A Candid Industrial Evaluation of Formal Software Verification using Model Checking

Matthew Bennion
Controls Software
Rolls-Royce plc
Bristol, UK
matthew.bennion@rolls-royce.com

Ibrahim Habli
Department of Computer Science
University of York
York, UK
ibrahim.habli@york.ac.uk

ABSTRACT
Model checking is a powerful formal analytical approach to verifying software and hardware systems. However, general industrial adoption is far from widespread. Some difficulties include the inaccessibility of techniques and tools and the need for further empirical evaluation in industrial contexts. This study considers the use of Simulink Design Verifier, a model checker that forms part of a modelling system already widely used in the safety-critical industry. Model checking is applied to a number of real-world problem reports, associated with aero-engine monitoring functions, to determine whether it can provide a practical route into effective verification, particularly for non-specialists. The study also considers the extent to which model checking can satisfy the requirements of the extensive DO-178C guidance on formal methods. The study shows that the benefits of model checking can be realised in an industrial setting without specialist skills, particularly when it is targeted at parts of the software that are error-prone, difficult to verify conventionally or critical. Importantly, it shows that model checking can find errors earlier in the design cycle than testing, which potentially saves money, due to reduced scrap and rework.

Categories and Subject Descriptors
D.2.4 [Software]: Software/Program Verification – *Formal Methods and Model Checking.*

General Terms
Reliability, Verification.

Keywords
Formal Verification, Critical Software, Certification, Aerospace.

1. INTRODUCTION
Model checking is a formal technique for automatically exploring a model of a system to check if some properties of interest hold [1]. These properties could be failures that should never occur or functional requirements that must be satisfied. Model checking has been an active area of research for several decades [14] and numerous tools have been produced to automate the task. Some significant examples are SPIN [14], NuSMV [4] and Simulink Design Verifier (SDV) [18]. An important benefit of model checking is that it gives strong evidence that desirable properties hold true, compared to conventional testing. This lends itself to safety-critical software development, such as avionics and automotive applications, where it is traditionally hard to verify thoroughly that safety requirements have been satisfied.

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
ACM 978-1-4503-2768-8/14/05
http://dx.doi.org/10.1145/2591062.2591184

However, the use of model checking seems to be limited in industry. Some reasons are well established: state explosion [10], regulatory limitations [10], lack of relevant skills and inadequate model validation [24]. To overcome these limitations, testing on the actual system or additional analysis to compare the model to the actual system may still be necessary. Despite this, some look forward to a promising future [14].

Yet, while there is a wealth of literature on improved tools and mostly small-scale case studies ([7] [20] [21] [23]), there is less evidence of widespread adoption of model checking. One of the notable exceptions is [24], which describes a complete tool chain for model checking that has been used in industry. The authors consider the study a success because they found errors in the design, which could be corrected earlier in the product lifecycle. However, the technique involved 16 tools, some were custom-written, and took 5 years to develop. This might be more effort than some companies are able to invest. The technique was also used to augment, rather than replace, traditional testing.

To provide some context to our study, a lightweight survey of 21 engineers working in the safety-critical software industry was carried out, focusing on three questions:

1. Awareness of model checking;
2. Whether model checking should be used; and
3. Why they thought that model checking was not used.

The results of the first two questions are shown in Figure 1. It shows that awareness of model checking is fairly low, but most people think it is a good idea. The responses to the third question were verbose. Figure 2 depicts the frequency with which common themes occurred. The largest obstacle is the lack of knowledge and support from management, closely followed by the lack of familiarity and tools. This reinforces our view that one of the ways to realise the benefits of model checking in industry is to take advantage of the tools already available on engineers' desks.

The objective of this study is to explore the extent to which model checking can be performed *'usefully'* in a *'standard industrial environment'*. *Usefully* here means that model checking is cheaper, faster, more thorough than conventional testing or review or able to find more subtle errors. A *'standard industrial environment'* is one that uses tools readily accessible to engineers (e.g. commercial off-the-shelf tools, as opposed to research or

experimental tools), a model of the kind likely to be encountered in industry (i.e. not a modelling technique that requires expert knowledge) and contains errors typical of those encountered in industry (as opposed to problems of purely research interest).

This paper is organised as follows. Section 2 provides an overview of safety-critical software and certification in the aerospace domain. Section 3 introduces the data used in the study while Section 4 presents the detailed study process and results. Section 4 evaluates the results of the study. Related work is discussed in Section 5, followed by observations in Section 6 and conclusions in Section 7.

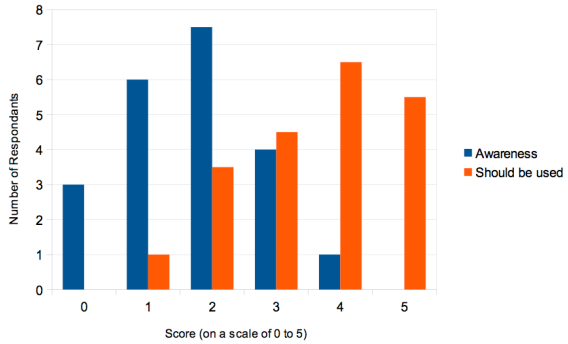

Figure 1: Views on model checking

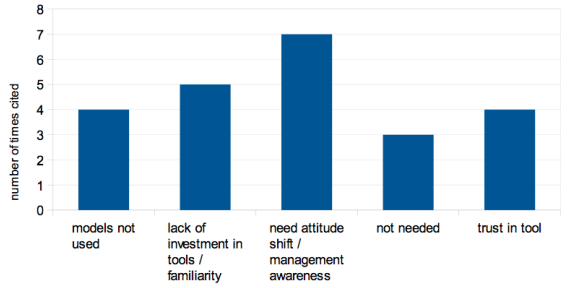

Figure 2: Obstacles to model checking

2. BACKGROUND

Software is often built into systems that may be, under certain failure conditions, hazardous, e.g. control systems for aircraft and medical devices [19]. A failure in the software can then affect the safety of people or property. Such software is termed *'safety critical'*. There are obvious ways in which software failures can contribute to harm, such as a failure in aircraft control software to command the required thrust. Knight points out less obvious examples, such as a failure of software controlling telephone access to the emergency services [17].

