# Software Engineering for Safety: A Roadmap

Robyn R. Lutz
Jet Propulsion Laboratory

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109-8099

This report describes the current state of software engineering for safety and proposes some directions for needed work that appears to be achievable in the near future.

Categories and Subject Descriptors: D.2 [**Software**]: Software Engineering
General Terms: Software safety, Safety, Roadmap

## 1. INTRODUCTION

Many safety-critical systems rely on software to achieve their purposes. The number of such systems increases as additional capabilities are realized in software. Miniaturization and processing improvements have enabled the spread of safety-critical systems from nuclear and defense applications to domains as diverse as implantable medical devices, traffic control, smart vehicles, and interactive virtual environments. Future technological advances and consumer markets can be expected to produce more safety-critical applications. To meet this demand is a challenge. One of the major findings in a recent report by the President's Information Technology Advisory Committee [1999] was, "The Nation depends on fragile software."

Safety is a system problem [Leveson 1995; McDermid 1996]. Software can contribute to a system's safety or can compromise it by putting the system into a dangerous state. Software engineering of a safety-critical system thus requires a clear understanding of the software's role in, and interactions with, the system. This report describes the current state of software engineering for safety and pro-

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, under a contract with the National Aeronautics and Space Administration. Funding was provided under NASA's Code Q Software Program Center Initiative UPN #323-08.
Address: Department of Computer Science, 226 Atanasoff Hall, Iowa State University, Ames, IA 50011-1041, U.S.A., email rlutz@cs.iastate.edu

poses some directions for needed work in the area.

The next section of the report gives a snapshot of six key areas in state-of-the-art software engineering for safety: *(1) hazard analysis, (2) safety requirements specification and analysis, (3) designing for safety, (4) testing, (5) certification and standards, and (6) resources.* The section provides a overview of the central ideas and accomplishments for each of these topics.

Section 3 of the report describes six directions for future work: *(1) further integration of informal and formal methods, (2) constraints on safe reuse and safe product families, (3) testing and evaluation of safety-critical systems, (4) runtime monitoring, (5) education, and (6) collaboration with related fields.* The criteria used to choose the problems in section 3 are that the problems are important to achieving safety in actual systems (i.e., that people will use the results to build safer systems), that some approaches to solving the problems are indicated in the literature, and that significant progress toward solutions appears feasible in the next decade.

The report concludes with a brief summary of the two central points of the report: (1) that software engineering for safety must continue to exploit advances in other fields of computer science (e.g., formal methods, software architecture) to build safer systems, and (2) that wider use of safety techniques awaits better integration with industrial development environments.

## 2. CURRENT STATE

This section provides a snapshot of the current state in six central areas of software engineering for safety.

### 2.1 Hazard Analysis

Since hazard analysis is at the core of the development of safe systems [Leveson 1995], we begin with a brief discussion of its use and the techniques used to implement it in practice. System-level hazards are states that can lead to an accident. An accident is defined as an unplanned event that results in "death, injury, illness, damage to or loss of property, or environmental harm" [Rushby 1994]. Hazards are identified and analyzed in terms of their criticality (severity of effects) and risk (likelihood of occurrence). The results of the system-level analysis are used to make decisions as to which hazards to address. Some hazards are avoidable, so can be eliminated (e.g., by changing the system design or the environment in which the system operates), while other unacceptable hazards cannot be avoided and must be handled by the system. System safety requirements to handle the unavoidable hazards are then specified.

Further investigation determines which software components can contribute to the existence or prevention of each hazard. Often, techniques such as fault tree analysis, failure modes, effects, and criticality analysis (FMECA), and hazards and operability analysis (HAZOP) are used to help in this determination [DeLemos et al. 1995; Ippolito and Wallace 1995; Leveson 1995; Raheja 1991; Storey 1996; Sullivan et al. 1999]. Combinations of forward analysis methods (to identify the possibly hazardous consequences of failures) and backward analysis methods (to investigate whether the hypothesized failure is credible in the system) have proven especially effective for safety analyses [Maier 1995; McDermid et al. 1995; Lutz

and Woodhouse 1997]. Safety requirements for the software are derived from the resulting descriptions of the software's behavior. These software safety requirements act as constraints on the design of the system. Software may be required to prevent the system from entering a hazardous state (e.g., by mutual exclusion or timeouts), to detect a dangerous state (e.g., an overpressure), or to move the system from a dangerous to a safe state (e.g., by reconfiguration).

The design specification is subsequently analyzed to confirm that it satisfies the safety-related software requirements. During implementation and testing, verification continues to assure that the design is correctly implemented so as to remove or mitigate hazards. The delivered system is validated against the safety-related requirements, with oversight continuing during operations to assure that the requirements were adequate. In practice the hazard analysis is usually iterative with, for example, additional safety requirements being discovered during design or integration testing.

Hazard analyses are also useful for helping prioritize requirements to focus resources (e.g., testing) on the components or features that offer the greatest vulnerability for the system. As we will see below, hazard analyses often guide the choice of which aspects or subsystems merit more intense scrutiny via formal methods.

### 2.2 Safety requirements specification and analysis

Extensive investigation into the specification and analysis of requirements for safety-critical systems has been performed in the last decade. This is especially true in the area of formal methods [Clarke et al. 1996; Rushby 1995]. Formal specification is described by van Lamsweerde elsewhere in this volume, so only highlights of its use for safety-critical systems are given here.

