Practical Limits On Software Dependability: A Case Study

Patrick J. Graydon, John C. Knight, and Xiang Yin

University of Virginia, Charlottesville VA 22903, USA
{graydon, knight, xyin}@virginia.edu

Abstract. The technology for building dependable computing systems has advanced dramatically. Nevertheless, there is still no *complete* solution to building software for critical systems in which *every* aspect of software dependability can be demonstrated with high confidence. In this paper, we present the results of a case study exploration of the practical limitations on software dependability. We analyze a software assurance argument for weaknesses and extrapolate a set of limitations including dependence upon correct requirements, dependence upon reliable human-to-human communication, dependence upon human compliance with protocols, dependence upon unqualified tools, the difficulty of verifying low-level code, and the limitations of testing. We discuss each limitation's impact on our specimen system and potential mitigations.

1 Introduction

The past several decades have seen dramatic advances in technology for building dependable computing systems. Proof of software functionality, for example, has advanced from a costly manual exercise to a tool-aided endeavor that is becoming practical for ever-larger systems [13, 22]. Using languages such as SPARK Ada and approaches such as Correctness-by-Construction, it is possible to prove that even large systems are demonstrably free of entire classes of defects including unhandled runtime exceptions, inadvertent memory overwrites, flow errors, and buffer overflows. These advances have not, however, provided practitioners with a *complete* solution to building software for ultra-critical systems in which *every* aspect of software dependability can be demonstrated with high confidence.

In this paper, we present the results of a case study conducted to explore the practical limitations on software dependability. In prior work, we implemented software for a specimen life-critical medical device and a developed a rigorous argument showing how evidence arising from our effort supports the conclusion that the software we developed is fit for use in the context of that device [6]. In this work, we analyze that argument and extrapolate from its weaknesses a set of practical limitations on the assurance of software dependability.

We describe the development effort and artifacts upon which our case study is based in section 2 and the case study process in section 3. In section 4, we discuss the limitations that we discovered, the effect of each upon the case study target, and any potential mitigations we are aware of. We discuss related work in section 5 and conclude in section 6.

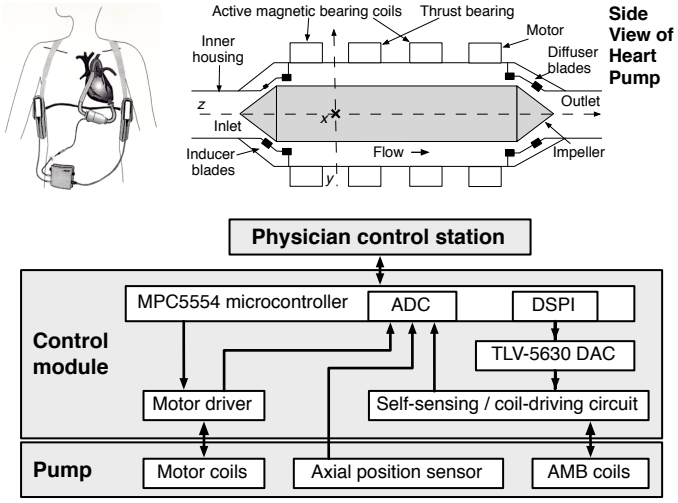

Fig. 1. LifeFlow structure and use

2 The UVA LifeFlow LVAD MBCS

In prior work [6], we introduced Assurance Based Development (ABD), an engineering approach to the synergistic creation of software and assurance of the software in the form of rigorous argument. In order to assess ABD and its unique process synthesis mechanism, we conducted a case study development of Magnetic Bearing Control Software (MBCS) for a safety-critical system, the University of Virginia's LifeFlow Left Ventricular Assist Device (LVAD). The details of this case study are reported elsewhere [6].

2.1 The UVA LifeFlow LVAD

LifeFlow is a prototype artificial heart pump designed for the long-term (10–20 year) treatment of heart failure. LifeFlow has a continuous-flow, axial design. The use of magnetic bearings and careful design of the pump cavity, impeller, and blades reduce the damage done to blood cells, thus reducing the potential for the formation of dangerous blood clots. Fig. 1 shows the placement of the pump, the batteries and the controller, a cross-section of the pump, and the overall structure of the controller. Control of the magnetic suspension bearings is provided, in part, by software running on a Freescale MPC5554 microcontroller. Table 1 summarizes the MBCS requirements.

