Industrial perspective on static analysis

by B.A. Wichmann, A.A. Canning, D.L. Clutterbuck, L.A. Winsborrow, N.J. Ward and D.W.R. Marsh

Static analysis within industrial applications provides a means of gaining higher assurance for critical software. This survey notes several problems, such as the lack of adequate standards, difficulty in assessing benefits, validation of the model used and acceptance by regulatory bodies. It concludes by outlining potential solutions and future directions.

1 Introduction

Static analysis tools are gaining ground as a complementary technique to conventional dynamic testing in order to obtain additional assurance on critical items of software. Unfortunately, the exact benefits of static analysis are hard to quantify. In addition, the absence of any effective standards in the area implies that there is no yardstick against which to measure benefits.

Two major safety-critical software standards give support for static analysis: the UK Interim Defence Standard 00-55 [1], in which the support is detailed and specific, and the international civil avionics standard in which design analysis forms a key role [2]. In the UK, 00-55 has had a significant impact, not only on the defence area, but in other related critical software, such as in the nuclear sector.

For a UK Nuclear Power Plant [3], the responsibility for demonstrating safety rests with the operator. Four main elements are being used to justify new software-based safety systems:

- the standard and quality of the design and production processes.
- independent assessments, including extensive static analysis of the code.
- a challenging dynamic test to give confidence on reliability.
- a one year trial before operation.

The analysis performed by Nuclear Electric for the Dungeness 'B' Single Channel Trip system and the Sizewell B Primary Protection System provides a good illustration of the current industrial application of static analysis techniques (see Sections 2.2 and 2.4).

In other application areas, the examples given here demonstrate that industry is increasingly using static analysis for quality assurance purposes. However, the encouragement of further adoption requires that the basic issues and the benefits clearly justify the costs. In this paper, we highlight the issues and indicate likely developments.

2 Current experience

Static analysis has two essential characteristics which must be borne in mind when studying any specific application. These two characteristic are

- ☐ *nature*; this is the broad objective of the analysis which could be portability against a language standard or correctness of some aspect against a program specification.
- ☐ *depth*; this indicates the semantic depth of the analysis. For example, analysis of program layout is very shallow (and hardly worth even mentioning) whereas program proof is very deep.

It is important to note that *depth* is not just a single dimensional measure; the fact that an algorithm has been proved to be mathematically correct does not guarantee other properties which may be of interest such as maintainability, portability or efficiency.

We now consider some examples to illustrate current practice.

2.1 Language code auditing

The idea that computer software should be used to analyse source programs rather than compile them, has a history of at least 25 years. The PFORT program was designed to locate potential problems with the portability of Fortran code. Vendors had distinct dialects, and normal compilation did not indicate the program was correct according to the standard.

All high-level language code *is* audited by at least one compiler. In consequence, the requirement is to analyse the code in ways the compiler does not, such that significant benefit is obtained.

This implies that the auditing that is useful depends on the requirements of the corresponding language standards. Implicit declaration in Fortran can lead to undetected program errors, and therefore a tool to list implicitly declared variables can be useful (although this feature is included with better quality compilers). On the other hand, such analysis is not required with most other languages, as declarations must be explicit.

The *nature* of several features of language auditing tools varies as follows:

- layout analysis and reformatting.
- identification of language constructs known to be non-portable.
- error detection not performed by compilers.
- flow-control analysis.
- detection of the use of data before a value has been assigned to it (an important special case of error detection).

For the deeper analysis features, many commercial products use a common analysis phase after converting the source into a standard form. This allows the tools to support several languages, but may reduce their effectiveness on aspects that are specific to just one language. (An interesting example of a language-specific analysis is that of exception propagation in Ada given elsewhere [4]).

Several commercial products are available for the analysis of C code. However, the weak type checking and lack of dynamic checking in C implies that modest tools are of benefit here. Such tools would not be useful for the more strongly typed languages such as Pascal, Ada and Modula-2; hence the importance of the choice of language for critical systems choice [5]. The fact that the primary purpose of one tool for C code is to detect the overwriting of the operating system indicates the difficulty of proving the integrity of such code.

