Model-driven Automated Software FMEA

Neal Snooke, PhD, Aberystwyth University
Chris Price PhD, Aberystwyth University

Key Words: Failure modes and effects analysis; software FMEA; model-driven software development

SUMMARY & CONCLUSIONS

This paper describes how software FMEA can be automated both for low-level languages intended for safety critical embedded systems, and also for model-driven software developments.

It is possible for a computer to achieve a qualitative analysis of software based on tracing dependencies through a body of code. This can reveal the propagation of any failure in the software, whatever the cause of the failure. Application of a higher level representation of the intended purpose of the software can then automatically interpret the implications of failure in terms of the requirements put on the software.

These techniques have been used to automate the analysis of several thousand lines of code. They have been shown to provide useful results for software engineers, and would suit embedded software in vehicles for example. This work is not a cure-all for badly written software, but provides assistance in software analysis for well designed systems in low-level "safe" languages such as MISRA C. The software FMEA can be used to improve automated or source code embedded testing since tests can exonerate many potential faults allowing the FMEA analysis to present an engineer with a reduced set of potential faults.

Model-driven development (MDD) is a software development philosophy which encourages the development of models of the software to be produced, for example using a language such as executable UML. The system is described in a platform independent manner, and then the software to be used is automatically generated from the model.

In MDD, the models make the intentions of the programmer much more explicit than is the case for low-level programming, and so the gap between the intended functions of the system and the description of the software is not so large. Representation of the design is much more explicit through use cases, component diagrams, state charts and sequence diagrams. All of this design information can be utilized for the automated generation of software FMEA. This means that FMEA for model-driven software can be done more easily than for a system implemented in a low-level language, because it is not necessary to attempt to reconstruct the intentions of the programmer from the functions of the system and the low-level code. The paper also discusses the advantages and dangers of doing such analysis at the design rather than the code level.

1 INTRODUCTION

Typically, FMEA is practiced on physical systems, and the failure modes considered are the failures of physical components, caused by wear or other damage to the system. Since software has been introduced into automotive and aeronautic systems, it has often been included in FMEA reports as a component with no failure modes (the ECU containing the software), and the system design FMEA has been produced assuming that the software works correctly.

The concept of software failure mode and effects analysis (software FMEA) has grown in attractiveness over recent years as a way of assessing the reliability of software. Like its hardware counterpart, software FMEA is immensely tedious for an engineer to perform, as well as being error-prone. Clearly, software components do not fail in the same manner as hardware components — a function or method does not break over time because it has become worn or damaged. Software FMEA considers all potential faults such as faulty inputs or software bugs (mutations) that could exist and ensures the worst-case consequences are known, possibly prompting actions to reduce risk. A software bug may be treated analogously to a hardware component failure, the essential difference being that hardware failures occur over time whereas software bugs exist undetected but usually only affect a very small (untested) region of the overall system behaviour.

One of the unique difficulties with software systems is the complex relationship between faults and effects. A minor fault can, for example, cause a complete crash of a software system or have almost invisible but very complex, subtle, and long lasting side effects. The result is that software often has very non-uniform quality in terms of the effects of potential failures, and it is not clear when effort is available, where it should be expended to improve quality. An FMEA provides just this information allowing targeting of effort at the highest risk areas

Code-level software FMEA has been performed for some years [1, 2, 3, 4], but has been considered impractical except when applied to small pieces of highly critical code, because of its cost. On the other hand, software FMEA of a more abstract specification of the system can ignore important implications of failures, especially where code is not automatically generated from the abstract specification.

This paper describes work to automate much of the effort

of performing software FMEA. It describes a novel method for automating code-level software FMEA based on treating the implemented software as a model of the desired system and propagating faults through the model to identify dependencies. High level descriptions of the implications of each failure are produced through the use of functional descriptions of what the software is intended to achieve.

The extension of this work to address the analysis of software generated from executable models built in a language such as executable UML is described, and the implications of doing the analysis at such a high level are discussed.

2 ASSESSING SOFTWARE QUALITY

In much software engineering practice, engineers rely almost exclusively on quality standards that define development processes and practices in order to assess whether quality software has been produced. While development processes are vitally important in any engineering endeavor, they do not guarantee the safety of a product, as McDermid [5] has observed.

