Invited paper (slightly expanded) for a special session on the Verified Software Initiative, 12th IEEE International Conference on the Engineering of Complex Computer Systems (ICECCS), Auckland, New Zealand, July 2007, pp. 270–276.

What Use is Verified Software?*

John Rushby
Computer Science Laboratory,
SRI International
Menlo Park CA USA
rushby@csl.sri.com

Abstract

The world at large cares little for verified software; what it cares about are trustworthy and cost-effective systems that do their jobs well. We examine the value of verified software and of verification technology in the systems context from two perspectives, one analytic, the other synthetic. We propose some research opportunities that could enhance the contribution of the verified software initiative to the practices of systems engineering and assurance.

1. Introduction

The Verified Software Initiative (VSI) aims to foster the science and technology of formal verification—and the culture of software development—so that it becomes routine for software to be delivered with a guarantee of correctness.

But we as users, and larger society as stakeholders, have little direct interest in the correctness of software; what we care about are *systems* (such as those for air traffic control, credit cards, or cellphones), whose operation is the result of complex interactions among many software subsystems, and whose failures and infelicities are generally due to subtle faults in those interactions, sometimes provoked by hardware malfunction, user error, or other unanticipated combinations of circumstances, and sometimes the result of misunderstood requirements and expectations.

What is the relationship between guaranteed properties of software programs and the reliability, safety, and general felicity of systems? It is not a simple one, for it is well-known that systems built on correct programs can fail (because they are correct with respect to inadequate properties) and that satisfactory systems can contain incorrect programs (because the system shields its programs from the circumstances that provoke their faults, or because it has ways of coping with the manifestations of those faults).

Furthermore, the technology of program verification can be applied in many different ways and to many different targets and for different purposes. For example, as static analysis it can be applied to large suites of executable programs in a highly automated way but can guarantee only relatively shallow and local properties (e.g., absence of runtime errors, such as those caused by dereferencing a null pointer, or dividing by zero); as theorem proving it is often applied under skilled human guidance to rather abstract representations of small programs (e.g., as algorithms described in a specification language) and can guarantee fairly strong properties (e.g., that the algorithm achieves its purpose); and as model checking it can be used for many purposes other than verification (e.g., for test generation, bug finding, or exploration).[1] These different applications of formal verification methods support very different claims and apply to very different artifacts in the software development process.

There seem to be two existing perspectives from which to view the potential contributions of verified software and of verification technology to systems. One is the perspective of system assurance, which is best developed in its application to safety-critical systems. Specifically, software verification can be included among the evidence that supports a safety case or, more generally, an assurance case. I consider this perspective in Section 2. The other perspective, which I will call the "systems view," holds that component reliability is not the most important factor in overall system quality, and that major system failures are generally the result of unanticipated interactions among system components or between the system and its environment. I consider this perspective in Section 3. My considerations raise more questions than answers, and I conclude in Section 4 with suggestions for further research.

* This research was partially supported by AFRL through a subcontract to Raytheon, by NASA Langley contract NNL06AA07B through a subcontract to ERA Corporation, and by NSF grant CNS-0644783.

1 These are examples only; each technology can be used for other purposes as well.

2. The Assurance Perspective

Many industries require a *safety case* to be demonstrated before a potentially hazardous system may be deployed. A safety case [1] is

> "A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment."

A safety case is generally structured as an explicit *argument*, based on documented *evidence* that supports suitable *claims* concerning system safety. This general approach is widely applicable, so that one hears of "security cases," or "dependability cases." Beyond critical systems, this seems a rational framework for justifying propositions that may be made about any particular system and the goals it is intended to achieve, and I refer to the general approach as providing an *assurance case*.

Formal verification is among the evidence that might be considered in an assurance case, but it is unlikely to be the only evidence. This is because the correctness properties that have been verified might not include everything that is important about the system, because only some parts of the system might have been formally verified, because the verification itself may be considered fallible, and because some aspects of behavior may be beyond the reach of formal verification (a topic that is considered in Section 3). Consequently, we need a way to assemble multiple items of evidence and their associated arguments into a coherent overall assurance case. The claims supported by most forms of evidence (and, indeed, the top level claims that we really care about) usually are conditional and are often stated probabilistically (e.g., a claim for the primary protection system for a nuclear plant might be that its probability of failure on demand (PFD) is less than 10^{-3}). The claims supported by formal methods, on the other hand, usually are unconditional (e.g., this program will generate *no* runtime errors). But although the claim may be unconditional, there will be some uncertainty about the evidence itself—even formal methods may be fallible—which can be expressed as a subjective probability; thus we may speak of 99.9% confidence that static analysis supports a claim of no runtime exceptions or, in the conditional case, of 95% confidence that testing evidence supports a claim of 10^{-3} PFD.