One motivation for specifying requirements formally is that some notations make review, design, implementation, and development of test cases easier and more accurate. Formal documentation of requirements has also been shown to improve the quality of the final product [Courtois and Parnas 1993]. Tabular notations, for example, are familiar to engineers and supported by many tool environments.

Another motivation for specification of requirements in a formal notation is that it allows formal analysis to investigate whether certain safety properties are preserved. For example, Dutertre and Stavridou specify an avionics system and verify such safety requirements as, "If the backup channel is in control and is in a safe state, it will stay in a safe state" [Dutertre and Stavridou 1997]. Automated checks that the requirements are internally consistent and complete (i.e., all data are used, all states are reachable) is often then available. Executable specifications allow the user to exercise the safety requirements to make sure that they match the intent and the reality. Interactive theorem provers can be used to analyze the specifications for desired safety-critical properties. As an example, on one recent spacecraft project there was concern about whether a low-priority fault-recovery routine could be preempted so often by higher-priority fault-recovery routines that it would never complete. Because the requirements were formally specified, it could be demonstrated using an interactive theorem prover that this undesirable situation could, in fact, occur, and remedy it before implementation [Lutz and Ampo 1994]. Model checkers can be used to investigate whether any combination of circumstances represented in the specification can lead the system to enter an undesirable state [Holzmann

1997].

Significant advances have been made in methods for translating system safety requirements to software requirements. Historically, the discontinuity between system and software safety requirements has been a problem. McDermid has criticized the typical safety case for a software-based system in this regard. He notes that too often a safety case first identifies which software components are critical, using classical safety analyses, and then argues that the likelihood of software contributing to a hazard is acceptably low by referring to the development *process* rather than whether the software product satisfies the system safety requirements [McDermid et al. 1995].

SpecTRM, a toolset built by Leveson and colleagues to support the development of embedded systems, was designed to reduce the discontinuity between system and software requirements. It reduces the gap by reflecting how people actually use specifications to think about a complex system. For example, the interface between the user and the controller (e.g., the displays) is explicitly modeled, and startup values (a frequent source of faulty assumptions) automatically default to the safer value of "unknown" [Heimdahl and Leveson 1996; Leveson et al. 1999].

Many of the problems involved in identifying, specifying, and verifying safety requirements are shared by the requirements engineering of non-safety-critical systems [Finkelstein 1994; Zave 1997]. The reader is referred to Nuseibeh's article elsewhere in this volume for further information on these shared issues in requirements engineering.

### 2.3 Designing for Safety

Substantial overlap exists between the design techniques used for safety-critical systems and those used for other critical or high-consequence systems. Rushby [1994] has provided an excellent discussion, excerpted here, of the similarities and differences among the safety engineering, dependability, secure systems, and real-time systems approaches and assurance techniques. A dependable system is one for which reliance may justifiably be placed on certain aspects of the quality of service that it delivers. Dependability is thus concerned primarily with fault tolerance (i.e., providing an acceptable level of service even when faults occur). Safety engineering focuses on the consequences to be avoided and explicitly considers the system context. Sometimes there is no safe alternative to normal service, in which case, the system must be dependable to be safe. Real-time systems typically must be fault-tolerant and often involve timing-dependent behavior that can lead to hazards if it is compromised. Secure systems concentrate on preventing unauthorized disclosure of information, information integrity, and denial of service, and on assuring noninterference (e.g., via a covert channel). As will be discussed in Section 3, some design techniques used to develop secure or survivable systems have applications in safety-critical systems.

In hardware systems, redundancy and diversity are the most common ways to reduce hazards. In software, designing for safety may also involve *preventing hazards* or *detecting and controlling hazards when they occur.* Hazard prevention design includes mechanisms such as hardware lockouts to protect against software errors, lockins, interlocks, watchdog timers, isolation of safety-critical modules, and sanity checks that the software is behaving as expected. Often such checks are assertions

stating either preconditions on the data input (that it is of the required type or in the required range), postconditions on the data output, or invariants that a dangerous state continues to be avoided.

Hazard detection and control includes mechanisms such as fail-safe designs, self-tests, exception-handling, warnings to operators or users, and reconfigurations [Leveson 1995]. Fault-tolerance mechanisms for detecting and correcting known faults in distributed, message-passing systems are well-developed; see, e.g., [Arora and Kulkarni 1998; Gärtner 1999]. Active protection (monitoring and response) often involves additional software.

The following paragraphs describe three obstacles to the goal of designing safe systems.

**Design tradeoffs.** As was mentioned previously, design decisions usually involve tradeoffs between safety and other desirable product attributes. Design methods for fault-tolerance can contribute to safer systems, (e.g., by providing predictable timing behavior), but they can also create additional interactions between components and levels of the system (e.g., to coordinate recovery from a hazardous state), which is undesirable in a safety-critical system [Lutz and Wong 1992]. Furthermore, as Leveson points out, "often the resolution of conflicts between safety constraints and desired functionality involves moral, ethical, legal, financial, and societal decisions; this is usually not a purely technical, optimization decision [Leveson 1991]." As more safety-critical applications are built, commercial and marketing issues such as time-to-market and liability may also become larger factors in design decisions.

**Vulnerability to simple design errors.** We tend to think of the problem of designing for safety as one of managing complexity, but many accidents have simple causes. As an example of a simple error with a large consequence, consider the recent loss of the Mars Climate Orbiter spacecraft [NASA 1999]. The root cause of the accident was a small error, i.e., use of an English measurement where the software required a metric measurement. The defect (type mismatch) was straightforward, well-understood, easy to prevent in design, and easy to catch in testing. However, the sensitivity of the system to this error was very high. Parnas, van Schouwen, and Kwan [1990] point out that in conventional engineering, every design is characterized by a tolerance, such that being within the specified tolerance is adequate. The underlying assumption is that "small errors have small consequences." In software, this is not true. "No useful interpretation of tolerance is known for software." The limits to our ability to develop safe systems is thus related to what is, as far as we know, an innate characteristic of software.

