Formal Methods by Stealth: Formal Methods Implemented in the LDRA Tool Suite

M. A. Hennell[1], and M. R. Woodward[2]

[1] LDRA Ltd., Portside, Monks Ferry, Wirral CH41 5LH, U.K.
mike.hennell@ldra.com
http://www.ldra.com
[2] Department of Computer Science, University of Liverpool, Chadwick Building, Peach Street, Liverpool L69 7ZF, U.K.
m.r.woodward@csc.liv.ac.uk

Abstract. This paper outlines a number of static analysis techniques that may be regarded as formal methods in the sense of being mathematically based. The techniques form part of a well-known, commercially available tool suite, the LDRA tool suite. Many customers who use the tool suite, particularly those in the avionics software industry, regularly use the techniques to conform to the standards of appropriate certification bodies. Such practitioners would not normally regard themselves as employing formal methods. It is as if the formal methods are there 'by stealth', somewhat akin to a stealth plane that is invisible to radar.

Keywords: formal methods; test tools; static analysis; software modelling.

1 Introduction

Many software practitioners, if asked about the use of formal methods in their environment, would reply that they never use such methods. Whilst this statement Many might be true in most environments, it is factually untrue in others. The reason for this is that these practitioners are often unaware that they are actually using such methods because this fact is not visible to them.

In this paper, the term 'formal methods' will be used in the sense of mathematically-based analysis techniques which are applied to an appropriate model of the complete, or partially complete, software. The formal methods built into a well-known test tool over a period of some thirty years are described. The paper describes how it is substantially due to these methods that the tool has been successful in some of the most demanding application areas.

The LDRA tool suite [1] commenced life in the 1970s as a dynamic analysis tool obtaining test coverage metrics [2]. In the intervening years, apart from being made available in some fourteen different languages from ADA to assembly language, it has been substantially enhanced in the area of static analysis. The main thrust of this static analysis has been to focus on the issue of detecting violations of programming standards and the enforcement of constraints such as strong type checking and the application of powerful algorithms to detect defects of many types.

One factor which marks this tool suite as unique is the fact that the various forms of static analysis are applied to the complete set of languages where applicable and, in most cases, are dialect independent. Most of the sophistication in the tool lies in this dialect independence.

The paper is organized as follows. Section 2 gives brief descriptions of the various techniques and Section 3 attempts to give some insights into typical tool usage. Finally, Section 4 makes some concluding remarks.

2 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 this paper. 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 flowmodel and the dataflow 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.

Dataflow modelling. The dataflow 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.

2.1 Data Flow Analysis

The dataflow 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.

2.2 File Handling Analysis

Olender and Osterweil [5, 6] were among the first to realise that traditional dataflow 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 utilize some knowledge about infeasible paths [7] to reduce false positive messages.

2.3 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 dataflow 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 tool suite, pointer assignments, dereferences and uses are superimposed on the dataflow model of the entire system. This permits dataflow 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.

2.4 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 dataflow 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 defuse 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.

2.5 Divide-by-Zero Analysis

This analysis is performed by searching the dataflow 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.

2.6 Array Bounds Checking

Array bounds checking using techniques based on dataflow has been extensively studied [10]. However, in addition to statically checking array bounds by suitable scanning of the dataflow model, the LDRA tool suite 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.

2.7 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.

2.8 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.

2.9 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 LDRA tool suite, assertions in the form of annotations are compared with the actual computations in order to detect violations. These annotations can be pre- and postconditions, loop invariants, etc.

The LDRA tool suite 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.

2.10 Information Flow Analysis

Information flow analysis aims to discover the relationships between input variables and output variables [12]. This is performed in the LDRA tool suite by scanning the system-wide control and dataflow graphs to discover such relationships. In practice, other dependencies such as those introduced by design artifacts 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.

2.11 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.

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

2.13 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 targetted test data. The tool produces a test case plan targetted 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 optimized in order to perform specific minimizations, 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.

2.14 MCDC Test Case Planning

Modified condition decision coverage (MCDC) 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 MCDC.

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 subconditions. It is not uncommon to have well in excess of 20 subconditions.

2.15 Automated Test Case Generation

The automatic generation of test data for the purpose of unit testing is one of the most important features of the LDRA tool suite. 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 darkcorner 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.

3 Tool Usage

The environments in which the tools are used are extremely varied but the main field in which the formal methods are likely to be used is that of real-time embedded systems, mostly in the avionics, nuclear and telecommunications industries. In the avionics application area, the principal purpose is to show that the software conforms to the avionics DO-178B standard [16]. For safety-critical software needing certification at level *A* of this standard, the code must satisfy rigorous dynamic analysis coverage demands, including full MCDC (also demonstrated with the aid of the tool but outside the scope of this paper) and, in addition, a set of detailed static analysis requirements. It is these static analysis requirements which are satisfied primarily by the formal methods techniques featured in the tool.

The principal benefit accruing from the use of the tool is the huge saving in manpower costs which are otherwise consumed in the task of showing conformance to the standard. Moreover, the DO-178B standard suffers from significant levels of ambiguity and variance in interpretation by the many certification bodies, both around the world and also within the United States. The totality of the methods presented by the tool spans a group of techniques, which have, to date, satisfied all these certification bodies.

One of the major handicaps faced by analysis tools is the potentially huge number of false positive defects reported which are due to the presence of infeasible paths. The manual checking of these false messages is time consuming so considerable effort has been spent over a long time to detect these infeasible components [7] and remove them from the analysis.

The LDRA tool suite has been used by every major avionics software developer around the world and the number of systems successfully certificated is already into the thousands. The high reliability of in-service avionics software controlled systems is a notable achievement and the contribution of this tool is significant.

Also, it must be noted that the use of this tool does not in any way exclude the use of other formal methods. There are specific characteristics of some types of software which need to be demonstrated and for which the tool has no capability as yet, e.g. livelock or deadlock potential. Then again, the unit test capability of the dynamic analysis requires a detailed specification of each unit to be available and one convenient representation is that derived from one or more of the formal methods notations.

4 Conclusions

The contribution of the formal methods capabilities of this widely used software tool to the achievement of levels of software quality required by many international regulatory bodies is considerable. Nevertheless, there is much still to do. The current drive is to integrate the tool more closely into techniques capable of providing a sufficiently rigourous description of test data which can be used in the unit testing and dynamic analysis components of the tool. The users require a seamless (and simple) way to combine the benefits of this and many other tools.

One of the major problems in the application of any technique that has been demonstrated experimentally to find defects is the type of software which users invariably generate. This means that the technique must be capable of handling huge systems, spread across many files with hugely different programming styles. Very often, the code producers are inexperienced in the area of careful code construction and hence the technique must be sensitive to hurt pride as well as responsive to the requirements of certification standards.

The future appears to be that more and more organizations are producing critical code and they are doing so with relatively unskilled labour. The commercial imperative seems to be such that only the use of automated tools to guide and encourage these producers to the required levels (of the certification bodies) will prevent a move to downgrade the current high standards.

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 th e 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).