



Structural Coverage Analysis for Safety-Critical Code

What is Structural Coverage?

LDRA

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/Defence Avionics and Ground Systems
- IEC 61508 for Industrial Controls
- ISO 26262 for Automotive
- IEC 62304 for Medical Devices
- EN 50128 for Rail

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:

Design Assurance Level	Verification Level
A, B, C	Statement Coverage
A, B	Decision Coverage
A, B, C	Data Coupling & Control Coupling Coverage
A	Modified Condition / Decision Coverage
A	Object Code Coverage

- 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

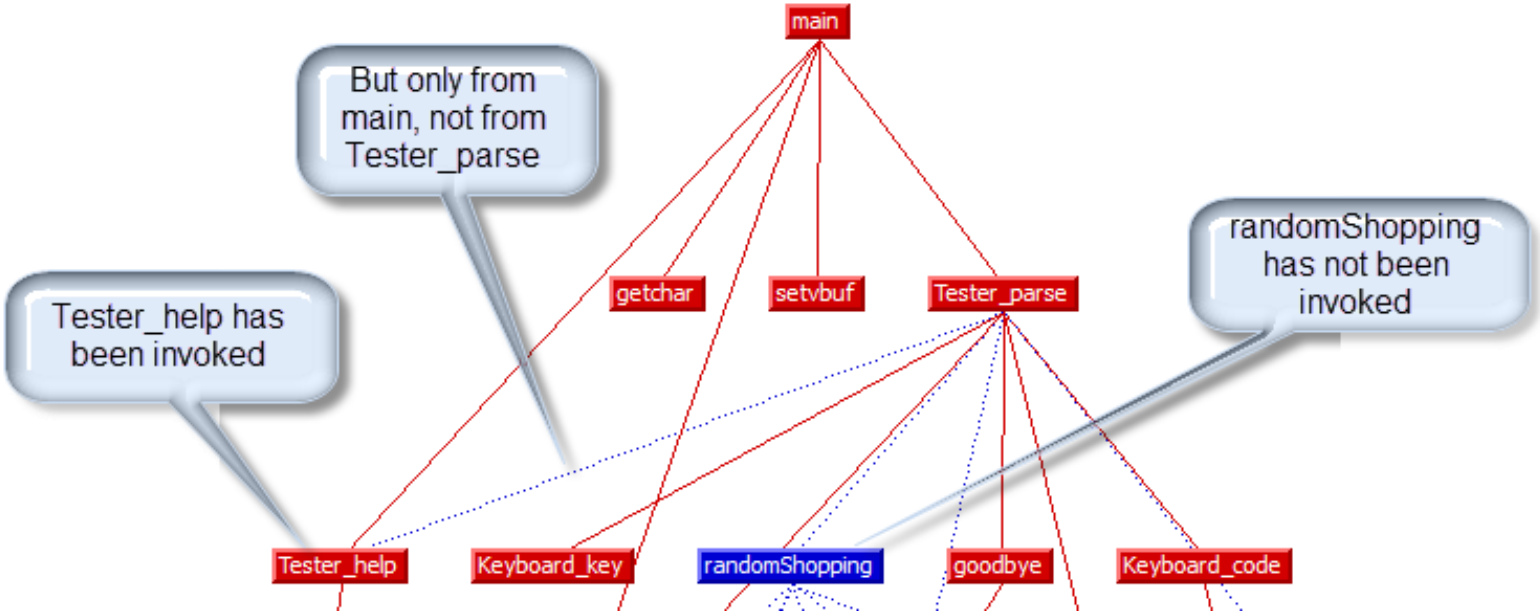
- The following Structural Coverage is required:

Safety Integrity Level	Verification Level
SIL 4	Entry Points + Statement + Branches + MC/DC
SIL 3	Entry Points + Statement + Branches
SIL 2	Entry Points + Statement
SIL 1	Entry Points

- 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

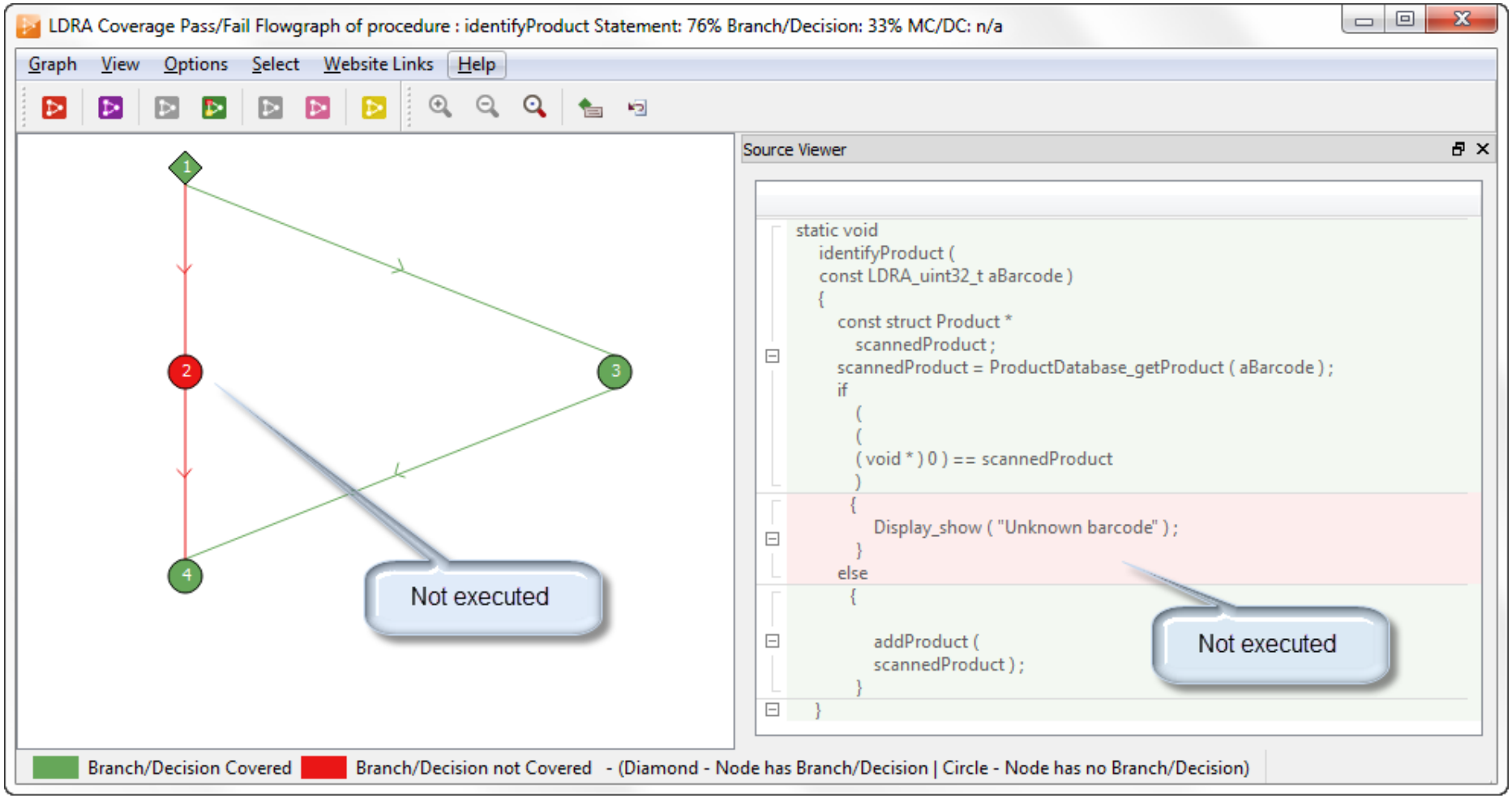
Entry Points

- 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?



