Zero Defect Software Development (ZDSD)

Bill StClair,
Technical Evangelist

LDRA Ltd.,
Portside, Monks Ferry,
Wirral CH41 5LH,
U.K.

Tel: +44 (0)151 649 9300
Fax: +44 (0)151 649 9666
Email: info@ldra.com

The following white paper provides some of the key features of Zero Defect Software Development but is not an exhaustive list of all of the features.

ZDSD Manifesto

1. Introduction

Zero Defect Software Development (ZDSD) is a results-oriented process that emphasises the analysis, testing and reporting of the causality of defects. This process, which has evolved from LDRA's commitment to safety-critical software development and verification, supplants the traditional approach of reacting to undesirable effects and treating symptoms while attempting to manage verification process using "trend analyses" or simply tracking the occurrences of defects. Under ZDSD, the first and most feasible recourse is to map system specifications and requirements to design, implementation and verification artefacts and foster requirements-based development. This process encourages the prevention and early stage detection of defects by facilitating requirements validation and verification, design verification and source code standardisation. The ZDSD advantage is two fold: First, it results in higher quality software products; Second, when practiced using ZDSD-appropriate development and verification tools, the economic cost of quality is greatly reduced. This paper explores the challenges and support mechanisms for successfully utilising ZDSD.

A proposal for ZDSD, or any overarching process of its breadth, must suit current and future contexts. The world of embedded systems and applications development is now compelling test tool suppliers to focus on the critical domain of requirements definition, allocation and traceability and its linkage to the traditional aspects of implementation analysis and verification. Increasingly, organisational and project-specific requirements must be balanced with the more tactical, target-specific criteria of embedded systems. At the same time, the target systems architectures and their capabilities are becoming increasingly complex. Emerging from this new focus and its inherent challenge is a pronounced need for a verification solution that is:

1.1 Collaborative
1.2 Global
1.3 Scalable
1.4 Analytical
1.5 Applicable

1.1 Collaborative

Technology has evolved from the Information Age to a new age of collaboration. The Wikopedia model of collaboration has established at a minimum a new compendium of reference information. At the same time, it has also provided an expanding universe of verification. The resulting value added provides a knowledge base that is both immediately extensible and verifiable. Complementing this age of collaboration, is the open source development movement which has produced Linux and a myriad of user applications, mostly at lower cost and higher quality then those produced by closed processes and arbitrary verifications. The new collaborative verification model can encompass widely dispersed teams that examine and exercise the system under test and provides an indigenous defect tracking and correction mechanism.

1.2 Global

In order to facilitate the benefits of collaboration, the verification solution must provide access to all the players whenever and wherever they interact. Development and verification teams are increasingly dispersed and frequently beyond the boundaries of secure corporate networks and information "smokestacks". Notwithstanding the Virtual Private Networks (VPNs) and the like, the need for secure web-based access is essential. Verification results can be served from a repository that is continually updated by daily events.

1.3 Scalable

The verification solution must be adaptable to any point where a development activity occurs at any stage in the development process; moreover it must scale to the system under development, (be it a cell phone or long-range stealth bomber) the number of developers & testers or the number of requirements allocated. Moreover, all the verification results must be synchronised with a larger matrix of requirements and results.

1.4 Analytical

The verification process of any system encompasses the determination of what it does and a measurement of how well it does it. The assumption here is that the system's development is predicated on requirements; these requirements not only dictate what functions are performed but necessarily how these functions are implemented. The "what" of a system is defined by functional (or high level) requirements. A critical characteristic of these functional requirements as defined, is that they be testable. For example, a best practice in Extreme Programming dictates that a test case be defined for any given requirement before the source code is written.

Of equal importance to what a system does, is how well it performs, especially with respect to customer satisfaction. The determination of software system reliability and robustness is accomplished by structural testing. Structural testing necessitates the execution of feasible (or, real) paths through the code and therefore a synergy of source code analysis and test case automation. Functional and structural testing must be combined to ensure the "goodness" of the software system.

1.5 Applicable

The general applicability of a verification solution encompasses several factors including minimised dependence on tool chains, portability of verification routines and most importantly the repeatability of these routines. If a verification solution is dependent on the tool chain that produced the system under test, its ability to perform independent verification is compromised. As architectures (both hardware and software) become more complex or customised, the tool chain that produces the software that executes on these architectures becomes increasingly complex as well. Consequently, the coupling of the verification tools deployed on these tool chains must remain loose, while not precluding further integration at the user or project levels.