For languages with insecure compilation of components, a very useful tool is one that checks whether the components of a program, prior to their integration, are compatible at the language level. This clearly requires source text control of the components, and an appropriate inter-procedural analysis. For C, `make` performs part of this, but even with function prototypes in ANSI-C, the analysis is far from complete. In contrast, Ada and Modula-2 require that this analysis is performed as part of the compilation process. The impact of languages on program quality is analysed in more detail elsewhere [6].

The industrial use of language source code auditing is mixed. Use of tools to establish portability is very widespread as users accept that testing on a large number of different platforms is impossible. However, the use of static analysis to establish the 'quality' of the source code is quite rare. A major area of concern is that there seems to be little appreciation of the differences in the strength of the various tools on offer.

2.2 Sizewell code analysis

This example of static analysis is an overview of the work undertaken to assess the conformity of a large PL/M program with its specification. Using the terminology above, this analysis is quite *deep*. The software is the Primary Protection System for the Sizewell Pressurized Water Reactor. This Section has been summarised from previous work [7].

The software contains about 100 000 lines of PL/M, which were to be checked against an English specification. It is therefore clear that the process could not be entirely automatic. The primary analysis makes use of the MALPAS tools, which were developed in the UK initially at RSRE Malvern (now DRA) and then at TA Consultancy Services Ltd. MALPAS is a tool kit which is capable of quite deep analysis, including formal proof (the compliance analyser).

The MALPAS tools require that the source code is translated into an intermediate code for subsequent analysis. For Sizewell, the analysis was undertaken bottom-up, starting from those procedures which called no others. To each procedure was added a specification of its input and output parameters, which involved some information not necessarily directly available from the source code. The analysis then produced the output values as functions of the input values for each procedure.

A number of problems were faced with this analysis.

- ☐ There is no precise definition of the PL/M language which would have provided a better foundation for the PL/M to intermediate language translator.
- ☐ MALPAS was unable to handle certain PL/M constructs directly, such as pointers (see below).
- ☐ The complete system was too large to handle in one analysis, and therefore the results from individual procedures had to be combined.
- ☐ As the requirements were informally expressed, manual analysis of the MALPAS results was required (with some subjective judgement).

All *deep* analysis tools have problems with pointers, due to the potential for aliasing. This problem is compounded with PL/M, as references are used for parameter passing. Fortunately, many such uses can be shown to involve the input of a value or the output of a result. However, a major use of explicit references in the PL/M source code is to refer to large tables giving the configuration of the system. These references can be handled by replacing the references by the object being referenced (as they are constant).

The first stage of the analysis involved the less *deep* methods: control flow, data use and information flow analyses. The first two are conventional and the last one reveals those inputs that affect each given output.

The *deepest* analysis is that for compliance against the specifications. Here, manual input is required to provide pre- and postconditions for procedures, and assert statements to cut loops (i.e. provide loop invariants so that the property of the loop can be verified).

Areas that have proved difficult to handle were double length arithmetic and the use of shared memory. Naturally, some areas were not been assessed with the MALPAS tool, and hence have been addressed by other means.

The validation required that all potential issues were handled formally. No errors were found that would have made the system unsafe, and only 6% of issues were categorised as a non-critical code change with the rest being documentation anomalies.

2.3 Sizewell object code validation analysis

For some applications, total dependence on the compiler is inappropriate. In such cases, some means must be introduced to validate the object code produced by the compiler. As compilers are known to contain errors, some protection against these would appear to be a wise precaution.

For the Sizewell Primary Protection System, an analysis of the PROM contents was undertaken [8] against the corresponding source text (mainly PL/M-86 code, as used in Section 2.2). This summary provides information which will be available shortly in a conference proceedings. Here, the two sources of information (source file and PROM) were unambiguous, and hence, at least potentially, the comparison could have been entirely automatic.

