Safety and Software Intensive Systems: Challenges Old and New

Mats P.E. Heimdahl

Mats Heimdahl earned an M.S. in Computer Science and Engineering from the Royal Institute of Technology in Stockholm, Sweden and a Ph.D. in Information and Computer Science from the University of California at Irvine. He is currently the Director of the University of Minnesota Software Engineering Center (UMSEC) and a Professor of Computer Science and Engineering at the University of Minnesota. He is the recipient of the McKnight Land-Grant Professorship, the McKnight Presidential Fellow award, and the Outstanding Contributions to Post-Baccalaureate, Graduate, and Professional Education award, all at the University of Minnesota. He is also the recipient of the NSF CAREER award.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

Safety and Software Intensive Systems: Challenges Old and New

Mats P.E. Heimdahl
Department of Computer Science and Engineering
University of Minnesota Software Engineering Center (UMSEC)
University of Minnesota, Minneapolis, Minnesota, USA
heimdahl@cs.umn.edu

Abstract

There is an increased use of software in safety-critical systems; a trend that is likely to continue in the future. Although traditional system safety techniques are applicable to software intensive systems, there are new challenges emerging. In this report we will address four issues we believe will pose challenges in the future.

First, the nature of safety is continuing to be widely misunderstood and known system safety techniques are not applied. Second, our ability to demonstrate (certify) that safety requirements have been met is inadequate. Third, modeling and automated tools, for example, code generation and automated testing, are introduced in a hope to increase productivity; this reliance on tools rather than people, however, introduces new and poorly understood problems. Finally, safety-critical systems are increasingly relying on data (configuration data or databases), incorrect data could have catastrophic and widespread consequences.

1. Introduction

Over the last several decades we have seen a continual increase in the use of software in safety-critical systems; software has for some time been an integral part in application domains such as aerospace and avionics, and we are seing in explosive growth in new application domains, for example, medical devices and automobiles. This expanded use of software in critical systems is a trend that inevitably will continue (and most likely accelerate) into the future.

In 1986 Nancy Leveson brought the notion of "software safety" to the broader computer science community and laid the foundation for a research area rich with challenging problems [46]. Nevertheless, Leveson and others have since then repeatedly pointed out that the phrase "software safety" is somewhat of a misnomer since software by itself is not dangerous: software does not have stored energy that can be released to harm persons, and software is not poisonous or radioactive to harm persons or the environment. Safety is a problem in physical systems and software can only contribute to safety (or hazards) in a systems context [47, 48]. Software can, however, create hazards through erroneous control of a system or by misleading the system operators into taking inappropriate actions.

Since safety is a system property, there are—in our opinion—really no software specific safety issues; from a safety perspective, software components are simply treated as any other components of the overall system—albeit components exhibiting the specific development challenges of software [9, 10]. Therefore, in short, to develop a safe systems we would (1) identify the system hazards and define safety requirements, (2) determine how the various components in the system can contribute to these hazards, (3) define derived safety requirements for the components (repeat recursively over the system hierarchy as needed), and then (4) develop our components to meet these safety requirements. Should one or more of these components be software components, we would simply apply solid software engineering techniques to ensure that the safety requirements allocated to the software have been met. (Or, better yet, redesign our system so that the software components are no longer critical [47].) The safety requirements on the software are really no different than any other (really important) software requirements. Therefore, to solve the safety problem in software intensive systems we need to apply rigorous systems safety engineering to design safety into our system and then use solid software engineering techniques described elsewhere in this volume to make sure the software meets any safety requirements; techniques such as requirements engineering [13], software and systems architecture [79], formal methods [18], and rigorous testing and inspections [2].

The End.

Well, maybe this is somewhat of an oversimplification. As we all know, the nature of software components is quite

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

different from the nature of physical components [9], and we are still grappling with how to develop software well. In particular, software has no random failures like physical components; software does not wear out. Flaws in software are systematic stemming from flawed requirements (plain wrong, incomplete, inconsistent), design flaws, or (to a lesser extent) implementation flaws [35, 47, 52]. McDermid *et al.* point out that the "software safety" activities boil down to eliminating these flaws during the software development and then demonstrating that they have, in fact, been eliminated so that we can put trust in our system. They refer to these two concerns as *achieving* and *assuring* safety [58], both of which we are currently not adequately equipped to address.

In the ICSE 2000 edition of The Future of Software Engineering (FoSE), Robyn Lutz provided a roadmap for the future of software engineering for safety [53] and pointed out five challenges that have to be addressed to (partially) solve the problems with achieving and assuring safety.

- Provide readier access to formal methods for developers of safety-critical systems by further integration of informal and formal methods.
- Develop better methods for safety analysis of product families and safe reuse of Commercial-Off-The-Shelf (COTS) software.
- Improve the testing and evaluation of safety-critical systems through the use of requirements-based testing, evaluation from multiple sources, model consistency, and virtual environments.
- Advance the use of runtime monitoring to detect faults and recover to a safe state, as well as to profile system usage to enhance safety analyses.
- Promote collaboration with related fields in order to exploit advances in areas such as security and survivability, software architecture, theoretical computer science, human factors engineering, and software engineering education.

These challenges are as valid today as they were seven years ago and have been only partially addressed since then. Therefore, in this edition of FoSE we will not revisit these challenges. (The reader is strongly encouraged to read Robin Lutz's paper for a detailed discussion of the challenges [53].)

None of the challenges Robyn Lutz identified nor any of the ones identified in this paper are specific to safety critical systems. The challenges faced in critical systems are general software development challenges that involve many aspects of software engineering. Therefore, most of the reports included in the volume on The Future of Software Engineering at ICSE 2007 in Minneapolis [6] (as well as the previous version from ICSE 2000 in Limerick [22]) are highly relevant. Nevertheless, in this report we single out four issues we believe are problem areas of particular relevance to the development of safety critical software. For consistency, we have used Lutz's criteria when selecting the issues [53]: (1) the issues must be important to achieving safety in actual systems, (2) some approaches to addressing the issues are indicated in the literature, and (3) significant progress can be made within the next decade. In addition, to avoid unnecessary repetition, we have avoided the still highly relevant issues identified in 2000.

First, the nature of safety is continuing to be widely misunderstood and there is seemingly a widespread ignorance of established safety techniques. For example, safety is often confused with reliability, thus leading to resources being spent on improving component reliability rather than designing safety into the system. Second, as pointed out above, there is a difference between *developing* a safe system (*achieving* safety in the McDermid *et al.* terminology) and *demonstrating* that the system is safe (*assuring* safety). Our ability to demonstrate (certify) that safety requirements have been met is currently inadequate. Third, reliance on models and automated tools in software development, for example, formal modeling, automated verification, code generation, and automated testing, promises to increase productivity and reduce the very high costs associated with software development for critical systems. The reliance on tools rather than people, however, introduces new and poorly understood sources of problems, such as the level of trust we can place in the results of our automation. Finally, data-driven safety-critical systems are becoming increasingly common, for example, medical systems responsible for dispensing prescription drugs or other treatments. Incorrect data provided to such systems could have catastrophic and widespread consequences; techniques to assure the validity of the data are needed.

In Sections 2 through 5 we discuss the four problem areas in order. Section 6 summarizes and concludes.

