Testability Issues in Hardware/Software Systems

1. Heterogeneous Systems and their Design Process
2. Design Validation
3. Formal Verification
4. Design Validation Practice
5. Design Validation by Model Execution
6. Hardware/Software Co-design for Testability.

Introduction

A typical embedded system is heterogeneous:

1. Hardware and software components are mixed.

2. Within each of these categories design styles are mixed:
   - control oriented processes can run on a microcontroller and supervise dataflow dominated processes executed on a DSP;
   - hardware components can be ASICs, controllers, ASIPs, memories, etc.

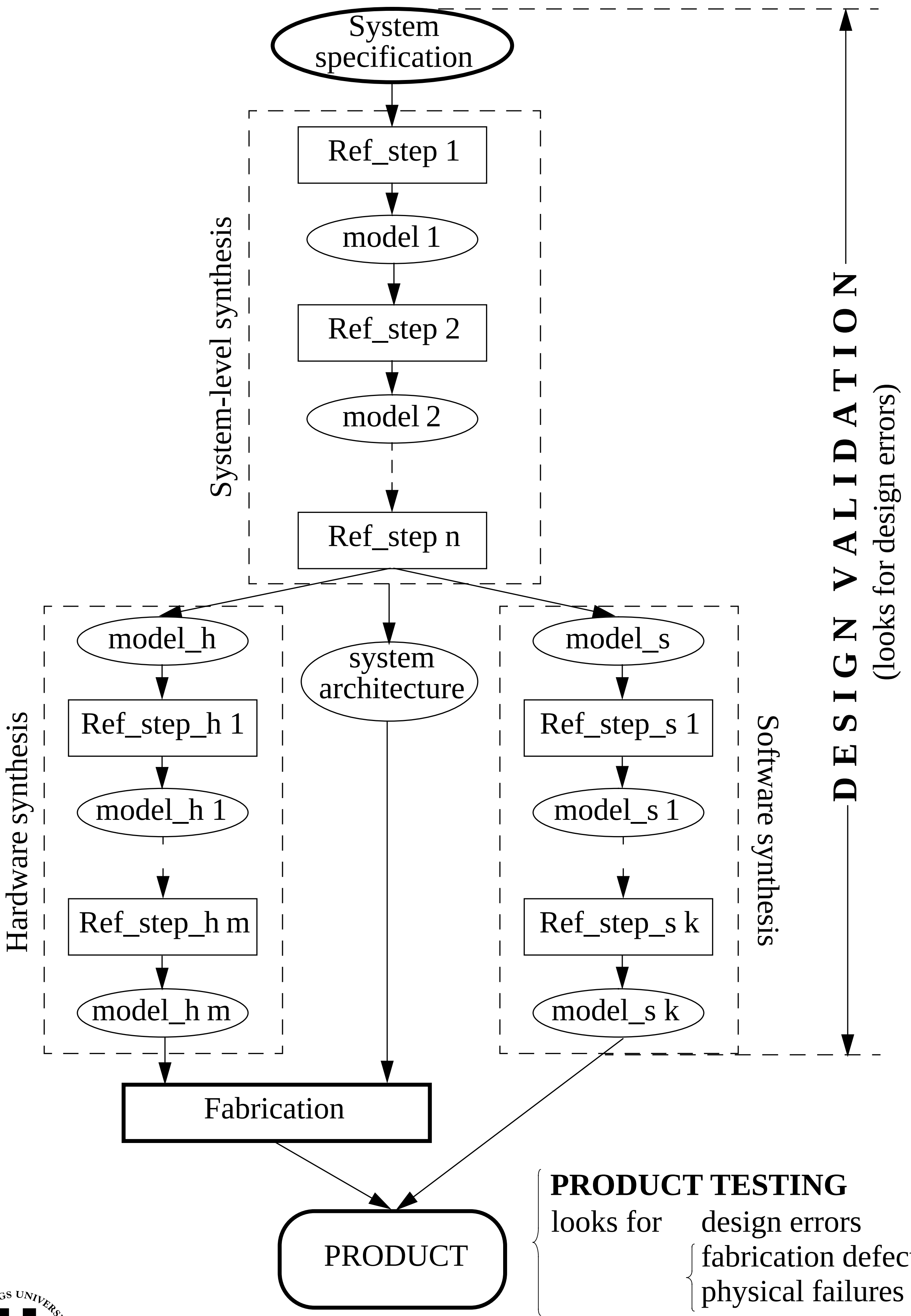
Specification and Refinement

- The design process consists of a sequence of steps: each step performs a transformation from a more abstract description to a more detailed one

- A formal model of a design consists of:
    1. A *functional specification*, as a set of explicit or implicit relations which involve inputs, outputs, and (possibly) internal state information.
    2. A set of *performance constraints* (cost, reliability, speed, size).

- A design step takes a model of the design at a level of abstraction and *refines* it to a lower one.

From Specification to Product



PRODUCT TESTING
looks for design errors
fabrication defects
physical failures

Design Validation

- Validation is the process of determining that *a design* is correct:

1. Model (specification) execution

   correctness is asserted only with respect to the actual *test vectors* provided by the environment during the validation process.