**Limited use of known design techniques** A recent incident provides a double illustration of the point that known, good-practice, design techniques for safe systems are too often ignored. First, in July, 1998, the Aegis missile cruiser, USS Yorktown, was crippled by the entry of a zero into a datafield, causing the database to overflow and crash all LAN consoles and miniature remote terminal units. Protection against such bad data is a known design technique that was not used. Second, the reported, corrective maintenance was not to fix the design, as would be expected, but to retrain the operators "to bypass a bad data field and change the value if such a problem occurs again" [Slabodkin 1998]. It may be that wider use of known, safe-design techniques can be encouraged by quantification of the cost of such failures [Strigini 1994].

### 2.4 Testing

The role of testing is critical both to the development of safe systems and to their certification. A recent book, based on technical reports from a research project in the UK, describes the testing of safety-related software [Gardiner 1998]. Safety requirements generated during system and software hazard analysis are tracked into testing to validate that the as-built system satisfies them. Since safety requirements often describe invariant conditions that must hold in all circumstances, testing often verifies the fault-tolerant aspects of the software. Tests can also demonstrate that the software responds appropriately to some anticipated or envisioned, abnormal situations. Test cases often emphasize boundary conditions (startup, shutdown) or anomalous conditions (failure detection and recovery), since hazards can result from improper handling of these vulnerable states [Weyuker 1996].

**Assumptions about environment.** Unsafe systems can result from incorrect assumptions about the environment in which the system will operate. This is a constant difficulty in developing spacecraft software, for example, since many aspects of the deep-space environment (vibration, radiation, etc.) are imperfectly known prior to operations. Correctly identifying the point at which a hazardous state will be entered and the set of adjustments that will return the system to a safe state is complicated by these environmental uncertainties. Precise environmental modeling is a great asset in developing such systems and in determining realistic, operational test cases [Tsai et al. 1998].

**Assumptions about users.** Similarly, incorrect assumptions about the user or operator of a system can lead to an unsafe system. For example, in testing a ride for a software-generated, virtual reality amusement park, Disney discovered that users were having problems "flying" their magic carpet [Pausch et al. 1996]. Some users felt that they were upside down when they weren't, got dizzy, or even fainted. The software allowed so much freedom in navigating the carpet that users sometimes became disoriented. Significant human factors research tries to establish accurate assumptions and benchmarks for such systems. However, it was in testing that the mismatch with reality was discovered.

**Assumptions about operations.** While it was in the context of spacecraft, not magic carpets, that the following remark was made, it sums up the tight link between testing and use needed for a safe system: "Test like you fly, fly like you test" [Dumas and Walton 1999]. The statement means that "deep knowledge and experience with the application area will be needed to determine the distribution from which the test cases should be drawn" [Parnas et al. 1990]. The statement also means that operations must be constrained by the scope of the tests. The implications of this limit on safe operation for reuse and evolutionary software is discussed below.

It has been proven that testing is not a sufficient condition for a safe system [Butler and Finelli 1993]. It is infeasible to test a safety-critical system enough to quantify its dependability. Littlewood and Wright [1997] have provided a conservative, reliability-based, Bayesian approach to calculate the number of failure-free tests following a failed test. Measuring and modeling software reliability during testing and operations, e.g., through error profiling, is an active research area [Voas and Friedman 1995], although the accuracy and use of reliability growth models

continue to be controversial [Parnas et al. 1990].

### 2.5 Certification and Standards

Certification of software involves assessing it against certain criteria. The problem is that certification criteria for safety-critical systems are both more complicated and less well-defined than for other software. This is of particular concern in light of the growing need for international certification.

There are many standards for the development of safety-critical systems; McDermid mentions 100 in 1996. A recent overview from the perspective of certification of safety-critical systems is [Rodríguez-Dapena 1999]. The author also provides a list of international software safety initiatives with respect to standards. Among the issues discussed is what standards are appropriate for large, safety-critical systems composed of subsystems from different domains (e.g., a remote telemedicine system that uses satellites and medical software). Often such systems contain COTS (Commercial Off The Shelf) components or subsystems, previously certified under different national authorities, that now must be integrated and certified.

There is widespread criticism of current safety standards. Problems include lack of guidance in existing standards, poor integration of software issues with system safety, and the heavy burden of making a safety case for certification. Some of these same concerns are echoed by Fenton and Neil, who critique the "very wide differences of emphasis in specific safety-critical standards." Recommendations include classifying and evaluating standards according to products, processes, and resources, and constructing domain specific standards for products.

### 2.6 Resources

Several good books exist that describe techniques used in software safety engineering [Raheja 1991; Storey 1996]. Leveson [1995] is the standard reference for the field. Another book, by Hermann, focuses on industrial practices and will be released late in 1999.

There are extensive resources for software safety on the web. Bowen's website, "Safety-Critical Systems," provides links to many of these resources, including newsgroups, mailing lists, courses, publications, conferences, the RISKS Forum, and key groups in software safety and related areas in academia, industry, and government [Bowen ]. A recent IEEE video on the subject is "Developing Software for Safety Critical Systems" [Keene 1998].