The portability of verification solution must occur across an increasingly distributed, variable and embedded systems space. This systems space necessitates that the verification solution should not carry "baggage"; instead it needs to make transparent usage of resources on the host or target systems.

Finally, the hallmark of applicability in a verification solution is repeatability. This requires that all elements of the verification routine be contained in a framework that is self referential and self contained.

2. The ZDSD Context Model

Depicted in Figure 1 is a context model for the verification solution being proposed. This model encompasses five essential levels in embedded systems development projects, though these levels may vary significantly with respect to process and project/corporate goals, artefacts generated and overall productivity.

Level 1

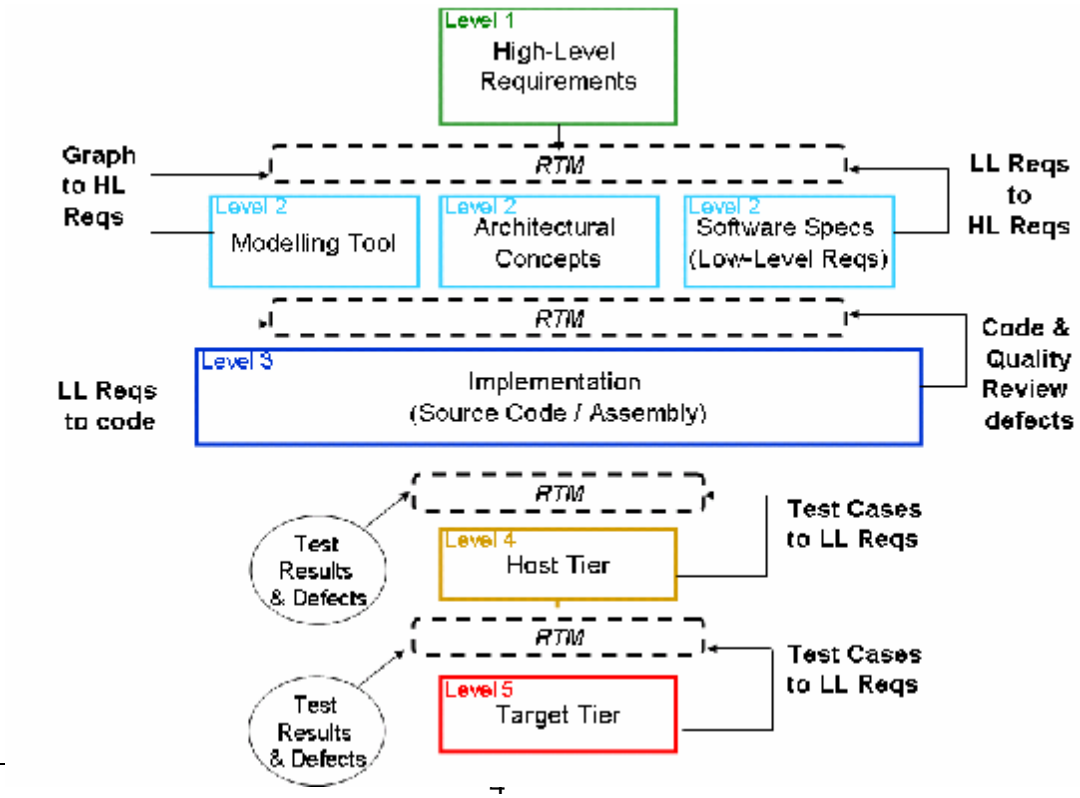
Level 1 is what should be a definitive statement of the system to be developed and the functional criteria it must meet, which is called High-Level Requirements. This level may or may not be elaborated, and its definition may be left to designers on Level 2 or even implementers of the system indicated on Level 3. Level 1, High Level Requirements may also be called System, Customer, Product requirements or something equivalent. These requirements may reside in a database such as Telelogic DOORS or exist as text in a format such as a Microsoft Word document.