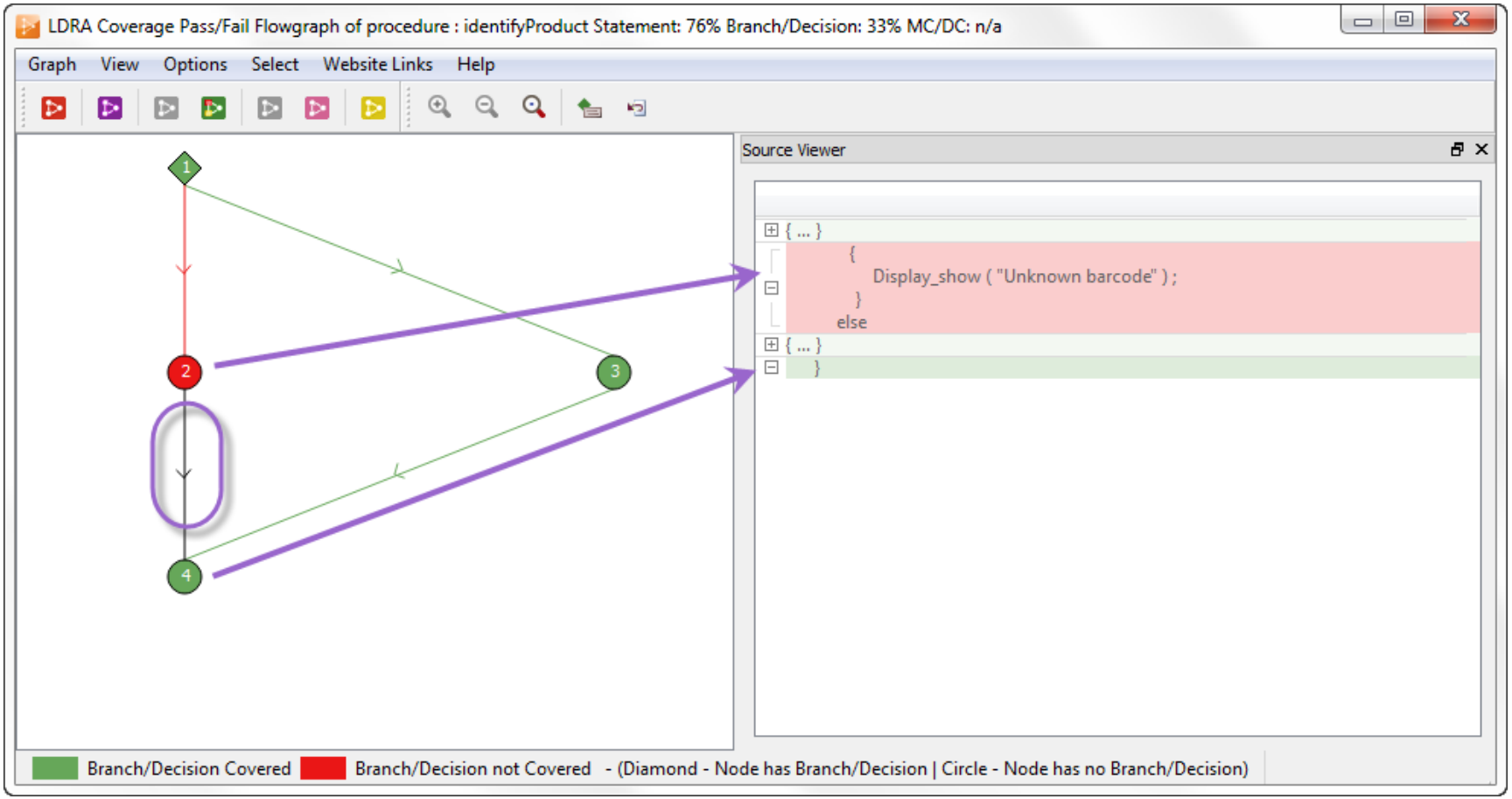
Statement Coverage

- We can view the coverage on a flow graph:



Branch / Decision Coverage

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



Branch / Decision Coverage

```
560     1         if
561     1            (
562     1            (
563     1            ( void * ) 0 ) == scannedProduct
564     1            )
-------------------------------------------------------------------
565     2            {
566     2               Display_show ( "Unknown barcode" ) ;
567     2            }
568     2         else
-------------------------------------------------------------------
569     3            {
570     3
571     3               addProduct (
572     3               scannedProduct ) ;
573     3            }
```

Branch/Decision Coverage Profile

LINE NUMBERS: FROM	REFORMATTED (SOURCE) TO	PREVIOUS RUNS	CURRENT RUN	COMBINED	CODE PRECEDING DECISION POINT
564 (180)	565 (181)	0	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

Conditions	MC/DC Combinations	BCCC Combinations
2	3	4
4	5	16
12	13	4096
20	21	1048576

MC/DC Example

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

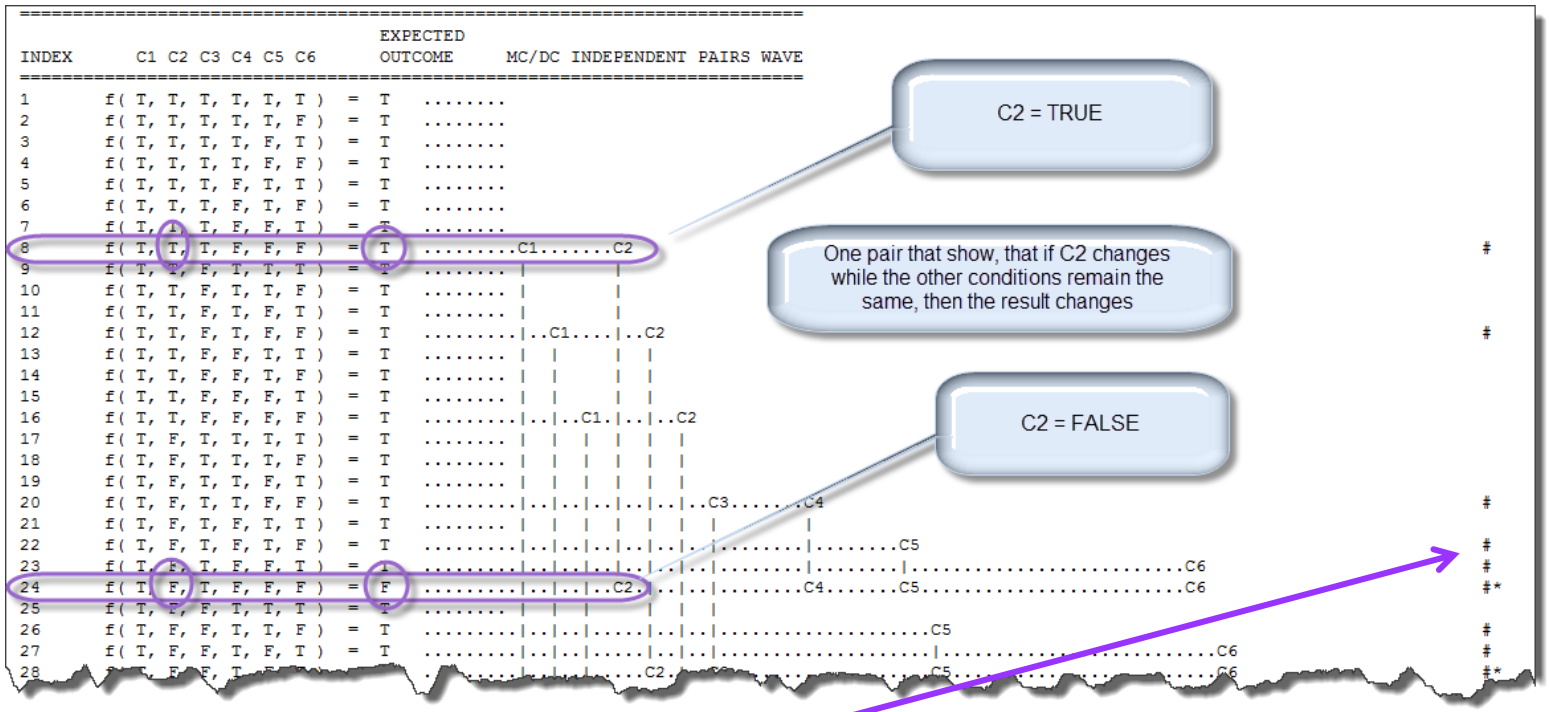
```
void check_change_gear (int *cg,int tg,int *rpm,int *cp,int mc,int fp)
{
   if (((*rpm >= M1) && (tg > *cg)) || ((*rpm <= M2) && (tg < *cg)) ||
       (mc == 1) || (fp < M5))
   {
```

- 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?