Testing is the primary (and often the only) evidence based analysis performed on software, and although testing can *initially* give indications of quality, once the results are used to improve the software, the tests are no longer a reliable indication of overall quality. The value of tests in assessing quality is further reduced by hierarchical testing, because lower-level modules will themselves have been made to pass a set of tests before being included in a system. It is likely that the system outside the scope of the tests performed has only the quality achieved at the very first execution of the code [5].

Put another way, testing only finds faults, it does not help prevent or assess them. The reason for this is clear. Tests can only ever verify a tiny sample of the expected behavioral envelope of the software and they are often focused on verification of nominal function, with perhaps a few limiting case examples.

Other domains such as electrical systems analysis also have an infinite number of possible numerical behaviors, but other kind of analysis are done. For example, failure effect analysis is carried out by abstracting the infinite number of possible behaviors into qualitatively similar regions, and concentrating on a worst case analysis.

All substantial software will enter unanticipated regions of behavior during its life, and all software will have to deal with external failures either from hardware or other software, and for most software there is limited understanding of how the system might behave. Formal methods are often proposed as the solution to these problems, but they can be difficult to use, specification capture remains an issue, and pragmatically these methods are too expensive for the majority of software. Software Integrity Levels are often used to demonstrate a specific level of integrity [6], however SIL level specification most often results in a mechanism to change product requirements into process requirements and for higher levels may also require formal methods.

These mechanisms are typically very expensive to implement and accordingly are used only for the highest risk systems, leaving the vast majority of software development relying solely on the talent and intuition of software developers along with testing to verify basic functionality.

Testing can be addressed by a variety of means, some of which begin to move towards a wider ranging type of analysis that can consider both more comprehensive and/or abstract behavioral considerations, for example functional analysis testing and cause-effect graph testing. Other types of testing such as statement coverage, path coverage, and modified condition decision coverage can help discover obvious inadequacies in test suites but they cannot reason about overall system behavior and potential failure modes or problematic operating states. Voas [7] proposed an analysis that identifies where faults are more likely to remain undetected, and although based on execution of test, sets it in the spirit of a broader assessment of quality.

The need for lower precision but higher coverage techniques that can reveal more general (good and bad) characteristics and potential problems in software has led to investigation of the adaptation of techniques from other areas such as electrical systems analysis to the needs of software analysis.

The next section considers developments in adapting failure modes and effects analysis (FMEA), originally applied to physical systems, to the task of software analysis, and describes how such analysis can be automated, providing useful results for little effort on the part of the engineer.

3 AUTOMATION OF SOFTWARE FMEA

Automation of software FMEA is inspired by success in automated electrical system FMEA [8], where automated analysis is now in routine use in the automotive industry [9]. In that work, the expected behavior of the system is simulated from a model of the design, and results are abstracted to the level of the function of the system.

A comprehensive set of possible failures is identified from the components that compose the structure of the system. For each possible failure, the simulation and abstraction is repeated, and the results with the failure present are compared with the results when no failure is present. The difference between the two sets of results gives the system-level effect of that failure. For electrical system FMEA, the software produces engineer oriented automated reports giving the consequences of each potential failure.

These techniques can be applied to software by treating a piece of software as a model of the desired system. It is then possible to reason about the ways in which faults can be propagated through the model in order to derive the possible causes of a given set of symptoms.

There are three facets to the production of a useful report on failures in a piece of software:

1. Automated model construction. Techniques have been developed capable of automatically constructing a fault propagation model which can generate all possible effects of a failure from the piece of software being analyzed.
2. Injection and propagation of faults. All possible

faults that could occur in the software need to be identified, so that their effect on the overall system can be explored. Given a specific fault, the fault propagation model constructed automatically from the software can be used to decide what parts of the piece of software could be affected by a specific fault, and in what way they could be affected.

3. Identification of system level effects. Generating a list of all variables affected by a fault would be far too much detail to report to an engineer. The results are abstracted using the system requirements as a focus.

Each of these three facets will be explained in turn.

3.1 Automated model construction

The source code of the software to be analyzed is parsed and transformed into a fault propagation model. This is essentially a graph, where the source code statements are the edges and the variables are the nodes of the graph. It is then possible to use the graph to reason about how the effect of a fault can propagate through the program.

This work is clearly programming language dependent. In the work described here, model construction has been achieved for a large subset of the JAVA language, but the work would be equally applicable to other imperative languages. Limitations of this approach are discussed later, but generally correspond to the kind of coding restrictions recommended when constructing safety-critical software [10].