2. State of the "Practice"

Although the notion of system safety has been around for decades and the role of software in critical systems is well understood, there is still a prevalent misunderstanding of the nature of safety as well as widespread ignorance of safety engineering techniques, in particular in the software engineering field [48]. We will illustrate the problems with a few anecdotes from the last few years. (Since there is—to our knowledge—no empirical data to support this point, these anecdotes will have to do.)

An alarmingly frequent question we encounter working with software and critical systems is paraphrased as follows.

Question: *"I have developed a software system for a safety*

critical application, how do I demonstrate that it has the failure rate (reliability) of 10^{-x} *or lower as required.*[1]*"*

This question begs the counter-question *"Why? Why do you need that level of 'reliability'?"* In many instances the software developer cannot provide an answer. In fact, in most cases it is not clear which critical functions must be provided with this level of reliability; the software is simply not allowed to "fail" more often that the reliability requirement states. As several researchers have repeatedly pointed out, most accidents involving software are not because the software stopped working (failed). Instead they are *systems accidents* where an accident occurred even though the software operated exactly as required and no component failure occurred [47, 48]; the system as a whole was simply not well understood, its required behavior inappropriate for the particular environment, and—given the right circumstances—an accident followed. Focusing on component reliability in such circumstances can be a waste of resources or even detrimental to safety. As an example, consider a car.[2] This car works like any other car with one notable exception; the requirements engineering team made a slight oversight and never captured the requirement that the car must have brakes. Such a vehicle will clearly be highly unsafe as soon as it starts moving. On the other hand, if the car is standing still it is perfectly safe. Therefore, if the "start" function of the car was unreliable (the car only starts once in a while) the car would at least be safe once in a while. If the failure rate of the start function was 100% the car would actually be safe. Any improvement in the reliability will in this case make the car less safe. Such an example may seem both trivial and contrived, but more convoluted variations of this theme are surprisingly common in practice.

Early this decade there was an informal workshop on certification of medical devices. This workshop was well attended, in particular by representatives from highly entrepreneurial start-up companies. Two exchanges between company representatives and the workshop organizers are illustrative of serious problems (again we paraphrase).

During the plenary session:
Question: *"We are preparing to submit our new [device with various real-time constraints] for certification. Our software has been developed using Visual Basic under Windows, do you have any advice on how to best prepare our certification package?"*
Response: *"[Stunned silence.]"*

In subsequent discussions it became clear that the engineers had tackled the problems and made technology decisions largely ignorant of the challenges they would face, the technologies readily available, and proper use of these technologies. The representatives from the company were basically in tears when they understood the seriousness of their situation and, to our knowledge, this product was never submitted for certification and the company is no longer in business.

Private exchange during a break:
Question: *"The feedback you provided suggested that we must add an emergency shutdown button for our [eye laser surgery] device in case the software malfunctions. Will this design be acceptable?"*
They proceeded to show a diagram where an emergency shutdown button—presumably big and red—had been added and wired into the computer.
Response: *"[More stunned silence.]"*

Needless to say, the very purpose of an independent emergence shutdown system to protect against faulty software and computer failures is nullified by such a design.

Fundamental misconceptions about safety and software development of this nature are—from what we have seen—alarmingly common. This state of "practice" is highly unfortunate since there is an extensive body of work on system safety. Various system safety processes have been extensively described in previous publications and this paper will only contain a brief overview below to provide context and a few pointers to relevant literature. For more detailed discussions, Leveson's book on system safety and computers is required reading and provides an excellent overview of all aspects of system safety and the impact of software in modern systems [47]. Robyn Lutz also provides a succinct overview of the field in her FoSE paper from ICSE 2000 [53].

As an example of a safety analysis process applicable in a number of dominans we will use SAE standard ARP 47-61 [73], a process widely used in the avionics industry. The descriptions of the various phases of the safety assessment process covered in this section are largely adopted from the ARP 47-61 document [73].

The safety assessment process is an inherent part of the system development process (Figure 1). The safety assessment process includes safety requirements identification (on the left side of the "V" diagram) and verification (on the right side of the "V" diagram) supporting the aircraft development activities. An aircraft-level Functional Hazard Analysis (FHA) is conducted at the beginning of the aircraft development cycle and a system design mitigating (or, ideally, eliminating) the hazards is iteratively derived. There are two levels of FHA for avionics systems; the aircraft-level FHA and the system-level FHA. The FHA establishes the derived safety requirements for each aircraft system. Techniques used in this stage would include, for example, Preliminary Hazard Analysis, Failure Modes, Effects, and Criticality Analysis (FEMECA), Fault

[1] Insert your favorite value for x; typically ranging from 3 to 9.
[2] The car example is adopted from Dr. Nancy Leveson of MIT.

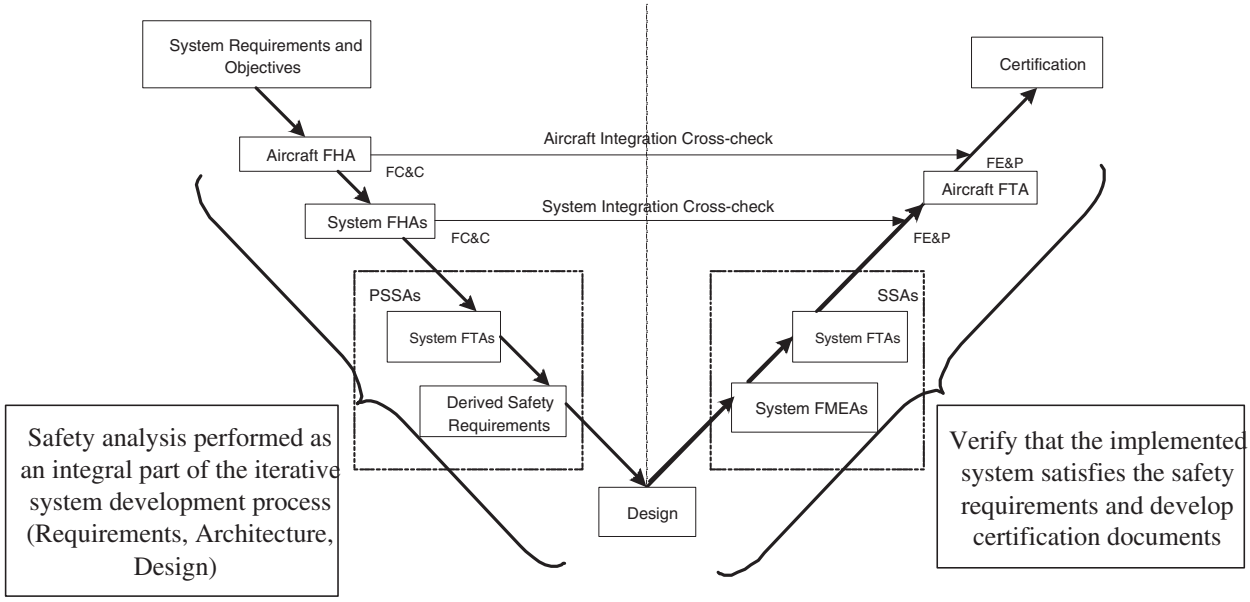

Figure 1. Traditional "V" Safety Assessment Process in the Avionics Industry

Tree Analysis (FTA), and Hazards and Operability Analysis (HAZOP) [47, 68, 76].