Decision Truth Table

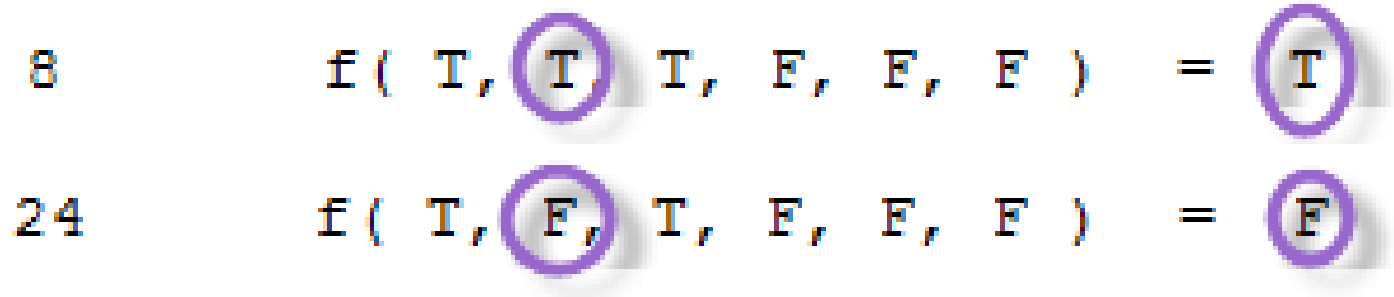
- The truth table shows all the combinations, and highlights for each condition all the pairs of combinations that independently affect the result



means "Essential", * means "Optional"

Pairs

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



- 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

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

Corresponding Assembler Code

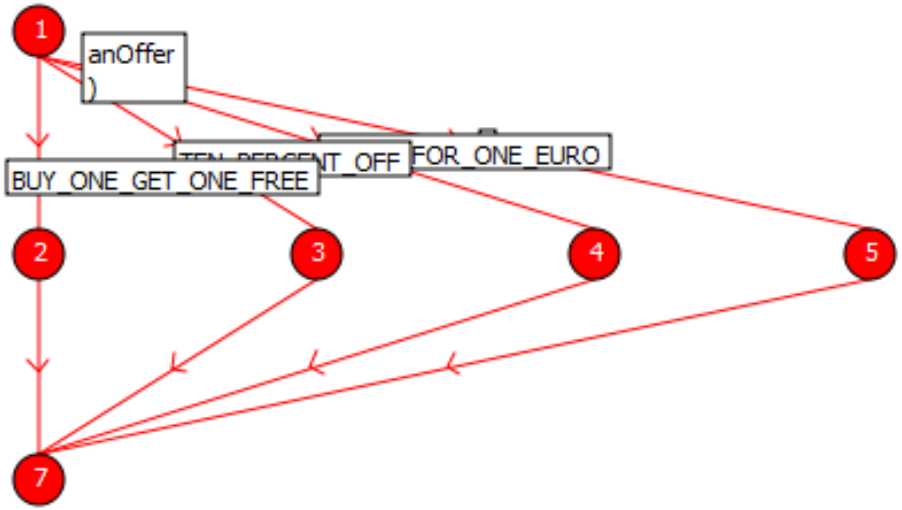
- The object code generated by a compiler will depend on the optimization setting, the compiler vendor, the target and a host of other issues

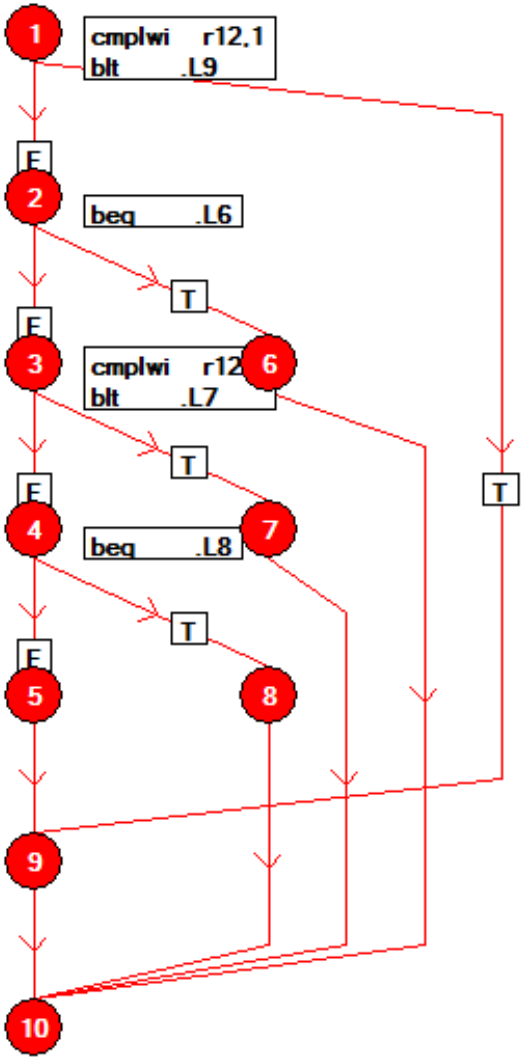
- The following is an example of the resulting assembler code generated by a widely used commercially available compiler with optimization disabled

```
#16: LDRA_uint32_t SpecialOffer_getPrice(const LDRA_uint32_t aQuantity,
      .text
..bof.C.3A.5CLDRA_Workarea.5CGHS_MULTI4_C_CashRegister_tbwrkfls.5CGHS_
      .align      2
      .globl      SpecialOffer_getPrice
SpecialOffer_getPrice:
      mr    r8, r3
      mr    r9, r4
      mr    r12, r5
#         .bf
.LDW01:
#17:     const LDRA_uint32_t aUnitPrice, const tSpecialOffer anOffer)
#18: {
#19:   LDRA_uint32_t price;
#20:   switch (anOffer)
      cmplwi      r12, 1
      blt   .L9
      beq   .L6
      cmplwi      r12, 3
      blt   .L7
      beq   .L8
      b     .L9
.L6:
#21:    {
#22:      case BUY_ONE_GET_ONE_FREE:
#23:        price = aUnitPrice * ((aQuantity + 1U) >> 1U);
#line23
..lin.C.3A.5CLDRA_Workarea.5CGHS_MULTI4_C_CashRegister_tbwrkfls.5CGHS_
.LDWlin1:
      addi  r12, r8, 1
      srwi  r12, r12, 1
      mullw r12, r9, r12
      b     .L4
.L7:
#24:        break;
#25:      case TEN_PERCENT_OFF:
#26:          /*LDRA_INSPECTED 96 S */
#27:        price = (aUnitPrice * aQuantity * 9U) / 10U;
      mullw r12, r9, r8
      mr    r11, r12
      slwi  r12, r11, 3
      add   r11, r12, r11
      li    r12, 10
      divwu r12, r11, r12
      b     .L4
.L8:
#28:        break;
#29:      case THREE_FOR_ONE_EURO:
#30:          /*LDRA_INSPECTED 96 S */
```

Assembler Code Structure

- 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

LINE NUMBER REF. (SOURCE)	STATEMENT			PREVIOUS RUNS	CURRENT RUN	COMBINED
34 (31)	SpecialOffer_getPrice:			4	4	8
35 (32)		mr	r8,r3	4	4	8
36 (33)		mr	r9,r4	4	4	8
37 (34)		mr	r12,r5	4	4	8
38 (36)	# .bf			-	-	-
39				-	-	-
40	.LDW01:			4	4	8
41 (38)	#17: const LDRA_uint32_t aUnitPrice, const tSpecialOffer anOffer)			-	-	-
42 (39)	#18: {			-	-	-
43 (40)	#19: LDRA_uint32_t price;			-	-	-
44 (41)	#20: switch (anOffer)			-	-	-
45		cmplwi	r12,1	4	4	8
46 (42)		blt	.L9	4	4	8
47 (43)		beq	.L6	3	3	6
48 (44)		cmplwi	r12,3	2	2	4
49 (45)		blt	.L7	2	2	4
50 (46)		beq	.L8	1	1	2
51 (47)		b	.L9	0 ***	0 ***	0 ***
52 (48)				-	-	-
53	.L6:			1	1	2
54 (50)	#21: {			-	-	-
55 (51)	#22: case BUY_ONE_GET_ONE_FREE:			-	-	-
56 (52)	#23: price = aUnitPrice * ((aQuantity + 1U) >> 1U);			-	-	-
57 (53)	#line23			-	-	-
58 (54)	#.lin.C.3A.5CLDRA_Workarea.5CGHS_MULTI4_C_CashRegister_tbwrkfls.5CGHS			-	-	-
59				-	-	-
60	.LDWlin1:			1	1	2
61 (55)		addi	r12,r8,1	1	1	2
62 (56)		srwi	r12,r12,1	1	1	2
63 (57)		mullw	r12,r9,r12	1	1	2
64 (58)		b	.L4	1	1	2
65 (59)				-	-	-
66	.L7:			1	1	2
67 (61)	#24: break;			-	-	-
68 (62)	#25: case TEN_PERCENT_OFF:			-	-	-
69 (63)	#26: /*LDRA_INSPECTED 96 S */			-	-	-
70 (64)	#27: price = (aUnitPrice * aQuantity * 9U) / 10U;			-	-	-
71		mullw	r12,r9,r8	1	1	2
72 (65)		mr	r11,r12	1	1	2

