Invited paper, presented at RV2008. Published in Leucker, M., ed. *Eighth Workshop on Runtime Verification: RV08.* Volume 5289 of Springer-Verlag Lecture Notes in Computer Science, Budapest, Hungary, (April 2008), pp. 21–35.

Runtime Certification[⋆]

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA

Abstract. Software often must be certified for safety, security, or other critical properties. Traditional approaches to certification require the software, its systems context, and all their associated assurance artifacts to be available for scrutiny in their final, completed forms. But modern development practices often postpone the determination of final system configuration from design time to integration time, load time, or even runtime. Adaptive systems go beyond this and modify or synthesize functions at runtime.

Developments such as these require an overhaul to the basic framework for certification, so that some of its responsibilities also may be discharged at integration-, load- or runtime.

We outline a suitable framework, in which the basis for certification is changed from compliance with standards to the construction of explicit goals, evidence, and arguments (generally called an "assurance case"). We describe how runtime verification can be used within this framework, thereby allowing certification partially to be performed at runtime or, more provocatively, enabling "runtime certification."

1 Introduction

Runtime verification, whose technology provides for automated construction of monitors for formally specified properties [1], can be considered from two viewpoints: one sees it as a form of testing, performed as part of pre-deployment verification activities, while the other sees it as a form of post-deployment monitoring. From the latter viewpoint, the ability to generate monitors that guarantee certain properties can be seen as valuable evidence that might be considered in certification.

Traditional approaches to certification are based on adherence to standards or guidelines and do not readily embrace new technologies, such as runtime verification. But other trends, such as the use of adaptive systems for greater resilience, create situations where runtime verification and monitoring could be particularly valuable. Hence, there is increasing interest in alternative approaches to certification that can better exploit new technical opportunities, as well as accommodate new hazards. Within suitable new frameworks, some of the evidence

⋆ This work was supported by National Science Foundation Grant CNS-0720908 and by NASA Cooperative Agreement NNX08AC64A.

required for certification can be achieved by runtime monitoring—by analogy with runtime verification, this approach can, somewhat provocatively, be named "runtime certification."

We do not argue that runtime methods should replace traditional, pre-deployment methods of assurance and certification. Rather, the argument is that traditional methods have become sufficiently effective that accidents seldom occur within the anticipated operating envelope of the system concerned, so that attention has turned to attempting to maintain safe control in unanticipated circumstances, such as those involving major structural damage. Software that attempts to maintain control in these circumstances is necessarily adaptive, and possibly heuristic. The role of runtime verification in these circumstances is first, through assumption monitoring and anomaly detection to contribute to the detection of novel circumstances and, second, to check that any attempted recovery or adaptive control does not violate essential safety properties. It is also possible that technology related to runtime verification can extend its contribution from analysis toward synthesis of safe methods for adaptive control.

More controversially, runtime verification can contribute to detection and recovery from software failure. This is controversial because certified software in a critical system should not fail, and methods for "software fault tolerance" such as n-version programming have not been particularly successful and have fallen into disfavor. However, serious software-induced incidents have been observed in certified critical systems (we describe some involving commercial airplanes later in the paper) and there is concern that these may become more significant as systems become more complex and evolve into systems-of-systems. An example is Next-Generation Air Traffic Control, where the software of different airplanes will interact to operate as a distributed system for maintaining separation without the ground-based supervision employed today. Experience suggests that the primary source of software failure will be violations of assumptions under which it was constructed and certified. Some of these violated assumptions may be due to oversights, some to unanticipated circumstances, and some to "software aging," where software remains constant while the environment in which it operates undergoes change [2].

This paper is organized as follows. The next section outlines an emerging new framework for certification, which we refer to as an "assurance case." The three sections that follow suggest how the framework of an assurance case can guide runtime monitoring for assumptions, anomalies, and safety, respectively. Section 6 considers diagnosis and recovery from failures detected by monitoring, and the final section provides a summary and conclusion.

2 Assurance Cases

Certification is a judgement that deploying a given system in a given context will not pose unacceptable risks of adverse consequences. The intellectual foundation for certification rests on three elements: *claims*, *evidence*, and *argument*. The claims identify the adverse consequences to be considered and the degree of risk considered acceptable; evidence comprises the results of analyses, reviews, and tests; the argument makes the case, based on the evidence, that the claims are satisfied.