The analysis used the same MALPAS tools, but in a slightly different way. The PROM contents were analysed by a number of tools to restore a near equivalent of assembly language. This output contained the source code names of locations, deduced from the assembler tables produced with the PROM and checked independently by variable placement based on source code analysis.

The PL/M and the assembler listings were separately translated into MALPAS Intermediate Language (IL). These forms of IL should be the 'same' when analysed by the algebraic simplifier. However, the comparison had to be undertaken at code segment level, which requires that nodes in the code were marked as boundaries between segments. The marking of nodes sometimes required manual assistance, but the comparison of simpler procedures was entirely automatic.

This process was used on 100 000 lines of PL/M code to reveal two 'bugs', in the sense that the PL/M did not match the object code. In both cases, the incorrect object code would not have made the system unsafe. One of the bugs was in register handling, a common cause of compiler bugs. The other bug was that the PL/M compiler accepted the comparison of two pointers (`ptr1 < ptr2`), even though this did not seem to be allowed according to the user guide. Interestingly, this compiler bug would have been detected by the conventional compiler validation services used for Pascal or Ada. A number of aspects of the code comparison needed to be carefully considered in the context of critical systems. A successful comparison would not guarantee that the object code was correct. For example, the algebraic comparison did not model all features of the finite arithmetic used in the actual code. Work by Pavey and Winsborrow [8] should be consulted for details of the modelling method and for a deeper understanding of the strength of the comparison.

2.4 Dungeness 'B' Single-Channel Trip System

The Single-Channel Trip System (SCTS) for the Dungeness 'B' Advanced Gas-cooled Reactors is a small microprocessor-based protection system designed by AEA Technology and based on the Inherently Safe Automatic Trip (ISAT) concept [3, 9]. Full static analysis including compliance analysis (formal proof) was performed using the MALPAS toolset by Rolls Royce & Associates under the direction of Nuclear Electric. An informal specification was available and it was necessary to derive a formal specification from this in order to proceed with full static analysis.

In this paper, we are not considering the validation of the formal specification itself, but the subsequent of showing that the software conforms to the formal specification. The software was coded in Intel 8086 and 6809 assembly language, with the advantage that static analysis was performed at a level very close to the machine code and the disadvantage that, because the code under analysis was low-level, the modelling was necessarily detailed and complex. No anomalies compromising safety were found. A path in the code was revealed which would have activated the SCTS trip if the cubicle temperature had fallen below freezing. As the cubicles are kept warm by the reactor, this would only have been a problem if the reactor had been shut down for some while and start-up was required on a very cold day. This error had already been found independently by AEA Technology.

In addition, a full comparison was performed between source code and PROM files using purpose-built source/code comparison tools written by Nuclear Electric. This was more straightforward than the source/code comparison on the Sizewell 'B' PPS code described in the Section 2.3, as the mapping between source and code was one-to-one.

The results of the static analysis described above were incorporated as part of the safety case of the SCTS and contributed to Nuclear Electric's successful submission to the Nuclear Installation Inspectorate. The SCTS is now in use; it is the first microprocessor-based protection system to be licensed for use in one of Nuclear Electric's power stations.

2.5 SPARK development tools

The SPARK system is designed to aid the development of critical code in Ada 83. The main tool in the system is the SPARK examiner, enforcing a small subset of Ada which makes some forms of analysis much easier. The tool also requires that some annotations are added to the program text. These annotations provide a form of weak specification of the functionality of each subprogram, but they are also needed to allow important properties of the SPARK subset to be checked by simple linear analysis, in particular the absence of side-effects in functions and of aliasing between parameters of a subprogram. Both aliasing and side-effects may cause Ada programs to be erroneous without being detected by an Ada compiler. Their absence is also a prerequisite for the validity of the later stages of analysis performed by the SPARK Examiner.