Linear Code Sequence and Jump (LCSAJ) Coverage

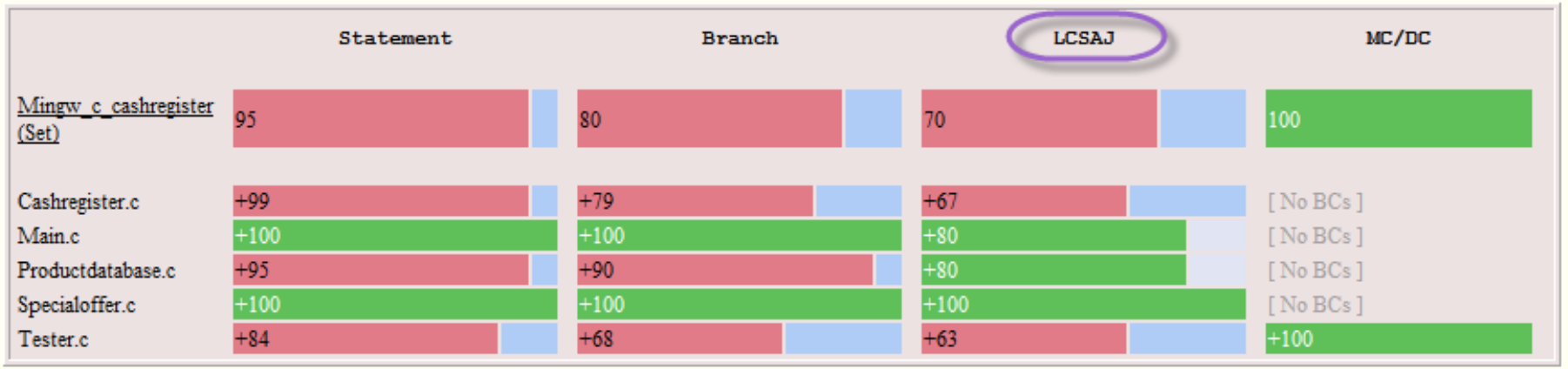
LDRA

- 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

Code Coverage : Cashregister.c : Dynamic Configuration - Custom

	Percentage	Percentage Change	Success Limit
Cashregister.c			
Combined Coverage Run	Failed		
Statement Coverage	99	+ 99	100
Branch/Decision Coverage	79	+ 79	100
LCSAJ Coverage	67	+ 67	75
Modified Condition / Decision Coverage			
Procedure / Function Call Coverage Profile	100	+ 100	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

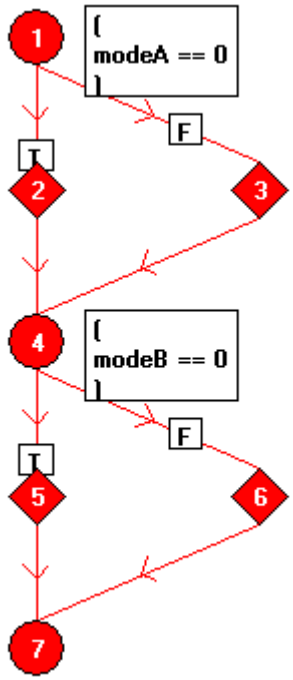
LCSAJ Coverage

- Consider this example code:

```
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;
}
```

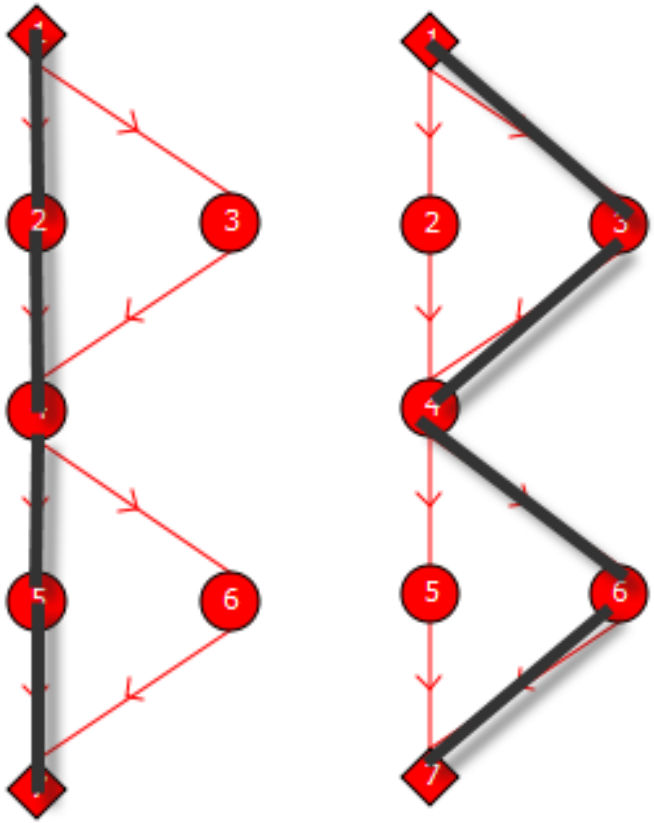


100% Statement & Branch Coverage

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

Test Case	Regression P / F	Procedure	Value	Name
1	PASS	getSpeed	0	modeA
2	PASS	getSpeed	0	modeB
			2000	distance
			2.000000e+001	%

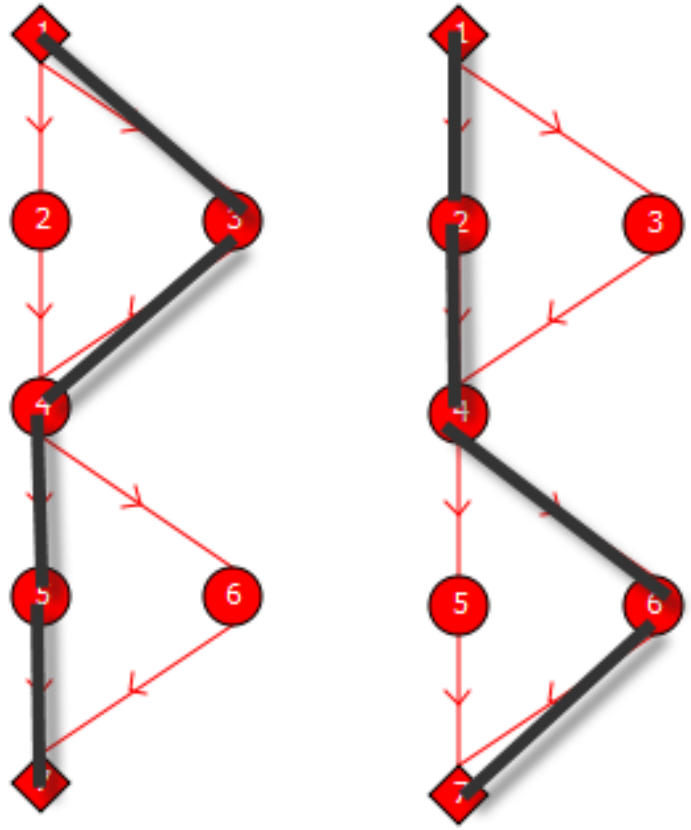
Test Case	Regression P / F	Procedure	Value	Name
1	PASS	getSpeed	1	modeA
2	PASS	getSpeed	1	modeB
			25000	distance
			2.500000e+002	%



So is this code fully tested and safe?

NO!

- No! Even though we have exercised every statement and every branch, there are two othe paths that we have not exercisec
- One path results in the result being infinite!



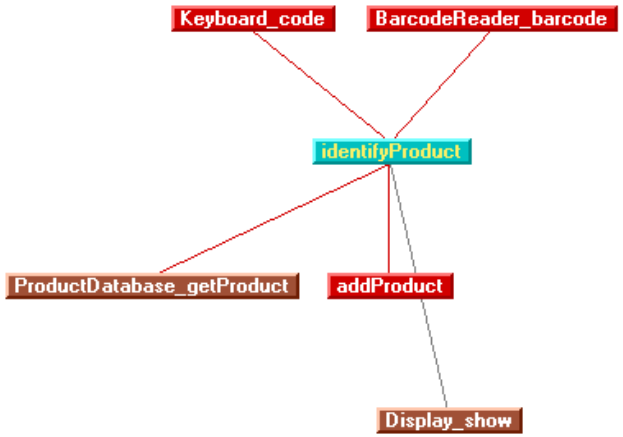
Test Case	Regression P / F	Procedure	Value	Name
1	PASS	getSpeed	0	modeA
2	PASS	getSpeed	1	modeB
			25000	distance
			1.250000e+002	%

Test Case	Regression P / F	Procedure	Value	Name
1	PASS	getSpeed	1	modeA
2	PASS	getSpeed	0	modeB
			25000	distance
			Infinite	%

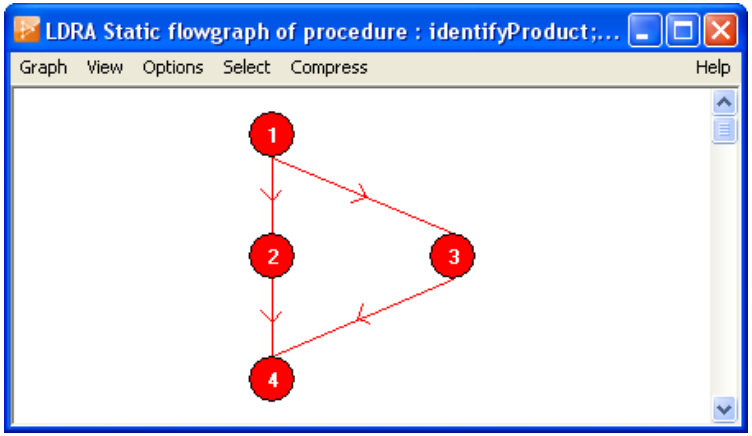
This is why LCSAJs are important

LCSAJ

- Here is another example:
- How many paths are there?

