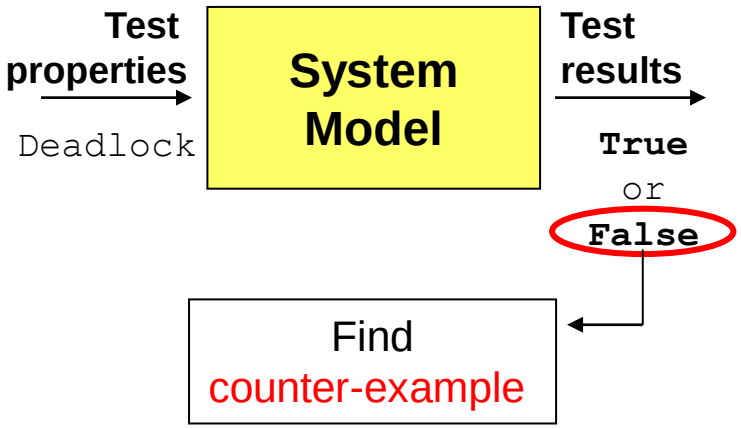
- Axiomatic and mathematical transformation of expressions to reach the specification
- This is a mathematical proof !

FORMAL VERIFICATION (STATIC)

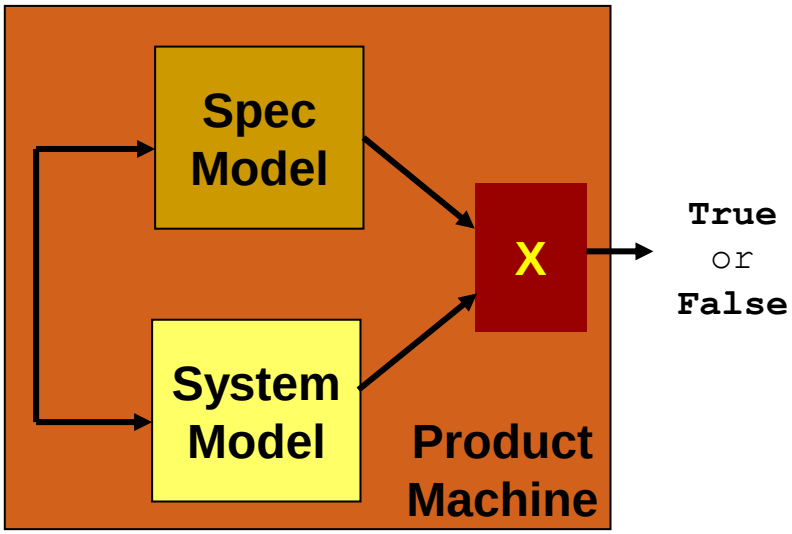
- "Prove" the correctness of designs
  - Both implementation and specification are represented with formally defined models (mathematical reasoning)
  - Equivalent to simulating "all cases" in simulation

- Possible mathematical models
  - Boolean function (Propositional logic)
    - How to represent and manipulate on computers
  - First-order logic
    - Need to represent "high level" designs
  - Higher-order logic
    - Theorem proving = Interactive method

CLASSES OF FORMAL VERIFICATION



Model (property) checking



Equivalence checking

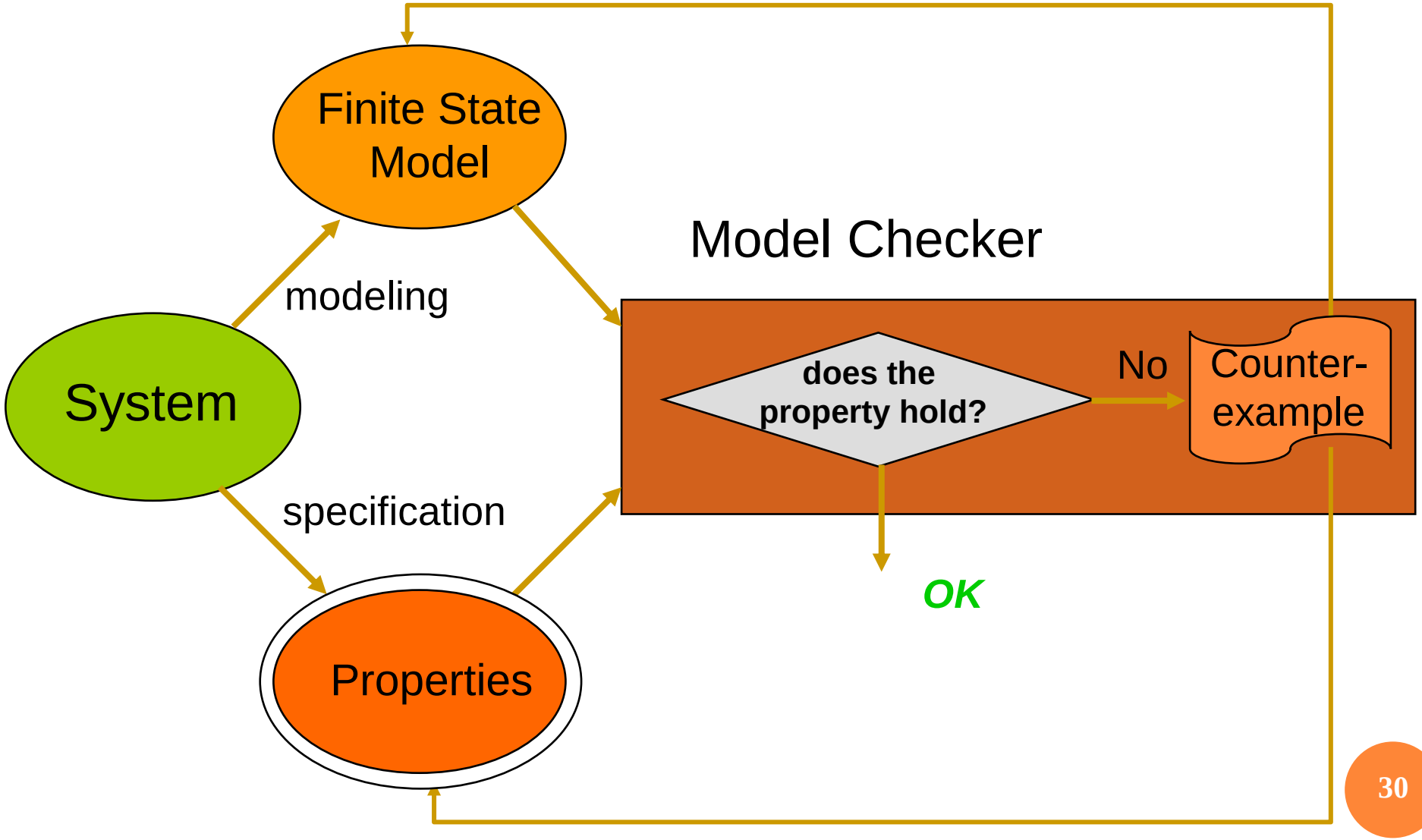
MODEL CHECKING

- Model Checking is an automatic technique for verifying finite-state reactive systems or communication protocols
  - a reactive finite machine is an automaton whose inputs come from the environment
- the technique is aimed to check that a property holds in a finite state model of a system
- the technique was pioneered by Edmund Clarke, at the CS Dept of CMU, in 1981

ASPECTS ON MODEL CHECKING

- The technique relies on exhaustive state space search in the automatic machine
  - (OBDD), SAT machines
- The major challenge is to fight state-explosion problem (although large state spaces, as 10^120, can be handled)
- Can uncover subtle design errors
- Has been successfully used to find bugs in published standard
- The verification completeness refers to the stated properties; the real completeness depends on the “quality of the properties.

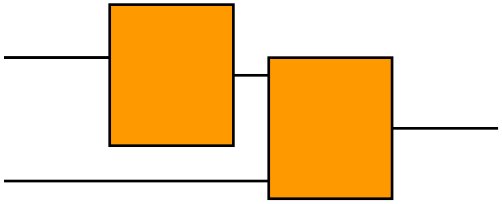
HOW DOES IT WORK? (MORE DETAILS)

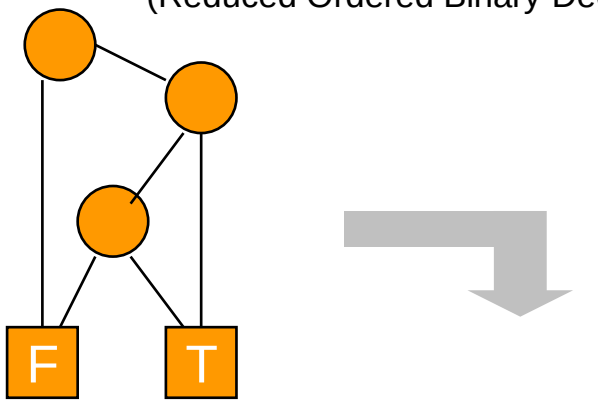


EQUIVALENCE CHECKING

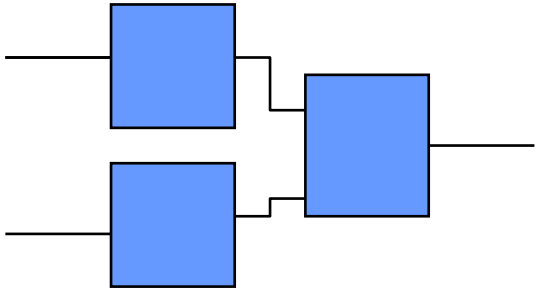
E.g. ROBDD
(Reduced Ordered Binary Decision Diagram)

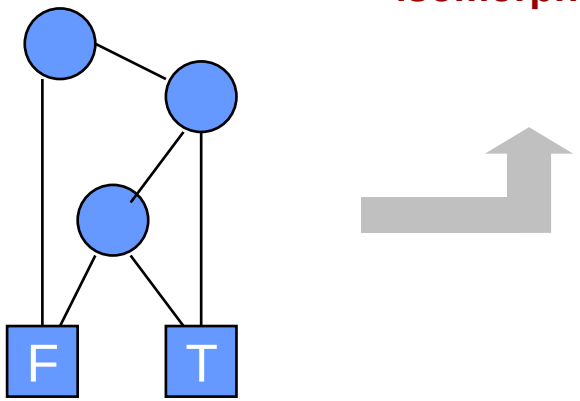
Implementation A





Implementation B

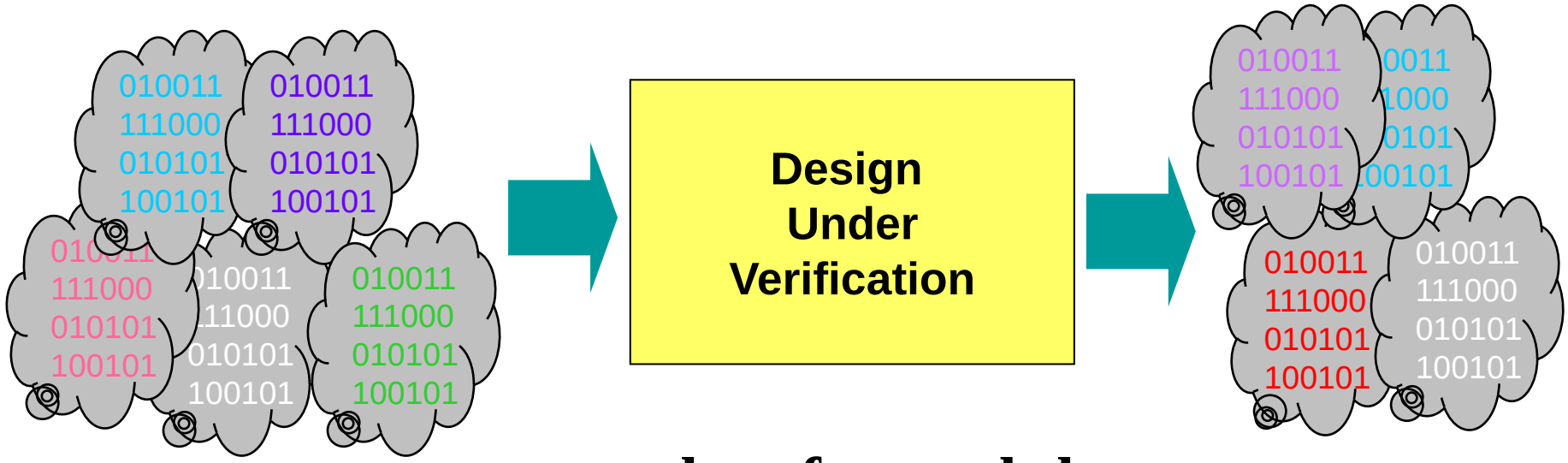




isomorphic ?

Given two designs, prove that for all possible input stimuli their corresponding outputs are equivalent

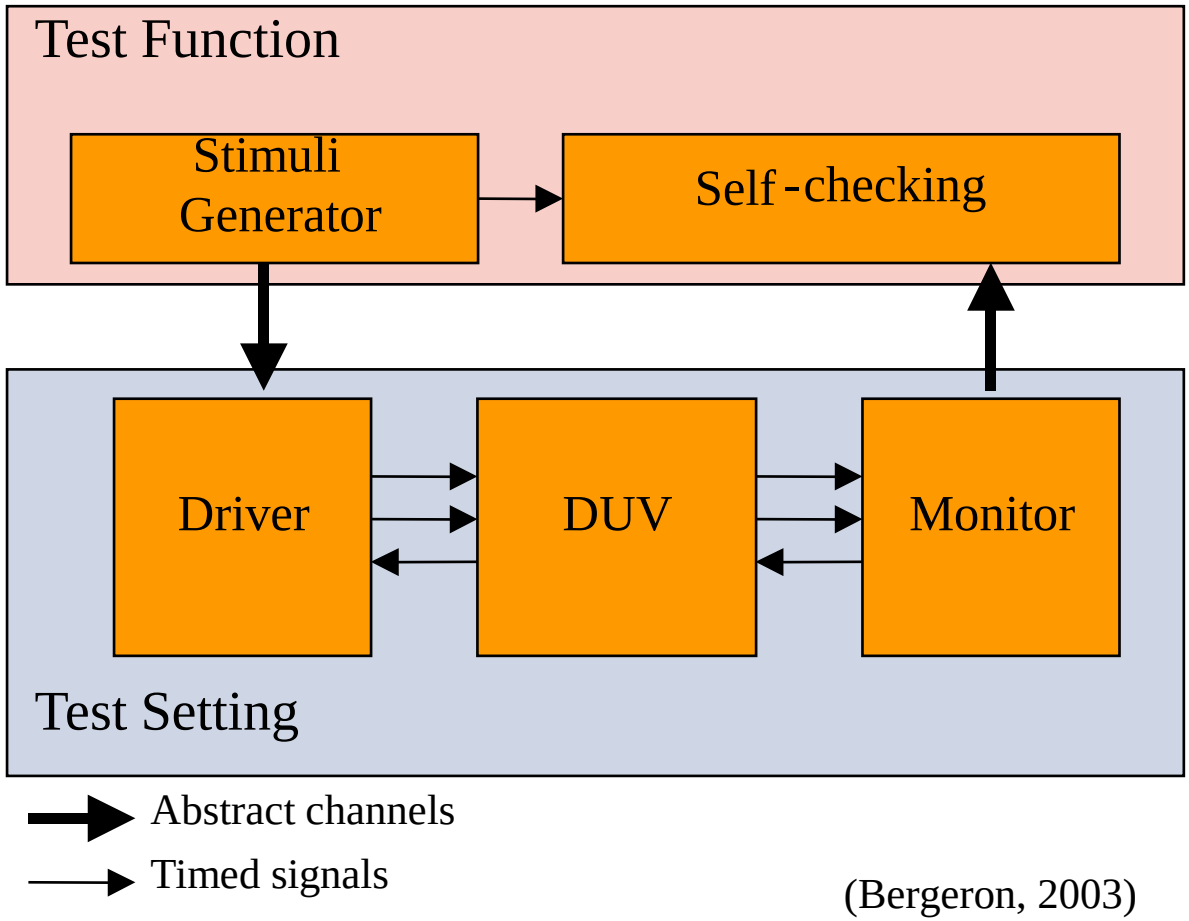
SIMULATION-BASED VERIFICATION (DYNAMIC)



Not possible to cover all cases,

therefore, subtle bugs may possibly survive (corner cases)

GENERIC TESTBENCH



(Bergeron, 2003)

SIMULATION-BASED VERIFICATION CLASSIFICATION BY INPUT SEQUENCE

- Test Program
  - sequence of instructions (typically assembly language)
  - verification of programmable processors
  - “good” sequences to lead to high/fast coverage
- Transaction Testcases
  - sequence of stimuli which traverse multiple cycles and states, performing individual tasks (read, write, etc.)
  - applicable to cores in general

Simulation-based verification Classification by Stimuli Origin (Type)

- Specification cases (direct testing)
  - traditional, simple test input
  - typical cases, recommended in specification manuals
  - not comprehensive at all
- Corner cases (direct testing)
  - unusual, “strange” cases
- Random pattern generation
  - within intelligent testbench automation
- Real cases
  - real network traffic from routers, video frames (films), etc.