Level 2

Level 2, Design, contains a representation of the design of the system characterised by Level 1. This level, in many contexts referred to as Low Level Requirements, must foremost establish links or traceability with Level 1. This linkage constitutes what is called a Requirements Traceability Matrix (RTM). The RTM is shared, either explicitly or implicitly, by all levels in the contextual model in Figure 1.

Level 2 typically is manifested by one of three design process categories. Specifically, Level 2 may consist of design model such as Unified Modelling Language (UML) or proprietary modelling systems such as Mathworks Simulink or National Instrument's Lab View. Alternatively, Level 2 may be realised by a design specification that delineates the physical characteristics of the software to be developed as well as the details of the interfaces that comprise the software system. Also, in more rapid prototyping or ad hoc contexts, the Architectural Concepts may be formed in meetings with key stakeholders in a project without the elaboration of more formal definitions.

Figure 1: ZDSD Context Model



Level 3

Level 3, Implementation, includes the production of source code or assembly code in accordance with Level 2 dictates. In the case of source code generated by a modelling tool, the linkage between Level 3 and Level 2 can be automated, assuring requirements traceability between the two levels. Frequently, however, embedded implementations are “hand coded” using the Integrated Development Environment (IDE) provided by an embedded operating system supplier, such as Green Hills or Wind River or LynxOS.

Level 3 is where verification activities typically begin. The most effective way to prevent run time errors is to preclude them. The first step in precluding runtime errors, as well as other implementation defects, is to systematically apply coding rules to the implementation. Coding rules such as MISRA or High Integrity C++ have been developed by industry leaders to analyse many aspects of source code implementation such as pointer usage, memory management and syntax. The results of these code reviews, as well as critical quality checks such as code complexity and maintainability, including overall data flow analyses, must be accomplished prior to proceeding to verification in Levels 4 and 5. Moreover, to effectively manage this first level of verification, non-conformances (as established by coding rules and project-specific quality models) must be recorded, tracked and resolved using a defect tracking system that is integral to the overall RTM.

A difficult challenge at Level 3 is the mapping of requirements, either high-level or low level, to the code. This linkage demands an understanding of the code, at least at the functional level, as well as documentation required to evidence this linkage. Moreover, in order to effectively perform the verification tasks in Levels 4 and 5, this requirement to link source code must be integrated into the RTM.

Level 4

Level 4 is the first level dedicated to verification. At this level embedded software can be tested as functional and structural entities. Test strategies are adopted, such as top-down, bottom up or some combination of these two strategies. This level may include simulators, software stimulation techniques as well as automated test harnesses and test case generators. Structural testing requires extended use of the analyses and predicates Formal Methods are performed at Level 3, if the testing of real paths is to be accomplished. Without analysis techniques the causality of defects can not be effectively exposed.

With respect to functional testing, a key success criterion for the verification tasks to be performed is that the functional testing performed at Level 4 be repeatable at Level 5, the embedded target. Without this host to target compatibility, verification traceability is more difficult to maintain.

At Level 4 the RTM gets fully extended to include verification artefacts such as test case identifiers, test specifications and test results. The test results must be augmented by defect reports if non-conformances are to be corrected. The emphasis in Level 4 is to correct defects at this level prior to target execution and system integration testing. Host-based testing is typically the least cost option, as it best accommodates the earliest testing and provides optimal access to most low-level functionality independent of the target platform.

Level 5

Verification for embedded software, especially safety critical software, typically is completed at Level 5. However, at this level, functionality frequently crosses hardware/software boundaries making defect resolution more challenging than at Level 4. Many of the host-based testing facilities such as run time libraries are impractical and become infeasible in multi-partioned operating systems. Another layer of traceability, going from source code to object code must also be covered.

Functional testing predominates at Level 5. In addition, structural coverage is used to measure the final traceability between functional requirements and the implementation. (i.e., is code left uncovered by functional test? If so, then why is the code there?). All of the test artefacts used at Level 4 are also relevant at Level 5.

3. The ZDSD Solution

The ZDSD Solution offered by LDRA fully meets the demands of the ZDSD contextual model described in the previous section. The ZDSD Solution is shown in Figure 2 in the context of a V model for software development. TBreq provides the RTM described above across all five levels of the ZDSD context model. TBreq supports the importation of requirements from any source including Telelogic DOORS or Microsoft Word and the mapping of requirements to the source code. Alternatively, TBreq facilitates the coupling requirements mapped into a design modelling tool such as Telelogic Rhapsody or Mathwork's Simulink. The TBreq integration with LDRA Testbed facilitates the Code Review and Quality Review described at Level 2 (Implementation), as well as externally performed system test code coverage analysis. Additionally the TBreq integration with LDRA TBrun (unit test tool) supports the software testing described at Levels 4 and 5 using the TBrun.

