Verifying HW/SW Integration

Test and Verification Solutions
Delivering Tailored Solutions for Hardware Verification and Software Testing
Why do things break?

**Design Process**
- **Systematic Errors**: All Devices, Minimize!

**Physical Effects**
- **Random Errors**: Individual Devices, Safeguard!

**Systematic Errors**
- **Machine Errors**
  - Synthesis bugs, ..
- **Human Errors**
  - Implementation bugs (protocol errors...)
  - Design bugs (broken features, ..)

**Random Errors**
- **Hard Errors**
  - Latch-ups, burnouts (stuck-at faults)
- **Soft Errors**
  - Transients (glitches, bit-flips)
System Development Process

System Development Process
ARP-4754/ED-79

DO-178C/ED-12C
Software Development Process

DO-254/ED80
Hardware Development Process

SAFETY ASSESSMENT PROCESS
ARP 4761 / ED-135
Flow Between System & Software Life Cycle Process

A:
- System Requirement allocated to SW
- System Safety Objectives
- Software Levels
- System Description and Hardware Definition
- Design Constraints
- Definition of System Verification Activities to be performed by SW Process
- Definition and Evidence of any Software Verification Activities performed by System Process
- Evidence of Acceptability of data

B:
- Description of Software Architecture
- Evidence of any system Verification Activities Performed
- Any Limitation to use
- Configuration Identification Data
- Data to Facilitate Integration
- Software Verification Activities to be performed by System Processes

C:
- HW/SW Integration Requirements
- Coordination and Feedback
- Identified HW/SW Incompatibilities

D:
- Information Flow Between System and Hardware Life Cycle Processes
System Development Process

Hardware/Software Integration Verification

Software Integration Verification

Hardware Integration Verification
System Development Process

- Hardware/Software co-simulation

  Hardware/Software Integration Verification

  Software Integration Verification
  - Formal interface specifications
  - Integration of behaviors
  - Multi-threaded software

  Hardware Integration Verification
  - Interface assertions
  - Security concerns
System Development Process

- Hardware/Software co-simulation

  Hardware/Software Integration Verification

  - Formal interface specifications
  - Integration of behaviors
  - Multi-threaded software

  Software Integration Verification

  Hardware Integration Verification

  - Interface assertions
  - Security concerns
Formal Specification Examples

BirthdayBook

\[ \text{known} : \mathbb{P} \text{NAME} \]
\[ \text{birthday} : \text{NAME} \rightarrow \text{DATE} \]

\[ \text{known} = \text{dom} \text{birthday} \]

BirthdayBook

\[ \Delta \text{BirthdayBook} \]
\[ \text{name?} : \text{NAME} \]
\[ \text{date?} : \text{DATE} \]

\[ \text{name?} \notin \text{known} \]
\[ \text{birthday'} = \text{birthday} \cup \{\text{name?} \rightarrow \text{date?}\} \]

Can clearly identify interface conditions
• Identify tests
• Identify interface assertions
Formal Specification Examples

\[
\begin{align*}
\text{max} : \mathbb{N} \\
\text{max} > 100
\end{align*}
\]

**Collection**

- \(\text{elems} : \mathcal{P} T\)
- \(#\text{elems} \leq \text{max}\)

**Init**

\[
\text{elems} = \emptyset
\]

**Add**

\[
\begin{align*}
\Delta(\text{elems}) \\
x? : T
\end{align*}
\]

\[
\begin{align*}
x? \not\in \text{elems} \\
\text{elems}' = \text{elems} \cup \{x?\}
\end{align*}
\]

**Delete**

\[
\begin{align*}
\Delta(\text{elems}) \\
x! : T
\end{align*}
\]

\[
\begin{align*}
x! \in \text{elems} \\
\text{elems}' = \text{elems} \setminus \{x!\}
\end{align*}
\]
System Development Process

- Hardware/Software co-simulation

Hardware/Software Integration Verification

Software Integration Verification

- Formal interface specifications
- Integration of behaviors
- Multi-threaded software

Hardware Integration Verification

- Interface assertions
- Security concerns
Traditional view of software integration

8.4 Integration Testing
View of a System

- System viewed as a hierarchy of components

Pfleeger and Atlee, Software Engineering: Theory and Practice  Chapter 8.36
The Pyramid of Proof

Integration of Pre-tested Components
Integration of behaviors in cyber physical systems

- Cyber-Physical Systems (CPS) are integrations of computation, networking, and physical processes.

