Structural Coverage Analysis for Safety-Critical Code - Who Cares?
What is Structural Coverage?

Measurement of Test Effectiveness

- How effectively did tests exercise code?
- Exercised, entry points, statements, branches, compound conditionals, execution paths
- Systems requirement reliability levels up with one defect per $10^9$ operating hours
- Metric that helps determine when a system is adequately tested

Structural Coverage is often mandated

- DO-178B/C DO-278(A) for Commercial/Defense Avionics and Ground Systems
- IEC 61508 for Industrial Controls
- ISO 26262 for Automotive
- IEC 62304 for Medical Devices
- EN 50128 for Rail
Why do Dynamic Analysis?

• It helps to prevent against problems like this…

• Consider the European Space Agency’s Ariane 5 flight 501 on Tuesday, June 4 1996

• Due to an error in the software design (inadequate protection from integer overflow), the rocket veered off its flight path 37 seconds after launch and was destroyed by its automated self-destruct system
Types of Coverage

Depending on the SIL or DAL level and functional safety standard being followed, coverage requirements and required methodology varies:

- Statement Coverage
- Branch Decision Coverage
- Modified Condition / Decision Coverage (MC/DC)
- Data Coupling and Control Coupling Coverage
- Object Code Coverage
- Linear Code Sequence And Jump Coverage (LCSAJ)
DO-178C: Structural Coverage

- The following Structural Coverage is required:

<table>
<thead>
<tr>
<th>Design Assurance Level</th>
<th>Verification Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, C</td>
<td>Statement Coverage</td>
</tr>
<tr>
<td>A, B</td>
<td>Decision Coverage</td>
</tr>
<tr>
<td>A, B, C</td>
<td>Data Coupling &amp; Control Coupling Coverage</td>
</tr>
<tr>
<td>A</td>
<td>Modified Condition / Decision Coverage</td>
</tr>
<tr>
<td>A</td>
<td>Object Code Coverage</td>
</tr>
</tbody>
</table>

Table A-7 5 - Test coverage of software structure (modified condition/decision coverage) is achieved
Table A-7 6 - Test coverage of software structure (decision coverage) is achieved
Table A-7 7 - Test coverage of software structure (statement coverage) is achieved
Table A-7 8 - Test coverage of software structure (data coupling and control coupling) is achieved
Table A-7 9 - Verification of additional code that can not be traced to Source Code, is achieved
IEC 61508: Structural Coverage

- The following Structural Coverage is required:

<table>
<thead>
<tr>
<th>Safety Integrity Level</th>
<th>Verification Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIL 4</td>
<td>Entry Points + Statement + Branches + MC/DC</td>
</tr>
<tr>
<td>SIL 3</td>
<td>Entry Points + Statement + Branches</td>
</tr>
<tr>
<td>SIL 2</td>
<td>Entry Points + Statement</td>
</tr>
<tr>
<td>SIL 1</td>
<td>Entry Points</td>
</tr>
</tbody>
</table>

- Table B.2 7a - Structural test coverage (entry points) 100 % - Unfulfilled
- Table B.2 7b - Structural test coverage (statements) 100 % - Unfulfilled
- Table B.2 7c - Structural test coverage (branches) 100 % - Unfulfilled
- Table B.2 7d - Structural test coverage (conditions, MC/DC) 100 % - Unfulfilled
Entry Points

• This is the simplest and most basic structural coverage measurement
• Has every function been invoked at least once?
• Has every function been invoked from all the places where it is called?
Statement Coverage

- We can view the coverage on a flow graph:
Branch / Decision Coverage

- Branch / Decision coverage can be viewed on a flowgraph:
Branch / Decision Coverage

```c
560 1 if
561 1 (  
562 1  
563 1 ( void * ) 0 ) == scannedProduct
564 1 )

565 2 {
566 2  
567 2  
568 2 else

569 3 {
570 3  
571 3  
572 3  
573 3 }
```

