Embedded Systems

Verification & Validation

Validation vs Verification

Validation is the process of checking whether the specification captures the customer's requirements, while verification is the process of checking that the software meets specifications.

Bugs are costly

- Pentium bug
  - Intel Pentium chip, released in 1994 produced error in floating point division
  - Cost : $475 million
- ARIANE Failure
  - In December 1996, the Ariane 5 rocket exploded 40 seconds after take off . A software components threw an exception
  - Cost : $400 million payload.
- Therac-25 Accident :
  - A software failure caused wrong dosages of x-rays.
  - Cost: Human Loss.

Importance of Embedded System Validation

- **Validation is a bottleneck in the design process**
  - High cost of design debug (designers, time-to-market)
  - High cost of faulty designs (loss of life, product recall)
- **Hardware/Software covalidation problem is more acute**
  - Hardware and software are often used together
  - Hardware and software are designed separately
  - Covalidation is performed late in the process, necessitating long redesign loops

What is Verification?

- Process of ensuring system behaves as intended
- System is bug-free
- Functional Correctness
- Important Step
- Nearly 70% of development time

Verification Scope

- Complex computations hidden behind a simple interface
- Ex. Cell phone, digital camera, automobile, etc.
- Design requirements are varied
  1. Power, performance, cost, life- critical
- Design components are varied
  1. Digital/analog hardware
  2. Systems/application software
  3. Mechanical sensors/actuators

Classification :VV & Test

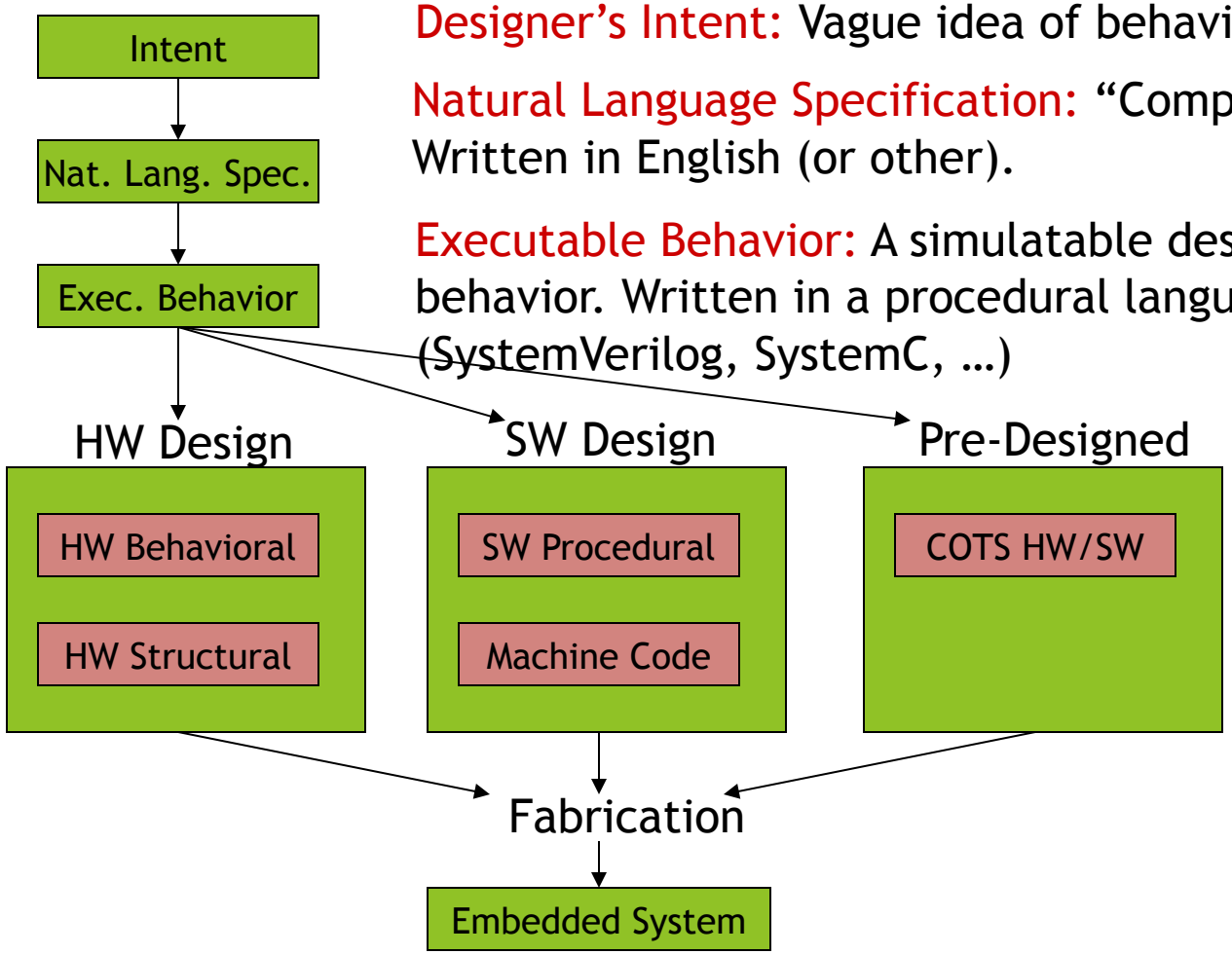
- Validation/Verification
  - Ensuring that the **design** matches the designer's intent/specification
  - Detection of design errors accidentally made by the designer(s)
  - Usually includes the debugging task as well