To regulate and assess software safety, authorities have created assurance standards (e.g. DO-178C [9] [12]). Generally, these standards identify different levels of criticality, reflecting the amount of damage that could be caused by a system that fails and therefore the amount of effort needed to reduce the risk of failure. In a regulated industry, developers need to obtain approval from a suitable authority that the system is acceptably safe to operate. In aerospace, this is a "type certificate" issued by an authority such as the European Aviation Safety Agency (EASA) [6].

DO-178 is the primary software assurance guidance aimed at civil aircraft [9]. The guidance is accepted by many aviation regulators, including the Federal Aviation Administration (FAA) and EASA. The guidance defines five "*software levels*" based on the impact of failure of the software, from Level E: "*no effect on aircraft operational capability or pilot workload*" to Level A: "*catastrophic failure condition for the aircraft*". DO-178 defines a large number of objectives that must be achieved by each part of the software development process. For example, test cases should be developed and reviewed and should achieve a suitable level of coverage. The guidance includes a simple table showing which objectives are applicable to which software level. For example, accuracy of algorithms need not be checked at all for levels E and D software, does need to be checked for level C, and needs to be checked by an independent person for levels B and A software.

The latest issue of DO-178 was released in 2012 (DO-178C) [9], which includes specific supplements on model-based development, object-oriented technology, formal methods and tool qualification. As well as explicitly stating that it is acceptable to use formal methods, DO-178C contains practical guidance and objectives to be satisfied when using formal methods. The formal methods supplement makes a number of points:

- A formal method consists of a formal model plus a formal analysis.
- The formal analysis must be sound – that is, must not assert a property to be true when it may not be true.
- Tools used must be qualified (i.e. developed and shown to work correctly with an appropriate level of rigour).
- Differences between the model and the artefact it represents must be analysed.
- Any suitable formalisation may be used, and it is acceptable for an analysis to fail to produce a result for part of the software (as long as any results it does produce are correct) as other methods may be used to fill in the gaps.

This latter point highlights a potential issue with the formal methods supplement, in that it suggests that everything is possible, without indicating what is realistic or suggesting how to achieve it. For example, it suggests that the accuracy and consistency of the source code, including worst-case execution time, may be checked using formal methods – but only if formal semantics of the source code exist, which may "*need to take into account the programming language standards, compiler information (for example, default behaviour and configuration options), and characteristics of the target computer*". As explained by Kirner et al [16], it is not practically possible to model the characteristics of a target computer pertinent to timing, so the use of formal methods for some tasks may be hypothetical.

3. CASE STUDY DATA

Our case study covers a model-checking environment consisting of a large industrial model and standard model checking tools, together with requirements and known fault records relevant to the model. The model is taken from the gas turbine aero engine domain, specifically health monitoring systems. Such systems provide trouble-shooting, early-warning and maintenance data. The model, MERVE, is an implementation of system-level requirements in MATLAB and Simulink. It consists of a large Simulink model of a generic engine monitoring system (approximately 7000 blocks plus thousands of lines of helper functions written in MATLAB code). It includes the core functionality of typical commercial products. It is detailed enough to generate functional code by automated means, and perform real-world engine monitoring functions using either simulated inputs or input data recorded on a real engine. The functions represented by MERVE would generally be classed as Level C in

DO-178C [9]. The MERVE model is derived from a merged set of requirements for several real engine health monitoring systems, but does not precisely match any of them. However, it is sufficiently similar that errors found during the development and verification of real products can be mapped onto the MERVE model. Errors in the real products take various forms, such as:

- Code that does not match software requirements;
- Code and software requirements that do not match system requirements;
- System requirements that are incorrect;
- Requirements (at any level) that are contradictory;
- Code that provides additional functions not specified in the requirements that affect system behaviour.

A database of Problem Reports (PRs) relating to the real products was used to find samples of errors that might be amenable to model checking. The MERVE model was treated as the *software* implementation of a set of *system-level* requirements. Simulink was chosen because it is a widely used industrial modelling tool, particularly in aerospace. This is a significant advantage when trying to introduce model checking to a wider audience, since Simulink may be already installed or easily deployable on many engineers' computers. Simulink includes the model checker SDV, which is based on a proprietary tool called Prover Plug-In.

4. CASE STUDY PROCESS

4.1 Investigated Errors

The purpose of this step was to determine the categories of problems that typically exist in an industrial setting. The aim is to consider a large number of problems, rather than focusing on a few examples that may have been selected to demonstrate a limited set of properties of interest. A total of 1477 PRs were examined, being the complete set of PRs relating to a commercial aerospace system. A PR does not necessarily describe a fault in the system. For example, PRs are often used to add new features or to correct ambiguous or duplicate requirements. A quick search to exclude the following types of PRs was first performed:

- PRs that have been cancelled;
- PRs relating to hardware;
- PRs that introduce new features;
- PRs that make general improvements;
- PRs for document-only updates.

A summary of the results of this initial search is depicted Figure 3. The *'possible'* category was then examined further.

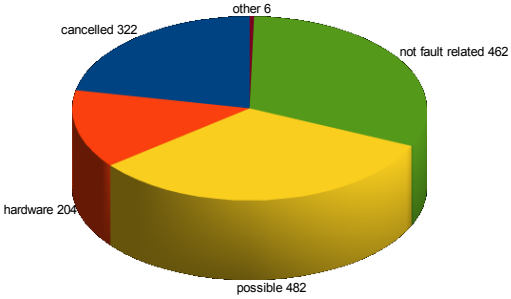

Figure 3: Problem Report Categories

Not all PRs in the *'possible'* category were examined in detail. Instead, the PR identifiers were ordered numerically and every tenth PR was selected. While not a truly random selection, it spans the development cycle and does not cherry-pick any particular kind of PR. In total, 49 PRs were examined in detail to determine how they might be detected by model checking and were categorised as follows:

- **Property violation:** a specific requirement was violated by the underlying requirements or design;
- **Emergent:** no requirement was violated, but the actual behaviour did not meet the designer's intent;
- **Consistency:** typographical and non-functional errors.

Figure 4 depicts the number of PRs in each category. This shows that the majority of PRs are property violations, which might be found directly by model checking. There are a significant number of emergent problems, which would not have been detected by checking current requirements, but could feed into the development of additional requirements.