Branch/Decision Coverage Profile

<table>
<thead>
<tr>
<th>LINE NUMBERS: REFORMATTED (SOURCE)</th>
<th>PREVIOUS RUNS</th>
<th>CURRENT RUN</th>
<th>COMBINED</th>
<th>CODE PRECEDING DECISION POINT</th>
</tr>
</thead>
</table>
| FROM TO                            |               | 0           | **0**    | (**void * ) 0 ) == scannedProduct |}
| 564 (180) 565 (181)               |               | 0           | **0**    | (**void * ) 0 ) == scannedProduct |}
| 564 (180) 569 (185)               |               | 0           | 1        | 1                            |
| 568 (184) 574 (188)               |               | 0           | **0**    | **0**                        | else
Modified Condition / Decision Coverage

MC/DC is a coverage measurement for multiple condition decisions. It does not require every possible combination to be executed.

If $n$ is the number of conditions, then a minimum of $n + 1$ combinations are required to achieve 100% coverage, as opposed to $2^n$ total combinations.

This only really comes into its own for 4 or more conditions as the number of combinations increases exponentially.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>MC/DC Combinations</th>
<th>BCCC Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>4096</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>1048576</td>
</tr>
</tbody>
</table>
MC/DC Example

• In this example, there are 6 conditions, and so a total of 64 possible combinations:

```c
void check_change_gear (int *cg, int tg, int *rpm, int *cp, int mc, int fp)
{
    if (((*rpm >= M1) && (tg > *cg)) || ((*rpm <= M2) && (tg < *cg)) ||
        (mc == 1) || (fp < M5))
    {
```

• To achieve MC/DC coverage on this example, a minimum of 7 combinations, each of which show conditions independently affecting the result, are needed

• The problem is, which 7 combinations?
The truth table shows all the combinations, and highlights for each condition all the pairs of combinations that independently affect the result. # means “Essential”, * means “Optional”
Pairs

• Executing the following combinations, shows the independence of C2:

\[
\begin{align*}
8 & \quad f( T, T, T, F, F, F ) = T \\
24 & \quad f( T, F, T, F, F, F ) = F \\
\end{align*}
\]

• An advantage of MC/DC is, that it does not matter which pair of combinations you use to show that a condition independently affects the result

• If there are several possible pairs, then this allows flexibility in choice of test data
Defensive Programming

It is rarely possible using just Dynamic Analysis, to achieve 100% Statement Coverage

For example, whenever there is any defensive programming. In these cases, unit testing could be done to get the statement coverage to 100%

LDRA recommends trying to achieve 100% Statement Coverage, with the exception that protective infeasible code should be retained

```c
471 (266) { 17 17 34
472 (267) if 17 17 34
473 (267) { 17 17 34
474 (268) ( c = getc ( input ) ) == EOF 17 17 34
475 (268) } 17 17 34
476 (268) { 0 *** 0 *** 0 ***
477 (269) break ; 0 *** 0 *** 0 ***
478 (270) } 0 *** 0 *** 0 ***
479 (271) window [ current_position + i ] = ( unsigned char ) c ; 17 17 34
480 (272) 17 17 34
481 (273) lookAhead_bytes = i ; 17 17 34
482 (274) 1 1 2
483 (274) 1 1 2
```
Object Code Coverage

• DO-178C section 6.4.4.2.b, states the following:

“Structural coverage analysis may be performed on the Source Code, object code, or executable Object Code. Independent of the code form on which the structural coverage analysis is performed, if the software level is A and a compiler, linker, or other means generates additional code that is not directly traceable to Source Code statements, then additional verification should be performed to establish the correctness of such generated code sequences.”
Sample “C” Code