2. Formal verification:

   proves mathematically that a certain property is true for a specification or that a refinement step produces a correct implementation relative to a given model.

Formal Verification

1. *Specification verification*: checks if a model satisfies some abstract property; can be used for the verification of the initial functional specification of the system.
2. *Implementation verification*: checks if a lower level model, resulted after one or several refinement steps, correctly implements a higher-level model.

- The computational complexity of such an analysis is very high.

Formal Verification (cont'd)

Properties of a model, which are of interest

1. *safety properties*: no matter what inputs are given, and no matter how nondeterministic choices are resolved, the system will not get into a specific undesirable configuration (deadlock, undesired outputs, etc.);

2. *liveness properties*: some desired configuration will be reached eventually or infinitely often (a certain response to an input will be given).

Basic Approaches to Formal Verification

1. Theorem proving methods

2. Finite automata-based methods
    - containment checking
    - model checking

Theorem Proving Methods

- They provide an environment that assists the designer in carrying out a formal proof of specification or implementation correctness:
  - checks the correctness of a proof given by the user;
  - performs some steps of the proof automatically.

Finite Automata-Based Methods

- The model is restricted to FSMs in order to apply formal verification methods.
- There is a well-developed theory for analyzing FSM models and checking their equivalence.

Containment checking

- Both the higher-level model (specification) and the lower-level implementation have to be captured as FSMs.
- The system checks that the specification machine is contained in the implementation machine, meaning that they give the same output for all inputs for which the specification is defined.

Finite Automata-Based Methods (cont'd)

Model checking

- The system is modelled as a synchronous or asynchronous composition of automata; the property to be checked is described as a temporal logic formula.
- The proof is carried out by traversing the state space of the FSM and marking the states that satisfy the temporal logic formula.
- The system to be verified is represented as an explicit state transition graph $\Rightarrow$ state explosion problem.
- Certain techniques avoid explicit state enumeration:

  symbolic model checking uses symbolic boolean representa-tions for states and transition functions without building a global state transition graph explicitly; boolean manipulation techniques are used to evaluate the truth of temporal logic formulas with respect to the model.

Some Problems with Formal Verification

- High complexity $\Rightarrow$ only relatively small designs, or parts of a design can be verified.
- Correctness can be proven only relative to certain properties which have to be explicitly formulated $\Rightarrow$ we maybe know that we have a correct implementation but we don’t know if we have implemented the right system.
- Using formal methods needs a strong and very specific theoretical background $\Rightarrow$ difficult to introduce in industrial practice.

$$\Downarrow$$

The design process has to include validation by model execution!

Currently Recommended Practice for Design Validation

(Synopsis&Mentor Graphics)

- Early design stages (before hardware/software partitioning):
  - Validation of input specification
  - Validation for refinement steps during early design phases.

  This is based on execution of implementation independent specifications.

- After hardware/software partitioning
  - Validation of behavioral hardware specification.
  - Validation of prototype software specification.

  This is based on behavioral simulation and software execution.

- Hardware and software development and validation

Currently Recommended Practice for Design Validation (cont'd)

The Methodology implies validation at two levels:

1. <u>Block-level verification</u> (possibly concerns an IP block and then is performed at the supplier);

2. <u>System-level verification</u>

Verification at Block Level

Verification Strategy:

1. Verification of individual subblocks
2. Block verification
3. Prototyping

Goal of the strategy: To achieve a very high level of test coverage at the subblock level and then to focus at the:

- interfaces between subblocks
- overall block functionality
- corner cases of behavior: complex scenarios that are most likely to break the design

Verification at Block Level (cont'd)

- Verification of individual subblocks and block verification is performed by (co-)*simulation*.
- Typically, simulation is performed at behavioral and RT level but has to be performed at gate-level in the later phases of the design and when high accuracy is needed.
- The main problem in this context is *simulation time*;
  it becomes critical when the hardware model has to be simulated with the real software running on it.
- The final goal of simulation is to achieve a high coverage.
- A testbench provides inputs and checks outputs during simulation.

Verification at Block Level (cont’d)

Prototyping:

An actual ASIC chip or FPGA is built for the block.

$\Downarrow$

Real code can be run at a real speed!

Verification at System Level

Verification Strategy:

1. Verify blocks (possibly done at IP supplier)
2. Verify interfaces between blocks (integration testing)
3. Run a set of increasingly complex applications on the full system
4. Prototype the full system and run a full set of application software at full speed for final verification

Verification at System Level (cont’d)

- Steps 1,2 and 3 are performed by simulation.
- Step 2 is a typical integration testing problem.
- In step 3 the problem of simulation speed becomes critical as the whole hardware has to be simulated together with the full software on top of it.
  Such a simulation is used for both hardware testing and testing of the software.
- There are different strategies for speeding up simulation. The main idea is to reduce the level of detail at which the hardware is simulated, while keeping the adequate accuracy of the simulation.