The critical requirements traceability and verification capabilities are performed in conjunction with TBmanager. TBmanager and TBreq operations are shown in Figure 3. As indicated by (1), TBreq you can capture requirements from requirements management tools such as Telelogic DOORS, IBM Rational RequisitePro or from a document or spreadsheet. TBreq acts as a gateway to these requirement sources for TBmanager. The requirements are made available to TBmanager (via a LDRA Testbed Project) for traceability and verification tasks. Indicated by (2), traceability and requirements mapping are performed directly in LDRA Testbed; information is captured from the Design Review analysis of source code files loaded into the LDRA Testbed project. This review includes the application of Formal Methods and a Real Path analysis of the source code, providing the basis for code and quality assessments and structural testing (eXtreme testing) in TBrun. Indicated by (3), verification results, including unit test case and code coverage results from TBrun, as well as analysis results, including Code Review, Quality Review and Dynamic Analysis from externally executed test cases, are returned to TBmanager. Also returned are defect reports from requirement non-conformances. Finally, as indicated by (4), all traceability and verification are put into the overall Requirements Traceability Matrix Verification results and traceability information can be uploaded into the requirement repositories.

Figure 2 – V Model for Software Development

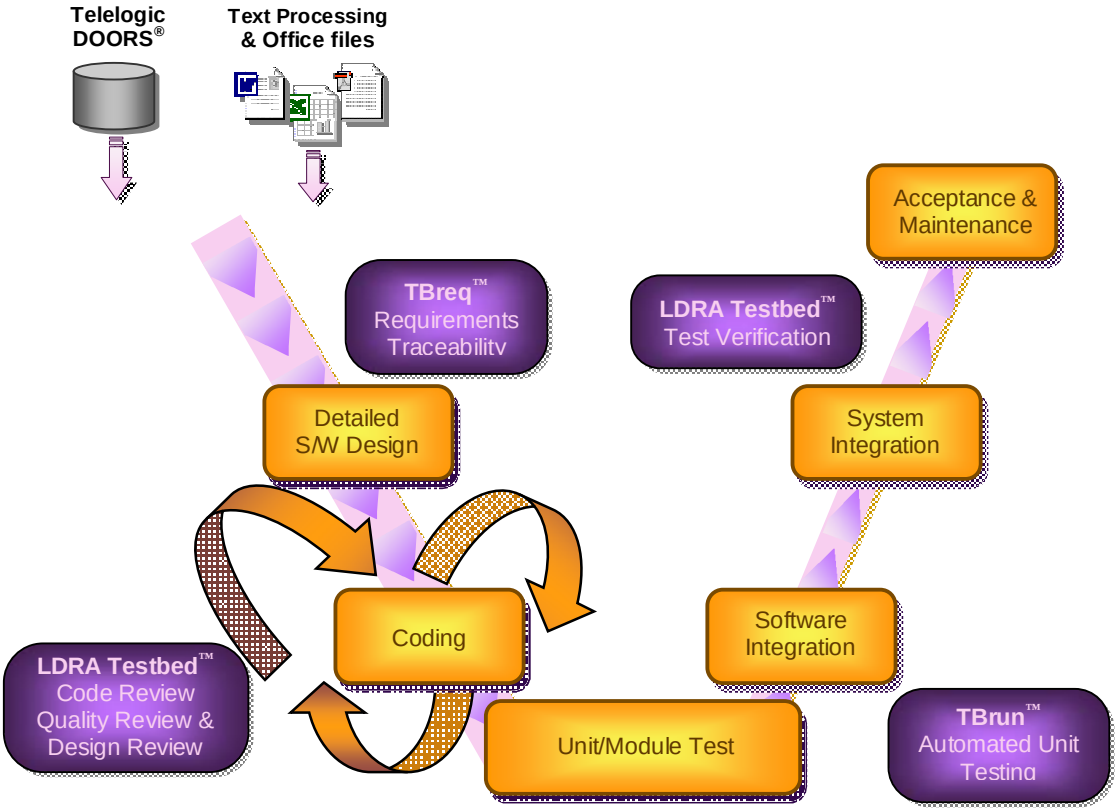
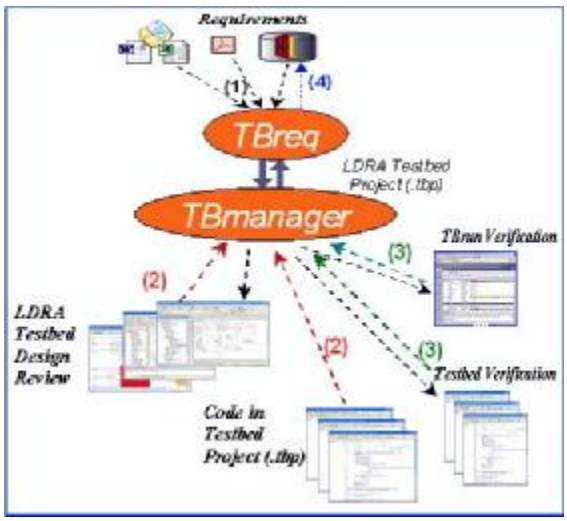


