Path to IEEE 1012

Delivering Software Quality and Security through Test, Analysis & Requirements Traceability
• Liverpool Data Research Associates
• Founded 1975 with roots in nuclear power
• Provider of Test Tools & Solutions
• Metrics Pioneer ex: LCSAJ
• LDRA Certification Services
• Active participation in standards ex: DO-178C, MISRA C/C++, CERT
LDRA Standards Experience & Pedigree

• Professor Mike Hennell
  – Member of SC-205 / WG-71 (DO-178C) formal methods subgroup
  – Member of MISRA C and C++ committees
  – Member of the working group drafting a proposed secureC annex for the C language definition (SC 22 / WG 14)

• Bill St Clair
  – Member of SC-205 / WG-71 (DO-178C) Object-Oriented Technology subgroup

• Dr Clive Pygott
  – Member ISO software vulnerabilities working group (SC 22 / WG 23)
  – Member of MISRA C++ committee
  – Member of the working group drafting a proposed secureC annex for the C language definition (SC 22 / WG 14)

• Liz Whiting
  – Member of MISRA C committee

• Chris Tapp
  – Chairman of MISRA C++ committee, Member of MISRA C committee
LDRA in Power
Challenges on the Path to IEEE 1012

Objectives and project documents  Verification evidence and audit trail

Integration with targets  Tool qualification

Structural coverage (Functional and unit Tests)

Data and control flow analysis

Meeting coding standards

Traceability of requirements through code and test

Reduce Cost of Compliance

Manage distributed team

Reduce time to compliance and market
Requirements Traceability Across Software Levels

- System requirements allocated to Software

High-Level Requirements or Specification Model

- Source Code
- Review and Analysis Results
- SW Architecture

Low-Level Requirements or Design Model

- Test Procedures
- Test Results

- Test Cases
Linking Requirements to Design

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

- Requirements
- Design
- Implementation
- Unit Test
- System Test
- Deployment

Simulink®

Linked Requirements and Design
Linking Low Level Requirements Design and Implementation

LLR/Design

Source Code

C
C++
Ada
Java
Building and Maintaining Links

VISUAL LINKING AND LIVE BI-DIRECTIONAL TRACEABILITY
Requirements Management Tools
Traceability, Verification Workflow, & Test

Requirements Traceability
- Requirements traceability
- Impact Analysis
- Traceability Matrix
- Import/Export
- Report Generation

Verification Workflow
- Process Unification and Enforcement
- Verification Workflow Management
- Test Automation
- Objective Fulfillment
- Audit trail

LDRA Verification
- Coding Standards
- Unit Testing
- Code Coverage
- Quality Metrics
- Target Integration
- Data/Control Flow

REDUCING COST ACROSS THE LIFECYCLE
Automating Coding Standards Adherence

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

High Quality Software

Java → C/C++ → ADA

MISRA C/C++
CERT C/ CERT JAVA
CWE
High Integrity C++
HIS
IPA/SEC C
JSF++ AV
Netrino C
Coding Standards Adherence

Uniformly Designed and Implemented

- Coding standards enforcement
- Individual desktop usage, integration with eclipse and other IDEs
- Batch mode for peer review
- API to extract data and report on how you’d like
- TBevolve to compare progress over time
- Command line automation, Tie in to TBmanager for overall reporting and mapping to objectives
- Qualification package for coding standard and audit trail

Writing verifiable code

- No unreachable lines or branches
- No infeasible paths
- Smaller function sizes
- Manageable complexity metrics, loop and conditional nesting

Reducing cost of structural coverage

Consistency of product delivery cycles
Automation of Unit/Low-Level Testing

Requirements
Design
Implementation
Unit Test
System Test
Deployment

LLR_0001
LLR_0002
.....
LLR_000n

LLT_0001
LLT_0001
.....
LLT_000m

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;

100% Structural Coverage

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>91</td>
<td>mDaysSinceCleaning</td>
<td>Sint_32</td>
</tr>
<tr>
<td>182</td>
<td>mDaysBetweenCleaning</td>
<td>Sint_32</td>
</tr>
<tr>
<td>50</td>
<td>mSoiledEfficiency</td>
<td>Sint_32</td>
</tr>
<tr>
<td>1.5000000e+000</td>
<td>%</td>
<td>Float_64</td>
</tr>
</tbody>
</table>
Functional Testing & Structural Coverage

Requirements
Design
Implementation
Unit Test
System Test
Deployment

HLR_0001
HLR_0002
.....
HLR_000n

Uncovered Code

case 'c':
    Keyboard_cancel ();
    break;

case 'e':
    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;
Aggregating Coverage from High and Low Level Testing

