# Certifiably Safe Software-Dependent Systems: Challenges and Directions*

John Hatcliff
Kansas State University
United States

Alan Wassyng
McMaster University
Canada

Tim Kelly
University of York
United Kingdom

Cyrille Comar
AdaCore
France

Paul Jones
US Food and Drug Administration
United States

## ABSTRACT

The amount and impact of software-dependence in critical systems impinging on daily life is increasing rapidly. In many of these systems, inadequate software and systems engineering can lead to economic disaster, injuries or death. Society generally does not recognize the potential of losses from deficiencies of systems due to software until after some mishap occurs. Then there is an outcry, reflecting societal expectations; however, few know what it takes to achieve the expected safety and, in general, loss-prevention.

On the one hand there are unprecedented, exponential increases in size, inter-dependencies, intricacies, numbers and variety in the systems and distribution of development processes across organizations and cultures. On the other hand, industry's capability to verify and validate these systems has not kept up. Mere compliance with existing standards, techniques, and regulations cannot guarantee the safety properties of these systems. The gap between practice and capability is increasing rapidly.

This paper considers the future of software engineering as needed to support development and certification of safety-critical software-dependent systems. We identify a collection of challenges and document their current state, the desired state, gaps and barriers to reaching the desired state, and potential directions in software engineering research and education that could address the gaps and barriers.

## Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; K.7.3 [**The Computing Profession**]: Testing, Certification, and Licensing

## General Terms

Verification,Standardization

## Keywords

Certification, safety, assurance, standards, verification, validation, requirements, hazard analysis

*Work supported in part by the US National Science Foundation (NSF) (#1239543), the NSF US Food and Drug Administration Scholar-in-Residence Program (#1355778) the National Institutes of Health / NIBIB Quantum Program, the US Air Force Office of Scientific Research (AFOSR) (#FA9550-09-1-0138), the Ontario Research Fund, and the Natural Sciences and Engineering Research Council of Canada.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*FOSE* '14, May 31 - June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2865-4/14/05 ...$15.00.

## 1. INTRODUCTION

The use of software in critical systems that impact daily life is increasing rapidly. New generations of medical devices, automobiles, aircraft, manufacturing plants, nuclear power generating stations, automated trains, banking and investment systems, manufacturing systems, and a growing number of automated systems within our homes rely on software to enable new functions, provide pre-existing functions more efficiently, reduce time to service a user need, and reduce effort and competence required by people providing services. Increasingly, independently-developed systems are being coupled more closely than before, as manifested in air traffic control, road traffic support, emergency response, healthcare-delivery, globalized trading of securities, the power-grid, and telecommunications. As society seeks value-addition through sharing of information and industry responds with increasing inter-dependence of resources (*e.g.*, through the *internet of things* [18]), the growth is accelerating with increasing severity of consequences. Often, the extent of inter-dependence is not even known. In most cases, a hazard-free result cannot be completely assured and the residual hazard space is not known.

In the automotive space, software affects automobiles critically [9], with upwards of 80 computers on board, and tens of millions of lines of code. The role of software now encompasses control of the engine(s), steering, braking, cruise control, doors and windows, entertainment – and all active safety functions such as stability control, blind spot detection and adaptive cruise control. Various degrees of autonomous driving have been demonstrated [68], and the concept of *connected vehicles* [65] is emerging. The car of the near future will be filled with safety-critical software on a scale we would not have anticipated just a short time ago. Car manufacturers and their suppliers are already struggling with the task of building safety-critical software so that their hard-won standards of safety can be maintained [3]. Continuation on this trend line will result in the contribution of the software-dependence to the hazard space not being analyzable with current technology.

In avionics, the use of embedded software has grown rapidly over the last 25 years as analog systems have morphed into digital systems [52]. Among recent generation aircraft, the Boeing 787 has more than 8 million lines of code, or about four times the total in the 777. The code, which has increased during the aircraft development effort, "determines everything from how information is shared between systems to how the pilot's commands are translated into action; from how humid the air is to when the coffee is done," according to Boeing. The number of software systems and their supply-sources of systems are also increasing rapidly. A Wind River Systems manager notes that there are over 70 applications from 15 different vendors running on the 787's Common Core computer [52]. In military avionics, an Air Force general characterized the importance of software by stating, "The B-52 lived and died on the quality of its sheet metal; today our aircraft will live or die on the quality of our software." [20].

In the medical space, software is a core component in radiation therapy machines, robotic surgery, infusion pumps, pacemakers, medical imaging machines, and a wide range of devices used in the home and hospitals for measuring temperature, blood pressure, pulse and physiological parameters. In addition, software is at the heart of hospital information systems, electronic health records and patient treatment plans. There is no doubt that these devices and systems save lives – but when they fail they are also responsible for causing injuries and death. Unfortunately, many of these devices are not sufficiently dependable for their intended use. For example, in the first half of 2010, the FDA issued 23 *Class 1 recalls* of defective devices – and 'Class I recalls' are categorized by the "reasonable probability that use of these products will cause serious adverse health consequences or death", [72]. According to the Software Freedom Law Center, at least six of the recalls were likely caused by software defects [64]. Even when software issues are not life-threatening, they can be disruptive. For example Forbes reports that a computer virus that infected catheters in a lab run by the Department of Veterans' Affairs required that the machines be turned off and patients transported to a different hospital that could continue their care. As if this were not enough of a challenge, manufacturers have now added generic wireless capability to many safety-critical devices. This has led to concerns about security related issues, such as privacy of patient data, as well as the safety consequences of hacking into devices such as insulin pumps [51].

## 1.1 Adverse Socio-technical Trends

Though each safety-critical domain has unique challenges, the following issues are common to multiple domains, and make it much more difficult to develop and assure critical systems.

**Advances in Technology:** Vincenti [76] distinguishes between *normal* design and *radical* design. Normal design is well-understood, well-structured and utilizes well-accepted design principles, while radical design does not, 'unleashing creativity' in the quest for rapid or breakthrough innovation, but increasing the engineering-contributed hazard space.

**Increasing Integration, Scale and Complexity:** Most safety-critical systems in every application domain are growing in both scale and complexity, and in the proportion of system functionality allocated to software or other implementations of complex logic. A number of factors are fueling this increase, in particular, the (sometimes unnecessary) *integration of devices/systems*. The commensurate increase in possible interactions between systems' elements, number of details that must be accounted for, and the number of teams/people to which requirements and verification and validation objectives must be communicated all make development and certification much more challenging.

**Changes in the Way Systems are Built:** Systems are increasingly being engineered by distributed work forces in diverse geographical locations with different cultures and languages. Systems are increasingly built from off-the-shelf components across longer supply chains, using different paradigms and quality management approaches, exacerbating difficulties of assurance and accountability.

**Need-supply Gap in Engineering Capability:** The trends described above have created new engineering needs in which the old work force is not experienced. Where the old experience is still valuable, the experienced engineers are retiring faster than the transfer of their knowledge to newer engineers [17]. Although the need in industry is increasing, the supply of qualified engineers is not, and the criticality of capability-requirements is not understood; the gap is being filled with less qualified people. In addition, many senior decision makers in companies and regulatory agencies do not have the required software engineering background to realize that their knowledge may be inadequate as a basis for directing the development or certification of software intensive safety-critical systems. This is exacerbated by the fact that there are few university software engineering courses that address relevant topics in safety-critical software development or evaluation.

## 1.2 Goals, Scope, and Organization

In many of the systems mentioned above, inadequate engineering can lead to economic disaster, injuries or death – characterized in this paper as the *engineering-contributed hazard space*. Most of the time, society does not recognize the potential of losses from deficiencies of systems due to software until after some mishap. Then there is an outcry, reflecting societal expectations and distrust of the supply-side market forces; however, few know what it takes to achieve the expected safety. To address the inability of the average person to grasp the intricacies of safety-engineering, society's confidence in safety-critical systems is typically established by government regulation, including requirements for licensing and certification (explained in Section 2.1). Given trends presented above, we are struggling not only to engineer safety-critical systems but also to gain society's trust through appropriate certification regimes.[1]

This paper addresses future needs of software engineering research and education to realize certifiably safe software-dependent safety-critical systems. Section 2.1 reviews foundational definitions used in this paper to discuss this subject. Section 2.2 contrasts the development of safety-critical certified systems with that of mainstream systems (*e.g.*, business processes; entertainment), to which current software engineering curricula cater. Sections 3 – 9 identity critical areas in which advances are needed. For each area, we summarize the challenge facing the community, the current state, the desired state, gaps/barriers, and some promising future directions (based on the demonstrated strengths of the software engineering community and discipline) to address these critical needs. A website for this paper [24] provides an extended version of this paper, along with supplementary documentation, presentations, and links to resources referenced in this paper.

Lutz [49] and Heimdahl [27] have provided good summaries of challenges related to software engineering for safety-critical sys-

[1] The assurance of safety implies security. However, the length constraints on this paper limit the scope to common engineering deficiencies, rather than the open-ended safety-related issues that might result from malicious intrusion.

tems in previous ICSE Future of Software Engineering tracks. While many of the challenges that they identified are still valid and overlap with some of our themes, our theme selection focuses to a greater extent on issues in *certification* of software-dependent critical systems.

