Formalism in Safety Cases

Appears in *Making Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium*, pp. 3–17. Chris Dale and Tom Anderson eds., Bristol UK, February 2010.

John Rushby

Abstract Suitable formalisms could allow the arguments of a safety case to be checked mechanically. We examine some of the issues in doing so.

1 Introduction

A safety case provides an *argument* that a system is safe to deploy; the notion of "safe" is made precise in suitable *claims* about the system and its context of deployment, and the argument is intended to substantiate these claims, based on *evidence* concerning the system and its design and construction. The approach can be applied recursively, so that substantiated claims about a subsystem can be used as evidence in a parent case. Evaluators examine the case and may certify the system if they are persuaded that the claims are appropriate, the evidence is valid, and the argument is correct.

The safety case approach to safety certification may be contrasted with the standards-based approach, where the applicant is recommended or required to follow certain guidelines and standards. These generally specify the development and assurance processes that should be used, the intermediate artifacts to be produced (requirements, specifications, test plans etc.), the kinds of reviews, tests, and analyses that should be performed, and the documentation that should record all of these.

The intellectual foundations for the two approaches are fundamentally very similar: we can think of the social process that generates guidelines and standards as constructing a generic safety case; documentation of the processes and products for a particular system then constitutes the evidence for an instantiation of this case. The main difference is that the argument

John Rushby
Computer Science Laboratory, SRI International, Menlo Park California USA
e-mail: `Rushby@csl.sri.com`

(and often the claims, too) are *implicit* in the standards-based approach: they presumably inform the internal debate that decides what evidence the standard should require, but are not formulated explicitly, nor recorded.

Although fundamentally similar, the two approaches do have their own advantages and disadvantages. Standards-based approaches generally incorporate much accumulated experience and community wisdom, and they establish a solid "floor" so that systems developed and assured according to their prescriptions are very likely to be adequately safe. On the other hand, standards tend to be slow-moving and conservative, and can be a barrier to innovation in both system design and in methods for assurance. Furthermore, a generic standard may not be well-tuned to the specifics of any given system—so that its application may be excessively onerous in some areas, yet provide insufficient scrutiny in others.

An explicit safety case can be customized very precisely for the specific characteristics of the system concerned, and therefore has the potential to provide stronger assurance for safety than a standards-based approach, and at lower cost (by eliminating unnecessary effort). Safety cases can also be more agile, allowing greater innovation than standards-based methods.

However, some observers express concern over the reliability of judgements about the quality of a safety case, particularly if some of its elements are novel. One experienced practitioner told me that he feared that regimes lacking a strong safety culture would accept almost any safety case, after demonstrating diligence by probing minor details. Of course, true diligence and competence and a strong safety ethic are required in the performance and evaluation of standards-based approaches as well as safety cases, but the social process that generates standards, and the infrastructure and skill base that develops around them, may provide stronger collective support than is available for a solitary safety case. On the other hand, the motivation for introducing safety cases in the first place came from investigations into a number of disasters where traditional approaches were deemed to have failed [14]. Perusal of recent aircraft accident and incident reports (e.g., [1, 27]) certainly erodes complacency about the standards-based approach employed for airborne software [19].[1]

We may conclude that safety cases seem to be the better approach in principle, but that it could be worthwhile to inquire if there might be some systematic processes that could help increase confidence in the soundness of a given case. Now, a safety case is an argument, and the branch of intellectual inquiry that focuses on arguments is *logic*, with *formal* logic allowing the checking—or generation—of certain kinds of arguments to be reduced to calculation, and thereby automated. So, this paper will explore some of the opportunities and challenges in applying formalism to safety cases. It is written from my personal perspective—which is as a practitioner of formal methods—and may not coincide with the views of those with more experience

[1] A recent report finds massive fault with a major safety case [9].

in safety cases. My hope is that it will help develop a dialog between these two bodies of knowledge and experience.

The next section considers the top-level argument of a safety case; this is followed by consideration of lower-level arguments, and then probabilistic arguments. The paper concludes with a summary and suggestions for further research.

2 The Top-Level Argument

The concepts, notations, and tools that have been developed for representing, managing, and inspecting safety cases (e.g., [3, 13]) provide strong support for structuring the argument of a safety case. Nonetheless, the safety case for a real system is a very large object and one wonders how reliably a human reviewer can evaluate such an argument: consider the thought experiment of slightly perturbing a sound case so that it becomes unsound and ask how confident can we be that a human reviewer would detect the flaws in the perturbed case? These concerns are not merely speculative: Greenwell and colleagues found flaws in several cases that they examined [8].