• Consider the following simple C code:

```c
#include "misrac_types.h"
#include "specialoffer.h"

/*
 * Get the price which depends on which special offer, if any, is used
 */
LDRA_uint32_t SpecialOffer_getPrice(const LDRA_uint32_t aQuantity,
                                    const LDRA_uint32_t aUnitPrice, const tSpecialOffer anOffer)
{
    LDRA_uint32_t price;
    switch (anOffer)
    {
        case BUY_ONE_GET_ONE_FREE:
            price = aUnitPrice * ((aQuantity + 1U) >> 1U);
            break;
        case TEN_PERCENT_OFF:
            price = (aUnitPrice * aQuantity * 9U) / 10U;
            break;
        case THREE_FOR_ONE_EURO:
            price = ((aQuantity / 3U) * 100U) + ((aQuantity % 3U) * aUnitPrice);
            break;
        /* no offer */
        default:
            price = aUnitPrice * aQuantity;
            break;
    }
    return price;
}
```
The object code generated by a compiler will depend on the optimization setting, the compiler vendor, the target and a host of other issues.

The following is an example of the resulting assembler code generated by a widely used commercially available compiler with optimization disabled.
As we can see the structure of the generated assembler code is quite different to that of the C code.
Assembler Code Coverage

- There is one **decision** that is not covered
Linear Code Sequence and Jump (LCSAJ) Coverage

LCSAJ coverage gives the highest confidence in the code, but does require considerably more test cases.

It is highly recommended to maximise the LCSAJ coverage for the parts of the code that are the most safety critical.

100% LCSAJ coverage is not always possible since there are often several paths that are impossible. As a result the threshold is often set to less than 100%.
LCSAJ Coverage

- **Combined Coverage Run**
  - Statement Coverage: 99% (+99%) with a success limit of 100%
  - Branch/Decision Coverage: 79% (+79%) with a success limit of 100%
  - LCSAJ Coverage: 67% (+67%) with a success limit of 75%

- **Procedure / Function Call Coverage Profile**
  - 100% (+100%) with a success limit of 60%
LCSAJ Coverage

- Is a maintainability measurement
- Detects unreachable code
- Detects untestable (or infeasible) code
- Mandated for European safety critical military avionics projects
- Basis for test path coverage measure
- Highest attainable coverage metric for highest level of testing
LCSAJ Coverage

- Consider this example code:

```c
static float64_t getSpeed(const int32_t modeA, const int32_t modeB, const int32_t distance)
{
    float64_t speed;
    int32_t startTime;
    int32_t stopTime;

    if (modeA==0) {
        startTime = 100;
    } else {
        startTime = 200;
    }

    if (modeB==0) {
        stopTime = 200;
    } else {
        stopTime = 300;
    }

    speed = (float64_t) (distance) / (float64_t) (stopTime - startTime);
    return speed;
}
```
LCSAJ Coverage

• Consider this example code:

```c
static float64_t getSpeed(const int32_t modeA, const int32_t modeB, const int32_t distance)
{
    float64_t speed;
    int32_t startTime;
    int32_t stopTime;

    if (modeA==0) {
        startTime = 100;
    } else {
        startTime = 200;
    }

    if (modeB==0) {
        stopTime = 200;
    } else {
        stopTime = 300;
    }

    speed = ((float64_t) distance) / ((float64_t) (stopTime - startTime));
    return speed;
}
```
100% Statement & Branch Coverage

• With just two tests, we can get 100% statement and 100% branch coverage
100% Statement & Branch Coverage

- With just two tests, we can get 100% statement and 100% branch coverage

So is this code fully tested and safe?
• No! Even though we have exercised every statement and every branch, there are two other paths that we have not exercised

• One path results in the result being infinite!

This is why LCSAJs are important
• No! Even though we have exercised every statement and every branch, there are two other paths that we have not exercised.

• One path results in the result being infinite!