Deterministic Direct Testing

- Very (most) common test methodology used today
- Stimuli are manually generated and normally correspond directly to the functional test plan.
- Verification engineer can determine/define all the testcases considered relevant.
- Limitations to be used comprehensively:
  - deterministic tests is a time-consuming, manual programming effort (if completeness is intended)
  - maintenance effort is high (for new design derivatives)

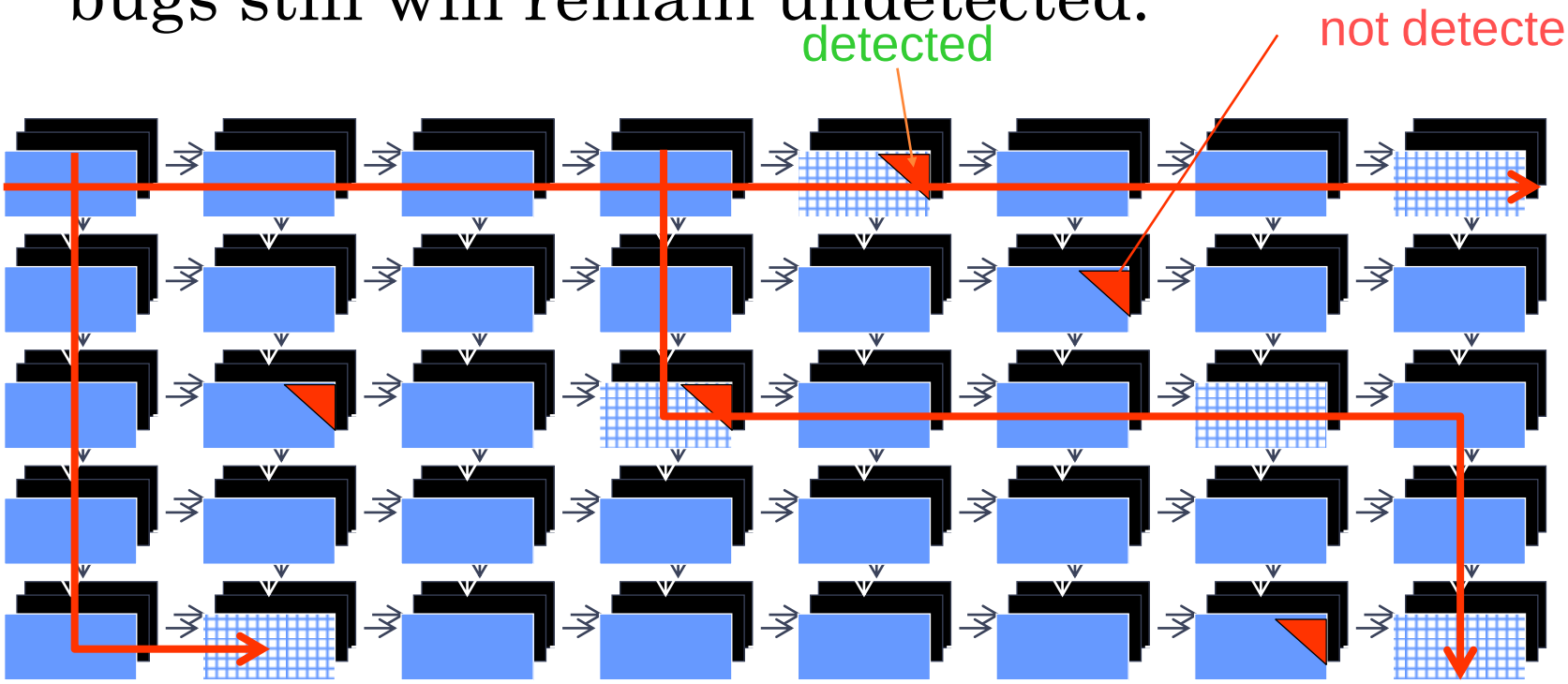
It alone can't implement all identified tests in test plan within project schedule!! Therefore, use random…

WHY RANDOM GENERATION?

- Will distribute cases better than direct cases, touching different interesting areas
- Automation (that is an assumption in stimuli random generation methodology)
- Less effort in writing tests
- Less sensitive to design changes
- Random will exercise “easy to reach” (we won’t waste our time on what automation can do)
- It is not used in opposition to direct cases, but in complementation

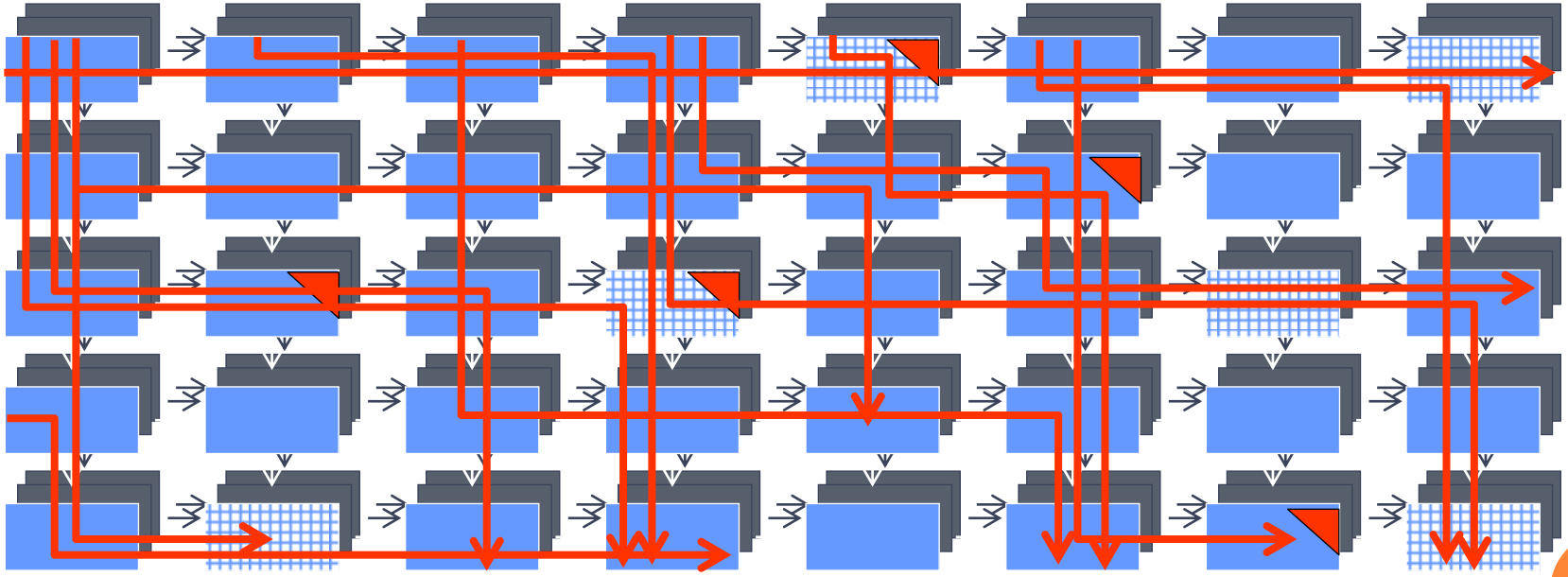
Illustrating Random Generation

- There are many ways to reach a given state
- Direct will lead to targeted states at expense of large development and planning time, and many bugs still will remain undetected.



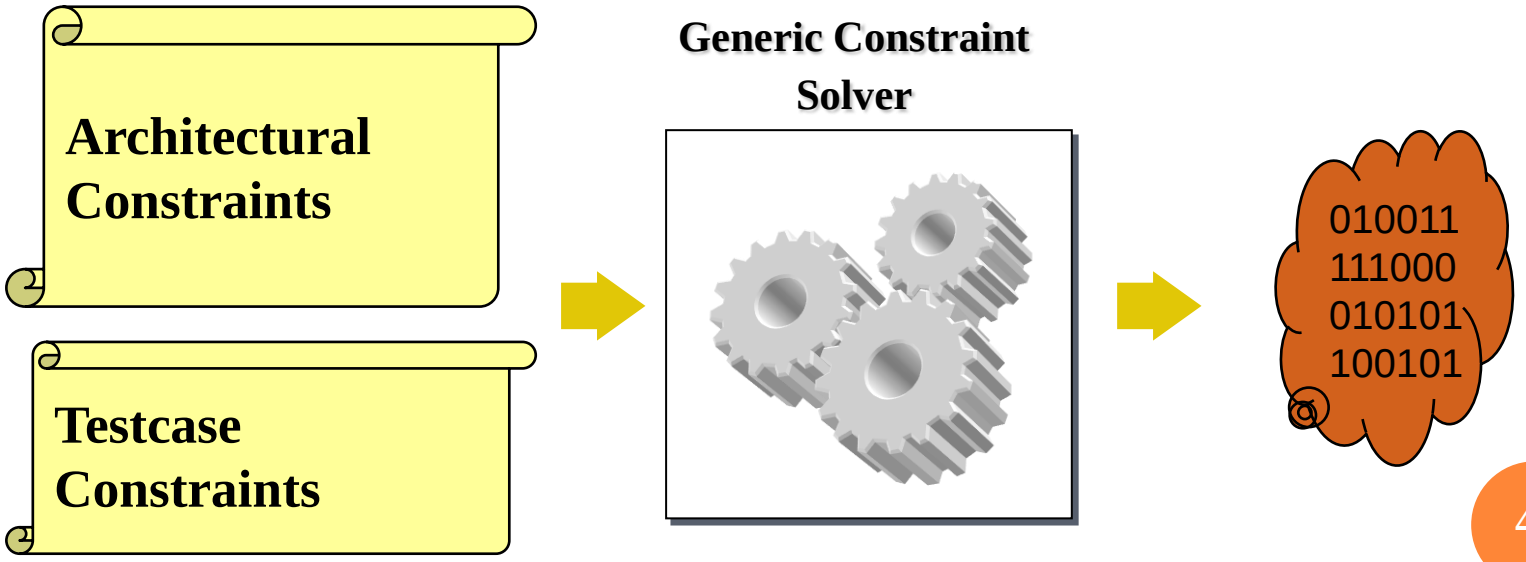
ILLUSTRATING RANDOM GENERATION

- Automatic generation can be biased toward goal areas
- Different paths will be explored quickly



CONSTRAINT SOLVER (AUTOMATION)

- Set of constraints may be listed
- A generic solver takes all the constraints and decides on ordering and combining constraint requirements
- The verification engineer focuses on providing the rules for generation, rather than creating the generator itself



SIMULATION-BASED VERIFICATION CLASSIFICATION BY CHECKING STRATEGY

- Checking refers to the act of assessing if differences occurred in the verification process. Three types:
- Manual checking per test case
  - A set of observations is defined for every test case
- Manual checking of logs
  - A log is generated for a run and analyzed
- Self-checking
  - Automated approach

SELF-CHECKING TESTBENCH

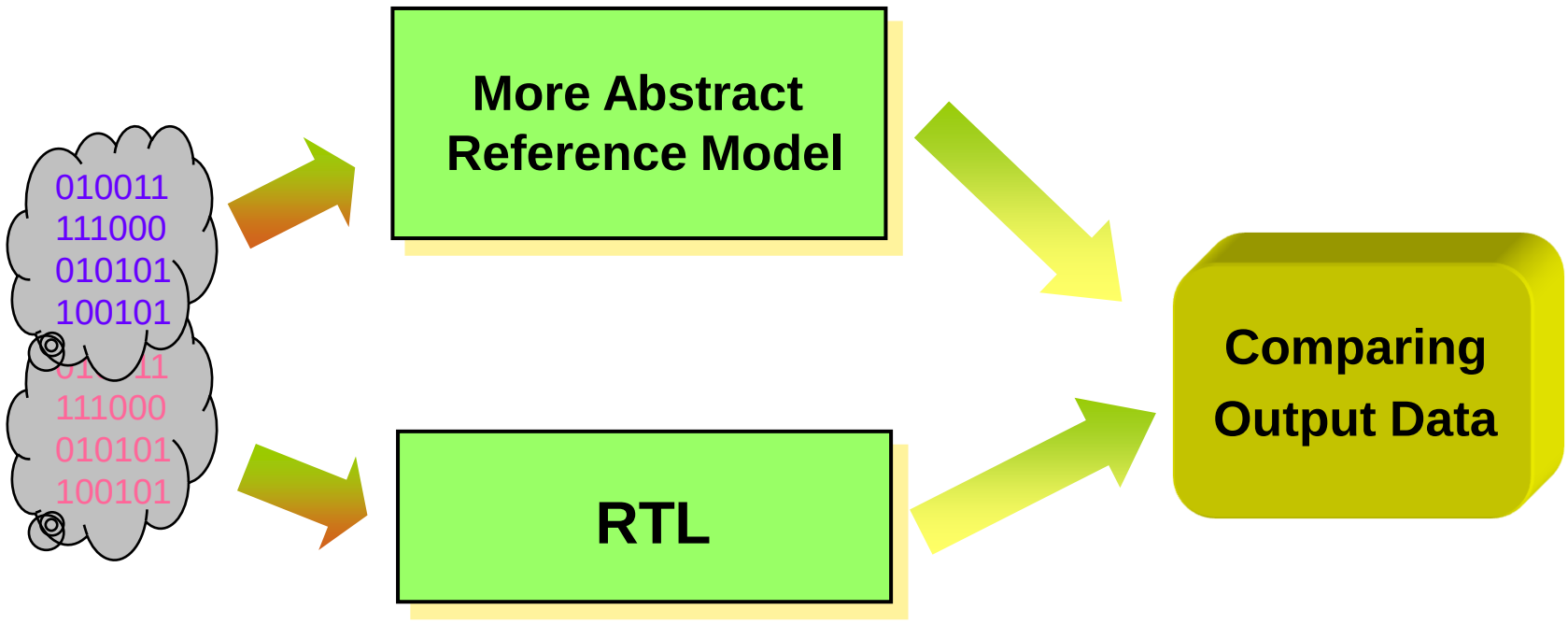
- All checks are part of the testbench
  - Written once, rather than in each test case
  - Real-time checking, while the test is running
  - All checks are in effect for ***every*** test case
  - Stop simulation immediately upon error detection

- Self-Checking – Two major Categories:
  - Data checks
    - *Are the **results** correct?*
  - Temporal properties
    - *Are the **protocols** being followed?*

How to Check Data?

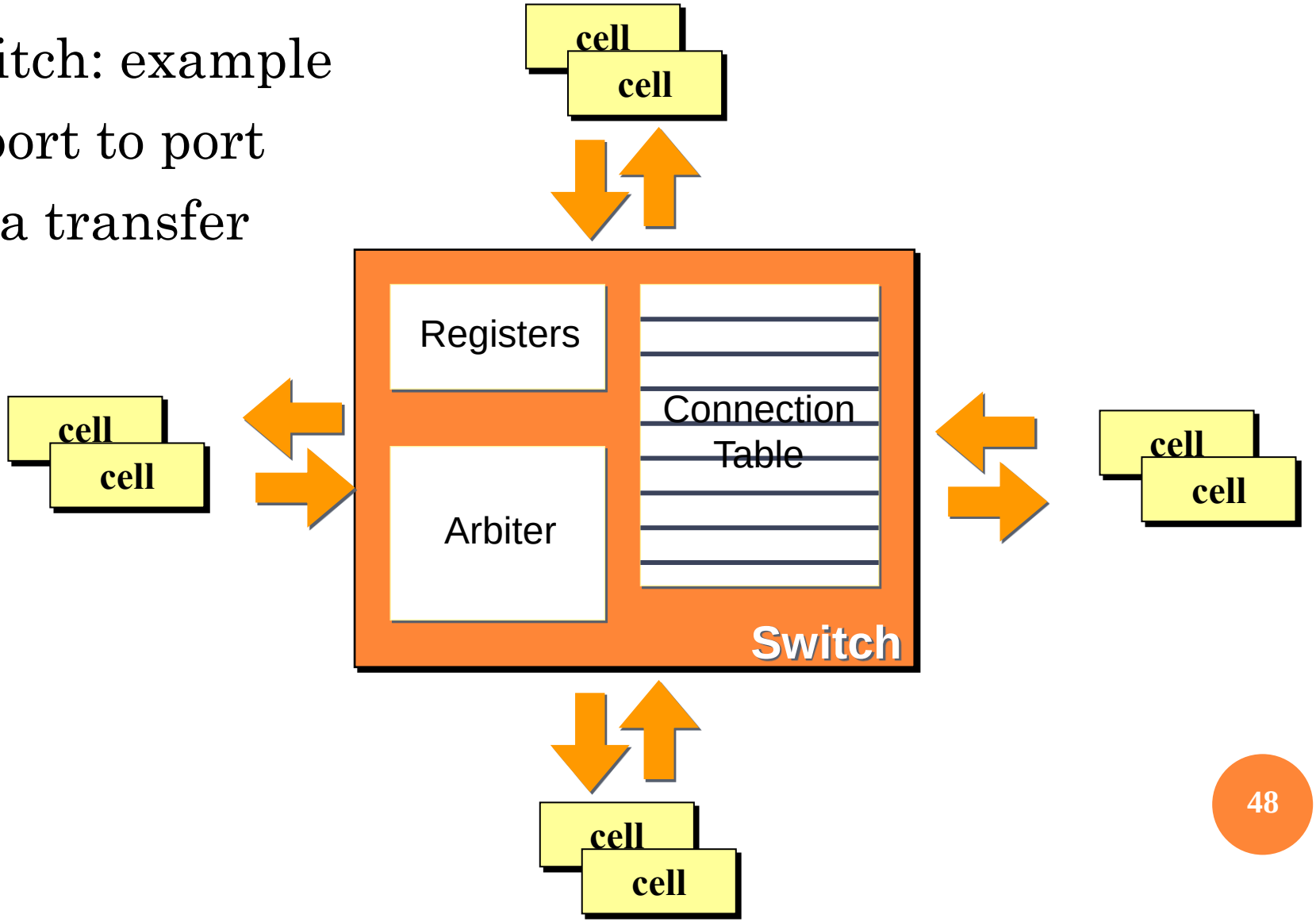
- Need to predict the expected results
  - Case 1: If the device performs complex algorithmic transformations, need to create “golden model”, basically a high-level reference model
  - Cae 2: If the device moves data from port to port, a “scoreboard” can be used for tracking the flow