The traditional approach to certification may be called "standards based" and largely requires (or strongly recommends) that system development follows prescribed processes (e.g., DO-178B [3] for airborne software) and generates specified evidence (e.g., MC/DC tests [4]). The standards-based approach focuses on evidence: the claims and the argument are largely implicit. Thus, it is not immediately clear whether the evidence from MC/DC testing is intended to support an argument for adequate testing, or one for high-quality requirements, or one for absence of unintended function. Standards-based certification can be very effective in fields where change is relatively slow, so that extensive experience can support the efficacy of the recommended processes and evidence. However, it is less appropriate when rapid innovation leads to systems that are very different to those anticipated in the standard, and it can inhibit the introduction of new assurance methods that provide novel kinds of evidence.

An emerging alternative to standards-based certification is known as a "safety case" [5]. In a safety case, the claims, evidence, and argument for assurance are presented explicitly and are evaluated by the certifying authority or some delegated third party. The exact form of the assurance case is a matter for negotiation by the parties involved, but must generally conform to a given outline (e.g., [6,7]). The advantage of the safety-case approach is that it focuses on the specifics of the system under consideration, and hence can tailor the methods of assurance appropriately (for this reason, it is sometimes referred to as a *goal-based* approach to assurance). The idea that certification should be based on explicit goal-based argumentation began in the UK (following inquiries into several disasters in the petro-chemical industry), and is becoming widely accepted—for example, it is a principal recommendation of a recent report of the National Research Council [8]—and it is now being generalized from safety, so that one hears of "dependability cases," "security cases," and general "assurance cases," which is the term we will use.

Assurance cases are attractive to runtime verification because they not only provide a flexible framework in which we may construct arguments to be discharged by evidence from runtime verification, but the assurance argument can be a source of properties to be monitored at runtime. We explore these topics in the following sections.

3 Runtime Assumption Monitoring

Certification is ultimately a human judgement that might not—or perhaps should not—be reduced to a completely formal or mechanized process. For this reason, some proponents of goal-based assurance look to Toulmin [9] rather than classical logic in framing assurance cases [10]; Toulmin stresses *justification* rather than *inference*. Toulmin's model of argument has the following six elements (from [11]), which are also portrayed in Figure 1.

Claim: This is the expressed opinion or conclusion that the arguer wants accepted by the audience.

Grounds: This is the evidence or data for the claim.

Qualifier: An adverbial phrase indicating the strength of the claim (e.g., certainly, presumably, probably, possibly, etc.).

Warrant: The reasoning or argument (e.g., rules or principles) for connecting the grounds to the claim.

Backing: Further facts or reasoning used to support or legitimate the warrant.

Rebuttal: Circumstances or conditions that cast doubt on the argument; it represents any reservations or "exceptions to the rule" that undermine the reasoning expressed in the warrant or the backing for it.

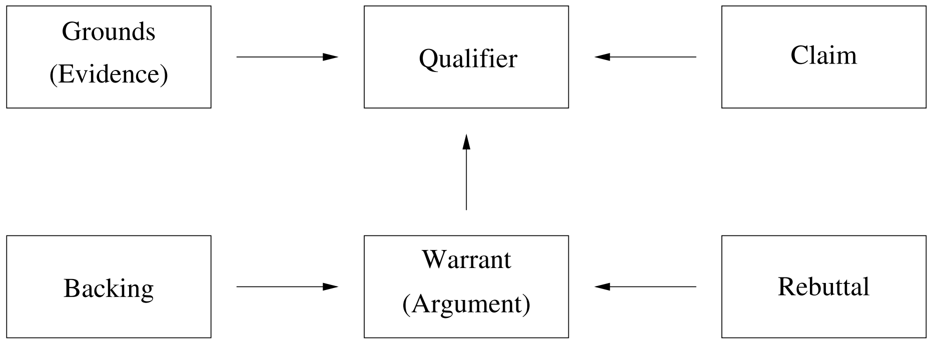

Fig. 1. Toulmin's Model of Argument

The *claim*, *grounds*, and *warrant* of Toulmin's approach correspond to the claim, evidence, and argument of an assurance case. The overall structure will often be hierarchical, with the (sub)claim at one level providing the grounds (evidence) at a higher level. Toulmin's *qualifier*, *backing*, and *rebuttal* find no direct correspondence in an assurance case and, in fact, represent elements in Toulmin's rejection of formal logic.

The case that Toulmin advances against formal logic has some appeal when the topics of discourse are ethics, or aesthetics, say, but it is less persuasive for the topic of certification. There may certainly be areas of doubt in an assurance case, and human judgement and experience may be the appropriate recourse, but these doubts concern our ignorance of the true state of affairs (i.e., facts), rather than genuine judgements (where differences—on aesthetics, for example—cannot be resolved by facts, and reasonable people may come to different conclusions), so the presence of uncertainty need not lead us to reject formal logic.[1] Furthermore, Toulmin's use of adverbial *qualifiers* ("presumably," "possibly" and so on)

[1] Although even within formal logic there are controversies about the treatment of probabilistic uncertainty in evidential reasoning [12,13].

rather than the "proves" ($\vdash$) or "models" ($\models$) of classical logic precludes use of automated tools such as theorem provers and model checkers.