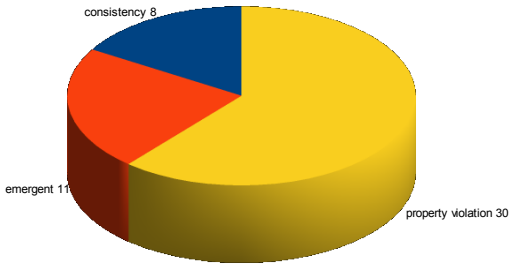

Figure 4: Detailed Problem Report Categories

Figure 5 shows the location of PRs within the requirements and design hierarchy. Only property violation and consistency PRs are included.

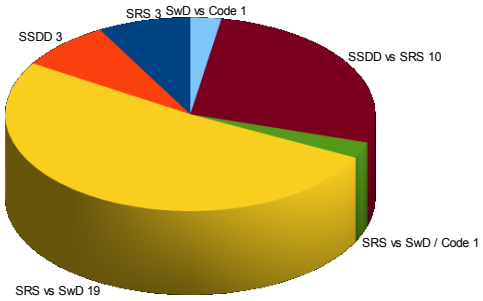

Figure 5: PRs within Development Hierarchies[1]

4.2 Model Checking

Each error was analysed to determine how it could be detected using a model checker with the MERVE model. Model checkers, including SDV, work by proving that properties always hold true, or finding a case where properties are violated. We attempted to convert the errors into properties within the MERVE model. This was a practical exercise that involved understanding the structure of the existing model, understanding the existing requirements, modelling properties in Simulink and operating SDV.

[1] SSDD = System level requirements, SRS = Software requirements and SwD = Software Design

MERVE is a large model containing over 7000 blocks and hundreds of inputs and outputs. We did not attempt to model-check the entire model for the following reasons:

- The state space is likely to be far too large for current tools to analyse.
- The model was written making full use of Simulink capabilities, not all of which are supported by SDV.
- The model was written for MATLAB version 2007, for which SDV was not available for this study. Instead, models had to be run in MATLAB version 2010, and the model as a whole is not compatible.
- Working with the model is quicker if parts of the system not related to the property being tested are removed.

The general approach was to isolate the part of the model that is sufficient to exhibit the problem and copy it into a new model on which the analysis is performed. The approach might miss possible interactions with other parts of the system and increase the risk of excluding relevant parts of the model. In reality, model checking is unlikely to be used as the sole means of verifying the whole model; rather it is targeted at specific functions and augmented with testing. Other test platforms could be used to verify system-wide interactions. Manual review could be used to confirm that the cut-down model provided sufficient coverage.

Figure 6 illustrates the use of SDV. The '*HighWins*' block exhibits a problem, in that the output flickers undesirably between two values when the inputs are similar but perturbed by electrical noise. The output was connected to a '*verification subsystem*', which uses Simulink modelling blocks to detect flickering. The '*HighWins*' block was driven using simulated noisy input signals. When run, SDV took 4 seconds to find a case where the 'verification subsystem' detects flicker (i.e. problem detected) after 4 steps. The case is presented as a tabular and graphical sequence of the inputs at each step. SDV also produces a playback block that may be used to drive the model via standard Simulink tools (e.g. single-step debugging, scopes and display blocks) to investigate the error. SDV does not highlight the location within the model that caused the error, but rather the input sequence over time that caused the error. Investigation is needed to understand the cause. The problem was corrected (by implementing a suppression system that filters transient output changes). When re-run, SDV asserts that the problem never occurs.

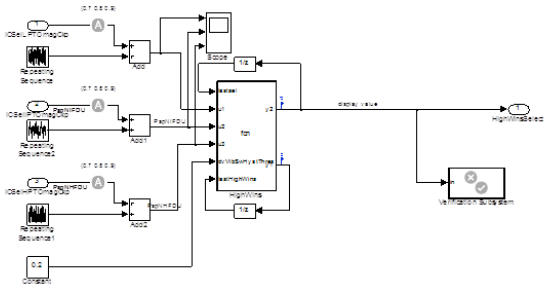

Figure 6: Illustrative Example

In a controlled development environment, the correction would be subject to design review and change control. It is important to note that SDV does not use written temporal logic. Instead, behaviour over time is defined using standard Simulink blocks, such as delays, counters, Boolean logic and state machines. In this case, a counter was used to check if the output had been steady for some length of time. It was possible to translate properties from natural language into Simulink with ease; this is due to familiarity with Simulink and its expressiveness. Knowledge of temporal logic was not required. However, such properties would need to be verified (e.g. by test) before use in an industrial application.

After completing this step, 49 problem reports were examined in detail to see if they were amenable to model checking. 42 were amenable, given an ideal model and model checking tool. However, only 9 were manifest in the model used in the case study. Of these 9, useful results were obtained (proofs, contradictions or increased confidence) from SDV for 8. This is low because MERVE does not cover the entire system. While this suggests that model checking with an existing model might not reveal many errors, we would expect that if model checking were adopted, more of the system would be modelled.

4.3 Top-Down Verification of Requirements

The preceding analysis of PRs looked at known problems. This is because we wanted to determine if a wide range of real-world problems can be identified by model checking. When incorporated into a full development process, we would expect model checking to be used top-down: starting from a set of requirements. This section records our investigation of this more representative use of model checking, in which we selected a group of system requirements collectively implementing a function known generally to be error-prone (hence success would support the case for using model checking in typical industrial settings) and attempted to prove that they are satisfied by MERVE, without the guidance of existing PRs. This directly mimics the manual review that is traditionally used to show that low-level requirements satisfy high-level requirements. The chosen area deals with selecting between multiple similar inputs depending on the validity of those inputs, e.g. coping with failed sensors. Figure 7 shows the scale of the model (details deliberately obscured for reasons of commercial sensitivity). The highlighted text states the requirement number satisfied by the corresponding part of the model; in total, 13 requirements are covered. The model consists of 150 blocks. A typical requirement is as follows: *A channel status shall be indicated as invalid if BIT [built-in test] indicates a channel fault or if, after a confirmation time is exceeded, either an in-range or out-of-range fault exists*. Nine requirements were modelled exactly as written. Due to the maturity of the requirements, no difficulty was encountered in translation into Simulink.