# 2. SOFTWARE CERTIFICATION IN CONTEXT

## 2.1 Terms and Definitions

To lay the foundation for subsequent discussions, we review definitions related to safety and certification as used in this paper. A *system* is a combination of interacting elements organized to achieve one or more stated purposes [73]. In our discussions, a system will typically include both software and hardware. A *stakeholder* is a person, team, or organization that has an interest in the successful development and deployment of the system, or, may be affected by the system.

**Safety:** *Safety* is freedom from harm (generally meaning injury or death). Although the paper is focused on safety, most of the reported findings can be applied to critical systems in general by expanding from 'harm' to 'loss', including damage to the environment, property, information, or economic loss. A *hazard* is an intrinsic property or condition that has the potential to cause harm or damage [73, 38]. While there are different forms of hazards that might be considered in safety reviews (*e.g.*, development hazards such as inadequate definition of the boundary of the system being analyzed, inadequate flow-down to identify requirements and constraints on technical processes [73]), in this paper we focus our attention on *system hazards*. *Hazard identification* is the process of recognizing that a hazard exists and defining its characteristics [38]. System hazard identification typically views hazards as having two dimensions: (1) a system state(s) that has the potential to cause harm, and (2) the particular environment conditions that must exist for the hazardous state to lead to harm [48]. For example, for a passenger rail car, a hazard may exist where the door of the car is in an open position (system state) and the train is traveling at high speed with a person on the train falling towards the door (environment condition). Both the system state and environment condition must be present for harm to occur [48]. *Safety-critical systems* are those systems whose failure could result in harm. *Safety-critical software* is any software that can directly or indirectly contribute to the occurrence of a hazardous system state [47].

**Safety Deficit:** The definition of safety above is absolute – *freedom* from harm – but very few realistic safety-critical systems are 'completely free from harm'. Therefore, it is common in some domains to introduce characterizations and metrics pertaining to the degree to which the safety falls short of being absolute – the *safety deficit*. One such notion is *risk* – the combination of the likelihood of harm and the severity of that harm [71].

**Reliability:** *Reliability* is the ability of a system or component to perform its required functions under stated conditions for a specified period of time [38]. A *failure* is the termination of the ability of a product to perform a required function or its inability to perform within previously specified limits [38].

**Safety vs. Reliability:** It is a common misconception that safety is equivalent to reliability or that reliability implies safety. Careful consideration of the definitions above reveals that this is not the case. For example, an aircraft whose engines fail to start may be perfectly safe because it can cause no harm, but it is not reliable because it fails to perform its intended function. Conversely, a nuclear power plant may be reliable in that it generates power as specified, but unless relevant hazards were identified and mitigated as part of its design, it may harm its operators or the environment through radiation leaks. The distinction between safety and reliability motivates the notion of a distinct system safety architecture [77] – which is sometimes, but not always, adopted in safety-critical systems. In this approach, functions for monitoring the occurrence of hazards, and then mitigating them are separated from the rest of the system's normal function into a specific and dedicated set of components (the safety architecture) to facilitate easier review and assurance of the safety features of the system. In safety engineering, hazards are the basic unit of management – one tries to determine all of the hazards that are theoretically possible (where the process is usually driven by multiple forms of iterative analyses (see, *e.g.*, ARP 4761 [67]), and then design a system where they are, if not impossible, then at least very unlikely. In reliability engineering, failures are the basic unit of management – one tries to design the system so that failures are unlikely or using techniques (*e.g.*, fault tolerance techniques) that help ensure that the system will perform its intended function in the presence of faults/failures. In practice, safety and reliability engineering are intertwined in safety-critical system development, and techniques for reasoning about hazards, their causes and their impact overlap with techniques for reasoning about failures and their impact. However, keeping the distinction between safety and reliability clear is crucial when developing and assuring safety-critical systems.

**Verification and Validation:** Verification and Validation (V & V) of a system in the scope of this paper are processes that provide evidence for 'conformity assessment', which is a term used in supporting the definition of certification, given below. *Verification* is confirmation that specified requirements have been satisfied. *Validation* is confirmation that the system satisfies the needs and expectations of the customer and other identified stakeholders. Informally, validation is often characterized as answering the question "Are we building the right system?", whereas verification is characterized as answering "Are we building the system right?". The activities of validation and verification both depend on properly written requirements. Assuming that verification is achieved, in theory to achieve validation it is only necessary to confirm that needs/expectations of the stakeholders are properly captured in the requirements. In practice, validation often includes a number of activities such as system testing against a test suite known to reflect the operational characteristics of the system's context and comparing the behavior of the system to an executable model of the system's behavior.

**Assurance:** Ultimately, the stakeholders of the system need to have confidence that the various claims made about the system are correct. We use the term *assure* to denote the task of confirming the certainty of a claim based on evidence and reasoning such that the evidence and reasoning used can be examined and assessed by an independent party [73]. A *claim* is a true-false statement about the value of a defined property of a system, where a property is a quality attribute of the system such as safety, security, *etc.* In this paper, we limit the scope of usage of these terms to the assurance of the system property 'safety' and to any supporting claims about safety properties, *e.g.*, 'software assurance' in support of system safety assurance. In general, the concepts discussed for safety assurance will also be applicable to security assurance.

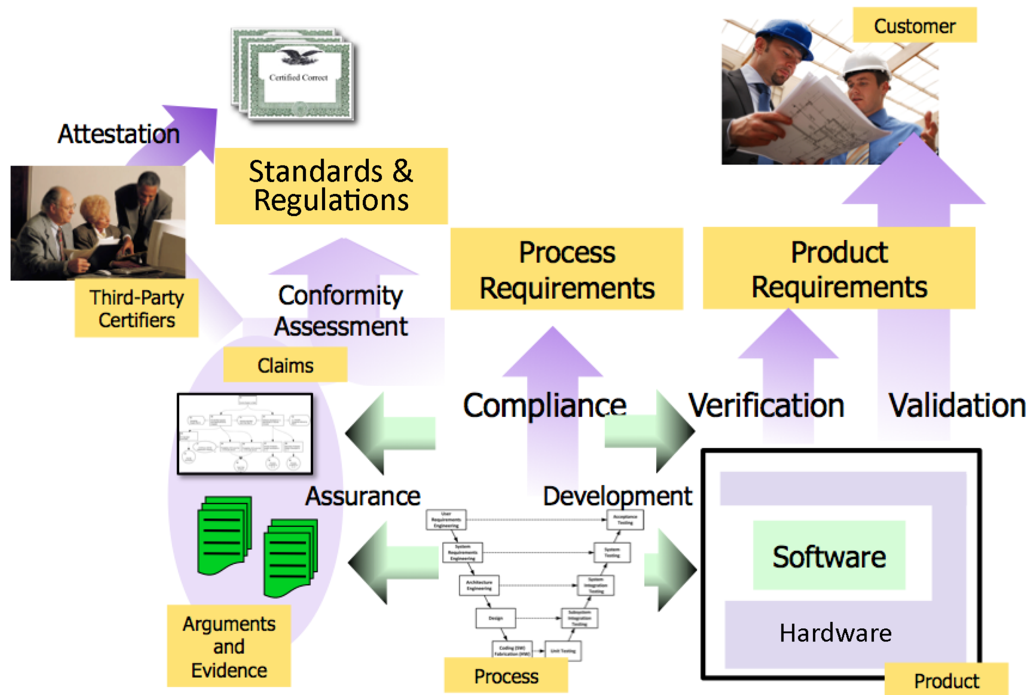


**Figure 1: Certification Concepts**

**Certification:** *Certification* is third party attestation related to products, processes, systems or persons [39]. In this paper, discussion of certification is limited to software-dependent safety-critical systems. A *third party* is a person or body that is independent of the person or organization that provides the object (product) of conformity assessment, and of user interests in that object [39]. Confidence in the certification process depends, in part, on the competency of the third party.

## 2.2 Contrasts with Conventional Software Engineering

One of the key barriers to facilitating research in safety-critical certified systems and to immediately transitioning software engineers into engineering positions in the safety-critical industry is that conventional software engineering does not address many vital needs for certifiably safe software-dependent safety-critical systems.

**Stakeholders:** Conventional training in requirements development focuses on stakeholders such as the software provider (first party), customer (second party), and user. In the safety-critical space, additional stakeholders such as third-party certifiers, regulators, and society at large (*e.g.*, via the legal system) generate requirements that drive development to a significant degree. In particular, the provider-customer cannot settle for a mutual implicit understanding based on implicit information that the system is 'safe enough'. The criticality demands independent assurance (*e.g.*, via third-party certifiers and regulators).

**Safety Goals:** Conventional training on writing goals in requirements development is typically limited to correct capture of desired system function, system evolution and maintenance, and perhaps appropriate management of costs. In the safety-critical space, goals include several extra dimensions. Early stages of development identify notions of *loss* and *mishap* relevant to stakeholders. These notions drive explicitly recognized development process iterations of requirements writing, hazard analysis, and design that prioritize eliminating or reducing hazards and the risk of loss. In addition, the system design process typically must identify failure mode goals (*e.g.*, fail-operational, fail-stop, *etc.*) that indicate how the system should behave under catastrophic failure conditions.

**System Context to Software Context:** Conventional training in software requirements often focuses on software service applications that typically involve no notions of hardware/software interfacing, nor do they consider a total systems perspective in development. In situations where a hardware or larger system context exists, the conventional software engineering perspective takes a narrow view of its role by declaring that "system engineering is responsible for developing the requirements that its software should satisfy." This narrow view leaves a communication gap between those who know the application domain and those who know software design and implementation. In safety-critical systems, software requirements are derived from and must trace to system requirements. In this context, the communication gap mentioned above can lead to disaster, and software engineers must understand the application domain, including the criticality of the functions, and be capable of eliciting, organizing, and validating the information from the domain specialists and the system engineers.