DATA CHECKING BY REFERENCE MODEL



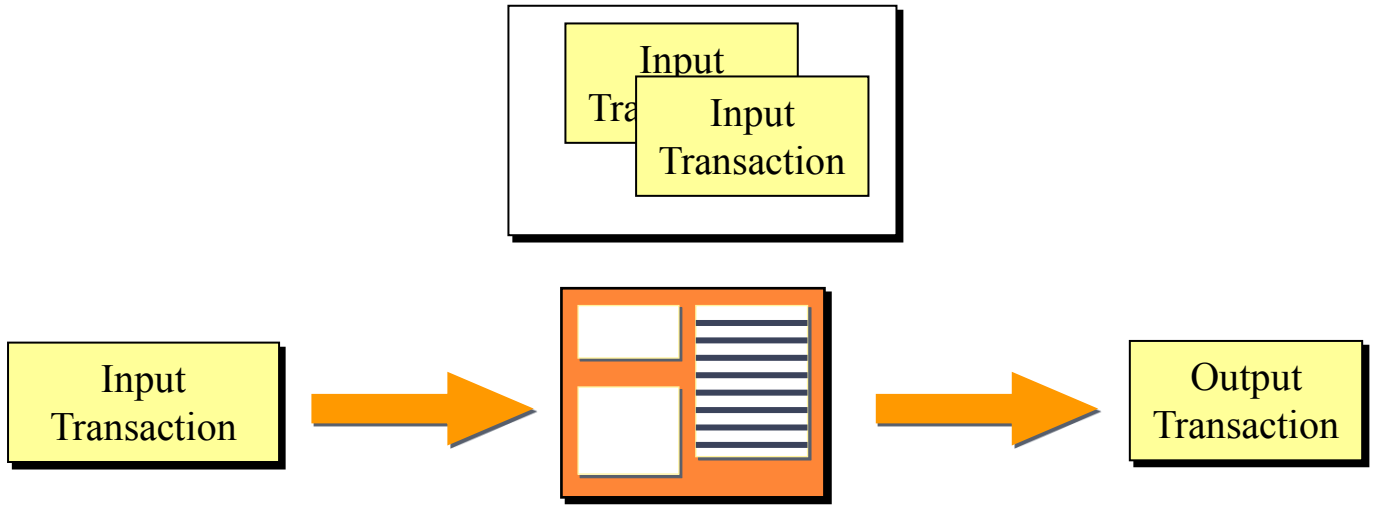
DATA CHECKING BY SCOREBOARD

Switch: example of port to port data transfer

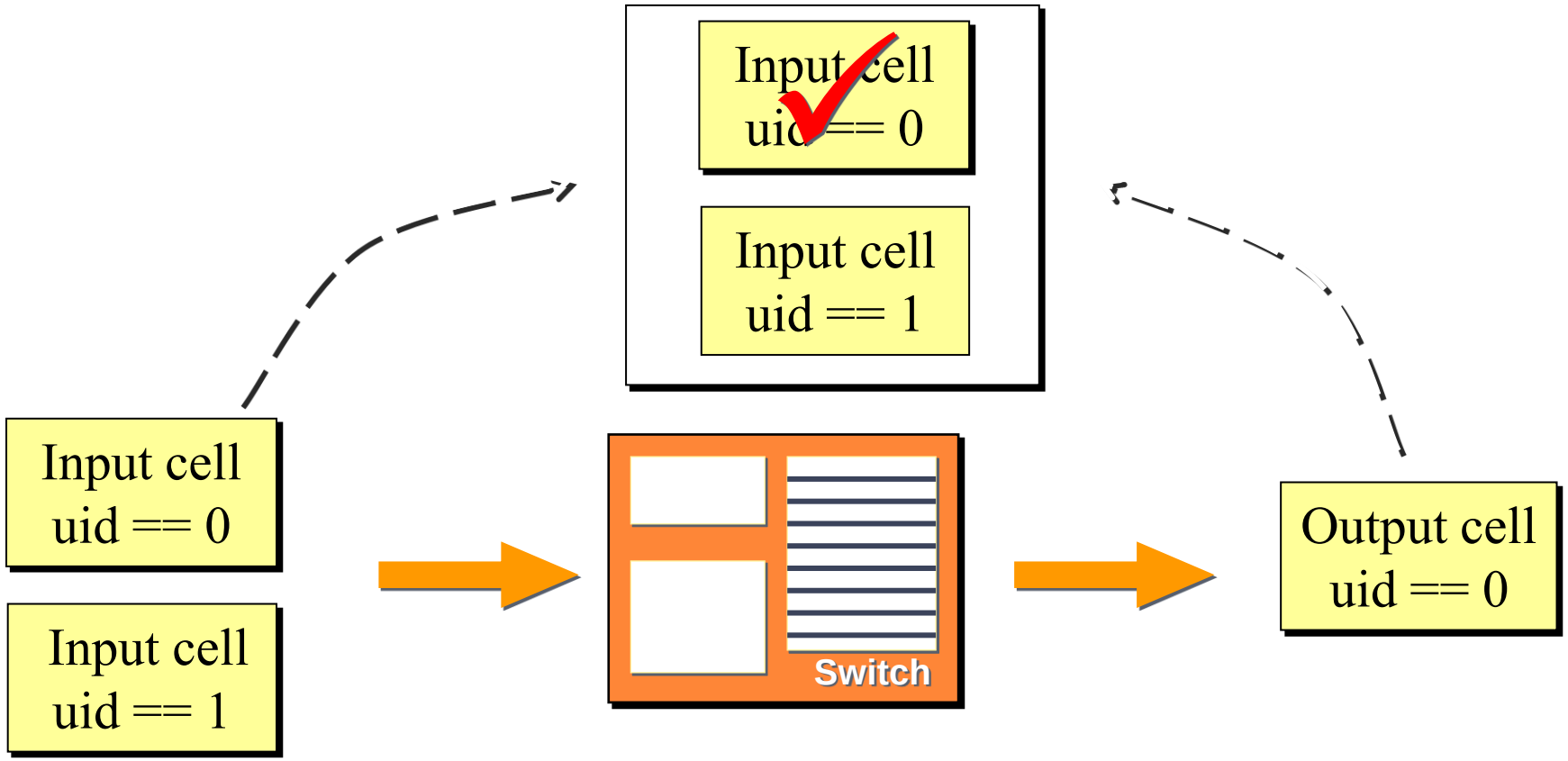


DATA CHECKING BY SCOREBOARD

- Put created transactions on the scoreboard
- Compare emerging transactions
  - Collect output transaction (none or minor differences to input data)
  - Find corresponding item on scoreboard
  - Use unique IDs



DATA CHECKING BY SCOREBOARD



HOW TO CHECK PROPERTIES?

- Assertions are formal and unambiguous statements about a design's intended behavior (property)
- Assertions state a safety (invariant) or a liveness property
- Temporal assertions monitor interface protocols, inter-module protocols, timings, sequences of events etc.

PROCEDURAL ASSERTIONS

- Suppose we had a generic logical checker that reads in procedural assertions (when procedure is active)…

- Spec:

  In occurring a transition in new_event, being REQ the current state, then assert that either req 1 or req 2 or both are in logic '1'.

- Implementation (System Verilog):

```
always@ (new_event)
  if (state=REQ)
    assert (req1 || req2)
```

Procedural Logical Assertion

DECLARATIVE ASSERTIONS

- Suppose we had a generic temporal checker that reads in declarative assertions (all times)…
- Spec: **Upon a request, an acknowledge should occur 2..6 cycles later, then a grant within 1..10 cycles.**

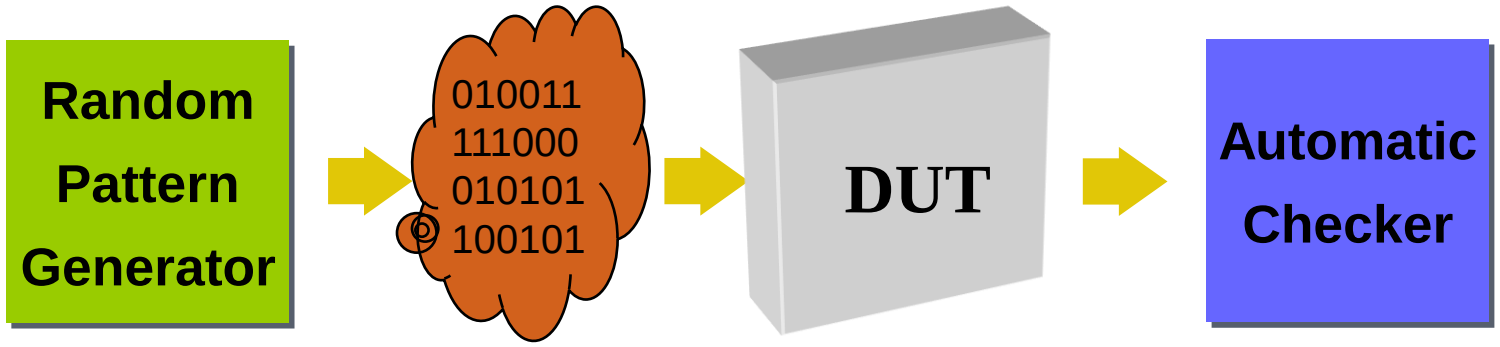
  If no acknowledge occurs, then an abort should occur within 10 cycles of the request.
- Implementation (e Language):

```
expect @request =>
    {[2..6]; @acknowledge; [1..10]; @grant}
  or
    {[1..10]; @abort}
else dut_error...
```

Declarative temporal assertion

AUTOMATED VERIFICATION

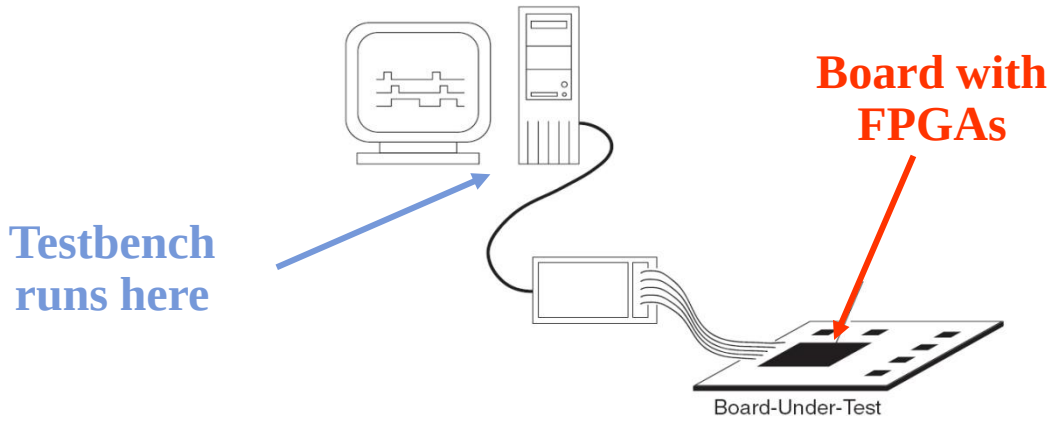
- Two tasks have been systematically automated in testbenches:
  - Automatic random generation of inputs
  - Automatic checking of the results
- Test plan: Test families rather than test cases
  - Saves manual effort of choosing values
  - Improves test coverage



(a group of cases)

“Real Life” Functional Verification

- Real hardware is implemented (FPGAs, development boards, etc.)
- Either by emulation or prototyping- the difference is on observability limitations.
- Although fast execution after implemented, development may take quite long time.



Functional Verification by Emulation

- Based on re-programmable FPGA technology.
- Modular approach
  - map the modules onto the components of the emulation machine (as many FPGAs is needed), keeping the design inter-module signals (high observability)
  - Observe the signals (high number of interconnects).
- Board development is high
- Close to final implementation.
  - Can boot operating system, give look and feel for final implementation into real system.
  - Timing approximate

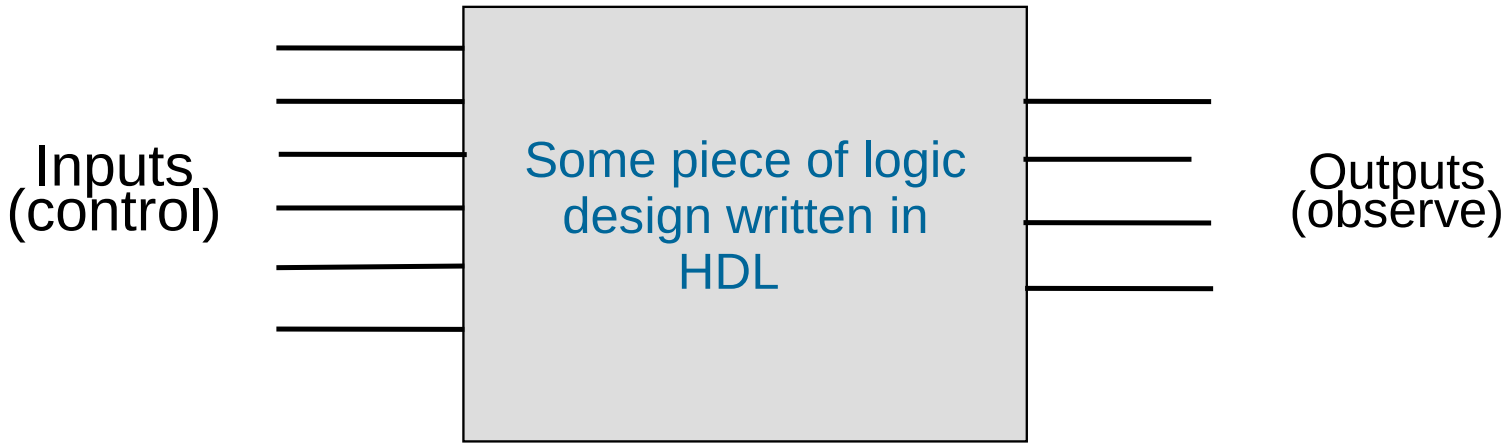
FUNCTIONAL VERIFICATION BY PROTOTYPING

- Also based on re-programmable FPGA technology.
- Build the hardware implementation of the design and operate it (not modular approach)
- Observe the I/O ports (low observability)
- Board development time better than emulation
- Close to final implementation.
  - Can boot operating system, give look and feel for final implementation into real system.
  - Timing approximate

FUNCTIONAL VERIFICATION BY DUV OBSERVABILITY

- Black-box:
  - without any idea of internal implementation of design
  - without direct access to the internal signals
- White-box:
  - has full controllability and visibility on internal structure
- Gray-box:
  - compromise in between

BLACK-BOX



•To verify a black box, one needs to understand the function and be able to predict the outputs based on the inputs (the specification, not the implementation).

• The black box can be a full system, a chip, a unit of a chip, or a single macro.