- Embedded computers and networks monitor and control the physical processes, with feedback loops where physical processes affect computations and vice versa.

- Verification Challenges
  - Large input space
  - They may also have some self-learning aspects but this is not a necessity
  - Integration of multiple behaviours
What are cyber physical systems?

- Cyber-Physical Systems (CPS) are integrations of computation, networking, and physical processes.
- Embedded computers and networks monitor and control the physical processes, with feedback loops where physical processes affect computations and vice versa.
- They may also have some self-learning aspects but this is not a necessity.
Examples of Cyber Physical Systems

- Autonomous and off-board systems
  - Unmanned Surface Vehicle steers its way from A to B
  - Potentially towing a payload
  - Steering clear of obstacles and collaborating with Unmanned Underwater Vehicles
  - Communicating via a central operations centre
  - Multiple goal driven behaviours

- Verify behaviours individually
- Verify integration of behaviours
Maritime Autonomy Framework – Open Architecture

An open architecture that supports:

- Multi-system goal based planning
- Autonomous task decomposition & management
- Vehicle & sensor level autonomy
- The integration of technology from multiple providers in a coherent and structured manner
Examples of Cyber Physical Systems

- Autonomous Vehicles

- Must
  - Get from A to B
  - With minimal time and fuel
  - SAFELY!
The V&V Challenge

- Cyber Physical Systems introduce a complex software testing challenge
  - A large input space
  - Difficulty predicting expected response

- Hardware faced a similar problem 20 years ago
  - Over the past 20 years a number of “Advanced Hardware Verification Techniques” (AHVT) have been introduced
  - To automate test generation and response checking

- Can this be done within a safety framework?
Advanced Hardware Verification Techniques

- Software Requirements
- Test Plan
- Test Results
- Coverage
- Checker
- Monitor
- Constrained Random Input
- Formal Model
- Software Under Test
- Active
- Passive
- Doors, etc
Results of Bubble Sort “Proof of Concept”

Lists of
- Integers
- Floats
- Ascii
- etc

Constrain towards
- Empty lists
- Equal values
- Reverse ordering

Checkers
- Check output list is ordered
- Output list contents == input list contents

Software Under Test

Coverage Metrics
- Empty List
- Reverse ordered
- Error cases (mix integers, floats, ascii
  etc)

List Generator

Lists
Example Constrained Random Inputs

- *Mimic sensor input data*
- *Need to constrain those inputs*
  - Only the legal space
  - Hit the corner cases
- *Example scenarios*
  - Valid ranges for data
  - Relationships between inputs
  - Next input within certain “distance” to prior input
Functional Coverage

From Kerstin Eder of the University of Bristol

- Requirements coverage
- “Cross-product” coverage

[A cross-product coverage model is composed of the following parts:
1. A semantic **description** of the model (story)
2. A list of the **attributes** mentioned in the story
3. A set of all the **possible values** for each attribute (the attribute value **domains**)
4. A list of **restrictions** on the legal combinations in the cross-product of attribute values

A **functional coverage space** is defined as the Cartesian product over the attribute value domains.

- Situation coverage

Example Checkers

- **Do not accelerate too fast**
  - Assert that output to motor is not too high

- **“always respond correctly”**
  - If A&B&C occur then check X happens
    - Assertion coverage “check A&B&C occurs” for free

- **Always safe**
  - Do not get too close to other objects
  - Requires some level of modelling

- **Minimise resources**
System Development Process

- Hardware/Software co-simulation

Hardware/Software Integration Verification

- Software Integration Verification
  - Formal interface specifications
  - Integration of behaviors
  - Multi-threaded software

- Hardware Integration Verification
  - Interface assertions
  - Security concerns
Multi-threaded software
From sequential to parallel

- Execution of a sequential program only depends on the starting state and its inputs.
- Execution of parallel programs also depends on the interactions between them.

Shared Memory:
- Communication is based on altering the contents of shared memory locations (Java).
- Usually requires the application of some form of locking (e.g., mutexes, semaphores, or monitors) to coordinate between threads.

Message Passing:
- Communication is based on exchanging messages (occam, XC).