**Hazard Analysis:** One of the most popular forms of analysis addressed in the software engineering research community in recent years is *automated*, *static* analysis of *source code* to locate possible faults such as null pointer dereferences, array bound violations, and misuse of resources, libraries, *etc.* In the safety-critical space, a significant portion of the development process is devoted to multiple forms of hazard analyses that aim to identify hazards, their possible causes, and their possible effects (*i.e.*, mishaps that they

might cause). Hazard analysis differs from static analysis in a number of dimensions. Rather than being applied rather late in the development process to implementation artifacts (*e.g.*, source code), different forms of hazard analysis are applied *throughout* the development process. For example, in ARP 4761, Functional Hazard Analysis (FHA) is applied very early in the development process to analyze the primary functions of the system (working from an abstract and informal description of the function's behavior) to identify hazards that might result when a function fails. Subsequently, the Preliminary System Safety Assessment (PSSA) phase of ARP 4761 applies Fault Tree Analysis (FTA) to design and architecture development to explore how hazards identified in FHA might be caused by failures of hardware and software components now identified in the design/architecture. Farther along in development, the ARP 4761 System Safety Assessment (SSA) phase includes Failure Modes and Effects Analysis (FMEA) to systematically identify failure modes of implementation components, the local effects of those failure, how errors associated with failures might propagate to the system boundary giving rise to hazards. These analyses may be 'top-down' (given a hazard, systematically drill down and identify possible causes at different levels of abstraction of the system) or 'bottom-up' (given individual components in an architecture, reason about how the effects of failures of the component propagate up through the architecture to possibly contribute to hazards). Rather than working with machine readable artifacts (*e.g.*, source code), hazard analysis must often be applied early in the development process to informal textual descriptions (*e.g.*, in FHA) or to architectures (*e.g.*, FTA or FMEA), where architectural descriptions used are not stated with the formality or precision necessary to support automation. Rather than propagating information along paths (*e.g.*, data flow or control paths) that can be automatically extracted from source code, many of the sources of information in a hazard analysis (*e.g.*, the behavior the components may exhibit when failing) and the paths along which failure effects propagate (*e.g.*, propagation due to proximity of components or interactions through the external environment) are not specified in development artifacts – they are identified manually by analysts based on training and experience. Moreover, the analysis requires identification of hazards from *unintended* behaviors rather than *intended behaviors*, which by their very nature cannot be extracted from conventional development artifacts that describe only the *specified* behavior of systems.

**Hazard-Based Safety Engineering:** As hazards are identified through hazard analysis, they are documented and their impact upon development and safety assessment is managed. While bug tracking, prioritization of bugs, and documentation of fixes, *etc.*, are somewhat related activities in software engineering, the importance and impact of hazard tracking in the safety-critical space is much greater. Hazards are initially recorded in a log – the title of which varies across domains. This document is subsequently revised to record a safety deficiency metric for the hazard. Many domains use the notion of risk for this, but as we note in Section 3, the scientific basis for this is questionable. Depending on the safety deficit associated with the hazard, designers may produce a modified design to *eliminate* a hazard. Deficits associated with remaining hazards are typically mitigated by choosing one of the following strategies – each with decreasing effectiveness. Hazards may be *controlled* through design modifications (*e.g.*, the inclusion of a safety interlock) or *detected* and operators notified. Finally, in some cases, the only action taken may be to train operators to recognize or prevent hazards during operation and respond appropriately. Updates to the design and architecture may introduce new hazards, so the process must be iterated.

**Architecture Design and Analysis:** In conventional development, most architecture design and assessment focuses on the abstracting, isolating elements that change more often, managing complexity, and supporting evolution and maintenance. In safety-critical development, additional forms of analyses are applied to the architecture as it evolves to reason about failures, effects of failures, propagation of faults, identification of elements whose failure may cause multiple forms of failures (*i.e.*, common cause analysis), identification and causes of hazards. Results are used to (a) generate safety requirements, (b) guide designs toward providing redundancy to increase reliability and fault tolerance, and (c) drive design to eliminate or mitigate hazards. In particular, safety-critical system design often emphasizes constructing a distinct *safety architecture*, often partitioned from primary system function, that detects and mitigates hazards, has greater resiliency, and is designed for easier verification and assurance.

**Assurance:** Typical software engineering activities focus on verification, and perhaps to a lesser extent validation. However, they seldom address the need to externalize evidence and arguments as part of the assurance process to support third-party certification. There is little emphasis on assurance cases and other related activities in software engineering curricula.

**Standards and Regulatory Compliance Goals:** Conventional development may include process goals such as complying with coding guidelines, etc. In the safety-critical space, there are much more stringent process requirements to guarantee compliance with standards (*e.g.*, DO-178C in the avionics domain) and regulatory guidelines. Moreover, the amount of documentation related to demonstrating compliance with process requirements is significantly greater. Unfortunately, almost no graduates of computer science and software engineering programs have ever seen or read a safety standard, much less applied it to a product.

## 3. DEVELOPING FOUNDATIONAL PRINCIPLES

**Challenge:** There are many standards that either directly or indirectly address safety-critical software. These have emerged through the standards-making process with differing philosophies (*e.g.*, regarding the treatment of software criticality) and differing requirements (*e.g.*, regarding the recommended use of certain software development techniques, such as formal methods). The presence of so many standards, and the differences between standards, can be bewildering to anybody operating in the domain, and can be a significant barrier in the education of new practitioners and researchers. In addition, the differences between standards make it hard to translate evidence of compliance between standards. It is desirable to establish foundational principles that provide a common basis for software safety assurance across domains and applications.

**Current State:** Many differences can be observed in the details of current software safety standards. For example, in civil aerospace, criticality of software is determined largely by consideration of the severity of the hazards to which the software may contribute. In other standards, such as IEC 61508, criticality is assigned to the embedded software system according to the degree of risk reduction required (expressed as a probability of failure per hour, or probability of failure on demand). Other standards (*e.g.*, ISO 26262 in the automative domain) use a mix of both the severity and probability of an accident sequence involving the software system to

determine criticality. DO-178B/C uses the determined level of criticality (expressed as a Development Assurance Level) to moderate the number of explicit assurance objectives to be addressed. IEC 61508 adopts a different approach, and uses the assigned criticality (expressed as a Safety Integrity Level) to guide the selection of software design features (*e.g.*, use of defensive measures such as recovery blocks) and assurance techniques (*e.g.*, different types of testing). Underlying such differences, however, it is also possible to observe some common themes in how software safety assurance is addressed. These can be expressed in terms of the following five principles (previously presented in [26]):

1. Software safety requirements shall be defined to address the software contribution to system hazards.

2. The intent of the software safety requirements shall be maintained throughout requirements decomposition.

3. Software safety requirements shall be satisfied.

4. Hazardous behavior of the software shall be identified and mitigated.

4+1. The confidence established in addressing the software safety principles shall be commensurate to the contribution of the software to system risk.

*Principle 1.* Software by itself cannot be safe or unsafe. It is only when placed in a system context that the 'safety' of software can be judged by considering the contribution that the software could make to system level hazards. The first challenge of software safety assurance is to identify these contributions and to capture the necessary behavior of the software in relation to these contributions in terms of a clearly defined set of software safety requirements at the system-software boundary.
*Principle 2.* The normal process of software development proceeds by decomposition of the high level requirements placed on the software into lower level specifications of behavior that ultimately can be implemented. This may take place as a structured (but informal) process, as part of a formal development process, or be supported by the development and transformation of models. Regardless of approach, a key concern is whether the *intent* (the stakeholders' view of the behavior they want/need) of an original requirement is maintained throughout the process of decomposition. Systematic (design) errors are introduced whenever there is a misalignment of the original intent of a requirement and its implementation.
*Principle 3.* Ultimately, it is necessary to demonstrate that any safety requirements allocated to software have been satisfied. It is important to present evidence that shows the satisfaction of safety requirements under anticipated operating conditions. This requires the presentation of evidence that addresses satisfaction under both normal and abnormal (fault) conditions.
*Principle 4.* Potentially hazardous emergent behaviors can result from well-intentioned but (in hindsight) flawed design decisions that, unfortunately, have unintended hazardous side effects. In addition, implementation (process execution) errors can be made during the software development process – *e.g.*, modeling errors, coding errors, and tool-use errors. It is necessary to ensure that assurance effort has been targeted at attempting to reveal both of these sources of errors.
*Principle 4+1.* This principle underpins the implementation of the first four principles (hence being termed *+1* rather than 5). Perfect assurance of the other four principles is, of course, desirable but in reality is unachievable. For example, it is impossible to prove that all software safety requirements have been identified. Consequently, it is necessary to consider how much effort to expend in addressing the first four principles, and how much evidence to generate. This principle states that the level of evidence (and confidence) needs to be proportional to the level of risk associated with the software in question.