Fault propagation is complicated by the reuse of memory, and this problem has been overcome by using a Single Assignment Form (SAF) model [11] that transforms each memory write to allow symbolic memory locations to be logically written only once. SAF is used to generate a graph with distinct nodes for each value of a variable. This is illustrated for a simple code fragment in figure 1.

	source code	SAF	$\mathcal{FP}$ arcs
1	`a=1;`	a_1=1;	$(\emptyset, a_1)$
2	`b=2;`	b_1=2;	$(\emptyset, a_2)$
3	`a=a+b;`	a_2=a_1+b_1;	$(a_1, a_2), (b_1, a_2)$
4	`b=b*b;`	b_2=b_1*b_1;	(b_1, b_2)
5	`c=a*b;`	c_1=a_2*b_2;	$(a_2, c_1), (b_2, c_1)$

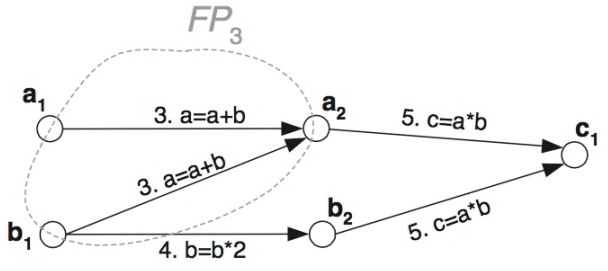

Figure 1: Code to propagation graph example

Fault propagation graphs of the kind shown in figure 1 can be constructed for much more complex pieces of code, and used to predict the worst possible implications of any fault in terms of which other parts of the software it will affect.

Fault propagation through non-sequential code, such as conditions and iterations can be simplified by assuming worst case scenarios for the propagation of faults [12]. When this assumption does not hold, either a set of faults will have no effect, or an effect will be generated where the engineer knows of application constraints that explain why the failure modes could not happen.

3.2 Injection and propagation of faults

All software faults can be characterized as faulty values:

- faulty I/O
- faulty transfer functions (whether due to wrong specification or wrong implementation)
- loss of transfer function (due to crashing or hanging)
- missing outputs
- unexpected outputs

In order to perform an FMEA, it is assumed that *any* value can become faulty and the propagation of that faulty value is explored to all statements that depend on it. This is achieved by propagating the faulty value using the fault propagation model already discussed, until the effects on system outputs have been derived. The scope and the path of the propagation define a failure mode.

3.3 Identification of system level effects

Functional interpretation [13] is a vital part of organizing and abstracting the results of structural and behavioral analysis into the form of an FMEA report able to flag significant potential problems. Information regarding the purpose(s) of the system is required, and a functional model is used to provide this information in a principled form. The functional model described in this section identifies the purposes of the system by means of associations with the system interface.

We use the following definition of function: "An object or system O has a function f if it achieves an intended goal (purpose) by virtue of some external trigger T resulting in the achievement of an external (behavioral) effect E."

The function interpretation language allows functions to be decomposed into subsidiary functions to build a functional hierarchy. A good deal of intuition regarding the function model can be obtained from an example functional description for a simple travel expenses payment program shown below:

```
FUNCTION pay_expenses
  ACHIEVES pay_money_owed
  BY run_program
    TRIGGERS display_expenses_total
    AND print_cheque

FUNCTION make_envelope
  ACHIEVES allow_delivery_cheque
  BY run_program
  TRIGGERS  address_envelope

FUNCTION display_expenses_total
  ACHIEVES verify_expense_items
```

```
 BY show_expense_items_on_screen
     AND
     show_total_on_screen

FUNCTION print_cheque
  ACHIEVES transfer_money
 BY print_in_figures
     AND
      print_in_text
      AND
      print_details
```

The functional model must be linked to the behaviour of the program via the causes (identified by "BY") and effects (identified by "TRIGGERS"). These will be system inputs and outputs and could be memory locations, or ports that influence external devices or system calls at higher levels.
The causes and effects from the function definitions need to be mapped onto variables in the program implementing the functions, either global variables or instantiated variables.

3.4 Application of these techniques

The techniques described here have been applied to a payroll program consisting of a few thousand lines of Java code. This has shown that the computation is tractable for useful sized programs, and that the results produced are useful when considering the reliability of software. Both the techniques used and the results obtained are described in greater detail in a longer paper elsewhere [12].

