Presented at the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM), Hanoi, Vietnam, November 2009. Appears in the SEFM proceedings pp. 3–10 ©IEEE Computer Society.

Software Verification and System Assurance

(Invited Paper)

John Rushby
Computer Science Laboratory
SRI International
Menlo Park California USA
Rushby@csl.sri.com

***Abstract*—Littlewood [1] introduced the idea that software may be *possibly perfect* and that we can contemplate its probability of (im)perfection. We review this idea and show how it provides a bridge between correctness, which is the goal of software verification (and especially formal verification), and the probabilistic properties such as reliability that are the targets for system-level assurance. We enumerate the hazards to formal verification, consider how each of these may be countered, and propose relative weightings that an assessor may employ in assigning a probability of perfection.**

***Keywords*-formal verification, assurance, reliability, probabilistic assessment, possible perfection**

I. Introduction

Using formal methods to find errors in designs or programs provides immediate satisfaction and measurable payoff. The same is true for some other applications of mechanized formal methods such as generating test cases, and synthesizing monitors. But what about formal verification? This was the original application for formal methods, and the motivation for much early research in the field, but proving the absence of errors seems now to attract less interest than revealing their presence. Part of the reason is surely that the claim to "prove the absence of errors" is overly broad and must be circumscribed with numerous caveats concerning the fidelity of models, validity of assumptions, soundness of deduction, and so on.

Another part of the reason may be that the claims we want to make at the system level are usually probabilistic (e.g., no more than so much downtime per year, no more than so many failures of a certain severity in the lifetime of the system, and so on) and it is not obvious how the absolute claims of formal verification can be used to support these.

This paper examines these topics and considers the claims supported by formal verification and their relation to system-level assurance, particularly for critical systems. In Section 2, drawing on the background to the results reported in [2], we describe probabilistic assessments of software, including the traditional notions of reliability and the novel idea of "possible perfection." We then describe aleatory and epistemic probabilities of perfection, and their utility in system-level assurance arguments. Section 3 considers the assessment of epistemic probabilities of perfection and the contribution of formal verification to this endeavor. We examine the issues that an assessor might face when employing evidence from formal verification to assign a probability of perfection. Section 4 concludes and offers suggestions for further research.

II. Software Probabilities and the Idea of Possible Perfection

Physical components of a complex system can wear out or break, possibly causing the system to fail. Failures can be graded according to the severity of their worst outcome, and it is usually required that there should be an inverse relationship between that severity and the likelihood of the failure. (This is to control *risk*, which is the product of the severity of an outcome and its likelihood.) In commercial aircraft, for example, *catastrophic* failures are "those which would prevent continued safe flight and landing" [3, paragraph 6.h(3)] and these are required to be "extremely improbable" [3, paragraph 7.d], which is defined as being "so unlikely that they are not anticipated to occur during the entire operational life of all airplanes of one type" [3, paragraph 9.e(3)]. If we consider the example of an airplane type with 100 members, each flying 3,000 hours per year over an operational life of 33 years, then we have a total exposure of about 10^7 flight hours. If hazard analysis reveals ten potentially catastrophic failures in each of ten subsystems, then the "budget" for each, if none are expected to occur in the life of the fleet, is a failure probability of about 10^{-9} per hour [4, page 37]. This serves to explain the well-known 10^{-9} requirement, which is stated as follows: "when using quantitative analyses…numerical probabilities…on the order of 10^{-9} per flight-hour…based on a flight of mean duration for the airplane type may be used…as aids to engineering judgment…to…help determine compliance" (with the requirement for extremely improbable failure conditions) [3, paragraph 10.b].

Software is a substantial component of many systems whose top-level safety requirements are stated probabilistically, as in the case of aircraft systems as described above. But software does not age or wear out, so how do its properties factor into a probabilistic assessment? Software contributes to system failures through faults in its requirements, design, or implementation, and these, in the language of safety analysis, produce "systematic failures," meaning they are not random but are *certain* to occur whenever circumstances activate the fault concerned. But although the failure is certain, given circumstances that activate the fault, those circumstances have a probability of occurrence: some faults are activated by

almost any input, others require very specific, and unusual combinations of inputs. Hence, failure probabilities can be associated with software (and may be stated in terms of a *failure rate*, or a *probability of failure on demand*) and are determined by the likelihood of encountering circumstances that activate its faults.

For modest values, say down to about 10^{-4}, it is feasible to measure software failure rates by statistically valid random testing [5], [6]. Here, "statistically valid" means that the test case selection probabilities are exactly the same as those that are encountered in real operation. This kind of testing has been used to assess the reliability of a protection system for a nuclear reactor [7].

But as we saw earlier, aircraft and some other systems have requirements that go to 10^{-9} and beyond, and it is infeasible to measure these tiny rates by statistically valid random testing: Butler and Finelli demonstrate that a suitable test campaign could require 114,000 years [6]. One proposal for evading this problem is "N-version software," where N independently developed and deliberately "diverse" software components operate in parallel and their outputs are voted. Assuming that failures of the different versions are independent, a naïve analysis suggests this could improve matters exponentially: a combination of N systems, each with probability of failure p, could yield a combined failure rate of p^N. Thus, the combination of three 10^{-3} versions could achieve 10^{-9}. This analysis is naïve in that it ignores the possibility of correlated failures, and both empirical [8], [9] and theoretical [10], [11] studies show that these simply cannot be ignored. If independent failures cannot be assumed, the probability of dependent failures must be measured—which is infeasible if it is small enough to be useful. For this and other reasons, the naïve idea of N-version software has largely been abandoned, but the use of deliberately simple software to "monitor" or "backup" a more complex software system is widely practiced; the reliability of these arrangements is a topic we will return to.

