Data and Control Coupling

Introduction

This document describes the features of the LDRA tool suite which contribute to the **DO178B/C** Data Coupling and Control Coupling objectives. In this section the definitions and descriptions of these terms from the DO178C standard are reproduced and the next section describes some of the faults that these objectives are designed to detect. The third section describes how the LDRA tool suite contributes to the certification process for these objectives.

The definition of the relevant terms in the glossary of DO178C is:

> **Data coupling** – The dependence of a software component on data not exclusively under the control of that software component.
>
> **Control coupling** – The manner or degree by which one software component influences the execution of another software component.

In addition the definition of component is:

> **Component** – A self contained part, combination of parts, subassemblies, or units that perform a distinct function of a system.

In general this term is interpreted as including: procedures, functions, subroutines, modules and other similar programming constructs.

The objective which is required to be achieved, table A-7 objective 8, is:

> 6.4.4 d. Test coverage of software structure, both data coupling and control coupling, is achieved.

Then in 6.4.4.2.c it further clarifies the activity:

> c. Analysis to confirm that the requirements-based testing has exercised the data and control coupling between code components.

Examples of clarifications between DO178B and DO178C include:

i. Clarified that the structural coverage analysis of data and control coupling between code components should be achieved by assessing the results of the requirements-based tests (see 6.4.4.2.c).

It is an unfortunate fact that these definitions do not uniquely identify the actual checks to be made. It is clear that they are targeted at faults which will not be detected by the other, better known coverage techniques.

In the following discussion the concepts will be explored in detail in order to formulate practical techniques to achieve these objectives. Then the contribution of the LDRA tool suite to these techniques and hence each of the objectives will be discussed.

The Concepts

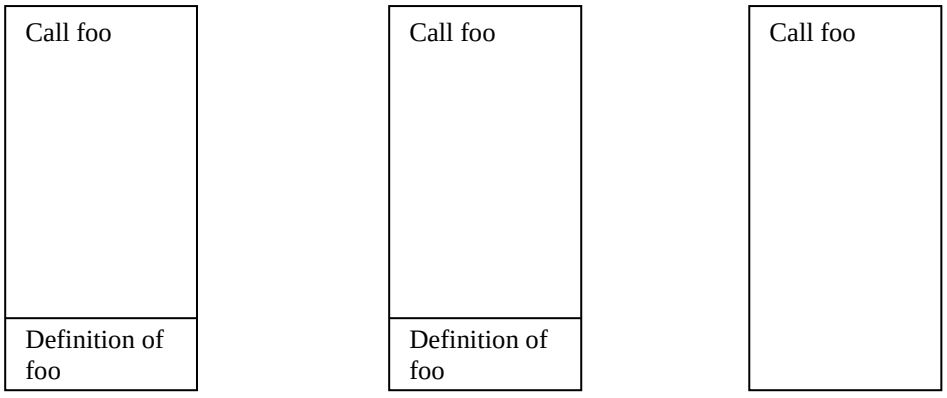
The definitions given above are, in general, inadequate to precisely determine the type of analysis which is required. This document uses examples to illustrate the type of defect which exhibits the unwanted behaviour. It then attempts to generalise in order to identify all the possible defects of that type.

Control Coupling

In many ways the issues are easiest to see by way of examples:

Example CC1. Control Coupling.

Consider the following model with three files and in each there is at least one call to a function foo. However two of the files contain a definition of a function foo. The two definitions may be identical or possibly similar (same interface).



The linker may choose to resolve all the calls to the function definition in file A (case 1) or it may resolve the calls in file A to the definition in file A, the call in file B to the definition in file B and the calls in file C may be resolved to either (case 2). This is an example of a Control Coupling defect because the user may be unaware of the ambiguity.

The design of the system should be able to resolve which of these two linkage cases is required.