![Diagram of multi-threaded program execution]
Many Potential Execution Orders

7 states

6 states

30 states

Evolution of $P_1$

Evolution of $P_2$
The problems with distributed processing

Non-determinism

- Non-determinism
  - We cannot guarantee the order of execution
  - And this can lead to race conditions

![Diagram showing code running on core1 and core2 with shared memory](image)
Race Condition Examples

static int num = 0;

thread1 () {
    int val = num;    // step 1
    num = val + 1;    // step 3
}

thread2 () {
    int val = num;    // step 2
    num = val + 1;    // step 4
}

static int num = 0;

thread1 () {
    int val = num;    // step 1
    num = val + 1;    // step 2
}

thread2 () {
    int val = num;    // step 3
    num = val + 1;    // step 4
}

So why don’t we just use “num++”?
Another example

```java
static void transfer(Transfer t) {
    balances[t.fundFrom] -= t.amount;
    balances[t.fundTo] += t.amount;
}
```

- **Expected Behavior:**
  - Money should pass from one account to another

- **Observed Behavior:**
  - Sometimes the amount taken is not equal to the amount received

- **Possible bug:**
  - Thread switch in the middle of money transfers
  - Second thread updates “Transfer” or “balances” in parallel

- **So what are the solutions?**
  - Locks and mutual exclusion
The use of locks

class mutex{
    mutex() { //get the appropriate mutex
    ~mutex() { //release it }
    private: sometype mHandle;
}
void foo() {
    mutex get; //get themutex
    ...
    if(a) return; //released here
    ...
    if(b) throw“oops”; //or here
    ...
    return; //or here
}

Must remember to release the “mutex”!

But there are so many possible “exits” from the code

Locks can cause bugs – especially DEADLOCK!
## Example deadlock

<table>
<thead>
<tr>
<th>Code for Process P</th>
<th>Code for Process Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock(M)</td>
<td>Lock(N)</td>
</tr>
<tr>
<td>Lock(N)</td>
<td>Lock(M)</td>
</tr>
<tr>
<td><strong>Critical Section</strong></td>
<td><strong>Critical Section</strong></td>
</tr>
<tr>
<td>Unlock(N)</td>
<td>Unlock(M)</td>
</tr>
<tr>
<td>Unlock(M)</td>
<td>Unlock(N)</td>
</tr>
</tbody>
</table>
Cause of deadlock

1. Tasks claim exclusive control of the resources they require ("mutual exclusion“ condition).

2. Tasks hold resources already allocated to them while waiting for additional resources ("wait for" condition).

3. Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion ("no preemption" condition).

4. A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain ("circular wait“ condition).
Commonly found bugs

- An operation is assumed to be atomic but it is actually not.
- Wrong or no lock
- Message protocol errors
  - Mismatch between channel ends
  - Missing send or receive
- Orphaned threads due to abnormally terminating master thread

Catching these bugs is challenging
Finding bugs in parallel SW

- Research shows that bugs due to parallel code execution represent only ~10% of the bugs
- But they are the hardest to find
  - If we run the same test twice it is not guaranteed to produce the same result (non-determinism)
    - Heisenbug
- A disproportionate number are found late or by the customer
  - Require large configurations to test
  - Typically appear only in specific configurations
  - These bugs are the most expensive!

- We need to have some ways of disturbing the execution
- We need to know when we are done
Heisenbugs

- A bug that disappears or alters its behavior when one attempts to probe or isolate it

Why?
- Using a debugger alters the execution order and the bug disappears
- Even adding a print statement changes behaviour!

For example
- Uninitialised variable in a sequential program
- In C, 9 out of 10 Heisenbugs are from uninitialized auto variables
- In parallel programs
- Race conditions and deadlocks are both examples

How much time do you currently spend in debug?
Disturbing the execution

Philosophy
- Modify the program so it is more likely to exhibit bugs (without introducing new bugs – no false alarms)
- Minimize impact on the testing process (under-the-hood technology)
- Reuse existing tests

Techniques to instrument concurrent events
- Concurrent events are the events whose order determines the result of the program, such as accesses to shared variables, calls to synchronization primitives
- At every concurrent event, a random-based decision is made whether to cause a context switch – noise injection
  - e.g., using a sleep statement
New coverage models

- **For shared memory:**
  - Synchronization coverage
    - *Make sure that every synchronization primitive was fully exercised*
  - Shared variable coverage
    - *Make sure shared variables were accessed by more than one thread*
  - ConTest from IBM implements these coverage models

- **For message passing:**
  - Communication coverage for multi-threaded message passing programs
New Coverage Model

