Verifying Functional, Safety and Security Requirements
(for Standards Compliance)

Mike Bartley (TVS)
in collaboration with

OneSpin Solutions
Tortuga Logic
TVS
TVS Agenda

- 11.00 Introductions
- 11.05 TVS
- 11.20 OneSpin
- 11.55 Tortuga Logic
- 12.10 TVS
- 12.25 Q&A (TVS, OneSpin, Tortuga Logic)
Introduction to your speakers

- **Dr Mike Bartley, CEO of TVS**
  - PhD in Mathematical Logic, MSc in Software Engineering, MBA
  - 25 years in Software Testing and Hardware Verification
  - Started TVS (Test and Verification Solutions) in 2008
    - 125 engineers worldwide
    - UK, France, Germany, Italy, Sweden, Turkey, India, Singapore, Korea, China, US
    - Delivering SW test and HW verification products and services
      - Focus on reliability, safety, security
  - asureSIGN
    - “Requirements Driven Test and Verification” methodology
    - Define requirements and refine them to verification plans and capture sign-off
Jörg Große recently joined OneSpin Solution as a Product Manager for Functional Safety.
He has more than 20 years of experience in EDA, functional verification and ASIC design, having served at companies in Europe, the United States and New Zealand.
As co-founder of a successful Silicon Valley based startup, he was central in developing the concept of fault/mutation testing into a state-of-the-art EDA tool. He deployed this technology in many leading semiconductor companies, increasing the quality of their functional verification.
He holds a Dipl.-Ing.(FH) in Electrical Engineering from the University of Applied Science Anhalt.
Dr. Ryan Kastner is a co-founder of Tortuga Logic and has over 10 years of experience in the realm of hardware security. He has served as a principal investigator on various government and industrial grants related to hardware security (over $3 million in toto). This includes the National Science Foundation Innovation Corps award, which focuses on commercializing technology from academia. Dr. Kastner is a professor in the Computer Science and Engineering Department at UCSD. He received a PhD in Computer Science at UCLA, a masters degree (MS) in engineering and bachelor degrees (BS) in both Electrical Engineering and Computer Engineering, all from Northwestern University.
TVS Agenda

- 11.00 Introductions
- **11.05 TVS**
  - Safety and security in Hardware and Software
  - Requirements Driven Test and Verification (RDTV)
  - Using an ECC example and breaking it down into a test plan
- 11.20 OneSpin
- 11.55 Tortuga Logic
- 12.10
  - Analysing the results and signoff
  - Advantages of RDTV
- **12.25 Q&A (TVS, OneSpin, Tortuga Logic)**
**Why are Safety and Security important?**

- **IC Insights research**
  - The automotive industry is set to drive chip demand over the coming years.
  - IC Insights research suggests the demand from automotive is expected to exhibit average annual growth of 10.8% into at least 2018.
  - Demand will come from safety features that are increasingly becoming mandatory, such as backup cameras or eCall, and the near-ubiquitous driver-assistance systems.

- **IoT**
  - Drones (avionics), autonomous cars, robots, ....
  - Connected devices have potential security threats

- **TTTech**
  - By 2020 50% of all ICs will be safety-related
  - By 2020 50% of all ICs will be connected
Safety Standards

- **IEC61508**: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems
- **DO254/DO178**: Hardware/Software considerations in airborne systems and equipment certification
- **EN50128**: Software for railway control and protection systems
- **IEC60880**: Software aspects for computer-based systems performing category A functions
- **IEC62304**: Medical device software -- Software life cycle processes
- **ISO26262**: Road vehicles – Functional safety
Safety

“Freedom from unacceptable risk of physical injury or of damage to the health of people, either directly, or indirectly as a result of damage to property or to the environment”

Functional Safety

“That part of the overall safety that depends on a system or equipment operating correctly in response to its inputs”
How Systems Fail

- **Random failures**
  - Can usually predict (statistically)
  - Can undertake preventative activities

- **Systematic failures**
  - Specified, designed or implemented incorrectly
  - Can’t usually predict

- **Systemic failures**
  - Shortcomings in culture or practices
Basics of Safety Standards

- The life cycle processes are identified
- Objectives and outputs for each process are described
  - Objectives are mandatory
  - But vary by Integrity Level
  - For higher Integrity Levels, some Objectives require Independence
### Dynamic analysis and testing