- (Manufacturing) Test
  - Detection of physical defects in the implementation
  - Defects appear as a result of manufacturing or of wear mechanisms
  - Tests the implementation, not the design

Verification and Validations

- **Formal verification** - Use of proof-based techniques
  - Model checking, equivalence checking
  - Time complexity is high
  - Confidence is high for specified properties
- **Simulation based validation** - Use of simulation
  - Full-chip/full-design validation
  - Confidence is not well quantified
    - **Inspection , and traditional walkthrough verification**

Embedded System Design Flow

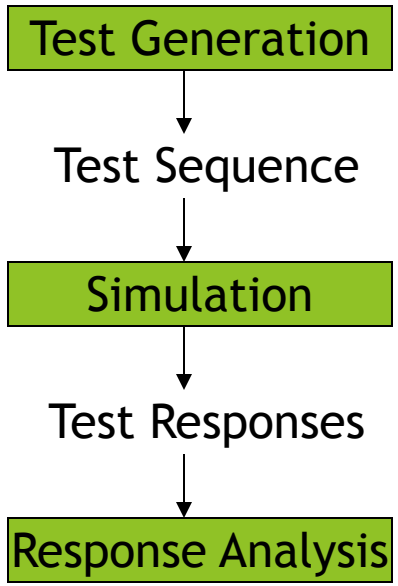


Designer's Intent: Vague idea of behavior known only to designer.

Natural Language Specification: "Complete" behavioral description Written in English (or other).

Executable Behavior: A simulatable description of the behavior. Written in a procedural language (SystemVerilog, SystemC, …)

Elements of Validation



- Writing a good Test Bench
- Evaluating a Test Bench - Coverage Metrics

- Using the simulator (vcs)

- Manual evaluation using the debugger
  - Viewing waveforms
  - -Insertion of breakpoints
- -Automatic evaluation
  - -Comparison with known-good results
  - -Assertions
  - -Self-checking code