The FHA is followed by Preliminary System Safety Assessment (PSSA), which derives safety requirements for the subsystems, primarily using Fault Tree Analysis (FTA). The PSSA process iterates with the design evolution, with design changes necessitating changes to the derived system requirements (and also to the fault trees) and potential safety problems identified through the PSSA feeding into the design process. Some of the important documents coming out of PSSA are planned compliance methods with FHA requirements, updated FHAs, lower-level safety requirements, qualitative FTAs, and operational requirements.

Once the design and implementation are completed (left side of the "V"), the System Safety Assessment (SSA) process verifies whether the safety requirements are met in the implemented design (the right side of the "V"). A System Safety Assessment is a systematic, comprehensive evaluation of the implemented system, along with its architecture and installation, to show that the relevant safety requirements are met. The difference between the PSSA and the SSA is that a PSSA is a method to evaluate proposed architectures and derive system/item safety requirements; whereas the SSA is a verification that the implemented design meets both the qualitative and quantitative safety requirements as defined in the FHA and PSSA. Typical techniques used in this process are Fault Trees and Failure Modes and Effects Analysis (FMEA).

At this point it is worth noting that high levels of safety are typically best achieved during system design by designing safety in from the start; not by attempting to add protection systems and additional complexities after system has been built [47]. Attempting to achieve safety during the activities on the right side of the "V" in Figure 1 is futile unless we have engineered safety into our system during the activities on the left side of the "V". Given the excellent resources available [47, 48, 68, 73, 76], the misconceptions about safety and the engineering process outlined earlier in this section simply should not occur.

The brief discussion above does not address software specifically; software is simply viewed as any other component in the overall system. The assurance that the software meets its identified safety requirements is achieved by demonstrating that the development of the software is in compliance with some standard governing software development for critical software, in our case DO-178B [71]. There is a rather extensive collection of standards covering the critical systems domain. For an comprehensive overview of the standards the reader is referred to Herrmann [30]. In general, these standards deal with safety and certification by mandating or recommending various development techniques and development processes; there is no requirement to produce direct evidence that the various safety requirements have actually been met. As long as the appropriate documents are produced and required activities have been performed we will simply accept the software as being correct. The shortcomings of this prevalent approach to certification will be discussed further in Section 3.

Areas Needing Work: Why do we encounter widespread lack of knowledge of basic system engineering and system

safety techniques when there are suitable guides such as ARP 47-61 [73] readily available?

The source of these problems is—in our opinion—primarily rooted in the education provided in our engineering and computer science curricula; engineers in the application domain typically do not appreciate the complexities of software and the computer science graduates are generally unaware of basic systems engineering and safety engineering techniques. At least in the United States, traditional engineering programs are filled with courses required for accreditation in the specific engineering discipline (may that be, for example, mechanical, electrical, aeronautical, or computer engineering) and there is simply no room for the introduction of software engineering as a specialization. (We cannot comment on the situation in other parts of the world, but we suspect it is similar.) The computer science curricula generally do not have a focus on developing quality software and are weak on engineering basics. The fundamentals of software development (topics such as abstraction, separation of concerns, modularity, etc.) are generally not covered in any detail and the focus of the education are on the currently fashionable languages, applications, development techniques, and research areas. As an example, a few years back the new edition of a popular software engineering textbook quietly dropped its chapter on structured analysis; to the students it now seems like the only design technique available is object oriented and the only design notation is the Unified Modeling Language [72]. Others have observed the problems with our educational system and provide a more thorough and thoughtful discussion [38, 39, 40, 66].

We do not have a ready solution to the education problem to present in this paper, but as a community this is an issue that must be addressed before we can expect the situation to improve [43]. As a low-cost start, however, any organization embarking on safety-critical software development should be aware of the challenges involved and require all managers and engineers to read *Safeware: System Safety and Computers* [47] for an overview of the problems.

At this point it is worth pointing out that improving our technical activities (as discussed in this paper) will only solve a small part of the safety puzzle. Most problems with safety are not technical but managerial or organizational; safety is simply not viewed as a priority. Recent work by Nancy Leveson on STAMP (Systems-Theoretic Accident Modeling and Process) takes a broader view of accidents and their causes than traditional (chain of events) accident models and should be required reading [48, 49, 50]. Although her work promises to have a dramatic impact on the development of safety-critical systems, it is not directly related to software and is outside the scope of this report where we focus on the challenges more directly related to software.

3. Software Certification

When software "fails" it is a result of a design fault introduced somewhere during the software development process. Therefore, most widely used standards, for example, IEC 61508 [34], DO-178B [71], and the former (British) Defence Standards 00-55 and 00-56 [62, 63], are focused on the development process and either recommend or require various development and assessment techniques. The standards are aimed at reducing the number of software faults *introduced* during the development (by requiring, for example, rigorous specification of the software requirements) and increasing the number of faults *eliminated* in the process (by requiring, for example, rigorous testing techniques).

The current standards typically distinguish between different "levels" of safety for a piece of software and there are increasingly stringent development practices required the higher the "safety level" of the software under consideration. There are doubts, however, if there is really a correlation between the quality of the software produced and the practices required at each "safety level" [57]. Although the observations are largely anecdotal, there are some indications that developing software to a higher "safety level" (in this case to the DO-178B standard) does not necessarily lead to lower failure rates [75]. This finding raises doubts as to the effectiveness of the various techniques prescribed for higher "safety levels" and casts doubt on the general approach of process oriented standards.

In addition, since process standards prescribe various techniques, adoption of new—and potentially much more effective—techniques is slow. For instance, with respect to the application of formal proof techniques, DO-178B will accept proofs, but only if we can demonstrate that these techniques are as effective as the prescribed testing techniques they are to replace. Since such arguments can be costly to prepare and there is no guarantee that they will be accepted by regulators, this serves as an obstacle to the introduction of new techniques (this issue will be further discussed in Section 4 in the context of tools and automation).

Areas Needing Work: To address these challenges and take full advantage of the advances in software development techniques we will have to make a move to an *evidence based* approach to certification. McDermid concisely states the point [57].

> *The principle of using software safety evidence is simple. First, identify the potential failure modes of software which can give rise to, or contribute to, hazards in the system context. Second, provide evidence that these failure modes:*
>
> - *Cannot occur, or*

- *Are acceptably unlikely to occur, or*
- *Are detected and mitigated so that their effects are acceptable.*

The evidence presented would take the form of a *safety case* (the names dependability case or assurance case are also commonly used). Bishop and Bloomfield define a safety case as follows [3]:

> *"A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment"*
> *To implement a safety case we need to:*
>
> - *make an explicit set of claims about the system*
> - *produce the supporting evidence*
> - *provide a set of safety arguments that link the claims to the evidence*
> - *make clear the assumptions and judgements underlying the arguments*
> - *allow different viewpoints and levels of detail.*

The impact of safety cases on software development will be that we have to identify and rigorously capture any safety requirements and then explicitly provide evidence that these requirements have been met. The difference from most current certification practice is that we are now free to structure our process, choose our development techniques, and use the tools we deem will best meet our needs. Many development practices advocated in current standards will most likely find a room in evidence based certification since they represent good software development practices. For example, there will still be a need for rigorous software development processes to ensure that appropriate activities have been performed, that evidence is traceable, that the version of the software installed is the same as the version of the software certified, etc.