This is why LCSAJs are important.
Here is another example:

How many paths are there?

```c
static void identifyProduct ( const LDRA_uint32_t barcode )
{
    const struct Product * scannedProduct ;
    scannedProduct = ProductDatabase_getProduct ( barcode ) ;
    if (
        !
        scannedProduct
    )
    {
        Display_show ( "Unknown barcode" ) ;
    }
    else
    {
        addProduct ( scannedProduct ) ;
    }
}
```
There are 7 LCSAJs

```c
560Fstatic void
561T identifyProduct {
562F const LDRA_uint32_t aBarcode
563F {
564F const struct Product *
565F scannedProduct;
566T scannedProduct = ProductDatabase_getProduct ( aBarcode );
567T if
568T {
569T {
570T ( void * ) 0 ) == scannedProduct
571T }
572T {
573T Display_show ( "Unknown barcode" );
574T }
575T else
576T {
577T addProduct {
578T scannedProduct );
580T }
581T }
```
Data and Control Coupling Coverage

- DO-178C section 6.4.4.2 c states:
  “Analysis to confirm that the requirements-based testing has exercised the data and control coupling between code components”
Data and Control Coupling Coverage

- DO-178C section 6.4.4.2 c states:
  “Analysis to confirm that the requirements-based testing has exercised the data and control coupling between code components”

- Control coupling coverage is ensuring that every invocation of a function has been exercised
Data and Control Coupling Coverage

• DO-178C section 6.4.4.2 c states:
  “Analysis to confirm that the requirements-based testing has exercised the data and control coupling between code components”

• Control coupling **coverage** is ensuring that every invocation of a function has been exercised

• Data coupling **coverage** is ensuring that we have exercised every access to the data
Data Flow and Control Flow

- Control Flow Analysis – performed both on the program calling hierarchy and on the individual procedures.
- Data Flow Analysis – Follows variables through the source code and reports on anomalous use.
- Can be verified through reports, or graphically with call graphs and flow graphs.

```c
#include <stdio.h>
#include "Types.h"
#include "DataFlow.h"

void run ( uint32_t cycles )
{
    uint32_t pulse=0U;
    /* (M) DATAFLOW VIOLATION : 70 D : DU anomaly, variable value is not used. : pul
     */ /* See also line 17 DataFlow.c(DATAFLOW) */
    uint32_t iter;

    if ( cycles > 0U )
    {
        for ( iter=0U; iter<cycles; iter++ )
        {
            pulse++;
            /* (M) DATAFLOW VIOLATION : 70 D : DU anomaly, variable value is not used. : pul
             */ /* See also line 17 DataFlow.c(DATAFLOW) */
        }
    }
    /* (O) DATAFLOW VIOLATION : 7 D : DU data flow anomalies found. */
```
Structural Coverage Visualisation

• With Structural coverage, we can easily see which parts of the code have not been tested!

• Any code that is not executed is potentially a serious risk
Visualizing Structural Coverage

<table>
<thead>
<tr>
<th>Source</th>
<th>Percentage</th>
<th>Percentage Change</th>
<th>Success Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtwdemo_sil.c</td>
<td>Passed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combined Coverage Run</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statement Coverage</td>
<td>100</td>
<td>+ 15</td>
<td>100</td>
</tr>
<tr>
<td>Branch/Decision Coverage</td>
<td>100</td>
<td>+ 43</td>
<td>100</td>
</tr>
<tr>
<td>Modified Condition / Decision Coverage</td>
<td>100</td>
<td>+ 100</td>
<td>100</td>
</tr>
<tr>
<td>Current Coverage Run</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statement Coverage</td>
<td>96</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>Branch/Decision Coverage</td>
<td>93</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>Modified Condition / Decision Coverage</td>
<td>100</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>Previous Coverage Run</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statement Coverage</td>
<td>85</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>Branch/Decision Coverage</td>
<td>57</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>Modified Condition / Decision Coverage</td>
<td>0</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>

```
int main () {
    switch (i)
        case 4: break;
        default:
            Keyboard_end ();
            break;
        case 's':
            Keyboard_start ();
            break;
        case 'r':
            randomShopping ();
            break;
        case 'q':
            goodbye ();
            break;
        case '\n':
            /* ignore crlf */
        case 'r':
            /* ignore crlf */
            break;
```
LLR/SDD Driven Testing and Structural Coverage