## 3. DIRECTIONS

This section describes six directions for needed work in software engineering for safety that appear to offer useful results in the near term.

### 3.1 Further integration of informal and formal methods

Work in the following three areas may provide readier access to formal methods for developers of safety-critical systems.

**Automatic translation of informal notations into formal models.** Recent research in software engineering has correctly emphasized closing the gap between the descriptive notations most widely used by software developers and the more formal methods that allow powerful automatic analyses. For example, Rockwell

Avionics used analysis and simulation of a machine-checkable formal model of requirements for flight guidance mode logic to find latent errors, many of them significant. One of the identified directions for future work at the end of the report was that "engineers wanted a greater emphasis on graphical representation" [Miller 1998]. Integrating graphical design analysis tools, such as fault trees, with formal methods can enhance safety analyses. (Fault trees have been formalized as temporal formulas in interval logic [Hansen et al. 1998].) More ambitiously, integration of visual programming environments with formal methods opens up the possibility of improved links between safety requirements and verification of implementation.

Tabular representation is another informal notation that has been widely linked to more formal notations and tools (see, e.g.,[Heimdahl and Leveson 1996]). The push to provide a formal semantics for UML notations and automated translators to formal languages will also support selective use of formal methods by developers [Mikk et al. 1998]. Continued work to support rigorous reasoning about systems initially described with informal notations, and to help demonstrate the consistency between informal and formal models, is needed.

**Lightweight formal methods.** The use of lightweight formal methods on safety-critical systems has obtained good results in several experimental applications, but more work is needed to better understand when it is appropriate. "Lightweight formal methods" refers to automated analysis approaches that involve rapid, low-cost use of formal methods tailored to the immediate needs of a project. This usually means limited modeling, flexible use, building on existing products, highly selective scope, and forgoing the extended capabilities of theorem provers or model checkers. In three case studies involving lightweight applications of formal methods for requirements analysis, the formal methods provided a beneficial addition to existing requirements engineering techniques and helped find important errors that had not been previously identified [Easterbrook et al. 1998]. In another critical application Feather [1998] instead used a database as the underlying reasoning engine for automated consistency analysis. Feather's work is also interesting in that he analyzes test logs, whereas most applications of lightweight formal models so far have been to requirements or design.

There is as yet no consistent methodology for using lightweight formal methods, nor for integrating results from multiple methods. In part this is due to the facts that ready customization to a project's immediate need drives the use of lightweight formal methods and that results to date are primarily case studies. Some consideration of methodological guidelines would be useful, however, both to make these approaches even more lightweight (easy to apply) and to investigate whether reuse of application methods (perhaps within the same domain) has merit. In addition, studies of which lightweight approaches best provide support specifically for safety analyses of evolving requirements, design revisions, and maintenance are needed.

**Integration of previously distinct formal methods.** Different formal methods have different strengths, so having the flexibility to choose the best-suited method for distinct aspects or phases of a system without additional modeling is beneficial. Work has been reported on the integration of theorem provers and model checkers, formal requirements toolsets and theorem provers, high-level languages and automatic verification, and architectural description languages and theorem provers [Heitmeyer et al. 1998; Jagadeesan et al. 1995; Mikk et al. 1998; Owre

et al. 1996; Stavridou 1999]. Clarke et al [1996] warn that the successful integration of methods must both find a suitable style and find a suitable meaning for using the different methods together.

The improved integration of informal and formal methods is significant for software system safety because it lets developers choose to specify or analyze critical software components at a level of rigor they select. Formal methods allow demonstrations prior to coding of crucial elements of the specification, e.g., that key safety properties always hold or that entry to a certain hazardous state always leads to a safe state.

An additional advantage of this integration from the perspective of safety is that many formal methods have been used for both hardware and software specifications. Critical software anomalies often involve misunderstandings about the software/system interface [Lutz 1996]. The use of formal methods may help bridge the gap that often is created between the software and the system developers. Executable specifications, especially those with a front-end that the user can manipulate, allow exploration of assumptions and help elicit latent requirements that may affect safety.

### 3.2 Constraints on safe reuse and safe product families

Two areas in which research in this area is currently needed are safety analysis of product families and safe reuse of COTS software.

**Safety analysis of product families.** With regard to the first direction, the wish-list of the user community is quite ambitious. A recent workshop on product families stated as one of the major goals, "to certify a set of safety-critical systems at once." One of the stated goals of product line architectural analysis was "any analysis that can be performed on the generic aspects that also applies to all derived instances" [Clements and Weiderman 1998]. To even approach these goals, we need a much better understanding of the extent to which systems with similar requirements can reuse requirements analyses. Clearly, it is the minor variations among the systems (requirements, environment, platform) and the interactions between these variations that will be hardest to characterize, formalize, and verify in terms of safety effects. Some initial work by Lutz [2000] with safety-critical product families has identified modeling decisions that can have safety consequences and derived some safety requirements.

**Safe reuse of COTS software.** With regard to the second item, there are two problems. The first is, in McDermid's words, "the need to better understand how to retrospectively assess the COTS product to determine its fitness for a particular application" [Talbert 1998]. He suggests that suppliers may soon provide a certificate that effectively guarantees the behavior of a software component. In addition, the system and the environment (both original and target) need to be understood sufficiently to identify when software is being used outside the "operational envelope" for which it was originally designed and tested [Gardiner 1998].