Although the notion of evidence based verification is intuitively appealing there are practical issues that will have to be addressed before it can fully replace current certification standards.

First, the safety cases must be represented in an intellectually defensible way. As we know from the requirements engineering community [13], natural langauge is generally unsuitable for the capture of the rigorous arguments needed for a safety case. Recent work on Goal Structuring Notation (GSN) [3, 36] is a direction that promises to assist in the construction of rigorous arguments. GSN is a graphical notation that explicitly represents the individual elements of a safety argument (requirements, claims, evidence and context) and the relationships that exist between these elements. Although such a notation might help in arguing a safety case, it is no guarantee that the arguments will be correct. Greenwell *et al.* surveyed a collection of safety cases and found that logical fallacies in the reasoning were surprisingly common [24]. Further investigations into how to effectively argue safety cases as well as the notations usable when constructing these cases are needed.

Second, the Greenwell *et al.* study only investigated the logic behind the arguments in the safety cases; they did not look at the evidence presented as a basis for the arguments [24]. The type of evidence that can be accepted as a basis for a safety case is currently unclear. Can we argue that source code implements its specification because it was auto-coded with a trusted code generator? Can we demonstrate the same relationship by testing the implementation up to MC/DC coverage [14]? Much work is needed to determine the various types of evidence acceptable when making a safety case.

Finally, the safety case must ultimately be accepted or rejected by some regulatory body. This will inevitably be a subjective process and it is not clear how such a regulatory body would be structured. A possible model could be the U.S. Food and Drug Administration (FDA) Drug Review Panels. These panels of experts review clinical data intended to demonstrate the safety and effectiveness of new drugs (in effect they review a safety case). Nevertheless, as some recent well publicized drug recalls, for example VIOXX [59], demonstrate, this is not a perfect process and mistakes will inevitable occur. Although product recalls have raised serious concerns as to the current integrity of the FDA review process [5], the panel system still seems promising.

There is a risk that different certification panels might view evidence in different ways. A specific panel might be likely to accept safety cases based on formal proof evidence and be sceptical towards test-based evidence, and vice versa. Developers of critical systems shopping around for "friendly" evaluation bodies would be a highly undesirable outcome. Issues of conflict of interest are also concerns [5]. Clearly, there are many "nuts-and-bolts" issues to address before this certification approach can be fully adopted, but is in our opinion the most promising direction in the long term.

4. Model-Based Development

Traditionally, software development in critical system has been largely a manual endeavor. Validation that we are building the right system has been achieved through requirements and model inspections and reviews. Verification that the system is developed to satisfy its specification is archived through inspections of design artifacts and ex-

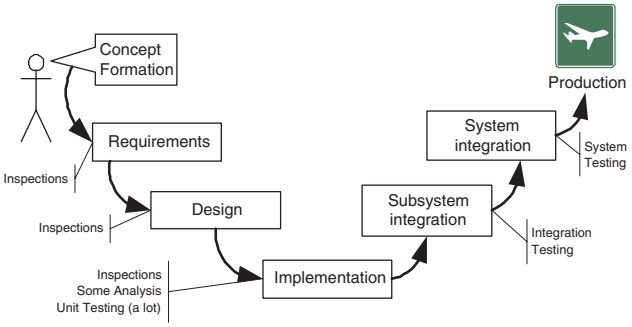

Figure 2. Traditional Software Development Process.

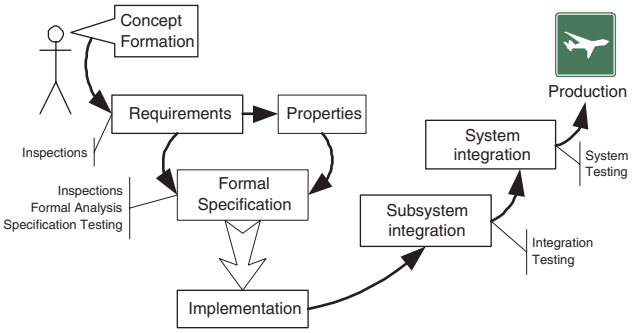

Figure 3. Model-Based Development Process.

tensive testing of the implementations (Figure 2). In critical software systems the validation and verification phase (V&V) is particularly costly and may consume 50%–70% of the software development resources. Thus, if we could devise techniques to help us reduce the cost of V&V, dramatic cost savings could be achieved. The current trend towards *model-based development* is one attempt to address this problem (see the paper in this volume for further details on this topic [23]).

In model-based development, the development effort is centered around a formal or semi-formal model of the proposed software system. Through manual inspections, formal verification, and simulation and testing we convince ourselves (and any regulatory agencies) that the model possesses desired properties. The implementation is then *automatically and correctly generated* from this model and little or no additional testing of the implementation is required. (Note here that only certain parts of an overall software system will be amenable to this type of model-based development; the issues covered in this section, however, are applicable to any tools intensive software development.) There are currently several commercial and research tools that aim to provide part or all of these capabilities—commercial tools are, for example, Simulink and Stateflow from Mathworks [54, 55], Esterel and SCADE from Esterel Technologies [19, 20], Statemate from i-Logix [25], SpecTRM from Safeware Engineering [51], and various UML tools from a collection of vendors.

The capabilities of model-based development enable us to follow a different development process. The development will now be centered around the model and the V&V has been largely moved from testing and analyzing the code (Figure 2) to analyzing and testing the model (Figure 3). Productivity improvements will (hopefully) be achieved through reduced emphasis on unit testing of code, increased reliance on automated analysis tools applied in the model domain, and trusted code generation. Our reliance on modeling and automation leads to several new challenges. It is now imperative that the model serving as the basis for our development is correct with respect to the customers' true needs (and safety requirements); a demand that can only be met through extensive model validation. Furthermore, increased reliance on tools requires that they can be trusted so that the results can be used as evidence in certification (see Section 3).

4.1. Model Validation

Consider a recent extensive formal modeling project [60, 61] conducted in collaboration between industry and academia (Rockwell Collins Inc. and the University of Minnesota). The subject was a Flight Guidance System (FGS), which is a component of the overall Flight Control System (FCS) in a commercial aircraft. The FGS can be broken down to mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. In this study we used the mode logic for horizontal and vertical modes.

The industry team collected the system requirements as informal "shall" statements. These requirements were relatively mature and thought to be well-understood. The next phase, modeling, consisted of constructing by hand an executable model that we believed exhibited the behavior informally stated in the shall statements; in this case we used the RSML^{-e} notation developed at the University of Minnesota. Throughout creation of the model, we continually executed the model to informally confirm that it behaved as we expected.

In the subsequent formal verification phase, we manually translated the shall statements into formal properties stated over the model in CTL [17] merged these formal properties with the translation of the RSML^{-e} model into

NuSMV [64]. We developed 300+ CTL properties based on the informal requirements. A detailed description of the model and its simulation environment is available in [61].

The process of creating a model from the English prose requirements caused us to go back and clarify the English statement of the requirements. In the same way, translating the English statements into SMV also prompted us to go back and clarify the English statement. In addition, the verification that the model satisfied the requirements (formalized as CTL properties) led to additional insight into the validation problem. For example, consider the well-validated and non-controversial requirement below.