Because assessment by direct measurement of the very small failure rates required for critical systems is infeasible for software, standards and guidelines for such software essentially focus on processes and methods that are intended to ensure that the software is *correct*, rather than reliable. Thus, while the upper levels of aircraft system design and analysis are concerned with hazard analysis, failure modes and effects analysis, and probabilistic assessment [12], [13], the main focus of the DO-178B guidelines for airborne software [14] is on methods for ensuring that the software correctly implements its requirements. In common with most other software certification standards and guidelines, DO-178B identifies a hierarchy of "Design Assurance Levels" from E (lowest) to A (highest) and prescribes additional assurance activities at the higher levels: for example, there are 28 assurance "objectives" at Level D, 57 at Level C, 65 at Level B, and 66 at Level A. The difference between the Level B and Level A assurance objectives, as prescribed by DO-178B, is that Level A must be subjected to requirements-based testing that achieves a structural code coverage criterion called Modified Condition/Decision Coverage (MC/DC) [15], whereas testing for Level B is required merely to achieve Decision Coverage (DC).

It seems reasonable that increasing the number of assurance objectives should increase confidence in the correctness of software,[1] but it is not clear how this relates to its reliability—yet the different design assurance levels are associated with probabilistic system-level assessments. Level A is considered appropriate for software that could lead to catastrophic failure conditions, while Levels B and C are appropriate for *hazardous*, and *major* failure conditions, respectively [13]. Major failure conditions are required to be "improbable," which may be quantified as 10^{-5} [3, paragraph 10.b] and, by interpolation, we may associate hazardous failure conditions with failure rates of the order 10^{-7}. But what is it about the eight additional assurance objectives that Level B adds to those for Level C that reduce the assessed probability of failure from 10^{-5} to 10^{-7}, and how does the addition of MC/DC testing reduce this further to 10^{-9}? The last point seems particularly problematic, since testing to MC/DC coverage is a completely different kind of testing from the statistically valid testing used to assess reliability.

An insightful and original solution to this conundrum was introduced nearly a decade ago by Littlewood [1]. The idea is that the top-level claim made for critical software is not that it is reliable, but that it is *perfect*. Perfection means that the software will never suffer a failure no matter how much operational exposure it receives; it differs from correctness in that correctness is assessed relative to requirements, while perfection includes a judgment that the requirements are the *right* requirements (i.e., the detection of a failure is more primitive than noncompliance with requirements).

Now, perfection is a strong claim and we may refuse to accept that software that has been assured to DO-178B Level A is perfect—but we may be willing to concede that it is *possibly* perfect. And we may further be persuaded that its possibility of perfection is greater than software that has been assured only to Level B. This suggests we could attach a probability to the possibility of perfection. We will refine this notion later, but one way to interpret it follows the spirit of [10], [11] and invites us to think of all the software that *might* have been developed by comparable engineering processes to solve the same design problem as the software at hand; the probability of perfection is then the probability that any software randomly selected from this class is perfect. This probability will be partly dependent on the nature of the problem being solved—for a "hard" problem it might be expected that the probability is smaller than for an "easier" problem—and partly on the quality of the software engineering and assurance methods employed.

Probability of perfection is attractive because it relates more naturally than reliability to the correctness-based assurance

[1] Some would disagree with this general assertion [16], and some standards do impose assurance objectives that have scant evidence for their effectiveness.

processes used for software. But probability of perfection can also be used to estimate reliability, as we will now show.

For simplicity, we assume a demand-based system, and will consider probability of failure rather than reliability. Then, by the formula for total probability

$$\begin{aligned} &P(\text{s/w fails [on a randomly selected demand]}) \qquad (1)\\ &= P(\text{s/w fails} \,|\, \text{s/w perfect}) \times P(\text{s/w perfect})\\ &\quad + P(\text{s/w fails} \,|\, \text{s/w imperfect}) \times P(\text{s/w imperfect}). \end{aligned}$$

The first term in this sum is zero, because the software does not fail if it is perfect. We can then, very conservatively, assume that the software always fails if it imperfect, so that the first factor in the second term becomes 1. Hence,

$$P(\text{software fails}) < P(\text{software imperfect}). \qquad (2)$$

In calculations such as this, it is generally the probability of *im*perfection that is most useful. We often write of a probability of (im)perfection to refer ambiguously to both.

