How Do We Certify For The Unexpected?

John Rushby*
SRI International, Menlo Park CA USA

By their very nature, loss of control accidents are unanticipated and rare, and their precursors are rare also. Onboard systems to detect and mitigate these precursors must work—and work correctly—when required but must not introduce new malfunctions or unintended functions. How can we provide assurance that software invoked in such rare and unanticipated circumstances is fit for certification?

We argue that software systems such as these are but an extreme example of general trends that undermine much of the standards-based approach to software assurance used in aircraft certification. These trends include component-based software, complex integration, continuous modification, and load- and run-time adaptation. We propose that safety cases based on explicit goals, evidence, and argument provide a firmer foundation for assurance, and a framework within which it is possible to address the rare and the unexpected. Specifically, we propose that just as methods to prevent loss of control move certain software adaptation processes to runtime, so should some of the assurance and verification processes move to runtime also. The paper outlines a technical approach to such "just-in-time certification."

I. Introduction

THE assurance activities that underpin certification strive to consider the behavior of the complete, finished system (e.g., a commercial aircraft) in all its interactions with its environment. The consideration extends to undesired and unintended interactions, such as those involving failures, upsets, and improper operator inputs; the ideal is to consider all behaviors and interactions within a defined "envelope" that is intended to encompass all plausible exigencies, and to establish that the system always remains safe.

We are interested in cases where this ideal breaks down. These may be due to dire circumstances that were not anticipated or were deliberately excluded from the safe envelope (e.g., massive mechanical damage, or improbable upsets), or faults in the design and assurance activities (e.g., a software error). We assume a "never give up" philosophy under which the onboard control software attempts to maintain safe operation in these unanticipated conditions, and to return the system to its safe envelope. By definition, the control software is operating outside its certified envelope in these circumstances, and we are interested in what design and assurance methods might, nonetheless, contribute usefully to an extended notion of certification.

During an unanticipated excursion, the control software is necessarily improvising its behavior at runtime: this is true even if it operates algorithmically, since its application to unanticipated circumstances is an improvisation, but we expect that in general there will be elements of adaptation driven by search and by learning in the software's response to the unanticipated circumstances. Since behavior is synthesized at runtime, it seems that support for assurance and certification must also operate at runtime—in the form of runtime monitors, for example.

Traditional methods for certification provide little guidance on how these runtime extensions to assurance might be organized and justified. We suggest that the emerging framework of a "safety case" provides a better context in which to develop this guidance and that it can contribute directly to the construction of suitable monitors. We describe the idea of a safety case in the following section, and contrast it with traditional methods of certification. In the section after that, we describe how monitors may be derived from a safety case, and in the final section we offer suggestions for more further development of these rather speculative proposals.

*Program Director, Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park CA 94025, Member AIAA

II. Certification and Safety 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[1] for airborne software) and generates specified evidence (e.g., MC/DC tests[2]). 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. Software for commercial aircraft is a paradigmatic example—though even here the recommendations have progressed at roughly ten-year intervals from DO-178A[3] to the current DO-178B[1] and on to DO-178C, which is in development: the progression reflects technological change in methods for software development and assurance, and in the context of its use (notably, the transitions to fly-by-wire and to integrated modular avionics).

Standards-based certification 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. As noted in the introduction, adaptive software that attempts to maintain safe control in unexpected circumstances is an innovation outside the expectations of standards and guidelines such as DO-178B, so we must either force an unnatural fit to the standard, or look elsewhere for guidance.

Adaptive software is not the only innovation that strains current approaches and standards for certification, or whose deployment is impeded by them. The design and construction of modern aircraft are massively outsourced, and this extends to their software. Software from different suppliers is assembled as "integrated modular avionics" (IMA) and we might hope that certification could be assembled in the same way: each software component would be pre-certified for its local function and these would be composed to yield the certification of the aggregate. Recent advisory circulars on reusable software components[4] and guidelines on integrated modular avionics[5] make some provision for taking the certification products for a software component from one aircraft certification into another, but they fall a long way short of endorsing a compositional approach to certification in which components can be separately certified and systems using these need not reexamine their content. Other innovations that pose challenges to traditional certification include product families (where we would like certification of a modified design to require effort commensurate with the extent of the modification), and highly customized configurations of software (so that the exact software instantiation for a particular aircraft may not be determined until boot time). An even larger challenge is posed by the development of systems-of-systems that are larger than a single aircraft. Next-Generation Air Traffic Control, where the software of different aircraft will interact to operate as a distributed system for maintaining separation without the ground-based supervision employed today, is an example. Such systems of systems are not only challenging innovations in themselves, but they expose existing systems to the stresses of "software aging," where software in a system remains constant while the environment in which it operates undergoes change,[6] thereby introducing new hazards that were not considered in the original design.