2.2 The MBCS Development Process

As part of the case study evaluation of ABD, we developed an implementation of the MBCS and completed a formal verification of its functionality. Briefly, our software development process included:

Table 1. Magnetic bearing control software requirements

Functionality	1. Trigger and read Analog-to-Digital Converters (ADCs) to obtain impeller position vector $\boldsymbol{u}$. 2. Determine whether reconfiguration is necessary. If so, select appropriate gain matrices **A**, **B**, **D**, and **E**. 3. Compute target coil current vector $\boldsymbol{y}$ and next controller state vector $\boldsymbol{x}$: $\boldsymbol{y_k} = \mathbf{D} \times \boldsymbol{x_k} + \mathbf{E} \times \boldsymbol{u_k}$ $\boldsymbol{x_{k+1}} = \mathbf{A} \times \boldsymbol{x_k} + \mathbf{B} \times \boldsymbol{u_k}$ 4. Update DACs to output $\boldsymbol{y}$ to coil controller.
Timing	Execute control in hard-real-time with a frame rate of **5 kHz**.
Reliability	No more than 10^{-9} failures per hour of operation.

1. Development of a formal specification in PVS [19] and an informal argument showing that this specification refines the requirements.
2. Design of a cyclic executive structure to manage the real-time tasks.
3. Design of the bearing control task routines by functional decomposition.
4. Implementation of the MBCS in SPARK Ada.
5. Implementation of bootstrap code in PowerPC assembly language.
6. Use of AdaCore's GNAT Pro compiler [1] to target the bare microcontroller.
7. Formal verification that the implementation refines the functional specification using the PVS proof checker and the SPARK tools in accordance with our Echo verification approach [21, 22].
8. Machine analysis of Worst-Case Execution Time (WCET) and stack usage.
9. Requirements-based functional testing to Modified Condition / Decision Coverage (MC/DC) [9] was planned but not completed because of limited resources. For the purposes of this study, we proceeded as if the testing had been completed and the expected evidence gained. Had the testing proved impossible to conduct as planned, we would have revised our testing plans, possibly resulting in more limited testing evidence.

The resulting software consisted of 2,510 lines of SPARK Ada, of which 579 implement the control calculations and 114 implement the main program and cyclic executive structure, with much of the remaining code implementing interfaces to the MPC5554's functional and peripheral units. The software also includes implementations of the `memcpy` and `memset` library routines called from compiler-generated code and a bootstrap routine consisting of 106 PowerPC assembly language instructions.

2.3 The MBCS Fitness Argument

The MBCS fitness argument, recorded in the graphical Goal Structuring Notation (GSN) [11], explains how evidence from the MBCS development effort

supports the claim that the MBCS is fit for use in the context of the LifeFlow LVAD. Fig. 2 illustrates the general form of the argument, which contains 348 GSN elements. After operationally defining "fit for use in the context of the MBCS" to mean demonstrably satisfying the given software requirements, the argument decomposes requirements obligations into real-time and non-real-time requirements. Our argument that the real-time requirements of the MBCS have been met rests largely upon the use of a WCET analysis tool to ensure that scheduled tasks complete within their scheduled execution periods. We use two independent lines of support to show that the MBCS's non-real-time requirements have been satisfied. The first is an appeal to requirements-based functional testing with MC/DC, and the second is a refinement argument based on our Echo approach to practical formal verification [22].

3 Case Study Process

Our case study evaluation of the practical limitations on software dependability was based upon and driven by the MBCS fitness argument. Dependability is irreducibly a system property: software by itself cannot do any harm to people, equipment, or the environment and so cannot be "unsafe." In order for safety to be a consideration, software has to be operating as part of a complete system for which damage is possible. When a system includes software, that software must have certain properties if the system is to be adequately dependable. Limitations on software dependability preclude demonstrating that software possesses such properties, and such limitations thus present threats to a software fitness argument's conclusion.

We identified two forms of threat to the MBCS fitness argument: (1) assumptions in the argument, such as the assumption that the PVS proof checker will not accept an invalid proof; and (2) reasoning steps in which the premises do not actually *entail* the claim, such as the argument that requirements-based functional testing with MC/DC supports the claim that the system meets its functional requirements. For each threat, we: (a) identified the general limitation of which the identified threat was a specific instance; (b) evaluated the limitation's impact on to the MBCS fitness argument; and (c) enumerated any potential mitigations of which we were aware.