We now need a method for "adding up" multiple forms of evidence, in which we have different degrees of confidence, to support a possibly conditional claim: this is called a "multi-legged" assurance case [3]. Bayes theorem is the principal tool for analyzing subjective probabilities [14]: it allows a prior assessment of probability to be updated by new evidence to yield a rational posterior probability. It is technically difficult to deal with large numbers of complex conditional (i.e., interdependent) probabilities but Bayesian Belief Nets (BBNs) provide a graphical way explicitly to represent dependence among different items of evidence, and they are supported by tools (e.g., HUGIN Expert [13]) that can perform the necessary calculations to estimate posterior probabilities.

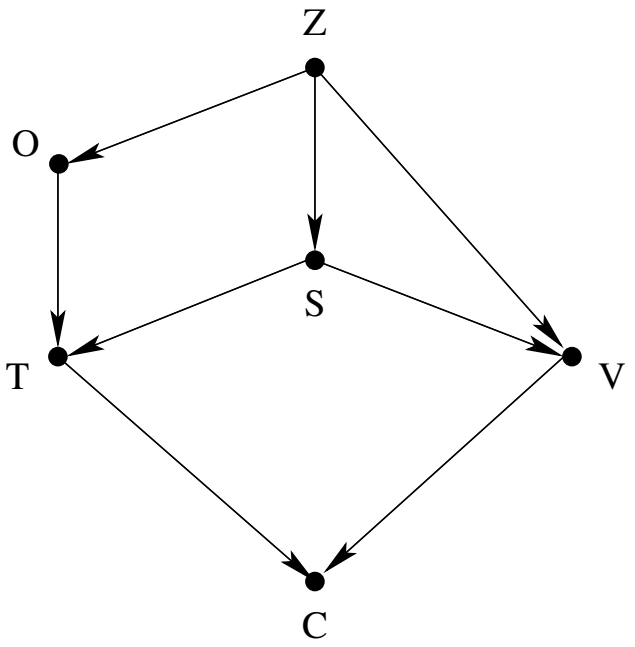

Figure 1. BBN For A Two-Legged Assurance Case (from [17])

Littlewood and Wright [17] examine a two-legged assurance case whose BBN is shown in Figure 1. Here, evidence from testing is combined with formal verification; the nodes represent judgments about components of the argument and the arcs indicate dependence between these. In particular, the node Z concerns the specification for the system and the analysis must consider two possibilities: that it is `correct` (i.e., accurately represents the true requirements on the system) or `incorrect`. The evaluator must attach some prior probability distribution to these possibilities (e.g., 99% confidence it is `correct`, vs. 1% that it is `incorrect`). The node V represents the outcome of formal verification (i.e., `pass` or `fail`); we presumably will undertake some remedial action if the verification fails, so we are only concerned with the case that it passes. The node S represents the true (but unknown) quality of the system (e.g., its probability of failure on demand, in which case S will have a value between 0 and 1). There are arcs from Z and S to V because the verification outcome should surely depend on the correctness of the specification and the quality of the system. O is the test oracle; it is derived in some way from the

specification Z and may be `correct` or `incorrect`—the probability distribution over these will be some function of the distribution over the correctness of Z (e.g., if Z is `correct`, we might suppose it is 95% probable that O is `correct`, but if Z is `incorrect`, then it is only 2% probable that O is `correct`). T is the outcome of testing: that is, whether or not failures were discovered. Again, we presumably will fix things if failures are discovered in testing, so we are only concerned with the case of `no failures`. T depends on the oracle O and the true quality of the system S and its probability distribution over these will represent the evaluator's confidence in the test quality (as indicated by coverage measures, or mutant detection, for example). Finally, the node C represents the outcome or conclusion of the analysis; presumably this will be to `accept` the system only if the system `passes` verification and `no failures` are discovered in testing.