**Desired State, Gaps and Barriers to Progress:** Whilst there is broad agreement on the principles described in the previous section, there still exists significant discussion and disagreement on the *implementation* of these principles. For example, there is much debate on the nature and desirable level of prescription in the criteria (*e.g.*, regarding the software development and assurance processes) used within standards. Similarly, there is debate as to whether to require the production of an explicit assurance case that justifies the satisfaction of these principles (for a specific project) or be content that an implicit assurance case exists through the satisfaction of the criteria of a standard. These issues are discussed further in Section 4. Principles 1 to 3 emphasize the key importance that the elicitation, validation and decomposition of safety requirements has in software safety assurance. However, there remains significant debate about the best means of capturing, expressing and managing safety requirements. This issue is discussed further in Section 5. With the growing complexity of today's safety critical software the challenge of identifying potentially hazardous emergent behaviors (as highlighted in Principle 4) is significant. The application of manual techniques (such as the application of HAZOP on software design) becomes infeasible for complex systems, and automation assistance is desirable. This issue is discussed further in Section 8. The use of tools to support the development and verification of software can underpin the implementation of Principles 1 to 4. For example, tools supporting model driven engineering can assist in the mapping and transformation of models through the development process to assist in the implementation of Principle 2. Verification tools can assist greatly in the demonstration of satisfaction of behavioral safety requirements. They can also assist in demonstrating the absence of potentially hazardous implementation errors, and the identification of hazardous software behavior (Principle 4). However, for tools to be used in this way they need themselves to be appropriately designed, assessed and qualified. It is necessary to assure that tools cannot be the source of error introduction in the software development process. This issue is discussed in Section 7. Principles 1 to 4 apply whether software is developed in a monolithic fashion or using principles of composition and component-based engineering. However, compositional approaches can present specific challenges in implementation - *e.g.*, the identification of safety requirements (Principle 1) when potentially hazardous behavior is distributed across software modules, the composition of verification results (Principle 3), or the identification of emergent behavior from composition (Principle 4). Such issues are discussed in Section 6.

The implementation of Principle 4+1 is perhaps one of the most contentious issues in software safety assurance - i.e. how to moderate the level of evidence required in software safety assurance according to the level of risk posed by the software. Firstly, one of the main issues of debate in this area is the notion of risk applied to software. To know the risk associated with a software system requires knowing the severity of the hazards to which the software may contribute, the probability of these contributions, and the probability of the hazards given the contributions. Many consider the probability of software failure to be an unobtainable parameter, or at the least one that is extremely difficult to obtain. Software does not fail randomly, however in theory we can reason about the prob-

ability of revealing systematic (design) errors at run time. In practice, however, a priori estimation of this probability is very challenging.

Some certification bodies (*e.g.*, [73]) deliberately exclude treatment of probability in estimating the criticality of hazardous software contributions, basing judgement solely on severity. Other standards (*e.g.*, IEC 61508) avoid the problem by reasoning about the probability of failure *required* rather than achieved (although this still leaves a question as to whether this requirement has been achieved). This raises the second key issue, namely how to provide a level of assurance (confidence) that is commensurate with the criticality of the software. Whilst the principle is easily stated, specific judgements of how *deficits* in assurance relate to risk are hard to make, context specific, and often lacking an empirical basis. This issue is discussed further in Section 4.

Finally, the lack of agreement in the implementation of the principles can be seen as a barrier to building competence in the engineering of safety critical software. For example, it is challenging to define an educational syllabus when there exists significant variations in philosophy and approach across the standards. This issue is discussed further in Section 9.

**Research Directions:** As described in the previous section, the implementation of foundational principles raises many research challenges. The research challenges in each of these areas are discussed in the referenced sections. However, underpinning all of these challenges remains the problem of gaining consensus on best practice in safety critical software development and assurance. At present many of the debates concerning the adoption of one approach or another can appear philosophical and/or subjective. Therefore there is growing interest in attempting to establish a more empirical basis for such discussions [59], *e.g.*, regarding the efficacy of certain standards. In addition, there is increasing research into cross-domain certification, such as the European funded OPENCOSS (Open Platform for EvolutioNary Certification of Safety-critical Systems) project [54]. Such projects are attempting to tackle the problem of mapping and cross-referencing the varying terminology, objectives, required activities, and required evidence of the different safety standards. Ongoing work such as this will contribute to establishing an increasing understanding of the commonality and variability in current safety-critical software development practice.

# 4. THE NATURE OF CRITERIA IN SAFETY CERTIFICATION

**Challenge:** Standards play a key role in providing uniform community-developed assessment criteria for systems within a safety-critical domain. What is the appropriate use of standards in safety certification of a software-dependent safety-critical system? What are the necessary and sufficient criteria that should be required in those standards? What specific processes and products should these criteria address? Surprisingly, these questions remain challenges, especially, in the case of highest-criticality systems [63].

**Current State:** Many acquirers of software-dependent safety-critical system and regulatory authorities depend upon compliance with international standards such as IEC 61508 [33], ISO 26262 [36], EN 50128 [14], DO-178C [60], IEEE 7-4.3.2 [31] and IEC 62304 [32]. Intrinsic to the concept of *standard*, is the idea that arbitrary variances in criteria and assessment of conformity to criteria are reduced by adopting *uniform* approaches agreed to by a broad collection of stakeholders. The challenge lies in the fact that the ultimate goal is to assess the safety attributes of *particular* systems, but systems and their safety attributes (*e.g.*, notions of harm, hazard, mitigation strategies, assurance arguments) *vary*. While the variance is smaller in some domains (*e.g.*, in avionics, most airplanes have similar notions of harm and hazard), variance is profound in domains such as medical devices. Given the variance, it is impossible for a standard to *uniformly* and *directly* specify sufficient safety requirements for all the systems to which it is intended to be applied. Therefore, standards must define criteria that *indirectly* address system-specific safety requirements.

*Process-based Approaches:* The most common approach is for a standard to specify criteria for the *processes* or process characteristics that are used to support the creation and evaluation of a system (the idea being that while the specific safety-related attributes may vary across systems, the processes by which they are identified and addressed can be relatively uniform). Standards and associated certification regimes that follow this approach are often termed *process-based* [50]. For example, such standards may establish criteria for clearly identifying system hazards, establishing a clear link between software safety requirements and system hazards, the traceable decomposition of requirements, and the verification of the product implementation against safety requirements identified for the system. Such standards also address in a general manner the problem of systematic error introduction within the software development lifecycle and recommend techniques for the revelation and mitigation of such errors. In these standards, compliance is often judged by confirming the existence and completeness of process-related artifacts, *e.g.*, have identified hazards been documented along with associated mitigation strategies (*e.g.*, [35]), is there a test plan (*e.g.*, [32]), *etc.*, rather than judging the quality of the artifacts and the degree to which they directly support claims/arguments for the system's safety. Although process-based criteria can provide useful bodies of knowledge and may codify best practices that, if followed correctly, tend to increase the likelihood of safety, conformance to such criteria may be neither completely necessary nor sufficient to establish that the product is certifiably safe. The implementation of processes and specific enactment (and interpretation) of these objectives can fall short on a given project. Often, users of these standards do not recognize that the process-based criteria merely define a minimum requirement. There is growing agreement within academia and industry that software assurance must go further than demonstrating compliance to process-based criteria and directly focus attention on the product itself [63, 57].

*Argument/Evidence-based Approaches:* Instead of establishing criteria that indirectly address safety via the proxy of process, there is increasing interest in establishing criteria for presenting arguments and evidence for claims about the system, including claims of safety (the idea being that while the specific safety-related attributes may vary across systems, the manner in which one presents arguments and evidence can be relatively uniform). An *assurance case* records explicitly a structured argument (reasoning), supported by a body of evidence, that substantiates a claim about the properties of the product in a given application and given environment [37]. Assurance cases do not preclude the application of process-based criteria. The contribution to safety of activities and artifacts produced for process-based criteria is often implicit. Assurance cases allow manufacturers to explicitly document how results of process-based activities contribute to and provide evidence for claims of safety [29]. Once explicit, these arguments of assurance can more easily be evaluated, criticized, and improved. Assur-

ance claimed predominantly from the identification and demonstration of attributes, properties and behaviour of the product is often termed *product-based* [79]. Although standards with process-based criteria predominate, an increasing number of standards and assurance regimes require the production and evaluation of assurance cases, such as the automotive domain ISO 26262 [36], railway domain EN 50128 [14], and defence (in the UK) [69].

**Desired State, Gaps and Barriers to Progress:** When developed correctly and applied in conjunction with process guidance, assurance cases can serve the purpose of certifiable assurance better than the common current practice of relying on process-oriented criteria alone. To begin with, the very requirement for an assurance case forces engineers to think about the specific safety claims of their system and the specific nature of arguments and evidence supporting those claims. More broadly, assurance cases can logically integrate all of the applicable and relevant objectives (e.g. regarding requirements validation) and recommended techniques (e.g. specific forms of testing) stated in standards, and in addition presenting compelling arguments for the specific enactment of process criteria (*e.g.*, presenting specific arguments of requirements validity, or the adequacy of selected verification approaches). Although the assurance case approach holds promise for the future, there are some known weaknesses and knowledge gaps.