Another attractive attribute of possible perfection arises in two-channel systems, such as those used in aircraft or, in different form, for nuclear shutdown. In aircraft, it is common to have a highly complex "operational" channel, which is a system that actually provides the function concerned (e.g., fuel management, autopilot etc.), and a much simpler "monitor" channel that looks for safety violations and triggers higher-level fault recovery when necessary. As explained earlier, in the context of N-version software, we cannot simply multiply the failure rates of the separate channels together to obtain the failure rate of the combined system because we cannot assume the failures of the two channels are independent. In contrast, it is established in [2] that failure of the operational channel and imperfection of the monitor channel *are* conditionally independent, and their probabilities can be multiplied together to yield a probability of failure for the combined system. We believe this analysis can be used to provide a rigorous underpinning for some of the architectures recommended in the ARP 4754 guidelines for complex aircraft systems [13, Table 5–2] where, for example, it is suggested that a Level A system can be achieved by a Level C operational channel and a Level A monitor (with any coordination mechanism also required to be Level A).

The treatment of probabilities used above, both for failures and perfection, is deliberately informal and omits many important topics in probabilistic modeling. Technical details can be found in [2]. Our focus here is the application of these ideas to formal verification, and to do this it is necessary to introduce a little more of the background to probabilistic modeling: specifically, the distinction between aleatory and epistemic uncertainty [17].

Aleatory uncertainty, or "uncertainty *in* the world," can be thought of as "natural" uncertainty that is irreducible. For example, if we were to toss a coin we would not be able to predict with certainty whether it would fall as "heads" or as "tails." This uncertainty cannot be eliminated, and any predictions about future tosses of the coin can only be expressed as probabilities. This is true even when we know the probability of heads for a single toss of the coin, which we may denote p_H; when the coin is "fair," $p_H = 0.5$. Real coins, of course, may not be fair, so there will be uncertainty about the value of the parameter p_H.

This second uncertainty is *epistemic* uncertainty, or "uncertainty *about* the world." This uncertainty is reducible: we may not know whether a given coin is fair, so we can toss it very many times and observe the frequency of heads in this sequence of tosses. That frequency is an estimate of p_H and the estimate gets closer to the true value of p_H as the number of observed tosses increases, so the uncertainty reduces.

In much scientific modeling, the aleatory uncertainty is captured conditionally in a model with unknown parameters, and the epistemic uncertainty centers upon the values of these parameters—as in the simple example of coin tossing. The analysis performed in (1) is an aleatory one, and is more properly presented as a model parameterized by the probability that the software is imperfect, which we denote p_{np} and the probability that it fails, if it is imperfect, which we denote p_{fnp}. The conclusion to (1) is that the probability of system failure is given by $p_{fnp} \times p_{np}$. To apply this result, we need to assess actual values for these parameters in the particular system concerned. This constitutes the second, or *epistemic* stage of the analysis.

Probabilities in the aleatory analysis can be given a classical frequentist interpretation (by the technique, described earlier, of considering the software to be a sample drawn from the population of software that might have been developed), but the epistemic analysis most naturally employs the subjective interpretation, where probabilities reflect degrees of belief. The person or organization responsible for assessing the acceptability of the system concerned must formulate beliefs about the values of p_{fnp} and p_{np}. Most likely these beliefs will not be independent (for example, beliefs about the probability of failure if imperfect may depend on the nature of the imperfection), so they will be represented by some joint distribution $F(p_{fnp}, p_{np})$ and the probability of system failure will then be given by the Riemann-Stieltjes integral

$$\int_{\substack{0 \le p_{fnp} \le 1 \\ 0 \le p_{np} \le 1}} p_{fnp} \times p_{np}\, dF(p_{fnp}, p_{np}). \qquad (3)$$

Elicitation of experts' subjective probabilities is an active research area in Bayesian statistics, and there have been considerable advances in recent years in techniques and tools [18]. However, it is likely to be considerably easier to elicit beliefs about single rather than joint distribution functions and so we generally seek arguments that could allow assessors to separate their beliefs. If this can be done, then $F(p_{fnp}, p_{np})$ factorizes as $F(p_{fnp}) \times F(p_{np})$ and (3) becomes $P_{fnp} \times P_{np}$ where P_{fnp} and P_{np} are the means of the posterior distributions representing the assessor's beliefs about the two parameters.

Arguments that can allow an assessor to separate his beliefs include conservative assumptions (e.g., the assumption

used earlier that, if imperfect, the system will fail on *every* demand—effectively setting $P_{fnp} = 1$), and locating a probability mass C at the $(1, 1)$ point (representing common factors that can affect both parameters) [2] (see also [19], which introduces the ACARP principle: "As Confident As Reasonably Practicable").

The salient point, which we hope is clear even in this simplified presentation, is that possible perfection provides a bridge between the verification activities used to ensure correctness of software and the probabilistic estimates required for failure at the system level. Aleatory uncertainty is captured in a probabilistic model for the system, and epistemic uncertainty concerns the values of the parameters employed in the model. Through conservative assumptions and other modeling techniques, it is generally possible to eliminate dependencies between epistemic assessments of the various parameters. One of the independent parameters will be the probability of imperfection for the software concerned, which represents the assessor's beliefs about the (im)perfection of the software. In the next section, we consider how evidence of formal verification can be used in formulating these beliefs.