Example CC2

This second example is based on parametric Control Coupling. Consider a function foo which has a procedural parameter, i.e. it is possible to pass the name of another procedure through the parameter list of foo.

```
int var;
void foo ( void (*pfunc)(void) ){
     if( var == 1 ) {
      *pfunc();   /* case 1 */
  } else {
      *pfunc();   /* case 2 */
  }

}
int main ( void ){
     loop:
       get ( var ); get ( glob );
       if( glob == 1 ){
          foo( &func1 );
       }else{
          foo( &func2 );
       }
     goto loop;
}
```

In this example the two data sets

var	glob
1	0
0	1

will ensure that every statement is executed, and additionally every branch (or control flow decision). However, potentially there are two functions which could have been called at the points labelled case 1 and case 2 whereas with these two data sets above only one function is called at each point, func1 at point case 1 and func2 at point case 2.

Control Coupling requires that all potential calls be executed at each point. To ensure that all the control flow calls are executed the requirements based test data will therefore need to have the additional two data sets.

Clearly this concept can be applied to all cases of the use of pointers to functions. Where a function is called by pointer dereference all the potential functions which could be called must be executed.

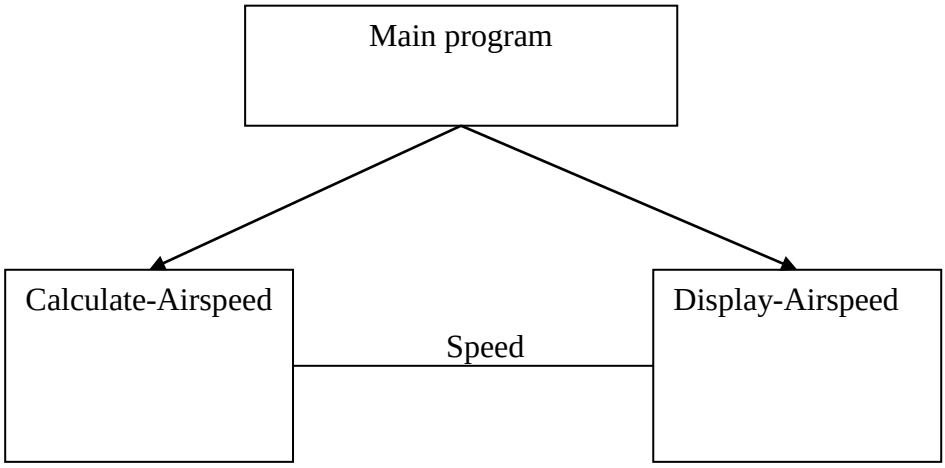
Similarly in C++ all virtual functions which can be called at a particular point must be executed by the requirements based test data.

To establish Control Coupling it is firstly essential that the set of possible functions to be called is known and secondly it must be known which members of this set are actually called. This information is normally proved by static analysis of the code under test.

Data Coupling

Data Coupling is totally dependant on the structure of the control flow graph of either the complete system or alternatively of sub-systems. Again consider some examples.

Example DC1. Data Coupling.



For this demonstration of a Data Coupling defect consider two functions, Calculate-Airspeed and Display-Airspeed both of which are called from the same main program (diagram above) and which share a global variable 'speed'. Calculate-Airspeed computes a value for the variable 'speed', i.e. a set operation, whilst the function Display-Airspeed outputs the value of 'speed' to a display device, i.e. a use operation. The main program may be of the form illustrated below.

```
int main(){
   int speed, order;
 loop:
   get(order);
   switch( order ){
     case 1: calculate_airspeed(speed);
     case 2: display_airspeed(speed);
   }
   goto loop;
}
```

It may be possible to construct one test case which executes Display-Airspeed and then another test case which executes Calculate-Airspeed. In this case the Control Coupling is

tested as required (i.e. every statement executed and every branch/decision executed, with no MC/DC requirements in this case) but the ordering of the calls is defective because Display-Airspeed has no valid speed to display. Similarly if a test case calls Calculate-Airspeed and there is no subsequent call to Display-Airspeed it is highly likely that there is another defect.