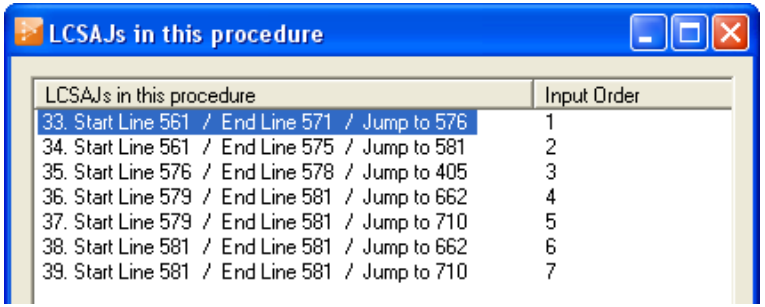


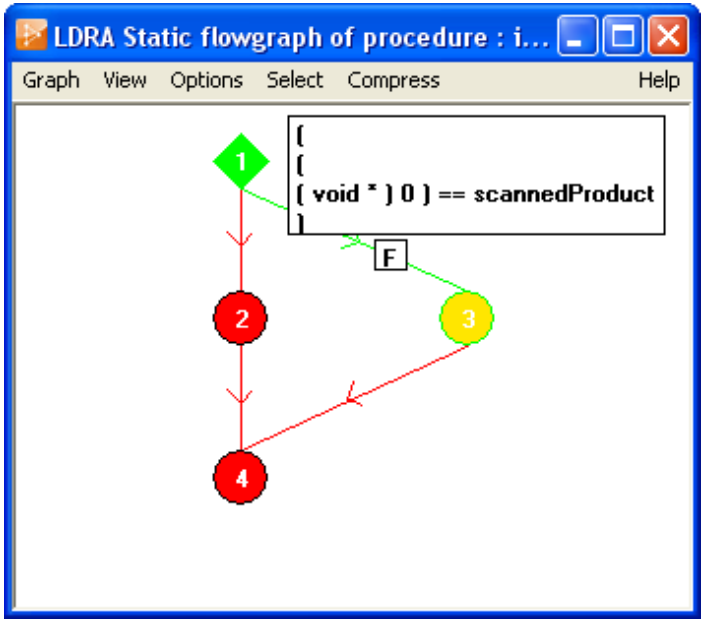

```
560Fstatic void
561T  identifyProduct (
562F  const LDRA_uint32_t aBarcode )
563F  {
564F    const struct Product *
565F      scannedProduct ;
566T    scannedProduct = ProductDatabase_getProduct ( aBarcode ) ;
567T    if
568T      (
569T      (
570T      ( void * ) 0 ) == scannedProduct
571T      )
572T      {
573T        Display_show ( "Unknown barcode" ) ;
574T      }
575T    else
576T      {
577T
578T        addProduct (
579T        scannedProduct ) ;
580T      }
581T  }
```



There are 7 LCSAJs

```
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
578T        addProduct (
579T        scannedProduct ) ;
580T      }
581T  }
```

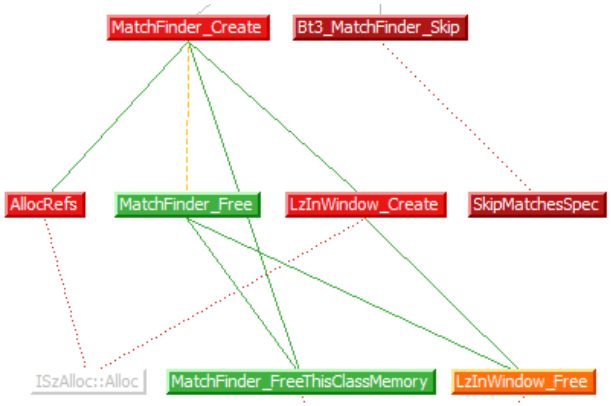




Data and Control Coupling Coverage

- 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”
- 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



(p)->cyclicBufferSize		MatchFinder_Init	P	R	278	
		MatchFinder_SetLimits	P	R	249	
(p)->dic	LzmaDec.c	LzmaDec_Allocate	P	R	954	958
			P	D	957	
		LzmaDec_DecodeReal	P	R	141	178
(p)->dic		LzmaDec_DecodeToBuf	P	R	869	
		LzmaDec_FreeDict	P	R	888	
			P	D	889	
		LzmaDec_TryDummy	P	R	512	522
		LzmaDec_WriteRem	P	R	432	
(p)->dicBufSize	LzmaDec.c	LzmaDec_Allocate	P	R	954 *****	
			P	D	964	
		LzmaDec_DecodeReal	P	R	142	

Data Flow and Control Flow

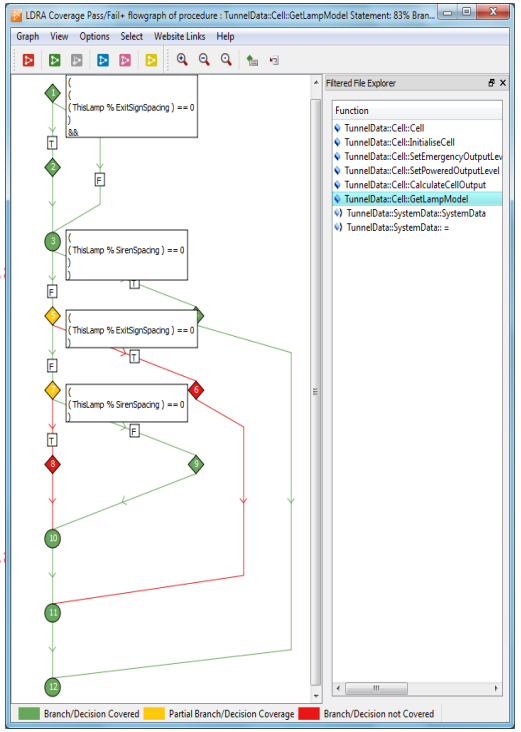
- Control Flow Analysis – performed both on the program calling hierarchy and on the individual procedures.
- Data Flow Analysis – Follows variables through the source code and reports on anomalous use.
- Can be verified through reports, or graphically with call graphs and flow graphs.

```
#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. */
```

pulse is Defined but Unused



Structural Coverage Visualisation

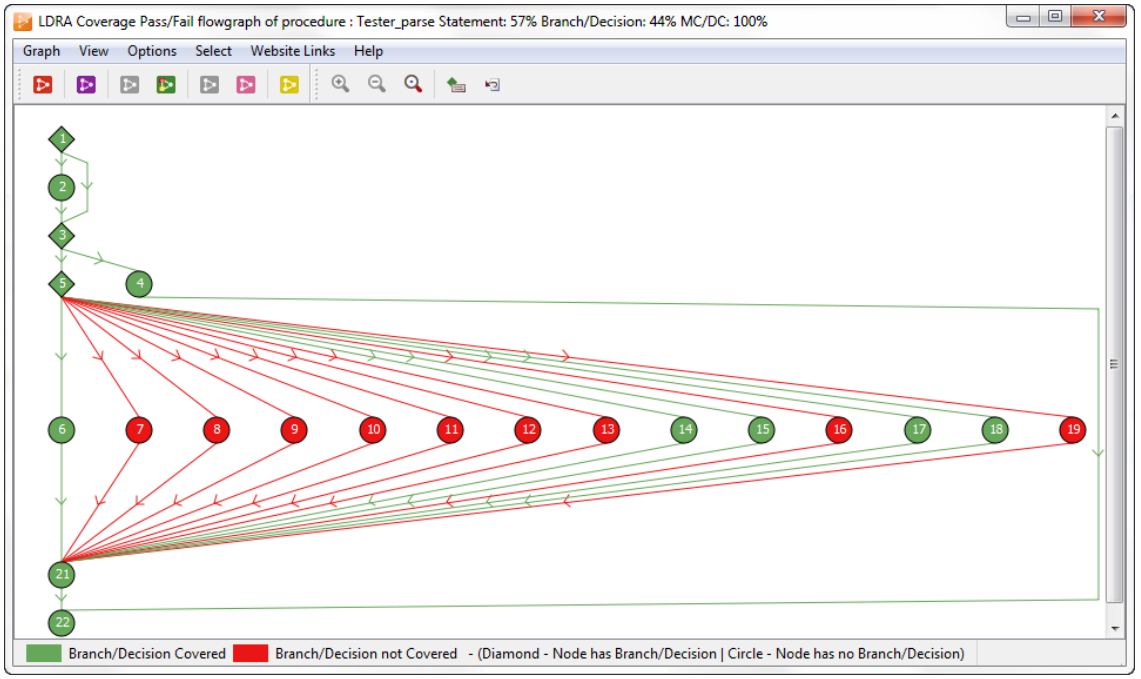
- With Structural coverage, we can easily see which parts of the code have not been tested!
- Any code that is not executed is potentially a serious risk

```
while
  (
  curPos < limit
  )
  {
    if
      (
      p -> hashBufPos == p -> hashBufPosLimit
      )
      {

        MatchFinderMt_GetNextBlock_Hash (
        p ) ;
        distances [ 1 ] = numProcessed + p -> hashNumAvail ;
        if
          (
          p -> hashNumAvail >= p -> numHashBytes
          )
          {
            continue ;
          }
        for
          (
          ;
          p -> hashNumAvail != 0
          ;
          p -> hashNumAvail --
          )
          {
            distances [ curPos ++ ] = 0 ;
          }
        break ;
      }
```