Although a safety case is an argument, it will generally contain elements that are not simple logical deductions: some elements of the argument will be probabilistic, some will enumerate over a set that is imperfectly known (e.g., "*all* hazards are adequately handled"), and others will appeal to expert judgement or historical experience. All of these are likely to require human review. While suggesting that there may be benefits in formalizing elements of a safety case, I do not propose that we should eliminate or replace those elements that may be difficult to formalize. Rather, my proposal is that by formalizing the elements that do lend themselves to this process, we may be able to reduce some of the analysis to mechanized calculation, thereby preserving the precious resource of expert human review for those elements that truly do require it. Furthermore, formalization of some elements may allow the context for human reviews (e.g., assumptions) to be more precisely articulated and checked.

By formalization and calculation, I mean representing elements of the argument in a formal notation that is supported by strong and automated methods of deduction—that is, theorem proving. I do not see good prospects for adoption of formalization in safety cases, nor much value in doing so, unless it is supported by pushbutton automation. Fortunately, I believe the prospects for achieving this are good: the arguments in a safety case are not intricate ones that tax a theorem prover—they are large, but simple.

An important choice for this enterprise is the logical system in which to formalize safety case arguments. Experiments and experience will be needed to make a well-informed decision, but I can suggest some considerations. On the one hand, we should choose a logic and theories that are supported by pushbutton automation, and on the other, we need a choice that is able to express the kinds of arguments used in a safety case. To make this concrete, here is the top level of an argument examined by Holloway [11]:

> "The control system is acceptably safe, given a definition of acceptably safe, because all identified hazards have been eliminated or sufficiently mitigated and the software has been developed to the integrity levels appropriate to the hazards involved."

We can decompose and slightly restructure this into the following elements.

1. We have a `system` in a `context`, and a safety `claim` about these, and the claim is appropriate for that system in that context.
2. There is a set `hset` of hazards, and the members of this set are all the hazards relevant to the claim for the system in its context.
3. The system `handles` all members of the set `hset` of hazards.
   Note: I have restructured the prose argument here: my notion of "handles" includes either elimination or mitigation of each hazard and, for the latter, assurance that the software has been developed to a suitable integrity level. The decomposition into elimination and mitigation-plus-integrity will be performed at a later stage of the argument.
4. Satisfaction of the preceding items is sufficient to ensure that the system is safe in its context.

We can formalize item 1 as

```
appropriate(claim, system, context)
```

where `claim`, `system`, and `context` are uninterpreted constants, and `appropriate` is an uninterpreted predicate. *Uninterpreted* means that no properties are known about these entities (other than that they are distinct from each other), apart from what we might introduce through axioms; this is in contrast to *interpreted* types and predicates (such as `integer`, or `iszero`) whose meaning is built-in to the theories of the logical system concerned. We can informally attach interpretations to the symbols (e.g., `system` means "the system under consideration"), or we can do so formally by supplying axioms or formal *theory interpretations* [24]. If the formal elaborations are done correctly (and part of what a theorem prover does is check that we do do it correctly), then anything we can prove about the uninterpreted constants remains true of their interpretations.

Here, the justification that the particular claim is appropriate presumably rests on precedent, legislation, experience, and judgement, and will be documented suitably. We can introduce an uninterpreted constant `approp_claim_doc` to represent existence of this documentation, and the documentation itself can be *attached* to the constant. Attachments are used quite widely in AI and in formal verification (e.g., [6]), usually to provide a computational interpretation to some term, in which case they are called "semantic attachments." Here, we have "documentation attachments" and a theorem prover could easily be augmented to assemble or cite the documentation that supports a particular chain of deduction. Mere existence of documentation is insufficient, however: the developers, reviewers, or evaluators of the safety case need to record their judgement that it is adequate. We can allow for this by an uninterpreted predicate `good_doc` and the following axiom

```
good_doc(approp_claim_doc)
  IMPLIES appropriate(claim, system, context)
```

The reviewers can indicate their assent by adding `good_doc(approp_claim_doc)` as an axiom; the theorem prover will then derive `appropriate(claim, system, context)` by forward chaining. The triviality of the deduction here does not negate its value: it provides a computationally effective way to record the existence of documentation, the evidence that it supports, and a judgement about its adequacy. By introducing variants to `good_doc`, we can distinguish the developers' judgement from those of the reviewers or evaluators.