> *If Heading Select mode is not selected, Heading Select mode shall be selected when the HDG switch is pressed on the Flight Control Panel.*

After formalization into CTL, this property did not verify in our model. The model-checker pointed out two ways in which this property could be violated. First, if another event arrived at the same time as the HDG switch was pressed, that event could preempt the HDG switch event. Second, if this side of the FGS was not active, the HDG switch event was completely ignored by this FGS side. (There are two FGSs installed on an aircraft, one is active and the inactive one operates as a hot spare.)

The counterexamples from NuSMV led us to modify the requirement to state

> *If this side is active and Heading Select mode is not selected, Heading Select mode shall be selected when the HDG switch is pressed on the FCP (providing no higher priority event occurs at the same time).*

We found that the process of proving the properties forced us to go back and modify virtually all of the English requirements (note here that these were the clarified requirements derived during the formalization process).

When developing formal models of any substantial system, the models will most likely be incorrect with respect to the real needs of the system. In our case, the three complementary artifacts—informal English language requirements, requirements formalized as CTL properties, and an executable formal model—served to check each other in a rigorous validation process. Had we only built the executable model and validated it through testing, chances are significant flaws would have remained. Similarly, had we been blessed with a correct-by-construction tool that would have helped us refine our 300+ CTL properties to an implementation, the implementation would certainly have been grossly incorrect with respect to the customers' real needs. It is clear that a *rigorous validation process* for both models and properties must be in place to ensure that any formal artifacts serving as the basis for downstream automation are correct; without this validation any breakthroughs in verifiably correct code generation and formal verification will achieve limited success.

Areas Needing Work: The model validation problem has received little attention and the sufficiency of the validation activities has been largely determined through ad-hoc methods. To adequately address this issue, there are several questions that must be answered: How do we know when we have validated the model sufficiently? How can we determine whether there are missing or unstated high-level requirements not captured in the model (or in the properties used for verification)? What techniques can we use for model validation? Can these techniques be automated?

Naturally, the first step to address this problem is to draw on the elicitation and validation techniques developed in the requirements engineering community [13] and continue develop these techniques to be applicable in the model-based domain.

As pointed out in Section 4.1, using the informal requirements and their formalization in terms of a model and a set of properties to hold over the model served as a useful internal crosscheck to catch flaws in either one of the artifacts. The question is if this process can be formalized and automated.

Whalen *et al.* have investigated if the notion of test-adequacy coverage criteria can be extended to apply directly to the requirements and the properties as opposed to the models [80]. They derive a test-suite from the set of requirements to provide high *requirements coverage* and then run that test-suite over the model. If one achieves high coverage of the requirements but low coverage of the model it would point to one (or more) of three problems:

1. The model is incorrect. The model allows behaviors not specified by the requirements. Hence a test-suite that provides a high level of requirements coverage will not cover these incorrect behaviors, thus resulting in poor model coverage.

2. There are missing requirements. Here, the model under investigation may be correct and more restrictive than the behavior defined in the original requirements; the original requirements are simply incomplete and allow behaviors that should not be there. Hence we need additional requirements to restrict these behaviors. These additional requirements necessitate the creation of more test-cases to achieve requirements coverage and will, presumably, lead to better coverage of the model.

3. The criterion chosen for requirements coverage is too weak.

This work is an initial step towards providing tool support to help assess the adequacy of a set of requirements. More work is clearly needed however. For example, it is not clear what would constitute a suitable requirements coverage criterion.

Other researchers have investigated this problem from a verification (as opposed to a testing) perspective. Recent work by Chockler *et al.* in assessing the completeness of temporal logic properties over models is closely related to the validation problem [15, 16]. They propose coverage metrics that help determine which parts of the model were covered by the properties during verification. These techniques attempt to help identify the reasons for poor model coverage and help in debugging the model or/and the properties

Vacuity checking of temporal logic formulas [1, 42, 67] is also relevant in this context. Intuitively, a model *vacuously satisfies* a property f if a subformula ϕ of f is not necessary to prove whether or not f is true. Using the techniques in [1, 42, 67], it is possible to investigate the "quality" of the properties (check if any properties are vacuous). If a valid property turns out to be vacuous in a model, then it can be replaced by a stronger valid property. Beer *et al.* [1] observed that formal verification of hardware at IBM revealed around 20% of the properties to be vacuous during the first formal verification runs of a new hardware design. Chechik *et al.* have also investigated the vacuity problem including how to detect problems with models of the environment [11].

The work discussed above make a point in that model validation cannot stop with simply ensuring that requirements hold of a model since requirements can be satisfied in several trivial and uninteresting ways, and may be incorrect themselves; we need rigorous means of validating the requirements as well as the model. Requirements-based testing, vacuity detection, and finding environment guarantees are all important steps in this direction and merit further exploration.

Inspections will always be an invaluable tool when determining whether or not a set of properties accurately captures the desired requirements and a model describes the system the customer really wants. Consequently, any property specification language or modeling language must be readable and understandable enough so that all stakeholders can be involved in the inspections process. Unfortunately, there has been a traditional dichotomy between formality and readability; developers of formal notations did not concern themselves with readability to any large extent and developers of modeling notations widely used in industry were generally not concerned with rigorous semantics. In our experience, the surface syntax of a language is hugely important and can make or break adoption of a modeling approach in practice [44, 51, 81]. Without a modeling language acceptable to all stakeholders, the language will simply not get used and all our research into formal techniques will not make it into software engineering practice.

Little work has been done in investigating the readability and understandability of modeling notations. A first step can be found in Zimmerman *et al.* [82] where they performed a pilot experiment indicating that complex condition expressed in a tabular notations as found in SCR [29], RSML [44], and SpecTRM [45] might be more readable than the logic-gates found in notations such as Simulink [55] and SCADE [20]. They also found that the notion of internal events in state machines (as found in Statecharts [25], Stateflow [56], and UML [72]) may hinder readability as compared to the data flow semantics found in, for example, SCADE, SCR, and SpecTRM. Ironically, Statecharts, SCADE, Simulink, Stateflow, and UML are by far the most influential notations in industry practice. In our opinion, significantly more research in modeling language design and usability is needed so the research community can better influence the next generation of modeling languages and tools.

4.2. Tool Certification

Ongoing research efforts are providing dramatic improvements in specification, analysis, and testing of formal models. Conferences such as, for example, Computer Aided Verification (CAV), Automated Software Engineering (ASE), International Symposium on Software Testing and Analysis (ISSTA), Formal Methods Europe (FME), and Tools and Algorithms for the Construction and Analysis of Systems (TACAS) are dedicated to novel tools and techniques for the development and analysis of systems. What is largely missing in current research efforts, however, is a discussion of how we will be able to trust our powerful automated techniques enough to allow us to use them as a replacement for traditional testing—*verification and validation of our automation* poses a serious challenge.

Problems with the automation can manifest itself in many places. For example,

- If the model execution environment misrepresents the semantics of the modeling language, all testing done in the model domain is invalid.
- If the code generation is incorrect, the resulting implementation will naturally be incorrect (and it may not be caught since we are now reducing testing in the code domain).
- If any of our analysis tools applied in the model domain provide false negatives (they fail to catch a faulty model), we may mistakenly accept a model as correct and use it for code generation (again, this problem is unlikely to be caught with the reduced code testing).