An alternative to expressing doubts and partial knowledge in the qualifier is to express these as explicit assumptions in the hypotheses to a theorem (i.e., using qualifier "proves"). Thus, the elements of an assurance case could be (mechanically analyzed) theorems of the form

$$A_1, \ldots, A_n, S \vdash R$$

where $A_1, \ldots, A_n$ are the assumptions under which the system or design S satisfies requirements or claim R. Toulmin's *backing* and *rebuttal* and can likewise be represented by further assumptions and by additional case analysis, respectively.

Once we have made assumptions explicit, we can subject them to analysis in the same way as other claims: we can ask whether they can be substantiated by subsidiary arguments and evidence, in what circumstances might they be invalidated (cf. fault-tree analysis), and what might be the consequences if they are false (cf. failure modes and effects analysis). And, of course, we can sometimes check them at runtime. We do not describe here how to generate suitable monitors for runtime assumption verification—we suggest use of existing languages and frameworks for runtime verification, such as EAGLE or RULER [14] and Monitoring-Oriented Programming (MOP) [15]—but we note that one of the strengths of runtime verification is that it provides technology to synthesize monitors automatically from their formal descriptions.

Our central point is that construction of formal arguments in support of assurance cases helps make assumptions explicit—and in a form that makes them available for runtime verification. Runtime detection of an assumption violation is not necessarily a harbinger of imminent system failure or safety violation, for the assumption might not be required for the specific execution scenario in progress (but then the assurance case could have been refined by a sharper analysis). Suitable responses to an assumption violation could range from merely logging it and waiting for a more definite indication of trouble, to proactively initiating some repair or fault recovery activity. We consider these in Sections 5 and 6, respectively, but first we consider other kinds of runtime "early warning" or anomaly detection.

4 Runtime Anomaly Detection

Not all assumptions can be verified at runtime. For example, one of the most serious in-flight incidents due to software occurred to a Boeing 777, registration 9M-MRG, near Perth, Australia, on 1 August 2005 [16]. The air data inertial reference unit (ADIRU) performed a restart in circumstances where two of its accelerometers were faulty—whereas the restart algorithm assumed at most one accelerometer would be faulty. The outcome was a series of wild excursions as the autopilot responded to essentially random inputs from its ADIRU. It is not always feasible to detect faulty components (if it were, fault tolerance would be easy), so direct assumption monitoring would not have been feasible in this case.

An alternative to monitoring assumptions and properties that are explicit in the requirements or in the assurance case is to monitor for properties learned by "experience": that is, we check that the system is behaving "as usual." This idea has its roots in methods for intrusion detection in computer security [17], which were subsequently refined to detect infections by computer viruses. An activated virus causes a program to change its behavior—as does an activated fault or violated assumption; hence, it is plausible that methods for detecting anomalies caused by viruses may also detect manifestations of a developing problem.

Most modern methods for anomaly detection work by constructing a model of the normal behavior of the software, in terms, for example, of the invariants that it maintains, or the execution paths that it follows. A program's execution paths can be represented as a context-free grammar or, more crudely, as a set of digraphs on monitored control points (i.e., the set of all pairs of monitored control points—which are often system calls—that are encountered consecutively). The program is monitored in execution and an anomaly alarm is raised whenever execution departs from the recorded model. Following the lead of Wagner and Dean [18], models are often generated by automated formal analysis: for example, an overapproximation to the set of expected execution paths can be constructed using static analysis, and invariants also can be generated in this way.

However, our context for anomaly detection is rather different than that of computer security, and this makes models constructed by static analysis less useful. In computer security, the context is a program that has been changed by activation of a virus, whereas our context is an unchanged program whose behavior has been changed by violation of an assumption (which we can think of as a bug if the assumption is unrecorded). Thus, in our context, static analysis will generate its models from a program in which the bug or faulty assumption is already present, and monitoring will therefore be ineffective. We need, instead, to generate models from bug-free representations of the program.