We do not claim that the list of limitations presented in section 4 is complete. Other development efforts might be subject to limitations that did not affect the MBCS effort, and others might find limitations in the MBCS effort that we overlooked. The range and impact of the limitations facing this effort, however, are one indication of the challenges of dependable software engineering generally.

4 Limitations Discovered

In this section, we present the limitations uncovered by our case study. These limitations, which overlap somewhat, are pervaded by three major themes:

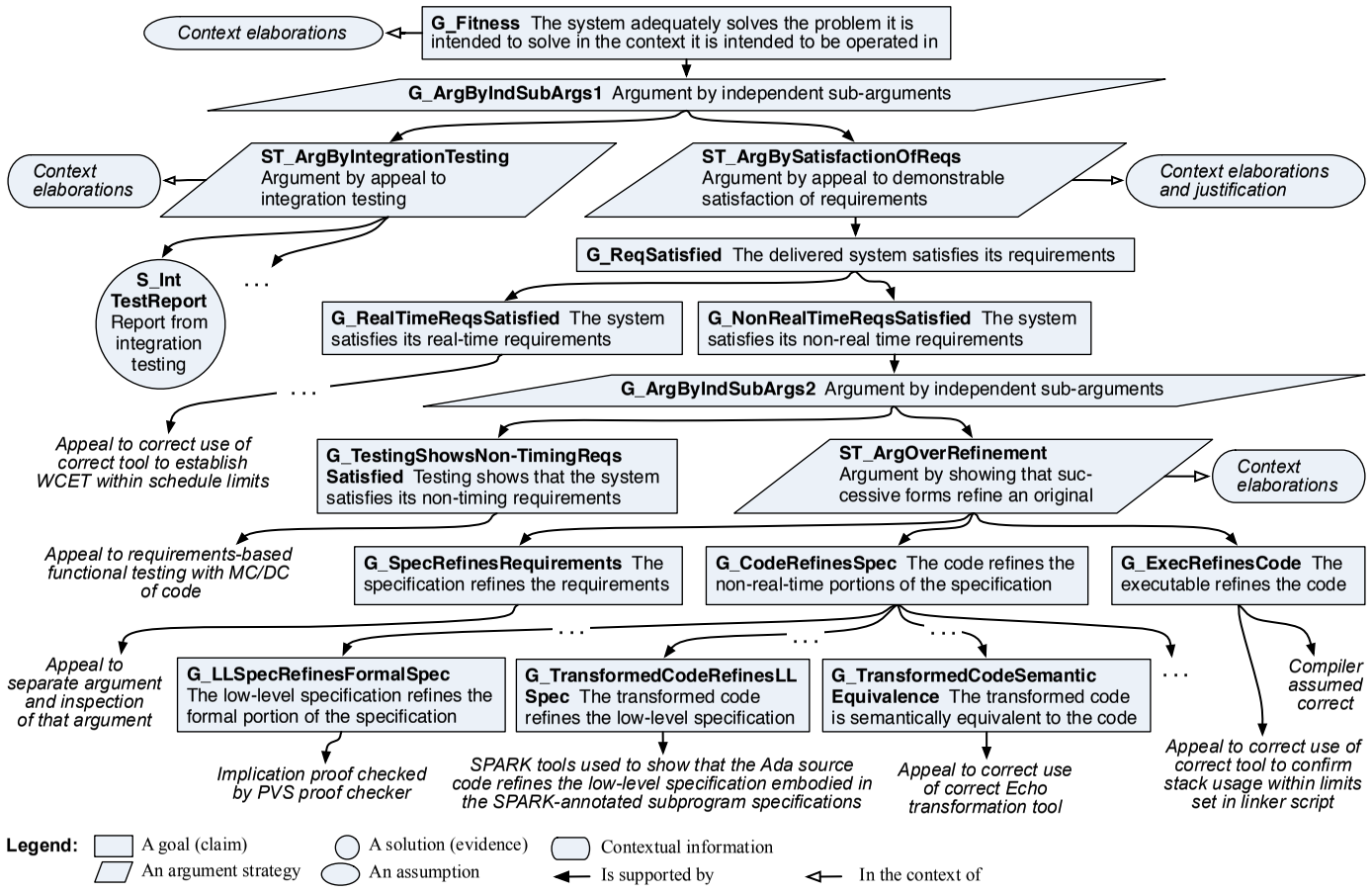

Fig. 2. Excerpt from the LifeFlow MCBS fitness argument

1. Dependence upon fallible human beings.
2. Incomplete or immature tools or technologies.
3. Techniques that cannot *practically* ensure the needed dependability.