The second problem is not so much how to confirm that the software does what it should, but how to confirm that it does not do other things as well. The problem of additional, unexpected behavior is an especial concern with safety-related COTS products since there is a need for predictable, limited interactions and dependencies among components [Profeta et al. 1996]. Rushby [1994] suggests that

traditional methods of hierarchical verification via functional refinement may be inadequate and that notions of architectural notions of refinement may provide better verification.

### 3.3 Testing and evaluation of safety-critical systems

This subsection of the paper describes four challenges to improved testing and evaluation of safety-critical systems.

**Requirements-based testing.** Better links are needed between safety requirements and test cases. This entails both tighter integration of testing tools with requirements analysis tools (see, e.g., [Knight and Nakano 1997]), and improved test-case generation for safety-related scenarios.

An additional challenge is to better support evolutionary development that uses exploratory programming as its process model [Sommerville 1996]. Finkelstein [1994] identified as an open problem how to, in an unconventional development process, maintain a link between requirements and the overall system development. Similarly, traditional hazard analyses assume that safety requirements are identified prior to implementation. However, in the actual development of many systems, safety requirements (e.g., constraints, user interfaces) are often derived primarily from testing of prototypes [Berry 1998]. Knowledge of these new safety requirements then needs to propagate in a predictable manner to later testing of the evolving product. Mechanisms for this are currently lacking.

**Evaluation from multiple sources.** Parnas, van Schouwen, and Kwan [1990] stated that "the safety and trustworthiness of the system will rest on a tripod made up of testing, mathematical review, and certification of personnel and process." The importance of combining evidence from multiple sources regarding the safety of a product is undisputed, but how to structure and combine this disparate information is still an open problem [Strigini 1994].

An additional source of evaluation that must be considered is field studies of deployed systems. Field data are important for requirements elicitation for subsequent members of a product family, for the maintenance required to assure safety of an evolving product, and for identification of realistic test scenarios. The following description of a pacemaker demonstrates how integral a field study can be to the safety of a system: "Observing implanting sessions at hospitals showed us that doctors and nurses may come up with numerous scenarios, some of which are difficult to foresee during system design. Unless we carry out a detailed field study at hospitals, we may not be able to identify these scenarios. Missing use scenarios can be disastrous. A problem may go undetected, and the device may fail in the field" [Tsai et al. 1998]. This "product in a process" assessment [Laprie and Littlewood 1992] has not yet been adequately incorporated into the testing and evaluation of safety-critical systems.

**Model consistency.** Mismatches between the actual behavior of a system and the operator's mental model of that behavior are common, especially in complicated systems, and are a contributor to hazardous states (e.g., mode confusion in pilots). Such discrepancies between actual (i.e., required) and expected behavior can be hard to discover in testing. Rushby [1999a] shows that by modeling both the system and the operator's expectation, a mechanized comparison of all possible behaviors of the two systems can be performed via formal models (here, the state exploration

tool Mur$\phi$). Proposed changes to remove the mismatches (e.g., improved displays) can also be run through the model checker to evaluate whether they remedy the problem. Rushby suggests that instruction manuals for operators could be similarly modeled to check their accuracy, and that the number of states required for the mental model might provide a useful measure of the mental load placed on the operator.

**Virtual environments.** The use of virtual environment (VE) simulations to help design, test, and certify safety-critical systems is on the horizon, driven by enthusiasm of industrial users. Methodologies to support the use of VE in testing, as well as standards for tool qualification of VE currently lag the market [Cruz-Neira and Lutz 1999]. The centrality of human factors and the widely varying response of individuals to a particular VE (e.g., some users experience disorientation and nausea) complicate understanding of a VE's fidelity to the actual system. For software engineers, virtual environments offer a powerful means of integration and systems testing. Their safe use in systems needs to be further addressed.

### 3.4 Runtime Monitoring

The use of autonomous software to monitor and respond to operational activity is widespread. Such software can be used to enhance the safety of a system by detecting and recovering from (or masking) hazardous states. This subsection briefly describes needed work to detect faults and to return to a safe state. It also describes work in profiling system usage to enhance safety analyses.

Runtime monitoring is especially well suited to known, expected hazardous conditions. Detection of known faults through runtime monitoring can involve tradeoffs between increased safety on the one hand and increased complexity, decreased availability, and decreased performance on the other hand. As was seen earlier, the basis for these tradeoffs is usually informal and often unconscious. Requirements and architectural analyses are needed that can help designers reason about these decisions.

Detection of unexpected, hazardous scenarios is more difficult. The use of remote agents to compare a system's expected state with its sensed state and request action if the difference is unacceptable offers promise in this field. For example, the remote agent software on the spacecraft Deep Space One searches its on-board models to diagnosis mismatches between expected and actual activities, and to recommend recovery actions [NWU ].

Runtime monitoring to profile usage has been used most widely to guide maintenance or ensure survivability (e.g., against hacker attacks). However, runtime monitoring techniques can also support safety in several ways. Profiling system usage can identify evolving conditions that may threaten the system, deviations from safety requirements, and operational usage that is inconsistent with the safety assumptions. Feather, Fickas, van Lamsweerde, and Ponsard, for example, combine runtime monitoring with goal-based reasoning about requirements (which can include safety requirements) and strategies for reconciling deviations of the runtime behavior from the requirements. Such an approach may be particularly useful for systems with reusable components (see discussion above) or evolvable, self-adapting architectures.

### 3.5 Education

Few courses are currently offered in universities on the software engineering of safety. At the graduate level, the courses are often part of the master's of software engineering curriculum in programs for practitioners. The focus of such courses thus tends to be methodological (e.g., how to perform an FMECA) rather than scientific. As discussed below, many of the advances in software engineering for safety will come from developments in related areas. There is a need for courses in safety that build on prior education in fault tolerance, security, systems engineering, experimental techniques, and specific application domains.