An emerging alternative to standards-based certification is known as a "safety case".[7] 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 safety case is a matter for negotiation by the parties involved, but must generally conform to a given outline (e.g.,[8,9]). 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 by the National Research Council.[10]

Although it is not couched in these terms, the upper levels of assurance for aircraft safety, already have much in common with the notion of a safety case. The system-level arguments and certification evidence

for aircraft 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.,[11,12]), and these penetrate down into subsystems and the top-level requirements for the software. Below the top level, however, software assurance currently focuses less on safety and more on correctness, as portrayed in Figure 1. Thus Conmy[13] and Amey and Hilton[14] 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[15] find no clear link between desired system properties and many of the evidence artifacts required by DO-178B. However, although DO-178B and other guidelines are often cited in the "Plan for Software Aspects of Certification" they are not mandatory and alternative "means of compliance" may be proposed.

A safety-case approach would likely cause safety analysis to penetrate deeper into the software design and would refocus software assurance on safety rather than correctness. By regarding adaptive software as a system, a safety case would focus on the specific hazards this system may introduce, and their mitigations, not on "correctness" of its learning or adaptive mechanisms, such as neural nets.

A safety case is hierarchical with (sub)claims at one level serving as evidence at a higher level. Dually, the argument that supports a claim will generally be contingent on assumptions that must be discharged as further subclaims. In the following section, we suggest that assumptions and other claims derived from a safety case serve as useful properties to be monitored at runtime.

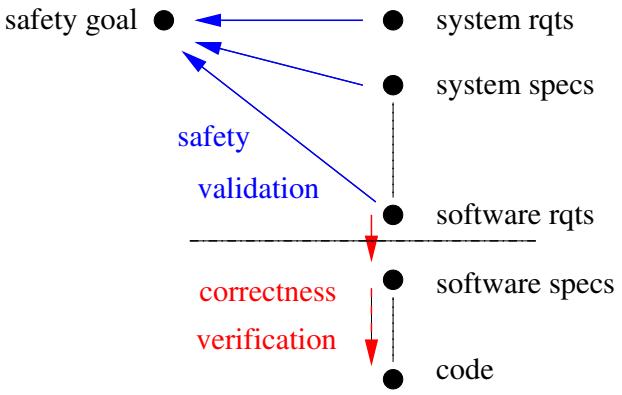

Figure 1. Safety vs. Correctness in Software Certification

III. Runtime Monitoring for the Unexpected

One of the first steps in dealing with the unexpected is to recognize when the system is outside its expected envelope. Runtime monitoring against the assumptions and claims of a safety case is one way to do this. Furthermore, generation of the monitors can sometimes be automated, as we now describe.

The claims and arguments of a safety case will often be expressed in natural language and evaluated by human judgment. But for software there is increasing interest in formal methods, aided by the rapid adoption of model-based design. Model-based design substitutes executable models for many of the later steps of software development that were traditionally undertaken in natural language and documented as low-level requirements and specifications. Models lend themselves to examination by simulation and execution, but these suffer from the same disadvantages as software testing: they examine only a fraction of the possible scenarios. By analyzing models using symbolic methods, it is often possible to explore their properties for *all* scenarios. The benefit of symbolic analysis is analagous to the comparison between the universal demonstration that $(x-y)\times(x+y)=(x^2-y^2)$ and experiments for specific values of x and y. In software analysis, these symbolic approaches are known as *formal methods*, and they include techniques such as static analysis, model checking, and theorem proving. Automation of these techniques has become quite effective of late, as seen in tools such as the Simulink Design Verifier.

There is a particular topic in formal methods known as *Runtime Verification*, whose technology provides methods for automated synthesis of monitors for formally specified properties.[16] Given properties specified in a language such as EAGLE or RULER,[17] the methods of runtime verification can automatically generate an efficient (and provably correct) monitor that will raise an alarm as soon as the property is violated. To make effective use of this capability, we need a suitable source of properties to monitor.

One obvious source for properties is the requirement specification for the software concerned. The problem with this choice is that these requirements are generally at the unit level and the software is often robustly correct at this level (due to the effectiveness of standards-based practices such as those of DO-178B). 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.