Figure 3 – TBreq and TBmanager Operations



The underlying analysis and test verification technology utilised in the ZDSD Solution is enabled by the industry unique application of Formal Methods as described in Section 4.

4. The Formal Methods Techniques

Traditionally, formal methods are regarded as the use of mathematically-based models in some specification or design notation, such as Z or finite state machines (FSMs). In recent times the use of various modelling techniques have also become associated with the term formal methods and it is this relationship which is relevant to ZDSD. In the current context, there are two underlying models which form the principal basis for most of the techniques to be described; these are the control flow model and the data flow model.

Control flow modelling. The tool constructs a graphical control flow model in which the nodes are basic blocks and the arcs are control flow jumps and branches. This model handles programs regardless of structure, including many types of interrupt and exception handling methods, recursion (self recursion and multi-procedural recursion) and multiple file representation. The model is produced system wide, i.e. for the entire program, and has an elaborate set of display, navigation and reproduction facilities. The model can be reduced to yield an annotated call graph or annotated flow graph. The representation is capable of handling procedural and label parameters, arrays of pointers-to-procedures and polymorphism.

Data flow modelling. The data flow model consists of an enhanced version of the control flow model annotated with operations on the program variables and constants. It performs the aliasing operations across procedure boundaries and other more specific aliasing operations (pointers and references, etc.). The model is again system wide. The model can be accessed to obtain the set of operations on specific variables, or collections of variables.

The basic models are, in essence, reverse engineered from the program code. They are then used to provide the basis for a number of specific algorithms which concentrate on discovering particular kinds of program defects and properties.

4.1 Data Flow Analysis

The data flow model, annotated with the variables and operations performed on them, is the basis for this technique [3]. Powerful graph theoretic algorithms are applied to the system-wide control flow model to yield a number of different types of analysis.

Defects detected include:

- References to uninitialised variables;
- Wasted computations on variables;
- Variables which do not contribute to outputs;
- Parameter mismatches of various types.

The technique is valid for all paths and handles recursion [4] and some types of interrupts and exceptions. The analysis is performed bottom up and proceeds across procedure boundaries with the corresponding multiple aliasing handled explicitly. Multiple file problems are handled by firstly predicting an appropriate compilation order and then performing the analysis of the procedures in this order. The precise interface, i.e. the one in use and not necessarily the one declared, is documented for user convenience and reference. The mismatch between the actual interface and declared interface is often the source of serious errors.

4.2 File Handling Analysis

Olender and Osterweil [5, 6] were among the first to realise that traditional data flow analysis of variables could be adapted to analyse the sequence of operations on files (i.e. open, read, write, close). When files are opened within a program and the program subsequently exits with some files not closed, there can be unfortunate side effects, for example, computer lock-up. By searching the system-wide control flow graph, annotated with file operations (over all files and all procedure boundaries), this algorithm checks that any file that is opened is subsequently closed on all exit paths. It also reports any possibility that a ‘write’ operation could be made to any unopened file. This can occur if there is any path from the start point of the program to the

specific ‘write’ statement. The technique is able to utilise some knowledge about infeasible paths [7] to reduce false positive messages.