<table>
<thead>
<tr>
<th>Technique</th>
<th>SIL 1</th>
<th>SIL 2</th>
<th>SIL 3</th>
<th>SIL 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural test coverage (entry points) 100%</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Structural test coverage (statements) 100%</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Structural test coverage (branches) 100%</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Structural test coverage (conditions, MC/DC) 100%</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>Test case execution from boundary value analysis</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Test case execution from error guessing</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Test case execution from error seeding</td>
<td>-</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Test case execution from model-based test case</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>generation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance modelling</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>Equivalence classes and input partition testing</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
</tbody>
</table>
Key Processes

- Plans & Standards
- Requirements
- Design Specifications
- Reviews and Analyses
- Testing (against specifications)
  - At different levels of hierarchy
- Test Coverage Criteria
- Requirements Traceability
- Independence
Key Deliverables

- Verification Plan
- Validation and Verification Standards
- Traceability Data
- Review and Analysis Procedures
- Review and Analysis Results
- Test Procedures
- Test Results
- Acceptance Test Criteria
- Problem Reports
- Configuration Management Records
- Process Assurance Records
Key Deliverables

- Verification Plan
- Validation and Verification Standards
- Traceability Data
- Review and Analysis Procedures
- Review and Analysis Results
- Test Procedures
- Test Results
- Acceptance Test Criteria
- Problem Reports
- Configuration Management Records
- Process Assurance Records
Traceability in Practice

Shows a mapping from features to verification and test plans
Example – Safeguarding a FIFO

**Safety Function**
- Detect 1-bit errors and correct them
- Detect 2-bit errors and raise alarm

**Design:**
- Encoder adds \( e \) data bits stored in RAM
- Decoder detects & corrects 1-bit faults on read (\( \text{error}=0, \text{corrected}=1 \))
- Decoder detects 2-bit faults on read (\( \text{error}=1 \))
A full set of requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 FIFO_SINGLE_BIT</td>
<td>The FIFO will be able to detect and correct single bit errors.</td>
</tr>
<tr>
<td>R2 ERR_REPORT_CPU</td>
<td>Single bit errors must be reported to the CPU.</td>
</tr>
<tr>
<td>R3 MULT_ERR_CPU</td>
<td>The FIFO will be able to detect and report multiple bit errors to the CPU.</td>
</tr>
<tr>
<td>R4 FIFO_NOT_FULL</td>
<td>Data arriving on the write interface shall be written into the FIFO as long as it is not full.</td>
</tr>
<tr>
<td>R5 FIFO_NOT_EMPTY</td>
<td>Requests to read data shall return the oldest data in the FIFO as long as it is not empty.</td>
</tr>
<tr>
<td>R6 FIFO_EMPTY_READ</td>
<td>Read attempts from an empty FIFO shall be reported to the CPU.</td>
</tr>
<tr>
<td>R7 FIFO_WRITE_FULL</td>
<td>Write attempts to a full FIFO shall be reported to the CPU.</td>
</tr>
<tr>
<td>R8 WRITE_APB_INTERFACE</td>
<td>Write data shall come across an APB interface.</td>
</tr>
<tr>
<td>R9 READ_APB_INTERFACE</td>
<td>Read data shall be sent across an APB interface.</td>
</tr>
<tr>
<td>R10 STATUS_REG_SINGLE_ERR</td>
<td>A status register will record a single bit error.</td>
</tr>
<tr>
<td>R11 STATUS_REG_MULTI_ERR</td>
<td>A status register will record a multibit error bit error.</td>
</tr>
<tr>
<td>R12 STATUS_REG_FIFO_FULL_</td>
<td>A status register will indicate a FULL fifo.</td>
</tr>
<tr>
<td>R13 STATUS_REG_FIFO_EMPTY</td>
<td>A status register will indicate an empty fifo.</td>
</tr>
<tr>
<td>R14 STATUS_REG_FIFO_OVERFLOW</td>
<td>A status register will indicate overflow.</td>
</tr>
<tr>
<td>R15 STATUS_BIT_OVERFLOW</td>
<td>A status bit will record underflow.</td>
</tr>
<tr>
<td>R16 PRIVILEGE_LEVEL_1</td>
<td>Only users with privilege level 1 can read from the FIFO.</td>
</tr>
<tr>
<td>R17 PRIVILEGE_LEVEL_1_2</td>
<td>Only users with privilege level 1 or 2.</td>
</tr>
</tbody>
</table>
Safety Requirement Decomposition (example)

Req: Safeguard Design against single bit soft errors

Sub-Concept/Req: Safeguard each FIFO

Safety Requirements for FIFO / Concept:
- Use ECC FIFO
- Detect 1-bit errors and correct them
- Detect 2-bit errors and raise alarm