*Inconsistency in Quality and Structure:* The generic nature of assurance cases is both the major weakness of this approach and its most significant strength. The weakness is that there are many possible forms of an assurance case – some good, some bad. For example, it is possible for someone to build an assurance argument that a system is safe by merely claiming that processes called out in a process-oriented standard were followed. This failure to introduce claims related to the safety attributes of the particular product being certified is one the specific pitfalls that assurance cases were designed to address. Whilst the requirement for safety cases exists in several widely-used system safety standards, there are few standards that (a) specifically address the production of *software* assurance cases, and (b) provide guidance on strategies and patterns for producing effective assurance cases. There is an ISO standard for assurance cases [37], but it focuses on the generic features, notations, and terminology of assurance cases. Without guidance there can be a wide variation in assurance case structure and evidence — making it difficult for third parties (*e.g.*, regulators) to make consistent acceptance decisions [79].

*Inadequate Methodologies for Integrated Development:* A frequent misuse of assurance cases is to postpone their construction until after system development is completed. Instead, best practice in assurance case development (*e.g.*, as presented in [69]) is that the case is developed in step with system development, such that it helps direct the development process to meet the safety and security goals effectively and efficiently. For example, independent team reviews can expose fallacies in reasoning, invalid assumptions, gaps in the evidence and sources of uncertainty. Then, corrective action items can be formulated to address the exposed weaknesses. Better methodologies are needed to integrate assurance case construction with system development.

*Inadequate Approaches for Managing Confidence:* The issue of managing confidence remains a significant challenge in assurance case production, *i.e.*, how to manage the level of confidence required in the arguments and evidence of an assurance case according to the declared level of acceptable safety deficit [19]. Although the assurance case concept [37] provides a structure to relate uncertainties and qualifiers, existing knowledge does not support meaningful evaluation of the contribution of these uncertainties (*i.e.*, the resulting detraction from the assurance of the claimed property). Confidence can be impacted by many factors, *e.g.*, the validity of models used in verification, the coverage of testing, the confidence in tools used, the compentence of the task performers, the adequacy of personnel conducting reviews, the effectiveness of quality requirements (aka non-functional requirements) to support the top-level safety property, and commensurate architectural constraints to satisfy these quality requirements.

*Combining Properties and Evidence to Support Overall Conformance Claims:* Safety-critical system development increasingly relies on using a diverse set of verification techniques including model-based analysis of architectures, static analyses that demonstrate absence of common run-time errors, contract-based deductive reasoning, and conventional testing. Ideally, correct and effective application of each of these techniques should discharge verification obligations so that multiple techniques (and redundant efforts) do not need to address the same properties. However, theoretical frameworks that provide a basis for accumulating coverage and evidence of property satisfaction across multiple forms of property specification and verification techniques are lacking [12, 74, 22], and they would have to be incredibly persuasive for us to abandon current *defense in depth* approaches.

**Research Directions:**
*Assurance Case Patterns:* It has been shown how it is possible to guide the production of software assurance cases (e.g. by means of software assurance case patterns) in such a way as to explicitly address the key principles of software safety assurance [26, 25]. These principles address software safety requirements identification, decomposition, satisfaction, and the absence of contributory hazards such as emergent unanalyzed behaviour. Assurance case patterns or templates should thus enable us to avoid a huge volume of differently structured assurance cases within a single application domain. They could also help to ensure that templates for assurance cases in a domain are used to direct development, and are maintained and refined as the project progresses.

*Reasoning about Uncertainty:* A number of approaches to reasoning about uncertainty are being researched, *e.g.*, reasoning based on a Baconian philosophy in terms of the identification and mitigation of argument and evidence defeaters [80]. However, even for the uncertainty in elemental evidence, significant research is needed in the appropriate underlying measurement science. The ability to evaluate the integrated effect of individual uncertainties also depends upon conformance to certain architectural constraints, such as rules of compositionality and other constraints derived from quality (aka non-functional) requirements, discussed earlier. Research in these topics holds promise, but is a long-term endeavor. Corresponding standards would be needed, but should be attempted only after the results of the identified research mature.

## 5. REQUIREMENTS

**Challenge:** One of the biggest challenges in engineering certifiably safe software-dependent systems is to establish valid requirements, i.e., requirements specifications that are complete, correct, unambiguous, and yet understandable to all stakeholders. It should be no surprise, because, even in mainstream IT, deficiency in requirements is the biggest source of unanticipated cost and delay. Whereas in the mainstream IT world, users 'learn to live with it;' in

life-critical systems, 'living' may not even be an option! Whereas in common commercial and consumer products issues surface through usage experience and may get fixed as and when possible, for certifiably safe systems, a third party should be able to assure system safety before deployment. Requirements Engineering should facilitate the validation needed for such assurance. The computer science and software engineering communities (especially those involved in safety-critical systems) have concentrated much more on the verification of the implementation against its requirements specification rather than the gap between the real needs and the specification.

**Current State:** In the case of certifiably safe software-dependent systems, often, developers use surrogates of the real requirements; for example, regulations, standards, guidelines, and 'good practices.' System-specific requirements engineering is mostly concerned with the functional requirements, focusing on the 'normal' (most common; most well-understood) cases. Current practice in engineering of requirements for system properties such as safety and security is not systematized. This weakness extends to the analysis of these requirements to derive architectural constraints. Common industrial practice uses natural language to express the known requirements. In contrast, leading edge practice has applied mathematics-based disambiguation techniques [78, 75]. The weak start in current practice of the engineering lifecycle also compromises subsequent engineering activities. It precludes use of modern analysis techniques, such as model checking, and older but still useful techniques, such as simulation, in the validation of the requirements. The inherent obscurity of the real intent also precludes drawing the appropriate attention of domain experts. It also precludes mathematical confirmation that the implementation satisfies its requirements specification.

Many industrial organizations use graphic-based design tools that have legitimate advantages in the design phase of the lifecycle. However, these designs are being presented as requirements as a basis for subsequent detailed design and implementation. This practice further increases the gap between real requirements and the surrogates driving the implementation. The added detail and complexity in the design specification further obscures the requirements, increasing validation difficulties.

Also, in spite of the fact that exhaustive testing of a typical system is not feasible, almost all industries tend to rely mostly on testing to verify that the system behaves as specified and that the specified behavior satisfies system properties such as safe and security.

**Gaps and Barriers to Progress:** The processes of elicitation (from the real world), analysis, and validation are weak, in general, even in mainstream IT. As the criticality of the system increases, the significance of off-normal cases (e.g., what can go wrong) increases, but the gap between the engineering need and capability increases even more rapidly. The competence aspect of this gap is addressed in Section 8 and the technology aspect, in the next subsection.

Many safety-critical systems developed today are built on (or derived from or modifications of) previous versions. In spite of the supposed 'head-start' given by the previous system, establishing valid requirements for the successor system is difficult. One major problem is the inadequate quality of the requirements specifications for existing systems. They are all too often imprecise and incomplete, or written at the level of a design specification, and hardly ever include the rationale for each requirement. In common practice, the team responsible for requirements elicitation and specification does not have the competence needed to perform this task effectively. The team has likely been doing this for years, and is oblivious even to modest advances in the field. Even the published advances in the field do not yet achieve the needed level of effectiveness and thoroughness.

The processes of system engineering, safety engineering, and software engineering are not well-integrated. The isolation starts from the competence-building infrastructure (addressed in Section 8) and continues through the vocabularies of discourse, the paradigms, and the tools. For example, requirements for safety should result from hazard analysis. Whereas requirements engineering is considered to be a part of systems engineering, hazard analysis is considered to be a part of safety engineering (often, a different part of the organization). A typical system developer is trained to think about the functions to be realized, not about what can go wrong. Tools for hazard analysis are not integrated with tools for requirements engineering. Inter-dependencies are not identified at a level of granularity to support effective configuration management.

**Research Directions:** The most pressing need is to develop effective methods for validating requirements. First, the gaps identified above across systems engineering, safety engineering, and software engineering should be bridged. Secondly, the gaps across domain experts, system developers, software developers and safety experts should be bridged. *Domain Specific Languages* (DSLs) may be an effective way of achieving this. However, the trend towards DSLs in the model driven engineering community does not seem to address these communication gaps. The potential of domain modeling has been known for a long time. Now, it is much more realizable by leveraging advancements in ontologies, modeling semantic networks, and knowledge representation combined with the use of stylized natural language.

The most severe gaps in knowledge, requiring the most advancement in research, exist in the systematization of the engineering for quality requirements (aka non-functional requirements): Starting from a top-level property such as *Safety* and decomposing it into a model of characteristics and sub-characteristics, such that the satisfaction of the top-level property can be evaluated consistently across different qualified people, teams, or organizations. The ISO 25000 family of standards [34] is a small step in this direction, based on decades of prior research efforts. Related research is needed in transforming these quality requirements into architectural constraints, such that the requirements can be satisfied verifiably. Some research had been undertaken at the Software Engineering Institute in mapping quality attributes into architecture evaluation criteria; however, much more advancement is needed, before the research results can be applied in practice.

Other topics of importance to safety-critical systems include: notions of completeness –identification of all hazards and the associated safety/security requirements, identification of all necessary interfaces to the system, and specified behaviour for the complete input domain for each interface; and automated simulation of the specified behavior.

We need recognition and unification of the multiple roles of the requirements specification: as an effective platform for validation; as support for documenting design behaviour without re-specifying all the behaviour; as a platform for verification, including testing and analysis; as support for recording rationale; and as support to build bi-directional traceability throughout the life-cycle.

## 6. COMPOSITIONAL CERTIFICATION

**Context:** In Section 1, we identified the increasing scale and complexity of systems as an important challenge in developing safety-critical systems. A general engineering principle for managing complexity is to (a) decompose a system into multiple smaller com-

ponents that can be worked with individually through multiple phases of development, and (b) integrate components in later stages of development to form a complete system. Decomposing systems into components can also lead to cost reductions and decreased development time when components are *reused* across multiple systems.