In essence, the objective with SPARK is to allow properties of programs to be composed from the properties of the constituent subprograms. For details of SPARK, see work by Carré [10, 11] and for an appraisal of some aspects of the language, work by Wichmann [12].

SPARK has been quite widely used in the UK, almost entirely on safety systems. The benefits are that the less *deep* forms of static analysis are an automatic consequence of the subset. Hence access to an unassigned variable is impossible. In particular, control flow analysis is no longer required (it is subsumed into the grammar), and data and information have simple recursive formulations allowing them to be performed as the language is parsed. (In fact, data flow is subsumed into information flow [13]). The most important property which is needed from a SPARK program, although not a consequence of the subset, is that of being exception-free (i.e. not dynamically breaking certain language rules).

Proving that a SPARK program is exception-free would be possible with a tool being developed for SPARK [14], but this could require additional annotations, and may be expensive to undertake. An existing tool can be used to verify properties of SPARK code [15].

As SPARK is an Ada subset, a conventional Ada compiler can be used for code generation. Indeed, SPARK does provide some escape mechanisms so that a complete system does not need to be written in the subset. One potential problem arises if a design tool is used to produce some of the Ada, which then does not fit within the subset.

To summarise, SPARK Examiner is very much a forward engineering tool (i.e. for program development rather than validation after development), and hence cannot be applied to arbitrary Ada code. However, the integration of the automatic forms of static analysis into the development process is claimed both to increase their benefit and reduce their cost.

2.6 *Assembly code proof in avionics*

A *deep* analysis of assembly code subroutines has been carried out for a civil aviation manufacturer seeking compliance to DO-178B. The subroutines, part of an engine control unit (FADEC), have been proved to implement a specification written in predicate logic using an approach essentially similar to Hoare logic. This work, and other similar analyses, has been carried out using the SPADE tools, developed in the UK initially at Southampton University and then at Program Validation Ltd. O'Neill *et al.* [16] describe the application and the process in detail.

The steps of this form of analysis are as follows.

2.6.1: The assembly code subroutine is translated into the formal description language (FDL) accepted by the SPADE tool, either using a translator specific to the assembly code (in this case Zilog Z8000) or by hand. (All the other steps are independent of the particular assembly code.)

The basis of this translation is a formal model of the assembly code specified in FDL. Only a subset of the assembly language can be accepted, for two reasons. First, some instructions, such as those dealing with interrupts, cannot be modelled using this approach. Secondly, rules must be introduced which allow both the control flow of the subroutine and the data variables used by it to be identified statically. For example, data may only be referred to using symbolic addresses; an annotation may be required to describe the properties of the data variable corresponding to each symbol. As these rules are similar to a typical 'code of practice' for writing assembly code, their adoption is easier than might be imagined; indeed the advantages of a tool to check adherence to the code of practice are readily accepted.

Clearly, assembly code programs cannot be written without explicit manipulation of memory addresses; this is accommodated by requiring an annotation to identify the 'array' (i.e. a contiguous region of memory) accessed by each instruction which uses indirect addressing; the correctness of these annotations are checked during the final stage of the process.

2.6.2: Analysis of the control, data and information flow of the subroutine is then carried out on the FDL model of the subroutine, using the SPADE flow analysers. This step is important for two reasons: first, flow analysis is largely automatic and so provides an inexpensive way to reject code with gross errors at an early stage. Secondly, the later (*deeper*) steps of the analysis may not be valid in the presence of certain forms of flow error.

2.6.3: The functionality of the routine can now be specified using a precondition and postcondition written in a formal notation (essentially a first-order predicate logic with a simple type system). Typically, these formal specifications must be derived from informally written functional specifications, which may require some dialogue with the originator of the code. Until formal specification is more widely practised as part of the development process, the difficulty of this step limits the applicability of this form of analysis.

2.6.4: The SPADE Verification Condition Generator is then used to generate the theorems, which must be proved to show that the routine correctly implements the specification. If the routine includes loops, invariant predicates must first be added using annotations. This step differs from the previous one because these invariants form the first stage of the correctness proof rather than being part of the specification of the routine.