III. Formal Verification and Assessment of the Probability of Perfection

We are interested in the extent to which formal methods, and formal verification in particular, can contribute to an evaluator's assignment of a probability of (im)perfection to a body of software. The assignment will take place in the larger context of an overall assessment of the system concerned, which, nowadays, is generally grounded in an argument-based safety or assurance case (see, for example [20]–[24]). Such a case begins with *claims* that enumerate the undesired loss events and the tolerable failure rate or risk associated with them (e.g., "as low as reasonably practicable" (ALARP) [25], or "so unlikely that they are not anticipated to occur during the entire operational life of all airplanes of one type" [3, paragraph 9.e(3)]). *Evidence* about the system and its processes of construction are developed, and an *argument* is constructed to justify satisfaction of the claims, based on the evidence. This process may recurse through subsystems, with substantiated claims about a subsystem being used as evidence in a parent case.

Elsewhere [26], [27], we begin to develop a case that techniques from formal verification can be used to provide mechanized support for the arguments in an assurance case, but here our focus is on use of formal verification to establish claims about a body of software; those claims are assumed to be used as evidence within the larger assurance case.

By formal verification, we mean construction of formal statements for the claims made about the software, formal specification of the software itself, and a proof that the latter achieves the former. The proof is generated or checked by tools that use the techniques of automated deduction, such as theorem proving, including SAT and SMT solving, model checking, and so on [28]. The verification tools may require interactive human guidance, or may be fully automated (e.g., using automated abstraction and invariant discovery), but it is the tools that guarantee the proof.

The formal specifications that are subjected to verification may describe the abstract design of the software and its algorithms, or detailed models taken from a model-based design framework, or the executable code, or all of these. In addition, there may be formalizations of the non-software elements that are part of the system in which the software operates—for example, its sensors and actuators and the physical processes that they monitor and control, the external environment that may interact with the system (e.g., by injecting faults), and any human operators and their mental models [29] and cognitive states. The claims verified may range from simple absence of runtime anomalies, such as those guaranteed by static analysis, to full functional correctness with respect to a detailed requirements specification.

We are interested in the subjective probability of perfection that might be assigned to software that has been formally verified in this way. The assignment will obviously consider the strengths of the guarantees delivered by formal verification and so it is important to examine the main hazards to the soundness of these guarantees.

The first hazard is that the basic requirements or assumptions for the system may have been misunderstood or established incorrectly. This seems to be the dominant source of failure in safety-critical systems: the software is built to the wrong requirements (see, e.g., recent aircraft incident and accident reports [30]–[34]). We anticipate that software that is subjected to formal verification, and which is used in a context for which a probability of perfection is required, will be relatively small and simple and will be for safety monitoring and backup purposes (essentially, to guard against failures of this very type in the mainline operational software). The requirements for software such as this are taken directly from the safety case, so any errors here reflect flaws in the safety case and invalidate far more than the software: they call the entire system and its certification into question. Consequently, we regard this hazard as falling outside the purview of formal verification.

The second hazard definitely is within that purview: it is that the requirements, assumptions, or design may be formalized incorrectly or incompletely. There are three subcases to this concern.

1) Elements of the specification may be inconsistent: this renders the specification meaningless and it becomes possible to prove anything.

   Constructive specifications in a suitable specification framework (e.g., one that requires, among other things, demonstration that all recursive functions are well-founded) are *conservative extensions* of their logic and are therefore always consistent (if the base logic is) [35]. ACL2 [36], Coq [37], HOL [38], and Isabelle [39] are examples of verification systems (i.e., theorem provers for logics that are suitable for system specification and verification tasks) that favor constructive specifications. However, constructive specifications are

not always appropriate; for example, when specifying assumptions, it is generally more appropriate to express them as axioms (we wish to state the assumptions, not implement them). In this case, consistency of the axioms may be ensured by showing that they have a model that is conservative (e.g., defined constructively). The PVS verification system [40], for example, supports this kind of demonstration through theory interpretations [41].

2) Elements of the specification may be just plain wrong: although logically consistent, they do not correctly formalize the requirements, design, or assumptions.

   This is generally the dominant hazard in formal specification and analysis. Formal specifications that have not been subject to some form of mechanized analysis are no more likely to be correct than programs that have never been run (in fact, less so, since nonspecialists generally have better intuition about programs than they do about formal specifications). When a number of unmechanized specifications in the Z language (including some of industrial significance) were subjected to mechanized analysis, most were found to contain flaws [42].

   The most effective ways to ensure that formal specifications capture their intent are to "challenge" them by attempting to prove putative theorems (i.e., "if I've got it right, this ought to follow, but that ought to yield a counterexample"), or to explore the behavior of executable interpretations of the specification: some constructive logics are directly executable, and a large fragment even of PVS's higher order logic with quantifiers can be evaluated efficiently (i.e., at speeds comparable to those of a functional programming language) [43].

   Some verification systems, such as PVS, identify all the axioms and definitions on which a formally verified conclusion depends: if these are correct, then logical validity of the verified conclusion follows by soundness of the verification system. Such identification allows particular scrutiny to be applied to these elements. The axioms and definitions that underpin a verification are generally of two kinds: those that are directly concerned with the subject matter of the system (its requirements, design, assumptions, and so on), and those used in developing the various theories that are required to express this subject matter (e.g., the theories of clocks, synchronous systems, ordinal numbers, and so on). Many of the latter theories will be widely used in other formal developments, so the burden of ensuring their correctness is supported by a broadly shared social process.

   Even if a theory or specification is formalized incorrectly, it does not necessarily invalidate all theorems that use it: only if the verification actually exploits the incorrectness will the validity of the theorem be in doubt (and even then, it could still be true, but unproven). The author has performed many formal verifications and flaws in some of his specifications have been identified by others [44], but in no case did these flaws invalidate the theorems claimed.