4.1 Reliance Upon Correct Requirements

The correctness of the requirements is assumed in many software development efforts. Unfortunately, requirements gathering depends unavoidably on fallible human beings. Requirement defects do occur: the majority of software defects in critical embedded systems, in fact, stem from requirements defects [18].

MBCS manifestation. The correctness of the requirements is assumed in 336 of 348 elements (97%) of the MBCS fitness argument. With the exception of an appeal to integration testing, the MBCS software fitness argument contends that the MBCS is fit for use in the context of the LifeFlow LVAD system because the MBCS meets the requirements imposed upon it by that system.

Potential mitigations. Many techniques have been suggested and used to reduce the incidence of requirements defects. Requirements can, for example, be structured in such a way as to make omissions and contradictions more obvious, thereby increasing the likelihood that they will be caught and corrected [10]. Prototypes can be built to demonstrate an understanding of the requirements so that stakeholders have the opportunity to correct misconceptions. None of these techniques, however, will yield demonstrably adequate requirements.

Requirements for embedded software are derived from the design of the system in which the software is embedded. This replaces reliance upon a subject matter expert to enumerate requirements with a reliance upon the derivation process. Embedded software requirements can be no more trustworthy than the process used to derive them: defects in hazard analysis, fault tree analysis, HazOp studies, or FMECA could all contribute to erroneous software requirements.

4.2 Reliance Upon Reliable Human-To-Human Communication

The construction of software requires precise communication of complex concepts, frequently between people with different backgrounds and expertise. Systems engineers, for example, must communicate software requirements completely and correctly to software engineers. Such communication again relies upon fallible humans and is fraught with the potential for error. In the worst case, the recipient of communication will be left with an understanding other than that which the originator intended *and will be unaware of the error.*

MBCS manifestation. The MBCS fitness argument does not explicitly address the sufficiency of human-to-human communication. The argument does, however, reference 16 documents written in whole or in part in natural language, including the requirements, the specification, test plans, inspection protocols, various reports, and tool manuals. Of these, the strength of the main fitness

claim rests most heavily upon correct understanding of the requirements. The potential for a misunderstanding of the requirements is mitigated somewhat by the use of an independent test team: unless the test team and the drafters and reviewers of the specification made the same mistake, the miscommunication would likely result in either a failed test case or a falsely failing test case.

Potential mitigations. The use of formal languages can partially address this problem. However, while formal semantics define precisely one meaning for a given formal text, it is impossible to formalize *all* engineering communication. Since formal languages are semantically void, natural language is unavoidable even if its use is restricted to binding formal tokens to real-world meanings.

Research has addressed the problem of communication deficiencies in engineering. The CLEAR method, for example, addresses misunderstandings of the terms used in requirements by building definitions that are demonstrably free of certain classes of common defects [8, 20]. Unfortunately, we are aware of no method that can demonstrably reduce the incidence of miscommunication to an adequate level when the consequences of miscommunication are severe.

4.3 Reliance Upon Understanding Of The Semantics Of Formalisms

In every software development effort, human beings read and write artifacts in at least one formal language, such as a programming language or specification language. Even if these languages contained no deliberate ambiguities (e.g. unspecified integer storage sizes in C), and even if their specifications were not given in natural language, the engineering processes surrounding them require fallible humans to accurately identify the meaning of each artifact written in them. As a result, a developer's misunderstanding of a language's semantics could lead to error. For example, a developer writing a formal specification in Z might forget which decorated arrow symbol corresponds with which type of relationship and so express a relationship other than that intended. Such errors become hard to reveal if the same misunderstanding is propagated to multiple artifacts (e.g. the specification and the source code) during development.