In this example, only T and V are directly observable, and C is fully determined by these. Using a BBN tool, it is possible to conduct "what if" exercises on this example to see how prior estimates for the conditional probability distributions of the various nodes are updated by the evidence (i.e., that verification passes and that testing finds no errors) and thereby to determine the posterior probability distribution that the conclusion is correct. Rather than "what if" exercises with a tool, Littlewood and Wright [17] examine this example symbolically. They observe that surprising outcomes are possible; for example, if the prior probability distributions on T are changed to represent harder (or more numerous) tests, and still no failures are detected, this may weaken confidence in correctness of the test oracle rather than increase confidence in the conclusion. They show that these surprising outcomes are eliminated when the claim supported by verification is unconditional (i.e., when formal verification supports the claim of "perfection," $S = 0$).

This is an attractive conclusion: it suggests that the unconditional character of formal verification evidence yields significant added value. However, Littlewood and Wright's analysis assumes that the correctness property guaranteed by formal verification is the full specification for the system (whose correctness is represented by the single node Z). As we noted earlier, formal verification may consider only weak properties, such as absence of runtime errors, or properties of an abstraction such as correctness of an algorithm.

If the formally verified properties are fragments of the full specification, then I believe we can split the BBN into two: one that considers the union of these verified fragments, which we can represent as Z', and one that considers the rest of the specification $Z'' = Z \backslash Z'$ (I am abusing notation here and using these symbols to represent both the specifications themselves, and the quantities actually used in the BBNs, which are estimates of their correctness). Analysis of Z'' proceeds without formal evidence, while that of Z' can use Littlewood and Wright's insights. A complicating factor is that the formally verified artifacts may be algorithms or other intermediate products rather than the actual system S, but I suspect this can be handled by adding nodes to the BBN to represent these artifacts (though only the nodes corresponding to these artifacts will have unconditional claims).

Adaptations to the BBN of Figure 1 seem more problematic when formal verification has delivered only implicit properties such as guaranteed absence of runtime errors. It is not viable, in my opinion, to treat such implicit properties as conjuncts of the full specification; they are at best derived properties that should be entailed by the full specification. It might seem that we could then extend Figure 1 by adding a Z' to represent the formal verified properties, and we might expect that the entailment of Z' by Z allows relatively straightforward analysis. Unfortunately, this may not be so.

Philosophers interested in the scientific method study topics similar to those considered here; they are interested in the extent to which evidence supports one hypothesis rather than another, and have notions of the *coherence* of evidence [4] and a general topic of *confirmation theory* [8]. The roots of much of their analysis lie in attempts to construct a Bayesian account of inductive reasoning that would be a close analog to classical logic for deductive reasoning [5]. It might be hoped to combine the two forms of reasoning, so that if evidence E supports a hypothesis H and H deductively entails H', then surely E should also support H'. This expectation is dashed under any plausible probabilistic interpretation of "supports" by the following counterexample. Let H be the hypothesis that a card drawn at random from a shuffled deck is the Ace of Hearts, let H' be the hypothesis that the card is red, and let E be the evidence that the card is an Ace. Certainly H entails H' and E supports H, but E cannot be considered to support H'. (I learned this counterexample from a talk by Brandon Fitelson of UC Berkeley; his website `http://fitelson.org/` contains much material on these topics.) It is interesting in this context to note that some exponents of goal-based assurance look to Toulmin [27] rather than classical logic in framing assurance cases [2]; Toulmin stresses *justification* rather than *inference*.

Inquiries by philosophers also raise interesting questions on how to estimate the strength of evidence. It seems implicit in the BBN approach that the extent to which evidence E tends to support hypothesis H is some function of the prior probability $P(H)$ and the posterior probability $P(H|E)$. Fitelson [8] considers measures related to these and other conditional probabilities and gives compelling arguments that the best are those that compare $P(E|H)$ and $P(E|\neg H)$ (in particular, the logarithm of their ratio is the

single most attractive choice). These measures are very different from one another and I suggest that some review the philosophers' considerations will be useful in developing multi-legged assurance cases.