100% Structural Coverage
HLR/SRD Driven Testing and SCA

Requirements  Design  Implementation  Unit Test  System Test  Deployment

HLR_0001  HLT_0001
HLR_0002  HLT_0001
.....  .....  HLT_0001
HLR_000n  HLT_000m

Uncovered Code

case LightSolo:
drain = 0;
break;
case Announcer:
drain = 5;
break;
case Guide:
drain = 6 * width * height;
break;
default:
drain = 5 + 6 * width * height;
break;
}

return drain;

"c":
    keyboard_cancel();
    break;
"e":
    keyboard_end();
    break;
"s":
    keyboard_start();
    break;
"r":
    randomShopping();
    break;
"q":
    goodbye();
    break;
"\n":
    /* ignore crlf */
case "\r":
    /* ignore crlf */
    break;
Aggregating Coverage from High and Low Level Testing

Requirements → Design → Implementation → Unit Test → System Test → Deployment

HLR_0001
HLR_0002
......
HLR_000n

LLR_0001
LLR_0002
......
LLR_000n

HLT_0001
HLT_0001
......
HLT_000m

LLT_0001
LLT_0001
......
LLT_000m

100% Structural Coverage

```c
case LightSolo:
    drain = 0;
    break;

case Announcer:
    drain = 5;
    break;

case Guide:
    drain = 6 * width * height;
    break;

default:
    drain = 5 + 6 * width * height;
    break;
}
return
    drain;
```
## Typical Decomposition within DOORS

<table>
<thead>
<tr>
<th>Requirement_Identifie</th>
<th>1 Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS_0010</td>
<td><strong>2 Reference Documents</strong></td>
</tr>
<tr>
<td></td>
<td><strong>3 Requirements</strong></td>
</tr>
<tr>
<td>SYS_0020</td>
<td><strong>3.1 Display</strong></td>
</tr>
<tr>
<td>SYS_0030</td>
<td><strong>3.2 Initialization and Configuration</strong></td>
</tr>
<tr>
<td>SYS_0040</td>
<td><strong>3.3 Output Calculation</strong></td>
</tr>
<tr>
<td>SYS_0050</td>
<td><strong>3.4 Photometer</strong></td>
</tr>
<tr>
<td>SYS_0060</td>
<td><strong>3.5 Cleanliness Factor</strong></td>
</tr>
<tr>
<td>SYS_0070</td>
<td><strong>3.6 Lighting Control Unit</strong></td>
</tr>
<tr>
<td></td>
<td><strong>3.7 Luminaries</strong></td>
</tr>
</tbody>
</table>

**3.1 Display**

The Tunnel Lighting System shall provide an human machine interface for emulation of input and examination of output data.

**3.2 Initialization and Configuration**

The Tunnel Lighting System shall be configurable via an external file and take into account tunnel dimensions, zones, spacing for signs, and efficiency factors.

**3.3 Output Calculation**

The Tunnel Lighting System shall manage output calculation in tiers so that lamps, cells, zones, and the tunnel can be controlled with precision.

**3.4 Photometer**

The Tunnel Lighting System shall include a photometer located near the entrance of the tunnel. The photometer will measure light intensity and be linked to the lighting control unit. Typically, such a meter will employ a spectral response close to that of the average human eye. Its reaction to changes in light levels is virtually instantaneous. The light receptor is used to measure average luminance (brightness) within an acceptance angle subtending 20° over a measurement range of 0-6500 cd/m².

**3.5 Cleanliness Factor**

The Tunnel Lighting System shall measure efficiency of lamps based on the number of days since cleaning the lamps.

**3.6 Lighting Control Unit**

The Tunnel Lighting System shall be controlled by a lighting control unit that will receive inputs from a power supply, a weather signal, and an external mounted photometer.

**3.7 Luminaries**

Luminaires shall be industrial units sealed to comply with IP66 specifications. Version 3.0 of the system requirements linked "down" to software.
## 1 Scope

The Tunnel Light examination of a tunnel. The tunnel is typically close to that of the instantaneous. The light receptors are angle subtending.