We can formalize item 2 in a similar way as

```
hset = allhazards(claim, system, context)
```

where `allhazards` is an uninterpreted function whose informal interpretation is that its value is the set of all hazards to the claim about the system in its context.

Then item 3 becomes

```
FORALL h IN hset: handles(system, h)
```

where `handles` is an uninterpreted predicate whose informal interpretation is that the system successfully eliminates or mitigates the hazard `h`, and `FORALL...IN...` is *universal quantification* (a concept from logic).

Item 4 can be expressed as

```
safe(claim, system, context)
```

where `safe` is an uninterpreted predicate whose informal interpretation is that the system is acceptably safe.

The structure of the top-level argument is then expressed in the following axiom

```
LET hset = allhazards(claim, system, context) IN
 appropriate(claim, system, context)
   AND FORALL h IN hset: handles(system, h)
 IMPLIES safe(claim, system, context)
```

where `AND` and `IMPLIES` are the logical symbols for conjunction and material implication, respectively, and are written in upper case simply to distinguish them from what logicians call the "nonlogical" symbols. The `LET...IN` construction is syntactic sugar that can be eliminated by simply replacing all instances of the left hand side by the right.

This axiom actually expresses one of several general tactics for constructing a safety case: namely, enumerating the hazards and showing that each is handled effectively. This general tactic could be expressed by replacing the constants `claim`, `system`, and `context` by variables (free variables are assumed to be universally quantified). The axiom shown above would then be an instantiation of the general tactic.

The next step in this example is to record the process of hazard identification. This is one of the most important elements of a safety case, and one that depends crucially on human judgement. Although formalization cannot and should not aim to replace this judgement and its supporting processes, it should record them, and lend calculational assistance where feasible. Human judgement in identification of hazards is usually supported by systematic but manual processes such as checklists, HAZOP/guidewords, or functional hazard analysis (FHA). Evidence that *all* hazards have been identified is generally by reference to documentation describing conformance with an accepted process or standard for performing hazard analysis.

In our example, we could express this in the following axiom

```
good_doc(hazard_doc)
  IMPLIES allhazards(claim, system, context) = {: H1, H2, H3 :}
```

where `H1`, `H2`, and `H3`, are the (otherwise undescribed) hazards named by Holloway, `{: ... :}` is the extensional set constructor, and `hazard_doc` is an uninterpreted constant associated with the documentation of the hazard analysis performed. As before, the predicate `good_doc` is used to indicate that human review, and other processes that might be required, concur that the documentation attached to `hazard_doc` does indeed establish that the hazards are just the three identified. We indicate that this "signoff" has been achieved by asserting `good_doc(hazard_doc)` as an axiom.

Observe that we have chosen to use the uninterpreted function `allhazards`, which returns the set of hazards. An alternative would be to quantify over all possible hazards and have a predicate `ishazard` that identifies those that are true hazards. We could then define `allhazards` as follows.

```
allhazards(claim, system, context) =
   { h: hazards | ishazard(h, claim, system, context) }
```

These two approaches seem almost equivalent from a logical point of view, but reflect a different balance between formalism and judgement. As mentioned previously, identification of hazards is one of the most delicate and important judgements required in a safety case, and formalization should be done in a way that respects that judgement. Quantifying over *all* potential hazards and picking those that are true hazards carries the implication that there is some objective, external set of potential hazards—which is not so. In the formalization used here, the "mystery" of hazard identification is hidden inside the `allhazards` function, where it will be described and justified—as it should be—as the application of human judgement, aided by a systematic, but informal process.