WHITE-BOX

- White-box verification means that the internal facilities are visible and utilized by the testbench stimulus.
- Assertions usually hold properties related to internal signals.
- Examples: Unit/Module level verification

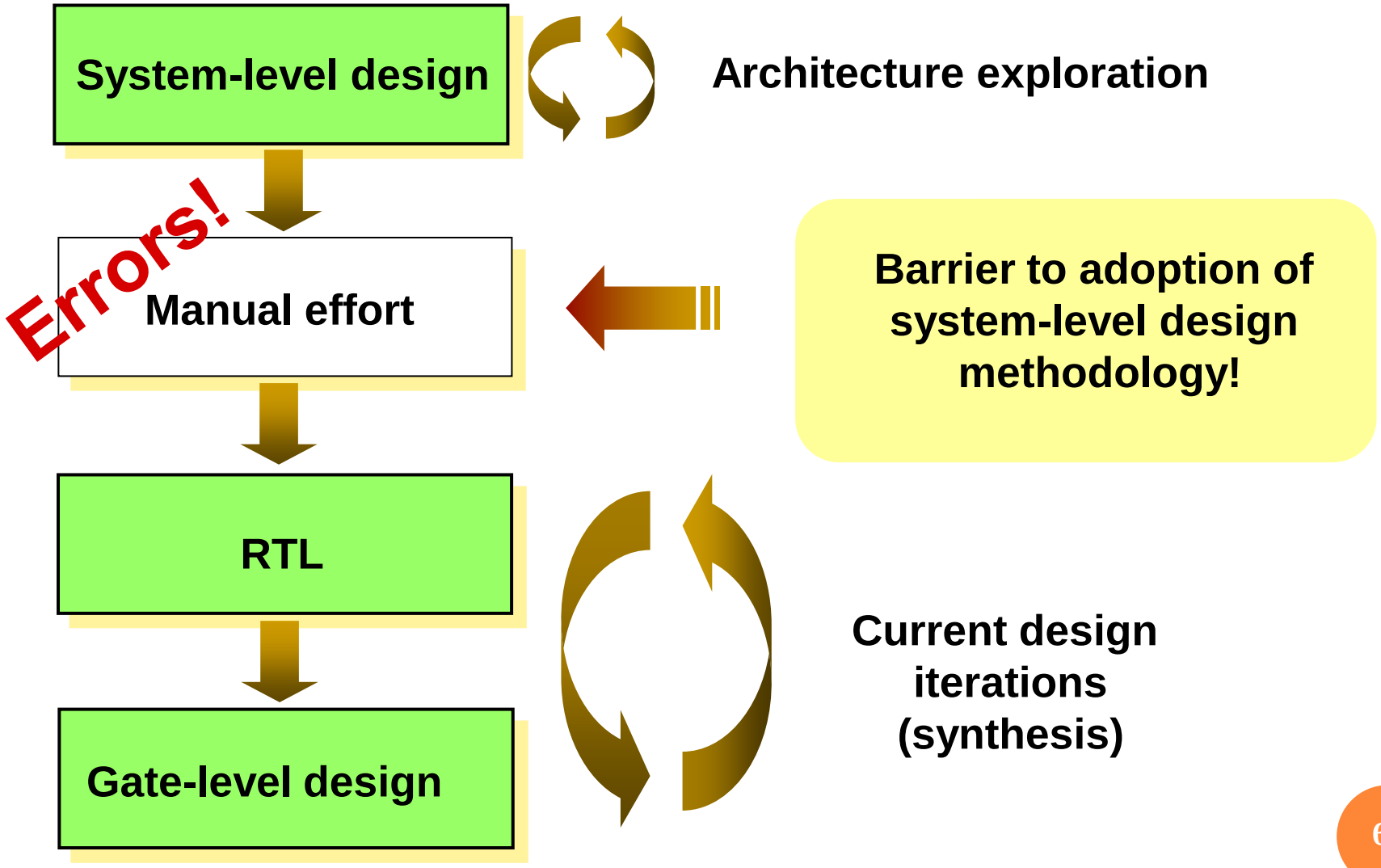
GRAY-BOX

- Gray-box verification means that a limited number of facilities are utilized in a mostly black-box environment.
- Example: Most environments! Prediction of correct results on the interface is occasionally impossible without viewing an internal signal.

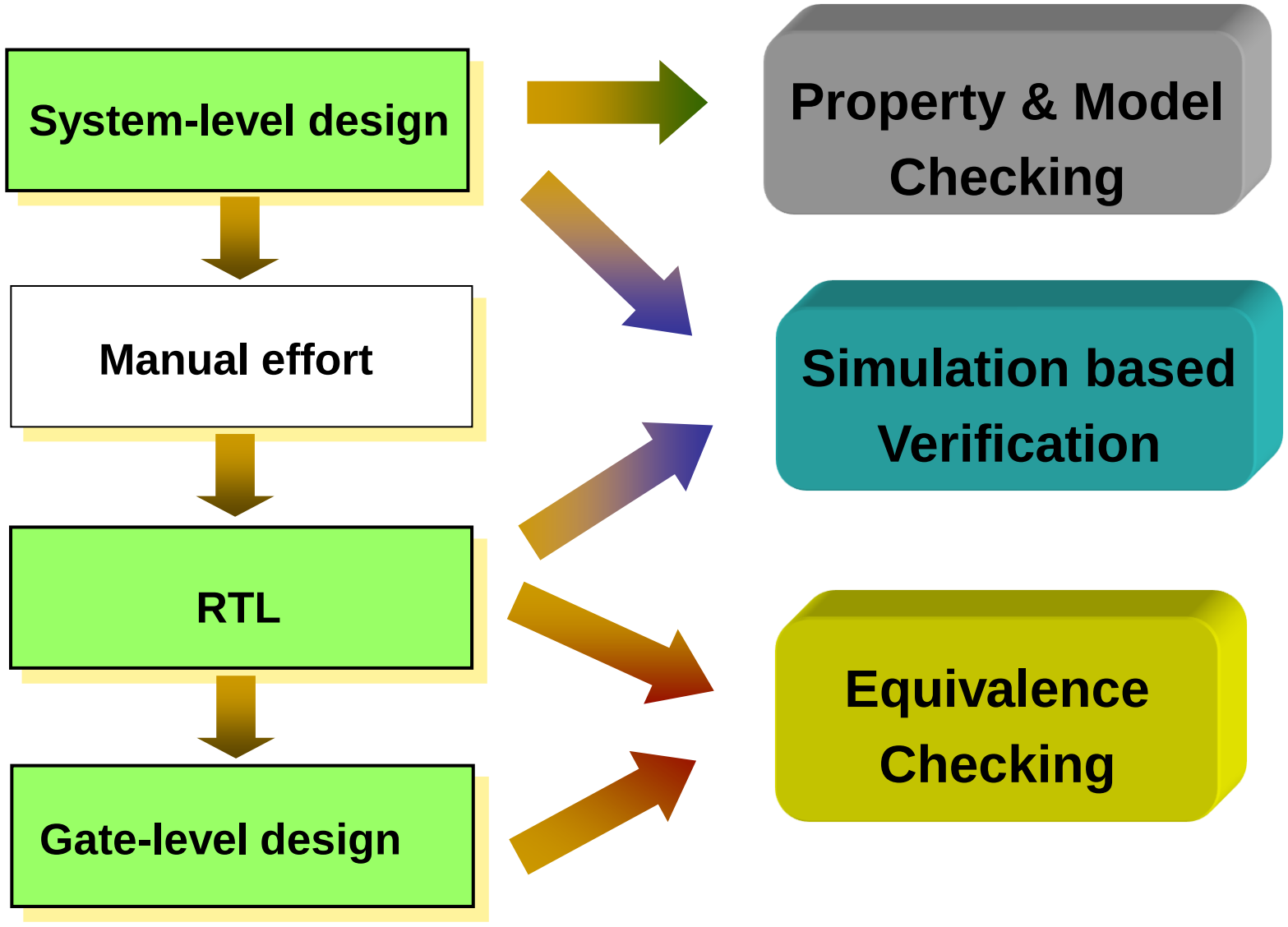
BLACK-BOX vs. WHITE-BOX ?

- In a black-box approach
  - The tests are more reusable(flexible) in the future
  - Good for early testing with behavioral models
  - The testbench can be developed in advance (along with DUV)
- In a white-box pieces
  - Generates interesting combination of states and checks them quickly
  - Requires deeper involvement of the designers!
  - Caution on test reuse (affected by changes in the RTL code)
- Gray box is usually suggested: start with a black-box approach and, later, add white-box tests.

CURRENT MOVE TO SYSTEM-LEVEL DESIGN



SYSTEM-LEVEL VERIFICATION



FUNCTIONAL VERIFICATION BY COVERAGE

- Verification team needs assurance on quality and completeness of their work.
- Coverage is the degree of completeness/quality.
- Since full verification is not possible, how to assess that?
- Different from manufacture testing, the fault points (bugs) are not directly related to any fault model; the bug is not structural.
- Formal vs. Simulation-based
  - In simulation-based verification the assessment is a must.
  - In formal verification, the coverage will depend on the completeness of the properties considered.

COVERAGE METRICS

- A coverage metric is a measurable parameter of the verification environment/device useful for assessing verification progress in some dimension.
- Coverage metric may be classified according its kind and its source. Two main classes:
  - Structural (implicit) coverage metrics: related to elements from the representation of the DUV at the referred abstraction level. Ex. Lines and transitions of signals described in code.
  - Functional (explicit) coverage metrics: related to elements extracted by the user from the specification or implementation. Ex. Buffer overflow, packet length

CODE COVERAGE

- Determines if all the implementation was exercised
- Metrics are defined by the source code (implicit)
  - Line/block, events, branch toggle…
- Strengths
  - + Reveals untested logic
  - + Reveals holes in functional test plan
  - + No manual effort is required to implement the metrics, auto. collect
- Weaknesses
  - – No signals cross correlations
  - – No multi-cycle scenarios(temporal correlation)
  - – Manual effort required to interpret results

SOME (CODE) COVERAGE METRICS

- Toggle coverage: Which bits of the signals in the design have toggled?
- Signal coverage: How well (# of times) were state signals or ROM addresses exercised?
- Statement coverage: How many times was each statement executed?
- FSM arc(transition)/state coverage: How many transitions/states of the FSM were processed/accessed?
- Branch coverage: Which "case" or "if...else" branches were executed?

FUNCTIONAL COVERAGE

- Determines if all the functionality was tested
- Metrics are user-defined (explicit)
- Strengths
  - + Complete expressiveness
    - cross-correlation, and multi-cycle scenarios (temporal correlation)
  - + Objective measure of progress in regards to test plan
  - + Very little effort needed to interpret the results (informative)
- Weaknesses
  - – Only as good as the coverage model- depends on test team expertise
  - – Manual effort is required to implement the metrics

COVERAGE PROGRESS

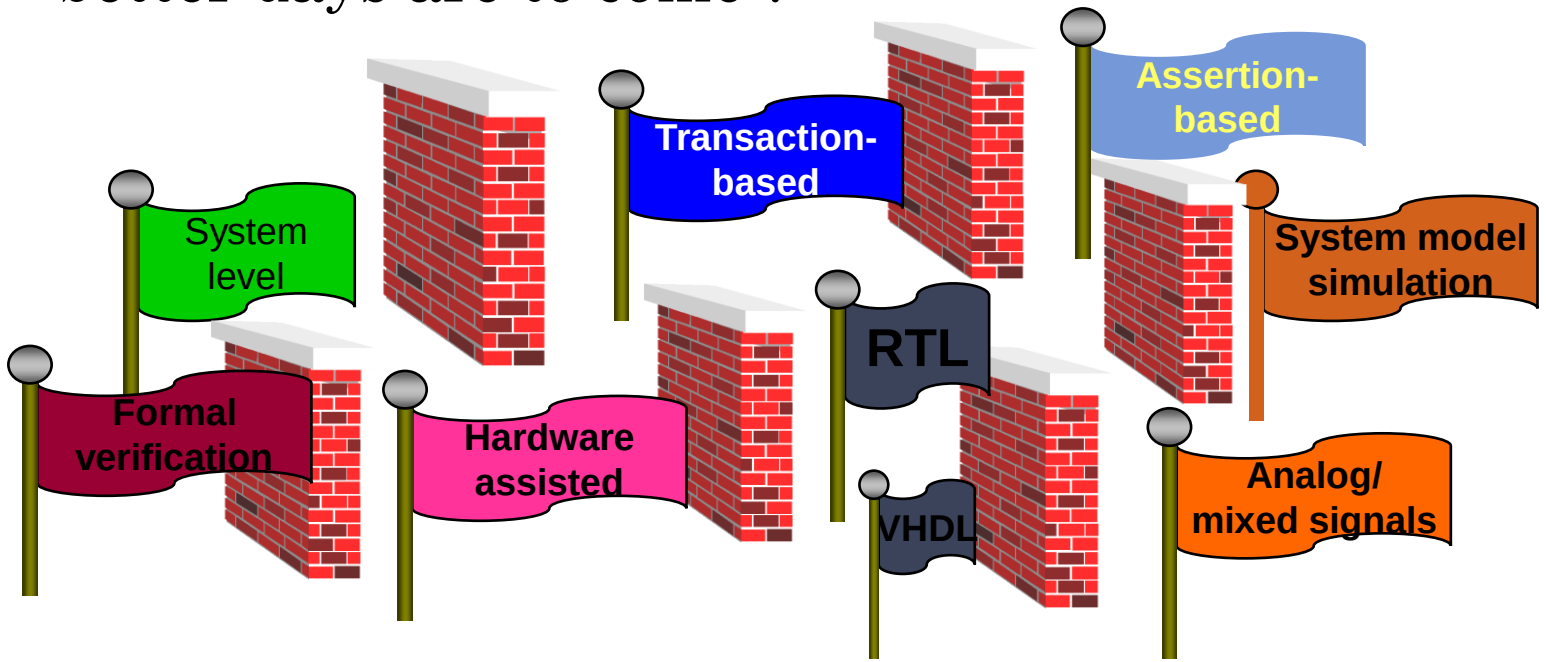
- 100% coverage means that all coverage items (properties, events) measured by coverage metrics have been reached the targeted coverage (defined by verification team in coverage model)
- During simulation, evolution of each individual coverage item follows a particular rate
- During simulation, the difference between the 100% coverage and the current coverage is generically denominate hole
- High coverage means proximity to100% coverage
- Low coverage means a coverage measurement far from 100% coverage

USING BOTH COVERAGE APROACHES

Functional Coverage	Code Coverage	Interpreting the Coverage
Low	Low	Less test scenarios covered, maybe too few pattern stimulated
Low	High	Multi-cycle scenarios/corner cases still to be covered
High	Low	Test plan and/or Coverage Metrics are inadequate, incomplete func. test scenarios
High	High	High confidence of test quality

VERIFICATION LANGUAGES

- Verification technologies/languages today are badly fragmented (e, Specman, SystemVerilog, OpenVera, etc)
  - forcing engineers to move from one incompatible environment to another
  - better days are to come ?



CONCLUDING REMARKS

- Simulation-based functional verification is a widespread practice throughout industry, but will never be “the solution”
- Formal verification is not the solution either
- Verification tasks are fundamental parts of design flow of digital systems

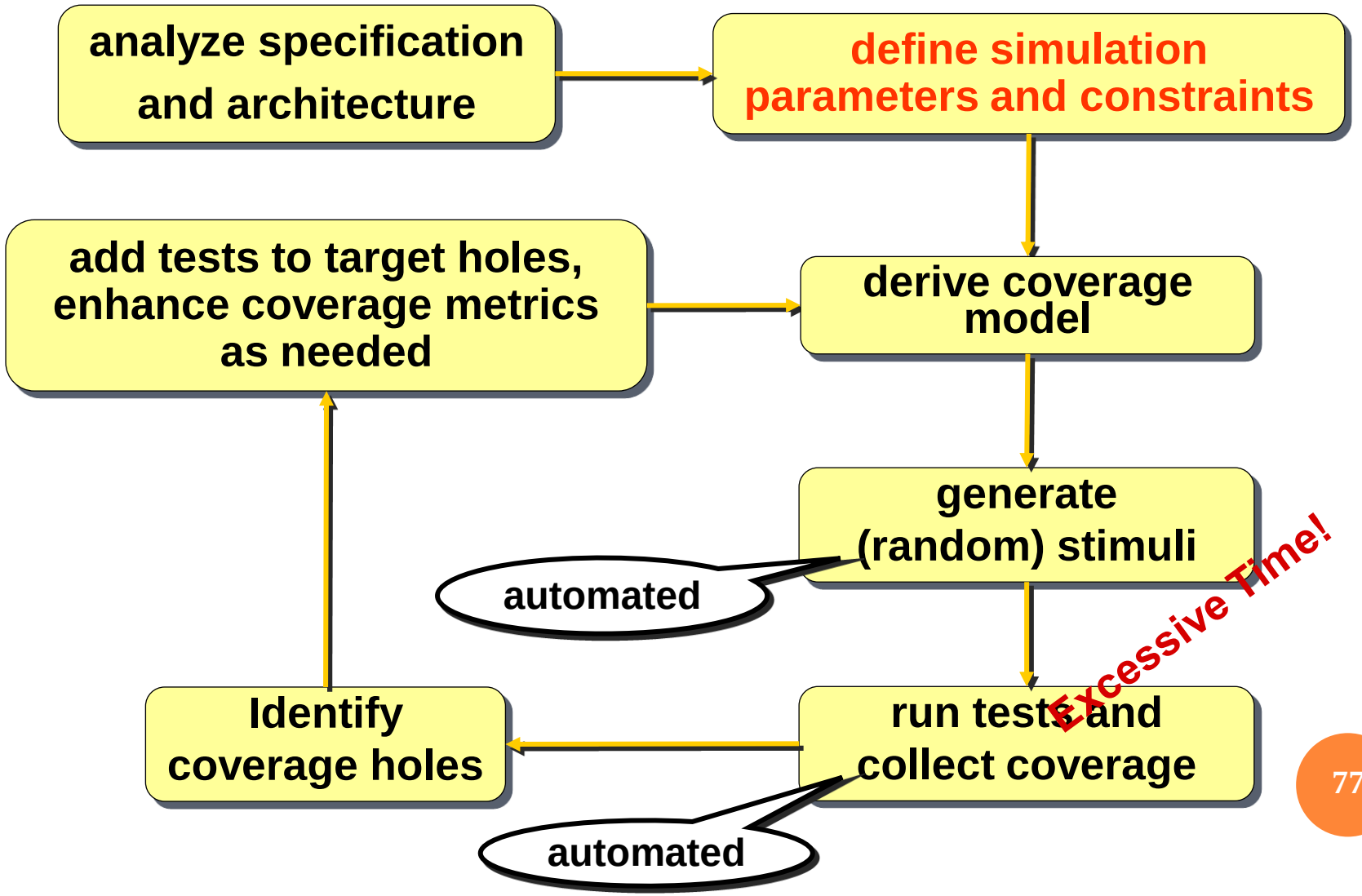
Thank you

Contact:
eromero@lme.usp.br

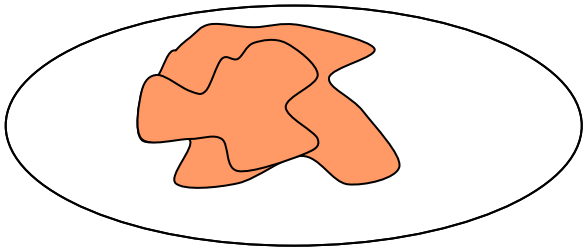
Coverage-Driven Verification (CDV)

- Verification teams try to assess the degree of completeness/quality of their work through coverage metrics
- The verification process, at least in a complete run, is finished (verification closure) when the coverage target is reached
- In CDV methodology, the coverage measurement and analysis are at the center of the process and determine how stimuli are generated (PIZIALI)
- CDV is still restricted to the internal use of some companies and academia.