3) The formal specification and verification may be discontinuous or incomplete.

   Discontinuities can arise when several analysis tools are applied in the same software development (e.g., a theorem prover, a model checker, a source code static analyzer, and formally-based security and timing analyzers). Concerns are that different tools may ascribe different semantics to the same specification, translations between notations may introduce flaws, and there may be unintended gaps that allow some aspect to escape analysis. There is no simple resolution to these concerns: combinations of specialized analysis tools are outstripping the capabilities and performance of monolithic tools and seem to represent the future of the field. Integrating frameworks such as an "Evidential Tool Bus" [28] suggest one way forward.

   The most significant incompleteness is generally the gap between the most detailed level of specification that is formally analyzed (e.g., algorithms expressed in a functional programming notation, or a model-based design using state machines) and the input (e.g., C code and "make" files) to the software development environment that generates executable code. Manual translation between these notations may introduce faults, and assumptions in the formal development (e.g., interpretation of mathematical functions such as *sqrt*) may be violated by the execution environment (e.g., due to finite precision).

   In an ideal world, the execution environment would itself be provided with a formal specification and strong evidence for correctness, such as that delivered by formal verification. Research projects have accomplished impressive feats in formally verifying such "stacks" of software and hardware [45], but most applications today must rely on informal evidence from extensive and widespread use of their execution platforms, buttressed by testing of their specific configurations.

   A plausible compromise would make formal analysis as comprehensive as reasonably practicable, but also employ comprehensive testing on the execution platform. Automated generation of test cases is a popular application of formal methods [46], [47] and tests generated from the lowest level formal specification (which also serves as the test oracle) can provide evidence that the execution behavior matches this specification.

   Even when formal verification is employed comprehensively, safety-critical software should be thoroughly tested, as this provides an independent "leg" to the assurance case [48] and also probes the assumptions under which the verification was performed. (The "penetration testing" performed on secure systems explicitly targets assumptions used in formal verification, as these are considered the most attractive points of attack.) Automatically-generated tests, derived from a formal

specification, can target specific coverage criteria such as MC/DC (Modified Condition/Decision Coverage) [49], which is required for safety critical software in commercial aircraft (i.e., DO-178B Level A [14]). Any failure discovered in final testing (as opposed to exploratory testing undertaken earlier in development) calls the whole system assurance into question.

The third and final hazard we consider is that a theorem prover, model checker, or other mechanized formal analysis tool employed in the verification may be unsound.

The concern here is that it may prove a false theorem, not that it may fail to prove a true one (that is completeness). Theorem provers are complex programs (often far more complex and sophisticated than the specifications and programs that they verify), and concern about their soundness is generally the dominant concern for nonspecialists ("who will guard the guardians"?).

There are several ways to mechanize theorem proving in support of verification. In one way, a search is performed to find a rule of inference that will move things forward from the current proof state. The search may be massive, but soundness depends only on correct application of the selected inference rule (which includes checking that it *is* applicable). Some theorem provers (generally referred to as "LCF-style" [50]) are deliberately designed to have a very small core set of fairly primitive rules; larger sets of more powerful rules can be defined in terms of these, but soundness depends only on the kernel code that implements the core rules. The hope is that a small kernel has a high probability of attaining perfection, and may even be proved correct. Indeed, the kernel of HOL Light, which is about 400 lines of OCAML, has been proven sound by a stronger version of itself [51] (by Gödel's second incompleteness theorem, a sound theorem prover cannot prove its own soundness, so strengthening is needed).

An objection to the LCF-style approach is that it is inefficient: the overhead (which can be exponential) of reducing a big proof step down to invocations of many core rules is always present, even when the prover is being used for the purpose of "exploration" in the early stages of proof development, rather than for final assurance. An alternative approach is for an untrusted theorem prover to generate "proof logs" that can be certified by an independent trusted checker; the overhead of log generation and checking can be turned off when not required. There is a tension between the size of the proof log and the complexity of the trusted checker: traditionally, the checker has been very simple, and the log correspondingly huge and expensive to generate. Recent work suggests the feasibility of more powerful checkers (in effect, small theorem provers), which are themselves formally verified [52]. These checkers can use much more succinct logs (little more than hints), and confidence in their own verification can be based on one-time use of more primitive checkers or on a diversity of checkers or verifiers. Diverse verifiers are available for certain standard classes of verification problems such as those that can be reduced to SAT and SMT solving.

We have enumerated three major hazards to the trustworthiness of the guarantees provided by formal verification. The first of these (incorrect requirements) is not specific to formal approaches (in fact, formal analysis could help reduce this hazard by "challenging" the requirements against putative theorems), and can therefore be removed from this calculation. The second hazard (incorrect or incomplete formalization) has three subcases and the first of these (inconsistency) can be eliminated by suitable technical methods (exhibiting constructive models). The remaining, active, hazards are incorrect and incomplete formalizations (the second and third subcases of the second hazard), and flaws in the verification system itself (the third hazard).