2.6.5: A complete machine-checked proof of each correctness theorem is then constructed interactively, using the SPADE Proof Checker.

So far applications of these techniques have analysed only single subroutines without subroutine calls. There is nothing preventing their application to multiple levels of subroutine except the point already noted; the difficulty of formally specifying pre-existing programs.

The validity of the analysis relies on a precise understanding of the semantics of the particular processor; as manufacturers do not (yet) publish formal descriptions of functionality, the formal model has to be derived from informal documentation.

Even more critically, the analysis relies on the definition of the subset, the enforcement of its rules and the accuracy of its model. For example, it is well known that the standard rules of a Hoare logic are not valid in the presence of aliasing (i.e. the use of multiple names for the same variable). As not all the assumptions of the model can be enforced automatically, a rigorous process with appropriate review is a vital component of the analysis.

Similar forms of analysis have also been carried out on industrial software for Motorola 68020 and other processors. A somewhat more formal approach, using the Boyer-Moore theorem prover, is described elsewhere [17].

2.7 *Accredited testing*

There is a key question posed by potential users: 'is the software appropriate for this critical context?' The context for the most critical software, for which static analysis is often used, is either safety or security. To provide any basis for the answer to such a question requires a deep understanding of the software and the context of its use. Inevitably, the answer must be a matter of professional judgement. However, if major features of the software can be determined by objective testing, then the subjective nature of the final judgement will be easier to justify and defend (which could involve the courts in the case of significant damages for safety systems).

International standards for the accreditation of testing involve assessment of the operations of testing laboratories, according to ISO Guide 25, or EN45001 (or NAMAS/M10 in the UK). These standards ensure that testing (or measurement) is carried out in an objective, repeatable fashion, which therefore allows other (accredited) laboratories to compete without jeopardising the quality of the work.

NAMAS has recently awarded ERA Technology the first accreditation for testing safety-critical software. Five of the procedures used by ERA fulfil the requirements of NAMAS for objectivity, although none of them are defined in existing standards. Laboratories may test products to in-house procedures when suitable standards do not exist.

Four of the five procedures within the scope of the ERA accreditation involve static analysis as follows:

- identification of unstructured constructs.
- static analysis of data use.
- determination of worst case execution times.
- static analysis of module calling.
- dynamic testing, not relevant to this survey.

In the ERA case, there are relationships between the procedures in use. Hence the data analysis depends on the absence of unstructured constructs and having analysed the module calling structure. In addition, the information gained from the static analysis is used in the dynamic testing. The methods used involve much manual analysis and therefore are complex to administer for larger items of software. Nevertheless, there is scope for automation using simple techniques which can themselves be checked rigorously.

Clearly, other procedures, which are likely to be more subjective, are needed to determine the suitability of an item of software for a safety-critical application. If procedures such as these could be standardised, then regulatory agencies would have a better foundation for the certification of systems, or authorising their use. In the context of the Single Market, a European scheme is needed, which should build on the widely recognised Germany scheme based on VDE801.

For the systems provider, testing undertaken by an accredited, independent testing laboratory has the advantage of providing third party assessment of the testing process, which must afford a degree of legal protection.

3 Problems and future trends

The absence of adequate standards or definitions for static analysis is a barrier to any comparison of the effectiveness of its use in producing critical systems. Unfortunately, producing an accurate, effective standard would not be easy. To illustrate this, consider a basic question on which much subsequent analysis can depend:

'Is the control flow of the program well defined?'

For the very simplest programs, this question can be answered with modest analysis. Now consider the Pascal fragment:

```
var
    B Boolean;
...
case B of
  true: ...
  false: ...
end;
```

This example can cause some systems to crash due to a flow control problem. The reason is that if B has not been assigned a value and the bit-pattern is neither true nor false, the program counter for the case statement makes an uncontrolled jump. An inspection of the code generated by the compiler would reveal this problem but that is not the natural method of resolving such issues. This example shows that analysis of control flow can depend on data flow.