One way to do this is to generate models from the behavior of the program during test. Critical software is subjected to very thorough testing (e.g., MC/DC coverage in the case of DO-178B Level A) so that models generated from tests should be very accurate, but they will not include faulty behaviors due to activated bugs or violated assumptions—for if those were to arise in test, they would be detected and fixed. The dynamic analyzer Daikon [19] can synthesize invariants from behavior observed in test, and digraphs or other compact representations of observed control flow can be constructed by monitoring test executions. By these means, we can build models that allow runtime monitoring to detect when software behavior departs from that observed during test. Monitoring execution against control flows encountered during test is related to the "vital coded processor" used in railway signaling [20] and is also suggested in the IEC 61508 standard [21, Part 7, page 159].

Violation of an invariant or control flow derived from tests may indicate a genuine error, or simply an untested scenario. If the latter is considered the more likely, then logging the anomaly, rather than initiating repair or fault recovery, may be the most suitable response. Logs of detected anomalies then provide a way to identify inadequately tested or poorly documented cases, and also provide information for *post-deployment testing* and *cooperative bug isolation* [22].

5 Runtime Safety Monitoring

Runtime monitoring for assumptions and anomalies can give early warning that things may be going wrong, but monitoring requirements and safety properties should provide more definitive indications of trouble or, dually, more assurance that the system is operating safely. However, this expectation must be tempered by consideration of the sources of the monitored properties.

Obvious sources for properties to monitor are the requirements for the software concerned. The problem with this choice is that critical software is developed and assured to exacting standards that provide rather effective guarantees that requirements—particularly low-level requirements—will be satisfied. For example, flight-critical software is generally developed and assured according to the guidelines DO-178B Level A [3]. These demand construction of high- and low-level software requirements and rigorous testing of the code against these requirements; in particular, tests generated from the low-level software requirements must achieve MC/DC coverage on the code [4]. These development and assurance processes seem very effective in producing software that is correct with respect to its requirements. Furthermore, these requirements are generally at the unit level and the correctness of the software is often robust at this level; that is to say, there may be problems present at the system level, but individual software units will still be operating correctly according to their unit-level requirements.

Thus, there is unlikely to be much benefit in monitoring requirements at or below the unit level: not only is critical software generally correct with respect to this level of specification, but larger problems may not be manifested at this level. Instead, we need to monitor properties that more directly relate to the safe functioning of the system, and that are more likely to be violated when problems are present—and this invites the question of how might we obtain such properties.

Certification guidelines such as DO-178B offer rather little support in this enterprise because the goal of assurance for the software development process is to establish that the delivered software exactly matches (i.e., is *correct* with respect to) its requirements, rather than that it is safe. Thus Conmy [23] and Amey and Hilton [24] argue that DO-178B is about software correctness, not system safety ("there is no relation of the software to the system hazards, the developer can only state that the whole box has been tested to level A") and Ankrum and Kromholz [25] find no clear link between desired system properties and many of the evidence artifacts required by DO-178B.

However, the system-level arguments and certification evidence for airplane safety are based on various kinds of system and safety analysis such as hazard analysis, failure modes and effects analysis, and fault tree analysis (e.g., [26,27]), and these penetrate down into subsystems and the top-level requirements for the software. Thus, although it is not couched in these terms, the upper levels of assurance for airplane safety, and possibly other classes of systems, too, already have much in common with the notion of a safety or assurance case, as introduced in Section 2.

Thus, we envisage that with modest amendments to current practices for system development and assurance, it will be feasible to introduce elements of a formal assurance case, and that this will yield explicit safety claims that can be subjected to runtime monitoring. Runtime monitoring for critical properties is not new: the idea of a "reference monitor" for security was introduced in 1972 [28]. Later, Rushby analyzed the general class of properties that can be *guaranteed* by monitoring [29], and this analysis was developed further by Wika and Knight [30] and, for the case of security properties, by Schneider [31]. Reduced to essentials, these analyses demonstrate that only safety (as opposed to liveness [32]) properties can be ensured by monitoring.

In this regard, it is worth recalling another serious in-flight incident due to software. An Airbus A340-642, registration G-VATL, suffered a fuel emergency on 8 February 2005 [33]. The plane was over Europe on a flight from Hong Kong to London when two engines flamed out. The crew found that the tanks supplying those engines were empty and those for the other two engines were very low. They declared an emergency and landed at Amsterdam. The subsequent investigation reported that two Fuel Control Monitoring Computers (FCMCs) are responsible for pumping fuel between the tanks on this type of airplane. The two FCMCs cross-compare and the "healthiest" one drives the outputs to the data bus. In this case, both FCMCs had known faults (but complied with the minimum capabilities required for flight); unfortunately, one of the faults in the one judged healthiest was the inability to drive the data bus. Thus, although it gave correct commands to the fuel pumps (there was plenty of fuel distributed in other tanks), these were never received. Backup systems were not invoked because the FCMCs indicated that not both were failed.