This example shows that there is a need to demonstrate that all the inputs to a given procedure have valid inputs at the point of a call and that all values assigned to global variables are used. In this context a global variable is a variable declared and set externally to a procedure in which it is used.

Example DC2.

Consider the following code: In the following function foo there are three uses of the parametric variable x.

```
 9Fint
10T  foo (
11F  int x )
12F  {
13T    if
14T      (
15T      x == 1
16T      )        // a use of x
17T      {
18T        return
19T        x ;     // a use of x
20T      }
21T    else
22T      {
23T        return
24T        x ;     // a use of x
25T      }
26T  }
```

Now consider the impact of calling this function from the following main program.

```
28Fint
29T  main()
30F  {
31F    int
32F      i ,
33F      y ,
34F      z ;
35T    for
36T      (
37T      i = 1
38T      ;
39T      i <= 2
40T      ;
41T      i ++
42T      )
43T      {
44T        y = i ;
45F             // y is set
46T
47T        foo (
48T        y ) ;  // y is aliased to x
49T      }
50T    scanf ( "%d" , & z ) ;
51F                    // z is set
52T
53T    foo (
54T    z ) ;  // z is aliased to x
55T  }
```

The first call to foo with argument y will cause every statement in foo to be executed and additionally every branch/decision because the loop generates both possibilities for y. It will also cause the execution of all three uses of y; The second call to foo with argument z does not cover any statements or branches/decisions in foo which are not already covered but it does introduce another three uses (of z) of which one is not executed. Data Coupling based testing will require two test cases, one with z having a unit value and a second case with a different value so that all three uses of z in foo (via the alias x) are covered.

From these discussions the DO178B/C requirement can be formalised as follows;

> Data Coupling involves all the global variables and also those local variables which are passed down through parameter lists to lower level components. These latter are local-global variables, but subsequently in this document both these two global variable types will be referred to simply as the set of global variables. Any of these global variables will be given a value by some operation such as an assignment or receive a value in an input operation. These are referred to as the *Set* operations. Then there are the points where these values contribute to the operations (calculations, outputs, etc.) of the program, these are referred to as the *Use* operations.
>
> There are two cases to consider; those variables which are declared globally and are used in many components but never appear as actual parameters in procedure calls, and those which do appear as actual parameters in procedure calls.
>
> For the first case, two lists are produced for each such global variable, the list of all Set operations, the *Set* list, and the list of all the use operations, the *Use* list. The impact of these lists on the measurement of Data Coupling will be discussed below.
>
> For the second case consider a specific global variable which occurs as an actual parameter in a specific procedure call. Prior to the call there will be one or more points in the control-flow graph where that variable was *Set* and from which the call is reachable. In general there will be a set of control-flow paths from each of these *Set* points to that procedure call. The list of *Set* points for that variable and procedure call combination will be called the *Set-Call* list. In the body of the procedure this variable will map to one of the formal parameters. In turn this formal parameter will have one or more *Use* points in the procedure body which here is termed the *Call-Use* list.
>
> The *Set-Call* and *Call-Use* lists will together be referred to as the variable Fan-In/Fan-Out at that procedure call. Therefore for every procedure call there is a super set variable Fan-In /Fan-Out consisting of all the separate *Set-Call* lists and *Call-Use* lists for all the input variables in the procedure call.
>
> If every statement in the program is executed every *Set* operation and every *Use* operation will have been executed but that can be achieved by executing just a subset of the *Set* operations for each procedure call (see previous example CC2). There are two schemes for improving the measurement of Data Coupling Coverage which can be derived from these lists. The first is to execute every Set operation in the fan-In lists and every Use operation in the Fan-Out lists. This will be referred to as Variable Fan-In/Fan-Out Coverage. The second scheme is to ensure that the all the combinations of the *Set-Use* pairs in these lists are executed, i.e. *Set-Use* Coverage. However, this second schema suffers from the possibility of a combinatoric explosion.