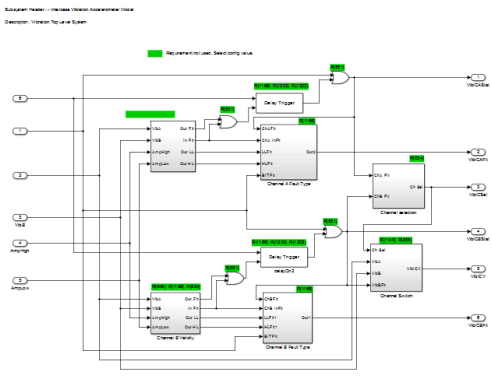

Figure 7: Requirements Check Context

For two requirements, the existing model used a fixed threshold, whereas the requirement calls for a variable threshold. This is a consistency error. We have overcome this by using the same fixed threshold in the property to be proved. We did not model the remaining two requirements as they do not readily map onto the

existing model. Of the 11 requirements modelled, SDV was able to prove the satisfaction of 8. No conclusion was reached about one requirement after running SDV for 10 hours. SDV was able to falsify 2 requirements:

(1): the requirement states a specific signal source to output at start up, without regard to the validity of that source. However, the model does not output that source if it is invalid. It is likely that the model's behaviour is preferred; nonetheless, the model does not satisfy the requirement.

(2): the requirement includes a timer indicating when a Boolean input has been true for a defined length of time. When the defined time period is zero, the model reports the timer as expired, regardless of the state of the input. This is an error in the model.

4.3.1 Comparison with Traditional Review

In the above example, model checking was used to show that low-level requirements satisfied certain high-level requirements. Traditionally, this would be done by a manual review. Our estimate is that the 11 requirements and corresponding parts of MERVE considered above could be reviewed for compliance and correctness in a one-hour inspection session involving three people (the author, a reviewer and a chairperson) plus one-hour preparation by the reviewer, totalling 4 hours. The estimates are based on measurements made in the study, engineering judgment and expert testimony.

In comparison, developing the properties and running SDV took about 3 hours. In a formal context (e.g. DO-178C), the modelled properties would also have to be reviewed, again taking an estimated 4 hours. Therefore using model checking would take about twice as long as conventional reviewing. However, there are other factors that favour model checking: (1) experienced users of SDV would be able to generate properties quicker; (2) once the properties have been generated, they can be re-run in a matter of seconds, with automatic generation of results and reports; and (3) if model checking were incorporated into a model-based development process, the properties could be developed as part of the high-level requirements rather than as a separate activity. However, there are also factors that count against model checking: (1) there is a cost in qualifying the model checker and (2) the modelled properties need to be manually reviewed.

4.3.2 Comparison with Traditional Testing

Since model checking is a technique for finding errors in a design, it is interesting to compare it with traditional testing. The following comparison treats MERVE as an implementation of the system requirements.

In a discussion with a testing team leader, it was estimated that the design and execution of conventional system-level tests for the 11 requirements would take one hour per requirement performed informally, and 20 hours per requirement with the level of formality required by DO-178C software level A. This overhead is due to activities such as independent reviews of the test cases and results, configuration control of test cases, models and results, recording of errors and quality assurance activities. In comparison, the average time to generate and run the test for a single requirement informally with SDV was approximately 17 minutes. Our model checking did not include any level of DO-178C formality. It is reasonable to assume that a similar overhead as with traditional testing would apply if done formally, since SDV does not automate reviews of the properties themselves, configuration management and quality control. In both cases, something has to be qualified as a verification tool under DO-178C: for traditional testing, the test rig, for model checking, SDV. Therefore the overall time saving of using model checking as a form of testing seems small.

The test engineer confirmed that this system-level testing would not have found the two errors above. For (1), his engineering judgment suggests that the error case is not significant and therefore would not have been covered by a system-level test case. For (2), system-level testing covers only typical values of timers, not boundary values such as zero. It is important to note, however, that full boundary value testing and condition coverage are typically done under DO-178C at lower levels of testing (i.e. these errors would be found later). As such, model checking could give a saving in rework by finding the errors earlier in the development process.

4.4 Certification using Model Checking

4.4.1 Relationship to DO-178C

The preceding sections discussed how model checking can be used to find a variety of errors between high-level and low-level requirements using real industrial examples. Since this case study focuses on airborne software, the next step is to consider how model checking relates to certification. The new aerospace guidance, DO-178C, includes a supplement for formal methods. This supplement defines a formal method as a combination of a formal model and a formal analysis. It gives several definitions of a formal model, including: "*Graphical models where the components of the diagram and the connections between them have mathematically defined syntax and semantics. For example, including but not limited to control diagrams or state machines*" [9]. The MERVE model meets these criteria. The supplement explains that it may not be possible to model the entire system (as found in our case study), but formal methods should be used in conjunction with other forms of verification (e.g. testing).

The supplement explains that formal models should be verified to provide assurance that if a property holds true for the model, it will hold true for the real system. If a model such as MERVE is auto-coded, the scope for divergence between the model and the real system is reduced, which should simplify the assurance process. Manual review could also be used to demonstrate that the software design matches the model. Further, according to the supplement, the following parts of DO-178C are not affected by the use of formal methods and are not discussed further in this paper: System Aspects Relating to Software Development (FM.2.0), Software Life cycle (FM.3.0), Software Configuration Management Process (FM.7.0), Software Quality Assurance Process (FM.8.0), Certification Liaison Process (FM.9.0) and Overview of Aircraft and Engine Certification (FM.10.0).

The following parts are affected and would be relatively simple to satisfy in a real development:

Software Planning (FM.4.0) and Software Development Processes (FM.5.0): It is necessary to define how formal methods will be used, what evidence is to be presented and any gaps with respect to conventional testing. MERVE can be considered as a 'design model' that could be used to generate code automatically and attempt to satisfy verification objectives using SDV. Since not all of the software is modelled, it would be necessary to define the boundary and describe the process that would be followed. This stage also records which certification objectives will be satisfied by formal methods (for specific parts of the system).

Software Life Cycle Data (FM.11.0): It is necessary to record definitions of model syntax, semantics and constraints. This would include the subset of Simulink used, the configuration of the tools and assumptions used in verification (for example, where

input values were constrained to speed up model checking). The definition of each verification case, including step-by-step procedure, results, and handling of any cases not proven by the tool, would also be recorded. It is also necessary to record details of the methods employed by SDV – this may prove difficult since it is a proprietary tool, but may be possible by involving the tool vendor in the certification process.

Additional considerations (FM.12.0): This section considers coverage analysis (that is, demonstrating that all of the software has been fully examined). Although SDV has features to support this, it is outside the scope of this case study.