We cannot suggest specific contributions from each of these active hazards to an assessed probability of imperfection, but we can suggest their likely relative significance. Based on personal experience, errors in formalization should be the dominant concern: the verification can be sound and complete but may not mean what we think it means. As described earlier, formalizations should be "challenged" in various ways, such as by proving putative theorems, checking counterexamples to nontheorems, and direct execution. Incompleteness should be the next level of concern. Some formal verification activities are deliberately very incomplete (e.g., static code analysis) and these may be unable to contribute very much at all to a probability of imperfection. But for those verifications that are reasonably comprehensive, it is important to consider what might have "fallen through the cracks" and to weigh the guarantee of formal verification accordingly. Finally, the possibility of flaws in the verification system itself is the least of the three concerns: although all the major verification systems have had some flaws, no false claim has "escaped" and been used in a larger context, to my knowledge.

IV. Conclusions

We reviewed Bev Littlewood's idea that software may be "possibly perfect" and that its probability of (im)perfection may be assessed. Possible perfection is a more plausible claim than either absolute correctness or reliability for the assurance delivered by most software verification and validation activities. Furthermore, the probability of (im)perfection provides a bridge from these correctness-based activities to the probabilistic claims generally employed at the system level.

In [2], we show that the possible imperfection of one channel is conditionally independent of the failures of another and that the overall failure rate of suitable two-channel systems is the product of the separate probabilities of these events. This approach can be applied to primary/backup systems for nuclear safety, and operational/monitor systems for aircraft.

We considered use of evidence from formal verification in assessing a probability of imperfection. We enumerated hazards to the soundness of the guarantees provided by formal verification and concluded that incorrect formalization (of software requirements and specifications, and of assumptions) and incomplete formalization and analysis (so that issues may "fall through the cracks") are the dominant concerns,

with soundness of the verification system a distant third. We suggested ways to reduce each of these hazards.

The hazards we identified are very similar to the concerns raised more than 20 years ago by Fetzer [53] in his jeremiad against formal verification. Most of those working in formal verification were, and remain, unmoved by Fetzer's alarums because they are well aware of these hazards. However, we believe the treatment given here, where apprehension of the hazards influences the assessed probability of imperfection, is the first that supports a measured response: we do not need heroic efforts to eliminate the hazards, nor should we shrug our shoulders; instead we can attempt to reduce the hazards to a level commensurate with the level of risk accepted for the system.

To make these proposals concrete, we may consider a dual-channel system for which an assessor can substantiate a claim of 10^{-4} for probability of failure of the reliable channel. A balanced case suggests we should require similar probability of imperfection for the possibly perfect channel (thereby achieving an overall probability of failure of 10^{-8}). We suggest that the bulk of this "budget" should be divided between the concerns of incorrect formalization and incompleteness of the formal analysis, with a small fraction, say 10^{-5}, allocated to unsoundness of the verification system.

We believe that through sufficiently careful and comprehensive formal challenges, it is indeed feasible and plausible that an assessor can assign a subjective posterior probability of imperfection on the order of 10^{-4} to the formal statements on which a formal verification depends. Through testing and other scrutiny, we believe a similar figure can be assigned to the probability of imperfection due to discontinuities and incompleteness in the formal analysis. And, by use of a verification system with a trusted or verified kernel, or trusted, verified, or diverse checkers, we believe an assessor can assign a posterior probability of 10^{-5} or smaller to the concern that the theorem prover and other components of the mechanized formal verification system may have incorrectly verified the theorems that attest to perfection.

Dual-channel systems are subject to two kinds of failure: the first is where *both* channels fail to operate safely; the second is where one channel activates a safety action (e.g., shuts down a reactor or signals failure of an avionics system) unnecessarily. Probabilities of imperfection on the order of 10^{-3} or 10^{-4} seem adequate for the verified channel in the first case (because its imperfection is conditionally independent of failure of the other channel). The safety consequences of the second kind of failure are generally less severe than the first, so probabilities of imperfection in the range 10^{-3} to 10^{-4} may be adequate here, too. However, other cases may require smaller probabilities of imperfection, such as 10^{-6} or even less (these could also be required some single-channel systems), and it is a topic for investigation and discussion whether such assessments should be considered feasible and credible.

Other topics for further investigation include dialog with assessors and certifiers on application of these ideas in real systems assessments. Concrete attempts to assign specific probabilities of imperfection to real systems will reveal areas of difficulty and doubt, and future research should explore technical means to alleviate these. Formal verification will generally be employed as one leg of a "multi-legged" assurance case and research is needed to see how best an assessed probability of perfection can be combined with evidence from other legs, possibly using Bayesian Belief Nets (BBNs) as examined by Littlewood and Wright [54].

ACKNOWLEDGMENT

This paper is based on joint work with Bev Littlewood of City University. My research was supported by NASA cooperative agreements NNX08AC64A and NNX08AY53A, and by National Science Foundation grant CNS-0720908.

REFERENCES

[1] B. Littlewood, "The use of proof in diversity arguments," *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 1022–1023, Oct. 2000.

[2] B. Littlewood and J. Rushby, *Reasoning about the Reliability of Fault-Tolerant Systems in Which One Component is "Possibly Perfect"*, City University UK and SRI International USA, 2009, in preparation.

[3] *System Design and Analysis*, Federal Aviation Administration, Jun. 21, 1988, advisory Circular 25.1309-1A.