Verification at System Level (cont'd)

Prototyping:

An ASIC or (possibly FPGA) implementation of the system is produced and the full application software is run for testing.

- *This is still design verification*. The intention, at this stage, is not to look for fabrication failures (although, they can be there, which makes the verification more difficult). We test for hardware design and software faults.

  At this phase mainly faults connected to interconnection between components, hardware/software interaction and timing are detected.

Verification at System Level (cont'd)

- After prototyping we suppose that, *to our best knowledge*(?), there are no hardware design and software faults in the system.

$\Downarrow$

When we go to fabrication, we can accept as a model, that the hardware design and the software are correct. Thus, after fabrication, we look only for manufacturing failures (the test vectors we apply have passed at prototyping).

- However, it is obvious that neither software, nor hardware design faults have been eliminated!

$\Downarrow$

For field/maintenance testing possible software and hardware design faults have to be considered!

Design Validation by Model Execution - Design Testing

- The initial specification as well as the models generated as result of refinement steps are, executable and are formulated in a certain formal language (C, VHDL, Statecharts, etc.).

- Validation of such a specification is to a large extent similar to *software testing*.

- We have to provide a set of test vectors until a confidence is gained that no major *design errors* are left undetected.

Design Validation by Model Execution - Design Testing (cont'd)

1. Levels of Testing
2. Functional vs. Structural Testing
3. Kinds of Faults
4. Testing Methods
5. Testability Metrics

Levels of Testing

Testing at three distinct levels:

1. Unit/component testing: testing of individual modules or groups of modules with related functionality. The goal is to show that the module does not satisfy a certain expected beahviour;
2. Integration testing: a set of interacting modules are tested. The goal is to show that even though the components were individually satisfactory, the combination of components are incorrect.
3. System testing: the final system is tested. The goal is to reveal bugs that cannot be identified by testing individual components or by integration testing. It concerns issues that can be tested only after the *entire system* is integrated on the target architecture: performance, communication aspects, protocols, time constraints, fault tolerance and recovery, etc.

Functional vs. Structural

- Functional testing: the system is treated as a black box. Only functionality and features are of concern and no implementation details are known.

- Structural testing: program structure/coding are considered for testing.

The boundary is fuzzy: it depends on the abstraction level, what we consider at the level of structural details and what as a functional “black box”.

Kinds of Faults

Unfortunately there are no nice and simple explicit fault models.

- Requirements and specifications faults: the earliest to invade the system and the last to be caught (maybe only after system testing).
- Structural faults
  - control faults: related to the order of executing sequences of operations, the conditions under which operations are executed, number of iterations, etc.
  - logic faults: related to use of boolean operators and the way they are evaluated (they often result in control faults);
  - processing faults: arithmetic faults, evaluation of mathematical functions, conversions, etc.
  - initialization faults.

Kinds of Faults (cont'd)

- Data faults: bugs related to definition of data objects, their formats and initial values.
- Coding faults: many are caught by compilers.
- Interface, integration, and system faults: external/internal interfaces, protocols, formats, hardware related problems, operating system related problems, aspects related to the load of the system (stress), timing aspects, deadlock, etc.