MBCS manifestation. Several formal artifacts appear in the MBCS fitness argument: the PVS specification, SPARK annotations, Ada and assembly language source code, the linker script, annotations for the WCET and stack usage analysis tools, and various tool configurations. Of these, the strength of the main fitness claim rests most heavily upon human understanding of the formal specification. While the result of a misunderstanding of the semantics of Ada or of the SPARK annotation language should be caught during Echo formal verification, there is no more authoritative artifact against which the specification can be mechanically checked.

Potential mitigations. Developers should be adequately trained and skilled in the languages that they employ, but training cannot guarantee perfect understanding. Mechanical verification of each formal artifact against a more authoritative

artifact cannot address errors in the most authoritative artifacts. Moreover, the use of independent personnel in inspections and hand proofs, while helpful, cannot guarantee a sufficiently low rate of misunderstanding-based errors.

4.4 Reliance Upon Reviews Or Inspections

Reviews or inspections, properly performed, can be an effective tool for finding and eliminating defects [14]. Unfortunately, inspections, however performed, are performed by fallible humans and cannot *guarantee* the absence of a defect.

MBCS manifestation. We rely upon inspection to validate: (1) a separate argument showing that the specification refines the requirements; (2) the loop bounds and other configuration of the WCET tool; (3) the hand-generated bootstrap code; (4) the linker script; (5) the (non-SPARK) `memset` and `memcpy` routines; (6) the hardware interface routines, some written in assembly language; and (7) our usage of floating-point arithmetic.

In our fitness argument, shown in Fig. 2, the refinement sub-argument rooted at **ST_ArgOverRefinement** relies solely upon inspection to establish the correctness of non-SPARK code, including the absence of side effects that would invalidate the assumption of non-interference made during formal verification of the remaining code. However, functional testing evidence complements this sub-argument as shown in the figure.

Potential mitigations. Inspections can be limited to specific parts of specific artifacts, focused on answering specific questions, and structured so as to force a thorough and systematic examination of the artifacts [14]. While these improvements increase the overall rate at which inspectors find specific kinds of defects in specific work products, their use cannot justify concluding with high confidence that the inspected work products are free of the defects in question.

4.5 Reliance Upon Human Compliance With Protocols

In many software development efforts, fallible human developers are required to precisely follow certain protocols. A configuration management protocol, for example, might require that a developer use configuration management software to label the version of source code to be built or ensure that the build machine is configured with the required version of the compiler and any libraries used. These protocols establish important properties, e.g. which versions of the source artifacts correspond with a given version of the executable. Should the developer fail to follow the protocol precisely, the property might not be established.

MBCS manifestation. The MBCS fitness argument requires human compliance with: (1) an integration testing protocol; (2) an argument review protocol; (3) a configuration management protocol; and (4) source code and tool configuration inspection protocols. In the cases of (1), (2), and (4), compliance is forced by requiring developers to sign off on the completion of protocol steps. Compliance

with the configuration management protocol is cited in many areas of the fitness argument, as the protocol is used to guarantee that the various testing and analysis activities were conducted on the correct version of the correct artifact.

Potential mitigations. Checklists and sign-off sheets can help to ensure that developers are aware of the responsibilities imposed upon them by a protocol. However, even if they are taken seriously rather than treated as meaningless paperwork, a developer might still misunderstand the protocol's implications and sincerely indicate compliance without actually establishing the needed property.

4.6 Reliance Upon Unqualified Tools

The fitness of software often relies on one or more of the tools used in its production. A defective compiler, for example, might produce an unfit executable from source code that has been mechanically proven to refine a correct formal specification. Clear and convincing evidence that tools are demonstrably fit for the use to which they are put is, unfortunately, rarely available. Even when available, such evidence will necessarily be limited in the same ways as evidence of the fitness of any software product.

MBCS manifestation. The MBCS fitness argument contains 14 instances of a "correct use of correct tool" argument pattern, explicitly denoting reliance upon the WCET tool, the test coverage tool, the test trace collection mechanism, the test execution and reporting tool, the PVS theorem prover, the Echo specification extractor [22], the SPARK Examiner, the SPARK POGS tool, the SPADE Simplifier, the SPADE proof checker, the Echo code transformer [22], the stack usage tool, the compiler (including its Ada compiler, PowerPC assembler, and linker), and the disassembler. In some of these cases, a defect in the tool or its configuration is unlikely to result in unfit software. A defective Echo transformer, for example, would be unlikely to produce transformed source code that did not preserve the semantics of the original and yet satisfied formal proofs of functional correctness. In other cases, correctness of a given tool is relied upon more heavily. The only provisions we made for catching an error introduced by a defective compiler, for example, are the functional and integration testing.