[Kyriakos Georgiou, K. Eder, TVS, XMOS]

- Captures communication infrastructure in a message-passing program
  - Fully automatic extraction of coverage tasks up to SYN-events
  - User input required for SYN-sequences (tool support, semantics)
  - Complements existing code and functional coverage models

- Added value in practice
  - Bugs can be captured before even running test cases on the code
  - SYN-sequences permit testing protocol compliance
System Development Process

- Hardware/Software co-simulation

Hardware/Software Integration Verification

Software Integration Verification
- Formal interface specifications
- Integration of behaviors
- Multi-threaded software

Hardware Integration Verification
- Interface assertions
- Security concerns
Inputs to Formal

- **3 inputs to the tool**
  - A model of the design
  - A property or set of properties representing the requirements
  - A set of assumptions, expressed in the same language as the properties
    - typically constraints on the inputs to the design

- **For example**
  - Usually RTL
  - Items are transmitted to one of three destinations within 2 cycles of being accepted
    - \((\text{req}_\text{in} \land \text{gnt}_\text{in}) \rightarrow \#\#[1;2] (\text{rec}_a \lor \text{rec}_b \lor \text{rec}_c)\)
  - The request signal is stable until it is granted
    - \((\text{req}_\text{in} \land \neg \text{gnt}_\text{out}) \rightarrow \#\#1 \text{ req}_\text{in}\)
    - We would of course need a complete set of constraints
Simulation Depth-first vs. Formal Breadth-first

- Where the nodes are states in the simulation
- And the arcs are clocked transitions
- **But the trees are**
  - Very wide
  - Very deep
Formal Verification – terms of reference

- Model Checking
  - Requirements of a design are expressed in a formal mathematical language
  - Tools are used to analyze whether there is a way that a model of the design fails to satisfy the requirements

- Not covered here
  - Equivalence Checking
    - Tools are used to analyze whether one model of a design is a “correct” implementation of another
  - Formal modelling and proofs

Currently mainly RTL-Gates, Gates-Gates Checking of sequential retiming possible
But SystemC to RTL is appearing
Model Checking – Outputs from the tools

- **Proved**
  - the property holds for all valid sequences of inputs

- **Failed($n$)**
  - there is at least one valid sequence of inputs of length $n$ cycles, as defined by the design clock, for which the property does not hold.
  - In this case, the tool gives a waveform demonstrating the failure.
  - Most algorithms ensure that $n$ is as small as possible, but some more advanced algorithms don’t.

- **Explored($n$)**
  - there is no way to make the property fail with an input sequence of $n$ cycles or less
  - For large designs, the algorithm can be expensive in both time and memory and may not terminate
System Development Process

• Hardware/Software co-simulation

Hardware/Software Integration Verification

Software Integration Verification

• Formal interface specifications
• Integration of behaviors
• Multi-threaded software

Hardware Integration Verification

• Interface assertions
• Security concerns
Security verification across HW interfaces

Untrusted (Wireless Radio)

Critical (Insulin pump)
Security verification across HW interfaces

Secret (HW Key)

Untrusted (Unknown IP)
Secure Data over Packet Channel

- No direct association between signal name and secure / non-secure!
  There’s a control / temporal component also.

- Secure Formal does not help in analyzing strength of Cryptography blocks.

RED = secure
Green = non-secure
Blue = mixed
Data Leakage

- Identify secure signals [sources] we are concerned about (typically not many)

- [Auto] identify all the places it might be able to reach (typically hundreds or thousands – ALL the outputs, or top level signals of the block)

- Confirm that signal can ONLY reach the places it’s supposed to, and not anyplace where the bad guys could steal it.

- **No way to directly specify this in SVA!**
System Development Process

- Hardware/Software co-simulation

Hardware/Software Integration Verification

- Software Integration Verification
  - Formal interface specifications
  - Integration of behaviors
  - Multi-threaded software

- Hardware Integration Verification
  - Interface assertions
  - Security concerns
Hardware Software Co-Verification

Application

Driver

Hardware

Test Application

Driver

RTL Simulation

DPI
Hardware Software Co-Verification

- Test Application
  - Test DPI
    - Test bench
      - Inject error conditions
  - Driver
    - DPI
        - RTL Simulation
      - RTL
        - DPI
  - Driver
    - DPI
  - Test Application
     - 90% of code to cover error conditions