Monitoring low-level requirements for the FCMCs would not detect this problem, since faulty requirements were the root of the problem. At the top level, the failure was a *loss of function*, so that the high-level requirement most directly violated was a liveness property: one that says "something good"—i.e., pumping fuel—eventually happens. As noted above, monitoring is effective only for safety properties: ones that say "something bad" does not occur, so it might seem that monitoring would not be effective for this example.

This conundrum is easily solved: most critical systems perform some kind of real-time control function, and a liveness property constrained by a deadline becomes a safety property. For example, "the fuel pumps should activate at least once per hour" is a safety property that can be monitored. In all likelihood, there are many other safety properties suitable for monitoring in this system (e.g., those concerning the acceptable distribution of fuel among the different tanks, or minimum levels in the tanks feeding the engines).

The other classes of major system faults—that is, *malfunction* and *unintended function*—are safety properties and should be suitable for runtime safety monitoring. Other properties that seem suitable for monitoring are interfaces and invariants for distributed algorithms (in the spirit of *interface automata* [34]) and cooperatively maintained data structures (in the spirit of *robust data structures* [35]). Although these may be below the level of properties cited in an assurance case, they do relate to component interactions, and so assurance of their health is a valuable benefit.

6 Diagnosis, Recovery, and Mitigation

Runtime verification for assumptions, anomalies, and safety properties can deliver strong evidence for an assurance case and, ultimately, for certification. As we noted in the introduction, a principal driver for adoption of these techniques is the desire to maintain safe control in the presence of unanticipated events. The idea is that control will be maintained by various "adaptive" methods, and that runtime verification will provide some assurance that these are operating safely. However, runtime verification derives from formal methods, and closely related techniques from this field could provide assured mechanization for some of the "adaptive" tasks—such as diagnosis, and recovery or mitigation of unplanned events—that currently often use ad-hoc methods.

The component whose monitor raises an alarm may not be the source of the fault. Given some symptoms in the form of alarms from software health monitors, *fault diagnosis* is the problem of identifying the source and nature of the fault. Early approaches to fault diagnosis in physical systems used rule-based "expert systems" but these proved fragile and modern methods are based on model-based reasoning "from first principles" [36].

The idea of model-based diagnosis is to perturb a model of the system until the modeled behavior matches that observed. The diagnosis is then derived from the perturbation. Models can range from simple graphs representing connectivity among components to interacting state machines. Models are perturbed by replacing the standard model of a component by one that is faulty; each component is generally provided with a set of fault models (or a single model that can manifest different faults under control of a set of Boolean "switches"), that may range from very specific kinds of fault to a generic "something's wrong," which may be represented by a fully nondeterministic state machine, or communication of a distinguished "bad" data value. The preferred diagnosis is generally one that accounts for the observed symptoms with the smallest number of postulated faults. Calculation of a diagnosis is performed using methods related to model checking, which effectively reduce the problem to one that can be solved using techniques from automated deduction such as SAT or SMT solving. More elaborate diagnostic methods can take probabilities into account, and the underlying methods of deduction then involve Markov decision processes, which can be solved by Monte Carlo methods or by model counting [37].

Much of the research in diagnosis is concerned with the challenge of making exactly the correct identification of the underlying fault. However, although there may be many possible faults, the number of possible reconfigurations or other mitigating actions may be rather few. For the case of jet engines, which were the target of NASA's pioneering Faultfinder system [38], there are just four possible actions: do nothing, reduce power, shut the engine down, or discharge its fire extinguisher. There is no point in performing diagnosis to greater precision than that required to identify the appropriate mitigation. Thus, we propose that diagnosis should be performed in tandem with the search for an appropriate mitigation. Some mitigations (e.g., reconfigurations) can be found through an extension to model-based diagnosis ([39] was the first to propose this), while others require methods akin to AI planning, or program synthesis.

Local mitigations for faults that are attributed to software include retrying a computation, reverting to a checkpointed state, or performing a reset or reboot [40]. Sometimes it will be preferable to adjust input data rather than the system state (cf. *data diversity* [41]): for example, if a sensor sample provokes overflow or division by zero, then we can perturb it slightly, or substitute a previous value (e.g., from a prior iteration in a cyclic control loop). Alternatively, we may be able to reconfigure the system so that a faulty software component is replaced by a diverse alternate. *Recovery blocks* [42] provide a systematic framework for such reconfigurations; alternates may perform graceful degradation rather than exactly reproduce the behavior of the failed primary, and a final alternate may be verified to guarantee some safe minimal functionality (this is *provably safe programming* [43]; a similar idea appears in monitoring-oriented programming and in the work of Sha [44]).