We will take this example just one step further. Holloway's description states that hazard analysis determines that hazard `H2` has potentially catastrophic consequences, and that the acceptable probability of such hazards is 1×10^{-6} per year. These can be recorded in the following axioms.

```
good_doc(hazard_doc) IMPLIES severity(H2) = catastrophic

max_prob(catastrophic) = 1/1000000
```

We can then state that a general tactic for mitigating hazards is to use fault tree analysis to show that their maximum probability of occurrence does not exceed that established for their severity level, and that the *integrity level* of the system software is at least that required for the given severity level. We can state this as a generalized axiom (with variables) as follows.

```
mitigate(s, h) =
  fta(s, h) <= max_prob(severity(h))
    AND integrity(s, h) >= sil(severity(h))

mitigate(s, h) IMPLIES handles(s, h)
```

Here, `s` and `h` are variables representing a system and a hazard; `fta` is an uninterpreted function whose value is informally understood to be the probability of hazard `h` in system `s` as determined by fault tree analysis (FTA), `integrity` is an uninterpreted function whose value is the integrity level of the software in `s` with respect to hazard `h`, and `max_prob` and `sil` give the required maximum probability and minimum integrity level for the `severity` level of `h`. Furthermore, we assert that mitigation is an acceptable way to `handle` a hazard.

We will then instantiate these general axioms for the case of our `system` and hazard `H2`, and assert axioms such as the following.

```
sil(catastrophic) = 5

good_doc(H2_fta_doc) IMPLIES fta(system, H2) <= 1/1000000

good_doc(H2_integrity_doc) IMPLIES integrity(system, H2) = 5
```

Here, `H2_fta_doc` is documentation that describes the fault tree analysis performed and justifies the claim that this establishes the given probability; similarly, `H2_integrity_doc` is documentation that justifies the claim that the software satisfies the requirements for integrity level 5 (in some scale).

My purpose in sketching this formalization is simply to identify suitable logics and theories in which to frame it. What has been used in this example so far is first order logic (with set theory), which is undecidable and so cannot be automated in its full generality. However, various fragments of this logic are decidable and have been found to be pragmatically adequate for most purposes. In particular, the unquantified fragment with uninterpreted symbols and equality is decidable. The example does use quantification, but only in elementary ways that are easily automated.

Thus, my conclusion is that to describe safety case arguments, we need a formalism that includes quantification, uninterpreted predicates and constants, set theory, and arithmetic—but the theorem proving needs pushbutton automation only for the unquantified case. These capabilities are a (subset) of the capabilities of formalisms built on, or employing, SMT solvers (i.e., solvers for the problem of Satisfiability Modulo Theories) [20]. Modern SMT solvers are very effective, often able to solve problems with hundreds of variables and thousands of constraints in seconds. They are the subject of an annual competition, and this has driven very rapid improvement in both their performance and the range of theories over which they operate.

Many specification and modeling formalisms are able to use SMT solvers to provide pushbutton automation. One example is the PVS verification system, which uses the Yices SMT solver (both of these are from SRI [25]). The formalization of the example safety case shown above can be typed into PVS almost verbatim and checked in seconds. PVS is in fact a higher order logic, and this allows a particularly straightforward mechanization of the simple set theory used in the example (sets are predicates). PVS is able to report the axioms actually used in the construction of a proof: for a fuller version of Holloway's example, PVS reports that it uses the top-level tactic of enumeration over hazards (shown above), and the lower-level tactics of eliminating and mitigating hazards (the latter also shown above), plus the axioms associating probabilities and integrity levels with hazard severities (also shown above). PVS also enumerates the `good_doc` axioms required to discharge the claims made in the case: these must justify the appropriateness of the claim, the identification of hazards and their severity, the elimination of the hazard `H1` (by formal verification), and the probability of occurrence (by fault tree analysis) of hazards `H2`, and `H3`, and the integrity level of associated software.

3 Lower-Level Arguments

Our formalization of Holloway's example safety case involves only the most abstract treatment of the system itself. Lower levels of the case, however, will be very much concerned with details of its design and implementation, and the assumptions underlying these. Formal verification is a very well-understood application of formal methods to these concerns. In formal verification, we develop detailed formal models of algorithms, designs, or programs, and use theorem proving, model checking, static program analysis, or other methods of automated deduction to show that these have desired properties. Verification systems such as PVS have been used to verify important properties of significant designs (e.g., [18]). However, PVS and its like are general purpose—that is why they can model abstract safety cases—and greater automation in verification of software systems and their designs can be achieved using notations and techniques specialized to these tasks. Tools employing these are generally referred to as "model checkers," even though most are not model checkers in the strict sense used by logicians. A particularly interesting kind of tool in this class is an "infinite bounded model checker," such as the one in the SAL suite developed at SRI [25]. Infinite bounded model checkers make very effective use of SMT solvers and thereby provide very powerful automation.