100% Structural Coverage
Determining the right ordering

**Scenario 1: HL -> LL**
- HL for credit
- Use Low Level RBT to fill coverage gaps
- Initial start on legacy code bases
- Smaller code bases for Level C applications
- Near term time/resource constraints

**Scenario 2: LL -> HL**
- “Blind” low level testing with “throw away” coverage
- Formal verification with HL RBT for coverage
- Create LL RBT with TBrunt to fill holes
- Existing unit test can be leveraged for “inspiration”
- Code is known to be “coverable” and eases structural coverage analysis for uncovered code.
- Tends to produce higher quality, easier to integrate and formally verify code

**Scenario 3: LL -> HL**
- LL RBT for structural coverage soon after implementation phase
- Formally measure HL RBT coverage soon after
- Utilize existing test cases to “fill gaps” that are difficult to reach with HL testing.
- Improves traceability of low level requirements to code, requirements to LLT, and “testability” of code.
- A little more time consuming upfront
- Improves quality of LL requirements and traceability to SRD earlier in the process
Control Flow Analysis

- Control Flow Analysis - Assess the correctness of the software by diagramming the logical control. Examine the flow of the logic to identify missing, incomplete, or inaccurate requirements. Validate whether the flow of control among the functions represents a correct solution to the problem.
Data Flow Analysis

<table>
<thead>
<tr>
<th>Data flow analysis</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Use of data flow diagrams</td>
<td></td>
</tr>
<tr>
<td>• Flow balancing</td>
<td></td>
</tr>
<tr>
<td>• Compare output data and subsequent input and derived data from each process</td>
<td></td>
</tr>
<tr>
<td>to ensure data is available when required</td>
<td></td>
</tr>
<tr>
<td>• Confirmation of derived data</td>
<td></td>
</tr>
<tr>
<td>• Involves examining derived data within a process for correctness, format and</td>
<td></td>
</tr>
<tr>
<td>availability</td>
<td></td>
</tr>
<tr>
<td>• Keys to index comparison</td>
<td></td>
</tr>
<tr>
<td>• Compare data keys used to retrieve data from data stores within a process to</td>
<td></td>
</tr>
<tr>
<td>the database index design to confirm that no invalid keys have been used and</td>
<td></td>
</tr>
<tr>
<td>uniqueness properties are consistent</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(p)-&gt;cyclicBufferSize</th>
<th>MatchFinder_Init</th>
<th>P</th>
<th>R</th>
<th>278</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MatchFinder_SetLimits</td>
<td>P</td>
<td>R</td>
<td>249</td>
</tr>
<tr>
<td>(p)-&gt;dic</td>
<td>LzmaDec.c</td>
<td>P</td>
<td>D</td>
<td>954</td>
</tr>
<tr>
<td></td>
<td>LzmaDec_Allocate</td>
<td>P</td>
<td>D</td>
<td>957</td>
</tr>
<tr>
<td></td>
<td>LzmaDec_DecodeReal</td>
<td>P</td>
<td>R</td>
<td>141</td>
</tr>
<tr>
<td>(p)-&gt;dic</td>
<td>LzmaDec_DecodeToBuf</td>
<td>P</td>
<td>R</td>
<td>869</td>
</tr>
<tr>
<td></td>
<td>LzmaDec_FreeDict</td>
<td>P</td>
<td>D</td>
<td>888</td>
</tr>
<tr>
<td></td>
<td>LzmaDec_TryDummy</td>
<td>P</td>
<td>R</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>LzmaDec_WriteRem</td>
<td>P</td>
<td>R</td>
<td>432</td>
</tr>
<tr>
<td>(p)-&gt;dicBufSize</td>
<td>LzmaDec.c</td>
<td>P</td>
<td>R</td>
<td>954</td>
</tr>
<tr>
<td></td>
<td>LzmaDec_Allocate</td>
<td>P</td>
<td>D</td>
<td>964</td>
</tr>
<tr>
<td></td>
<td>LzmaDec_DecodeReal</td>
<td>P</td>
<td>R</td>
<td>142</td>
</tr>
</tbody>
</table>
LDRA Tool Suite Integrations

- Modeling Tools
- Compilers
- Processors
- IDE’s
- Communication Protocols
- Version Control
- RTOS
- Source Languages & Host Platforms
- Requirements
- RTM
- RTOS
Integrations with IDEs
The purpose of the tool qualification process is to obtain confidence in the tool functionality.

LDRA’s tool qualification packages are tailored per standard and software integrity level.
Application Life Cycle
For further information:

www.ldra.com  info@ldra.com

@ldra_technology  LDRA Software Technology  LDRA Limited