Visualizing Structural Coverage

	Percentage	Percentage Change	Success Limit
rtwdemo_sil.c			
Combined Coverage Run	Passed		
Statement Coverage	100	+ 15	100
Branch/Decision Coverage	100	+ 43	100
Modified Condition / Decision Coverage	100	+ 100	100
Current Coverage Run			
Statement Coverage	96		100
Branch/Decision Coverage	93		100
Modified Condition / Decision Coverage	100		100
Previous Coverage Run			
Statement Coverage	85		100
Branch/Decision Coverage	57		100
Modified Condition / Decision Coverage	0		100




```
  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 Verification SCA

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

VT: Unit Test for REQ_0002 (VG_DYN), Cleanliness Factor - Cody, Bill
- Input: Tunnel_STP.docx (Asset)
- TBrun Regression Report Artifact Placeholder fulfilled by 1 item
- TCI_1_290: 0 days since cleaning
- TCI_2_24: 91 days since cleaning -- Integration Unit/Module Test

Requirments Driven Low-Level Test

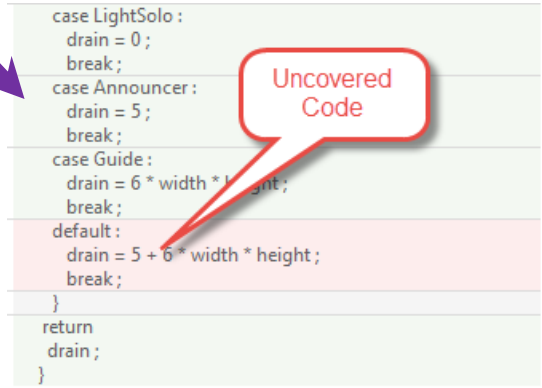
Value	Name	Type
I 91	mDaysSinceCleaning	Sint_32
I 182	mDaysBetweenCleaning	Sint_32
I 50	mSoiledEfficiency	Sint_32
◇ 1.500000e+000	%	Float_64

HLR/SRD/SRS Verification and SCA

Requirements | Design | Implementation | Unit Test | System Test | Deployment

HLR_0001
HLR_0002
…..
HLR_000n

HLT_0001
HLT_0001
…..
HLT_000m




```
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

HLR_0001
HLR_0002
…..
HLR_000n

HLT_0001
HLT_0001
…..
HLT_000m

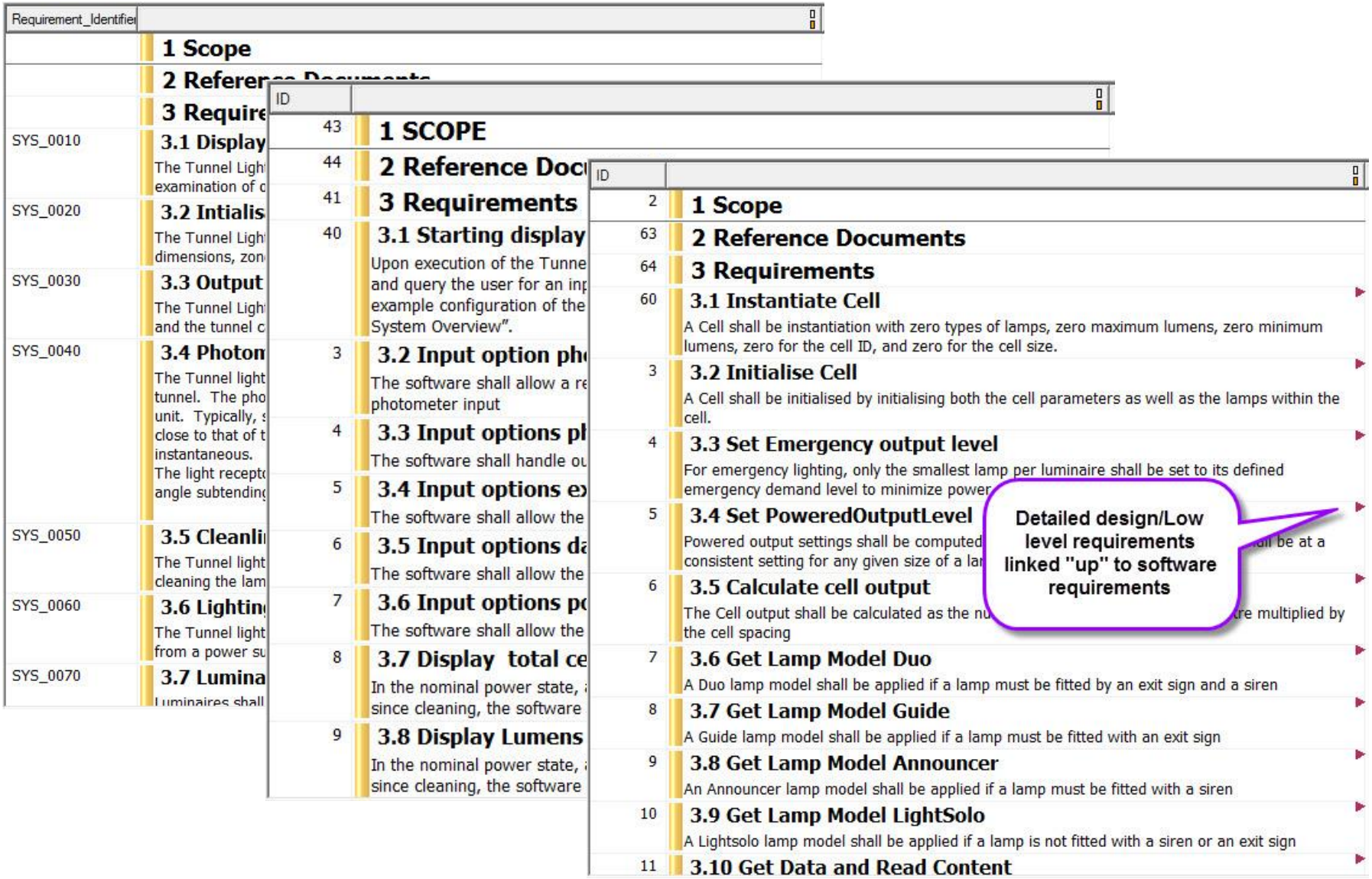
LLR_0001
LLR_0002
…..
LLR_000n

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

Multiple Levels of Requirements (DOORS Ex.)