Traditional Verification Flow

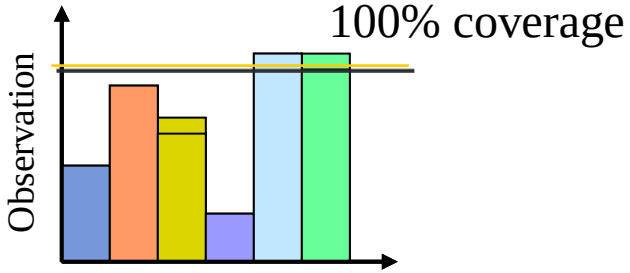


Why Coverage-Driven Verification?



testcases (input) space

Unknown Relationship



coverage events

Advances of individual coverages are unbalanced

there exists delay for reaching 100% coverage

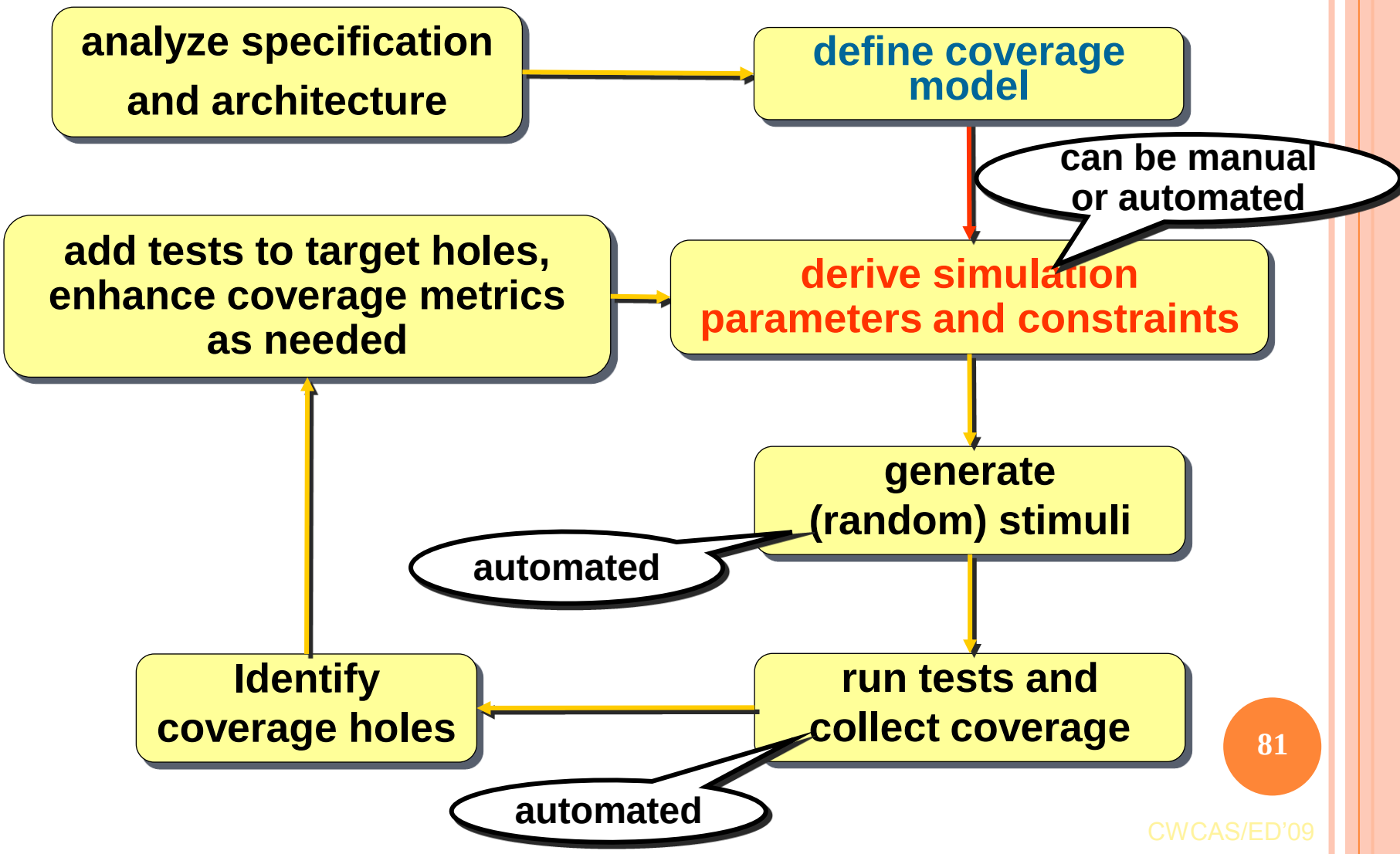
Understanding Coverage-Driven Verification

- What defines coverage-driven verification are
  - the fact that coverage measurement and analysis become the driver of the process
  - the constraint definitions and stimuli generation take a “back-seat” in this process
- There are several different techniques on CDV, which depend on
  - when and how stimuli generation constraints are updated
  - the mode of the measured coverage parameter (input, output or internal)

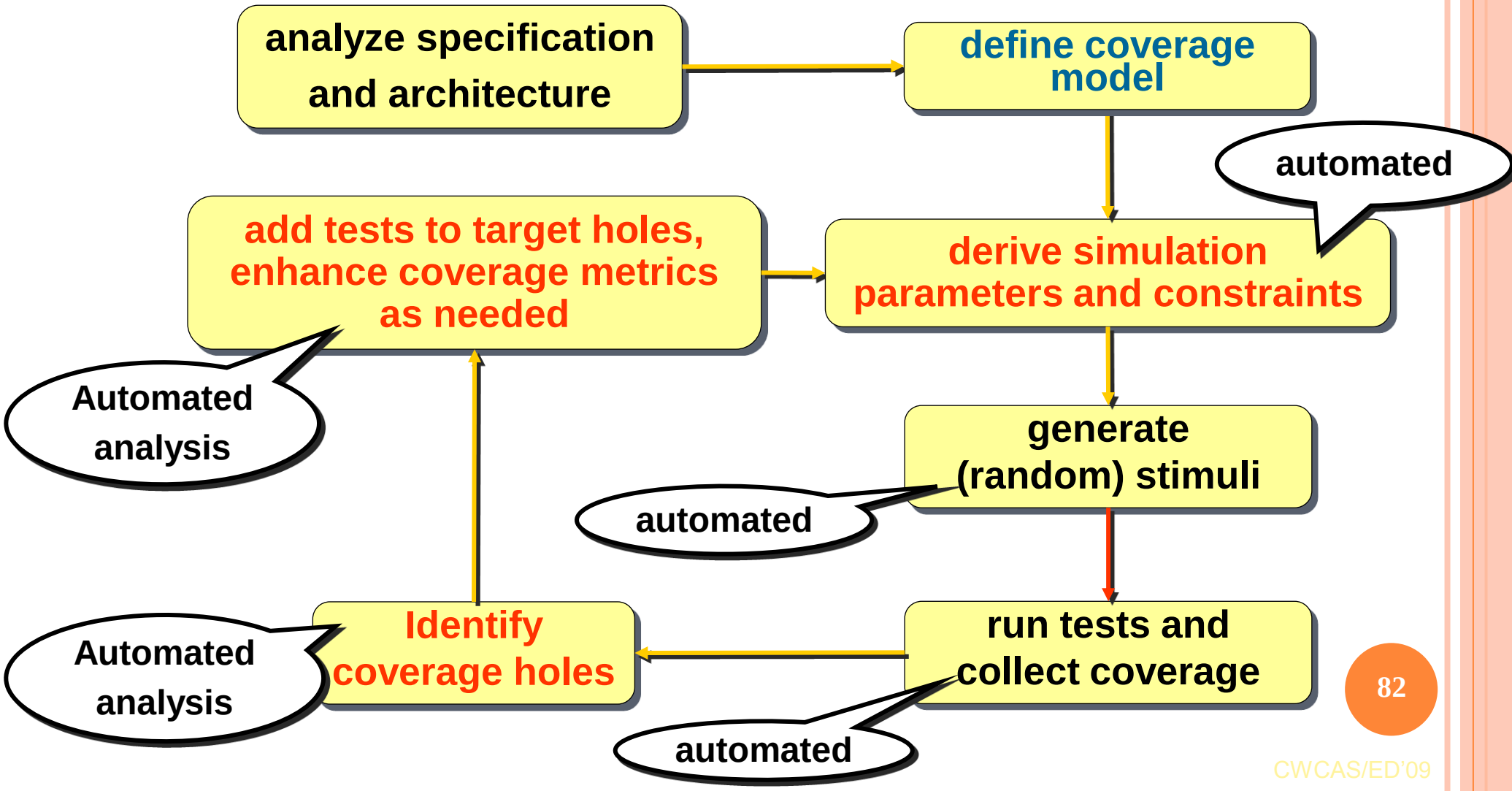
Understanding Coverage-Driven Verification

- CDV can be classified according to the form and moment the generation constraints are defined:
  - static CDV: coverage model is analyzed before any simulation and the constraints are derived (one-pass task)
  - dynamic CDV: coverage measurement is analyzed and the constraints are derived after simulation has started
- Static approach may be for direct or random cases
- Dynamic CDV aims the detection of holes in coverage and then to change the stimuli generation to correct it. It is a.k.a. **Coverage-directed Generation** (CDG)- random cases in closed loop

Static Coverage-Driven Verification



Dynamic Coverage-Driven Verification (a.k.a. CDG)



Understanding Coverage-Driven Verification

- Coverage can be classified according to the mode of measured coverage parameter :
  - input coverage: coverage is measured at the primary inputs of a device. Ex # samples of input signal x at value k
  - output coverage: coverage is measured at the primary outputs of a device. Ex. # samples of output signal z at value k
  - internal coverage: coverage is measured at the internal interface of a device. Ex # samples of internal node…
- I/O coverage is related to functional coverage objects whereas internal coverage may be also structural ones

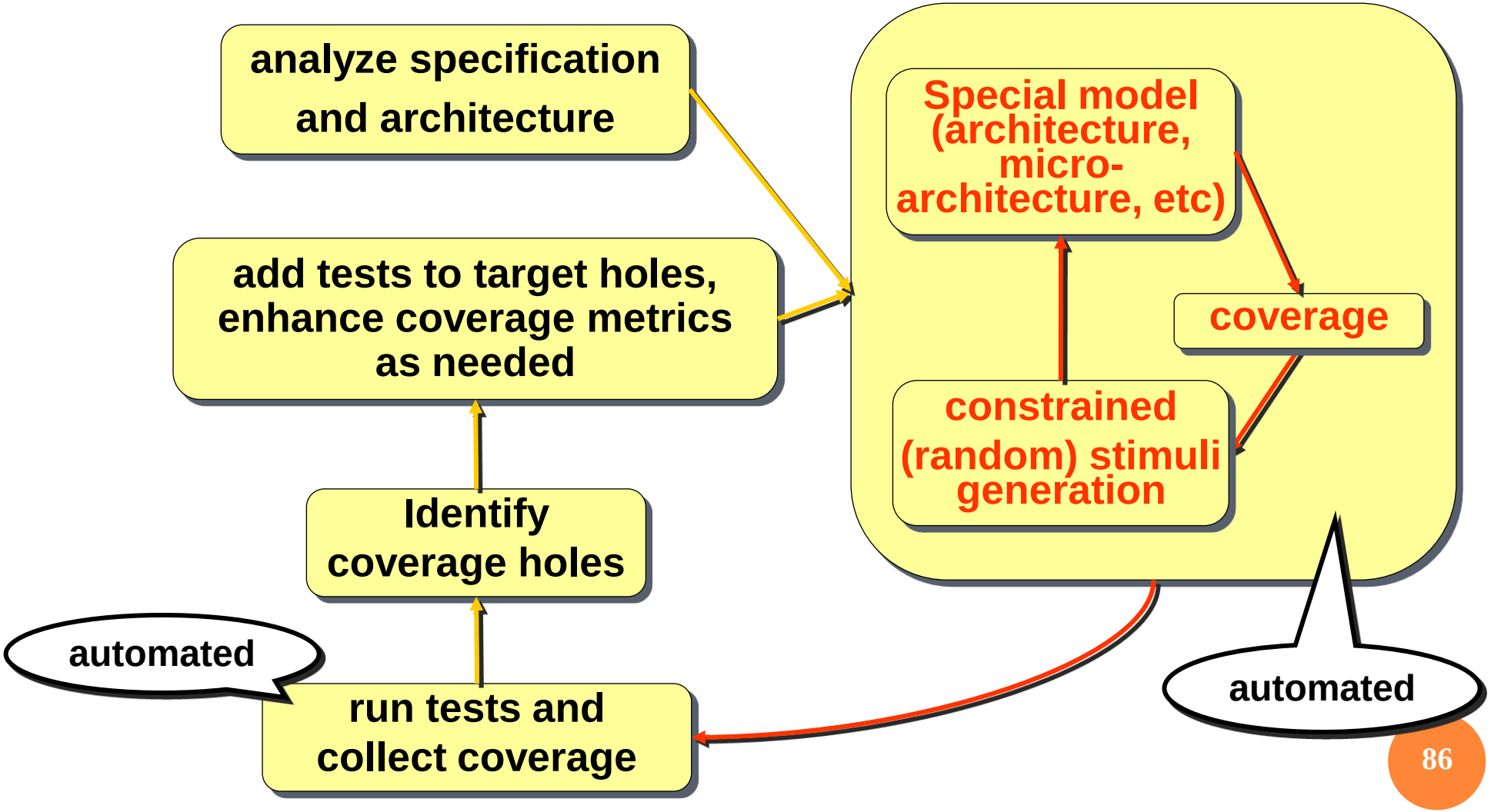
Based on Input Coverage

- With input coverage in coverage-driven verification, the random stimuli generation constraints are easily determined/computed.
- The coverage model parameters are also simulation input parameter
- The stimuli generation is similar for both static and dynamic approaches
  - in static CDV, the considered input coverage is the targeted coverage
  - in dynamic CDV, input coverage is assessed at any pre-defined time and holes are determined