Testing Methods

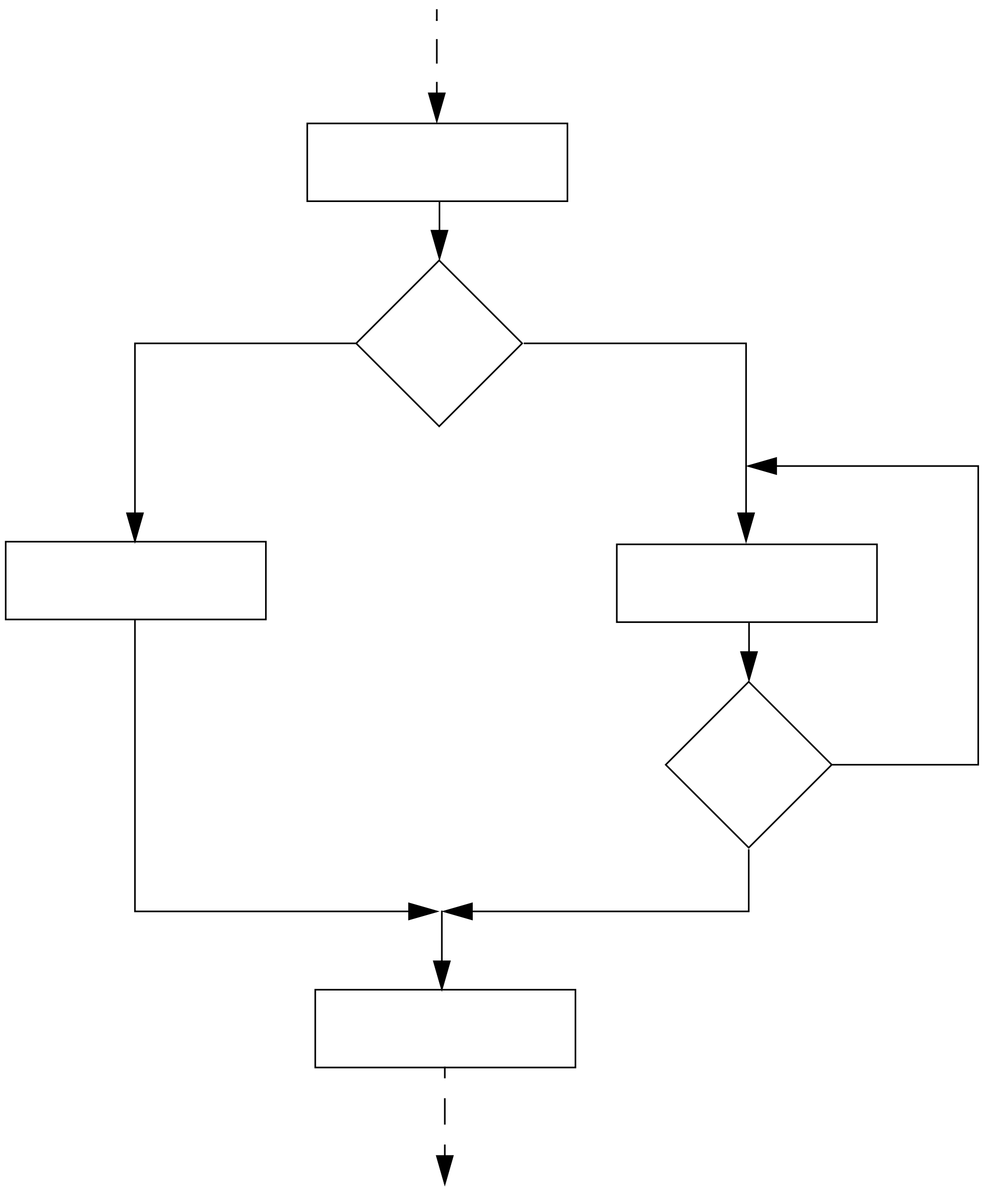
1. Path testing
2. Transaction flow testing
3. Data-flow testing
4. Transition testing
5. Mutation testing

Path Testing

- Is a test technique based on selecting a set of test paths through the program.
- Selecting the right set of paths $\Rightarrow$ achieve some test thoroughness.
- Path testing is basically a component testing method. However, techniques very much related to path testing are used for functional testing of systems.

Path Testing (cont'd)

- The typical representation for path testing is the control flowgraph.



Path Testing (cont'd)

Path testing strategies:

- Path testing (P_∞): execute all possible control flow paths through the program. If achieved $\Rightarrow$ *complete path coverage*; generally it is impossible to achieve.

- Statement testing (P_1): Execute all statements in the program at least once. If achieved $\Rightarrow$ *complete statement coverage.*

- Branch testing (P_2): Execute enough tests to assure that every branch alternative has been exercised at least once. If achieved $\Rightarrow$ *complete branch coverage.*

Path Testing (cont'd)

P_∞ is the strongest; it includes both P_1 and P_2.

For structured specifications $P_2 \supset P_1$

- Statement coverage is established as a minimum testing requirement by IEEE and ANSI standards.
- Statement and branch coverage are minimum mandatory test requirements for new code at IBM.

Path Testing (cont'd)

<u>Problem</u>

Path sensitization: find the set of inputs so that you cover the selected paths.

```
select the paths so that you achieve coverage;
for each path
    trace the path and compose the path predicate; express it in
    terms of inputs.
solve the set of inequalities ⇒ set of inputs for the path.
```

For the general case this is not achievable in practice.

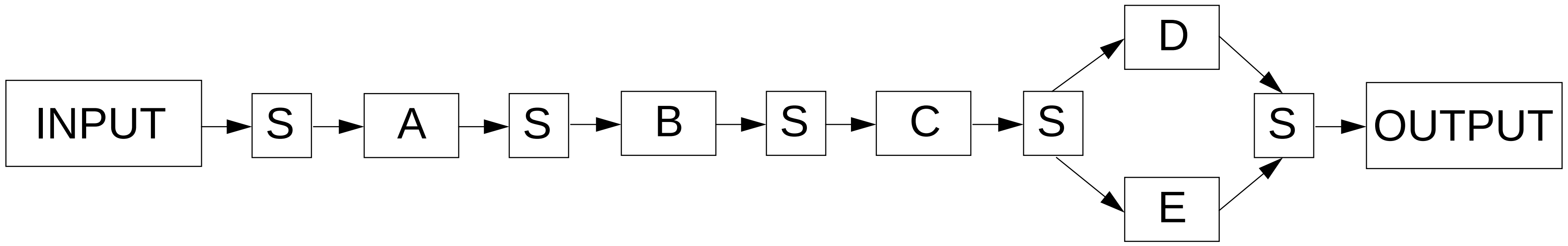
There are tools which can generate test cases with good coverage, based on static analysis, symbolic execution and heuristics.