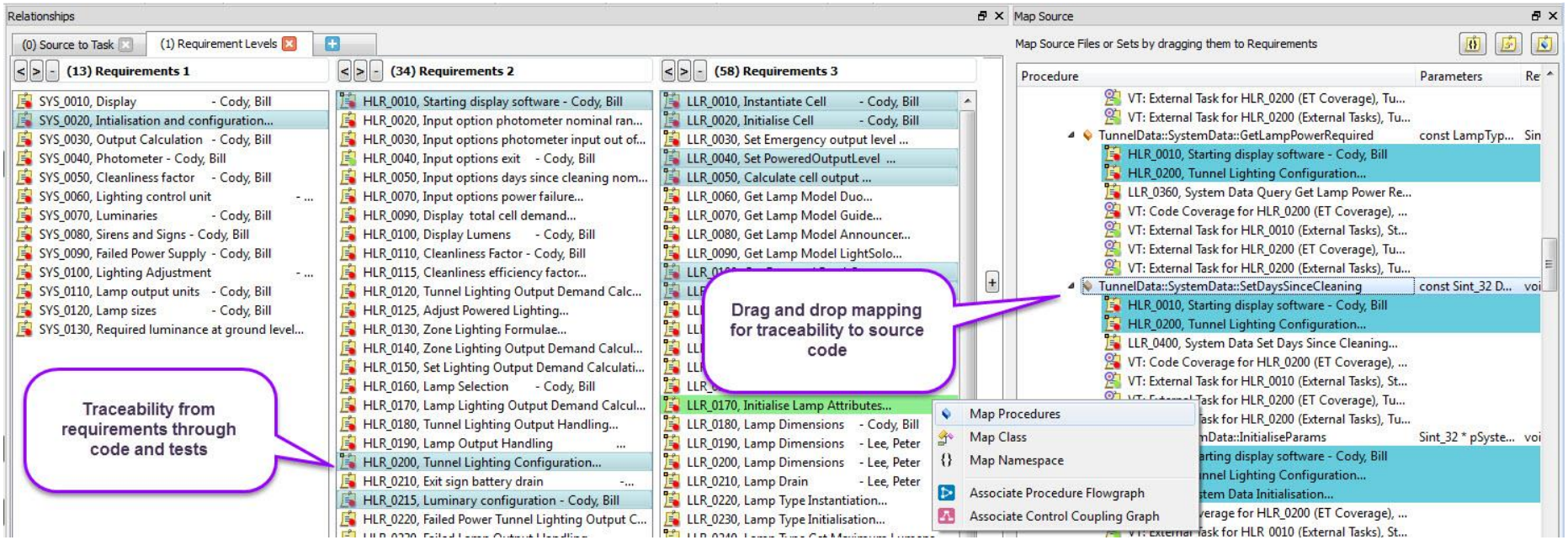


Extending Traceability to Source Code

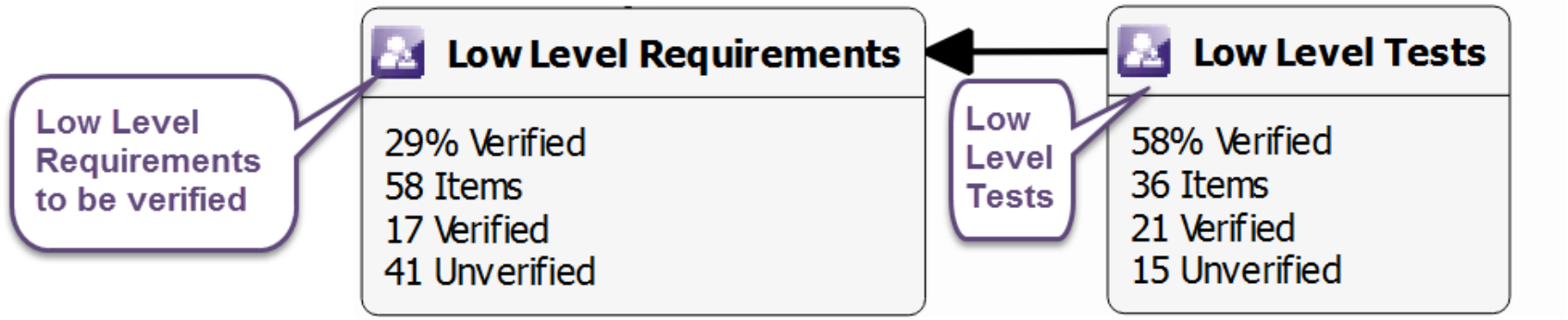
Reducing error-prone and time consuming exercise of coping function prototypes and tagging code

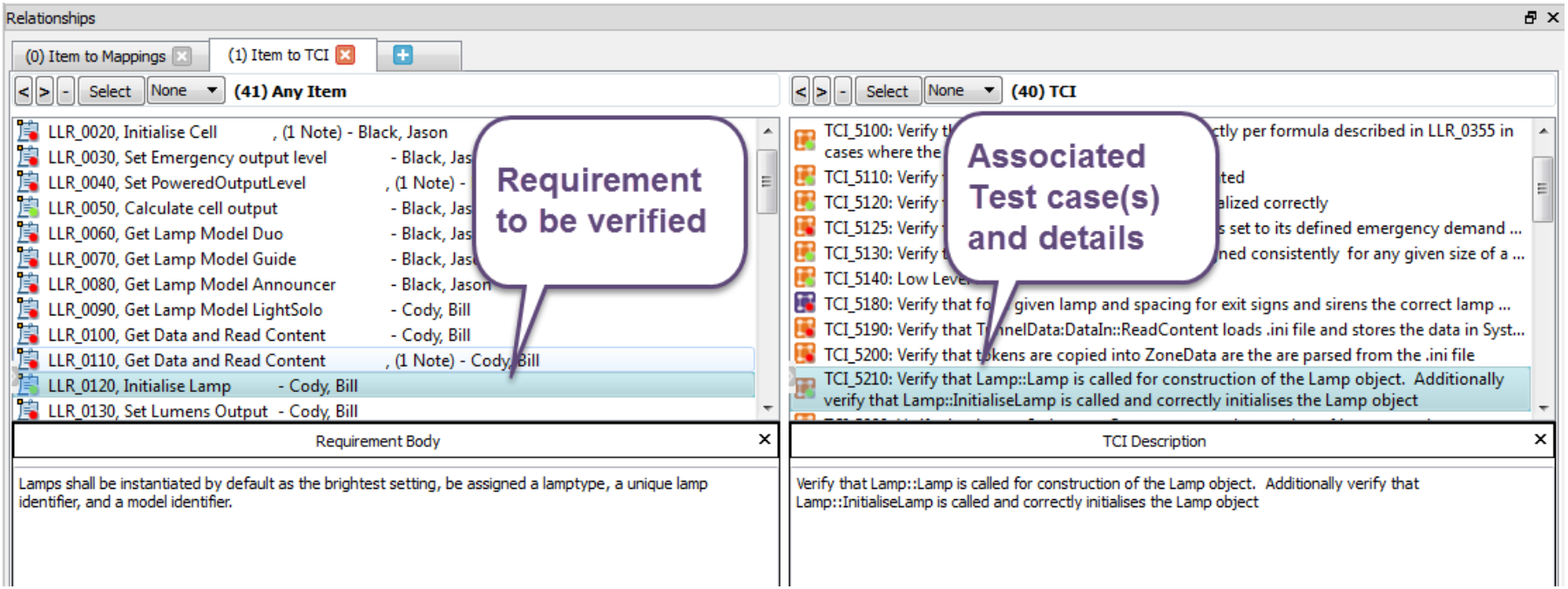
Code base is closer to executable object code in TBmanager as pre-processing phase has been completed

Easier to manage downstream changes in code and test to support updated traceability dynamic environments