[4] E. Lloyd and W. Tye, *Systematic Safety: Safety Assessment of Aircraft Systems*. London, England: Civil Aviation Authority, 1982, reprinted 1992.

[5] B. Littlewood and L. Strigini, "Validation of ultrahigh dependability for software-based systems," *Communications of the ACM*, pp. 69–80, Nov. 1993.

[6] R. W. Butler and G. B. Finelli, "The infeasibility of experimental quantification of life-critical software reliability," *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 3–12, Jan. 1993.

[7] J. May, G. Hughes, and A. D. Lunn, "Reliability estimation from appropriate testing of plant protection software," *IEE/BCS Software Engineering Journal*, vol. 10, no. 6, pp. 206–218, Nov. 1995.

[8] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. P. J. Kelly, "An experimental evaluation of software redundancy as a strategy for improving reliability," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 692–702, Jul. 1991.

[9] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 96–109, Jan. 1986.

[10] D. E. Eckhardt, Jr. and L. D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1511–1517, Dec. 1985.

[11] B. Littlewood and D. R. Miller, "Conceptual modeling of coincident failures in multiversion software," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1596–1614, Dec. 1989.

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

[13] *Aerospace Recommended Practice (ARP) 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems*, Society of Automotive Engineers, Nov. 1996, also issued as EUROCAE ED-79.

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

[15] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," Issued for information under FAA memorandum ANM-106N:93-20, Aug. 1993.

[16] N. E. Fenton and M. Neil, "A strategy for improving safety related software engineering standards," *IEEE Transactions on Software Engineering*, vol. 24, no. 11, pp. 1002–1013, Nov. 1998.

[17] W. L. Oberkampf and J. C. Helton, "Alternative representations of epistemic uncertainty," *Reliability Engineering and System Safety*, vol. 85, no. 1–3, pp. 1–10, 2004.

[18] A. O'Hagan, C. E. Buck, A. Daneshkhah, J. R. Eiser, P. H. Garthwaite, D. J. Jenkinson, J. E. Oakley, and T. Rakow, *Uncertain Judgements: Eliciting Experts' Probabilities*. Wiley, 2006.

[19] R. E. Bloomfield, B. Littlewood, and D. Wright, "Confidence: Its role in dependability cases for risk assessment," in *The International Conference on Dependable Systems and Networks*. Edinburgh, Scotland: IEEE Computer Society, Jun. 2007, pp. 338–346.

[20] *Safety Assessment Principles for Nuclear Facilities*, 2006th ed., UK Health and Safety Executive, Bootle, UK, available at http://www.hse.gov.uk/nuclear/saps/saps2006.pdf.

[21] *Licensing of Safety Critical Software for Nuclear Reactors: Common Position of Seven European Nuclear Regulators and Authorised Technical Support Organizations*, AVN Belgium, BfS Germany, CSN Spain, ISTec Germany, NII United Kingdom, SKI Sweden, STUK Finland, 2007, available at http://www.bfs.de/de/kerntechnik/sicherheit/Licensing_safety_critical_software.pdf.

[22] *Air Traffic Services Safety Requirements, CAP 670*, Safety Regulation Group, UK Civil Aviation Authority, Jun. 2008, see Part B, Section 3, Systems Engineering SW01: Regulatory Objectives for Software Safety Assurance in ATS Equipment; Available at http://www.caa.co.uk/docs/33/cap670.pdf.

[23] *Defence Standard 00-56, Issue 4: Safety Management Requirements for Defence Systems. Part 1: Requirements*, UK Ministry of Defence, Jun. 2007, available at http://www.dstan.mod.uk/data/00/056/01000400.pdf.

[24] *Engineering Safety Management (The Yellow Book), Volumes 1 and 2, Fundamentals and Guidance, Issue 4*, Rail Safety and Standards Board, London, UK, 2007, available from http://www.yellowbook-rail.org.uk/site/the_yellow_book/the_yellow_book.html.

[25] *Health and Safety at Work etc. Act*, UK Health and Safety Executive, 1974, available at http://www.hse.gov.uk/legislation/hswa.htm; guidance suite at http://www.hse.gov.uk/risk/theory/alarp.htm.

[26] J. Rushby, "A safety-case approach for certifying adaptive systems," in *AIAA Infotech@Aerospace Conference*. Seattle WA: American Institute of Aeronautics and Astronautics, Apr. 2009, aIAA paper 2009-1992; available at http://www.csl.sri.com/users/rushby/abstracts/aiaa09.

[27] ——, "Runtime certification," in *Eighth Workshop on Runtime Verification: RV08*, ser. Lecture Notes in Computer Science, M. Leucker, Ed., vol. 5289. Budapest, Hungary: Springer-Verlag, Apr. 2008, pp. 21–35.

[28] ——, "Harnessing disruptive innovation in formal verification," in *Fourth International Conference on Software Engineering and Formal Methods (SEFM)*, D. V. Hung and P. Pandya, Eds. Pune, India: IEEE Computer Society, Sep. 2006, pp. 21–28.

[29] ——, "Using model checking to help discover mode confusions and other automation surprises," *Reliability Engineering and System Safety*, vol. 75, no. 2, pp. 167–177, Feb. 2002, available at http://www.csl.sri.com/users/rushby/abstracts/ress02.