Transaction Flow Testing

- A transaction is a unit of work, seen from the system user’s point of view; performing a transaction results in a sequence of operations.
- The system as a whole is specified as a set of transactions it can process.
- Transaction flow testing is a *typical functional testing approach* and is used for system testing.

Transaction Flow Testing (cont'd)

- A transaction can be represented as a *transaction flowgraph*. It consists of *a set of tasks* and conditional/unconditional branches representing the sequence in which they are executed.
- The transaction flowgraph is a pictorial representation of a functionality; it does not represent the control structure corresponding to a particular program that implements the functionality.



Transaction Flow Testing (cont'd)

- Transaction flow testing is based on path selection using the transaction flowgraphs. Technically, this is similar to path testing.

ATTENTION!

With transaction flow testing we use input vectors to activate paths corresponding to a functional specification and not to a structural representation which corresponds to a given implementation! This is a typical way to use structural methods for functional testing.

Data Flow Testing

- Data flow testing uses the control flowgraph to select particular paths to be exercised, in order to explore “unreasonable things” that can happen to data.

- Data flow testing introduces additional paths to cover a part of the gap between *path coverage* and *statement and branch coverage*.

Data Flow Testing (cont'd)

- Typical data flow anomaly: use of a variable without previous initialization (definition).

- Possible coverage strategies:
  - exercise *every path* from every definition of a variable *to every use* of that definition;
  - exercise *at least one path* from every definition of a variable *to every use* of that definition;
  - exercise paths so that every definition of a variable is covered by *at least one* use of that definition.
  - - -

Transition Testing

- Transition testing is used when the specification is a *Finite State Machine* model.
- The system is captured as a state graph, where nodes are states and paths represent transitions.
- The testing strategy is similar to path-testing with flowgraphs: the goal is to achieve a coverage of paths through the state graph.

Mutation Testing

- Mutation testing is a *fault -based approach*: design fault-specific tests, tests that would detect the faults if they are present.
- In mutation testing, faulty versions, called *mutants*, are created by infecting the original program.
  Each mutant contains one fault generated by applying a *mutation operator* to a certain statement.
- Test generation is guided by the goal to distinguish the original program from its mutants.

Mutation Testing (cont'd)

How do we generate mutants?

```
Function Max(integer: m,n): integer;
begin
    Max := m;
    Max := ABS(m);                RC: true     NC: m<0
    if (n>m) then
    if (n<m) then                 RC: true     NC: (n>m) ≠ (n<m)
    if (n>=m) then                RC: true     NC: (n>m) ≠ (n≥m)
        Max := n;
        Max := m;                 RC: n>m      NC: n≠m
end;
```

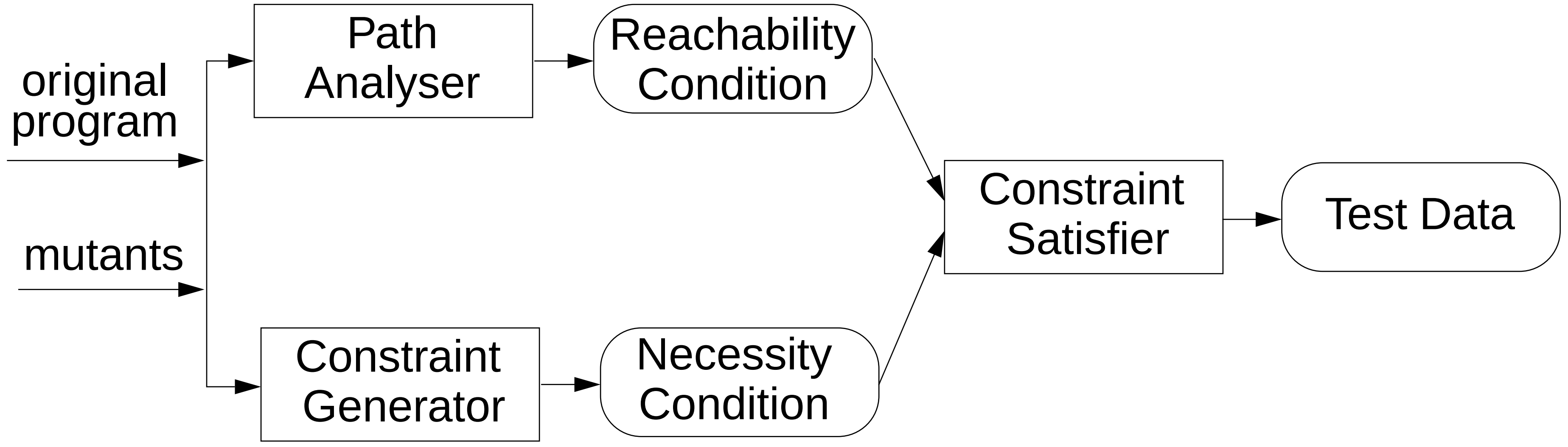
Mutation Testing (cont'd)

The two basic hypotheses:

1. Coupling effect: complex faults are coupled with simple faults in such a way that a test set that detects all simple faults will also detect most complex faults.

2. Competent programmer: a competent programmer tends to write programs that are close to being correct $\Rightarrow$ they may be incorrect, but differ from the correct one by relatively simple faults.