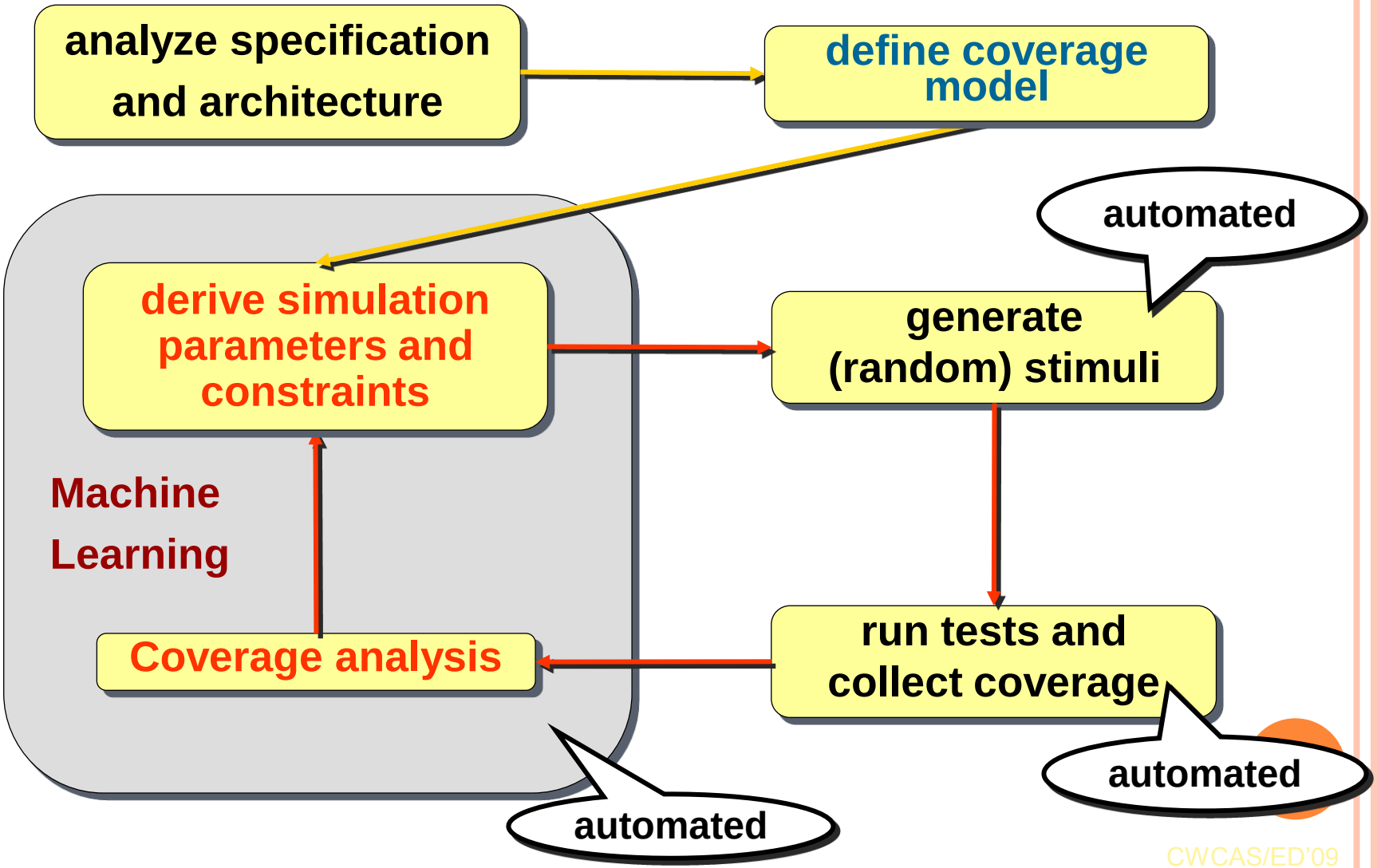
Based on Output/Internal Coverage

- Modeling of the relationship between coverage measurement and constraints derivation, or suitable testcases, is not trivial.
- Static and dynamic coverage-driven verification approaches are based on very different techniques
  - in static: to develop some model of stimuli= f(coverage holes), for the DUV
  - in dynamic: to develop some model based on data provided by simulation. The more data we have, the more precise the model will be (training)

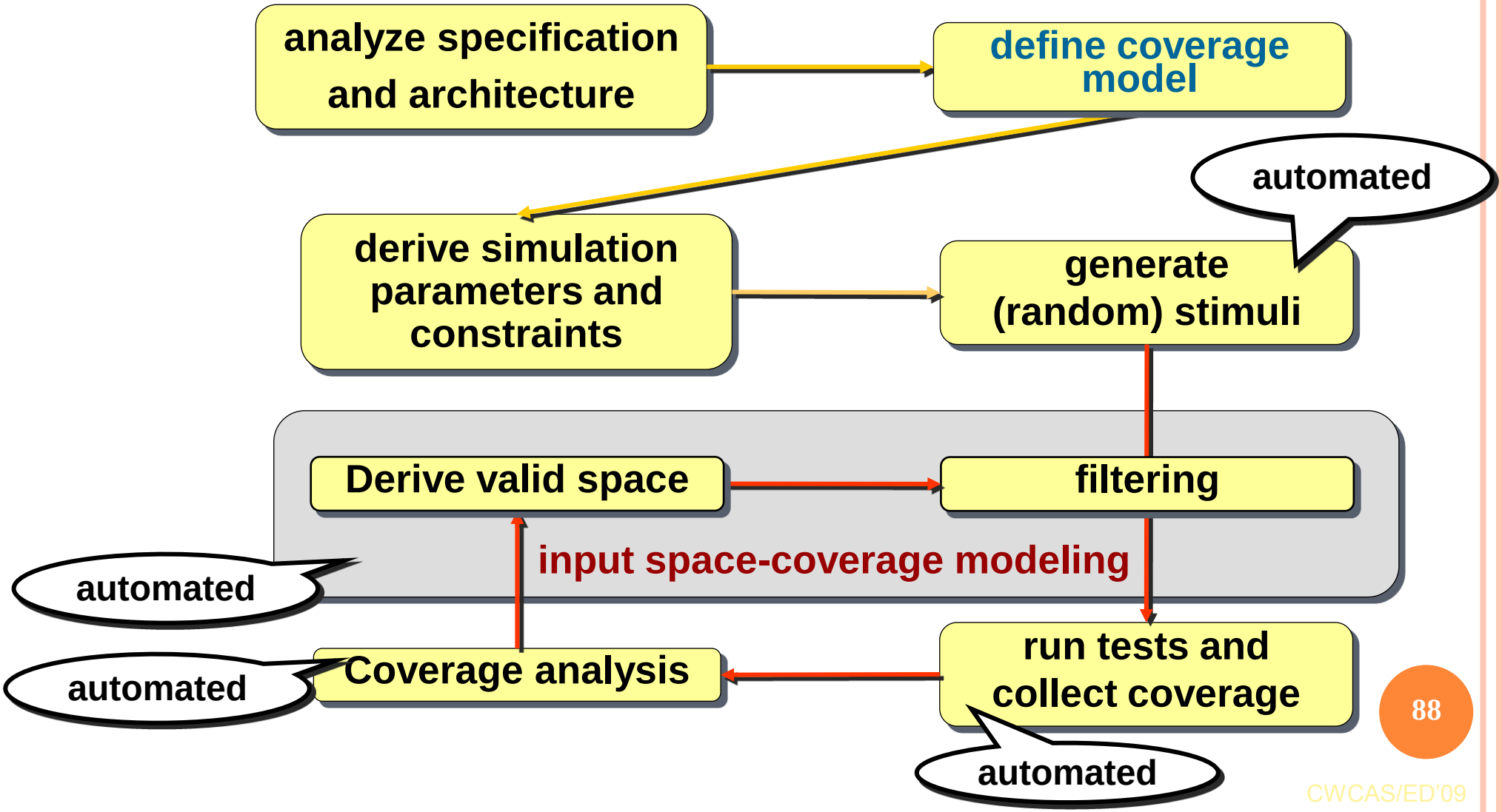
Output/Internal Coverage in Static CDV



Output/Internal Coverage in CDG by Constraint Generation



Output/Internal Coverage in CDG by Filtering



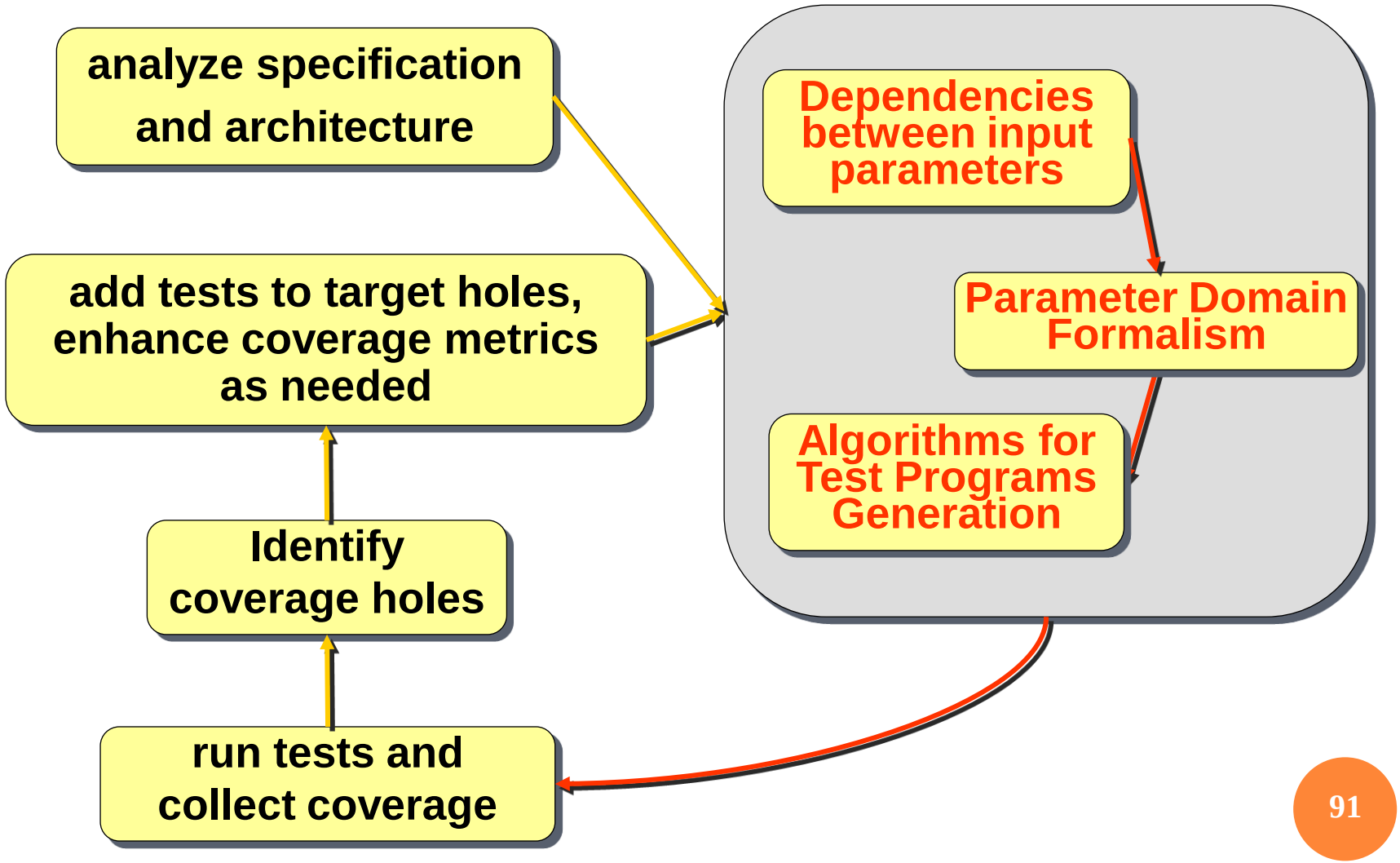
State-of-Art in Coverage-DrivenVerif.

	Input Coverage	Output/Internal Coverage
Static CVD	Jerinic 04, Castro 09	Ur 98 Saxena 00 Mishra 05
Dynamic CDV (Coverage-Directed Test Generation, CDG)	Guzey 08	Tasiran 01, Nativ 01, Bose 01, Fine 03, Braun 03, Corno 04, Romero 05, Hsueh 06

Static Coverage-Driven Verification

Input Coverage

Jerinic 04, Castro 09



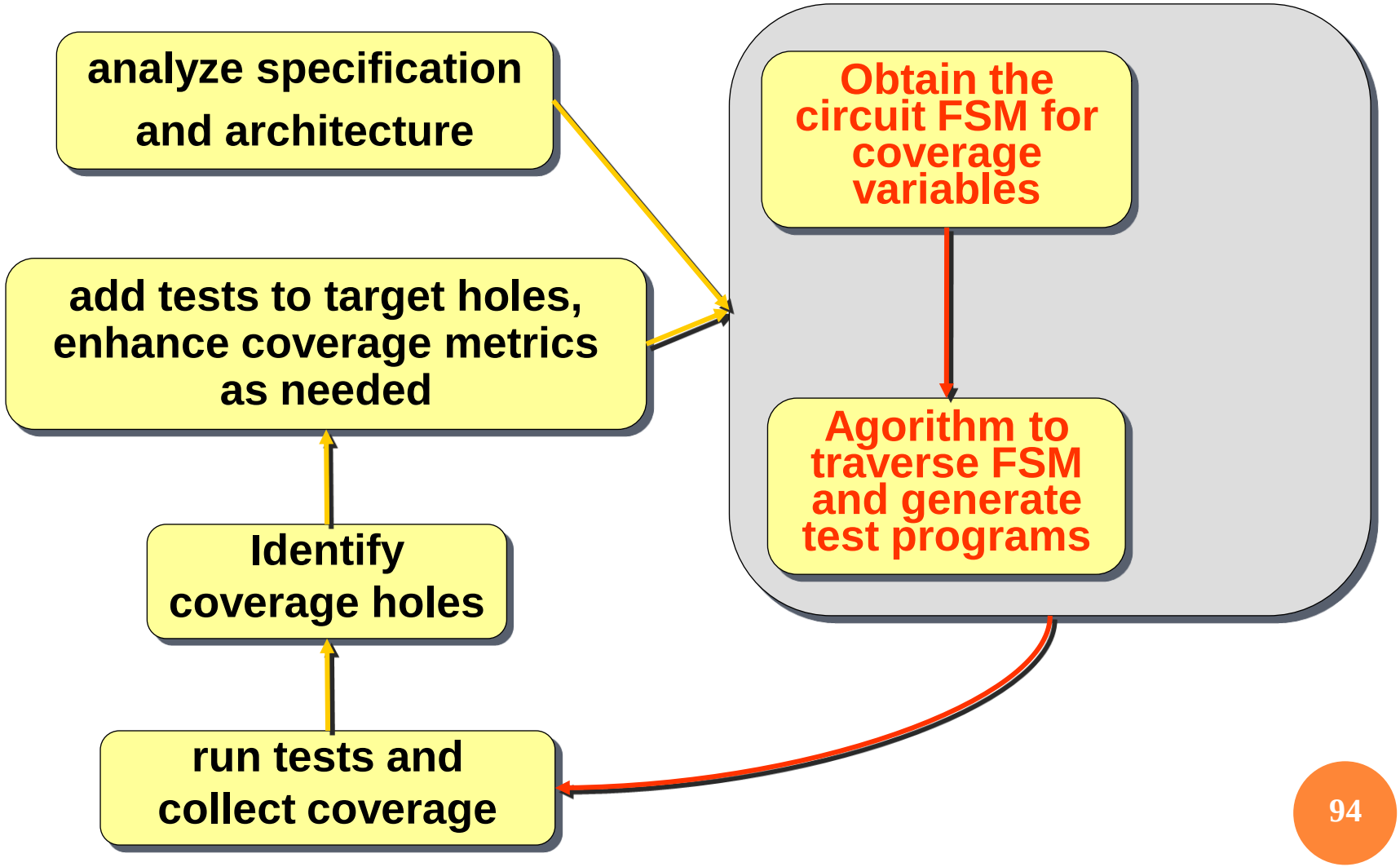
Jerinic04, Castro09

- Execution Unit, Bluetooth communication core
- Verify the tests cases but focused on the input space
- Elimination of invalid and redundant spaces causing coverage iefficiency
- Based on Parameter Domain Formalism.
- Directed testcases (Jerinic) and Random stimuli (Castro)