Several important trends in system development now emphasize notions of (de)composition and reuse.

*Systems of Systems:* A *system of systems* is generally understood to be a collection of systems in which each system functions as a stand-alone system, and the systems are integrated to achieve new mission capabilities not obtainable from the collected behaviour of the individual systems.

*Software Product Lines:* Product line engineering is applied when one has a family of similar systems. An effort is made to (a) identify functionality that is common across multiple systems within the family, (b) design and implement components that provide that functionality, and (c) systematically design systems so that common components can be reused across multiple systems within the family.

*Platform Approaches:* Sharing of run-time environment, common services, and frequently-used application components between stakeholders is now a common approach for achieving reuse. With this approach, one need not develop system functionality from the ground up; instead, one focuses on developing application logic using the shared services and application building blocks to produce an application that executes in the provided run-time environment. Smart phone platforms such as the iPhone and Android are prominent examples in the consumer space. This approach may also encourage innovation since it allows people with less capital to enter the market and contribute ideas.

Multiple safety-critical domains are beginning to emphasize the notion of platform. UK Defense Standard 23-09 [1], standardizes interfaces within the Generic Vehicle Architecture, an open source architecture that aims to encourage reuse in military ground vehicles. The Future Airborne Capability Environment (FACE) Consortium [11] emphasizes an open platform approach for interoperability to facilitate software application reuse across civilian and military aircraft platforms. Current activities in medical software aim to develop interfacing and safety standards for safe and secure *medical application platforms* [23] – safety- and security-critical real-time computing platforms for (a) integrating heterogeneous devices, medical IT systems, and information displays via a communication infrastructure, and (b) hosting application programs (i.e., *apps*) that provide medical utility via the ability to both acquire information from and update/control integrated devices, IT systems, and displays.

**Challenge:** To support notions of (de)compositional development such as those described above, the challenge is to develop engineering and assurance approaches that support *compositional certification* and *reuse of certified components* while maintaining the same confidence levels in safety as one would have when assessing complete systems. As Rushby notes [62], "although it is generally infeasible at present to guarantee critical [safety] properties by compositional (or 'modular') methods, it is a good research topic to investigate why this is so, and how we might extend the boundaries of what is feasible in this area." Specifically, research is needed to determine the extent to which individual components could be certified to conform to interface properties in the context of rigorously defined architectures that constrain emergent properties, and the interface properties, architectural principles, and associated assurance would be sufficiently strong so as to allow systems assembled from those components to avoid a full assessment of all component implementations to justify correctness and safety.

**Current State:** Unfortunately, the effectiveness of compositional development strategies is limited in the context of certified systems, because almost all certification regimes for safety-critical domains certify complete systems [61]– not system components. This is driven by the fact that safety is a global, non-inductive property – safety issues often arise due to incorrect context assumptions or unanticipated interactions between components, between software and associated hardware, and between a functioning system and its environment. As Rushby notes, the problem is that conventional design and interfaces are concerned with normal operation, whereas much of the consideration that goes into assurance of safety concerns abnormal operation and the failure of components. Safety assurance addresses the hazards that one component may pose to the larger system, and these hazards may not respect the traditionally defined programmatic interfaces that define the boundaries between components in normal operation [61, p. 4]. Therefore, reasoning about safety, using current practices, is most easily achieved when working with the system as it will actually be deployed.

There are several notions of reuse and pre-cursors to compositionality that currently exist in the context of certified systems. In the security domain, one of the original motivations for the Multiple Independent Levels of Security (MILS) architecture [6] was to promote a commodity market of reusable components for security systems to be certified according to various protection profiles within the Common Criteria [10]. Separation kernels provide space partitioning and time partitioning, and this provides the initial foundation for compositional justifications of security and safety by removing certain common interference modes leading to emergent behaviors. In the avionics domain, guidance for Integrated Modular Avionics (DO-297) describes how a particular architecture supports flexibility, modularity, reusability and interoperability. ISO 26262, which addresses functional safety for road vehicles, includes the notion of 'safety element out of context' (SEooC) which allows the statement of assumptions and guarantees for a particular element (informally, a sub-system) whose development is being carried out by, e.g., a sub-contractor. The FAA Advisory Circular on Reusable Software Components (AC 20-148) provides a mechanism for reusable software component (RSC) developers, integrators, and applicants to gain: FAA 'acceptance' of a software component that may be only a part of an airborne system's software applications and intended functions, and credit for the reuse of a software component in follow-on systems and certification projects, including 'full credit' or 'partial credit' for compliance to the objectives of RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification. The cross-domain functional safety standard IEC 61508, allows the notion of a 'pre-certified' component that can be integrated into a 61508-certified system while reusing some of the certification effort.

In certain domains such as avionics (ARINC 653, and more recently FACE [11]) and automotive (AUTOSAR [4]), interoperability standards have been defined and are implemented by multiple vendors so that systems can be built from interchangeable components from different vendors.

**Gaps and Barriers:** There are a wide variety of barriers preventing compositional approaches to certification ranging from lack of

standardization, scientific basis for reasoning, appropriate infrastructure, and regulatory pathways.

*Lack of Appropriate Interoperability, Architecture, and Platform Standards:* Before one can be concerned about safety in composition, one must be able to get components to interface correctly. The medical domain is behind in this regard. The primary interoperability standard in the medical space, IEEE 11073 [40], has made contributions, primarily in the area of nomenclature and identification of common device functionality, but few vendors implement the standard, there is no systematic cross-vendor interface compliance evaluation, and it is missing important security, quality of service, and safety properties in its interfacing language. Regarding architecture and platform notions, IEEE 11073 does not include a notion of application hosting (which would include accompanying partitioning mechanisms) nor does it include the richer collection of services offered by ARINC 653 and AUTOSAR.

*Limited Engineering Approaches and Ready-to-Use Solutions for Partitioning and Delimiting Emergent Behaviors in Dynamic Contexts:* We do not deal adequately with emergent behaviors – behaviors that are not present or at least cannot be understood well in individual components but only arise when components are integrated. Configuration in existing separation kernels is largely static and does not provide significant support for applications or resource components to join the system during operation. In a vision for medical application platforms [23], new applications and devices can be composed via the platform after the platform has been certified and with other applications that have never been tested together. We need to develop run-time systems that can dynamically moderate composition, that enable only those compositions that will produce safe executions. Also needed are better and more systematic post-deployment means of detecting harmful emergent behaviors that could not be anticipated and planned for statically.

Compositional development relies on well-defined and precisely specified interfaces. Currently, we do not have adequate methods for capturing interface properties needed to support compositional reasoning about safety, including (a) *non-functional* properties, (b) interactions that a component might have with its context when it is failing, and (c) ways of detecting possible interactions through the environment.

*Lack of Standards, Regulatory Guidelines, and Legal Frameworks for Compositional Approaches:* Allowance for compositional approaches in existing certification and regulatory guidelines is minimal. To a large degree this is justified because the community has not yet arrived at a convincing approach for demonstrating safety using these approaches. Systems composed from components from different manufacturers often lead to questions about liability, *i.e.*, which manufacturer is liable when a system causes harm.

**Directions for the Future:** We believe that there are several specific capabilties, technologies, and products to which the software engineering community can contribute for moving toward more compositional approaches to building safety- and security-critical systems.

*Open-source High-Assurance Infrastructure:* The software engineering, networking, and operating system communities have been very successful in creating open-source infrastructure such as Linux, Eclipse, and various open-source middleware implementations that have accelerated research and development. Similar contributions are needed for high-assurance platforms, networking, building blocks for safety systems, and other reusable infrastructure components targeting safety-critical systems. The CompCert compiler [46], an optimizing compiler for a large subset of C proven correct in the Coq theorem prover provides the basis for building machine-verified compilers for other programming languages and behavioral modeling languages. This could include collections of machine-verified static analyses. The recently released open-source Muen separation kernel [82] developed in the high-assurance Spark language [5] provides another potential foundation on which to build, *e.g.*, partitioning middleware, application platforms for safety-critical embedded applications.

*Interface Contract Languages Oriented to Safety and Security:* Techniques for specifying and checking component interface contracts need to be extended to include better support for timing, resource utilization, and other non-functional properties. Instead of just capturing desired behavior of a component when it is functioning correctly, interfaces need to capture how components generate or propagate errors when they suffer failures.

*Compositional Approaches to Hazard Analysis and Assurance Arguments:* The core techniques of safety engineering such as hazard analyses will need to be adapted, to as great an extent as possible, to work compositionally, perhaps following the architectural or functional decomposition of a system. Similarly, notations and techniques for assurance cases need to be extended to support notions of compositionality (*e.g.*, [42]).

*Independent Safety Systems:* A post-deployment approach, as mentioned earlier could be implemented by an independent safety-system, such as used in the nuclear power domain, in which there is strict separation between safety and control systems [77].