Safety Verification Requirement for ECC FIFO Implementation
- If no error occurs, nothing is flagged and the data is uncorrupted
- If one error occurs, no error is flagged, the data is uncorrupted and the correction is flagged
- If two errors occur, an error is flagged, but no correction

Formal Safety Properties to verify Implementation
- Separate slide
Mapping Security Requirements to Features

- R16 - PRIVILEGE_LEVEL_1: only users with privilege level 1 can read from the FIFO

<table>
<thead>
<tr>
<th>ECC_SECURITY_1</th>
<th>Reads without privilege level 1 or 2 will cause a bus error</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECC_SECURITY_2</td>
<td>Reads with privilege level 1 or 2 will be successful</td>
</tr>
</tbody>
</table>
Mapping Requirements to Verification Metrics

Relationships can be:
- Bi-directional
- Many-many

Metrics can be:
- From HW verification
- From Silicon validation
- From SW testing
asureSIGN Demo

- Mapping the requirements to a test plan
TVS Agenda

- **11.00 Introductions**
- **11.05 TVS**
  - Safety and security in Hardware and Software
  - Requirements Driven Test and Verification (RDTV)
  - Using an ECC example and breaking it down into a test plan
- **11.20 OneSpin**
- **11.55 Tortuga Logic**
- **12.10 TVS**
  - Analysing the results and signoff
  - Advantages of RDTV
- **12.25 Q&A (TVS, OneSpin, Tortuga Logic)**
Requirement Driven Verification for Safety & High Reliability

Jörg Große
Why Safety & Reliability Verification is important - Risk Drivers

• Cars are computer on wheels
  – But reset is not an option, especially not when diving at high speeds

Systematic Errors
  – Machine Errors
    • Synthesis bugs, ..
  – Human Errors
    • Implementation bugs
    • Design bugs
  – Driven by
    • Ever increasing complexity
    • Time to market and budget

Random Errors
  – Hard Errors
    • Latch-ups
    • Burnouts (struck-at faults)
  – Soft Errors
    • Transients (glitches, bit flips)
  – Driven by
    • Decreasing geometries
    • Decreasing supply voltage
    • Increasing area
Consequence?

Design Process
- Systematic Errors
  - All Devices
    - Minimize!

Physical Effects
- Random Errors
  - Individual Devices
    - Safeguard!

Functional Verification + Safeguard Verification = Functional Safety Verification
Safeguarding against Random Errors

Fault Detection
- Raise alarm

Fault Handling
- Enter into safe mode
- Or correct erroneous output

Examples
- Parity, ECC, lock-step
Additional Verification Effort for

Functional Safety Verification

Minimize Systematic Errors
- Rigorous Verification
- Quantification of Verification

Safeguard Random Errors
- Verification of Safety Mechanisms
- Diagnostic Coverage

Requirement Driven

Puts additional pressure on Time-to-Market & Budget!
=> Automation
Minimizing Systematic Errors with Rigorous Requirement Based Verification
Generic Verification Flow with Requirement Tracing
Example Design
Safeguarding a FIFO with ECC

- **Safety Functions**
  - Detect 1-bit errors and correct them
  - Detect 2-bit errors and raise alarm
- **Design:**

  - Encoder adds $e$ data bits stored in FIFO

```plaintext
wr_en  
rd_en  
wr_data
```

<table>
<thead>
<tr>
<th>w</th>
<th>w+e</th>
</tr>
</thead>
<tbody>
<tr>
<td>wr_data</td>
<td>rd_data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>decoder</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>corrected</td>
</tr>
<tr>
<td>error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>w+e</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
</tr>
<tr>
<td>empty</td>
</tr>
</tbody>
</table>

© Accellera Systems Initiative
**Functional & Safety Requirements**

**Functional Requirements**
- The FIFO is not full and empty at the same time
- The FIFO is empty after DEPTH many reads without writes
- The FIFO is full after DEPTH many writes without reads
- The FIFO is no longer empty after a write
- The first data written to an empty FIFO leaves the FIFO unmodified on the first read

**Safety Requirements**
- If no error occurs, nothing is flagged and the data is uncorrupted
- If one error occurs, no error is flagged, the data is uncorrupted and the correction is flagged
- If two errors occur, an error is flagged, but no correction
Mapping Requirements to Properties