Mutation Testing (cont'd)



- *Necessity Condition* (NC): necessary condition for the mutant to behave differently from the original program (the mutant to be killed).
- *Reachability Condition*: necessary condition for the respective instruction to be executed.

Mutation Testing (cont’d)

Once the test set is generated:

- Run original program and update if needed, until it behaves correctly for each test.
- Run mutants for each test vectors and *kill* if they behave differently from original program.

Mutation Testing (cont’d)

What about mutants which have not been killed?

- They are functionally equivalent to the original programme (see 3d mutant in example).
- They have induced a fault, but, with the given test set, it does not propagate to the output (is not *observable*) $\Rightarrow$ new tests are needed.

A Fault Model for Hardware Design Testing

(Ferrandi et al, Politecnico di Milano, ‘99)

- The authors propose a methodology very similar to mutation testing. It is applied to hardware specifications in VHDL

  It is based on one single fault model: bit failures.
  - Each variable, signal or port is considered as a vector of bits; each bit can be stuck-at zero or one.
  - Each condition can be stuck-at true or stuck-at false.

- Test vectors seem to be better than those generated with statement coverage.

- Test vectors generated from behavioural specifications can be successfully used to test the RT level description.

Testability Metrics

Questions they can answer:

- How many bugs can we expect;
- How difficult is it to detect bugs?
- How large is the test set to be applied?
- When can we stop testing?
- How should we change the specification (design) in order to improve testability?

Testability Metrics (cont'd)

1. Complexity metrics
2. Extending complexity metrics to large OO systems
3. Observability and the Domain/Range Ratio
4. A Dynamic Testability Metric

Complexity Metrics

The more complex the program (design specification), the more likely that errors are introduced and hence the more testing will be required

- Primitive metrics: lines of code, statement count, etc.

Complexity Metrics (cont'd)

Halstead's metrics

N_1: number of operators in the program

N_2: number of operands in the program

Halstead length: $N = N_1 + N_2$

n_1: number of *distinct* operators in the program

n_2: number of *distinct* operands in the program

Predicted Halstead length: $H = n_1 \times \log_2 n_1 + n_2 \times \log_2 n_2$

Extremely interesting: The effective length (*N*) can be predicted based on the specification, before the program has been written!

Predicted number of bugs: $B = \frac{(N_1 + N_2) \times \log_2(n_1 + n_2)}{3000}$

Complexity Metrics (cont’d)

Structural metrics: *McCabe’s cyclomatic complexity metric*

- McCabe’s metric is not based on program size but more on information/control flow. It is based on the specification’s flow-graph representation:

  L: number of links in the graph
  N: number of nodes in the graph
  P: the number of disconnected parts of the graph (calling program and subroutines)

  $M = L - N + 2P$

Complexity Metrics (cont’d)

- Correlation studies show that McCabe’s metric can be used as a guideline for the number of required test cases.
- A very good measure of fault risk is given by the correlation between McCabe’s complexity and size (number of executable statements):
    - Modules with high complexity for a relatively small number of executable statements present the highest risk.

Complexity Metrics for Large OO Systems

NASA Program for Risk Assessment

- Internal object complexity:
  - individual methods: McCabe’s complexity correlated with size;
  - for object classes: sum of method complexities;
- Complexity related to object interaction:
  - number of methods that can be invoked in response to received messages;
  - number of other classes to which a class is coupled.

A Metric for Testability: The DRR

- The Domain Range Ratio (DRR): the DRR of a unit (module, function, operation) is the ratio between the cardinality of the domain (input) to the cardinality of the range (output).
- A high value of DRR indicates a high potential of the module to hide faults $\Longrightarrow$ low testability.

```
function F (in integer X, Y; out boolean B);
begin
   --------------------
   if C then A:=(X+Y)*K1 else A:=(X+Y)*K2;
   B:= A>1000;
end;
```

DRR(F) = ∞ : 2

A Metric for Testability: DRR (cont'd)

- A high DRR indicates modules of the program that are less likely to expose existing faults (faults are hidden because of *low observability*)

$\Downarrow$

This modules have to be tested intensively

OR

Testability has to be improved by

1. adding new output parameters $\Rightarrow$ increase observability
2. introducing ASSERTIONS $\Rightarrow$ BIST

A Metric for Testability: DRR (cont'd)

```
function F(in integer X,Y;out boolean B;out integer A);
begin
    --------------------
    if C then A:=(X+Y)*K1 else A:=(X+Y)*K2;
    B:= A>1000;
end;
```

....

```
function F (in integer X, Y; out boolean B);
begin
    --------------------
    if C then A:=(X+Y)*K1 else A:=(X+Y)*K2;
    ASSERT(cond(A), “erroneous state in function F”);
    B:= A>1000;
end;
```

A Dynamic Testability Metric

- All the metrics discussed so far are *static*: they are based on static analysis of the specification.
- *PIE* (propagation, infection, execution) is a *dynamic* technique, based on program execution in order to derive *statistical estimates of testability*.