Local mitigations such as those described above require additional implementation mechanism, add complication, and are of uncertain effectiveness. In some cases, the mitigation may be more hazardous than the fault. For example, on 12 May 1997, hard-coded anomaly detection and mitigation caused the display system (EFIS) of American Airlines Flight 903 (an Airbus A300) to go blank: the indicated roll rate of more than 40 degrees/second was considered implausible, and so a bus reset was performed. In fact, the pilots were attempting recovery from a major upset and the roll rate was real; the loss of all instruments at this critical time jeopardized the recovery.

Just as we believe that runtime monitoring will be performed most effectively against properties derived from a system-level assurance case, so we suspect that diagnosis and mitigation will best be performed at the system level also. Safety critical systems such as airplanes already contain massive, well-designed redundancy to protect against anticipated hardware faults, and it will often be possible to invoke this so that safe operation may continue in the presence of unanticipated events or software faults. For example, in the case of the 777 ADIRU problem, we could switch the autopilot to a different source of air data. Even when redundant components have identical designs and are running identical software, their internal state and sensor inputs are likely to differ slightly. Hence, the circumstances that provoke failure in one component (e.g., two faulty accelerometers) may not be present in another, and the same assumptions and the same software that has failed in one component may continue to operate perfectly well in the other.

Diagnosis at the system level may involve a number of steps (e.g., to see if the symptoms persist when various components are reset or shut down) and mitigation may also require several steps rather than a simple reconfiguration. In these cases, we need to synthesize a multi-step *program* of action and the appropriate framework for doing this is supervisory controller synthesis, introduced by Ramadge and Wonham [45]. Controller synthesis can be formulated as a game between the controller and its environment: the controller seeks a strategy to maintain or achieve a given property no matter how the environment behaves. Simple instances, such as certain kinds of AI planning, can be reduced to SAT solving—for example, when the system is deterministic with respect to the inputs and the task is to find a sequence of inputs that places the system in some specific state. In more complicated cases, the controller must really be a strategy that reacts to the environment rather than a simple sequence or a schedule. In this situation, the controller synthesis problem can be solved using techniques derived from model checking [46].

The advantage of a formal, model-based approach to mitigation is that it can consider multiple possible diagnoses and calculate the best overall response. The model can also be cognizant of system-level safety properties, so that we can be sure that an action that seems reasonable at the local level does not have adverse consequences at a higher level (as in the case of American Flight 903). Above all, correctness of the formally synthesized approach is guaranteed, relative to the model. Thus, assurance and certification can focus on the models employed, unlike more heuristic methods whose behavior must be determined experimentally.

It is likely that mitigations undertaken at the system level will require participation by human operators (e.g., to power-cycle a subsystem or to switch to a backup system). In these cases it will be important that the recovery and mitigation procedures communicate effectively with the operators so that they understand the possible states of the system, the available courses of action, and the reasons behind those recommended. One way to do this is to include an explicit representation of the information available to the operators as part of the model that drives the search for diagnosis and mitigations. The feasibility of doing this is supported by [47], which shows how pilots' *mental models* can be represented and used in formal analysis to help avoid mode confusion and other forms of automation surprise, and to guide selection of information presented to the pilots

7 Summary and Conclusion

There is extensive prior work on runtime monitoring for assurance and for error detection and recovery (e.g., [48,49]). The main novelty in the approach proposed here is use of an assurance case as the source of monitored properties.

Runtime monitoring of safety properties related to an assurance case can provide potent evidence to support the case. Such runtime evidence is most useful in adaptive systems that attempt to maintain safe control in unanticipated circumstances that are beyond those considered in the standard design and predeployment certification of the system. Assurance delivered by runtime monitoring can therefore contribute to certification of systems that follow a "never give up" strategy, in the spirit of autonomic and resilient systems [50].

Unanticipated circumstances and violation of assumptions may cause even certified software to fail. Monitoring for assumptions—also derived from an assurance case—and for anomalies—which may be regarded as departures from behaviors encountered in test—can give early warning that problems are at hand, while monitoring for safety properties can give assurance that those problems are being contained or, dually, that they are not and that recovery should be attempted. Formal methods related to runtime verification can provide automated techniques for diagnosis, mitigation, and recovery. These methods for monitoring, analysis, and synthesis are driven by formal models, so their assurance can focus on the models. This may be contrasted with ad-hoc methods, where assurance must often be obtained experimentally.

Complex modern systems, such as airplanes, increasingly incorporate sophisticated functions for sensing, monitoring, and managing the "health" of the system; in airplanes these functions are called *Integrated Vehicle Health Management* (IVHM). We hope the techniques proposed here will contribute to the effectiveness and to the assurance and certification of IVHM systems, and to the emerging field of *software health management*.