The major part of the supplement considers the Software Verification Process (FM.6.0). There are three specific objectives for formal methods (FM.6.2.1):

"*FM.6.2.1 (a) All notations used for formal analysis should be verified to have precise, unambiguous, mathematically defined syntax and semantics; that is, they are formal notations.*"

Simulink is a mathematics-based notation, which is executable and deterministic. Scaife et al [22] state that the semantics are only informally defined in a reference manual. However, this does not make it impossible to define the semantics formally. For example, one could specify a subset of Simulink whose semantics are well-defined. Enforcement of this subset could be automated, and indeed SDV checks each model prior to execution to ensure that no incompatible features are used.

"*FM.6.2.1 (b) The soundness of each formal analysis method should be justified. A sound method never asserts that a property is true when it may not be true.*"

SDV is a proprietary tool and therefore its internal methodology is not known, making it difficult to establish whether it is sound. Given the tool's intent (to prove properties), it is plausible that the underlying methodology was chosen to be sound, but it is not possible to confirm this without further information. An approach to this justification would be to qualify SDV as a verification tool: the qualification activity demonstrates that the tool correctly implements the underlying methodology without introducing unsoundness. The supplement also explains that it is acceptable for the analysis to give an inconclusive verdict, as this simply means that an alternative method of verification must be used.

"*FM.6.2.1 (c) All assumptions related to each formal analysis should be described and justified; for example, assumptions associated with the target computer or about the data range limits.*"

Further work is needed to justify assumptions, for example:

- Justifying that model checking results performed on a desktop computer would still be valid on the target computer, which may use a different floating-point system or smaller sizes for storing values. This seems feasible for SDV.
- Justifying approximations made by SDV (such as rational approximation of floating-point numbers).
- Justifying any constraint blocks used on inputs to speed up model checking, for example by reference to input ranges given in the system requirements.

The supplement then describes how formal methods can achieve various verification objectives. Section FM.6.3.2 "*Reviews and Analyses of Low-Level Requirements*" is particularly pertinent to this case study; the objectives of this section are discussed below:

FM.6.2.3 (a) Compliance: SDV could be used to demonstrate that the design complies with the high-level requirements, since both are formally modelled (except for derived requirements, which do not have parent requirements).

FM.6.3.2 (b) Accuracy and Consistency: Modelled requirements are implicitly precise and unambiguous, since Simulink (if a suitably defined subset is used) has a precise and unambiguous meaning. Consistency (the absence of conflicts) can be interpreted in several ways. For example, it is possible for multiple blocks to overwrite each other's data in global storage, and it is unlikely that the whole model could be checked in one go due to its size, therefore there may be undetected conflicting requirements. Manual review could be used to check this. However, if two requirements require contradictory proofs, it will not be possible to satisfy both, which will be visible as a verification failure. The supplement does not define what "accuracy" means for a requirement. SDV can provide evidence that the design model accurately matches the high-level requirements, and that outputs produced by the model accurately match expected values.

FM.6.3.2 (c) Compatibility with the target computer: This cannot be checked as there is no model of the target hardware.

FM.6.3.2 (d) Verifiability: As noted in the supplement, requirements are implicitly verifiable once they are expressed formally. The supplement suggests that requirements including the concepts of "always" or "never" can be problematic. However, this seems to be at odds with the concept of formal methods, which is that proofs can be found that properties always hold true or are never violated. Further work may be needed to determine any features of Simulink models that are not verifiable.

FM.6.3.2 (e) Conformance to standards: This could be checked automatically or by review, and SDV includes some checks.

FM.6.3.2 (f) Traceability: Automated tools may help with traceability, but this is not a feature of SDV and has not been investigated in this case study.

FM.6.3.2 (g) Algorithm Aspects: SDV could be used to check the accuracy and behaviour of algorithms, assuming the desired properties are defined in the high-level requirements and the algorithm is small enough to check.

4.4.2 Assurance Argument for Model Checking

Figure 8 through Figure 11 depict a high-level assurance argument for using SDV as a model checker in the context of DO-178C. The argument is represented using the Goal Structuring Notation (GSN) [13], which breaks a top-level claim into a hierarchy of sub-claims, ultimately supported by evidence. The argument is simplified by not showing some objectives (e.g. configuration control and tool qualification), which we have assumed can be readily satisfied. These additional objectives would be shown as further sub-goals under G1.2. Most of the bottom-level claims are still to be developed. We expect some of these to be straightforward to complete (G1.1.1.2, G1.2.1, G1.2.3). However, some are novel, and the argument as a whole relies on being able to find the evidence. These are:

G1.1.1.1: Simulink has a precise, unambiguous and mathematically defined semantics, in that it can be executed by a computer program in accordance with a set of rules that could be written down using mathematics and logic. This contrasts with natural language, which is more difficult to execute. However, the description of the semantics, as supplied with the tool, is written partly in natural language, rather than a mathematical notation such as linear temporal logic or set theory. Further work would be necessary to define it more precisely.