Potential mitigations. Ideally, developers of critical software systems would be able to choose from a range of tools, each accompanied by assurance of correctness that is adequate given the use to which the tool will be put. In the absence of such tools, developers must employ a development process in which an error introduced by an unqualified tool is adequately likely to be caught.

4.7 Reliance Upon Tools That Lack Complete Hardware Models

Ideally, an embedded system developer would specify the desired system behavior in terms of signals visible at the boundary between the computer and the larger system and then use mechanical tools to prove that the software, running on the

target computer, refines that specification. Unfortunately, the present generation of analysis tools typically models the hardware more abstractly and cannot easily be used to verify software functionality in this complete end-to-end sense.

MBCS manifestation. As part of the Echo formal verification to which we subjected the MBCS, we documented the behavior required of the implementation's subprograms using SPARK annotations. We then used the SPARK tools to prove that our implementations satisfied these specifications. (We used PVS proofs to complete the verification by showing that the subprograms, taken together, refine the formal specification.)

Unfortunately, this approach did not allow us to verify all aspects of the MBCS's hardware interface routines. We could not prove, for example, that a loop waiting on a hardware flag indicating the completion of analog-to-digital conversion would terminate in bounded time. Such a proof would require knowledge that the writes to memory-mapped variables that preceded the blocking loop would cause the hardware to set the flag in question in bounded time. Lacking an end-to-end solution for formally verifying this code, we relied upon a combination of testing and inspection to establish the needed properties.

Potential mitigations. Formal models of the behavior of computing hardware almost certainly exist, as they would be indispensable in the verification of the hardware. If these models could be extracted and translated into a form that could be used by software verification tools, it might be possible to extend formal verification to routines that interact with peripherals.

We note that a complete approach to end-to-end formal verification of embedded software might require using multiple tools. Tools such as the SPARK tools, which are based on a pre- and post-condition model, might need to be complemented by a tool such as a model checker that supports Linear Temporal Logic modeling. Such a combination would allow us to prove a more complete set of properties provided techniques were developed to permit the synergistic operation of the proof systems and a machine checked synthesis of the results.

4.8 The Unavoidable Use Of Low-Level Code

High-level languages operate on an abstract model of the machine. When this model is inadequate, either because it does not permit control over some aspect of machine state that has been abstracted away or because efficiency needs preclude the use of a compiler, developers must write code in a low-level language such as the target assembly language. The verification technique chosen for the high-level code may not be applicable to this low-level language, the nature of which might limit the available analysis tools and techniques.

MBCS manifestation. We used the GNAT Pro Ada compiler to target the bare microcontroller with no Ada run-time library. While this obviated the need to procure a suitably-qualified operating system and standard libraries, it created the need for a startup routine that would configure the microcontroller to the

state required for executing compiler-generated code. Our startup routine, written in PowerPC assembly language, configured the memory controller and enabled the microcontroller's floating-point unit. This choice also obliged us to supply implementations of the `memcpy` and `memset` library routines that are called by compiled code. Because implementing these routines required using access types, they had to be coded in plain Ada rather than SPARK Ada and verified via inspection (and testing) rather than formal verification.

Potential mitigations. Verification of assembly-level code and hardware interactions has been shown to be feasible [2, 3]. A hybrid verification using multiple tools and techniques would allow the developer to exploit the unique capabilities of each tool, again with the difficulty of showing that the combination of techniques selected permits complete verification of the entire program.

4.9 The Ability To Verify Floating-Point Arithmetic

Functional requirements for computations are often conceived of in terms of real-valued or integer-valued arithmetic, but digital computers can only implement arithmetic on finite types. In the case of integer arithmetic, the practical distinction is well understood by programmers, who take care to allocate enough storage to handle the largest and smallest values a given variable might take on. The distinction between real-valued arithmetic and its floating-point approximation is less-well understood by average programmers. Even if each step is required to be correct by the IEEE-754 standard, the floating-point semantics (rounding and exceptions) might make the behavior of a program difficult to foresee and analyze.