**Functional Requirements**

- The FIFO is not full and empty at the same time
- The FIFO is empty after DEPTH many reads without writes
- The FIFO is full after DEPTH many writes without reads
- The FIFO is no longer empty after a write
- The first data written to an empty FIFO leaves the FIFO unmodified on the first read
Formal Property

Requirement based verification

→ Create assertions for each requirement!

```
not_empty_after_write_a: assert property
(disable iff (!FIFO.reset_n) wr_en |=> !empty);
```

Example: `assert.not_empty_after_write_a`
“The FIFO is no longer empty after a write”
Formal Assertion Based Verification

- **Early:** No stimulus or testbench is needed
- **Efficient:** Typically check-debug-fix in minutes
- **Exhaustive:** If assertion holds -> no simulation needed
Mapping Requirements to Properties

**Functional Requirements**

- The FIFO is not full and empty at the same time
- The FIFO is empty after DEPTH many reads without writes
- The FIFO is full after DEPTH many writes without reads
- The FIFO is no longer empty after a write
- The first data written to an empty FIFO leaves the FIFO unmodified on the first read
Reliable Quantification of Formal Assertion Sets Coverage Reloaded
Quantitative Analysis of Verification

How much of my DUV is verified?

- Requirements
- coverage metrics
- stimulus/constraints
- checkers/assertions
- DUV

How good are my test vectors & constraints?

Often discounted: How good are my checkers and assertions?
Cone-of-Influence Coverage

Covered by COI of A

Assertion A

DUV
A Trivial Example – COI Coverage

```
module shift_reg(input clk, input rst, input in, output reg out);

logic [3:0] s;
assign out = s[3];
always @(posedge clk)
begin
    if(rst)
        s <= 4'b0000;
    else
        begin
            s[0] <= in;
            s[1] <= in;
            s[2] <= s[1];
            s[3] <= s[2];
        end
    end
state_2i3 : assert property (@(posedge clk) disable iff(rst) s[2] => s[3]);
endmodule
```

But lines: 7,10,11,12 are not verified.
Potential bugs could escape!

What line coverage would you expect from this assertion when using COI coverage?
Prover Coverage

Whatever the prove engine needs is considered covered.

Corresponds to abstractions inside prove engines.

Each prove engine uses different abstractions.

No guarantee that what the prove engine needs is fully covered!
A Trivial Example – Prover Coverage

```
module shift_reg(input clk, input rst, input in, output out);

logic [3:0] s;
assign out = s[3];
always @(posedge clk)
begin
  if(rst)
    s <= 4'b0000;
  else
    begin
      s[0] <= in;
      s[1] <= s[0];
      s[2] <= in;
      s[3] <= s[2];
    end
state_2i3 : assert property (@(posedge clk) disable iff(rst) s[2] |==> s[3]);
endmodule
```

But line 12 is not verified. Potential bugs could escape!

Prove engine needs at least s[2] and s[3].
Observation Coverage Principle

- Has the statement been activated?
  - If a statement has not been activated during verification, it can’t break a check.
  - Measures reachability.
- Has the effect been observed?
  - If a statement is modified and activated, some assertion should fail.
  - Measures quality of assertions.

Example: Statement Coverage

case (state)
  ...  
  burst:
    if (cancel_i)  
    done_o <= 1
  ...

active

Coverage

Activation

Observation

case (state)
  ...  
  burst:
    if (cancel_i)
    done_o <= v
  ...

modify

Been there! Done that!
Unlike COI coverage, observation coverage identifies all unchecked assignments.

Need better or more assertion(s).
Quantification of Properties

Functional Requirements

- The FIFO is not full and empty at the same time
- The FIFO is empty after DEPTH many reads without writes
- The FIFO is full after DEPTH many writes without reads
- The FIFO is no longer empty after a write
- The first data written to an empty FIFO leaves the FIFO unmodified on the first read

Assertions Hold

All assertions are proven, but how good are they?

Apply OneSpin’s Quantify observation coverage technology.
### Quantify Coverage Report

#### Structural Coverage Overview

<table>
<thead>
<tr>
<th>Status</th>
<th>Statements</th>
<th>Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DR</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>22</td>
<td>10</td>
</tr>
<tr>
<td>DC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DD</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sum</td>
<td>37</td>
<td>31</td>
</tr>
</tbody>
</table>

#### Structural Coverage by File

<table>
<thead>
<tr>
<th>File</th>
<th>Statements</th>
<th>Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>decodey</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>encodery</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>fifo_fifo.sv</td>
<td>20</td>
<td>26</td>
</tr>
</tbody>
</table>