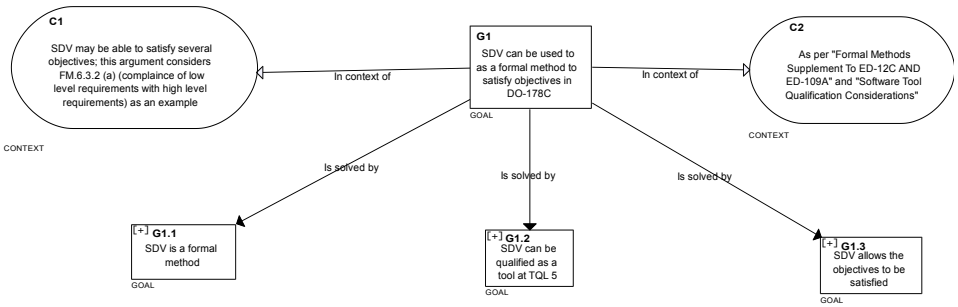

Figure 8: Top Level Assurance Argument

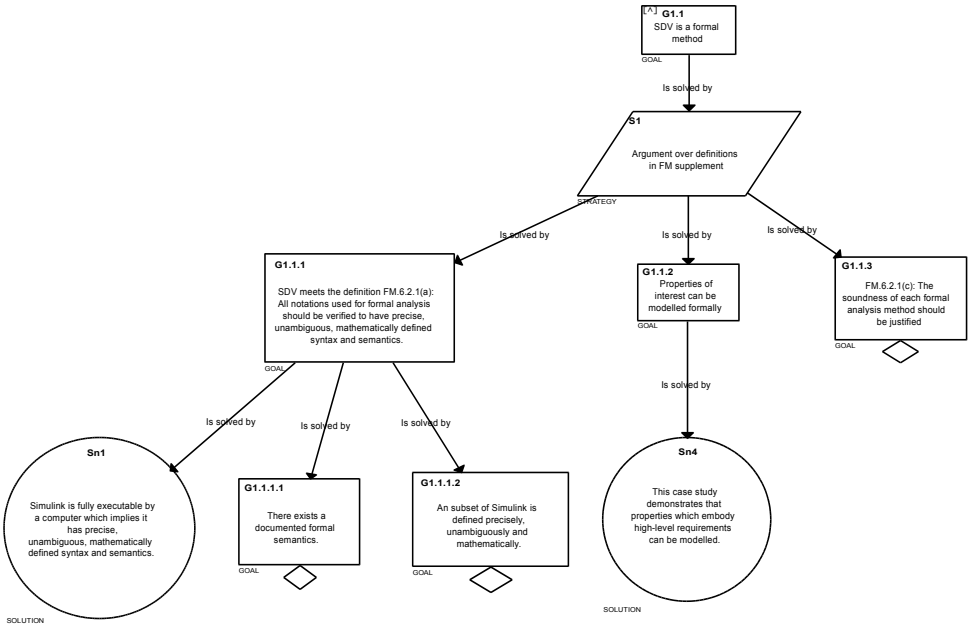

Figure 9: G1.1. Sub-Argument

G1.1.3: It is difficult to determine if the methods used by SDV are sound without further information from the tool vendor. Without knowing this, it is not possible to use SDV as a formal method within DO-178C. Based on an example within DO-178C, one way of showing that a method is sound is to cite published academic proofs. It may be possible to take this approach with SDV, once the internal methods are known.

G1.2.2: Having defined the operational requirements for the tool, it is necessary to show that they have been met. This may prove difficult, as it may not be possible to substantiate a proof for all but the most trivial model.

The above argument was presented to the European Chair of the DO-178C Formal Methods sub-group. While this informal check does not imply suitability of SDV for any particular purpose, it raises an interesting point: "*I think the whole argument is about replacing verification objectives on LLR [low-level requirements] and LLR to HLR [high-level requirements] so it is only seeking to replace review and analysis objectives, not test ones. This makes the whole thing much easier… The replacement of test by FM [formal methods] is a much more contentious issue*" [3].

This discussion also highlighted that soundness of a formal method can be considered separately from qualification of the associated tool. If the method is known to be sound, then qualification might become the simpler task of showing that the tool correctly implements the method, rather than proving that the implementation is sound.

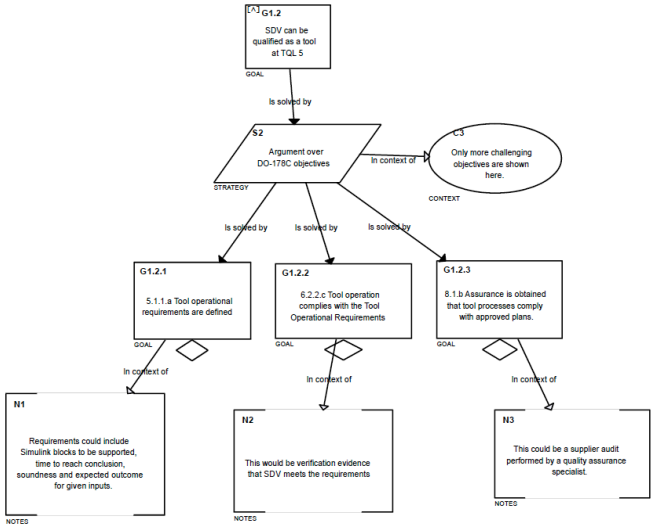

Figure 10: G.1.2 Sub-Argument

By creating this assurance argument, we have provided a structure in which the strengths or weaknesses of the case may be discussed. The reader is encouraged to challenge and critique any part of the argument or evidence.

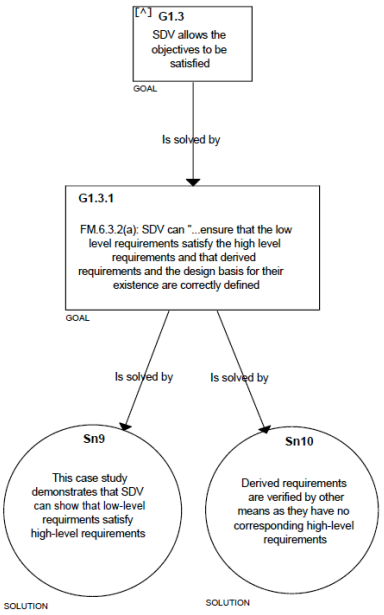

Figure 11: G.1.3 Sub-Argument

5. DISCUSSION AND OBSERVATIONS

The following discussion reflects on the experience gained in this study regarding various aspects of usefulness.

5.1 Errors Found

The premise of model checking is that it provides proof of a property, thus giving a high degree of confidence in the correctness of a model. Traditional reviews and testing are not perfect (this is evident from the fact that commercial software does contain errors), so model checking promises an improvement in the number of errors detected. In this case study, we primarily used known errors to demonstrate that detection of real-world errors is within the scope of model checking. This does not give direct evidence that model checking finds errors that would have been missed by conventional verification. However, we found that many existing errors can be defined as property violations, therefore could in principle be found by model checking, and, importantly, found earlier in the development cycle. This reduces development effort, due to a reduction in the number of artefacts that need reworking when errors are found. We have also attempted to prove that the model correctly implements a set of high-level requirements. This was largely successful in that SDV both proved some requirements to be satisfied, and others to be incorrectly implemented (although in one case, SDV failed to reach a conclusion). Although we do not have factual information on whether these errors were found by traditional methods, we think this is likely, since experienced reviewers (and testers) should be looking for boundary conditions and failure cases. In a minority of cases, SDV was unable to reach a conclusion even after 10 hours of operation. In these cases, it is still possible to gain confidence in the model's correctness by running SDV for a limited number of time steps. In summary, we have found evidence that model checking can find errors earlier in the development cycle than conventional verification. We have not found evidence that model checking finds more errors than conventional verification.