At the undergraduate level, student exposure to safety-critical systems is minimal. Despite extensive media coverage of software hazards (Y2K, transportation and communication disasters, etc.), the notion that one's own software might jeopardize a system, much less a life, is novel to many students. Three partial remedies are as follows: (1) There is a need for case-based learning modules to encourage a systems approach to software safety (along the lines of Pfleeger's use of Ariane 5 as a case study or the Dagstuhl case study in [Abrial et al. 1996]). (2) A textbook on software engineering for safety is needed (currently Storey's is the only textbook with problem sets). (3) Wider use of popular accounts of accidents and their causes (e.g., [Neumann ; Neumann 1995; Peterson 1995; Petrowski 1992]) in software engineering courses will reinforce the notion that software can contribute to hazards.

### 3.6 Collaboration with Related Fields

Progress in software engineering for safety can exploit advances in related fields. This subsection briefly presents problems in related fields whose solutions have potential benefits for safety. The inverse topic, i.e., advances in software engineering for safety that may be useful to other fields, can be inferred from the discussion, but is not explicitly addressed here.

**Security and survivability.** Ties between safety and security have begun to be explored as offering productive ways to reason about and design safe systems. As Berry [1998] noted, "There is a whole repertoire of techniques for identifying and analyzing security threats, and these are very similar in flavor to the techniques used for identifying and analyzing system hazards."

Examples include anomaly-based intrusion detection; noninterference and containment strategies; security kernels; coordinated responses to attacks (faults); and robust, open-source software [Neumann 1998; Rushby 1994]. Sullivan, Knight, Du, and Geist [1999] have recently demonstrated survivability hardening of a legacy information system by a wrapping technique that allows additional control (e.g., for reconfiguration).

**Software architecture.** The relationships between architectural attributes and safety are still largely undefined. Four problems of particular interest are the following: (1) The safety consequences of flexible and adaptable architectures (e.g., using integrated systems for in-flight reconfiguration) [Stavridou 1999]; (2) Evaluation of architectures for safety-critical product families [Gannod and Lutz ]; (3) Partitioning to control hazards enabled by shared resources [Rushby 1999b]; and (4) Architectural solutions to the need for "techniques that augment the robustness

of less robust components [Neumann 1998]." For example, when a safety-critical system is built using legacy subsystems or databases, an operating system with known failure modes, and COTS components from multiple sources, architectural analysis offers an avenue for safety analysis of the integrated system.

**Theoretical computer science.** The report put out by a recent NSF-sponsored Workshop identifies "Safe and Verifiable Software" as one of five areas in which theoretical computer science can help meet the technological challenge [Workshop 1999]. Specifically, advances in model checking, logics of programs, and program-checking techniques can improve the capabilities and performance of formal specification and verification methods.

**Human factors engineering.** Human factors engineering is another area in which both additional research and additional assimilation of existing results are needed. Better understanding of usage patterns, based on field studies, and formal specification of operator's mental models can yield more accurate safety requirements and safer maintenance. One of the ways that we can avoid past mistakes is by cataloging them in such a way that future developers take note. A technique that merits extension to other domains is the list of design features prone to causing operator mode awareness errors [Leveson et al. 1997]. The items in such a list can be included in checklists for design and code inspections, investigated in formal models, or used in test-case generation.

**Other Areas.** Several important areas have been excluded from discussion here due to space limitations. For example, domain-specific designs for fault tolerance can contribute significantly to safe systems. Advances in operating systems (support for real-time safety-critical applications), programming languages (safe subsets of languages, techniques relating programming languages to specification languages and natural languages), and temporal logics (reasoning about critical timing constraints) are other areas important to safety. The reader is referred to [Alur and Henzinger 1991; Cullyer et al. 1991; Gunter et al. 1996; Sifakis 1996] for discussions of these topics.

## 4. CONCLUSION

This report has described the current state of software engineering for safety in several key areas and presented directions for future work to improve these areas. In summary, the future seems to demand (1) continued exploitation of advances in related fields in order to build safer systems, and (2) better integration of safety techniques with industrial development environments.

## REFERENCES

Abrial, J.-R., Borger, E., and Langmaack, H. 1996. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, Volume 1165 of *LCNS*. Springer-Verlag.

Alur, R. and Henzinger, T. A. 1991. Logics and models of real time: a survey. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg Eds., *Real Time: Theory in Practice*, Volume 600 of *LCNS*, pp. 74–106. Springer-Verlag.

Arora, A. and Kulkarni, S. S. 1998. Detectors and correctors: A theory of fault-tolerance components. *IEEE Trans on Software Eng 24*, 1, 63–78.

Berry, D. M. 1998. The safety requirements engineering dilemma. In *Proc of 9th Int Workshop on Software Specification and Design* (1998).

Bowen, J. Safety-critical systems. http://archive.comlab.ox.ac.uk/safety.html.

Butler, R. W. and Finelli, G. B. 1993. The infeasibility of quantifying the reliability of life-critical real-time software. *Trans on Software Eng 19*, 3–12.

Clarke, E. M., Wing, J. M. et al. 1996. Formal methods: State of the art and future directions. *ACM Computing Surveys 28*, 4, 626–643.

Clements, P. C. and Weiderman, N. 1998. Report on 2nd Int Workshop on Development and Evolution of Software Architectures for Product Families. Technical Report 98-SR-003, CMU/SEI.