## 2 Reference Documents

## 3 Requirements

### 3.1 Starting display software

Upon execution of the Tunnel lighting system software, the software shall read the Tunnel.ini file and query the user for an input. Configuration files will provide the appropriate data for the example configuration of the system, as described in the document “Tunnel Lighting Control System Overview”.

### 3.2 Input option photometer normal

The software shall allow a real number value input for photometer input.

### 3.3 Input options photometer instantaneous

The software shall handle out of bound range.

### 3.4 Input options exit

The software shall allow the user to exit the application.

### 3.5 Input options days since cleaning nominal

The software shall allow the user to set the number of days since cleaning.

### 3.6 Input options power failure

The software shall allow the user to invoke all power states.

### 3.7 Display total cell demand

In the nominal power state, after entering a nominal range photometer input or nominal days since cleaning, the software shall display Total Cell demand and lumens per metre.

### 3.8 Display Lumens

In the nominal power state, after entering a nominal range photometer input or nominal days since cleaning, the software shall display lumens for each lamp, its power setting, and the lamp.
## Typical Decomposition within DOORS

<table>
<thead>
<tr>
<th>Requirement</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Scope</strong></td>
<td></td>
</tr>
<tr>
<td><strong>2 Reference Documents</strong></td>
<td></td>
</tr>
<tr>
<td><strong>3 Requirements</strong></td>
<td></td>
</tr>
</tbody>
</table>

### SYS_0010
- **3.1 Display**
  - The Tunnel Light
  - The Tunnel Light is an instrument for examining the Tunnel Light, which can be set to a specific state. Typically, it is used for testing the Tunnel Light, especially when the Tunnel Light is not in use. The light source can be set to a specific state, and the light can be set to a specific state.

### SYS_0020
- **3.2 Initialise**
  - The Tunnel Light is an instrument for examining the Tunnel Light, which can be set to a specific state. Typically, it is used for testing the Tunnel Light, especially when the Tunnel Light is not in use. The light source can be set to a specific state, and the light can be set to a specific state.

### SYS_0030
- **3.3 Output**
  - The Tunnel Light is an instrument for examining the Tunnel Light, which can be set to a specific state. Typically, it is used for testing the Tunnel Light, especially when the Tunnel Light is not in use. The light source can be set to a specific state, and the light can be set to a specific state.

### SYS_0040
- **3.4 Photon**
  - The Tunnel Light is an instrument for examining the Tunnel Light, which can be set to a specific state. Typically, it is used for testing the Tunnel Light, especially when the Tunnel Light is not in use. The light source can be set to a specific state, and the light can be set to a specific state.

### SYS_0050
- **3.5 Cleaning**
  - The Tunnel Light is an instrument for examining the Tunnel Light, which can be set to a specific state. Typically, it is used for testing the Tunnel Light, especially when the Tunnel Light is not in use. The light source can be set to a specific state, and the light can be set to a specific state.

### SYS_0060
- **3.6 Lighting**
  - The Tunnel Light is an instrument for examining the Tunnel Light, which can be set to a specific state. Typically, it is used for testing the Tunnel Light, especially when the Tunnel Light is not in use. The light source can be set to a specific state, and the light can be set to a specific state.

### SYS_0070
- **3.7 Luminaire**
  - The Tunnel Light is an instrument for examining the Tunnel Light, which can be set to a specific state. Typically, it is used for testing the Tunnel Light, especially when the Tunnel Light is not in use. The light source can be set to a specific state, and the light can be set to a specific state.

### Detailed design/Low level requirements linked “up” to software requirements

- **3.1 Instantiate Cell**
  - A cell shall be instantiated with zero types of lamps, zero maximum lumens, zero minimum lumens, zero for the cell ID, and zero for the cell size.

- **3.2 Initialise Cell**
  - A cell shall be initialised with the cell parameters as well as the lamps within the cell.

- **3.3 Set Emergency output level**
  - For emergency lighting, only the smallest lamp per luminaire shall be set to its defined emergency demand level to minimize power consumption.