4.3 Pointer Analysis

Pointer analysis is a much-studied problem [8, 9]. A pointer is a program variable in its own right and hence must obey the usual data flow rules as applied to ordinary variables. In addition, however, it can be dereferenced which means that operations are performed on the entity to which the pointer points.

In the LDRA Testbed, pointer assignments, dereferences and uses are superimposed on the data flow model of the entire system. This permits data flow analysis to take into account assignments and other uses made by dereferencing pointers. Every pointer dereference is both a use of the pointer itself and of the variable or location to which it points. In general, dereference is a dynamic issue because a given pointer may, at various times, point to many different locations. However, there are a number of static analysis representations which can detect many, but not all, possible defects. The algorithm implemented works with a ‘last assigned value’ approach except for procedural pointers in which it uses an ‘all possibilities’ approach. The latter is not used for pointers in general, due to the possible combinatorial explosion.

4.4 Null Pointer Checking

When assignments to pointer variables are made with function return values, other pointer values or explicit null values, it is important to check that these pointer variables have sensible values before they are used. This technique searches the data flow model annotated with the pointer operations and function calls, together with any conditional operations at splitter nodes, to ensure that every pointer which is assigned a value is checked (for null say) on all paths involving a use (i.e. a def-use path) of that pointer. This can detect most troublesome program pointer problems because it works over the complete control flow graph. It cannot detect cases when the wrong valid pointer value is used.

4.5 Divide-by-Zero Analysis

This analysis is performed by searching the data flow model annotated with values and operations, to check when variables, which might have a zero value, are used as a denominator. The algorithm does not attempt to compute the set of values achieved by the variables; rather, it examines the constructs to predict when a variable might have a zero value. In this way, a fast algorithm is obtained which produces a minimum of false positive results.

4.6 Array Bounds Checking

Array bounds checking using techniques based on data flow has been extensively studied [10]. However, in addition to statically checking array bounds by suitable scanning of the data flow model, the LDRA Testbed also permits dynamic checking by means of instrumentation. Both techniques work on a system-wide basis projecting the bounds down to lower levels where the language fails to provide these details.

4.7 Storage Analysis

One of the major sources of faults in the execution of software in some languages is the exhaustion of available storage. Often this is caused by the programmer allocating memory and then omitting to free it subsequently. The control flow model is searched for uses of those constructs which explicitly allocate and free memory to ensure that all allocated memory is correctly freed on all exit paths. The algorithm also checks for the potential release of unallocated memory.

4.8 Dead Code Analysis

The flow graph, annotated with variables and operations is scanned to detect the case when specific computations do not lead to any changes in any outputs. Such computations can be

safely removed from the code. Categories reported include unreachable, infeasible and ineffective code. The unreachable code can be identified by checking reachability from the program start point (or other points if required), and detecting infeasible branches of various types.

Additionally, variables declared and never used, and variables used only once are identified for removal. It is also possible to detect infeasible branches during the dynamic analysis phase. The main purpose in detecting these defects in the static analysis phase is that it is cheaper to remove these defects before commencing dynamic analysis.

4.9 Exact Semantic Analysis

The validation process can be substantially enhanced if the user can provide a tool with information which is either hard to obtain by analysis or is from the application domain. Traditionally, this is supplied in the form of comments (usually referred to as annotations) which can be transformed automatically into allegations or assertions [11].

In the tool, assertions in the form of annotations are compared with the actual computations in order to detect violations. These annotations can be pre- and post conditions, loop invariants, etc.

The tool uses annotations in two modes: static analysis mode, and dynamic analysis mode. With the former, the technique becomes approximate semantic analysis and with the latter, it becomes exact semantic analysis because the annotations are checked in the actual execution environment. In the static case, the engine which checks the annotations is software based and in the dynamic case, it is the run-time system. Clearly, checking semantic issues in the actual environment is more accurate than in a simulated environment.