Testability:

The probability that the next execution will fail during testing (with a particular assumed input distribution) if there exists a fault.

- testing reveals faults;
- testability suggests locations where faults can hide from testing (the portions which should be redesigned or where testing effort should be concentrated).

Dynamic Testability Metric (cont'd)

PIE analysis is based on repeated execution with a certain input distribution; it derives three probability estimates for each location *l*:

1. Execution estimate: estimate of the probability that program location *l* is executed;
2. Infection estimate: estimate of the probability, that given the program location *l* is executed, a mutant M_l adversely affects the data state.
3. Propagation estimate: estimate of the probability, that given that a variable in the data state following location l changes, the program output that results also changes.

Dynamic Testability Metric (cont'd)

- Based on the PIE estimates, the global sensitivity of a location can be predicted.
- *Global sensitivity* is an expression of the degree to which testing must be performed in order to be convinced that the location is probably not protecting a fault from detection.

Hardware/Software Co-Design for Testability

- Tackling the testability problem in an early phase of system design has a major impact on the global test cost.

Research efforts have been concentrated into two directions:

1. Developing tools for hardware/software partitioning with the specific goal to improve testability of the final system.

2. Developing a design methodology and guidelines so that a high degree of system testability is achieved.

Hardware/Software Partitioning Based on Testability Metrics

(LSR-IMAG Grenoble)

Application of *mutation testing* for hardware test:

- Test vectors generated with mutation testing, based on the behavioral VHDL specification, are used for design validation.
- The same test vectors + a set of additional vectors (depending on the dimension of operands) are successfully used for test of the final hardware; test coverage is similar to that obtained with traditional ATPG tools, applied on lower levels of the design.

⇓

A very efficient way to generate tests from the behavioral specification.

Hardware/Software Partitioning Based on Testability Metrics (cont’d)

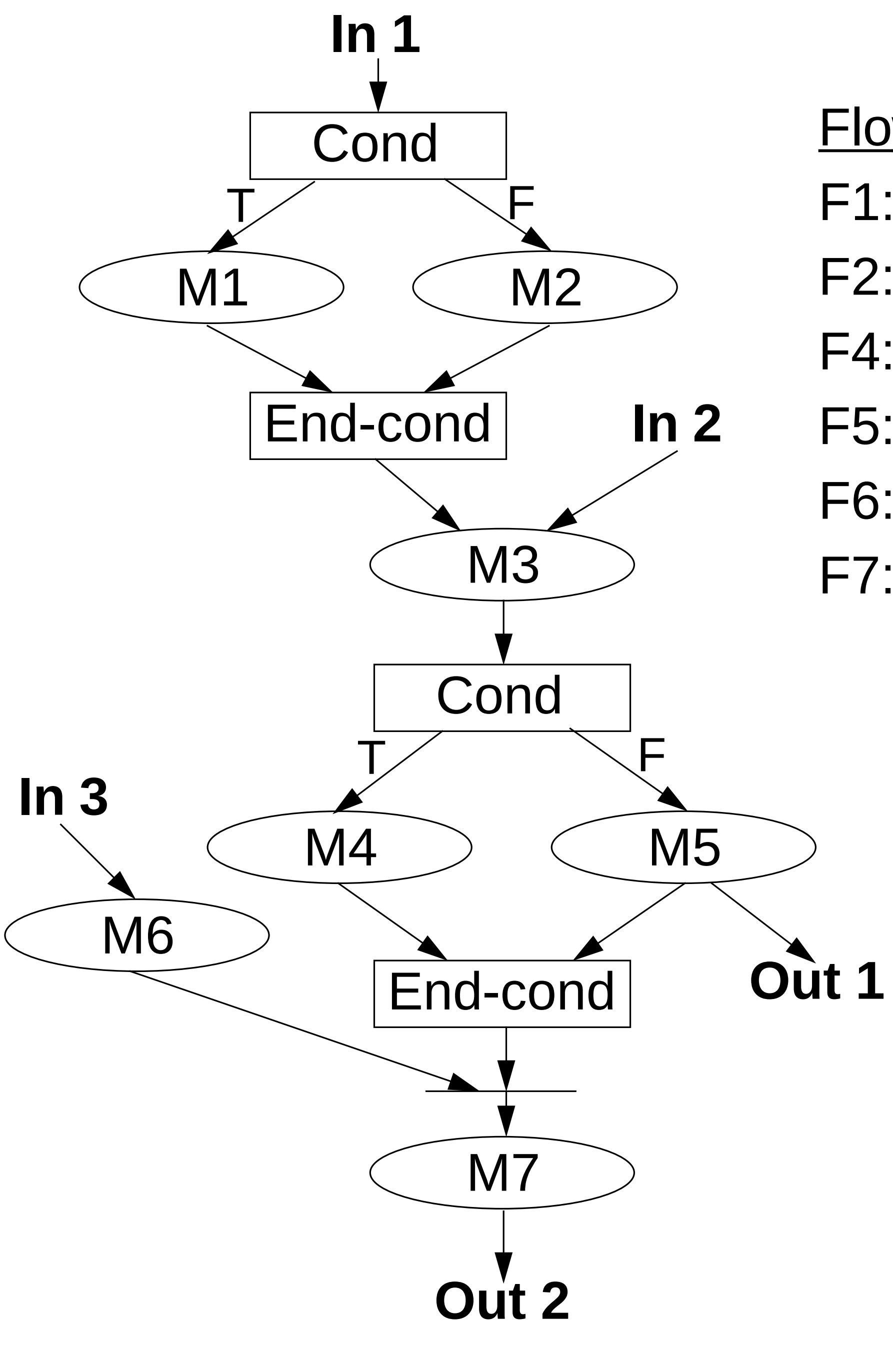
- Find a metric which combines software and hardware testability.