The DO178B/C standards committee have already ruled against a similar combinatoric explosion in formulating MC/DC coverage rather than requiring the full Branch Condition Combination Coverage (i.e. testing all the true/false combinations of the Boolean subconditions). Therefore the *Set-Use* coverage is not suitable for practical consideration for the same reasons of practicality. The consequence of this is that Data Coupling Coverage can only be realised practically via variable Fan-In/Fan-Out Coverage. In the same way that MC/DC Coverage realises much of the benefits of condition coverage so too does variable Fan-In/Fan-Out Coverage realise much of the benefits of global variable *Set-Use* Coverage.

Returning to the case of global variables which never appear in parameter lists it can be seen that statement coverage already ensures that all entries in their *Set* and *Use* lists have been executed. However it does not cover all the possible *Set-Use* combinations. Therefore the realisation of the measurement of Data Coupling as a non combinatoric technique means that only those global variables which are passed as parameters need to be considered.

The realisation and measurement of this metric is straightforward. Consider a trace of the execution of the software during a test. For a given variable a specific *Set* operation will be encountered, then, subsequently a use of this variable will be encountered. This means that both entries in the variable Fan-In/Fan-Out lists can be marked as covered. Later the same *Set* operation may be encountered and subsequently another *Use* point may be executed and hence the new *Use* can be marked as covered.

Note that this is reported for every procedure call and indeed each procedure call may appear more than once. This can happen if the procedure call is itself in the body of another procedure and a formal parameter of this enclosing procedure appears in the actual parameter list of the function call under consideration. Then this enclosed procedure must be evaluated for all the appropriate actual parameters of the outer call.

The extra costs incurred by the measurement of Data Coupling are difficult to estimate due to the impact of branch/decision coverage, procedure call coverage and MC/DC coverage which inevitably will execute more entries in the variable Fan-In/Fan-Out lists than simple statement coverage. In practice it might be a sensible strategy to only investigate the Data Coupling after these coverage metrics have been maximised to the required levels.

It is an objective of DO178C that all the Data Coupling shall be satisfied by the requirements based test data and that there shall be no *Uses* which are not preceded by a *Set*.

LDRA tool suite Features

In this section the features of the LDRA tool suite which contribute to Control and Data Coupling are described. There are two issues which need to be considered. Firstly, how can the need for special test cases be identified and secondly how can specific test cases be shown to provide the required coverage? The LDRA tool suite solves the first problem through the

use of Static Analysis and for the second case combines the knowledge produce by this Static Analysis with detailed Dynamic Analysis of a control flow trace.

Control Coupling

A key element of the Structural Coverage Analysis facilities provided by the LDRA tool suite is the control flow model which is automatically determined through an extensive Static Analysis of the source code under test. In reporting Structural Coverage Analysis the tool suite then compares the actual flow of control with that predicted through Static Analysis. This enables the tool to identify any errors in linkage or other departures from the required control flow behaviour. Linkage errors can arise, for instance, if the same procedure or function name occurs in several source code files. It is dependant on the Linker which procedures or functions get linked to which calls. The tool suite provides Static Analysis based warnings of the possible ambiguity and then produces Dynamic Analysis generated messages based on the requirements-led testing if the actual linkage is different from the predicted linkage. The user then explores the graphical representations to resolve what the design or architectural representations were predicting.

In support of this primary aspect of Control Coupling reporting the LDRA tool suite also reports Procedure Function Call Coverage and provides a graphical indication of Control Coupling via the Callgraph Display. This latter facility provides a visual representation of the dependence of a given software component on those components that call it. From the Callgraph Display users may then target specific instances of Control Coupling by selecting an individual software component (procedural node) and selecting the 'Nuclear Spider' option from the right-mouse button menu. This display provides a graphical representation of the immediate Control Coupling and the extended or hierarchical Control Coupling may then be shown with the 'Extended Spider' display that is selected from the same menu. Alternatively the calling components, together with their calling frequency may be listed textually via the 'Interface Information' option that is again available from the same menu.