The conclusion I draw from this discussion and the counterexample above is that it may not be straightforward to develop schema for multi-legged assurance cases that use evidence of formal verification for weak properties. In particular, the entailment relationships among the various partial specifications may not yield simpler BBNs than unrelated analyses. On the other hand, their unconditional character should allow all the formal analyses to be "added up" separately in a fairly simple way (as evidence for the unconditional conjunction of their separate claims), and only that "sum" need be added into the full BBN.

Absent more principled analyses that might follow from reexamination of multi-legged cases to include formal evidence for weak properties, we can describe an intuitive argument why this evidence may be valuable. This is an argument I call "coupling," based on use of this term for a similar idea in testing [19]. The idea from testing is that tests that expose simple errors often catch subtle ones too; transferred to verification, it is the idea that violation of a small property may indicate violation of a big one too.

Formal verification, even for weak properties, has the attribute that it considers all possible executions. Thus, formal verification may detect violation of a weak property by discovering an unanticipated scenario; the detected violation (e.g., a runtime error) acts as a "canary in the mine" that alerts us to overlooked cases that require deeper consideration. Testing might overlook the scenario because the tester shares the same lacunae as the developer, or because the scenario is very rare and difficult to construct, but formal verification will find it because it considers every case. I suspect it is this examination of all possible scenarios that explains how static analysis has been able to find bugs in avionics code that had already been subjected to the testing and other assurance methods for the highest level of FAA certification (DO-178B Level A) [9]. Viewed from this perspective, it seems that the main value in static analysis and other formal methods that examine implicit or local properties is that they provide a check on the efficacy of other assurance activities: if testing and other assurance methods did not find errors uncovered by static analysis, then they cannot have been thorough enough and should be reexamined.

Verification for weak properties has obvious value when it exposes otherwise undetected problems; it is less obvious what value should be attached to successful verifications of this kind. Certainly, those who espouse the system view attach very little value, and it is this perspective that we consider next.

3. The System Perspective

Accident analysis is a mirror-image to assurance; by studying how things fail, we can learn how to develop them so they will not fail (at least, not in the same way as the last accident) and how to provide assurance that we have done so. The traditional view of accidents, which developed in the mid-20th century, was that they are triggered by (often multiple) component faults that lead to a cascading chain of further events and, ultimately, to some bad outcome. Remedies suggested by this analysis are to use reliable components, to detect latent faults, and to have mechanisms that interrupt the cascade. A more recent view, famously introduced by Perrow [20], is the notion of a *system accident*. Here, accidents are not (mainly) the result of component failures but of flaws in the system as a whole, which can create interactions among its components so that bad outcomes follow from (what were thought to be) correct behaviors. Perrow identifies *interactive complexity* and *tight coupling* as system attributes that contribute to accidents. Leveson [15, 16] develops related ideas, with particular applications to computer-intensive systems.

Those who adopt the system perspective focus much attention on human organizations and related topics (e.g., the notion of *resilience* [12]) rather than specific engineering technologies such as formal verification. However, I think it is fair to say they would attach relatively little importance to verified software as a contribution to system safety. This is because they see software as a component and do not regard component reliability as the main issue: rather it is interactions between components where the big problems lie. Thus, Leveson, in particular, places great stress on requirements engineering, but treats it from the point of view of human problem solving.

One can agree with much of the systems view without agreeing with all its diagnoses and prescriptions. In particular, we have the "luxury" of system accidents only because components have become sufficiently reliable that they are no longer the chief precipitators of accidents—and the technology of formal verification may be, or may become, the most effective and cost-effective way to ensure reliable software components. However, the systems view is surely correct to identify the importance of interactions among components and the crucial significance of good requirements engineering. The verified software initiative will not achieve its full potential if it focuses solely on verification of software with respect to its specifications without also addressing correctness and suitability of those specifications and the requirements from which they are derived.

Conversely, traditional requirements engineering needs help, for it demands great feats of human imagination: we have to imagine the interaction of the proposed system with its environment (to identify both its desired function and un-

desired hazards), imagine its design and its components and imagine their interactions, and so on. Imagination may be supported by sketches and physical models or prototypes, and guided by checklists and by a carefully managed engineering process, but it is chiefly a mental activity, and a difficult one that benefits from long experience. We should not expect—nor desire—to eliminate the need for human imagination, intelligence, and experience from this process, but surely we can augment these precious resources by the power of computation.