*Expecting FIFO to be fully covered!*
Stronger Assertion Exposes Bug

property data_not_corrupted_p;
... (empty & wr_en, dat=wr_data[WIDTH-1:0]) #1
!rd_en[*0:$] #1 rd_en |=> rd_data[WIDTH-1:0]==dat
|| (!full | empty); // Bad!
Quantify Coverage Report

### Structural Coverage Overview

<table>
<thead>
<tr>
<th>Status</th>
<th>Statements</th>
<th>Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 covered</td>
<td>22, 59.43%</td>
<td>26, 63.87%</td>
</tr>
<tr>
<td>R reached</td>
<td>0, 0.00%</td>
<td>0, 0.00%</td>
</tr>
<tr>
<td>U unknown</td>
<td>0, 0.00%</td>
<td>0, 0.00%</td>
</tr>
<tr>
<td>CR unobserved</td>
<td>0, 0.00%</td>
<td>0, 0.00%</td>
</tr>
<tr>
<td>O uncovered</td>
<td>15, 40.54%</td>
<td>5, 16.13%</td>
</tr>
<tr>
<td>CC constrained</td>
<td>0, 0.00%</td>
<td>0, 0.00%</td>
</tr>
<tr>
<td>CD dead</td>
<td>0, 0.00%</td>
<td>0, 0.00%</td>
</tr>
<tr>
<td>Sum quantify targets</td>
<td>37</td>
<td>31</td>
</tr>
</tbody>
</table>

### Structural Coverage by File

<table>
<thead>
<tr>
<th>File</th>
<th>Statements</th>
<th>Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>decoder.v</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>encoder.v</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>fifo_fix4.v</td>
<td>20</td>
<td>26</td>
</tr>
</tbody>
</table>

Much better after fix! But still something wrong. Visit our booth P6 for full demo!
## Quantify Properties

<table>
<thead>
<tr>
<th>Functional Requirements</th>
<th>Assertions Hold</th>
<th>Coverage Achieved</th>
</tr>
</thead>
<tbody>
<tr>
<td>The FIFO is not full and empty at the same time</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>The FIFO is empty after DEPTH many reads without writes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>The FIFO is full after DEPTH many writes without reads</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>The FIFO is no longer empty after a write</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>The first data written to an empty FIFO leaves the FIFO unmodified on the first read</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Summary Observation Coverage with Quantify

- Observation coverage algorithm drives precise coverage metric
  - Qualifies for safety-critical
  - Also identifies dead code and over-constrained code
  - Provides comprehensive progress metric

- Don't trust COI coverage
  - Maybe good for sanity/quick check
  - But not for safety-critical

- Prover coverage is also problematic for safety-critical
  - Not objective, results depend on prove engine
Verification of Safety Mechanism
Efficient Verification of Safety Functions

Safety Verification Problem
- Safety functions are inactive under normal operation!
- Artificially inject faults into verification to activate

Fault Injection complexity for bit vectors:
- $2^{\text{width}}$ possible data input combinations
- $(\text{width})$ 1-bit errors
- $(\text{width} \times \text{width}-1)$ 2-bit errors

Simulation Based Verification is not a good solution:
- Hard to anticipate all relevant conditions
- **Hard to deal with huge number of faults + combinations!**
- No exhaustive testing feasible

Formal ABV with fault injection
Three Simple Steps to Success

1. Describe expected behavior with no fault injected and prove that it holds.
   \[ \text{property} \ (\text{<antecedent> } \implies \neg \text{Alarm}) \]

2. Describe expected behavior with the fault(s) injected, inject the fault(s) and prove that it holds.
   \[ \text{property} \ (\text{<antecedent> } \implies \text{Alarm}) \]

3. Describe expected behavior with correctable faults injected, inject the correctable faults and prove that it holds.
   \[ \text{property} \ (\text{<antecedent> } \implies \text{Input'} == \text{CorrectedOutput} \wedge \text{Corrected} \wedge \neg \text{Alarm}) \]
FIFO Safety Requirements

**Functional Requirements**

1. The FIFO is not full and empty at the same time
2. The FIFO is empty after DEPTH many reads without writes
3. The FIFO is full after DEPTH many writes without reads
4. The FIFO is no longer empty after a write
5. The first data written to an empty FIFO leaves the FIFO unmodified on the first read

**Safety Requirements**

1. If no error occurs, nothing is flagged and the data is uncorrupted
2. If one error occurs, no error is flagged, the data is uncorrupted and the correction is flagged
3. If two errors occur, an error is flagged, but no correction
Formal ABV with Fault Injection