These techniques provide an evidence-based analysis that deals with a wide range of possible behaviors at a low level of precision. They consider hypothetical faults and determine the possible effects assigning significance to them. Any statement, module, or subsystem is assumed to have the potential to fail or produce incorrect output, and the techniques assess the potential effects.

The results can be used to identify design or implementation issues that might lead to faults with disproportionate or unexpected consequences, identifying areas of design or implementation that allow significant failures to occur. Engineering effort can then be expended to address the most significant faults.

The work described here does not cover all constructs in all languages. It would be impossible to apply these techniques to assembler programming, for example. However, modern software languages both encourage and enforce higher levels of structuring and this helps to inherently constrain faults. Decades ago, high level-languages introduced typed data and procedures to help structure data and code and these have been refined ever since. Object-oriented methods provide common structures to partition data with code. These techniques make software FMEA analysis feasible at the level of code because they constrain potential fault impacts.

The main limitations of this approach at present come from the need to trace dependencies between variables. This is impossible with dangerous programming constructs such as pointer arithmetic. In such cases, it would be impossible to accurately propagate faults - a fault could propagate to any part of the system. Some other programming constructs such as generation of dynamic structures, keeping track of variables in dynamic heap based structures, and recursion are still to be included in the analysis. However, the types of programming construct that cannot presently be addressed with this approach are similar to the types of construct that the MISRA C guidelines for safety critical embedded software recommend should be avoided [10]. Embedded software is a promising area for early application of this software FMEA tool, because of the safety critical demands and the limited, well defined external functionality. We are currently working on a further case study application in the embedded domain.

4 EXTENSION TO MODEL-DRIVEN ARCHITECTURES

Model driven software development (MDD) is characterised by descriptions of the operation of a system which can be transformed automatically into code to run on a specific platform using a specific architecture. The models in MDD explicitly represent the important aspects of the system. This contrasts with the work in the previous section, where a model of the functionality of the system is inferred from the code. In this case, the model is explicit, and so it would seem that application of the propagation and abstraction techniques described previously should be much simpler.

This section will examine the models used in MDD and where potential problems might occur in using them in an automated software FMEA.

The first assumption in this work is that generation of code from the models is completely automatic. If code is generated by hand from UML (as has often been the case in model-driven software design in the past), then correspondence between the model and the implementation would need to be proven in order to have any confidence in the analysis. Another level of compromise would be that platform dependent annotations are made to the models in order to generate code for a specific platform - those annotations would need to be taken into account in the analysis if they existed.

The second assumption is that the automated process of transformation to a platform dependent model does not itself introduce new potential for generating and propagating errors. This assumption is related to the assumption for a high level language that the compiler does not introduce errors to the program, although the potential for problems to be introduced in transformation is probably greater than the likelihood of a compiler error. Clearly people using such tools for important work should ask questions about the risks that the tools themselves introduce to the process.

If both of these assumptions were true, then the models would be a complete representation of the operation of the system, and could give a good indication of the potential for problems presented by the system.

4.1 Generating software FMEA from models

Let us say that the system is specified in executable UML

with a specific set of models. USE CASES are used to describe the purpose of the system. Class diagrams are used to describe objects in the system. The behavior of the system is described with lifecycle statements for each class containing state-based snippets of high level business logic describing the actions in each state.

The three facets of code-level automated software FMEA can be adapted to this situation in the following way:

1) Automated model construction. This stage has been made much simpler. The interactions between objects and variables are much more explicit in this representation, and can be extracted from the class descriptions, the lifecycle statements and the business logic. Complications are added by the dynamic objects, but an easy simplification of the analysis might be to consider only one instance of each class and the links between the classes. This simplification is likely to err on the side of caution in identifying potential problems.

2) Injection and propagation of faults. This does not change from the previous version.

3) Identification of system level effects. The USE CASE diagrams are useless in themselves, merely identifying the functions that are required. Typically, they would be fleshed out with a textual description of what is involved in each USE CASE. In order to be used for automated software FMEA, that description needs to be more formalized, ideally given as the kind of functional description that was used for the code-level software FMEA. If this is done, then that clear description of requirements can be linked to the model in the same way, and used to abstract results. It has further advantages, in that it clarifies the functional requirements of the system, and records them in a clear, unambiguous, executable form.

4.2 Challenges and potential

In some ways, automated generation of software FMEA from model-driven software is an easier task than it is for a system implemented in a low-level language. In a lower level language, it is necessary to attempt to reconstruct the intentions of the programmer from the functions of the system and the low-level code. Because the model-driven software expresses the intentions of the programmer much more explicitly, the gap between the intended functions of the system and the description of the software is not so large. The models are executable, typically expressed as state-charts, and it is possible to use those executable model descriptions to understand how the different parts of the system relate to each other, and to understand how failures might be propagated through the system.