The recent and growing adoption of model-based development has created what seems to me a once-in-a-lifetime opportunity to apply the technologies underlying formal verification to the important topics of requirements analysis and development. Model-based design environments such as Esterel/SCADE, Matlab/Simulink/Stateflow, AADL, or UML provide graphical specification notations based on concepts familiar or acceptable to engineers (e.g., control diagrams, state machines, sequence charts), methods for simulating or otherwise exercising specifications, and some means to generate or construct executable programs from the models. Until the advent of model-based methods, artifacts produced in the early stages of system development were generally descriptions in natural language, possibly augmented by tables and sketches. While they could be voluminous and precise, these documents were not amenable to any kind of formal analysis. Model-based methods have changed that: for the first time, early-lifecycle artifacts such as requirements, specifications, and outline designs have become available in forms that are useful for mechanized formal analysis. Some of the notations used in model-based design environments have quite awkward semantics, but they present no insuperable difficulties (see, e.g., [11]) and formal methods have been applied successfully to most model-based notations.

The opportunity as I see it is to combine the strengths of man and machine: people are good at describing how things work and at stating some of the things they do and do not want to happen, but they are not good at imagining the consequences of collections of such descriptions and statements;[2] computers are good at tireless calculation and, in the guise of formal methods they can calculate these consequences for us. The unique value of formal methods is that they can compute properties of all reachable states, and this extends their value far beyond that of simulation, which can merely sample that space. The use of simulation in model-based development does provide a significant benefit, however, which carries over to formal methods: namely, design models are augmented by models of the environment (e.g., the controlled plant, in the case of embedded systems) and these are no less valuable in verification than in simulation.

Another distinctive value of formal methods is that they can calculate properties of highly abstract models: in the early stages of exploration, a few axioms may adequately characterize a component and may serve our purposes better than a detailed model. (Training engineers to appreciate and exploit abstraction may be one of the more difficult tasks in technology transfer for formal methods.) Through reachability analysis, initial models and properties can be iteratively refined as oversights and undesirable behaviors are discovered, and a more complete, precise, and consistent requirements specification can be developed through this symbiosis of man and machine. Many traditional safety engineering analyses such as hazard analysis and failure modes and effects analysis can be seen as informal ways to do reachability analysis, and these can be recast as formal analyses and integrated in this process. An early and partial, but very encouraging, application of this approach is described by researchers at Rockwell Collins [18]. Counterexamples generated by formal analysis can be used to drive the simulator of the modeling environment, or they can be presented to the user in one of its modeling notations (e.g., as message sequence charts).

A weakness in my advocacy of formal analysis for model-based designs in requirements development is that there generally are many stakeholders, each with a partial view of the system, and often conflicting expectations; each of these may develop and analyze their own models, but then we need ways to integrate these and to discover and reconcile their inconsistencies. Integration is not easy because each constituency may have its own modeling methods that are entirely silent about the concerns of others (e.g., the scheduling people may say nothing about security, and vice-versa), yet certain topics cut across both (e.g., covert timing channels in security). Or we may find that different constituencies have specified conflicting requirements (e.g., those scheduling the CPU and those scheduling the bus may violate each others assumptions). I think these difficulties should be seen as research opportunities, and there are already some encouraging developments, such as those that show how modeling and analysis for real time can be undertaken within a standard state machine framework [7]. There are notations such as the Architecture Analysis and Design Language (AADL) [23] that allow a single model to be annotated in different ways, but its semantics are weak for formal verification and do not support cross-cutting analyses. These limitations should be seen as a further research opportunity: we need to find ways to establish that different views are projections of a common model and to combine specialized analyses performed along different projections.

2 In evidence, I cite one very experienced software architect who explained that there are two phases in requirements acquisition: one performed at the beginning of the project, and a second performed after the first attempt at component integration reveals how much has been overlooked. The idea here is to move the second phase into the first, by using formal methods to explore "integration" issues early in the development.

Whereas the assurance perspective encourages us to seek ways in which the guarantee of correctness conferred on software by formal verification can be elevated to support claims about the overall system, the systems view encourages us to think about how the technology of formal verification can help us engineer good systems from the beginning: the first view is analytic, the second synthetic.