Application Scenario: FIFO

- For FIFO Example:
  - Create no_error, corrected_no_error and error assertions according to the safety requirements
  - Depending on the assertion, inject Bit-Flip faults at the FIFO output

Inject Faults Here!
Application Scenario: FIFO

SV Assertions for Safety Features

No error $\rightarrow$ nothing flagged, data uncorrupted:

\[
\text{no\_error}: \text{assert property (disable iff } (!\text{reset\_n}) \\
\text{empty} \& \text{wr\_en} \#1 \text{rd\_en} \\
|=> \text{rd\_data} == \text{\$past}(\text{wr\_data},2) \& !\text{rd\_error} \& !\text{rd\_corrected});
\]

One error $\rightarrow$ no error flagged, data uncorrupted, correction flagged:

\[
\text{corrected\_no\_error}: \text{assert property (disable iff } (!\text{reset\_n}) \\
\text{empty} \& \text{wr\_en} \#1 \text{rd\_en} \\
|=> \text{rd\_data} == \text{\$past}(\text{wr\_data},2) \& !\text{rd\_error} \& \text{rd\_corrected});
\]

Two errors $\rightarrow$ error flagged, no correction flagged:

\[
\text{error}: \text{assert property (disable iff } (!\text{reset\_n}) \\
\text{empty} \& \text{wr\_en} \#1 \text{rd\_en} \\
|=> \text{rd\_error} \& \& !\text{rd\_corrected});
\]
How to inject the faults?

• Conveniently use formal fault injection:
  
  Fault location

  Injector

  Formal setup for n-bit faults of desired type

  • User can automatically enable different number/kind of faults for individual assertions
  • Possible to verify generic assertions like “a 2-bit fault gets detected”
  • Supporting FLIP, ST0, ST1, OPEN
Using the Formal Fault Injection

inject_fault -location rd_data_FIFO -type <type> -assert <assertion>

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Inject Fault Type</th>
<th>Expect</th>
</tr>
</thead>
<tbody>
<tr>
<td>safety.no_error</td>
<td>NONE</td>
<td>HOLD</td>
</tr>
<tr>
<td>safety.corrected_no_error</td>
<td>FLIP 1 bit</td>
<td>HOLD</td>
</tr>
<tr>
<td>safety.error</td>
<td>FLIP 2 bit</td>
<td>HOLD</td>
</tr>
</tbody>
</table>
Application Scenario: FIFO

Failing Assertion for Safety Feature

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Inject Fault Type</th>
<th>Got</th>
</tr>
</thead>
<tbody>
<tr>
<td>safety.corrected_no_error</td>
<td>FLIP 1 bit</td>
<td>FAIL</td>
</tr>
</tbody>
</table>
### FIFO Safety Requirements

#### Functional Requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>The FIFO is not full and empty at the same time</td>
<td></td>
</tr>
<tr>
<td>The FIFO is empty after DEPTH many reads without writes</td>
<td></td>
</tr>
<tr>
<td>The FIFO is full after DEPTH many writes without reads</td>
<td></td>
</tr>
<tr>
<td>The FIFO is no longer empty after a write</td>
<td></td>
</tr>
<tr>
<td>The first data written to an empty FIFO leaves the FIFO unmodified on the first read</td>
<td></td>
</tr>
</tbody>
</table>

#### Safety Requirements

<table>
<thead>
<tr>
<th>Condition</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>If no error occurs, nothing is flagged and the data is uncorrupted</td>
<td></td>
</tr>
<tr>
<td>If one error occurs, no error is flagged, the data is uncorrupted and the correction is flagged</td>
<td></td>
</tr>
<tr>
<td>If two errors occur, an error is flagged, but no correction</td>
<td></td>
</tr>
</tbody>
</table>
Summary Verification of Safety Mechanism

- ISO 26262-5 (page 28) highly recommends to apply model based fault injection testing:

Table 11 — Hardware integration tests to verify the completeness and correctness of the safety mechanisms implementation with respect to the hardware safety requirements

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1 Functional testing</td>
<td>++</td>
</tr>
<tr>
<td>2 Fault injection testing</td>
<td>+</td>
</tr>
<tr>
<td>3 Electrical testing</td>
<td>++</td>
</tr>
</tbody>
</table>

- OneSpin provides formal fault injection to meet ISO 26262 and verify safety mechanisms
  - No modification of source code required
  - Supports different fault types and number of faults
  - Unlike simulation, it provides complete proof of all faults in one step
  - Easily maps assertions to faults and checks them
Diagnostic Coverage
Diagnostic Coverage
ISO 26262 Analysis Requirements