4.10 Information Flow Analysis

Information flow analysis aims to discover the relationships between input variables and output variables [12]. This is performed in the LDRA Testbed by scanning the system-wide control and data flow graphs to discover such relationships. In practice, other dependencies such as those introduced by design artefacts are also discovered. The relationships are explored in detail to find possible sources of faults. In fact, when used in the basic mode of identifying the relationships, very few general classes of obvious fault can be identified.

However, the technique does become extremely powerful when combined with some knowledge of the application. The application knowledge can be encapsulated in the form of annotations describing the required relationships either system wide or for each procedure. The actual results can be compared automatically with those predicted from the requirements leading to a fast and powerful facility. These annotations are usually obtained by the application of formal methods to the requirements analysis and design which leads to accurate predictions of the required relationships.

4.11 Side Effect Analysis

The use of functions in complex expressions can be a source of error if the functions concerned have side effects. In particular, the result can be affected by the compiler's order of evaluation. Frequently, compilers utilise any freedom permitted in the language definition of the order of execution to perform optimisation.

The side effects which the tool identifies are classified as:

- Parameter side effects;
- Global variable side effects;
- I/O side effects, both file and volatile location based;
- Class member side effects.

Class member side effects are distinguished from global variable side effects purely because exponents of class-based languages need to feel that these languages are significantly different from others. As far as the tool is concerned, they are the same. All uses of such functions in

positions where there could be evaluation problems are reported so that the relevant code can be restructured.

4.12 Data Coupling Analysis

This technique investigates the way in which procedures interact with data items which are not local to that procedure. The two mechanisms by means of which procedures acquire external data items are parameters and global variables. The term 'global variable' in this context covers all items visible inside a procedure and declared externally, so that class members can fall into this category.

The task is to ensure that there are no possibilities of dangerous defects arising from the various aliasing mechanisms possible in many languages; for example, a global variable when passed as a parameter in a call then has two access mechanisms inside the procedure. The danger arises firstly from the programmer failing to appreciate this fact and thinking they are distinct and secondly from a compiler treating them as distinct when the programmer thinks they are the same. These problems can become quite subtle when a system has a complex hierarchy and the locations are treated in alternate ways as they filter down that hierarchy. The use of pointers in such a scenario adds further complexity, to the point where it is usually beyond humans to comprehend.

The tool has algorithms to detect problems of this type. They are again based on the data flow model, are system wide and handle the aliasing complications of cross procedure boundaries and the use of pointers.

4.13 Real Path (LCSAJ) Analysis

The set of 'linear code sequence and jump' (LCSAJ) subpaths forms a basis set for the generation of program paths [13, 14]. As such, LCSAJs are a powerful vehicle for analysing path structure and generating targeted test data. The tool produces a test case plan targeted to the achievement of testing all the LCSAJs.

The LCSAJ test case planning component is particularly important to users who wish to achieve a high level of test coverage at minimum cost. Essentially, the tool generates a spanning tree of the LCSAJs which can then be optimised in order to perform specific minimisations, such as construction of a set of paths (connected LCSAJs) to cover all the LCSAJs. The set of conditions which achieves this is then the input criterion for test case generation.

4.14 MC/DC Test Case Planning

Modified condition decision coverage (MC/DC) requires testing of decisions in a program such that changing the truth value of each individual condition within the decision forces a consequence on the overall decision's outcome [15]. This is another area where use of mathematics is required. The problem is that an expression containing N conditions combined with the logical **and** and **or** operators, leads to 2_N test combinations, of which only a set of $N + 1$ tests is actually needed to satisfy MC/DC.

In addition, there may be a number of different test data sets that satisfy MCDC and most programmers involved in this work cannot perform the necessary analysis manually. The tool therefore provides a test case planner which either guides testers through the process from start to finish or rescues them when they are part way through and have lost track of what to do next.

The need for such a tool has increased as the avionics industry has moved to ever more complex conditional expressions, many of which have interdependent sub conditions. It is not uncommon to have well in excess of 20 sub conditions.

4.15 eXtreme Testing

The automatic generation of test data for the purpose of unit testing is one of the most important features of the LDRA Testbed. The analysis involved is able to handle recursive procedures, groups of mutually recursive procedures, and code distributed across files. The technique which

is application independent is based around the relationships of literal values and associated constraints which are observed in association with the variables involved in the units under test. This has been implemented largely under pressure from experienced users who find that dark-corner testing is usually hard because the actual test data is not obvious but can be inferred from the presence of these program literals and their context. Surprisingly high coverage rates have been achieved for some classes of software. The user still has to provide the assurance that the actual outputs are correct.