A software FMEA of the platform independent model is potentially extremely valuable, as it can identify problems with the design of the system, as opposed to the way in which it is chosen to implement it. However, the models leave many of the lower level implementation decisions to the platform dependent code generation, and it is possible for extra problems and interactions to creep into an implemented system at that point.

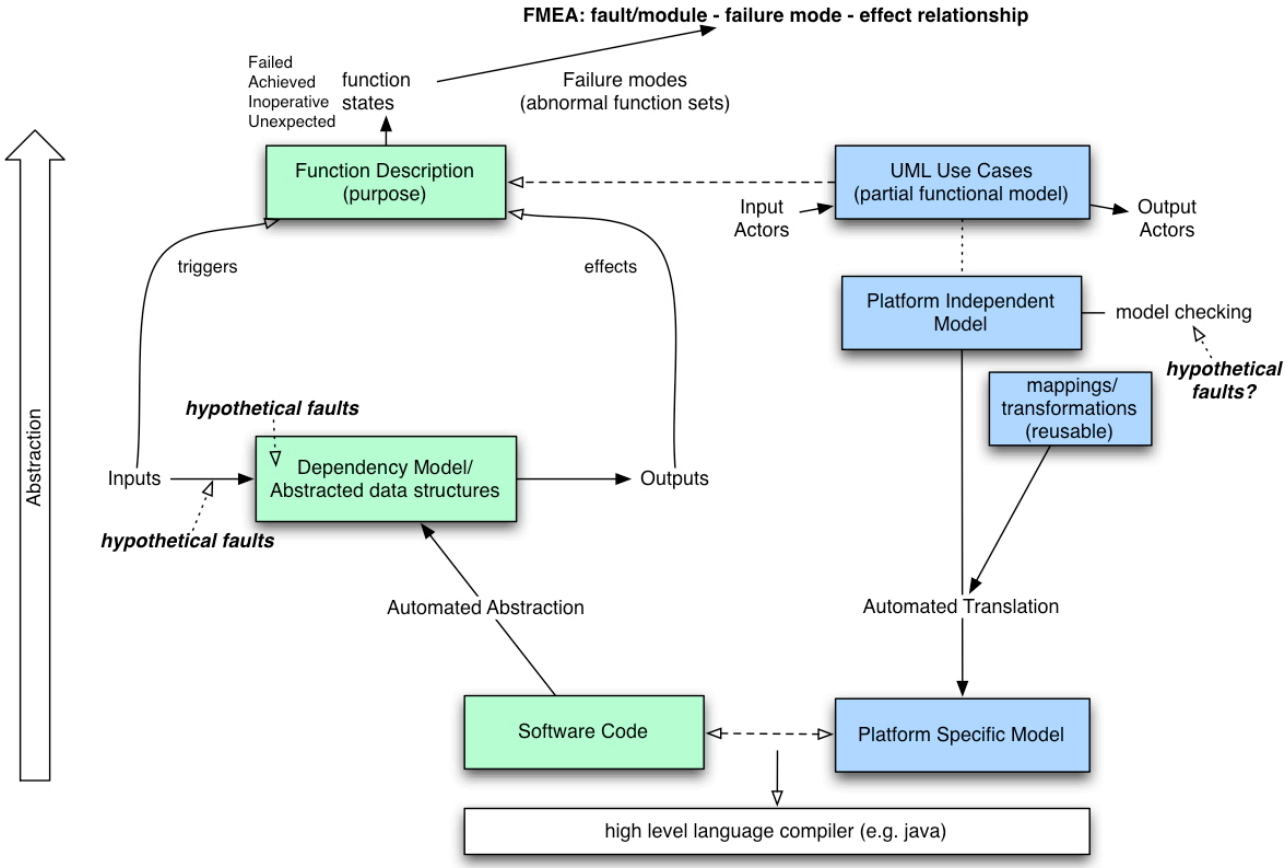

Figure 2: Possible architecture for automated software FMEA testing robustness of generated code

In order to assess the fault tolerance of the generated system, you would need to know the reliability of the transformation from model to code. The proponents of MDD claim that in future, we will have the same level of confidence in the model-to-code transformation as we presently have in compilers for high-level languages, but that is not presently the case. In addition, they typically think in terms of the correctness of the transformation, but even if the transformation is correct for the normal behavior of the system, that does not guarantee that it will be robust under failure. Faults often propagate through a software system because of the lack of checks that values are consistent and within sensible values. The level of confidence in a model to code generator should depend on the degree to which it had automated such protection against failure.