$$\Downarrow$$

Test cost: the number of cases needed to test the system

- Total system test cost is evaluated based on test cost for system components.

Hardware/Software Partitioning Based on Testability Metrics (cont'd)



Flows:

F1: M1,M3,M4,M6,M7

F2: M2,M3,M4,M6,M7

F4: M1,M3,M5,M6,M7

F5: M2,M3,M5,M6,M7

F6: M1,M3,M5

F7: M2,M3,M5

- The system representation is a *control-data flowgraph* (it can be interpreted similar to a transaction flowgraph).
- We can identify several flows (transactions).

Hardware/Software Partitioning Based on Testability Metrics (cont'd)

- Several test strategies can be imagined: test all flaws, test flaws until all modules have been covered, etc.

- A particular strategy has to be adopted, based on the system specification; it implies the activation of certain flows F_i, $i = 1..n$.

  The test cost for each flow:

  $$T_{Fi} = \sum_{m_j \in F_i} N_{mj};$$

  where N_{mj} is the test cost for module m_j ($m_j \in F_i$);

  The system test cost:

  $$C = \sum_{i=1}^{n} T_{Fi}$$

Hardware/Software Partitioning Based on Testability Metrics (cont'd)

- The test cost N_{mj} for a particular module m_j is estimated as the number of test vectors generated, using *mutation testing* (this number differs depending on the module's implementation in hardware or software).

- Using the cost C as an objective function, a hardware/software partitioning can be generated, which is optimal from the testability point of view.

Hardware/Software Partitioning Based on Testability Metrics (cont'd)

What is good with this approach:

- It tries to define a metric for system testability starting with an implementation independent system specification.
- It uses this metric in order to guide hardware/software partitioning.

Problems with this approach:

- System test cost is more than simply the sum of test costs for the components.
- There is no differentiation between applying *N* test vectors for software test and performing hardware test with the same number of test vectors.
- Before manufacturing, hardware design has to be tested also, exactly like software.
- What about the hardware on which the software is executed?
- Observability/controllability is not considered.

Design for Test/Debug in Hardware/Software Systems

(TU Eindhoven)

An incremental approach to testing: each hardware and software component is tested first in isolation; when the component passes the test, it can be integrated into the system.

System level testing is the task of verifying the correctness of the system by applying test stimuli to the hardware/software implementation of the system and observing the responses.

⇓

We verify the correctness of the system as a whole, constituted of all its hardware and software components.
System testing aims at detecting design and fabrication/physical failures that are revealed by the complex interactions between many hardware and software components.

Design for Test/Debug in Hardware/Software Systems (cont’d)

- Testing and debugging a system through its external interfaces does usually not provide sufficient control and observation of the internal system operation.

⇓

The approach primarily concentrates on design for test&debug by *improving visibility* into communication interfaces and the state information of software processes and hardware components. This is achieved by inserting *Points of Control and Observation* (PCOs).

Design for Test/Debug in Hardware/Software Systems (cont'd)

- PCOs are inserted into the implementation independent system specification, based on initial test requirements.
- During successive synthesis steps PCOs are implemented in hardware and software.
- Software implementation usually implies additional methods and input/outputs of the objects.
- Hardware implementation:
  - synthesis of scan and BIST facilities;
  - using existing DFT/DFD facilities of microprocessors or IP cores.

Summary

- Embedded systems typically are heterogeneous hardware/software systems.
- Design validation and product testing are aimed to guarantee correct functionality and efficient maintenance.
- Design validation can be performed by formal verification and model execution.
- Formal verification can mathematically demonstrate certain features of the models or equivalence between two models.
- Design practice includes validation by model execution and prototyping.
- Model execution implies practically the testing of a software system. A set of test vectors have to be provided until a certain degree of confidence is gained that no major design errors are left undetected.

Summary (cont’d)

- Several approaches to model testing have been studied in order to efficiently achieve a high degree of test coverage. These techniques are related to hardware testing techniques and some of them can be directly used for hardware and hardware/software testing.
- Testability metrics provide a measurement of how difficult it is to test a certain system.
- Applied to a system specification, some of these metrics provide a measure of the overall testability (regardless of later implementation decisions)
- System testability metrics can be used in order to guide system-level design for testability; hardware/software partitioning is a good example.