• Diagnostic coverage: proportion of hardware element failure rate that is detected or controlled by safety mechanisms

• High diagnostic coverage is needed to achieve a high Automotive Safety Integrity Level (ASIL)
Discussing Diagnostic Coverage of Safety Mechanisms

![Diagram showing the flow of input and output with different stages: Normal Design Function, Fault activation, propagation, and Optional Correction, leading to Corrected Output and Alarm.]
Fault Classification in Semiconductor Context

- Safe faults
  - Faults which cannot propagate
  - Faults which only propagate to non-safety-critical functions (don't violate a safety goal)
  - Faults which are detected by a safety mechanism before they can cause harm

- Unsafe faults
  - Faults which propagate to a safety-critical function without being detected
  - Faults with unknown behavior

Minimize Unsafe Faults
Increase Diagnostic Coverage
Formal Propagation Analysis Summary

• Formal propagation analysis can identify
  – Faults which cannot propagate
  – Whether a fault propagates to a safety-critical function
  – Whether a fault propagates to a safety mechanism

• This information helps to classify faults as safe or unsafe and creates more precise diagnostic coverage of the safety mechanism

More Precise Diagnostic Coverage  Meet Safety Goal
Summary

Functional Safety Verification

Minimize Systematic Errors
- Rigorous Verification
- Quantification of Verification

Safeguard Random Errors
- Verification of Safety Mechanisms
- Diagnostic Coverage

Requirement Driven
Thank you!

To learn more about safety critical design & verification:

• Read Safety Critical News
  – http://safetycritical.onespin-solutions.com/

• Visit us at Booth P6
Questions

Finalize slide set with questions slide
TVS Agenda

- **11.00 Introductions**
- **11.05 TVS**
  - Safety and security in Hardware and Software
  - Requirements Driven Test and Verification (RDTTV)
  - Using an ECC example and breaking it down into a test plan
- **11.20 OneSpin**
- **11.55 Tortuga Logic**
- **12.10 TVS**
  - Analysing the results and signoff
  - Advantages of RDTTV
- **12.25 Q&A (TVS, OneSpin, Tortuga Logic)**
Not just these devices....

How A Creep Hacked A Baby Monitor To Say Lewd Things To A 2-Year-Old

Security risks found in sensors for consumer electronics

Published on May 16, 2013
Contact Nicole Covel Moore, U-M, (234) 647-7067, ncmoore@umich.edu or Lan Yoon, RAST, +82-42-130-2295, hyoon@umich.edu

ANN ARBOR—The type of sensors that pick up the rhythm of a beating heart in implanted cardiac defibrillators and pacemakers are vulnerable to tampering, according to a new study conducted in controlled laboratory conditions.

Implantable defibrillators monitor the heart for irregular beating and, when necessary, administer an electric shock to bring it back into normal rhythm. Pacemakers use electrical pulses to continuously keep

Car Hacking: The Next Big Threat?

posted by Fox Van Allen on July 28, 2013
in Travel & Entertainment, News, Computers and Software, Car Tech & Safety, Blog, Automotive

Before I hacked a stranger’s smart home, I asked for permission. An anonymous

Charlie Miller and Chris Valasek are authors of a 100-page research paper maliciously hack a car’s computer, and potentially kill its occupants. I did so with the support – and funding – of the National Security Agency.
Hackers are now focusing on hardware
Current “State-Of-The-Art”
Designing Secure Hardware

Security Engineers

Did you make it secure?

Hardware Designers

Yes we did!

Tortuga Logic Software
Tortuga Logic’s PROSPECT

Enable “Design-for-Security” from the ground up to minimize security breaches in hardware and systems.

Tortuga Logic’s PROSPECT Software Solution
Prospect Tool flow

Prospect GUI

Results and Debug feedback

Prospect Tool flow

Security Properties

key $\rightarrow$ out $\rightarrow$ coreA $\rightarrow$ coreB
PROSPECT: Key Values

- Automates HW security design
  - Reduce security validation from months to hours
  - Significant cost savings for certification

- Increase security coverage and reduce risk
  - Many checks cannot be done manually

- Makes design for security a priority
Tortuga Logic’s PROSPECT

- Types of addressable security properties
Tortuga Logic’s PROSPECT

- Critical component is adversely affected

Untrusted (Wireless Radio)