A recent in-flight incident illustrates this topic. It concerns an Airbus A340-642, registration G-VATL, which suffered a fuel emergency on 8 February 2005.[18] 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.

This example illustrates that 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.

The claims and assumptions of a safety case can provide exactly these properties. One of the assumptions in a safety case for the FCMCs would likely be that commanding a pump causes it to operate, and a monitor for this property would have detected the malfunction. 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).

Not all assumptions can be verified at runtime. For example, another in-flight incident due to software occurred to a Boeing 777, registration 9M-MRG, near Perth, Australia, on 1 August 2005.[19] 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 might not have been feasible in this case, but monitors for higher-level claims in the safety case (e.g., concerning plausible rates of change in air data) could have identified the arrival of unexpected circumstances.

An alternative to monitoring assumptions and properties that are explicit in the requirements or in the safety 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,[20] 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 represented, crudely but compactly, 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.

One way to generate models is by recording the behavior of the software during test. Flight 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, and not contaminated by 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[21] 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. Such a departure may indicate an unexpected scenario, or simply an untested one. If the latter is considered the more likely, then merely logging the anomaly, rather than initiating more drastic action, may be the most suitable response.

IV. Further Prospects

There is extensive prior work on runtime monitoring for assurance and for error detection and recovery (e.g.,[22,23]). The main novelty in the approach proposed here is use of a safety case as the source of monitored properties.

Runtime monitoring of safety properties related to a safety 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 pre-deployment assurance 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.[24]

Unanticipated circumstances and violation of assumptions may cause even certified software to fail. Monitoring for assumptions—also derived from a safety 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 further recovery should be attempted.

A loss of control event will likely trigger many safety alarms and possibly inappropriate subsystem mitigations taken in ignorance of the system-level problem. 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.

Although we have focussed on runtime monitoring, related techniques from the field of formal methods could also support system-level recovery and the related tasks of diagnosis and mitigation. 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".[25]

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 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 techniques related to those used in formal methods.[26]

Much of the research in diagnosis is concerned with 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 Faultfinder system,[27] 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.[28]

In loss of control events it may be that no component has suffered a fault; rather it may be that some component has been taken outside its assumed operating envelope by a major upset. Although this may not be considered a fault, recovery may best be organized as if it were: diagnose the component whose assumptions have been violated, then seek a reconfiguration that either does not require this assumption, or that has a control function that can reduce and ultimately eliminate the violation. Meanwhile, manage the secondary alarms that are consequences of this one.

Just as we argue that runtime monitoring is performed most effectively against properties derived from a system-level safety case, so we suggest 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. Alternatively, we may be able to switch to backup software (e.g., adaptive control) that does not require the violated assumption.

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.[29] 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 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, can be reduced to AI planning. 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 related to those used in formal methods.[30]

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 the 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. Previous work has shown 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.[31]

In summary, the combination of a safety case with runtime monitors derived from the assumptions and claims of the case, yields a plausible method for certifying adaptive systems that attempt to operate safely in the presence of unexpected events. We hope the techniques outlined here will contribute to the assurance and certification of systems for Integrated Vehicle Health Management or, more provocatively, to "just-in-time certification."

Acknowledgments

Discussions with Kelly Hayhurst and Paul Miner of NASA Langley Research Center, and funding through NASA Cooperative Agreement NNX08AC64A are gratefully acknowledged. I also 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]Requirements and Technical Concepts for Aviation, Washington, DC, *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, Dec. 1992, This document is known as EUROCAE ED-12B in Europe.

[2]Chilenski, J. J. and Miller, S. P., "Applicability of Modified Condition/Decision Coverage to Software Testing," Issued for information under FAA memorandum ANM-106N:93-20, Aug. 1993.

[3]Radio Technical Commission for Aeronautics, Washington, DC, *DO-178A: Software Considerations in Airborne Systems and Equipment Certification*, March 1985.

[4]Federal Aviation Administration, *Reusable Software Components*, Dec. 7, 2004, Advisory Circular 20-148.

[5]Requirements and Technical Concepts for Aviation, Washington, DC, *DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*, Nov. 2005, Also issued as EUROCAE ED-124 (2007).

[6]Parnas, D., "Software Aging," *16th International Conference on Software Engineering*, IEEE Computer Society, Sorrento, Italy, May 1994, pp. 279–287.

[7]Bishop, P. and Bloomfield, R., "A Methodology for Safety Case Development," *Safety-Critical Systems Symposium*, Birmingham, UK, Feb. 1998, Available at `http://www.adelard.com/resources/papers/pdf/sss98web.pdf`.