The models verified by model checkers are usually very detailed and explicit—equivalent to executable programs. However, and this is not widely understood, infinite bounded model checkers can be applied to rather abstract descriptions that use uninterpreted functions to hide detail. This is feasible because the underlying SMT solvers provide effective automation for this theory. Properties can be attached to the uninterpreted functions by means of axioms supplied directly to the SMT solver or, indirectly, by synchronous observers attached to the model supplied to the model checker [22].

The value in applying formal verification to very abstract designs is that this can be used to automate, or provide automated assistance for, some kinds of safety analyses traditionally performed informally. Many of these analyses can be thought of as informal ways to examine all the possible states of a system, to see if any are unsafe or otherwise undesirable. The reachable states of any interesting system are vast, if not infinite, in number. To examine the reachable states in reasonable time using unaided informal reasoning, we group many similar states together (that is abstraction), and consider only those states encountered on paths that are considered likely to exhibit interesting cases. For example, Failure Modes and Effects Analysis (FMEA) explores only those paths that start from a state in which some component has failed; Fault Tree Analysis (FTA) explores paths backwards from an undesired state to see if some combination of events (usually failures) can render it reachable. These analyses are typically applied to very abstract models; this is because they are often performed early in design exploration, before detailed designs have been developed, and because abstraction reduces the search space.

The benefit in applying automation to these activities is that, unlike informal analyses, they can examine *all possible* states and scenarios. Infinite bounded model checkers are particularly suitable for this purpose because they can operate on abstract models (using uninterpreted functions); however, because of the power of the automation available, they may be able to operate on more realistic abstractions than those used informally. Furthermore, like all model checkers, they not only verify true properties, but also provide explicit counterexamples to false ones (cf. a cut set in FTA). The counterexample capability can be exploited for other purposes, such as the generation of test cases [10].

Holloway's example states that hazard `H1` is eliminated by formal verification, and that the probabilities of hazards `H2` and `H3` are established by FTA. The formalized top-level safety case simply makes reference to the documentation for these, but we can imagine that they could themselves be partially or fully formalized and automated. For example, infinite bounded model checking on a detailed formal model of the system design could verify that `H1` is unreachable, and similar model checking on more abstract models could identify the precipitating events for `H2` and `H3`; separate, informal analysis could then estimate their probability. The following section considers probabilistic arguments in more detail.

4 Probabilistic Arguments

Probability plays an important part in safety cases, quite apart from its use in FTA. Safety is about controlling *risk*, which is the product of the severity of an outcome and its probability, so a significant part of most safety cases is concerned with assessment of probabilities. Estimating the probability of system failure given probabilities for component failures is a well-understood task, with its own methods and tools. The task is more challenging, however, where software is concerned. 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 are determined by the likelihood of encountering circumstances that activate its faults.

For modest values, say down to about 1×10^{-4} probability of failure on demand, it is feasible to measure software failure probabilities by statistically valid random testing [5], where "statistically valid" means that the test case selection probabilities are exactly the same as those that are encountered in real operation. When the required probabilities are smaller than can be verified by direct measurement, the general recourse is to show that the software has been developed to some Software Integrity Level (SIL), as in Holloway's example. However, the practices recommended for most high-level SILs (e.g., DO-178B Level A), such as elaborate documentation of requirements, specifications, and designs, traceability among these, and extensive reviews and testing, are really about ensuring *correctness*, and there is no clear justification for determining a correspondence between SILs and failure probabilities.

In contrast, Littlewood [15] introduced the idea that software may be *possibly perfect* and that we can contemplate its *probability of perfection*. This is attractive because probability of perfection can be interpreted as a subjective assessment of confidence in the verification activities performed on the software. Furthermore, a probability of perfection can be related to reliability, and this has particularly great utility in fault-tolerant systems, where the possible perfection of one "channel" can be shown to be conditionally independent of the reliability of the other; hence, the probability of system failure is the product of these individual probabilities [16].

Using the idea of possible perfection has two ramifications on a safety case. One is that the upper level assessment of the probability of system failure will employ probabilities of software perfection; the other is that the subcase concerned with software must consider the possibility (and probabilities) of its own imperfections. These are likely to be smaller when parts of the case, particularly any verifications and analyses, are formalized and subject to mechanical checking. I suggest considerations for the assessment of these probabilities in a recent paper [23].