Solutions to such problems must be provided before we can confidently use extensive automation in the development of critical systems as well as evidence in certification (Section 3). Unfortunately, execution environments, code generators, and analysis tools are not simple pieces of software and it is highly unlikely that we will in the foreseeable future be able to provide the level of confidence necessary to trust a specific tool as a development tool [71] in critical systems development. (A development tool in this context is a tool that is directly used to derive an artifact, for example, code generators and compilers are typical development tools.) Also, consider tool evolution (for example, maintenance, upgrades, and migration to new platforms) and the situation looks grim. Because of these difficulties, regulatory agencies have been quite reluctant to qualify any tools for use on critical systems projects and they are actively debating how to address the issue [21, 74]. In our opinion, their reluctance is quite justified; what we need is additional research into means of trusting the results from our tools.

Areas Needing Work: If our conjecture that we will be hard pressed to fully trust individual tools anytime soon is correct, what other avenues are available? From our experiences with working with organizations subjected to regulation in the critical systems domain, there are two directions that seem feasible and acceptable in practice; (1) redundant proof paths and (2) automated test-case generation.

The notion of N-version programming has been suggested as one technique to increase our confidence in a software system [12]. The idea is to execute N implementations of the same system in parallel and vote on the results. If the results are not identical, it is assumed that the majority is correct. By applying a similar idea in verification, we *may* be able to raise the confidence in our verification results. For example, if we verify a property using the symbolic model checker in NuSMV [64], the explicit state model checker SPIN [31], and the theorem prover PVS [65] we may be able to rule out the possibility of any false negatives.

The underlying hypothesis of N-version programming is that there is independence between the implementations so no common cause failures are present. Unfortunately, as we know from the Knight and Leveson experiments [7, 8, 37], this assumption is not true—we rarely (if ever) have truly independent failure modes of our N versions of the software. Even if we take great care in attempting to assure that the N versions are developed independently, conceptual problems inherent in the problem domain are likely to lead to common mode failures. This is naturally an issue when using redundant proof paths as well. Consider our example above. Although NuSMV, SPIN, and PVS are built independently and rely on radically different approaches to verification, a misunderstanding of some obscure parts of the semantics of the modeling language could show up as a common mode failure in all three of our proof paths.

Even with these known problems, redundant proof paths seem like a practical ways of assuring that we can trust the results of our analysis. There are several challenges, however, that must be met before this will be a reality.

- We must understand the level of independence between the various proof paths. We do not want to be in a situation where we are led into a false sense of security—the issues Knight and Leveson identified [7, 8, 37] are as valid today as they were fifteen years ago and must be carefully addressed and controlled.
- Current analysis techniques are not cheap nor easy to use, and applying several redundant techniques will not make it any cheaper. Thus, full automation of the analysis is key to success.

If we could use the model running in its execution environment and the generated code running in its host environment for "back-to-back" testing we may be able to increase our confidence in both the execution environment as well as the code generator. In this scenario, we would automatically generate test-cases from the model, execute the test-cases on both the model and the generated implementation, and compare the results. Any discrepancy between the executions indicates a problem with either (1) the model execution environment, (2) the code generator, or (3) the implementation run-time environment. Recent developments in test-case generation from formal models [4, 28, 32, 33, 69] leads us to believe that this will be a feasible solution to part of our V&V problem in the near future. Nevertheless, there are several additional research questions that must be answered.

How do we know when we have tested enough? Some recent results raise issues with the commonly used model coverage criteria [26, 27]. New model and/or code test-coverage adequacy criteria suitable for this type of testing must be developed and validated.

How do we compare the test results? The results of executing a test in the model domain and in the code domain are likely to differ in subtle, but significant, ways. For example, the model may operate on abstract data whereas the implementation operates on concrete data, the real-time behavior of the model and the code may differ, and the sequencing of events might differ. Techniques to effectively specify when the deviations between model and implementation represent actual faults and when they are acceptable are needed.

4.3. Loss of "Collateral Validation"

Manual processes, may that be design, coding, testing, or putting a medical device through clinical studies, draw on

the collective experience and vigilance of many dedicated software professionals; professionals that provide *"collateral validation"* as they are working on the software. Experienced professionals designing, developing code, or defining test-cases provide informal validation of the software system; if there is a problem with the specified functionality of the system, they have a chance of noticing and taking corrective action. As an example, consider the FGS requirements from the previous section. Although the facts that the FGS had to be active and that no higher-priority events were received at the same time as the HDG Switch were not explicitly stated in the requirements, the engineers implemented the FGS functionality correctly; these problems were caught and corrected in the manual development process. As discussed in the previous section, when replacing these manual efforts with automation, proper validation of the formal requirements models on which the automation is based becomes absolutely essential; there may be no safeguards in the downstream development activities to catch critical flaws in the formal model—the collateral validation has been lost.

Areas Needing Work: Our concern about loss of collateral validation is currently an unsupported conjecture; to our knowledge there has been no studies investigating if this is a valid concern or not. How well do the manual processes work? How error prone are the tools? What, if anything, do we lose when replacing manual processes with tools?

As we ponder these problems, we cannot lose track of one important fact—*although perfection is the goal with our tools, perfection is not necessary for deployment and highly effective use.* After all, our aim with increased use of tools is to replace costly, time consuming, and error prone manual tasks such as inspections and testing, and all that is really needed from our tools is that they are *better* than the manual tasks they replace. Unfortunately, we do not know much about how error prone our manual processes really are, nor do we know how to compare the effectiveness of an automated process to a manual process—much important analytical and empirical research is needed to help answer these questions.

5. Data Intensive Systems

When discussing software development for critical systems we typically focus on the software itself (the program). Often, however, there is a significant *data-component* to a critical system; what Storey and Faulkner [77, 78] call *data-driven systems*. Such systems include critical systems that are customized for their particular installation, for example, a railway control system where the track layout must be defined for each installation or an air-traffic control system that must be configured based on the characteristics of the particular airspace to be controlled. Another category of data-driven systems would be systems that rely on large dynamic databases to function properly. Consider, for example, a traditional infusion pump. Such a pump is critical in that it is essential that it accurately delivers the dosage that the nurse has entered. A patient might be harmed if the pump malfunctions (either hardware or software failure) or the nurse enters the wrong dosage (operator error). Erroneous data entry is a fairly common problem and in an attempt to reduce such errors there is a move towards networked infusion pumps where the dosage would be retrieved from the patient information system and automatically uploaded to the pump; thus, eliminating data entry errors. The potential for widespread calamity from corrupted databases should be obvious. (Incidentally, an infusion pump is regulated as a medical device in the United States, the patient information system—the database—is not regulated in any way.)

As a concrete example of the potential problems, consider a large hospital where the the pharmacy receives a call from a nurse (incident reported by Cook and O'Connor [70]). She is concerned that a patient seemingly had the wrong medication delivered. Delivering the wrong medication is not unheard of, but this instance is different in several respects. The medication delivered had never been used by this patient in the past, there were no records of the patient having the drug prescribed, but—strangely—the medication matched the newly printed medical administration record (MAR). The pharmacy technician also checked the medication against the on-line computerized patient medication list and the list precisely matched the printed MAR. As the technician investigated this incident other calls from wards all over the hospital began coming in. Patients were receiving drugs seemingly inappropriate for their ailments or did not receive any drugs when they should have. In each case, however, the delivered medication (or lack thereof) matched the information in the MAR. Whatever the problem, it was hospital wide. The hospital quickly realized the severity of the incident, stopped all drug delivery, had all wards identify the hard copies of the MAR from the previous day, and then delivered drugs based on the previous day's information as an interim (manual) solution. (Incidentally, this process was greatly hampered by the lack of a manual backup procedures in case of a computer failure.) The source of the problem was a minor software maintenance fix the previous night necessitating a reload of the database from a backup tape. This backup tape was corrupted leading to erroneous prescription information in the database used for the generation of the MARs. After this problem was discovered the database was resorted from a previous backup tape and the problem seemingly solved. Luckily, given the vigilance of the nursing staff, there were no medication misadministration during this incident. Nev-

ertheless, given large scale computerization of, for example, patient information systems, the heavy reliance on commercial off the shelf software (COTS), and the inexperience in critical systems of the software developers in this domain, large scale incidents with much less favorable outcomes seem like a certainty.