This information may also be mapped back directly to the source code by 'drilling-down' to the specific predicates within the source code which must be satisfied in order to effect the call. This can be achieved either directly from the Callgraph Display or via the intermediate step of selecting the Procedural Flowgraph Display and then clicking on the appropriate node of the graph.

It should also be noted that Exact Semantic Analysis can be used to provide additional monitoring of global variables should this be considered desirable. For example annotations can be added which will check the actual function pointer values.

For the examples given previously:

Example CC1.

The LDRA tool suite reports the ambiguity caused by the two definitions and graphically shows the linkage which it assumes is correct. The user is then able to either confirm or refute this linkage model and remove the ambiguity. Alternatively, the Dynamic Analysis will show the actual linkage in terms of coverage. For case 1 the procedure definition in file B will get zero statement and branch coverage and all the coverage will appear in the definition in file A. For case 2 both definitions will

show coverage. Which resolution is used by the calls in file C will be shown in the procedure call table coverage. The defect can therefore be detected.

Example CC2.

In this example the LDRA tool suite statically scans all the calls to the procedure foo and obtains all the functions which are passed as actual parameters. The subsequent data and control flow analysis will utilise the list of functions which are called. This analysis will trace the called functions over the whole of the graph, through other procedure calls if necessary.

In conclusion, the Control Coupling verification is greatly simplified by the extensive Static Analysis checks which then increase the confidence of the Control Coupling coverage obtained from the Dynamic Analysis of the coverage of the requirements based test data.

Data Coupling

The LDRA tool suite addresses the Data Coupling defects in two ways, with Static Analysis and then with Dynamic Analysis. It is the power of these two techniques combined which yields the appropriate coverage metrics.

Firstly the tool suite applies system-wide Static Data Flow Analysis to the full graphical representation of the system-under-test and identifies not only the *Set* and *Use* points referred to previously but also the *Dec-Use* pairs (declaration to use paths), the *Set-Set* paths and the *Set-Unuse* paths (assignment to variable-out-of-scope path). Note *Unuse* points are sometimes referred to as variable kill points or variable *Undefine* points. Of these pairs the *Dec-Use* and *Set-Unuse* pairs are clear indicators of defects. The *Set-Set* pairs are common but whilst they can sometimes indicate a defect, in general they are too expensive to remove because they are subject to combinatoric possibilities. Note that in the notation of traditional Data Flow Analysis the *Dec-Use* pairs map to *UR* anomalies, the *Set-Set* pairs to *DD* anomalies and the *Set-Unuse* pairs to *DU* anomalies.

Since the setting of variables is often intimately related to I/O operations it is imperative to have no defects in the I/O operations. Therefore the tool suite also searches the system-wide control flow graph to show that all I/O operations are appropriate, coupling this with the use patterns of the input variables. The file operation checks ensure that on every path all reads and writes are preceded by an open, a currently open file is not reopened and all opened files are closed at program termination. Writing to a previously closed file or closing a previously closed file and other subtle I/O defects are also diagnosed.

It is possible for the analysis to produce unnecessary pairs (of all the types discussed) due to the presence of infeasible paths (termed unreachable paths in DO178B/C). The tool suite reports many causes of infeasible paths in the Static Analysis. This enables the software developers to simplify their code which in turn helps to reduce the cost of constructing requirements based tests. It is very frustrating to spend time attempting to execute code components in order to obtain coverage of a particular *Set-Use* pair only to discover that the paths which connect them are infeasible.