MCBS manifestation. The Echo approach to formal verification treats floating-point arithmetic as if it were real-valued arithmetic with a bounded range. It tells us that our implementation does not use one variable when another was meant or multiplication where addition was meant. It cannot, however, tell us whether single-precision floating-point arithmetic is adequately precise for this application. We are forced to assume that it is.

Potential mitigations. Using floating-point computation to adequately substitute for real-valued arithmetic is quite complicated. Programmers might know rules of thumb such as "don't test for equality" and "avoid adding numbers of vastly-dissimilar exponent," but most programmers are not experts in the numerical field and we should not rely on the programmers' knowledge to produce correct floating-point arithmetic. Formal methods have been successfully used both for hardware-level and high-level floating-point arithmetic. If the rounding and approximation semantics are built into verification condition generation for the source code, one might be able to reason about the bound between floating-point results and the real-value results that they approximate. Such a technique does exist [4] and has been demonstrated to be useful for small C programs.

4.10 Reliance Upon Testing

Requirements-based testing is used in all software projects to establish that the software meets its requirements, either alone or in parallel with formal verification evidence. Demonstrably adequate testing, however, is difficult or impossible for many critical software projects for several reasons:

1. Toy examples aside, *complete* input space coverage is unattainable. Furthermore, testing sufficient to establish high levels of reliability is infeasible [5].
2. The strength of testing evidence is limited by the degree to which one can trust the test oracle not to pass a test that should fail.
3. Instrumentation of the tested software might be required, making it uncertain that the results apply to the software which will be released.
4. Special computing hardware might be required, making it uncertain that the results apply to software running on the target hardware.
5. Test sequencing and test result collection tools may be defective.
6. Human developers may fail to follow the testing protocol faithfully.

MBCS manifestation. Functional testing evidence *complements* formal verification evidence in the MBCS fitness argument as shown in Fig. 2. We avoid instrumentation and assume that standard precautions such as the use of skilled, independent testers and test kit with an established history are sufficient. Since the functional correctness of the MBCS is *also* established by formal verification, some risk that the functional testing will miss an error can be tolerated.

Potential mitigations. Much has been written on the subject of testing, and there are many ways in which testing can be improved in one respect or another. However, we are aware of no approach to testing that can positively establish functionality where ultra-high levels of assurance are required.

4.11 Reliance Upon Human Assessment Of Dependability

Were the evidence of dependability perfect — the requirements, test results, and so on completely trustworthy — there would still be practical limitations on the degree to which adequate dependability could be assured. Safety cases and other rigorous assurance arguments have gained attention recently as a means of uniting and explaining dependability evidence, but the technology for validating such arguments is both immature and reliant upon fallible humans.

MBCS manifestation. Our confidence in the fitness of the MBCS rests entirely upon the sufficiency of its fitness argument. If the fitness argument were to contain a logical fallacy, for example, this confidence might be misplaced.

Potential mitigations. Validation of assurance arguments is an area of active research interest. Researchers have proposed argument review techniques [12] and taxonomies of argument fallacies to avoid [7]. In addition, research into improving the dependability of natural language communication in other domains [8] might prove useful if adapted to arguments. Nevertheless, we are aware of no approach than can positively establish the soundness of the assurance argument.

5 Related Work

The related work in specific areas of limitation have been discussed in section 4. The practical and theoretical limits of software dependability assurance have been the subject of numerous papers and discussions, e.g. [15–17], and the limitations discussed in this paper are known in isolation or related groups. Our contribution lies in analyzing the MBCS fitness argument to derive the dependability limitations affecting the MBCS and to assess the impact of each.

6 Conclusion

Analysis of our assurance argument revealed 11 major practical limitations on software dependability that affected our specimen software development effort. Each of these limitations embodied one or more of three themes: (1) dependence upon fallible human beings; (2) incomplete or immature tools or technologies; and (3) techniques that cannot *practically* ensure the needed dependability.

While time and investment may mitigate problems embodying only the second theme, the first and third themes reflect fundamental problems. Addressing these limitations in a given engineering effort requires structuring the development process so that the resulting weaknesses in dependability assurance are compensated for by the use of complementary efforts.

This case study demonstrates a significant benefit of assurance arguments: they convey comprehensively and intuitively precisely what the dependability of each software system depends upon. Furthermore, an argument explicitly documents where developers are relying upon the independence of evidence.