The above example raises the issue of analysis at different levels, in this case, at programming language level or machine code level. Compilers often insert code that is not actually required in an application, because the compiler does not have the information needed to remove it. Static analysis may then reveal the unnecessary code, which can cause a conflict if the programming rules require that no such code is present [2]. Some compilation systems are specifically designed to exclude unnecessary code [18].

Program analysis will depend on the semantics of the language and, as noted above, the strength of the analysis may well depend on subtle features of the language. Even apart from the language, the strength of static analysis tools varies considerably. Hence even a means of quantifying the *depth* would be helpful.

From the perspective of the application, the concerns are different. For example, design and development methods could be rigorous enough to avoid the use of a conventional language, as with the B-Methodology [19]. Some safety and security standards do have requirements on languages; details are to be found elsewhere [20].

3.1 Ada 9X safety and security annex

The production of the Ada 9X Safety and Security Annex [21] required a study of how static analysis could be made more effective. First, some of the known insecurities in Ada 83 were simply removed in Ada 9X [22]. Secondly, features were added to the Annex to aid validation:

☐ The user can state that certain language features are not being used in the entire program. This aids static analysis and also provides the compiler with information to avoid including unwanted machine code.

☐ A requirement to provide transparency between the Ada 9X source code and the actual machine code has been added. It is then easier to relate an analysis at one level to the program at the other level.

3.2 *Effective formal definition of programming languages*

Accurate static analysis of a program text depends on a precise definition of the programming language in question. Unfortunately, interpreting the semantics of all current programming language standards depends, to a certain extent, on subjective judgement. This can create uncertainty which can make static analysis less effective, or worse, mean that the software does not have a well defined behaviour. You can produce a description of a language just for the purpose of program validation [23], but such a description should be part of the standard or at least widely reviewed. Currently, a formal definition [24] of the SPARK subset of Ada 83 is being developed specifically with the aim of overcoming such problems.

3.3 *Objective assessment of validity of analysis*

Many forms of analysis, especially the *deeper* types, are carried out on a model of the program, intended to represent only the properties required for the analysis. For example, all binary arithmetic operators can generally be considered equivalent for flow analysis. We need to be able to assess objectively the adequacy of this model for the intended analysis.

Aliasing of variables, which can serve as an example of the difficulties which must be faced, occurs when two variables refer to the same area of memory (either wholly or partly) and therefore behave as a single variable. Aliasing does not affect the validity of control flow analysis; however, as a modification to a variable changes the values of all its aliases as well, data flow analysis is affected. Two basic approaches tackle this difficulty:

- take account of aliasing in the calculation of data usage
- show that aliasing does not occur as part of the complete analysis.

We cannot argue the superiority of one approach against the other for all forms of analysis and situations in which it may be applied; however, returning to the example, it is clear that data flow analysis is not rigorous unless it allows for aliasing or excludes it.

A precise definition of the conditions required to ensure the validity of an analysis is complicated because many sources of difficulty are specific to different languages. Aliasing, for example, can arise from `COMMON` blocks in Fortran, from multiple forms of addressing (e.g. numeric, symbolic, relative) in assembly code and from variant records in Pascal.

Even when it is chosen to exclude a potential difficulty from the program under analysis, it may not be possible to do so in a fully automatic way; this means that a precise understanding of the limits of the analysis is required by all involved.

4 Conclusions

Our conclusions are as follows.

☐ Static analysis is effective and complementary to dynamic testing. Hence its use is to be recommended in the context of the majority of critical software. The less *deep* analysis methods which do not require extensive design information could be used for almost all software.

☐ There are no appropriate standards. This implies that the specification of the static analysis to be performed requires care and effort. Equally, the potential benefits from such analysis cannot always be readily appreciated.