Another area where formalization intersects with probability is in assurance for fault-tolerant systems. Many system failures are due to flaws in fault tolerance: the very mechanisms that are intended to prevent failure become the dominant source of failure! Formal verification of these mechanisms produces two very valuable results: first, it requires precise specification of assumed component failure modes, the number of these to be tolerated, and their assumed probabilities; second, it provides convincing evidence (i.e., a proof) that the mechanisms work, provided the number and modes of component failure are consistent with those specified. This bipartite division separates assurance for the correctness of the mechanisms from calculation of system reliability.

The reason that many fault-tolerant systems fail is that their components fail in ways different than assumed in the design of the mechanism for fault tolerance. When the fault-tolerance aspects of the safety case are informal, the failure assumptions may be imprecise, and their probabilities assessed optimistically [12]. Formal verification forces precision in the statement of failure mode assumptions and, thereby, explicit recognition of the cases not tolerated—and realistic assessment of their probability. The latter should drive the design of fault-tolerant mechanisms toward those that make minimal assumptions and are uniformly effective (e.g., Byzantine-resilient algorithms) and away from the special-case treatments that are prevalent in homespun designs.

Even principled designs can benefit form this type of consideration; for example, it is well-known that Byzantine-resilient algorithms that use "signed messages" can tolerate more faults than those that use "oral messages"; but if signatures are flawed for some reason, the signed messages algorithms will fail. Given this information, a developer or assessor can perform principled analysis of the tradeoff between a design that makes fewer assumptions vs. one that tolerates more faults at the cost of more assumptions—or they can be motivated to explore algorithms that combine the best of both choices [7].

Analysis of fault-tolerant systems is one example where appropriate formalization allows the case for correctness to be separated from the case for reliability: formal verification provides assurance that the system does not fail, given assumptions about the failures of components; separately, we estimate the probability of the assumptions, and thereby calculate the reliability of the system. There can be other circumstances in a safety case where logic assures a conclusion, given certain premises, but we are not completely confident in the premises. Our (lack of) confidence in the premises can be represented by attaching a probability to them.

For fault tolerance, calculation of the probability of the conclusion given the probabilities of the premises is very straightforward, but the general case is more difficult—largely because the probabilities on the premises may not be independent. In its general form, this topic enters the domains of probabilistic logic and methods for probabilistic and evidential reasoning, such as Bayesian Belief Nets (BBNs) and Dempster-Shafer theory.

Since safety is about risk, which involves probability, it is quite likely that some of the argument at or near the top level of a safety case will involve probabilistic reasoning of these kinds. For example, we may have evidence for software based on testing and on its integrity level, and we will wish to combine these two "legs" to yield a "multi-legged" case, perhaps using BBNs [17]. A question is whether these probabilistic calculations should be opaque to the formalization, in the way that hazard analysis is, or at least partially represented in the formalization—e.g., by attaching probabilities to formal statements representing uncertain evidence or deductions. There are techniques that combine formal methods with probabilistic calculations, such as probabilistic model checkers, and there are also techniques that use formal methods to estimate probabilities, such as Monte Carlo model counting using SAT solvers. Experimentation is needed to understand how best to meld the logical and probabilistic elements of a safety case, but my own belief is that no matter how it is done, both kinds of analysis must be driven from the same representation of the structure of the case.

5 Summary, and Suggestions for Future Work

I have adumbrated some of the issues in using formalization to represent arguments in a safety case. One benefit of formalization is that it allows use of automated tools to check the logical soundness of the case. Whether this is worthwhile or not depends on whether unsoundness is a significant hazard to real safety cases. My own experience in formal verification is that I have repeatedly been humbled as the theorem prover finds flaws in arguments that I considered either cast iron, or obvious. And in reading even tutorial examples of safety cases, I have been unsettled by the size and diverse tactics of the arguments. Other small examples have been found to employ flawed reasoning [8], but I do not know whether this is a threat in real cases.

Kelly makes a strong argument for a systematic approach to safety case development, and has introduced GSN (Goal Structuring Notation) as a means to facilitate this [14]. In advocating formalization, my aim is not to supplant

GSN or other methods for developing and documenting safety cases in a systematic and reader-friendly manner: rather, it is to provide a means for mechanically checking the logical soundness of cases developed through these or any other methods. Kelly himself describes a semantics for GSN in first order logic that could, in principle, be used to check cases for soundness; the techniques considered in this paper differ only in focusing on methods that can be supported by powerful automation.