This synthetic view leads, inevitably in my opinion, to advocacy for *correctness by construction* [10], which is a process in which the products of every step of development are subjected to rigorous analysis, both internal to the product (e.g., static analysis of source code), and with respect to the products of earlier steps (e.g., specification-based testing of the source code); this is in contrast to the traditional "V Model," where the verification and validation steps follow the development steps. The idea is to find and fix problems early, and before moving on to the next stage. Such approaches are widely advocated in safety engineering (e.g., [21, 24, 25]), where intensive (informal) verification is performed within each step and extensive *traceability* is required from one step to the next. The difference is that the technology of formal verification could provide automated assistance for many of these activities, thereby reducing their cost and increasing their efficacy. Examples include automated generation and monitoring of tests (at the integration and systems levels, not merely the unit level), model exploration (e.g., "show me an execution in which both these states are active and this value is zero"), and improved specification and enforcement of constraints on programming at the unit level.

To illustrate the last point: faults often arise at the interfaces between software components. Extended type annotations for interfaces would allow formal analysis of limited—but better than current—checks that components respect their interfaces. Stronger checks require specification of how the interface is to be used (e.g., a protocol for interaction); *typestate* [26] and *interface automata* [6] provide ways to do this. Formal methods can then attempt to verify correct interface interactions, or can generate monitors to check them at runtime or test benches to explore them during development (rather like the bus functional models used in hardware).

Integration frameworks such as the Time Triggered Architecture (TTA) and operating system kernels for partitioning and separation provide yet stronger mechanisms for enforcing interfaces: those of well-behaved components are guaranteed, even in the presence of faulty and malicious components. Formal verification of these frameworks is a challenging undertaking, but one that reduces the burden for other components. I discuss these and related topics in a companion paper [22].

4. Summary and Recommendations

Systems are more than software and the relationship between verified software and trustworthy and attractive systems is not simple. I have outlined two ways in which verified software and the technology of formal verification can contribute to high quality systems.

The first way is analytic: it uses verification as evidence in developing an assurance case for the system concerned. Verification will be combined with other evidence, so we are concerned with multi-legged assurance cases, and I described some of the benefits and difficulties in using verification evidence in such cases. The difficulties raise interesting research questions for those skilled in BBNs and other methods for analyzing and combining evidence: in particular, how to factor in evidence delivered by static analysis (where the properties verified are not directly related to the system specification), and how to respond to issues raised by philosophers working on confirmation theory.

The second way is synthetic: it uses verification technology to aid in the construction of high-quality systems (an approach sometimes called correctness by construction). The engineering challenges here are to integrate verification technology into the processes and tools used in systems engineering; the rise of model-based development provides an opportunity to do this. The research challenges are to find ways to deliver the singular advantages of formal analysis (the ability to work with highly abstract models, and the ability to explore all reachable states) in contexts where knowledge (e.g., of the real world, or of the customer's expectations) is imperfect, where some requirements may conflict, and where properties other than functional correctness (e.g., cost, performance) must also be considered.

The value of both analytic and synthetic formal verification will surely increase as systems become more interconnected and subject to constant evolution. It is no longer sensible to think of systems as ever finished: components are modified and added as new needs or opportunities emerge, whole subsystems are grafted on, and deliberate and accidental integrations are created between previously separate systems. The local mechanics of adaptation and integration may be mastered while emergent properties, both good and ill, are left to chance. Medical systems provide interesting examples: many devices that each manage some aspect of physiology can be attached to a single patient, creating an accidental system of systems that interact through the controlled plant—the patient. It is known that patients respond better when different elements of their physiology operate in harmony (e.g., so many heartbeats to each breath) but the separately designed devices each manage their own parameter in ignorance of the others.

Manual methods of analysis and design have limited utility in the face of continual evolution: it is hard to apply these

methods to a single static system and vastly harder to revisit the assurance case or the requirements capture or design rationale for separate systems and components, years after their initial construction, to explore the consequences of modifications, extensions, or integrations. But automated formal methods bring the same scrutiny to a specification many years later as on the day of its creation, and in juxtaposition with new environment specifications as with the old: they are a reusable asset.

Acknowledgments. Presentations and discussions at meetings for the verified software initiative and its earlier incarnations helped me formulate my views on these topics, as did discussions with my colleagues Rance DeLong and Shankar, and with Martyn Thomas. I am grateful to Robin Bloomfield and Bev Littlewood and their colleagues for educating me on safety cases and BBNs during a visit to CSR at City University in November 2006.