☐ The depth and nature of an analysis need to be specified. To say that static analysis has been undertaken is just as meaningless as saying that the software has been tested. Hence at least these two issues (of depth and nature) need to be specified to some extent. The prerequisites for the validity of different forms of analysis must be recognised, in order to allow an objective assessment of the validity of a particular application of an analysis technique.

☐ The input language influences the *depth* of the static analysis that can be undertaken easily. Languages that are essentially dynamic, like C, are more difficult to analyse than languages, like Ada, that include strong typing and range constraints. Hence, the nature of the input language needs to be taken into account in the specification of the static analysis to be undertaken. The use of more rigorous 'software engineering' languages and appropriate subsets can allow the less *deep* forms of analysis to be subsumed into the rules of the language. Just as type checking is an integral part of a modern language, so we should expect that separate steps of control and data flow analysis should no longer be required.

☐ Problems of scale: unlike compilation, some of the *deeper* static analysis methods are essentially polynomial in time (or space). This implies that the effort (both human and computer) to analyse a large program can become significant; the effort expended on the Sizewell 'B' PPS demonstrates this.

☐ Reverse engineering versus forward engineering: the deeper analysis methods typically require additional information. Such information gives details of the design not immediately apparent from the source, and for the *deeper* forms of analysis, will include the specification. This can be planned in advance with tools aimed at supporting development or produced during analysis with tools that use reverse engineering. This choice must be made very early in development.

☐ Ability to exploit design information: an analysis tool should be able to check whether functions have side-effects as their presence may make further analysis impossible. Hence, conflicts can arise between the actual code and the ability of a specific tool. Such conflicts must be resolved very early in the life-cycle. Increased integration of certain forms of analysis into the development process is required to reduce their cost and increase their benefit. In particular, it is widely agreed that the *deeper* forms of functional correctness analysis using proof cannot be fully exploited retrospectively.

☐ The provision of the program specification in an unambiguous machine-processable form is useful. *Deep* static analysis can verify aspects of the program specification, but can only be undertaken with reasonable effort if the program specification is available in a suitable form.

□ Objective testing versus assessment: the user often pose to a question like 'is it safe to use this signalling software?' The conventional method of assessment is based on professional judgement as well as testing (and static analysis). Subjective judgement cannot be avoided, but more use could be made of objective testing, preferably by accredited testing laboratories.

5 Acknowledgments

The authors would like to thank Mr. R. Scowen, Dr. P. Vaswani (NAMAS) and Mr. N. North (NPL) for their useful comments on an earlier drafts of this paper; and the referees whose comments have improved the clarity significantly.

6 References

[1] Interim Defence Standard 00-55: 'The procurement of safety critical software in defence equipment', Ministry of Defence, (Part 1: Requirements; Part 2: Guidance), April 1991

[2] Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B); European Organization for Civil Aviation Electronics (EUROCAE document ED-12B)

[3] HUGHES, G., and BOETTCHER, D.B.: 'Developments in digital instrumentation for Nuclear Electric's (UK) power plant', *Nuclear Energy*, 1993, **32,** (1), pp. 41–52

[4] SCHAEFER, C.F., and BUNDY, G.N.: 'Static analysis of exception handling in Ada', *Softw. Pract. Exp.*, **28,** (10), pp. 1157–1174

[5] CULLYER, W.J., GOODENOUGH, S.J., and WICHMANN, B.A.: 'The choice of computer languages in safety-critical systems', *Softw. Eng. J.*, 1991, **6,** (2), pp. 51–58

[6] WICHMANN, B.A.: 'The contribution of standard programming languages to software quality', *Softw. Eng. J.*, 1994, pp. 3–12

[7] WARD, N.J.: 'The rigorous retrospective static analysis of the Sizewell 'B' primary protection system software'. SafeComp '93

[8] PAVEY, D.J., and WINSBORROW, L.A.: 'Demonstrating equivalence of source code and PROM contents', *Computer J.*, 1993, **36,** (7), pp. 654–667