Acknowledgment. I appreciate helpful discussions on these topics with Robin Bloomfield and Bev Littlewood of City University, and with my SRI colleagues Bruno Dutertre, Bob Riemenschneider, and Hassen Saïdi.

References

1. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. Software Tools for Technology Transfer **6**(2) (August 2004) 158–173
2. Parnas, D.: Software aging. In: 16th International Conference on Software Engineering, Sorrento, Italy, IEEE Computer Society (May 1994) 279–287
3. Requirements and Technical Concepts for Aviation Washington, DC: DO-178B: Software Considerations in Airborne Systems and Equipment Certification. (December 1992). This document is known as EUROCAE ED-12B in Europe.
4. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. Issued for information under FAA memorandum ANM-106N:93-20 (August 1993)
5. Bishop, P., Bloomfield, R.: A methodology for safety case development. In: Safety-Critical Systems Symposium, Birmingham, UK (February 1998). Available at http://www.adelard.com/resources/papers/pdf/sss98web.pdf.
6. UK Ministry of Defence: Interim Defence Standard 00-56, Issue 3: Safety Management Requirements for Defence Systems. Part 2: Guidance on Establishing a Means of Complying with Part 1. (December 2004). Available at http://www.dstan.mod.uk/data/00/056/02000300.pdf.
7. Safety Regulation Group, UK Civil Aviation Authority: Air Traffic Services Safety Requirements, CAP 670. (2005)

8. Jackson, D., Thomas, M., Millett, L.I., eds.: Software for Dependable Systems: Sufficient Evidence? National Academies Press, Washington, DC (May 2007)
9. Toulmin, S.E.: The Uses of Argument. Cambridge University Press (2003). Updated edition (the original is dated 1958).
10. Bishop, P., Bloomfield, R., Guerra, S.: The future of goal-based assurance cases. In: DSN Workshop on Assurance Cases: Best Practices, Possible Obstacles, and Future Opportunities, Florence, Italy (July 2004)
11. Adelman, L., Lehner, P.E., Cheikes, B.A., Taylor, M.F.: An empirical evaluation of structured argumentation using the Toulmin argument formalism. IEEE Transactions on Systems, Man, and Cybernetics—Part A: Systems and Humans **37**(3) (May 2007) 340–347
12. Fitelson, B.: Studies in Bayesian Confirmation Theory. PhD thesis, Department of Philosophy, University of Wisconsin, Madison (May 2001). Available at http://fitelson.org/thesis.pdf.
13. Joyce, J.M.: On the plurality of probabilist measures of evidential relevance. In: Bayesian Epistemology Workshop of the 26th International Wittgenstein Symposium, Kirchberg, Austria (August 2003). Available at http://www.uni-konstanz.de/ppm/kirchberg/Joyce_1.pdf.
14. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: From EAGLE to RULER. In: Runtime Verification (RV 2007). Volume 4839 of Lecture Notes in Computer Science., Vancouver, British Columbia, Canada, Springer-Verlag (March 2007) 111–125
15. Monitoring-Oriented Programming home page. http://fsl.cs.uiuc.edu/index.php/Monitoring-Oriented_Programming.
16. Australian Transport Safety Bureau: In-flight upset event, 240 km north-west of Perth, WA, Boeing Company 777-200, 9M-MRG, 1 August 2005. (March 2007). Reference number Mar2007/DOTARS 50165, available at http://www.atsb.gov.au/publications/investigation_reports/2005/AAIR/aair200503722.aspx.
17. Denning, D.E.: An intrusion-detection model. IEEE Transactions on Software Engineering **13**(2) (February 1987) 222–232
18. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proceedings of the Symposium on Security and Privacy, Oakland, CA, IEEE Computer Society (May 2001) 156–168
19. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering **27**(2) (February 2001) 99–123. Daikon home page: http://pag.csail.mit.edu/daikon.
20. Chapront, P.: Vital coded processor and safety related software design. In Frey, H.H., ed.: Safety of Computer Control Systems (SAFECOMP '92), Zurich, Switzerland, International Federation of Automatic Control (October 1992) 141–145
21. International Electrotechnical Commission Geneva, Switzerland: IEC 61508—Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. (March 2004). Seven volumes; see http://www.iec.ch/zone/fsafety/fsafety_entry.htm.
22. Liblit, B.: Cooperative Bug Isolation, Winning Thesis of the 2005 ACM Doctoral Dissertation Competition. Volume 4440 of Lecture Notes in Computer Science. Springer-Verlag (May 2007)
23. Conmy, P.: Safety Analysis of Computer Resource Management Software. PhD thesis, Department of Computer Science, University of York, York, UK (2005)
24. Amey, P., Hilton, A.J.: Practical experiences of safety- and security-critical technologies. Ada User Journal **22**(1) (March 2001)