# 7. USE OF TOOLS IN CERTIFICATION

**Challenge:** The use of tools for the development and the verification of critical software in a certification context has been increasing steadily – in particular, with model-based technologies during the development process and sophisticated static analysis tools during the verification phase. This increase is driven in part by (a) the need to automate tasks to cope with the increasing scale and complexity described in Section 1, (b) the fact that automated tools are (potentially) more reliable than humans at performing certain tasks, and (c) improvements in underlying verification and analysis technologies. If certification can be viewed as a demonstration of conformance to safety-related requirements (see Section 2.1) how much confidence do we need in the tools that participate directly or indirectly in this demonstration? How do we establish confidence in these tools? How can the claims, evidence, and assurance artifacts associated with the use of different tools be combined to substantiate an overall assurance case for a system? How can we create a standards and regulatory ecosphere that will grow the market of certification-relevant tools and encourage innovations within this space?

**Current State:**
*Lack of Progress in the Software Engineering Community:* The Software Engineering community has a long history of producing tools for development, testing, and verification. Software Engineering conferences have entire tracks devoted to tool demonstrations, and there is increasing emphasis on providing test suites and other artifacts that allow other researchers to assess the quality and effectiveness of research tools. Many of these tools have the potential to support safety assessments, and authors often make explicit claims

about the relevance of the tools to certification. Yet relatively little is known within the SE community about the tool qualification notions present in current certification standards. Heimdahl addressed the challenge of incorporating tools in the certification process in his 2007 ICSE FOSE paper [27], but there has been little significant progress on the issue in the Software Engineering community since then. Researchers are not encouraged to engineer their tools to support qualification. There is little research on solutions for architecting and engineering tools to support qualification.

*Progress in Standards:* In the recent past, most safety standards, with the exception of DO-178B, provided little guidance regarding assessing the adequacy of tools used to develop or verify the software subject to certification. However, in an encouraging trend, recent standards such as ISO 26262, DO-178C and the latest version of EN50128 clearly recognize the role of tools and in particular the fact that their deficiencies can have a potentially serious impact on the safety of a system. Slotosch [66] notes that all these standards have a three phase approach for using tools safely.

First, tools are *classified* into categories that describe the confidence required in the development process of the system. The classification is based on the analysis of potential errors in the tool and their detection or prevention probabilities within the process [28, 81]. For instance, EN50128 for railway applications defines three categories of tools:

- Class T1: tools whose deficiencies cannot have any impact on the final system (e.g., a word processor),
- Class T2: tools, usually used in the verification phase, whose deficiencies could mask an error in the final system (e.g., a test-harness generator or a structural coverage analyzer), and finally
- Class T3: tools whose deficiencies could insert an undetected error in the final system (e.g., model to source code generator).

Second, tools for which confidence of correctness is required must be *qualified* by assuring their correctness for use in the development of a given system. The degree of assurance and specific assurance requirements are tied to the *usage context* – which includes the criticality level of software (tools applied to software of higher criticality require greater assurance), the tool features (as categorized above) required in development (correctness of unused features does not need to be assured), and the organization of the tool chain (are tool results used to justify eliminating the development steps?, are potential errors in the tool detectable by downstream tools?). Thus, qualification will not focus exclusively on tool correctness; an analysis of risks associated with tool malfunction is required along with mitigation strategies. In essence, a hazard analysis is applied to the tool chain.

It is important to understand that, because tools must be qualified with respect to their usage context, tool vendors technically do not sell *qualified* tools; instead, they sell *qualifiable* tools presented in the form of *qualification kits* (aka, *certification kits*) (Validas AG and Esterel are two companies which have developed significant infrastructure for this purpose). Application of the kit to achieve tool qualification for a particular system development project is typically carried out by the tool vendor and customer working in close collaboration. In ISO 26262, there are four possible qualification methods: increased confidence from use, process assessment, validation, and development according to a safety standard [66]. If a tool is not qualified, it can still be used but its output needs to be verified as part of the system certification activities.

Third, *usage* of the tool must be restricted according to declarations made during qualification. Documentation must contain the constraints from the process that have been considered in qualification analysis phases and workarounds for all restrictions found during tool qualification [66].

**Gaps and Barriers:**
*Overhead in Producing Qualifiable Tools and in Qualifying Tools:* The effort required for a tool vendor to produce a qualification kit for a tool is significant. At least 8 to 12 different documents are required (totaling multiple hundreds of pages). Some documents are the same for each qualification, but other documents are designed to be instantiated to the customer's project context for each qualification. Other required artifacts include an assurance case for the tool (providing a risk analysis of both the development of the tool and use of the tool), and extensible test suites to be customized by the user to the particular use cases of their project. Proprietary tool development artifacts including design and source code, verification cases and procedures, verification results, software configuration management and software quality assurance records are held at the vendor's site to be inspected by certification authorities.

*Unnecessary Overhead in Qualifying Tools for Multiple Domains:* Very few development or verification tools are specific to one domain, yet the qualification requirement vary significantly across safety standards for different domains – requiring tool vendors to develop different versions of their tool qualification kits for each standard supported. Such a situation decreases the attractiveness of the qualifiable tool market.

*Widely-used Tools Not Designed for Qualification:* Many of the recent development and verification tools are not created specifically for the safety-aware segment of the software market, and thus they do not provide any of the evidence that could be used to raise the level of confidence in their usage that would make them suitable without substantial additional work in a certification context. Furthermore, requiring specialized artifacts coming from a typical embedded software development life-cycle can be a major barrier for such tools: most of them are not standalone and rely on other tools and libraries that have their own independent life (e.g. all the tools that have matured in the Eclipse ecosystem).

**Research Directions:**
*Unified Approaches to Tool Qualifications:* Following the recommendations in Section 3, research should aim to identify and present in a domain-independent manner the foundational principles of tool qualification. This will facilitate the dissemination of tool qualification issues while bypassing the burden of acquiring expensive standards for each domain. In addition, it may be possible for research in this area to drive evolution of standards toward more unified approaches. In an encouraging sign, the recently released tool qualification requirements for avionics (DO-330) are designed to be domain neutral.

*Different Architectures and Approaches for Producing Evidence:* Existing qualification requirements and vendor kits are oriented around testing. Within the software engineering and programming languages communities, just as conference program committees have come to give 'credit' to authors that provide machine checked proofs of correctness of algorithms, open source tools, or artifacts used in empirical evaluations (all of which aim to facilitate independent review of claims), they should similarly give credits for research tool developers in verification and safety space that pro-

vide assurance artifacts and various forms of evidence that facilitate independent review of a tool correctness and effectiveness. Different approaches to high-confidence tool assurance (*e.g.*, verifying tools via machine-check proofs of correctness versus having tools emit proofs that the tool has performed correctly for a particular program) should be systematically enumerated and their tradeoffs examined.

*Development and Modeling Environments for Qualifiable Tools:* Open source projects should be pursued that provide development and modeling environments for building qualifiable tools. An interesting initial step in this direction is the *Auto IWG WP5: Eclipse Qualification Process Support* project [66] that aims to enable tool developers using Eclipse as a platform to develop qualifiable plug-ins.

*Applying Product-line Approaches:* Tool qualification kits, have commonalities and variabilities across different domains and different vendor projects. Researchers should investigate the extent to which product-line engineering [58] can be applied to improve the organization, effectiveness, and application of qualification kits.

# 8. INCREASING AUTOMATION IN HAZARD ANALYSIS

**Challenge:** As noted in Section 2.2, hazard analysis is a crucial activity in safety-critical development. Not only does it drive design, but it also plays a key role in assurance arguments. Significant modeling and automation tooling has been developed within the software engineering community for requirements elicitation and management, architecture modeling, code generation, maintenance, formal verification, and testing. Despite its central role in safety-critical development, there is very little support for modeling/automation of hazard analysis and other safety-related analyses. This often results in a lack of rigor in execution and documenting of the analyses (leading to safety-related design flaws), difficulties in managing and updating results as design and implementation evolves, and difficulties on the part of certification agents in assessing the completeness and accuracy of results (leading to increased review time and overlooked safety problems). An important dimension of this challenge is that, in contrast to conventional forms of software analysis, reasoning about hazards requires reasoning about not just planned interactions with the environment, but about unplanned interactions. Reasoning about the causes of hazardous states includes not only reasoning about intended system behavior, but about behaviors that the system may exhibit when components are failing or when functions are erroneous.

**Current State, Gaps, and Barriers:** The current state of practice is reflected fairly well in Ericson's book on Hazard Analysis Techniques (HAT) [13]. Intermediate results and outcomes are recorded in text-based forms, which are typically captured using word processing or spreadsheet tools. Some automated support exists for specific techniques such as fault tree analysis (FTA), but these address only limited slices of the overall safety analysis space and are typically not integrated well with other development artifacts. Below we list other gaps in the ability of the community to effectively carry out hazard analyses.

*System Hazard Identification and Definition:* The early stages of development, as defined in safety methodologies such as ARP 4761 [67], attempt to identify relevant system hazards. This is often challenging due to the bipartite nature of hazards, namely, a *system hazard* needs to capture how a *combination* of a (possibly erroneous) system state together with an environment condition/state will lead to harm. While it is easier (but not trivial) for engineers to conceptualize possible system states, conceiving of harm-relevant environment states is typically more difficult since the conceptual state space of the environment is much larger, more complex, and less predictable. While intuition about system states can be gained by early prototyping and design reviews, extensive knowledge of harm-relevant environment states is typically gained through experience, which is not easily captured, transferred between, nor accumulated across engineers. Hazard identification and subsequent analysis is also closely related to validation, and it is impacted by analogous challenges. While verification assesses the relationship between two rather precisely designed artifacts (the system implementation and requirements/specifications), validation assesses the relationship between the system and a human's conceptualization of the system (which is often imprecise and difficult, if not impossible, to capture completely in a concrete artifact). Similarly, identifying hazards rests largely on a human's conceptualization of harm, the environment, and the system, along with the human's ability to systematically explore combinations of these. More effective techniques are needed to help engineers explore the hazard space, capture hazards at a greater level of precision and formality to better drive the system engineering process, and to accumulate and organize domain-specific knowledge of hazards derived from experience.