Courtois, P.-J. and Parnas, D. L. 1993. Documentation for safety critical software. In *Proc IEEE 15th Int Conf on Software Eng* (1993), pp. 315–323.

Cruz-Neira, C. and Lutz, R. R. 1999. Using immersive virtual environments for certification. *IEEE Software 16*, 4, 26–30.

Cullyer, W. J., Goodenough, S. J., and Wichmann, B. A. 1991. The choices of computer languages for use in safety critical systems. *Software Engineering Journal 6*, 51–58.

DeLemos, R., Saeed, A., and Anderson, T. 1995. Analyzing safety requirements for process-control systems. *IEEE Software 12*, 3, 42–53.

Dumas, L. and Walton, A. 1999. Faster, better, cheaper: an institutional view. In *Proc 50th Annual Int Astronautical Congress* (1999).

Dutertre, B. and Stavridou, V. 1997. Formal requirements analysis of an avionics control system. *IEEE Trans on Software Eng 23*, 5, 267–278.

Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., and Hamilton, D. 1998. Experiences using lightweight formal methods for requirements modeling. *IEEE Trans on Software Eng 24*, 1, 4–14.

Feather, M. 1998. Rapid application of lightweight formal methods for consistency analysis. *IEEE Trans on Software Eng 24*, 11, 949–959.

Feather, M. S., Fickas, S., van Lamsweerde, A., and Ponsard, C. 1998. Reconciling systems requirements and runtime behavior. In *Proc 9th IEEE Int Workshop on Software Specification and Design* (1998).

Fenton, N. E. and Neil, M. 1998. A strategy for improving safety related software engineering standards. *IEEE Trans on Software Eng 24*, 11, 1002–1013.

Finkelstein, A. 1994. Requirements engineering: a review and research agenda. In *Proc 1st Asian and Pacific Software Engineering Conference* (1994), pp. 10–19.

Gannod, G. C. and Lutz, R. R. An approach to architectural analysis of product lines. submitted.

Gardiner, S. Ed. 1998. *Testing Safety-Related Software*. Springer-Verlag, London.

Gärtner, F. C. 1999. Fundamentals of fault-tolerant distributed computing. *ACM Computing Surveys 31*, 1, 1–26.

Gunter, C., Mitchell, J., and Notkin, D. 1996. Strategic directions in software engineering and programming languages. *ACM Computing Surveys 28*, 4, 727–737.

Hansen, K., Ravn, A. P., and Stavridou, V. 1998. From safety analysis to software requirements. *IEEE Trans on Software Eng 24*, 7, 573–584.

Heimdahl, M. P. E. and Leveson, N. 1996. Completeness and consistency in hierarchical state-based requirements. *IEEE Trans on Software Eng 22*, 6, 363–377.

Heitmeyer, C., Kirby, J., Labaw, B., Archer, M., and Bharadwaj, R. 1998. Using abstraction and model checking to detect safety violations in requirements specification. *IEEE Trans on Software Eng 24*, 11, 927–949.

Hermann, D. S. 1999. *Software Safety and Reliability*. IEEE Computer Society Press.

Holzmann, G. J. 1997. The model checker Spin. *IEEE Trans on Software Eng 23*, 5, 279–295.

Ippolito, L. M. and Wallace, D. R. 1995. A study on hazard analysis in high integrity software standards and guidelines. Technical Report NISTR 5589, U.S. Dept. of Commerce.

Jagadeesan, L. J., Puchol, C., and Olnhausen, J. E. V. 1995. Safety property verification of Esterel programs and applications to telecommunications software. In *Proc of 7th*

*Int Conf on CAV*, Volume 939 of *LNCS* (1995), pp. 127–140. Springer-Verlag.

Keene, S. J. 1998. Developing software for safety critical systems. IEEE, NTSC ISBN 0-7803-4573-8.

Knight, J. C. and Nakano, L. G. 1997. Software test techniques for system fault-tree analysis. In *Proc of 16th Int Conf on Computer Safety, Reliability, and Security* (1997).

Laprie, J.-C. and Littlewood, B. 1992. Probabilistic assessment of safety-critical software: Why and how? *CACM 35*, 2, 13–21.

Leveson, N. 1991. Software safety in embedded computer systems. *CACM 34*, 2, 35–46.

Leveson, N. 1995. *Safeware*. Addison-Wesley, Reading, MA.

Leveson, N. G., Heimdahl, M. P. E., and Reese, J. D. 1999. Designing specification languages for process control systems: Lessons learned and steps to the future. In *SIGSOFT Foundations of Software Engineering* (1999).

Leveson, N. G., Pinnel, L. D., Sandys, S. D., Koga, S., and Reese, J. D. 1997. Analyzing software specifications for mode confusion potential. In *Proc Workshop on Human Error and System Development* (1997), pp. 132–146.

Littlewood, B. and Wright, D. 1997. Some conservative stopping rules for the operational testing of safety-critical software. *IEEE Trans on Software Eng 23*, 11, 673–683.

Lutz, R. R. 1996. Targeting safety-related errors during software requirements analysis. *Journal of Systems and Software 34*, 223–230.

Lutz, R. R. 2000. Extending the product family approach to support safe reuse. *Journal of Systems and Software*. to appear.

Lutz, R. R. and Ampo, Y. 1994. Experience report: Using formal methods for requirements analysis of critical spacecraft software. In *Proc of 19th Annual Software Engineering Workshop* (1994), pp. 231–248.