References

[1] P. Bishop and R. Bloomfield. A methodology for safety case development. In *Safety-Critical Systems Symposium*, Birmingham, UK, Feb. 1998. Available at `http://www.adelard.com/resources/papers/pdf/sss98web.pdf`.

[2] P. Bishop, R. Bloomfield, and S. Guerra. The future of goal-based assurance cases. In *DSN Workshop on Assurance Cases: Best Practices, Possible Obstacles, and Future Opportunities*, Florence, Italy, July 2004. Available from `http://www.aitcnet.org/AssuranceCases/agenda.html`.

[3] R. Bloomfield and B. Littlewood. Multi-legged arguments: The impact of diversity upon confidence in dependability arguments. In *The International Conference on Dependable Systems and Networks*, pages 25–34, San Francisco, CA, June 2003. IEEE Computer Society.

[4] L. Bovens and S. Hartmann. *Bayesian Epistemology*. Oxford University Press, 2003.

[5] R. Carnap. *Logical Foundations of Probability*. Chicago University Press, second edition, 1962.

[6] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. Association for Computing Machinery, 2001.

[7] B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, Grenoble, France, Sept. 2004. Springer-Verlag.

[8] B. Fitelson. *Studies in Bayesian Confirmation Theory*. PhD thesis, Department of Philosophy, University of Wisconsin, Madison, May 2001. Available at `http://fitelson.org/thesis.pdf`.

[9] A. German. Software static code analysis lessons learned. *Crosstalk*, Nov. 2003. Available at `http://www.stsc.hill.af.mil/crosstalk/2003/11/0311German.html`.

[10] A. Hall. Software verification and software engineering: A practitioner's perspective. In N. Shankar, editor, *IFIP Working Conference on Verified Software: Theories, Tools, and Experiments*, Zurich, Switzerland, Oct. 2005. Available at `http://vstte.inf.ethz.ch/papers.html`.

[11] G. Hamon and J. Rushby. An operational semantics for Stateflow. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering: 7th International Conference (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 229–243, Barcelona, Spain, 2004. Springer-Verlag.

[12] E. Hollnagel, D. D. Woods, and N. Leveson, editors. *Resilience Engineering*. Ashgate, 2005.

[13] *HUGIN home page*. `http://www.hugin.com/`.

[14] R. Jeffrey. *Subjective Probability: The Real Thing*. Cambridge University Press, 2004.

[15] N. Leveson. A new accident model for engineering safer systems. *Safety Science*, 42(4):237–270, Apr. 2004.

[16] N. G. Leveson. *Safety Engineering: Back to the Future*. Draft available at `http://sunnyday.mit.edu/book2.pdf`.

[17] B. Littlewood and D. Wright. The use of multi-legged arguments to increase confidence in safety claims for software-based systems: a study based on a BBN analysis of an idealised example. *IEEE Transactions on Software Engineering*, 33(5):347–365, May 2007.

[18] S. P. Miller, A. C. Tribble, and M. P. E. Heimdahl. Proving the shalls. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *International Symposium of Formal Methods Europe, FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 75–93, Pisa, Italy, Mar. 2001. Springer-Verlag.

[19] R. A. D. Millo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, Apr. 1978.

[20] C. Perrow. *Normal Accidents: Living with High Risk Technologies*. Basic Books, New York, NY, 1984.

[21] Requirements and Technical Concepts for Aviation, Washington, DC. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, Dec. 1992. This document is known as EUROCAE ED-12B in Europe.

[22] J. Rushby. Just-in-time certification. In *12th IEEE International Conference on the Engineering of Complex Computer Systems (ICECCS)*, pages 15–24, Auckland, New Zealand, July 2007. IEEE Computer Society. Available at `http://www.csl.sri.com/~rushby/abstracts/iceccs07`.

[23] *AADL home page*. `http://www.aadl.info/`.

[24] Society of Automotive Engineers. *Aerospace Recommended Practice (ARP) 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems*, Nov. 1996.

[25] Society of Automotive Engineers. *Aerospace Recommended Practice (ARP) 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, Dec. 1996.

[26] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, Jan. 1986.

[27] S. E. Toulmin. *The Uses of Argument*. Cambridge University Press, 2003. Updated edition (the original is dated 1958).