Analysis and Validation

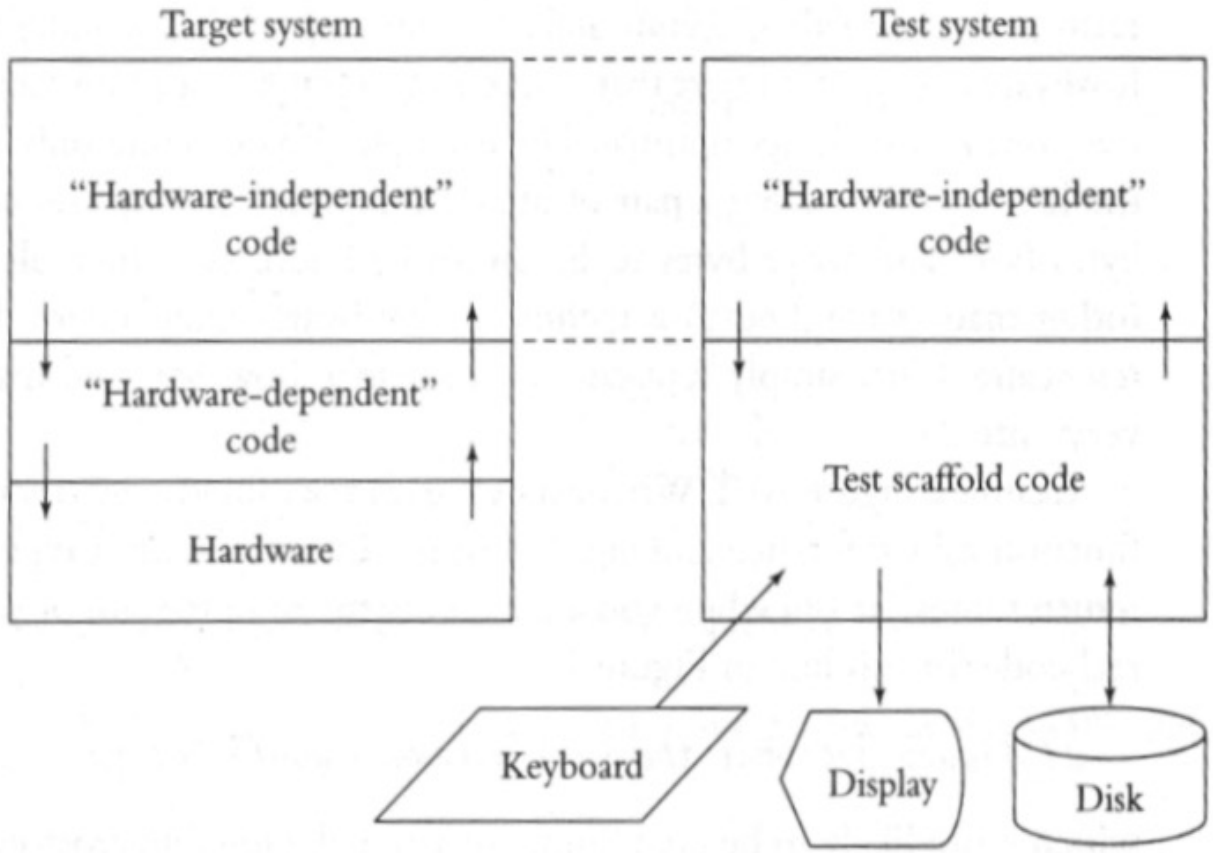
- Functional validation of design models at each step using simulation or formal verification
- Analysis to estimate quality metrics and make design decisions
- Tools
  - Static analyzer - program, ASIC metrics
  - Simulator - functional, cycle-based, discrete-event
  - Debugger - access to state of behaviors
  - Profiler - dynamic execution information
  - Visualizer - graphical displays of state, data

Testing On Your Host Machine

Most often it is best to keep as much testing as possible on the host machine because:

- the embedded environment is not stable (a.k.a. the hardware is “buggy”).
- a richer set of debugging tools and capabilities exist on the host system (i.e. file system, real-time debugger, simulator, terminal, user I/O, etc.).
- it is easier to create reproducible test procedures and simulate “hazard” situations.
- the main algorithms and flow control can be tested without dependencies on the underlying hardware.