Where the transformation from model to code uses a standard high-level language as its target, then an intermediate scheme between just analyzing the high-level model and just doing low-level analysis on code might be possible as pictured in figure 2. Further confidence in the robustness of the designed system can be obtained by linking the analysis of the high-level model to the kind of low-level analysis of the generated system that our original system was capable of. The high-level model can be used both to focus the low level analysis and to decompose it into manageable pieces.

The combination of low-level analysis of the dependencies within specific code, and the high-level analysis of the dependencies within the abstract design provides the potential to efficiently produce a useful software FMEA for a complex system of a practical size, giving early warning of design problems but also taking into account the robustness of the generated code.

REFERENCES

1. J. B. Bowles, "Failure modes and effects analysis for a small embedded control system". *Proc. Ann. Reliability & Maintainability Symp.*, (Jan.) 2001, pp1-6.
2. T. Cichocki, J. Gorski. "Failure mode and effect analysis for safety-critical systems with software components." *SAFECOMP 2000*, vol. LNCS1943, 2001, pp382-394.
3. P.L. Goddard. "Software FMEA techniques". *Proc. Ann. Reliability & Maintainability Symp.*, (Jan.) 2000, pp118-123.
4. N. Ozarin and M. Siracusa. "A process for failure modes and effects analysis of computer software". *Proc. Ann. Reliability & Maintainability Symp.*, (Jan.) 2003, pp365-370.
5. J. McDermid. "Software safety: Where's the evidence?" *6th Australian Workshop on Industrial Experience with Safety Critical Systems (SCS '01)*. Australian Computer Society, 2001, http: //www-users.cs.york.ac.uk/ jam/.
6. Functional Safety of Electrical / Electronic / Programmable Electronic Safety-related Systems (IEC 61508). International Electrotechnical Commission, International Electrotechnical Commission, 3 rue de Varembé, Geneva, Switzerland, http://www.iec.org.ch.
7. J. M. Voas,. "Pie: A dynamic failure-based technique". *IEEE Transactions on Software Engineering*, 1992, vol18, pp717– 727.
8. C. J. Price, N. A. Snooke, S. D. Lewis. "A layered approach to automated electrical safety analysis in automotive environments". *Computers in Industry*, 2006, vol 57: pp451–461.
9. P. Struss and C. J. Price. "Model-based systems in the automotive industry", *AI Magazine*, special issue on Qualitative Reasoning, Winter 2003, vol 24(4), pp17–34.
10. ISO/TR 15497: 2000. Road vehicles – Development Guidelines for Vehicle Based Software. 2000.
11. J. Collard, *Reasoning About Program Transformations*. Springer, 2003.
12. C. J. Price, N. Snooke; "An Automated Software FMEA"; *Proceedings of the International System Safety and Reliability Conference*, (ISSRC 2008); Singapore; April 2008; *ISBN: 978-981-08-0446-6*
13. C. J. Price. "Function Directed Electrical Design Analysis", *Artificial Intelligence in Engineering*, 1998, vol 12(4), pp445–456.

BIOGRAPHIES

Neal Snooke, *PhD*
Computer Science Department
Aberystwyth University
Aberystwyth, SY23 3DB
United Kingdom

e-mail: nns@aber.ac.uk

Neal Snooke is a Lecturer in the Computer Science Department at Aberystwyth University. He has 15 years experience of working with automotive and aerospace companies building software tools that assist engineers by automating the design analysis of engineering designs. The two latest applications of such work have concerned the automated generation of on-board diagnostics from engineering designs, and the use of the techniques that have proved successful in other engineering domains for the analysis of software.

Chris Price, *PhD*
Computer Science Department
Aberystwyth University
Aberystwyth, SY23 3DB
United Kingdom

e-mail: cjp@aber.ac.uk

Chris Price is a full professor in the Computer Science Department at Aberystwyth University. He is a Fellow of the BCS, and a Chartered Engineer, He leads a research group concerned with the application of artificial intelligence techniques to engineering problems. The group has worked with many leading companies in the automotive and aerospace industries developing advanced software.