The Dynamic Data Flow Coverage Analysis then identifies the actual (as observed from the control flow trace) *Dec-Use* anomalies and the actual *Set-Unuse* anomalies because these

may indicate a possible incorrect ordering of the components. A *Dec-Use* defect (i.e. uninitialised variable) is always a fault whereas a *Set-Unuse* defect can arise from a clumsy programming style. This dynamic information is obtained from a trace of the execution path as the program executes the requirements based test data and hence provides an independent confirmation of results reported in the Static Data Flow Analysis. Note that it is usually more cost effective to remove these defects after the Static Analysis has been performed and before requirements based testing commences. The *Set-Use* and *Use-Call* lists and their coverage are reported as shown in Diagram 1.

Diagram 1

VARIABLE NAME	DPTH	PARAM ALIAS	FILE	PROCEDURE NAME	TYPE CODE	ATTRIBUTE CODE	USED ON LINES..	
i			ex_dc2.c	**main**	L	E	32	
					L	R	39	41
							44	
					L	D	37	41
x			ex_dc2.c	**foo**	P	E	11	
					P	R	15	19
							24	
y			ex_dc2.c	**main**	L	E	33	
					L	R	48	
					L	D	44	
	1	x	ex_dc2.c	**foo**	P	E	11	
					P	R	15	19
							24	
z			ex_dc2.c	**main**	L	E	34	
					L	I	50	
					L	R	54	
					L	D	50	
	1	x	ex_dc2.c	**foo**	P	E	11	
					P	R	15	
							19	*****
							24	

These results are derived from analysing a dynamic trace of the actual execution path taken by the program on its last run. Line numbers with ***** after have not been executed on this test run.

A key to terms used in Diagram 1 can be seen in Diagram 2.

Diagram 2.

```
                                  ******************
                                  *                *
                                  *  Key to Terms  *
                                  *                *
                                  ******************

        Type and Attribute codes are shown for each variable
        ----------------------------------------------------

        Variable Type Codes
        -------------------
             ======================
             TYPE           MEANING
             ======================
             C              Constant
             L              Local
             G              Global
             P              Parameter
             LG             Local-Global

        Variable Attribute Codes
```

```
------------------------
     ======================
     ATTRIBUTE      MEANING
     ======================
     E              Declaration
     D              Definition
     R              Reference
     I              Input
     O              Output
     N              Indirect Usage
     U              Unused parameter


Other Terms in Dynamic Data Flow Table
--------------------------------------
=======================
ITEM        DESCRIPTION
=======================
Variable    Main variable name matching user selection
Alias       Call depth + aliased name when passed as procedure parameter
n           Variable used on line n
n *****     Use of variable on line n not hit in any data set
            Use of aliased parameter on line n not hit for function call
(n)         Use of variable on line n hit in last data set
n -         Variable used on non-executable line n
```

The LDRA tool suite provides comprehensive Data Coupling information in both the Static and Dynamic domains as detailed below in order to assist developers select requirements based test data to execute any uncovered items in the variable Fan-In/Fan-Out lists.

Certain aspects of Data Coupling may again be accessed via the LDRA Testbed Callgraph and Flowgraph Displays where users may 'drill-down' to the source code containing the specific instances of predicates within the source which influence the behaviour of the dependent software component.

The Cross Reference Analysis and Data Object Analysis may be utilised to show ALL instances of the data items which are accessed by a particular software component. This includes local variables declared within the scope of the component and global variables accessed by the component, but declared elsewhere. Significantly, through the sophisticated Data Object Analysis module, the LDRA tool suite is able to provide a 'filtered' and hence extremely focussed analysis which will track and report user-defined data items across file and procedure boundaries even in cases where they are aliased as parameters to procedure calls or accessed via a call-chain. With the addition of Information Flow Analysis output, this reporting mechanism is further enhanced to include both direct and conditional data object dependence and inverse dependence tables.