Figure 10.1 Test System



Testing/Debugging Techniques

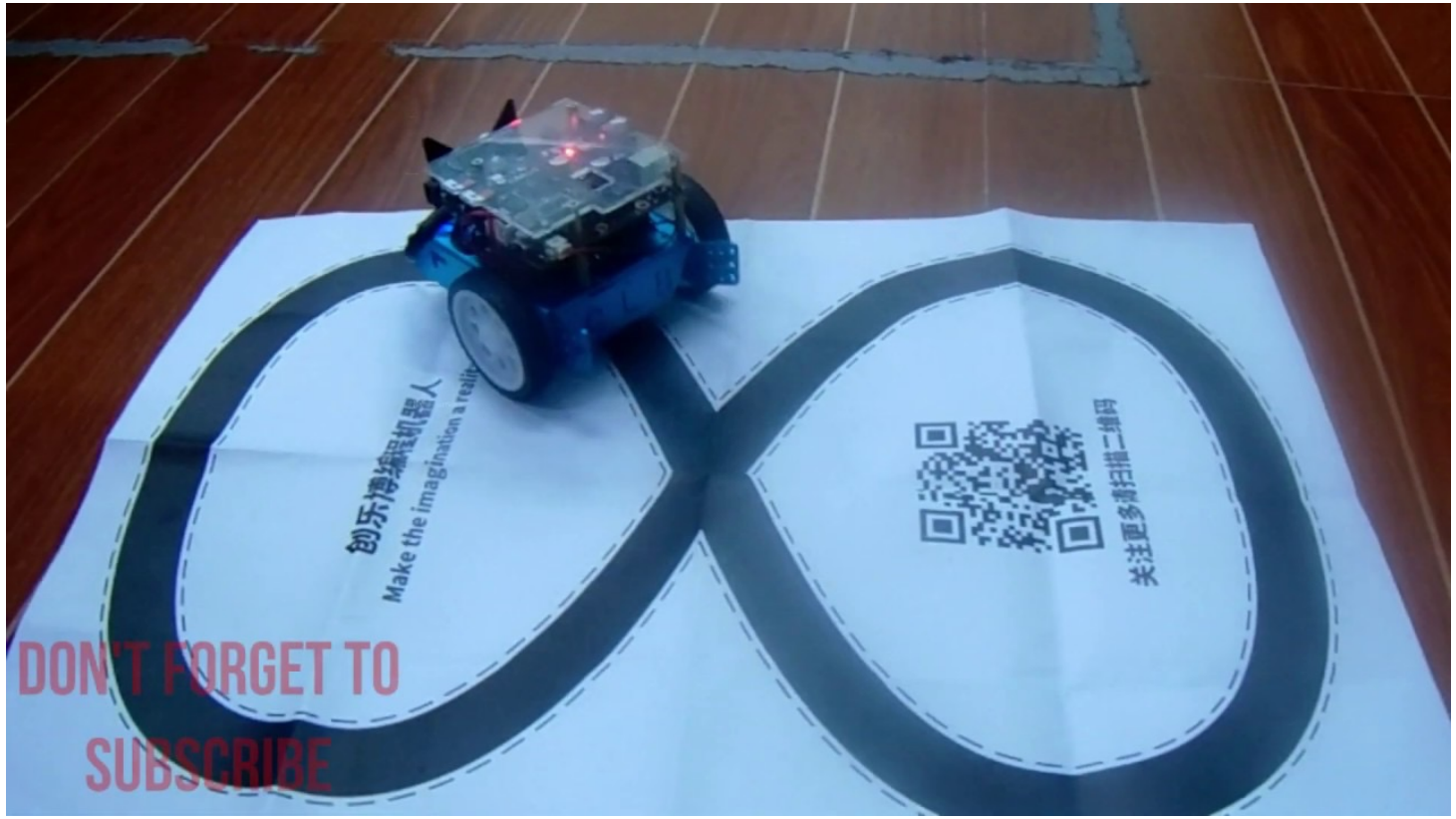
Some effective techniques for testing embedded systems include:

- *scaffolding* - allows you to verify the overall correctness of the program and “abstract” the hardware as working properly.
- *event simulation* – allows you to simulate program events (i.e. timers, interrupts, etc.) to insure that responder routines are executing properly.
- *scripting* – provides a general way of testing program flow and simulating hazard conditions.
- *real-time debugger/simulator* – allows you to execute the program on the host as if it were running on the embedded system.

Robustness Testing

Once the correctness of the system has been verified, its robustness can be tested by:

- *erroneous input* – purposefully try to crash/disrupt system operation. Find the "what ifs" and try them out.
- *overloading* – determine conditions that are unlikely to exist and subject your code to them. Does the system respond gracefully?
- *field-testing* – put the system into its actual environment and test it for some fixed length of time (a.k.a. "burn-in" period).
- *extension* – place the system into an environment for which it was not originally designed. Does the system still function? If not, can the algorithms used be made more general to handle this situation?


DON'T FORGET TO
SUBSCRIBE