4.16 Exception Handling Analysis

In languages like C++ it is possible for users to raise exceptions at any point in their programs. This leaves open the question as to whether there is an appropriate exception handler in scope. In Exception analysis the system wide control flow graph is annotated with the points where exceptions are explicitly thrown and where the handlers are declared. Cases where there are no handler are on any path to the raised exception are reported.

4.17 Timing Analysis

This is a dynamic analysis feature in which the time taken to execute specific procedures can be recorded and reported to the user.

Suitable hooks are provided at user request during the instrumentation phase. The options are:

- No performance timing
- Performance timing only (no coverage instrumentation)
- Performance timing plus coverage.

When performance timing is requested calls to two routines are planted in each function:

- At the start of the function
- At each return point or end of the function.

Both functions take a single parameter that provides the procedure number which identifies the procedure to LDRA Testbed.

5. Conclusion

ZDSD is a largely automated yet agile process that links all the stakeholders (customers, development and verification teams and business operations) in a cohesive entity focused on measurable results. ZDSD maintains that intentionality (requirements) must be thoroughly and consistently verified in order to minimise, if not eradicate defects while producing software products that add demonstrable value at the lowest possible cost over a product's lifecycle. The obverse of this conclusion is that software "accidents" rarely accrue to a customer's (or supplier's) benefit. In fact, the software accident has always been the nemesis of safety critical software and presents the greatest risk to all other software development disciplines.

6. References

1. Fergus, E., Hedley, D., Riddell, I.J., Hennell, M.A.: Software testing tools. In: Ince, D. (ed.) *Software Quality and Reliability: Tools and Methods*, Chapman and Hall, London, 56-70 (1991).

2. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. *ACM Computing Surveys*, **29**(4): 366-427 (1997).

3. Fosdick, L.D., Osterweil, L.J.: Data flow analysis in software reliability. *ACM Computing Surveys*, **8**(3): 305-330 (1976).

4. Fairfield, P., Hennell, M.A.: Data flow analysis of recursive procedures. *ACM SIGPLAN Notices*, **23**(1): 48-57 (1988).

5. Olender, K.M., Osterweil, L.J.: Specification and static evaluation of sequencing constraints in software. In *Proceedings of the Workshop on Software Testing*, Banff, Canada, IEEE Computer Society Press, Los Alamitos, CA, 14-22 (1986).

6. Olender, K.M., Osterweil, L.J.: Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, **1**(1): 21-52 (1992).

7. Hedley, D., Hennell, M.A.: The causes and effects of infeasible paths in computer programs. In *Proceedings of the 8th International Conference on Software Engineering*, London, U.K., IEEE Computer Society Press, Los Alamitos, CA, 259-266 (1985).

8. Hind, M., Pioli, A.: Which pointer analysis should I use? In: Harrold, M.J. (ed.): *Proceedings of the ACM SIGSOFT 2000 International Symposium on Software Testing and Analysis*, Portland, OR, *ACM Software Engineering Notes*, **25**(5): 113-123 (2000).

9. Hind, M., Pioli, A.: Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, **39**(1): 31-55 (2001).

10. Gupta, R.: A fresh look at optimizing array bound checking. In: *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, NY, ACM Press, New York, 272-282 (1990).

11. Rosenblum, D.S.: A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, **21**(1): 19-31 (1995).

12. Bergeretti, J.-F., Carré, B.: Information-flow and data-flow analysis of 'while' programs. *ACM Transactions on Programming Languages and Systems*, **7**(1): 37-61 (1985).

13. Hennell, M.A., Woodward, M.R., Hedley, D.: On program analysis. *Information Processing Letters*, **5**(5): 136-140 (1976).

14. Woodward, M.R., Hedley, D., Hennell, M.A.: Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, **6**(3): 278-286 (1980).

15. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, **9**(5): 193-200 (1994).

16. RTCA. Software considerations in airborne systems and equipment certification. Report DO-178B, Radio Technical Commission for Aeronautics (RTCA) Inc., Suite 1020, 1140 Connecticut Avenue NW, Washington DC 20036, U.S.A. (1992).