In addition to contributing to aspects of the Data Object Analysis reporting, the Information Flow Analysis will identify and report data dependencies within the code under analysis and will also automatically detect and report inappropriate (unexpected) data dependencies. The ability to determine and verify the data interfaces between modules/components is then provided by the Static Data Flow Analysis module which reports the interface of each procedure in the source code in terms of its parameters, global variables and any returned results. Each of these items is listed together with its determined usage, i.e. whether the item is referenced only, changed within the procedure or unused.

In the Dynamic domain the Dynamic Data Flow Coverage facility provided by the tool suite indicates which data components have been accessed at run-time by the requirements based test data. In so doing, it utilises the execution trace associated with each specific test data set and thereby provides the Data Coupling for that particular test case as well as accumulated results for all test runs.

Data Coupling Analysis is a software integration technique because it looks at the way data is transmitted through the system. There are three ways in which integration can proceed:

1. Integrate the whole system in one operation,
2. Integrate bottom up, testing each level with the use of drivers,
3. Integrate top down, testing each level with the use of stubs.

The LDRA tool suite can accomplish the Data Coupling Analysis in any of these three alternatives. It can automatically generate both drivers and stubs with the minimum of user effort, substantially reducing costs.

The Dynamic Data Flow Coverage can also list each instance of an initialisation or *Set* operation of a global variable and all the subsequent uses of that variable. These are the variable Fan-In/Fan-Out lists. This includes all aliasing of a global variable as it is subsequently passed through procedure interfaces via parameter lists. The total set variable Fan-In/Fan-Out lists for each global variable and local variables passed through parameter lists is generated for any of the three integration options above. Then from the Dynamic Coverage Analysis which is generated using the requirements based test data the tool suite reports which of the *Set* or *Use* instances has been covered by the tests and critically those which have not been covered.

From the resultant coverage it is possible to detect when all the Variable Fan-In/Fan-Out instances have been executed and the DO178C objective has been met.

In conclusion the tool suite provides exceptionally detailed Static Analysis based checks to reduce the possibility of defects which make the task of assessing Data Coupling more difficult. The Dynamic Analysis produces explicit coverage of the coupling coverage of the requirements based test data.

Conclusion

This document describes the extensive features of the LDRA tool suite which can document conformance to the specification of requirements based test data Control and Data Coupling.

It shows that the LDRA tool suite can provide the information and analysis to achieve the DO178C objectives for Control and Data Coupling.

Appendix

For illustrative guidance on the detailed checks necessary the following is the appropriate table from ‘Developing Safety-critical Software’ by Leanna Rierson, CRC Press, 2013, ISBN 978-1-4398-1368-3. In the appropriate sections below the contribution of LDRA tool suite to each bulleted item will be shown.

TABLE 9.2

Example Data and Control Coupling Items to Consider During Design and Code Review/Analysis

Example Data Coupling Items to Consider

Example Data Coupling Items to Consider	**Example Control Coupling Items to Consider**
1. All external inputs and outputs are defined and are correct 2. All internal inputs and outputs are defined and are correct 3. Data is typed correctly/consistently 4. Units are consistent and agree with data dictionary 5. Data dictionary and code agree and are both complete 6. Data is sent and received in the right order 7. Data is used consistently 8. Data corruption is prevented or detected 9. Data is initialized or read-in before being used 10. Stale or invalid data is prevented or detected 11. Data miscompares or data dropouts are prevented or detected 12. Unexpected floating point values are prevented or detected 13. Parameters are passed properly 14. Global data and data elements within global data constructs are correct 15. I/O is properly accessed from external sources 16. All variables are set (or initialized) before being used 17. All variables are used 18. Overflow and underflow is identified and correct 19. Local and global data are used 20. Arrays are properly indexed 21. Code is consistent with the design	1. Order of execution is identified and correct 2. Rate of execution is identified and correct 3. Conditional execution is identified and correct 4. Execution dependencies are identified and correct 5. Execution sequence, rate, and conditions satisfy the requirements 6. Interrupts are identified and correct 7. Exceptions are identified and correct 8. Resets are identified and correct 9. Responses to power interrupts are identified and correct 10. Foreground schedulers execute in proper order and at the right rate 11. Background schedulers are executed and not stuck in infinite loop 12. Code is consistent with the design