Critical (Pacing unit)
Tortuga Logic’s PROSPECT

- Secret data is unintentionally leaked
Case Study – Top-25 Semi Company
Key Flowing Out Of Design

- **Assertion:** Key only flows through AES
  - `assert iflow (key /==> $all_outputs ignoring aes.$all_outputs);`
  - If assertion holds, key only flows to outputs through AES first

- **Real world results**
  - State-of-the-art design with over 10 million gates
  - Actual required properties, impossible to visually inspect
Case Study – Top-25 Semi Company
Key Flowing Out Of Design

- **Assertion:** Key only flows through AES
  - `assert iflow (key => $all_outputs ignoring aes.$all_outputs);`
  - If assertion holds, key only flows to outputs through AES first

- **Real world results**
  - State-of-the-art design with over 10 million gates
  - Actual required properties, impossible to visually inspect
Case Study – Top-25 Semi Company
Key Flowing Out Of Design

- **Assertion:** Key only flows through AES
  - `assert iflow (key =/= all_outputs ignoring aes.$all_outputs);`
  - If assertion holds, key only flows to outputs through AES first

- **Real world results**
  - State-of-the-art design with over 10 million gates
  - Actual required properties, impossible to visually inspect
**Demo: AES Key Leakage**

**Property:**
assert iflow (key /==> data_o);

**Fails in 4 cycles**
Key XOR Data flows to pins, security flaw

![Diagram of AES Key Leakage](image)
Demo: AES Key Leakage

Property:
assert iflow (key =/= data_o);

Fails in 506 cycles
Encrypted data flows to pins
Flow is allowed, ready_o
Demo: AES Key Leakage
Demo: AES Key Leakage

Property:
assert iflow (key $\neq$ $\Rightarrow$ data_o) || ready_o;

Result
Assertion Holds

![Diagram of AES Key Leakage with Key Storage, Encryption Module, Data, data_o, and ready_o connections.]
TVS Agenda

- **11.00 Introductions**
- **11.05 TVS**
  - Safety and security in Hardware and Software
  - Requirements Driven Test and Verification (RDTV)
  - Using an ECC example and breaking it down into a test plan
- **11.20 OneSpin**
- **11.55 Tortuga Logic**
- **12.10 TVS**
  - Analysing the results and signoff
  - Advantages of RDTV
- **12.25 Q&A (TVS, OneSpin, Tortuga Logic)**
Metrics can be:
• From HW verification
• From Silicon validation
• From SW testing

Relationships can be:
• Bi-directional
• Many-many
Use a bi-directional mapping to track backwards

Use an SQL database to hold the mappings and results
asureSIGN™ at the heart of HW/SW V&V

- Requirements
  - Excel
  - Doors
  - Jira
  - etc

- SystemC Simulation
- Hardware Simulation
  - Coverage
  - Assertions
  - Mentor, Aldec
  - Etc.

- UCIS API
- Formal Verification
  - OneSpin

- XML API
- Directed test results

- asureSIGN™ Run API
- Automated SW Test Tool

- Manual API
- Lab Results

- Matlab

- SW Test Tools

- Requirements Engineering tools
Supporting Hierarchical Verification

- A requirement might be signed off at multiple levels of hierarchy during the hardware development
  - Block
  - Subsystem
  - SoC
  - System
    - Including Software
  - Post Silicon
asureSIGN Demo

- Mapping the results to the test plan
Retention of Verification Results (DO 254)

- Verification records should contain a clear correlation to the pass/fail criteria
  - These verification records should contain the author/reviewer, date, and any items used in the including their versions.
  - Any failures or issues found should be correlated to the standard that has been violated.

- Test results should be clearly linked to their associated tests and requirements

- Test Results should be reviewed to be sure that the actual and expect results are giving the correct results and that the tests are passing.
Requirements Driven Verification

- **Compliance to various safety standards**
  - hardware and software (and systems)

- **Some advantages**
  - Identify test holes and test orphans
  - Retention of verification results
    - Build historical perspective for more accurate predictions
  - Better reporting of requirements status
  - Risk-based testing
  - Prioritisation and Risk Analysis
  - Filtering Requirements based on Customers and releases
  - Impact and conflict analysis
Any questions?

I'll need to know your requirements before I start to design the software.

First of all, what are you trying to accomplish?

I'm trying to make you design my software.

I mean what are you trying to accomplish with the software?

I won't know what I can accomplish until you tell me what the software can do.

Try to get this concept through your thick skull: the software can do whatever I design it to do!

Can you design it to tell you my requirements?