Areas Needing Work: The problems of critical systems highly configurable through data and systems relying on large databases for critical functions have been largely ignored since standards do not adequately address the problems. To our knowledge, not much work has been done in this area. Faulkner and Storey identify the problem and provide initial discussion [77, 78]. Knight *et al.* also address the problem, and suggest a specification and verification technique for data [41]. Nevertheless, there is much work to be done in this area both in terms of development and verification techniques as well as certification.

In our opinion, the potential for severe accidents in systems relying on databases is significantly larger than that for the traditional embedded systems which have received most (if not all) of our attention in the past. As the prescription drug example above illustrates, a corrupted database can easily turn "retail" accidents (for example, a nurse administering the wrong drug to a patient) into "wholesale" accidents (all patients in a hospital receiving the wrong drug).[3] Without appropriate attention we believe this type of incidents will overshadow the more traditional accidents associated safety-critical computer systems.

As a final note, evidence based certification (Section 3) would require us to include the database, the COTS database management system, the manual backup procedures, etc. in any safety case for the system used in the example above. A move to this type of certification will most likely accelerate the awareness of the problems associated with data intensive systems and drive us to develop better software.

6. Conclusions

Although software has been an integral part of many safety-critical systems for some time and we have a sizable body of techniques to help us develop software acceptable in such systems there are new challenges emerging. In this report we addressed four issues we believe will pose challenges in the future.

First, the nature of safety is continuing to be widely misunderstood and known system safety techniques are not applied; an issue we believe can be addressed largely through education and training of our software engineering professionals.

Second, our ability to demonstrate (certify) that safety requirements have been met is inadequate. Here we advocated a move towards evidence-based certification and some notion of safety-cases.

Third, the move towards various forms of model-based development with its increased reliance on tools rather than people in the software development process introduces new and poorly understood problems. Research efforts directed towards (1) validation of the artifacts (models) forming the basis for tool intensive development, (2) assuring correctness of our automated tools, and (3) investigating the effect of replacing human activities with automated tools are needed. It would be highly disappointing if the enormous advantages in modeling, verification, and code generation technology we have seen the last decade, and will most likely see in the future, are used to, for example, verify that faulty models have been implemented correctly. A few well publicized failures are enough to make widespread industry adoption and regulatory acceptance very difficult and set our efforts back a decade—let us make sure that this does not happen.

Finally, data-driven safety-critical systems are becoming increasingly common; incorrect data could have catastrophic and widespread consequences. Techniques to assure the validity of the data are needed and we need to closely monitor the convergence of our critical control systems and large information systems.

Acknowledgements

The author wishes to acknowledge the contributions of Steven P. Miller, Michael W. Whalen, and Alan Tribble of Rockwell Collins Inc. for their modeling and analysis efforts, and valuable discussions; the support of our work by Ricky Butler, Kelly Hayhurst, and Celeste Bellcastro of the NASA Langley Research Center; the support of our work by Michael Lowry of NASA Ames Research Center; and the efforts of his current and former graduate students Anjali Joshi, Ajitha Rajan, Yunja Choi, Sanjai Rayadurgam, Devaraj George, and Dan O'Brien of the University of Minnesota.

The authors also wish to thank Michael Peterson of Rockwell Collins Inc. for his invaluable and expert feedback on system safety.

We must, however, emphasize that this paper represents the views of the author, which are not necessarily those of our collaborators or sponsors.

[3] We were first introduced to the notion of "retail" versus "wholesale" accidents by Dr. John C. Knight of the University of Virginia.

References

[1] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Formal Methods in System Design*, pages 141–162, 2001.

[2] A. Bertolino. Software testing research: Achievements, challenges, dreams. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE-CS Press, 2007.

[3] P. Bishop and R. Bloomfield. A methodology for safety case development. In F. Redmill and T. Anderson, editors, *Industrial Perspectives of Safety-critical Systems: Proceedings of the Sixth Safety-critical Systems Symposium*, pages 194–203. Springer, 1998.

[4] M. R. Blackburn, R. D. Busser, and J. S. Fontaine. Automatic generation of test vectors for SCR-style specifications. In *Proceedings of the 12th Annual Conference on Computer Assurance, COMPASS'97*, June 1997.

[5] *The Future of Drug Safety: Promoting and Protecting the Health of the Public*. Board on Population Health and Public Health Practice (BPH), Institute of Medicine (IOM), 2006.

[6] L. Briand and A. Wolf, editors. *Future of Software Engineering 2007*. IEEE-CS Press, 2007.

[7] S. Brilliant, J. Knight, and N. Leveson. The consistent comparison problem in N-version programming. *IEEE Transactions on Software Engineering*, Vol. SE-165(No. 11), November 1989.

[8] S. Brilliant, J. Knight, and N. Leveson. Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering*, Vol. SE-16(No. 2), February 1990.

[9] F. P. Brooks. No silver bullet – essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[10] F. P. Brooks Jr. *The Mythical Man Month*. Addison-Wesley, anniversary edition, 1995.

[11] M. Chechik, M. Gheorghiu, and A. Gurfinkel. Finding environmental guarantees. In *Proceedings of Fundamental Approaches to Software Engineering (FASE'07)*, To appear in 2007.

[12] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, pages pp. 3–9, Tolouse, France, June 1978.

[13] B. Cheng and J. Atlee. Research directions in requirements engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE-CS Press, 20007.

[14] J. Chilenski and S. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9:193–200, September 1994.

[15] H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi. A practical approach to coverage in model checking. In *Proceedings of the International Conference on Computer Aided Verification (CAV01), Lecture Notes in Computer Science 2102*, pages 66–78. Springer-Verlag, July 2001.

[16] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for temporal logic model checking. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 2031*, pages 528–542. Springer-Verlag, April 2001.

[17] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[18] M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, and W. Visser. Formal software analysis: Emerging trends in software model checking. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE-CS Press, 20007.

[19] Esterel-Technologies. Corporate web page. www.esterel-technologies.com, 2004.

[20] Esterel-Technologies. SCADE Suite product description. http://www.esterel-technologies.com/v2/ scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html, 2004.

[21] Software Tools Workshop of FAA and Embry-Riddle Aeronautical University. http://www.erau.edu/db/campus/softwaretoolsforum.html, May 2004.

[22] A. Finkelstein, editor. *ICSE 2000 - Future of Software Engineering Track*, New York, NY, USA, 2000. ACM Press.

[23] R. France and B. Rumpe. Model-driven development of complex systems: A research roadmap. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE-CS Press, 20007.