The items in table 9 above can be addressed as follows:

1. All external inputs and outputs are defined and are correct. The procedure interface analysis shows details of each component's parameters, return values, I/O and global variables. This information is extremely helpful for Design Analysis and Review.
2. All internal inputs and outputs are defined and are correct. The procedure interface analysis shows details of all variables mapped to hardware and variables which appear in parameter lists to sub-components or output statements
3. Data is typed correctly/consistently. Strong type checking is a powerful feature of the tools invoking up to 340 checks on declarations and 166 checks on expressions..
4. Units are consistent and agree with data dictionary. The toolsuite checks that the interfaces between units satisfies some 90 checks on number, type and attributes of the objects in the interfaces.

5. Data dictionary and code agree and are both complete. The tool ensures that all requirements map to code and all code maps back to requirements which is a form of completeness.
6. Data is sent and received in the right order. There are checks that file reads, writes and closes are always in the right order. Other checks ensure that related components are executed on all paths in a predefined order. There are numerous checks to identify such ordering faults as variables being reassigned values with no intervening uses.
7. Data is used consistently. There are 166 checks on the structure of expressions.
8. Data corruption is prevented and detected. There are checks on array bound overflow and the use of overlapping declarations.
9. Data is initialized or read-in before use. The tool suite system wide data flow analysis provides a formal proof that this constraint is satisfied.
10. Stale or invalid data is prevented or detected. The tool suite checks that all input data is sanitized before use (ie subjected to constraints).
11. Data miscompares or data dropouts are prevented or detected.
12. Unexpected floating point values are prevented or detected.
13. Parameters are passed properly. There are some forty six checks performed on parameters.
14. Global data and data elements within global data constructs are correct. The strong typing is also performed on the fields of data structures ensuring that they are correctly initialised and used.
15. I/O is properly accessed from external sources. Checks are performed on the ordering of file opens, reads, writes and closes. The input values from reads are checked to ensure that their values are sanitized on all paths before use.
16. All variables are set (or initialized) before being used. The tool suite system wide data flow analysis provides a formal proof that this constraint is satisfied.
17. All variables are used. Checks ensure that all variables given a value are used before going out of scope.
18. Overflow or underflow is identified and correct. The tool suite reports some cases of potential integer overflow.
19. Local and global data are used. Checks ensure that all named locations holding data are used.
20. Arrays are properly indexed. Comprehensive array bound checks are performed statically. There is also a dynamic capability.
21. Code is consistent with design. The graphical facilities help to match the code to design representation.

In addition to the Static Analysis facilities described above, in the dynamic domain the Dynamic Callgraph Display incorporates structural coverage analysis information to demonstrate the degree to which the identified control coupling has been exercised at run-time.

The information sources described above may all be utilised to identify control dependencies. Additional, non-graphical, data such as that generated by the LDRA tool suite Information Flow Analysis and Data Object Analysis modules, further enhance the available reporting facilities by identifying module/component data object dependencies.

1. Order of execution is identified and correct. 85 checks
2. Rate of execution is identified and correct.
3. Conditional execution is identified and correct. 27 checks
4. Execution dependencies are identified and correct.

5. Execution sequence, rate and conditions satisfy the requirements.
6. Interrupts are identified and correct. 17 checks.
7. Executions are identified and correct.
8. Resets are identified and correct.
9. Responses to power interrupts are identified and correct.
10. Foreground schedulers execute in proper order and at the right rate.
11. Background schedulers are executed and not stuck in infinite loops. Infinite loops identified 26 checks
12. Code is consistent with the design.