Acknowledgement

We thank Brett Porter of AdaCore and the AdaCore corporation for their support of the MBCS project, Paul Allaire and Houston Wood for details of the LifeFlow LVAD, and Kimberly Wasson for her constructive criticism. Work funded in part by National Science Foundation grants CNS-0716478 & CCF-0905375.

References

1. AdaCore: GNAT Pro High-Integrity Family. Web page, http://www.adacore.com/-home/products/gnatpro/development_solutions/safety-critical/
2. Alkassar, E., Hillebrand, M.A., Leinenbach, D.C., Schirmer, N.W., Starostin, A., Tsyban, A.: Balancing the load. Journal of Automated Reasoning 42(2), 389–454 (April 2009)
3. Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism. Tools and Algorithms for the Construction and Analysis of Systems pp. 109–123 (2008)
4. Boldo, S., Filliatre, J.C.: Formal verification of floating-point programs. In: Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH '07). pp. 187–194. IEEE Computer Society, Washington, DC, USA (2007)

5. Butler, R., Finelli, G.: The infeasibility of experimental quantification of life-critical software reliability. In: IEEE Trans. on Software Engineering. pp. 66–76 (1991)
6. Graydon, P.J., Knight, J.C.: Software process synthesis in assurance based development of dependable systems. In: Proceedings of the 8th European Dependable Computing Conference (EDCC). Valencia, Spain (April 2010)
7. Greenwell, W., Knight, J., Holloway, C., Pease, J.: A taxonomy of fallacies in system safety arguments. In: Proceedings of the 2006 International System Safety Conference (ISSC '06). Albuquerque, NM, USA (July 2006)
8. Hanks, K., Knight, J.: Improving communication of critical domain knowledge in high-consequence software development: An empirical study. In: Proceedings of the 21st International System Safety Conference. Ottawa, Canada (August 2003)
9. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A practical tutorial on modified condition / decision coverage. Technical Memorandum TM-2001-210876, NASA, Hampton, VA (May 2001)
10. Heninger, K.L.: Specifying software requirements for complex systems: New techniques and their application. IEEE Transactions on Software Engineering 6(1), 2–13 (1980)
11. Kelly, T.: A systematic approach to safety case management. In: Proc. of the Society for Automotive Engineers 2004 World Congress. Detroit, MI, USA (2004)
12. Kelly, T.: Reviewing assurance arguments — a step-by-step approach. In: Proceedings of the Workshop on Assurance Cases for Security — The Metrics Challenge, Dependable Systems and Networks (DSN) (July 2007)
13. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proc. of the 22nd ACM Symposium on Operating Systems Principles. Big Sky, MT, USA (Oct 2009)
14. Knight, J.C., Myers, E.A.: An improved inspection technique. Communications of the ACM 36(11), 51–61 (1993)
15. Laprie, J.C., Le Lann, G., Morganti, M., Rushby, J.: Panel session on limits in dependability. In: Highlights from Twenty-Five Years, Twenty-Fifth International Symposium on Fault-Tolerant Computing. pp. 608–613 (June 1995)
16. Littlewood, B.: Limits to dependability assurance — a controversy revisited. In: Companion to the proceedings of the 29th International Conference on Software Engineering (ICSE '07). Washington, DC, USA (2007)
17. Littlewood, B., Strigini, L.: Validation of ultrahigh dependability for software-based systems. Communications of the ACM 36(11), 69–80 (1993)
18. Lutz, R.R.: Analyzing software requirements errors in safety-critical, embedded systems. In: Proc. of the IEEE International Symposium on Requirements Engineering (RE '93). San Diego, CA, USA (January 1993)
19. SRI International: PVS specification and verification system. Web page, http://pvs.csl.sri.com/
20. Wasson, K.S.: CLEAR requirements: improving validity using cognitive linguistic elicitation and representation. Ph.D. thesis, University of Virginia, Charlottesville, VA, USA (May 2006)
21. Yin, X., Knight, J., Nguyen, E., Weimer, W.: Formal verification by reverse synthesis. In: Proc. of the 27th International Conference on Computer Safety, Reliability and Security (SAFECOMP). Newcastle, UK (Sept 2008)
22. Yin, X., Knight, J.C., Weimer, W.: Exploiting refactoring in formal verification. In: Proc. of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09). Lisbon, Portugal (June 2009)