[24] W. S. Greenwell, J. C. Knight, C. M. Holloway, and J. J. Pease. A taxonomy of fallacies in system safety arguments. In *Proceedings of the 2006 International System Safety Conference*, 2006.

[25] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[26] M. P. Heimdahl and G. Devaraj. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, Linz, Austria, September 2004.

[27] M. P. Heimdahl, G. Devaraj, and R. J. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE)*, Tampa, Florida, March 2004.

[28] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *3rd International Worshop on Formal Approaches to Testing of Software (FATES 2003)*, 2003.

[29] K. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, Januaray 1980.

[30] D. S. Herrmann. *Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors*. Wiley-IEEE Computer Society Press, USA, 2000.

[31] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on software Engineering*, pages 279–295, May 1997.

[32] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Proceedings of the International Conference on Software Engineering*, Portland, Oregon, May 2003.

0-7695-2829-5/07 $20.00 © 2007 IEEE

[33] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '02)*, Grenoble, France, April 2002.

[34] *IEC-61508: Functional Safety of Electrical/ Electronic/ Programmable Electronic Safety-Related Systems*. International Electrotechnical Commission (IEC), 1999.

[35] M. S. Jaffe, N. G. Leveson, M. P. Heimdahl, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

[36] T. P. Kelly and R. A. Weaver. The goal structuring notation–a safety argument notation. In *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[37] J. Knight and N. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, Vol. SE-12(No. 1):96–109, January 1986.

[38] J. C. Knight. Focusing software education on engineering. *ACM SIGSOFT Software Engineering Notes*, Vol. 30(No. 2), March 2005.

[39] J. C. Knight and N. G. Leveson. Should software engineers be licensed? *Communications of the ACM*, Vol. 45(No. 11):87–90, November 2002.

[40] J. C. Knight and N. G. Leveson. Software and higher education. *Communications of the ACM*, Vol. 49(No. 1):160, January 2006.

[41] J. C. Knight, E. A. Strunk, W. S. Greenwell, and K. S. Wasson. Specification and analysis of data for safety-critical systems. In *22nd International System Safety Conference*, Providence, RI, August 2004.

[42] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *Journal on Software Tools for Technology Transfer*, 4(2), February 2003.

[43] T. Lethbridge, J. Diaz-Herrera, R. LeBlanc, and J. Thompson. Improving software practice through education: Challenges and future trends. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE-CS Press, 20007.

[44] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[45] N. Leveson, J. Reese, and M. Heimdahl. SpecTRM: A CAD system for digital automation. In *Proceedings of the 17th Digital Avionics Systems Conference*, November 1998.

[46] N. G. Leveson. Software safety: Why, what, and how? *ACM Computing Surveys*, 18(2), June 1986.

[47] N. G. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, 1995.

[48] N. G. Leveson. *System Safety Engineering: Back To The Future*. On line publication: http://sunnyday.mit.edu/book2.pdf, 2002.

[49] N. G. Leveson. A new approach to hazard analysis for complex systems. In *Proceedings of the International Conference of the System Safety Society*, Ottawa, Canada, August 2003.

[50] N. G. Leveson. A systems-theoretic approach to safety in software-intensive systems. *IEEE Transactions on Dependable and Secure Computing*, Vol. 1(No. 1):66–86, January-March 2005.

[51] N. G. Leveson, M. P. Heimdahl, and J. D. Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, pages 127–145, September 1999.

[52] R. R. Lutz. Targeting safety related errors during software requirements analysis. *Journal of Systems Software*, 34(3):223–230, September 1996.

[53] R. R. Lutz. Software engineering for safety: a roadmap. In A. Finkelstein, editor, *ICSE 2000 - Future of Software Engineering Track*, pages 213–226, New York, NY, USA, 2000. ACM Press.

[54] MathWorks. The MathWorks Inc. corporate web page. http://www.mathworks.com, 2004.

[55] Mathworks Inc. Simulink product web site. Via the world-wide-web: http://www.mathworks.com.

[56] Mathworks Inc. Stateflow product web site. vVia the world-wide-web: http://www.mathworks.com.

[57] J. A. McDermid. Software safety: where's the evidence? In *SCS '01: Proceedings of the Sixth Australian workshop on Safety critical systems and software*, pages 1–6, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[58] J. A. McDermid and D. J. Pumfrey. Software safety: Why is there no consensus? In *Proceedings of the 19th International System Safety Conference*. System Safety Society, 2001.

[59] Merck. Vioxx home page. http://www.vioxx.com/, 2004.

[60] S. Miller, A. Tribble, T. Carlson, and E. J. Danielson. Flight guidance system requirements specification. Technical Report CR-2003-212426, NASA, June 2003.

[61] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *Int. J. Softw. Tools Technol. Transf.*, 8(4):303–319, 2006.

[62] *Requirements for Safety Related Software in Defence Equipment, Issue 2*. UK Ministry of Defence, 1997.

[63] *Safety Management Requirements for Defence Systems, Issue 2*. UK Ministry of Defence, 1996.

[64] The NuSMV Toolset, 2005. Available at http://nusmv.irst.itc.it/.

[65] S. Owre, N. Shankar, and J. Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, March 1993.

[66] D. L. Parnas. Education for computing professionals. *IEEE Computer*, Vol. 23(No. 1), January 1990.

[67] M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. In *Proceedings of the 14th Conference on Computer Aided Design*, pages 485–499. Springer-Verlag, 2002.

[68] D. Raheja. *Assurance Technologies: Principles and Practices*. McGraw-Hill, 1991.

[69] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91. IEEE Computer Society, April 2001.

[70] R. I. C. RI and M. F. O'Connor. *Medication Safety: A Guide to Health Care Facilities*, chapter Thinking about accidents and systems, pages 73–87. American Society of Health-System Pharmacists, Bethesda, MD, 2005.

[71] RTCA. *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[72] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1998.

[73] SAE-ARP4761. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.

[74] RTCA SC-205 (Joint with EUROCAE WG-71) Software Considerations. http://www.rtca.org/comm/Committee.cfm?id=55.

[75] M. L. Shooman. Avionics software problem occurrence rates. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, pages 55–64, 1996.

[76] N. Storey. *Safety-Critical Computer Systems*. Addison Wesley Longman, Harlow, England, 1996.

[77] N. Storey and A. Faulkner. The characteristics of data in data-intensive safety-related systems. In *SAFECOMP*, pages 396–409, 2003.

[78] N. Storey and A. Faulkner. Data—the forgotten system component? *Journal of System Safety*, Vol. 39(No. 4):10–14, 36, 2003.

[79] R. Taylor and A. van der Hoek. Software design and architecture: The once and future focus of software engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE-CS Press, 20007.

[80] M. Whalen, A. Rajan, M. Heimdahl, and S. Miller. Coverage metrics for requirements-based testing. In *Proceedings of International Symposium on Software Testing and Analysis*, July 2006.

[81] E. V. Wyk and M. P. Heimdahl. Flexibility in modeling languages and tools: A call to arms. In *Proceedings of the IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation*, Columbia, Maryland, USA, September 2005.

[82] M. K. Zimmerman, K. Lundqvist, and N. Leveson. Investigating the readability of state-based formal requirements specification languages. In *Proceedings of the 24th International Conference on Software engineering*, pages 33 – 43, Orlando, Florida, May 2002. ACM Press.