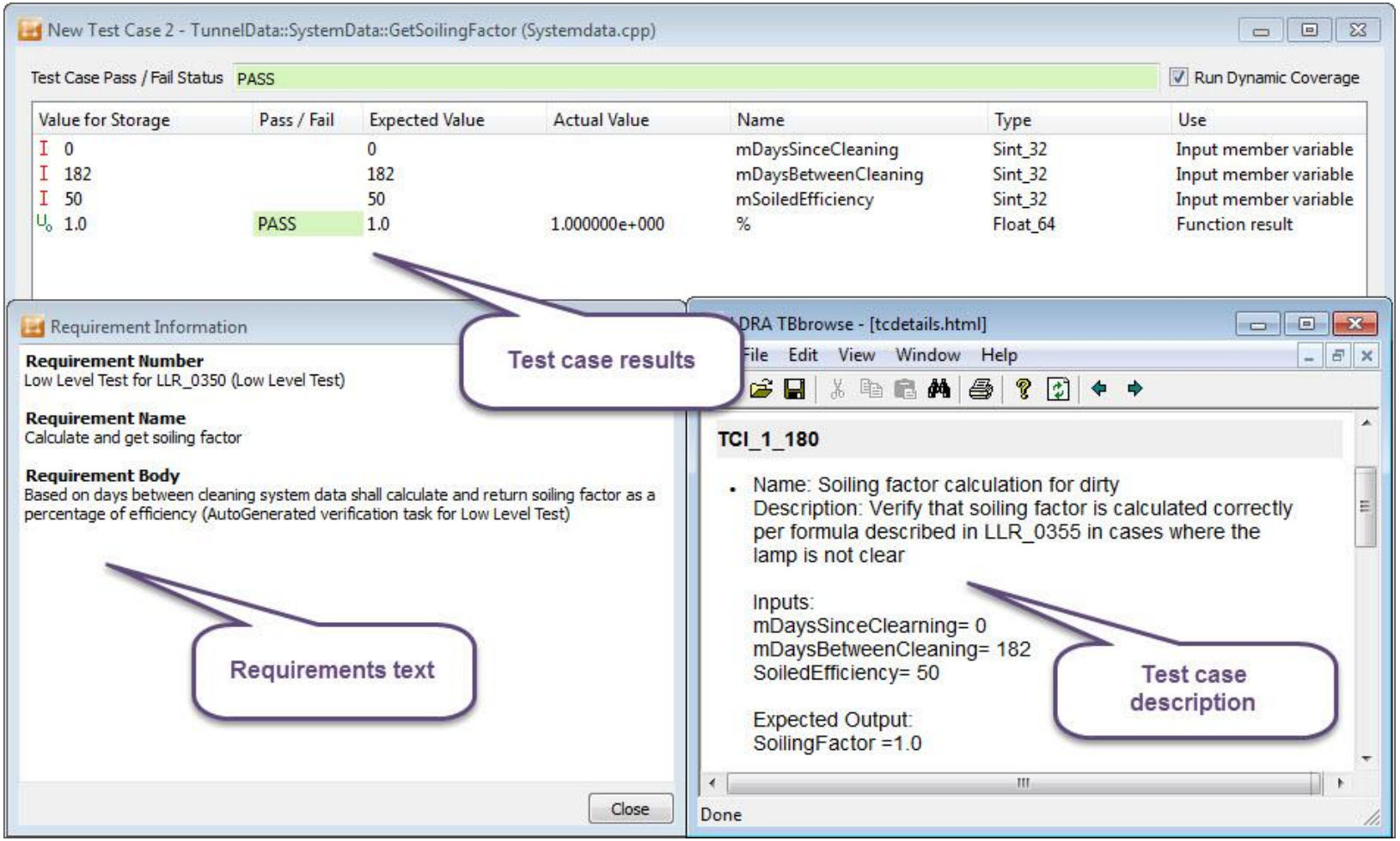


Developing, Executing, and Reviewing Tests



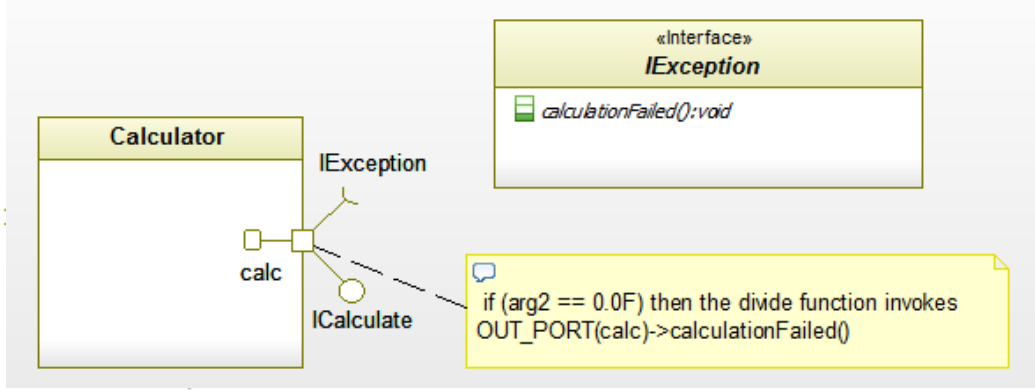


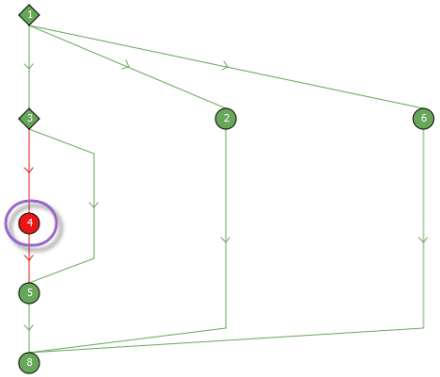
The Requirement, Test Case and Results



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

Models and Verification (Rhapsody)



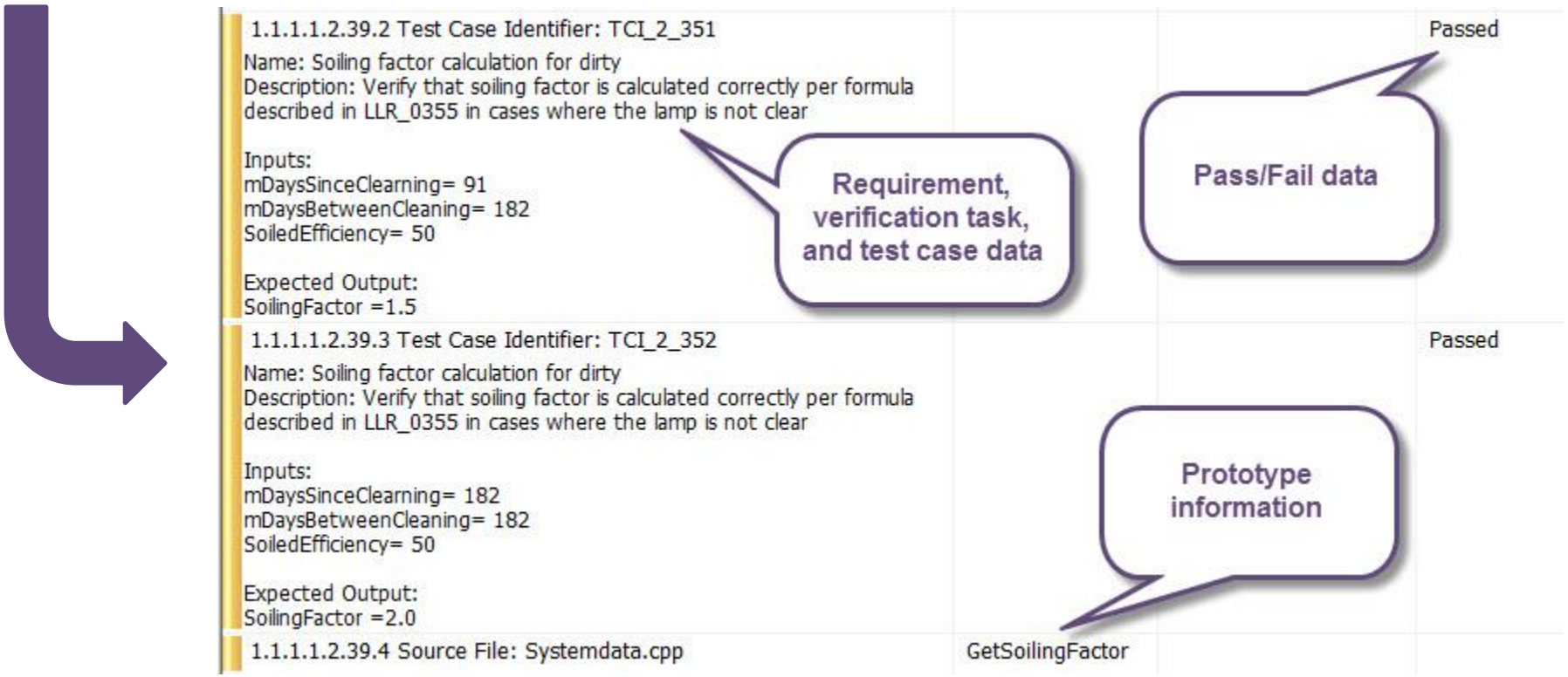


```
rootState_active
)
{
case on :
  {
    res =
      on_processEvent ( ) ;
  }
  break ;
case off :
  {
    if
      (
      IS_EVENT_TYPE_OF ( evOnOff_RadioPkg_id ) == 1
      )
      {
                //#[ state ROOT.off.(Exit)
        itsFrequency.copy ( itsCurrentWaveband ->
                  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

91	PASS	RadioPkg::Radio::rootState_processEvent
92	PASS	RadioPkg::Radio::rootState_processEvent
93	PASS	RadioPkg::Radio::rootState_processEvent

Returning Data Back to DOORS

SDD_40	**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
SDD_41	**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
SDD_42	**3.41 System Data Query Get Lamp Power Required** For each lamp type power required shall be returned upon query



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 TBrun 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

TBeXtreme Usage in Low-Level Testing

Robustness

- Boundary conditions for types and conditionals
- Null pointers, data structure overflows, and error conditions
- **“demonstrate the ability of the software to respond to abnormal inputs and conditions” (para. 6.4.2.2 DO-178B)**
- TBeXtreme can often uncover Robustness conditions so robustness requirements and test modeling can be updated

Supplemental

- Additional supplemental test cases can be generated once requirements based LL testing has been performed
- Usually this coverage is discarded but inputs are used for inspiration and insights for additional test/requirements creation

Rapid Unit Testing

- When no unit testing is performed due to cost constraints, TBeXtreme can be a low cost approach to get started. Coverage should be discarded.
- During creation of low-level test, certain declarations may be needed. TBeXtreme can rapidly generate many of these declarations.

For further information:

www.ldra.com info@ldra.com

@ldra_technology LDRA Software Technology LDRA Limited