[30] *Safety Recommendations A-07-65 though -69*, National Transportation Safety Board, Washington, DC, Oct. 2007, available at http://www.ntsb.gov/recs/letters/2007/A07_65_69.pdf.

[31] *Safety Recommendation A-07-70 though -86*, National Transportation Safety Board, Washington, DC, Oct. 2007, available at http://www.ntsb.gov/Recs/letters/2007/A07_70_86.pdf.

[32] *In-Flight Upset Event, 154 km West of Learmonth, WA, 7 October 2008, VH-QPA Airbus A330-303*, Australian Transport Safety Bureau, Mar. 2009, reference number AO-2008-070 Interim Factual, available at http://www.atsb.gov.au/publications/investigation_reports/2008/AAIR/pdf/AO2008070_interim.pdf.

[33] *In-Flight Upset Event, 240 km North-West of Perth, WA, Boeing Company 777-200, 9M-MRG, 1 August 2005*, Australian Transport Safety Bureau, Mar. 2007, reference number Mar2007/DOTARS 50165, available at http://www.atsb.gov.au/publications/investigation_reports/2005/AAIR/aair200503722.aspx.

[34] *Report on the incident to Airbus A340-642, registration G-VATL en-route from Hong Kong to London Heathrow on 8 February 2005*, UK Air Investigations Branch, 2007, available at http://www.aaib.gov.uk/publications/formal_reports/4_2007_g_vatl.cfm.

[35] J. R. Shoenfield, *Mathematical Logic*. Reading, MA: Addison-Wesley, 1967.

[36] M. Kaufmann and J. S. Moore, "An industrial strength theorem prover for a logic based on Common Lisp," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 203–213, Apr. 1997, aCL2 home page: http://www.cs.utexas.edu/users/moore/acl2/.

[37] C. Cornes, J. Courant, J. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saibi, and B. Werner, "The Coq proof assistant reference manual, version 5.10," INRIA, Rocquencourt, France, Tech. Rep., Feb. 1995, coq home page: http:///coq.inria.fr.

[38] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge, UK: Cambridge University Press, 1993, hOL home page: http://www.cl.cam.ac.uk/Research/HVG/HOL/.

[39] L. C. Paulson, *Isabelle: A Generic Theorem Prover*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1994, vol. 828, isabelle home page: http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[40] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 107–125, Feb. 1995, pVS home page: http://pvs.csl.sri.com.

[41] S. Owre and N. Shankar, "Theory interpretations in PVS," Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep. SRI-CSL-01-01, Apr. 2001.

[42] M. Saaltink, "Domain checking Z specifications," in *LFM' 97: Fourth NASA Langley Formal Methods Workshop*, ser. NASA Conference Publication 3356, C. M. Holloway and K. J. Hayhurst, Eds. Hampton, VA: NASA Langley Research Center, Sep. 1997, pp. 185–192, available at http://atb-www.larc.nasa.gov/Lfm97/proceedings/.

[43] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, "Evaluating, testing, and animating PVS specifications," Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep., Mar. 2001, available from http://www.csl.sri.com/users/rushby/abstracts/attachments.

[44] L. Pike, "A note on inconsistent axioms in Rushby's "Systematic formal verification for fault-tolerant time-triggered algorithms"," *IEEE Transactions on Software Engineering*, vol. 32, no. 5, pp. 347–348, May 2006.

[45] J. S. Moore, "A grand challenge proposal for formal methods: A verified stack," in *Formal Methods at the Crossroads: From Panacea to Foundational Support*, ser. Lecture Notes in Computer Science, vol. 2757. Lisbon, Portugal: Springer-Verlag, 2003, pp. 161–172, 10th Anniversary Colloquium of UNU/IIST the International Institute for Software Technology of The United Nations University.

[46] G. Hamon, L. de Moura, and J. Rushby, "Automated test generation with SAL," Computer Science Laboratory, SRI International, Menlo Park, CA, Technical Note, Sep. 2005, available at http://www.csl.sri.com/users/rushby/abstracts/sal-atg.

[47] M. Utting and B. Legeard, *Practical Model-Based Testing*. Morgan Kaufmann, 2006.

[48] 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*. San Francisco, CA: IEEE Computer Society, Jun. 2003, pp. 25–34.

[49] S. Rayadurgam and M. Heimdahl, "Coverage based test-case generation using model checkers," in *Eighth International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*. Washington DC: IEEE Computer Society, Apr. 2001, pp. 83–91.

[50] M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF: A Mechanized Logic of Computation*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1979, vol. 78.

[51] J. Harrison, "Towards self-verification of HOL Light," in *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, ser. Lecture Notes in Computer Science, U. Furbach and N. Shankar, Eds., vol. 4130. Springer, 2006, pp. 177–191. [Online]. Available: http://dx.doi.org/10.1007/11814771_17

[52] N. Shankar, "Trust and automation in verification tools," in *6th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, ser. Lecture Notes in Computer Science, S. S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, Eds., vol. 5311. Springer-Verlag, Oct. 2008, pp. 4–17.

[53] J. H. Fetzer, "Program verification: The very idea," *Communications of the ACM*, vol. 31, no. 9, pp. 1048–1063, Sep. 1988.

[54] 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*, vol. 33, no. 5, pp. 347–365, May 2007.