- **3.4 Set PoweredOutputLevel**
  - Powered output settings shall be computed on a consistent setting for any given size of a lamp.

- **3.5 Calculate cell output**
  - The cell output shall be calculated as the number of lamps times the cell size.

- **3.6 Get Lamp Model Duo**
  - A Duo lamp model shall be applied if a lamp must be fitted by an exit sign and a siren.

- **3.7 Get Lamp Model Guide**
  - A Guide lamp model shall be applied if a lamp must be fitted with an exit sign.

- **3.8 Get Lamp Model Announcer**
  - An Announcer lamp model shall be applied if a lamp must be fitted with a siren.

- **3.9 Get Lamp Model LightSolo**
  - A Lightsolo lamp model shall be applied if a lamp is not fitted with a siren or an exit sign.

- **3.10 Get Data and Read Content**
Extending Traceability to Source Code

Reducing error-prone and time consuming exercise of coping function prototypes and tagging code

Code base is closer to executable object code in TBmanager as pre-processing phase has been completed

Easier to manage downstream changes in code and test to support updated traceability dynamic environments
Developing, Executing, and Reviewing Tests

Low Level test-case linked to LLR

Collection of results and reports

Traceability to verification tasks and requirements

Automated regression of test cases
### The Requirement, Test Case and Results

#### Visibility into requirements and test data at the point of test creation

<table>
<thead>
<tr>
<th>Value for Storage</th>
<th>Pass / Fail</th>
<th>Expected Value</th>
<th>Actual Value</th>
<th>Name</th>
<th>Type</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>I 0</td>
<td>Pass</td>
<td>0</td>
<td>0</td>
<td>mDaysSinceCleaning</td>
<td>Sint_32</td>
<td>Input member variable</td>
</tr>
<tr>
<td>I 182</td>
<td>Pass</td>
<td>182</td>
<td>182</td>
<td>mDaysBetweenCleaning</td>
<td>Sint_32</td>
<td>Input member variable</td>
</tr>
<tr>
<td>I 50</td>
<td>Pass</td>
<td>50</td>
<td>50</td>
<td>mSoiledEfficiency</td>
<td>Sint_32</td>
<td>Input member variable</td>
</tr>
<tr>
<td>U_0 1.0</td>
<td>PASS</td>
<td>1.0</td>
<td>1.000000e+000</td>
<td>% SoiledEfficiency</td>
<td>Float_64</td>
<td>Function result</td>
</tr>
</tbody>
</table>

#### Requirement Information

- **Requirement Number**: Low Level Test for LLR_0350 (Low Level Test)
- **Requirement Name**: Calculate and get soiling factor
- **Requirement Body**: Based on days between cleaning system data shall calculate and return soiling factor as a percentage of efficiency. (AutoGenerated verification task for Low Level Test)

#### Test Case Description

- **Name**: Soiling factor calculation for dirty lamp
  - **Description**: Verify that soiling factor is calculated correctly per formula described in LLR_0355 in cases where the lamp is not clear
  - **Inputs**:
    - mDaysSinceCleaning = 0
    - mDaysBetweenCleaning = 182
    - mSoiledEfficiency = 50
  - **Expected Output**: SoilingFactor = 1.0

---

**Test case results**

**Requirements text**

**Test case description**
Models and Verification (Rhapsody)

```c
rootState_active
}

 case on :
 { 
   res =
   on_processEvent ( ) ;
 }
 break ;
 case off :
 { 
   if ( IS_EVENT_TYPE_OF ( evOnOff_RadioPkg_id ) == 1 )
   {
     // [ state ROOT.off.(Exit)
     itsFrequency.copy ( itsCurrentWaveband ->
     getItsCurrentFrequency ( ) ) ;
     itsDisplay ->
   }
 ```
Returning Data Back to DOORS

3.39 System Data Initialisation
All system data tied to the tunnel lighting system as a whole shall be managed to provide data needed for system level data acquisition and decision making.

3.40 Calculate and get soiling factor
Based on days between cleaning system data shall calculate and return soiling factor as a percentage of efficiency.