Static Coverage-Driven Verification

Output/Internal Coverage

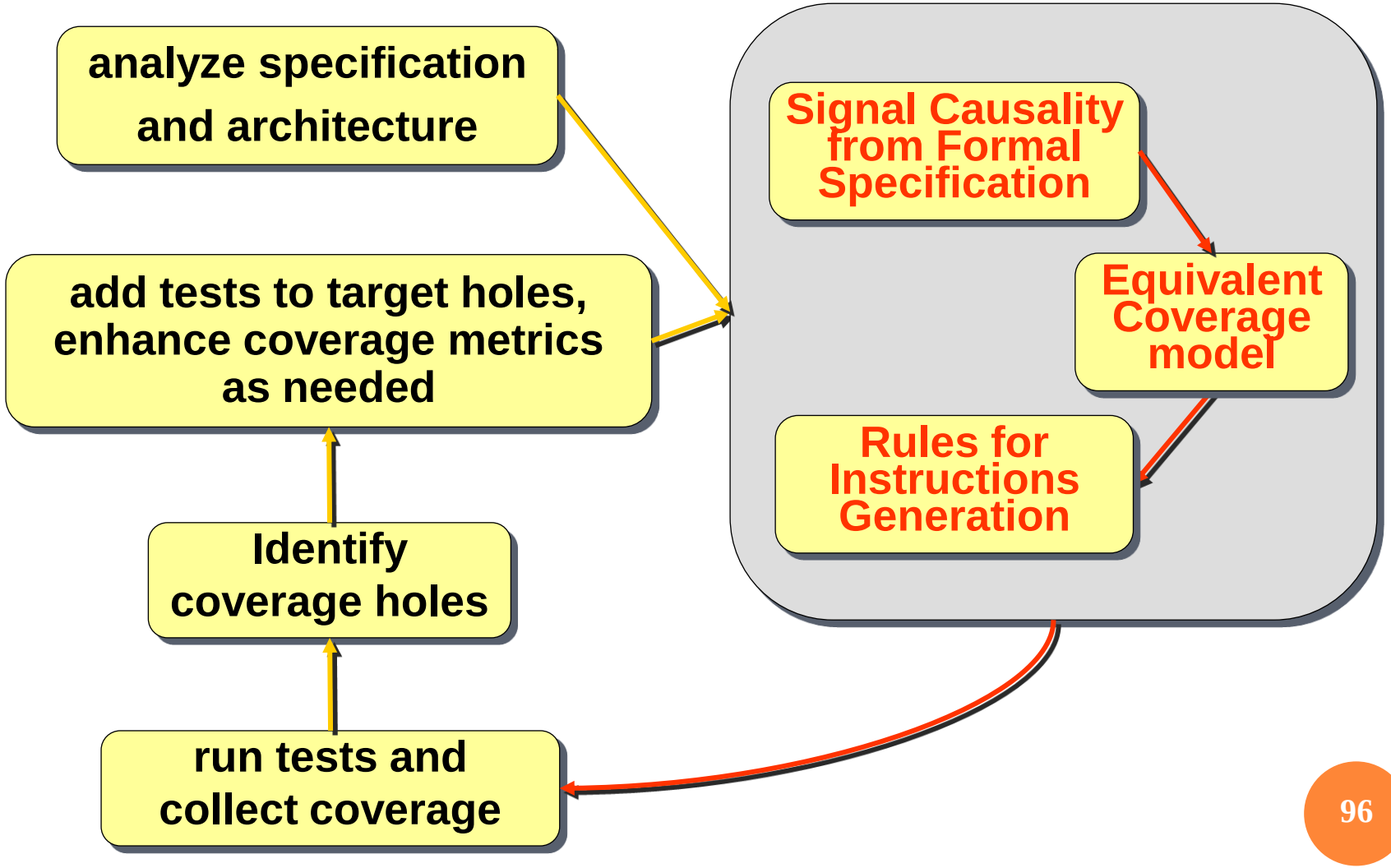
Ur 99


analyze specification and architecture
add tests to target holes, enhance coverage metrics as needed
Identify coverage holes
run tests and collect coverage
Obtain the circuit FSM for coverage variables
Agorithm to traverse FSM and generate test programs

Ur 99

- Generates test programs to a PowerPC
- From FSM description, choose variables as coverage points
- A tool traverse the FSM and generate de instructions capable of provoke the events related to the variables
- Generated test programs are direct tests
- Domain knowledge based approach (a strong knowledge of the specification or implementattion is due)

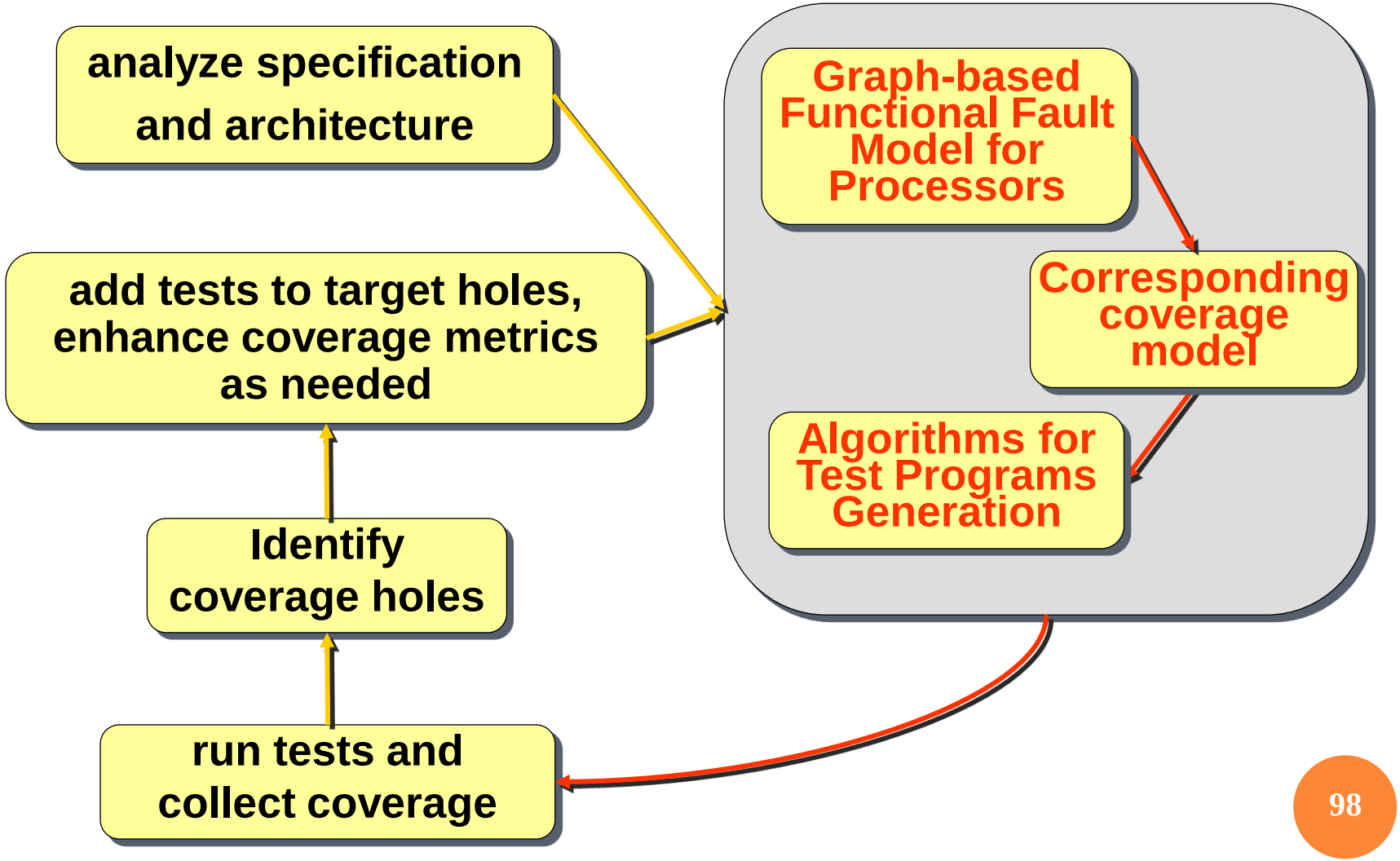
Saxena 00


analyze specification and architecture
add tests to target holes, enhance coverage metrics as needed
Identify coverage holes
run tests and collect coverage
Signal Causality from Formal Specification
Equivalent Coverage model
Rules for Instructions Generation

Saxena 00

- Generates test programs to a GL85 processor
- Using formal specifications (from formal verification), the causality between signals (as described in an assertion) are computed and kept in a lookup table
- From the lookup table, based in rules, a test is generated
- Direct tests are generated
- Domain knowledge approach

Mishra 05


analyze specification and architecture
add tests to target holes, enhance coverage metrics as needed
Identify coverage holes
run tests and collect coverage
Graph-based Functional Fault Model for Processors
Corresponding coverage model
Algorithms for Test Programs Generation

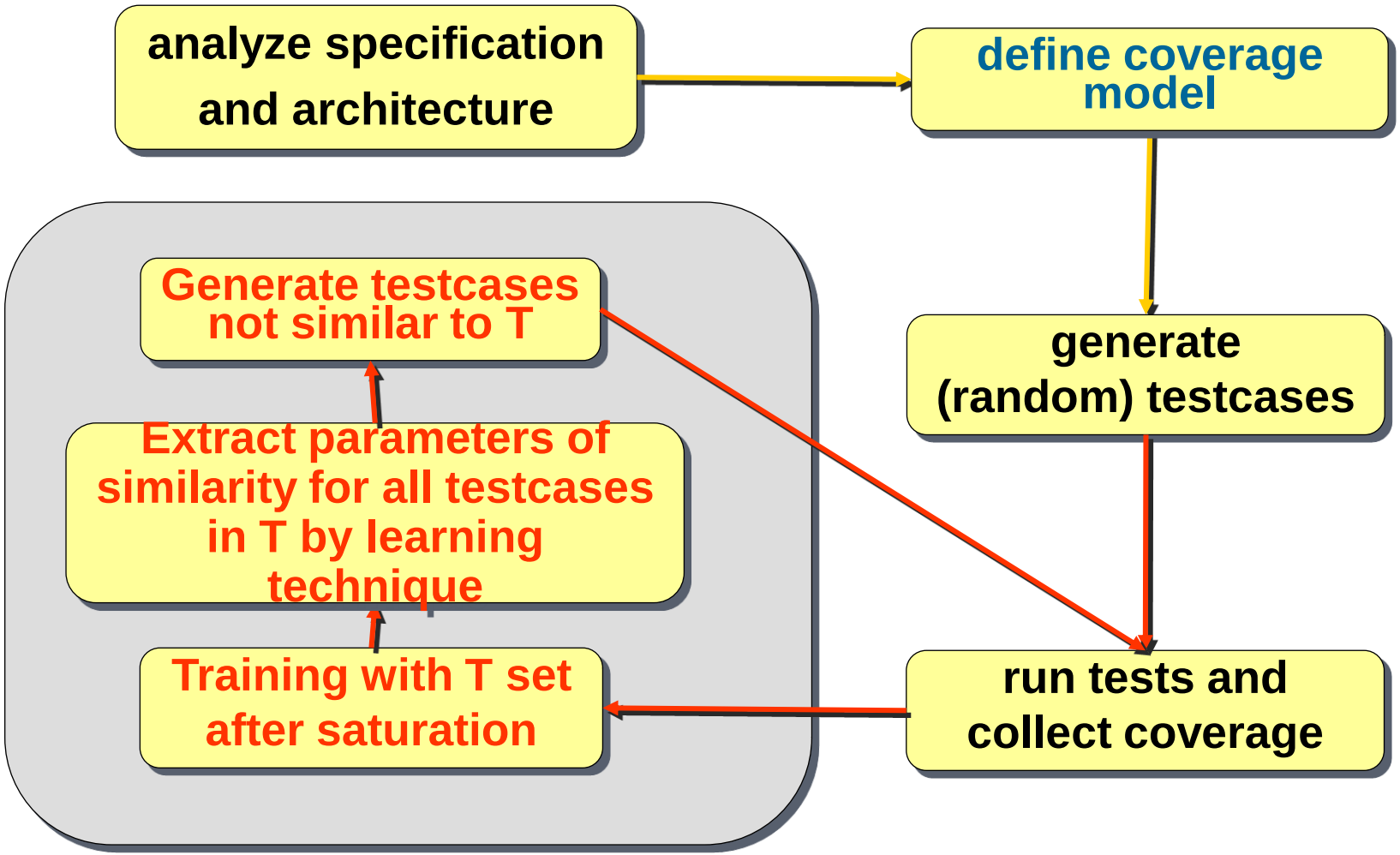
Mishra 05

- Generates test programs to DLX and LEON2 pipeline processors
- A graph-theoretic functional fault model for pipelined processors is proposed, including architecture fault models as in Register Read/Write, Operation Execution, etc.
- Events that may detect such faults are listed and serve as coverage itens (e.g. a register can be written first and read later, to focus a Register Read/Write fault); therefore, number of tests depends on architecture (number o pipeline paths, registers, etc.)
- Random sequences are generated according to the coverage given
- Domain knowledge

Dynamic Coverage-Driven Verification (Coverage Directed Generation)

Input Coverage

Guzey 08


analyze specification and architecture
define coverage model
generate (random) testcases
run tests and collect coverage
Training with T set after saturation
Extract parameters of similarity for all testcases in T by learning technique
Generate testcases not similar to T

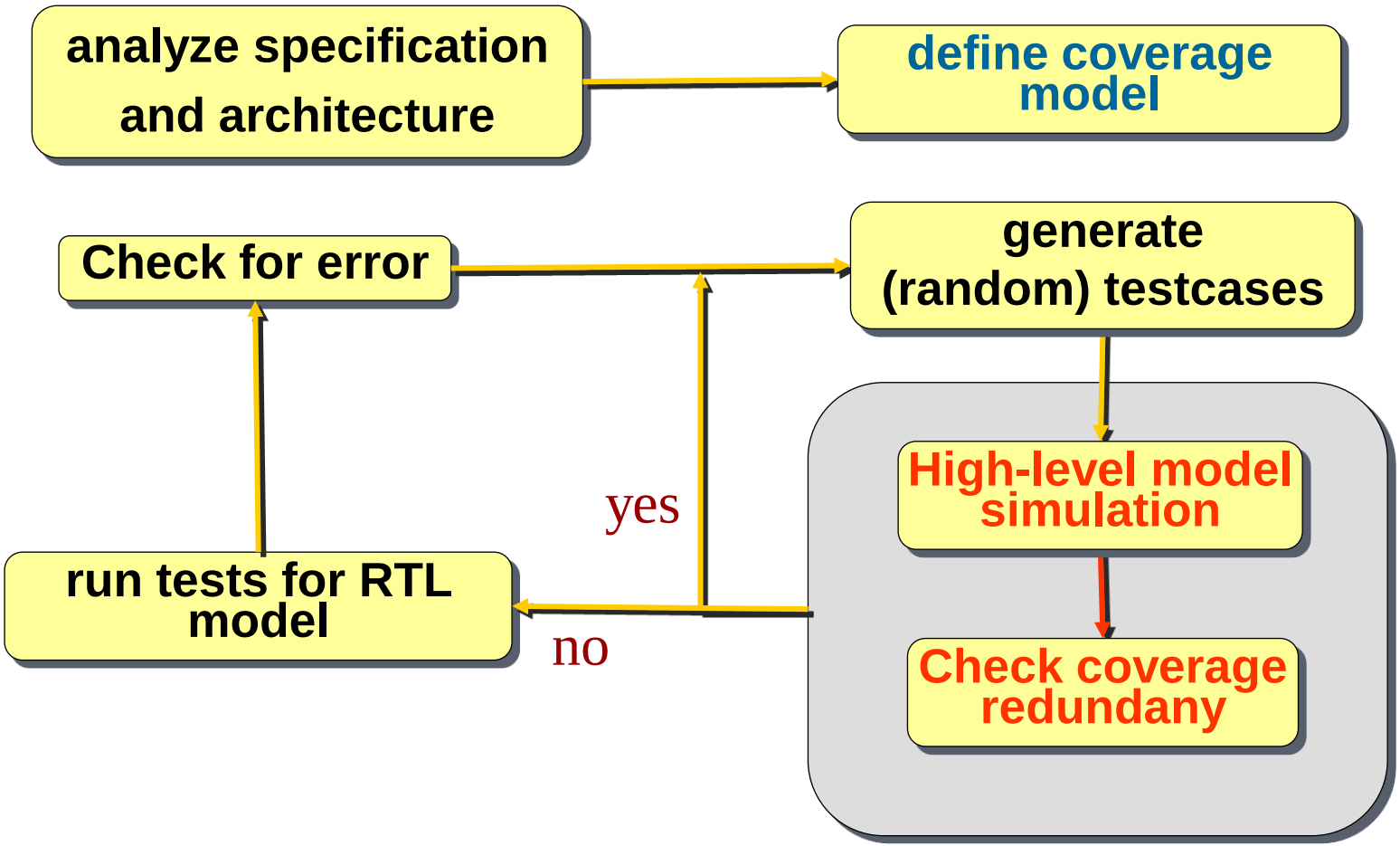
Guzey 08

- Experimented on a processor execution unit
- Monitor the testscases by focusing on the input space
- Learning machine is a support vector machine (for analysis)
- Learn similarity between previous tests, after coverage saturates.
- After the testcase similarity is computed, exclude new testcases similar to the ones referred in training
- Domain knowledge

Dynamic Coverage-Driven Verification (Coverage Directed Generation)