25. Ankrum, T.S., Kromholz, A.H.: Structured assurance cases: Three common standards. In: High-Assurance Systems Engineering Symposium (HASE'05), Heidelberg, Germany, IEEE Computer Society (March 2005) 99–108
26. Society of Automotive Engineers: Aerospace Recommended Practice (ARP) 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems. (November 1996). Also issued as EUROCAE ED-79.
27. Society of Automotive Engineers: Aerospace Recommended Practice (ARP) 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. (December 1996)
28. Anderson, J.P.: Computer security technology planning study. Technical Report ESD-TR-73-51, US Air Force (October 1972). (Two volumes, available at http://seclab.cs.ucdavis.edu/projects/history/seminal.html).
29. Rushby, J.: Kernels for safety? In Anderson, T., ed.: Safe and Secure Computing Systems. Blackwell Scientific Publications (1989) 210–220. (Proceedings of a Symposium held in Glasgow, October 1986).
30. Wika, K.G., Knight, J.C.: On the enforcement of software safety policies. In: COMPASS '95 (Proceedings of the Tenth Annual Conference on Computer Assurance), Gaithersburg, MD, IEEE Washington Section (June 1995) 83–93
31. Schneider, F.: Enforceable security policies. ACM Transactions on Information and System Security **3**(1) (February 2000) 30–50
32. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters **21**(4) (October 1985) 181–185
33. Report on the incident to Airbus A340-642, registration G-VATL en-route from Hong Kong to London Heathrow on 8 February 2005. Report 4/2007, UK Air Investigations Branch (2007). Available at http://www.aaib.gov.uk/publications/formal_reports/4_2007_g_vatl.cfm.
34. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), Association for Computing Machinery (2001) 109–120
35. Taylor, D.J., Morgan, D.E., Black, J.P.: Redundancy in data structures: Improving software fault tolerance. IEEE Transactions on Software Engineering **6**(6) (November/December 1980) 585–594
36. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32** (1987) 57–95
37. Williams, B.C., Ingham, M., Chung, S.H., Elliott, P.H.: Model-based programming of intelligent embedded systems and robotic space explorers. Proceedings of the IEEE **91**(3) (January 2003) 212–237
38. Abbott, K.H., Schutte, P.C., Palmer, M.T., Ricks, W.R.: Faultfinder: A diagnostic expert system with graceful degradation for onboard aircraft applications. In: Proceedings, 14th Symposium on Aircraft Integrated Monitoring Systems, Friedrichshafen, W. Germany (September 1987)
39. Crow, J., Rushby, J.: Model-based reconfiguration: Toward an integration with diagnosis. In: Proceedings, AAAI-91 (Volume 2), Anaheim, CA (July 1991) 836–841
40. Grottke, M., Trivedi, K.: Fighting bugs: Remove, retry, replicate, and rejuvenate. IEEE Computer (February 2007) 107–109
41. Ammann, P.E., Knight, J.C.: Data diversity: An approach to software fault tolerance. IEEE Transactions on Computers **37**(4) (April 1998) 418–425
42. Anderson, T., Kerr, R.: Recovery blocks in action: A system supporting high reliability. In: Proceedings of the 2nd International Conference on Software Engineering, San Francisco, CA, IEEE Computer Society (1976) 447–457

43. Anderson, T., Witty, R.W.: Safe programming. BIT **18** (1978) 1–8
44. Sha, L.: Using simplicity to control complexity. IEEE Software **18**(4) (July 2001) 20–28
45. Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proceedings of the IEEE **77**(1) (January 1989) 81–98
46. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: 16th ACM Symposium on Principles of Programming Languages. (1989) 179–190
47. Rushby, J.: Using model checking to help discover mode confusions and other automation surprises. Reliability Engineering and System Safety **75**(2) (February 2002) 167–177. Available at http://www.csl.sri.com/users/rushby/abstracts/ress02.
48. Bauer, A., Leucker, M., Schallhart, C.: Model-based runtime analysis of distributed reactive systems. In: Proceedings of the Australian Software Engineering Conference (ASWEC'06), Sydney, Australia, The Institute of Electrical and Electronics Engineers (April 2006) 243–252
49. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV (June 1999) 279–287
50. Hollnagel, E., Woods, D.D., Leveson, N., eds.: Resilience Engineering. Ashgate (2005)