[8]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*, Dec. 2004, Available at `http://www.dstan.mod.uk/data/00/056/02000300.pdf`.

[9]Safety Regulation Group, UK Civil Aviation Authority, *Air Traffic Services Safety Requirements, CAP 670*, 2005.

[10]Jackson, D., Thomas, M., and Millett, L. I., editors, *Software for Dependable Systems: Sufficient Evidence?*, National Academies Press, Washington, DC, May 2007.

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

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

[13]Conmy, P., *Safety Analysis of Computer Resource Management Software*, Ph.D. thesis, Department of Computer Science, University of York, York, UK, 2005.

[14]Amey, P. and Hilton, A. J., "Practical Experiences of Safety- and Security-Critical Technologies," *Ada User Journal*, Vol. 22, No. 1, March 2001.

[15]Ankrum, T. S. and Kromholz, A. H., "Structured Assurance Cases: Three Common Standards," *High-Assurance Systems Engineering Symposium (HASE'05)*, IEEE Computer Society, Heidelberg, Germany, March 2005, pp. 99–108.

[16]Havelund, K. and Rosu, G., "Efficient Monitoring of Safety Properties," *Software Tools for Technology Transfer*, Vol. 6, No. 2, Aug. 2004, pp. 158–173.

[17]Barringer, H., Rydeheard, D., and Havelund, K., "Rule Systems for Run-Time Monitoring: From EAGLE to RULER," *Runtime Verification (RV 2007)*, Vol. 4839 of *Lecture Notes in Computer Science*, Springer-Verlag, Vancouver, British Columbia, Canada, March 2007, pp. 111–125.

[18]UK Air Investigations Branch, *AAIB Special Bulletin S1/2005: Airbus A340-642, G-VATL*, 2005, Available at `http://www.aaib.dft.gov.uk/cms_resources/G-VATL_Special_Bulletin1.pdf`.

[19]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`.

[20]Denning, D. E., "An Intrusion-Detection Model," *IEEE Transactions on Software Engineering*, Vol. 13, No. 2, Feb. 1987, pp. 222–232.

[21]Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D., "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering*, Vol. 27, No. 2, Feb. 2001, pp. 99–123, Daikon home page: `http://pag.csail.mit.edu/daikon`.

[22]Bauer, A., Leucker, M., and Schallhart, C., "Model-Based Runtime Analysis of Distributed Reactive Systems," *Proceedings of the Australian Software Engineering Conference (ASWEC'06)*, The Institute of Electrical and Electronics Engineers, Sydney, Australia, April 2006, pp. 243–252.

[23]Lee, I., Kannan, S., Kim, M., Sokolsky, O., and Viswanathan, M., "Runtime Assurance Based On Formal Specifications," *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, June 1999, pp. 279–287.

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

[25]Reiter, R., "A Theory of Diagnosis from First Principles," *Artificial Intelligence*, Vol. 32, 1987, pp. 57–95.

[26]Williams, B. C., Ingham, M., Chung, S. H., and Elliott, P. H., "Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers," *Proceedings of the IEEE*, Vol. 91, No. 3, Jan. 2003, pp. 212–237.

[27]Abbott, K. H., Schutte, P. C., Palmer, M. T., and Ricks, W. R., "Faultfinder: A Diagnostic Expert System with Graceful Degradation for Onboard Aircraft Applications," *Proceedings, 14th Symposium on Aircraft Integrated Monitoring Systems*, Friedrichshafen, W. Germany, Sept. 1987.

[28]Crow, J. and Rushby, J., "Model-Based Reconfiguration: Toward an Integration with Diagnosis," *Proceedings, AAAI-91 (Volume 2)*, Anaheim, CA, July 1991, pp. 836–841.

[29]Ramadge, P. J. G. and Wonham, W. M., "The Control of Discrete Event Systems," *Proceedings of the IEEE*, Vol. 77, No. 1, Jan. 1989, pp. 81–98.

[30]Pnueli, A. and Rosner, R., "On the synthesis of a reactive module," *16th ACM Symposium on Principles of Programming Languages*, 1989, pp. 179–190.

[31]Rushby, J., "Using Model Checking to Help Discover Mode Confusions and Other Automation Surprises," *Reliability Engineering and System Safety*, Vol. 75, No. 2, Feb. 2002, pp. 167–177, Available at `http://www.csl.sri.com/users/rushby/abstracts/ress02`.