Lutz, R. R. and Wong, J. S. K. 1992. Detecting unsafe error recovery schedules. *IEEE Trans on Software Eng 18*, 8, 749–760.

Lutz, R. R. and Woodhouse, R. 1997. Requirements analysis using forward and backward search. *Annals of Software Engineering 3*, 459–475.

Maier, T. 1995. FMEA and FTA to support safe design of embedded software in safety-critical systems. In *Proc CSR 12th Annual Workshop on Safety and Reliability of Software Based Systems* (1995).

McDermid, J. A. 1996. Engineering safety-critical systems. In I. Wand and R. Milner Eds., *Computing Tomorrow, Future Research Directions in Computer Science*, pp. 217–245. Cambridge: Cambridge University Press.

McDermid, J. A., Nicholson, M., Pumfrey, D. J., and Fenelon, P. 1995. Experience with the application of HAZOP to computer-based systems. In *Proc of 10th Annual Conf on Computer Assurance* (1995), pp. 37–48.

Mikk, E., Lakhnech, Y., Siegel, M., and Holzmann, G. J. 1998. Implementing statecharts in Promela/Spin. In *Proc 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques* (1998).

Miller, S. P. 1998. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proc Formal Methods in Software Practice Workshop* (1998), pp. 44–53.

NASA Mars Climate Orbiter Mishap Investigation Board. 1999. Phase I report.

Neumann, P. G. The Risks digest. `http://www.csl.sri.com/~risko/risks.html`.

Neumann, P. G. 1995. *Computer Related Risks*. ACM Press.

Neumann, P. G. 1998. Robust open-source software. *CACM 41*, 2, 128.

Northwestern University's Qualitative Reasoning Group. Welcome to the principles of operations. `http://rax.arc.nasa.gov:80/activities/pofo/docs/index.html`.

Owre, S., Rajan, S., Rushby, J., Shankar, N., and Srivas, M. 1996. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger Eds., *Computer-Aided Verification, CAV '96*, Number 1102 in LNCS (Jul/Aug 1996), pp. 411–414. Springer-Verlag.

Parnas, D. L., van Schouwen, J., and Kwan, S. P. 1990. Evaluation of safety-critical software. *CACM 33*, 6, 636–648.

Pausch, R., Snoddy, J., Taylor, R., Watson, S., and Haseltine, E. 1996. Disney's Aladdin: First steps toward storytelling in virtual reality. In *Proc Siggraph* (1996), pp. 193–203.

Peterson, I. 1995. *Fatal Defect: Chasing Killer Computer Bugs*. Times Books, New York.

Petrowski, H. 1992. *To engineer is human*. Vintage Books, New York.

President's Information Technology Advisory Committee. 1999. *Information Technology Research: Investing in Our Future*.

Profeta, J. A. I., Andrianos, N. P., Yu, B., Johnson, B. W., DeLong, T. A., Guaspari, D., and Jamsek, D. 1996. Safety-critical systems built with COTS. *Computer 29*, 11, 54–60.

Raheja, D. 1991. *Assurance Technologies: principles and practices*. McGraw-Hill.

Rodríguez-Dapena, P. 1999. Software safety certification: A multidomain problem. *IEEE Software 16*, 4, 31–38.

Rushby, J. 1994. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety 43*, 2, 189–214.

Rushby, J. 1995. Formal methods and their role in the certification of critical systems. In R. Shaw Ed., *Safety and Reliability of Software Based Systems*, pp. 1–42. Springer.

Rushby, J. 1999a. Using model checking to help discover mode confusions and other automation surprises. In *Proc 3rd Workshop on Human Error, Safety, and System Development* (1999).

Rushby, J. M. 1999b. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report (March), SRI.

Sifakis, J. 1996. Research directions for formal methods. *ACM Computing Surveys 28*, 4es.

Slabodkin, G. 1998. Software glitches leave navy smart ship dead in the water. `http://www.gcn.com/archives/gcn/1998/July13/cov2.htm`.

Sommerville, I. 1996. *Software Engineering* (5th ed.). Addison-Wesley.

Stavridou, V. 1999. Provably dependent software architectures for adaptable avionics. In *Proc 18th Digital Avionics Systems Conf* (1999).

Storey, N. 1996. *Safety-Critical Computer Systems*. Addison Wesley Longman.

Strigini, L. 1994. Considerations on current research issues in software safety. *Reliability Engineering and System Safety 43*, 177–188.

Sullivan, K., Dugan, J. B., and Coppit, D. 1999. The Galileo fault tree analysis tool. In *Proc 29th Annual IEEE Int Symposium on Fault-Tolerant Computing* (1999).

Sullivan, K., Knight, J. C., Du, X., and Geist, S. 1999. Information survivability control systems. In *Proc of 21st Int Conf Software Engineering* (1999), pp. 184–192.

Talbert, N. 1998. The cost of COTS: An interview with John McDermid. *Computer 31*, 6, 46–52.

Tsai, W.-T., Mojdehbakhsh, R., and Rayadurgam, S. 1998. Capturing safety-critical medical requirements. *Computer 31*, 4, 40–41.

Voas, J. and Friedman, M. 1995. *Software Assessment: Reliability, Safety, Testability*. John Wiley and Sons.

Weyuker, E. J. 1996. Using failure cost information for testing and reliability assessment. *ACM Trans on Software Eng and Methodology 5*, 2, 87–98.

Workshop on Research in Theoretical Computer Science. 1999. Challenges for theory of computing.

Zave, P. 1997. Classification of research efforts in requirements engineering. *ACM Computing Surveys 29*, 4, 315–321.