5.2 Certification

We have shown how model checking could be used to satisfy several objectives of DO-178C, particularly in relation to verification of low-level requirements. Further, we have found that model checking compares favourably with conventional system-level testing in terms of speed and thoroughness, although we have not considered this in relation to achieving DO-178C objectives for system testing. This suggests that SDV can be used in the development of high-integrity airborne software. However, such use would require the tool to be qualified and shown to be sound. Since SDV is a proprietary tool, and certification activities are generally proprietary, it is not possible to state whether SDV has, or could be, qualified in practice.

5.3 Effort

We have estimated the effort that would be needed by conventional verification and compared this with the effort expended using SDV. The time taken to convert high-level requirements into properties for model checking was similar to traditional manual review of high-level requirements, and if used in a formal development process (such as DO-178C) would require an additional review of the properties. However, if the properties were developed as part of the existing high-level requirements generation and review activity, modelling the properties might not take noticeably longer. The time spent actually operating SDV was a matter of seconds per requirement. This is significantly faster than manual verification, but the real-world benefit would depend on how often verification is repeated (for example, if a particular product was developed with overnight regression testing, fast proofs may be very useful; whereas if testing was performed only once on the final product, this would be less advantageous). Further work would be needed to investigate this, or whether model checking facilitates additional verification activities, such as automated regression testing, which would not otherwise be considered. A benefit of SDV is that it automatically generates a written report of results. Manual verification often requires records to be generated manually.

5.4 Simplicity

We found converting high-level requirements into formally modelled properties generally straightforward because SDV integrates directly into the modelling environment and most requirements map onto simple '*implies*' relationships. However, we were unable to model some of properties (e.g. involving a counter, a state machine and a division) and in a minority of cases SDV was unable to find a solution. It may be that further effort could overcome these problems, or it may be that some problems are not amenable to checking with SDV.

5.5 Modifications to the Model

In some cases, we had to remove blocks from the original model in order to make it compatible with SDV. These are generally non-functional, for example removing signal attribute checks left in for debugging purposes. In general, we used SDV on small extracts of the overall model, in isolation, containing a few tens of blocks. Further work is needed to determine if SDV can be run on the whole model (containing 7000 blocks). Since SDV is not compatible with all blocks supported by Simulink, it will generally be necessary to modify the model to support SDV. This

creates a problem because the properties proven on the modified model may not hold true on the original model from which code would be generated. This may be avoidable by using only blocks supported by SDV when developing the model. Alternatively, traditional reviewing could be used for unsupported blocks.

6. VALIDITY AND LIMITATIONS

This study looked at a single model. Although it is a typical model in that it already exists in industry for purposes other than model checking, other models may have different properties. For example, MERVE uses the Simulink option of a 'fixed step discrete solver', which happens to be compatible with SDV. A model that used a continuous solver would not be compatible with SDV. Other models may contain features that are not decidable by SDV, making it impossible to perform model checking. Further work would be needed to assess a wider variety of models.

By way of comparison, the PathStar project [15] used model checking to find 75 errors in 18 months of use. 5 of these errors were also found by system-level testing. The report does not discuss if any traditional model reviews or software testing were performed, so it is not apparent whether the remaining 70 errors would have been found by traditional methods (if such methods were used). The primary motivation for using model checking appears to be the complexity of the PathStar software, which is an intricate distributed system. Our findings do not contradict any of the conclusions of the PathStar project. Another comparison is the SLAM system, produced by Microsoft. SLAM is a good example of a model checker being used in an industrial context by people with no specialist knowledge of model checking. Unlike SDV, it is targeted to a specific problem (i.e. checking device drivers). Microsoft has highlighted the obstacles that many organisations would face in converting a research tool into an industrial tool [2]. These include recruiting model-checking experts and developing their own formal tool-chain, which may be beyond the capabilities of smaller organisations.

Further, some authors have highlighted potential difficulties with model checking that were not apparent in this case study. For example, Fraser et al [11] mention performance limitations due to the state explosion and applicability of each theoretical model checking technique to particular problem types. We found that SDV completed most tests within minutes, so performance was not a problem. For a minority of tests, SDV failed to produce a result even after hours of operation. We would argue that although it might be impractical to use SDV for those tests, this is not a problem if model checking is combined with traditional verification. In any practical application of model checking, it is unlikely that all parts of the system will be modelled, so it would be necessary to maintain a traditional verification capability. Fraser et al also comment on the difficulty in obtaining a model. However, large numbers of industries and projects (based on our experience) have already invested in developing Simulink models. This is perhaps an important advantage of using Simulink and SDV compared with other model checkers (and indeed Fraser et al suggest that an over-arching model-based development process might be a good source of models).

Fantechi et al [10] note that model checking is not mentioned in safety-critical software standards, and that there are no known qualified model checkers. As described in this case study, DO-178C rectifies this, and qualification of a model checker seems possible in principle. Further work is needed to validate and populate the argument we have provided for using SDV as a formal method for model checking.

Our approach of using SDV to prove properties directly on Simulink models seems at odds with the approach of Whalen et al [24], who have created a whole verification suite that involves nine model checking tools. However, we believe this is because they were researching the general problem and trying to determine the relative advantages of different tools and languages. As with our case study, their method used Stateflow and Simulink as an input, and Prover/Design Verifier as one of the model checkers. In contrast to our case study, they convert the input model into a variety of intermediate formats. Further work would be needed to determine if, and in what contexts, it is beneficial to use multiple model checkers. Cofer et al suggest several objectives in the DO-178C formal methods supplement that could be satisfied by this process [5]. However, these suggestions had not been evaluated in a real certification environment.

Some tool vendors provide tool qualification kits. For example, The MathWorks provides a qualification kit for some parts of Simulink for use with DO-178 [8]. However, this covers only a small number of certification objectives (such as measuring test coverage and checking design standards). The kit does not cover qualifying SDV as a verification tool.

7. CONCLUSIONS

Model checking has the potential to find more errors than traditional verification and find them earlier in the product development cycle. This is attractive to the developers of safety-related software. However, such software is usually developed to strict standards, such as DO-178C for airborne software, which require the adequacy of any new technique to be justified. In this paper, we have examined the possibility of using an existing and a widely-used industrial modelling tool (Simulink) with its built-in model checker (SDV) to study an existing industrial model (not designed specifically for model checking).