Formalization supported by automation brings a further benefit: by assuring us that the overall argument is sound, it allows us to focus on the evidence and assumptions that support the argument. Being able to concentrate on each such item in isolation seems a valuable benefit to me. In addition, some new opportunities become available: for example, the validity of certain kinds of assumptions can be assured by checking or monitoring them at runtime. If the assumptions are formalized, then construction of monitors can be automated by methods developed in the field of *runtime verification* [21]. Reliability of monitored architectures with formal (and possibly perfect) monitors is an interesting topic [16].

Yet another benefit of formalization is that it could allow development of canonical representations for various tactics of argument, and of "metacases" (cases about cases). I think this could be of value in its own right, as it would allow a social process of community review and thereby reduce the vulnerability of intellectually isolated "one-off" cases. Current work at Adelard is exploring these topics.

Using a simple example [11], I illustrated one way to formalize the top-level argument of a simple case in classical logic (I actually used the higher order logic PVS). Basir and colleagues [2] have undertaken a similar exercise using pure first order logic. The example illustrates only one tactic for safety argumentation: namely, enumeration over hazards. The work at Adelard has identified eight different tactics and it remains to be seen whether each of these can be formalized effectively.

Some proponents of safety cases look to Toulmin [26] rather than classical logic in framing cases [3]; Toulmin stresses *justification* rather than *inference*. My opinion is that Toulmin's approach has merit in arguing topics such as aesthetics or morality, where reasonable people can hold different views; but a safety case should be based on agreed evidence about a designed artifact, and here the expectation is that reasonable people must concur on the concluding claim if the argument is sound. Thus, I remain of the opinion that classical logic is adequate for formalizing safety cases, but I do agree that it is worth seeking ways to represent Toulmin's "warrant," "backing," and "rebuttal" within the formalization. The predicate `good_doc` that I used in the example can be seen as a way to link to an extralogical "warrant" for certain steps in the argument. However, the larger goals of Toulmin's approach—namely, justification and persuasion—are better achieved, in my opinion, by using formalization and automation to allow reviewers to explore arguments in an active, dynamic manner—for example, by conducting "what if" experiments.

At the upper levels of a safety case, the system is represented very abstractly, or even indirectly (e.g., by its hazards); at lower levels, there is generally an explicit model of the system and the reasoning is closer to traditional formal verification, or its variants (such as mechanized FMEA). There is obvious benefit if the formalization and reasoning at these levels can be connected in some way. Similarly, we would like a connection between the logical and probabilistic modes of formalization and reasoning. It is not at all clear how to do this, but a *tool bus* may be one way forward, as it does not require all tools to share a common representation [20].

A tool bus or other integration for the different modes and kinds of formalization and reasoning used in safety cases is a good topic for future investigation. Another is the identification, formalization, and analysis of canonical tactics for safety case argumentation. Techniques for developing safety cases in a modular or *compositional* manner would be a breakthrough; the topic of *emergent properties* is particularly interesting in this context [4]. The most important tasks for the future, however, are experiments to determine whether formalization does deliver benefit in the development and assessment of safety cases.

Acknowledgements My research was supported by NASA cooperative agreements NNX08AC64A and NNX08AY53A, and by National Science Foundation grant CNS-0720908. I am grateful to Robin Bloomfield and his colleagues at Adelard and City University for exposing me to some of these topics and sharing their own ongoing investigations. However, the views expressed here are mine alone and do not represent those of my sponsors or collaborators.

References

1. *In-Flight Upset Event, 240 km North-West of Perth, WA, Boeing Company 777-200, 9M-MRG, 1 August 2005.* Australian Transport Safety Bureau, March 2007. Reference number Mar2007/DOTARS 50165, available at `http://www.atsb.gov.au/publications/investigation_reports/2005/AAIR/aair200503722.aspx`.
2. Nurlida Basir, Ewen Denney, and Bernd Fischer. Deriving safety cases from automatically constructed proofs. In *4th IET International Conference on System Safety*, The Institutions of Engineering and Technology, London, UK, October 2009.
3. Peter Bishop, Robin Bloomfield, and Sofia 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.
4. Jennifer Black and Philip Koopman. System safety as an emergent property in composite systems. In *The International Conference on Dependable Systems and Networks*, pages 369–378, IEEE Computer Society, Estoril, Portugal, June 2008.
5. Ricky W. Butler and George B. Finelli. The infeasibility of experimental quantification of life-critical software reliability. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.
6. Judy Crow, Sam Owre, John Rushby, N. Shankar, and Dave Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Sci-

ence Laboratory, SRI International, Menlo Park, CA, March 2001. Available from `http://www.csl.sri.com/users/rushby/abstracts/attachments`.

7. Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In Ravishankar K. Iyer, Michele Morganti, W. Kent Fuchs, and Virgil Gligor, editors, *Dependable Computing for Critical Applications—5*, Volume 10 of IEEE Computer Society *Dependable Computing and Fault Tolerant Systems*, pages 139–157, IEEE Computer Society, Champaign, IL, September 1995.
8. William S. Greenwell, John C. Knight, C. Michael Holloway, and Jacob J. Pease. A taxonomy of fallacies in system safety arguments. In *Proceedings of the 24th International System Safety Conference*, Albuquerque, NM, 2006.
9. Charles Haddon-Cave. The Nimrod Review: An independent review into the broader issues surrounding the loss of the RAF Nimrod MR2 Aircraft XV230 in Afghanistan in 2006. Report, The Stationery Office, London, UK, October 2009. Available at `http://www.official-documents.gov.uk/document/hc0809/hc10/1025/1025.pdf`.
10. Grégoire Hamon, Leonardo de Moura, and John Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods (SEFM)*, pages 261–270, IEEE Computer Society, Beijing, China, September 2004.
11. C. Michael Holloway. Safety case notations: Alternatives for the non-graphically inclined? In *3rd IET International Conference on System Safety*, The Institutions of Engineering and Technology, Birmingham, UK, October 2008.
12. Chris W. Johnson and C. Michael Holloway. Why system safety professionals should read accident reports. In *1st IET International Conference on System Safety*, The Institutions of Engineering and Technology, London, UK, June 2006.
13. T. P. Kelly and R. A. Weaver. The goal structuring notation—a safety argument notation. In *DSN Workshop on Assurance Cases: Best Practices, Possible Obstacles, and Future Opportunities*, Florence, Italy, July 2004.
14. Tim Kelly. *Arguing Safety—A Systematic Approach to Safety Case Management.* PhD thesis, Department of Computer Science, University of York, UK, 1998.
15. Bev Littlewood. The use of proof in diversity arguments. *IEEE Transactions on Software Engineering*, 26(10):1022–1023, October 2000.
16. Bev Littlewood and John 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.
17. Bev Littlewood and David 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. Paul Miner, Alfons Geser, Lee Pike, and Jeffrey Maddalon. A unified fault-tolerance protocol. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 3253 of Springer-Verlag *Lecture Notes in Computer Science*, pages 167–182, Springer-Verlag, Grenoble, France, September 2004.
19. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification.* Requirements and Technical Concepts for Aviation, Washington, DC, December 1992. This document is known as EUROCAE ED-12B in Europe.
20. John Rushby. Harnessing disruptive innovation in formal verification. In Dang Van Hung and Paritosh Pandya, editors, *Fourth International Conference on Software Engineering and Formal Methods (SEFM)*, pages 21–28, IEEE Computer Society, Pune, India, September 2006.
21. John Rushby. Runtime certification. In Martin Leucker, editor, *Eighth Workshop on Runtime Verification: RV08*, Volume 5289 of Springer-Verlag *Lecture Notes in Computer Science*, pages 21–35, Springer-Verlag, Budapest, Hungary, April 2008.
22. John Rushby. A safety-case approach for certifying adaptive systems. In *AIAA Infotech@Aerospace Conference*, American Institute of Aeronautics and Astronautics,

Seattle, WA, April 2009. AIAA paper 2009-1992; available at `http://www.csl.sri.com/users/rushby/abstracts/aiaa09`.

23. John Rushby. Software verification and system assurance. In Dang Van Hung and Padmanabhan Krishnan, editors, *Seventh International Conference on Software Engineering and Formal Methods (SEFM)*, pages 3–10, IEEE Computer Society, Hanoi, Vietnam, November 2009.
24. Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, MA, 1967.
25. *SRI International Formal Methods Program, home page*. `http://fm.csl.sri.com/`.
26. Stephen Edelston Toulmin. *The Uses of Argument*. Cambridge University Press, 2003. Updated edition (the original is dated 1958).
27. *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`.