Output/Internal Coverage

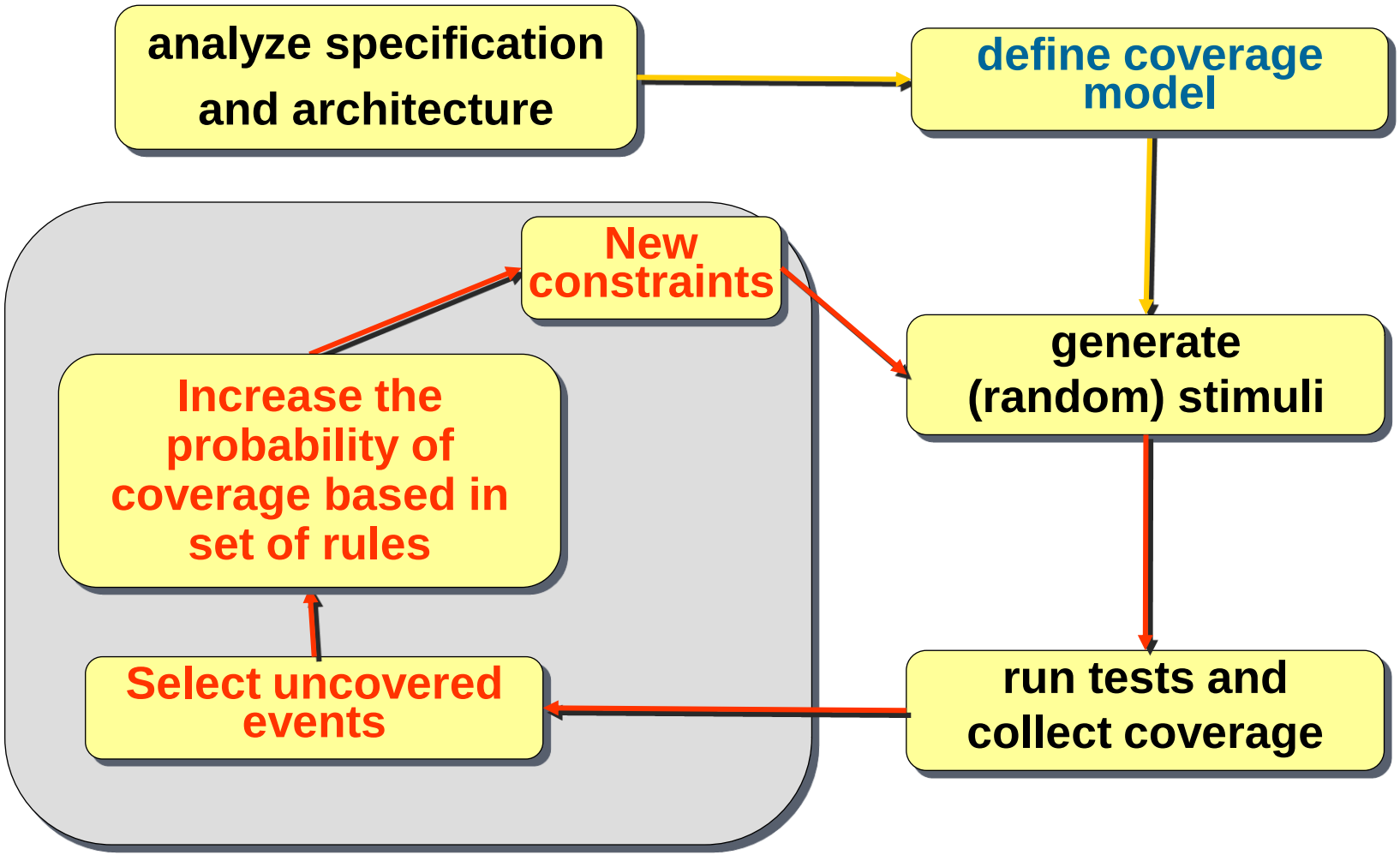
Romero 05


analyze specification and architecture
define coverage model
Check for error
generate (random) testcases
High-level model simulation
Check coverage redundany
yes
no
run tests for RTL model

Romero 05

- Experiments on Bluetooth communication core
- Elimination of redundant coverage itens responsible for inefficiency
- Based on reference model (faster than RTL for evaluation of output values)
- High level reference model checks the coverage; if it does not add, then RTL simulation is discarded
- Black-box approach.

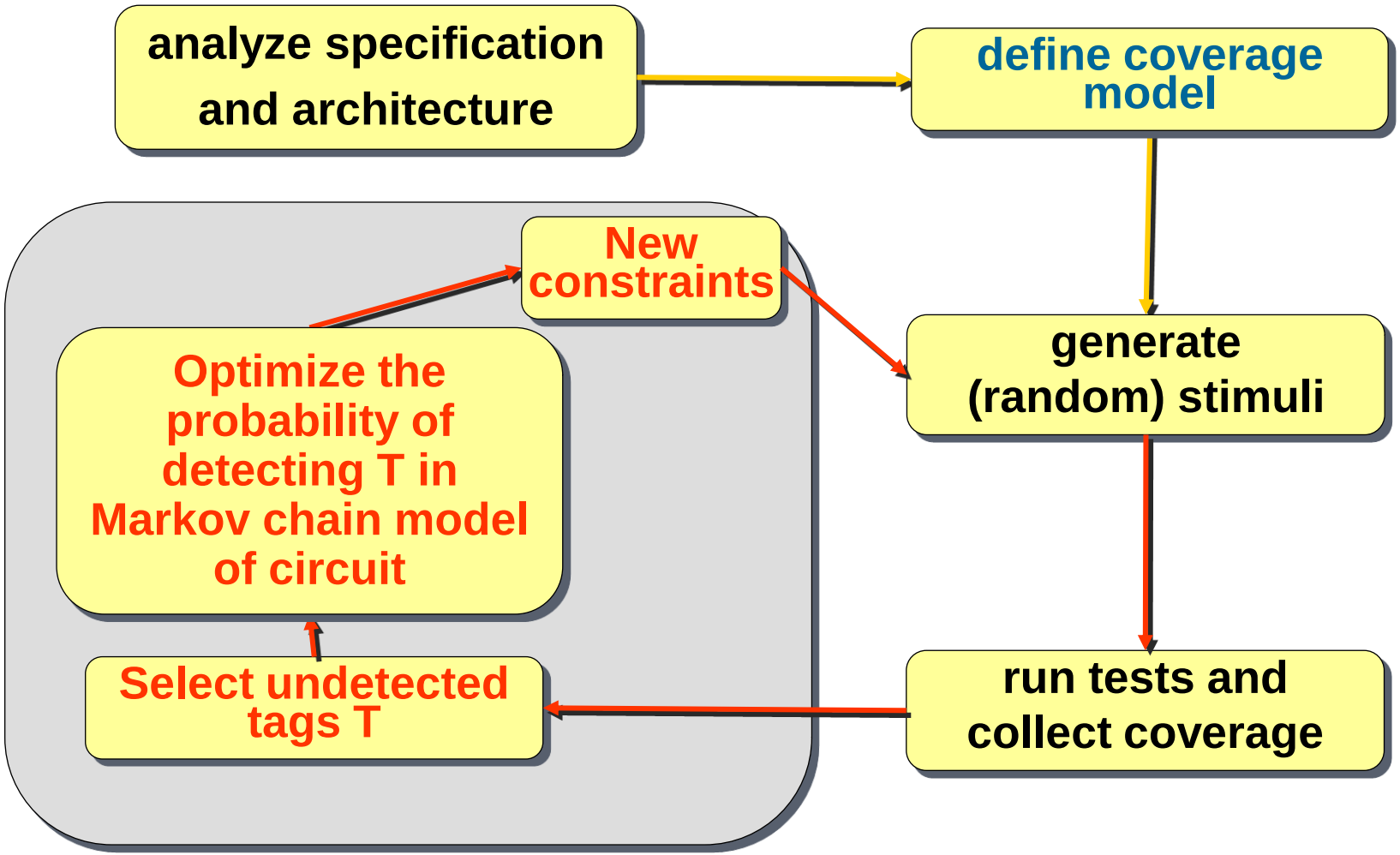
Nativ 01


analyze specification and architecture
define coverage model
New constraints
generate (random) stimuli
Increase the probability of coverage based in set of rules
Select uncovered events
run tests and collect coverage

Nativ 01

- DUV: a coherent switch with a cache, i.e. an interface unit between multi processors and memory in Z-series (S390) servers
- A model of probabilities involved in a command/response behavior is developed, composed by several rules
- Coverage is combined conditions of five atributes as command, the processor identification, etc.
- Holes are detected and rules are used to find contrained random sequences with better cvoverage probability.
- Domain knowledge

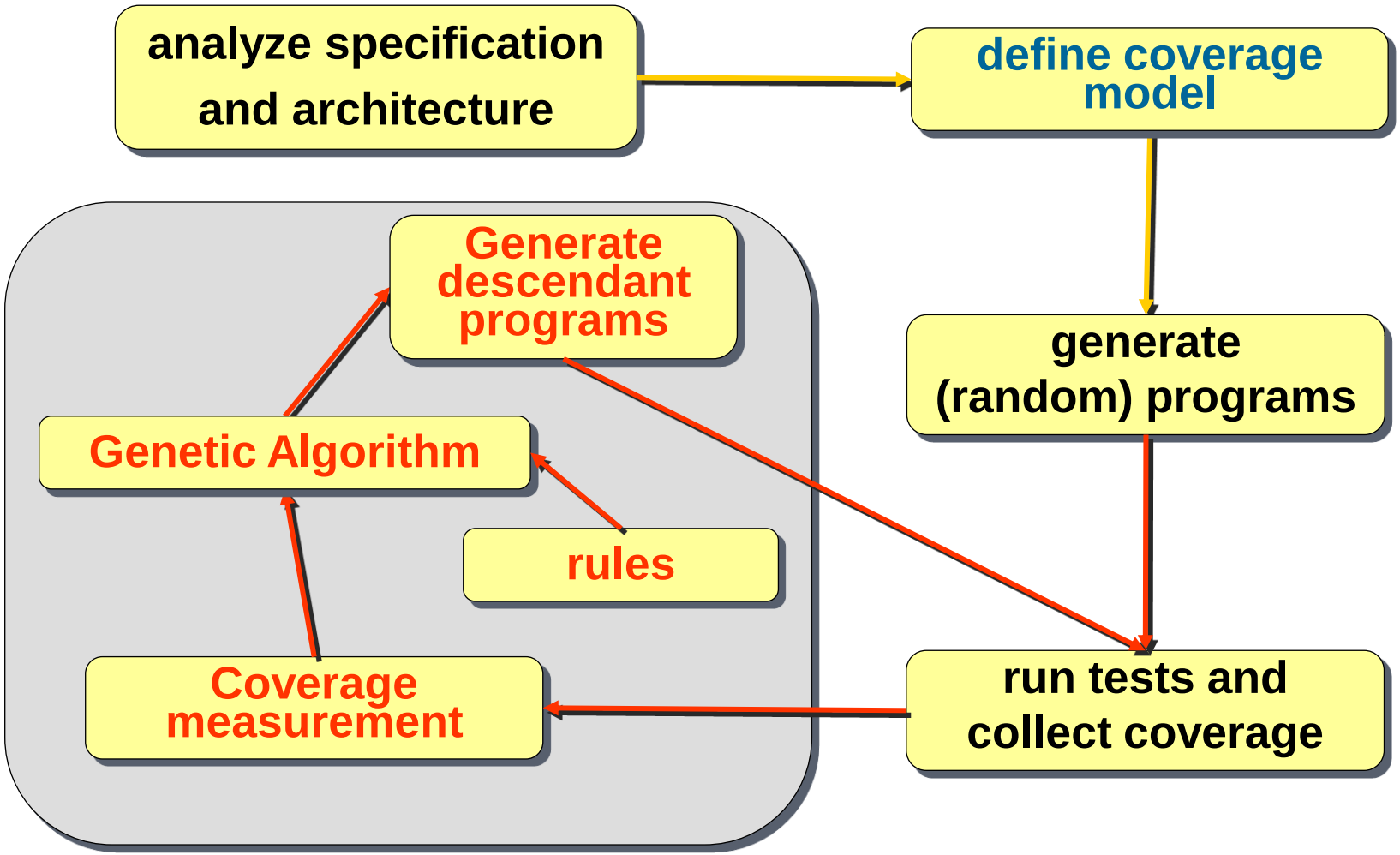
Tasiran 01


analyze specification and architecture
define coverage model
New constraints
generate (random) stimuli
Optimize the probability of detecting T in Markov chain model of circuit
Select undetected tags T
run tests and collect coverage

Tasiran 01

- Experiments on sequential circuits (ISCAS98), and DLX and Intel 8085 processors
- Circuits are modeled as Markov chains (statistical approach) computed by probabilities of a detecting a tag
- Tag is the fault model and also the related coverage item. Tags are faulty signal (and coverage items) which may assume values (expected + Δ).
- Through optimization procedure (dectectability of uncovered tags) applied to the Markov chain, random sequences or test programs are generated, with different constraints.
- Domain knowledge

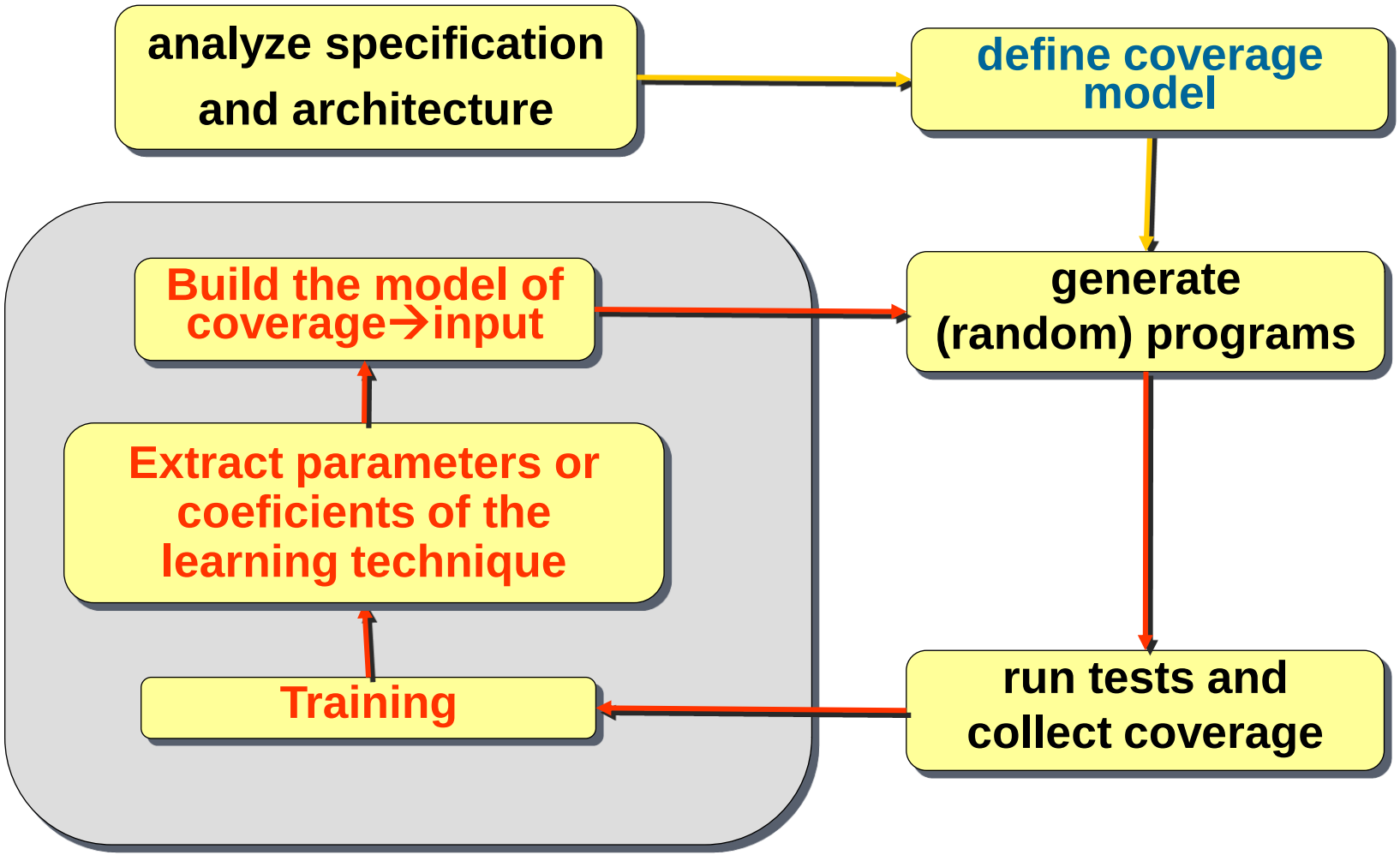
Bose 01, Corno 04



Bose 01, Corno 04

- Test programs for UltraSparc10 and Leon processors
- Test Programs (32 of 10 lines +fixed part)
- Applies test programs randomly generated and then collect results of coverage
- The coverage and fitness function ara give by usage of buffer (full condition is deseired) and number of code lines
- GA gets the programs, results and the fitness function and generates descendants.
- Domain knowledge

Fine 03, Braun 03, Hsueh 06, Romero 09



Fine 03, Braun 03, Hsueh 06, Romero 09

- Mainly generation of test parameters (but test programs are also treated)
- Bayesian Networks, Decision Trees, Inductive Logic Programming (rule construction), Support Vector Machines are different techniques for modeling the relationship between the set of input parameters/constraints and coverage items.
- Based on training with varied coverage objectives: architectural aspects of processor/ parameters
- Training is made through pre-simulated samples
- Domain knowledge only for Bayesian Networks