We have investigated 49 real-world problems to determine if they could have been found by model checking. We have found that 51% could in principle be found by checking a full model of the software against higher-level requirements because they represent property violations. We also tried proving that part of the model matched the corresponding high-level requirements without first looking at existing errors. We checked 11 such requirements and were able to achieve useful results for 10. This exercise found 2 errors in the model. However, it is likely that they would have been found by thorough traditional manual review and testing. We found the model checker to be quick and effective at finding both proofs and counter-examples that high-level requirements hold true. Re-running the verification cases takes a matter of seconds, significantly faster than manual review.

However, this case study also revealed several factors that impaired the usefulness of the technique. The time taken to develop and review models of the high-level requirements, and make adjustments to the model under test to be compatible with the model checker, is comparable to the time taken to perform manual tests and reviews. SDV was not able to reach a conclusion in all cases, either because of errors in the modelling of high-level requirements, or because of limitations of the tool. This means that conventional verification would still be needed.

We have considered how SDV fits in with the new airborne software guidance DO-178C, which includes extensive guidance on the use of formal methods and tool qualification. We have discussed how SDV can be treated as a suitable formal method for automating review and analysis, subject to a better understanding

of the internal mechanisms used by the tool, in particular that the underlying methods are sound.

This study shows that there is potential for the useful application of model checking in industry, but there are a number of open issues still to address. In many cases, these are presented in context throughout this paper; the following provides a summary.

(1) The industrial use of other tools could be investigated further – it may be that tools other than SDV are in widespread use in industry, and may have been qualified, but this is not widely reported due to commercial sensitivities. Similarly, companies may have attempted to qualify tools and failed; this is useful information but unlikely to be published.

(2) Promoting the ideas presented in this study is a challenge; further work could investigate possible approaches and obstacles.

(3) In order to make general conclusions about the number of problems that might be detectable by model checking, further case studies would be needed covering a variety of domains.

(3) Our comparisons of the amount of time taken by traditional verification versus model checking are based on engineering judgement. Further work is needed to validate these comparisons.

(4) In some cases, we were unable to prove properties conclusively with SDV. Further work would be needed to determine if this was due to limitations of the tool itself, or our use of the tool.

(5) Although we have presented an assurance argument for how SDV could meet the needs of DO-178C, significant further work is needed in this area to investigate soundness and qualification of the tool and which objectives might be satisfied by the tool. Certification is only valid for a specific context and involves agreement on the approach between the applicant and the certifying authority; further work is needed to determine if our assurance argument is acceptable.

8. REFERENCES

[1] Anderson, R. 1996, Model checking large software specifications. 4th *ACM SIGSOFT Symp. on Foundations of Software Engineering*, San Francisco, US, 156-166.

[2] Ball, T. et al. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft [Online]. Available: http://research.microsoft.com/pubs/70038/tr-2004-08.pdf

[3] Brown, D. private communication, Dec. 2012.

[4] Cimatti, A. 1999. NuSMV: a new symbolic model verifier. *11th Conference on Computer-Aided Verification*, Trento, Italy, 495-499.

[5] Cofer, D. 2010. Model checking: cleared for take off", *17th Int. SPIN conf. on Model Checking Software*. Enschede, The Netherlands, 76-87.

[6] Dodd, I., Habli, I. 2012. Safety certification of airborne software: An empirical study. *Reliability Engineering & System Safety*, vol.98, no.1, 7-23.

[7] Drusinsky, D., 2008. A framework for computer-aided validation, *Innovations in Systems and Software Engineering,* 4, 2, (2008), 161-168.

[8] Erkkinen, T., Potter, B. 2009 Model-based design for DO-178B with qualified tools. *AIAA Modelling and Simulation Technologies Conference*, Chicago, IL, USA, 1-13.

[9] EUROCAE/RTCA. 2012. *DO178C Software Considerations in Airborne Systems and Equipment Certification*.

[10] Fantechi, A., Gnesi, S. 2011. On the adoption of model checking in safety-related software industry. in *30th Int. Conf. on Computer Safety, Reliability and Security*, Naples, Italy, 2011, 383-396.

[11] Fraser, G., Testing with model checkers: a survey. *Software Testing, Verification & Reliability*. 19, 3, (Sept. 09) 215-261.

[12] Graydon, P., Habli, I., Hawkins, R., Kelly, T., Knight, J. 2012. Arguing conformance. *Software, IEEE*, vol.29, no.3, 50-57.

[13] GSN Standard: http://www.goalstructuringnotation.info

[14] Holzmann, G. J., 1997. The model checker SPIN. *IEEE Trans. Softw. Eng.* 23, 5 (May 1997), 279-295.

[15] Holzmann, G. Smith M., 2000. Automating software feature verification, Bell Labs, Murray Hill, NJ, USA*, Bell Labs Technical Journal*, vol. 5, 72-87.

[16] Kirner, R., Puschner, P. 2004. Measurement-based worst-case execution time analysis using automatic test-data generation. *3rd IEEE Workshop Software Tech. For Future Embedded and Ubiquitous Systs.*, Washington, US, 7-10.

[17] Knight J. 2002. Safety critical systems: challenges and directions. *24th Int. Conf. on Software Engineering*, Orlando, FL, USA, 547-550.

[18] Mathworks. 2013. Simulink Design Verifier [Online]. Available: http://www.mathworks.co.uk.

[19] McDermid, J. A. 2012. Safety Critical Software. *Encyclopedia of Aerospace Engineering*.

[20] Quan, T. 2008. MAFSE: a model-based framework for software verification, *IEEE Int. Conf. Secure Software Integration and Reliability Improvement*, Singapore, 150-156.

[21] Robby MB., 2006. Bogor: a flexible framework for creating software model checkers. *Testing: Academic & Industrial Conf. – Practice And Research Techniques*, Windsor, UK.

[22] Scaife et al, 2004. *Defining and translating a "safe" subset of Simulink/Stateflow into Lustre*, Verimag, Tech. Rep. TR-2004-16.

[23] van den Berg, L. 2007. Introducing time in an industrial application of model-checking, *Formal Methods for Industrial Critical Systems*, Berlin, Germany, 56-67.

[24] Whalen, M., Cofer, D., Miller, S., Krogh, B., Storm, W., 2007. Integration of formal analysis into a model-based software development process. *12th Int. Workshop on Formal Methods for Industrial Critical Systems*, Germany, 68-84.