[9] SMITH, I.C., and WALL, D.N.: 'Programmable electronic systems for reactor safety', *Atom*, 1989, **395,** pp. 10–13

[10] CARRÉ, B.A., JENNINGS, T.J., MACLENNAN, F.J., FARROW, P.F., and GARNSWORTHY, J.R.: 'SPARK — the SPADE Ada kernel'. Version 3.1. Program Validation Ltd., May 1992

[11] CARRÉ, B.A., GARNSWORTHY, J., and MARSH, W.: 'SPARK: a safety-related Ada subset: Ada in transition' *in* TAYLOR, W.J. (Ed.) (IOS Press, 1992)

[12] WICHMANN, B.A.: 'Strategy on the use of SPARK'. NPL Report DITC 227/94, June 1994

[13] BERGERETTI, J.F., and CARRÉ, B.A.: 'Information-flow and data-flow analysis of while-programs', *ACM Trans Prog Lang.*, 1985, **7,** pp. 37–61

[14] GARNSWORTHY, J., O'NEILL, I., and CARRÉ, B.: 'Automatic proof of the absence of run-time errors. Ada: towards maturity', COLLINGBOURNE, L. (Ed.) (IOS Press, 1993) pp. 108–122

[15] Generation of Path Functions and Verification Conditions for SPARK Programs, Edition 1.1b. Program Validation Ltd., January 1992

[16] O'NEILL, I.M., CLUTTERBUCK, D.L., FARROW, P.F., SUMMERS, P.G., and DOLMAN, W.C.: 'The formal verification of safety-critical assembly code. safety of computer control systems' (Pergamon Press, 1988) pp. 115–120

[17] BOYER, R.S., and YUAN, Y.: 'Automated correctness proofs of machine code for a commercial microprocessor: automated deduction', *Lect. Notes Artif. Intell.*, 1992, **607**

[18] BRYGIER, J., and RICHARD-FOY, M.: 'Ada run time system certification for avionics applications.' Ada-Europe Conf., June 1993

[19] CARNOT, M, DA SILVA, C., DEHBONEI, B., and MEJIA, F.: 'Error-free software development for critical systems using the B-methodology'. Third IEEE Int. Conf. on Software Reliability, October 1992, pp. 274–281

[20] WICHMANN, B.A.: 'Requirements for programming languages in safety and security software standards', *Comput. Stand. Interfaces.*, 1992, **14,** pp. 433–441

[21] WICHMANN, B.A.: 'Programming critical systems — the Ada 9X solution', *Comput. Bull.*, 1993

[22] WICHMANN, B.A.: 'Insecurities in the Ada programming language'. NPL report 137/89, January 1989, p. 54. NTIS ref: PB89-193627/WFT. (summary with Dawes, S. J.: *Ada User*, 1990, **11,** (1), pp. 21–26)

[23] BOYER, R.S., and MOORE, J.S.: 'A verification condition generator for FORTRAN: the correctness problem in computer science' (Academic Press, 1981)

[24] Formal Semantics of SPARK. Program Validation Ltd., Version 1.1. April 1993

IEC 880:86: 'Software for computers in the safety systems of nuclear power stations'. 1986

IEC/SC65A/(Secretariat 122): 'Software for computers in the application of industrial safety-related systems'. Draft, December 1991

The paper was first received 21 July 1994 and in revised form on 27 January 1995.

B.A. Wichmann is with the National Physical Laboratory, Teddington, Middlesex, TW11 0LW, UK; A.A. Canning and D.W.R. Marsh are with ERA Technology, Leatherhead, KT22 7SA, UK; D.L. Clutterbuck is with Program Validation Ltd, Southampton, SO1 1BQ, UK; L.A. Winsburrow is with Nuclear Electric, Barnwood, Gloucester, GL4 7RS, UK; N.J. Ward was previously with TA Consultancy Services Ltd, Farnham, GU9 7TB, UK.