3.41 System Data Query Get Lamp Power Required
For each lamp type power required shall be returned upon query.

<table>
<thead>
<tr>
<th>Requirement, verification task, and test case data</th>
<th>Pass/Fail data</th>
<th>Prototype information</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1.1.2.39.2 Test Case Identifier: TCI_2_351</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>Name: Soiling factor calculation for dirty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description: Verify that soiling factor is calculated correctly per formula described in LLR_0355 in cases where the lamp is not clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inputs:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mDaysSinceCleanning = 91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mDaysBetweenCleaning = 182</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoiledEfficiency = 50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected Output:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilingFactor = 1.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.1.1.2.39.3 Test Case Identifier: TCI_2_352</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>Name: Soiling factor calculation for dirty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description: Verify that soiling factor is calculated correctly per formula described in LLR_0355 in cases where the lamp is not clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inputs:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mDaysSinceCleanning = 182</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mDaysBetweenCleaning = 182</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoiledEfficiency = 50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected Output:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilingFactor = 2.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Determining the Right Ordering

<table>
<thead>
<tr>
<th>Scenario 1: HL -&gt; LL</th>
<th>Scenario 2: LL -&gt; HL</th>
<th>Scenario 3: LL -&gt; HL</th>
</tr>
</thead>
<tbody>
<tr>
<td>• HL for credit</td>
<td>• “Blind” low level testing with “throw away” coverage</td>
<td>• LL RBT for structural coverage soon after implementation phase</td>
</tr>
<tr>
<td>• Use Low Level RBT to fill coverage Gaps</td>
<td>• Formal verification with HL RBT for coverage</td>
<td>• Formally measure HL RBT coverage soon after</td>
</tr>
<tr>
<td>• Initial start on legacy code bases</td>
<td>• Create LL RBT with TBrunt to fill holes</td>
<td>• Utilize existing test cases to “fill gaps” that are difficult to reach with HL testing</td>
</tr>
<tr>
<td>• Smaller code bases for Level C applications</td>
<td>• Existing unit test can be leveraged for “inspiration”</td>
<td>• Improves traceability of low level requirements to code, requirements to LLT, and “testability” of code</td>
</tr>
<tr>
<td>• Near term time/resource constraints</td>
<td>• Code is known to be “coverable” and eases structural coverage analysis for uncovered code</td>
<td>• A little more time consuming upfront</td>
</tr>
<tr>
<td></td>
<td>• Tends to produce higher quality, easier to integrate and formally verify code</td>
<td>• Improves quality of LL requirements and traceability to SRD earlier in the process</td>
</tr>
</tbody>
</table>
## TBeXtreme Usage in Low-Level Testing

<table>
<thead>
<tr>
<th>Robustness</th>
<th>Supplemental</th>
<th>Rapid Unit Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Boundary conditions for types and conditionals</td>
<td>• Additional supplemental test cases can be generated once requirements based LL testing has been performed</td>
<td>• When no unit testing is performed due to cost constraints, TBeXtreme can be a low cost approach to get started. Coverage should be discarded.</td>
</tr>
<tr>
<td>• Null pointers, data structure overflows, and error conditions</td>
<td>• Usually this coverage is discarded but inputs are used for inspiration and insights for additional test/requirements creation</td>
<td>• During creation of low-level test, certain declarations may be needed. TBeXtreme can rapidly generate many of these declarations.</td>
</tr>
<tr>
<td>• “demonstrate the ability of the software to respond to abnormal inputs and conditions” (para. 6.4.2.2 DO-178B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• TBeXtreme can often uncover Robustness conditions so robustness requirements and test modeling can be updated</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Questions & Answers

Delivering Software Quality and Security through Test, Analysis & Requirements Traceability
For further information:

www.ldra.com  info@ldra.com

@ldra_technology  LDRA Software Technology  LDRA Limited

Delivering Software Quality and Security through Test, Analysis & Requirements Traceability