*Increasing the Speed and Efficiency of Regulatory Reviews:* The issues of scale and complexity noted in Section 1 are making it increasingly difficult for regulatory authorities to review system safety arguments and evidence. Non-integrated informal documentation currently used in most regulatory submissions makes navigating through hazards, hazard analysis results, and mitigation strategies implemented in the system more difficult.

**Research Directions:**

*Automated tools for exploring the hazard space:* Techniques from the software engineering community for formalizing and animating use cases (*e.g.*, [21]) help engineers conceptualize interactions between a system and its environment. Tools like Alloy [41] use automated state-space exploration technology to help engineers visualize system states. Engineers can add logical constraints to focus attention on state space subsets of particular interest. Researchers might investigate if similar techniques could be applied to explore combinations of the system and environmental states potentially associated with hazards.

*Dependency Tracking:* Reasoning about dependencies as well as the origin and propagation of values is common in conventional software engineering tools including compilers, model checkers, program slicers, and extended static checking tools. Experience with these tools has illustrated the usefulness of attaching various forms of annotations and pragmas directly in source code and other machine-readable development artifacts to capture origins of certain categories of values (*e.g.*, null references, possibly out of range scalars). These annotations and artifacts into which they are embedded can then be used as inputs to automated analyses that propagate information through artifacts and either produce additional annotations or analysis results directly linked to the artifacts. For example, in modern development environments for Java, C#, and Ada, a developer may introduce simple annotations indicating that certain variables are intended to hold reference values that are *always non-null* (remaining variables are allowed to hold ei-

ther null *or* non-null values). This information is then propagated throughout the program to (a) deduce when other variables are always non-null and (b) to detect possible run-time exceptions due to dereference of null pointers.

Many aspects of hazard analysis techniques (HAT) include identifying component failure modes, which could be captured as simple enumeration annotations attached to a component in formal architecture descriptions. Many steps in reasoning about error propagation are highly repetitive and intermediate results are calculated manually in a straightforward manner from other development artifacts (*e.g.*, identifying enclosing components or subsystems in a Failure Modes and Effects Analysis (FMEA)). If linked to formal system architecture descriptions including both software and hardware components, reasoning about propagation of information (*e.g.*, how faults and effects of faults flow through the system) could be automated and traceability would be facilitated. HAT information including error types, error propagation paths, and component failure probabilities to be captured directly as annotations in a system architecture description. Unfortunately, despite longstanding formal approaches from academia and automation in some commercial tools, inputs to HAT are most commonly captured informally and cannot support the envisioned automation described above.

*Architecture-integration Automation for Hazard Analysis:* If architectures and dependencies are captured formally in models, annotations languages for hazard- and safety-related properties could be added to these models so that associated analysis tools could provide automation for many of the steps in FMEA, FTA, and other HATs. Preliminary work in this direction can be found in the Error Modeling (EM) Framework of the Architecture and Analysis Definition Language (AADL) [16]. As described in the EM annex document [15], EM enables semantic modeling of faults and errors in a non-standard type system, with annotations to AADL architectural models capturing the origin and propagation of different categories of errors. Error state machines and intra-component dependency information can be added to components to capture how the component generates or propagates errors in different failure modes. Annotated AADL component connectors provide information about how errors progagate between components. Using state space algorithms similar to those found in conventional probabilistic model checking and program slicing algorithms, AADL EM tools can automatically generate substantial portions of analsyses such as FMEA and FTA. Further work is needed to directly model hazards in AADL EM. In addition, the current HATs in AADL tend to be failure/reliability oriented (FMEA is currently the best supported analysis in AADL EM) and only indirectly address hazards and safety.

# 9. BUILDING COMPETENCE TO ENGINEER SOFTWARE FOR SAFETY CRITICAL SYSTEMS

**Challenge:** Current competence-building practices, including the educational infrastructure and curricula, are not sufficient to engineer the systems characterized in Section 1. On the one hand, market forces have not been effective in driving a corrective response, because these systems are a relatively small segment of the vast software engineering market, which seems to value delivering functionality 'fast to market' much more than 'get it right the first time'. On the other hand, engineering such systems increasingly requires breadth (especially system safety engineering [27]; as well as domain knowledge; hazard analysis; assurance cases; regulatory frameworks; related standards and guidelines) and depth of knowledge, far beyond the delivery-capability of the current educational infrastructure.

Even within the small market of safety-critical systems, the 'market forces' are not generated, because the management in many of the developer organizations is not aware of the competence needed, and has little appreciation for the dangers introduced by the discontinuous nature of software and the growing complexity inherent in software based systems) [56].

**Current State:** There are very few courses at any level in software engineering or computer science curricula that deal adequately with essential topics in building safety-critical systems [27, 49, 44]. There is no explicit consensus-supported statement of the competence needed for engineering a safety-critical system – not even for a narrowly-defined domain (*e.g.*, a digital safety system for a nuclear power plant or a medical device or a drive-by-wire automobile). As a consequence, and as noted in Section 1, most people working on safety-critical software do not have the requisite knowledge. We do not know of any university programs in software engineering that are directed towards certification of software-dependent safety-critical systems. However, examples in the needed direction can be seen at two universities: University of York High Integrity Systems Engineering group, UK and MIT Engineering Systems Division, USA. Both offer Master's and Doctoral degree programs, as well as short courses for practitioners in industry. Graduate education programs in computer science and software engineering programs do include research in safety critical systems. However, much of this research is far removed from the practical necessities of industrial projects. While society has elaborate licensing and refresher requirements for a surgeon, even when some of the surgical work is offloaded to a digitally controlled robotic device, there are no such requirements on those who engineer the automated system. This is a grave deficiency.

**Gaps and Barriers to Progress:** Universities view the development of software for use in safety-critical systems as a small, specialized area, and the workload for students in an accredited undergraduate software engineering program is already considerable, so adding material to a curriculum inevitably entails a discussion of what material would have to be removed. Universities have not adopted the undergraduate curriculum guidelines, developed by the ACM/IEEE Computer Society [2], based on the SWEBOK 2004 body of knowledge[30], and these curricula include the possibility of 'streams' such as safety-critical systems or highly secure systems through the use of technical electives. Similar difficulties exist in including other techniques relevant to safety-critical systems, but usually not taught in generic undergraduate software engineering programs – hazard analysis and assurance cases are two such examples. Many references used in practice, such as standards, are costly (beyond the budget), further limiting the students' readiness to enter practice. Contrast this with the preparation of an undergraduate accounting student, preparing to be a CPA concurrently. In this regard, IEEE should be commended for making their standards available to universities through existing library copyright agreements. Meaningful end-to-end design projects are difficult to include in current coursework, due to the scale and needed hardware. Many companies/agencies involved in the development or certification of safety-critical systems have not yet fully realized that graduates from electrical engineering or mechanical engineering, while quite adept at writing code, do not have the depth of software specific knowledge required. Added to this, graduates from

theoretically oriented computer science, or conventional software engineering programs are not likely to have the knowledge required for software-dependent safety-critical systems.

**Penetrating the Barriers:** To engineer certifiable software-dependent safety critical systems, a comprehensive competence-building infrastructure is needed to produce certifiable engineers, to whom the end-users of their services can entrust their lives, just as to a surgeon. The competence-building infrastructure is needed not only for undergraduate and postgraduate university education, but also for in-career continuing education to update engineers, managers, and senior executives. While the classroom teaching would provide a certified theoretical foundation, it should be supplemented with certified practical training, as done for medical doctors and accountants.

*Undergraduate Education:* Undergraduate curricula should consider the recommendations in [2], providing different concentration streams through technical electives. The curricula should also allow credit for accredited courses on line. Commensurate online courses should be developed with synergy across interested organizations. Going beyond the recommendations in [2], in determining the 'core' curriculum, educators should recognize that 'software engineering for critical application domains' is not merely *applied computer science*. It is a *techno-socio* challenge and depends critically on social processes as well. Capstone design (project-oriented) courses in an undergraduate software engineering safety-critical stream should be supported with reusable resources (e.g.: tool suites; library of software and hardware building blocks; case studies; standards or mock standards), such that student teams can build a certifiable safety-critical system within the time budget of the course.

*Postgraduate Education:* As recommended for undergraduate programs above, graduate education programs should also include provisions for different concentration streams; for example see [53]. Furthermore, graduate education programs should also offer course-series and practical training opportunities that equip the scholar for certification in a concentration stream, such as 'Software-dependent systems engineering for life-critical medical devices.' To prepare a scholar for certification in a particular application domain, curricula developers should recognize that graduate work in such specializations of software engineering is not about 'going narrower and deeper into the application of computer science', but about integrating knowledge across different disciplines; for example, see [8] – a graduate program. The software engineering discipline can learn from many features of this program: interdisciplinary integration; project-based learning; shared 'library' resources such as platform elements, reusable building blocks (components), and tool suites; teamwork; and collaboration with entrepreneurs. Throughout this paper we have highlighted *research directions* to further our technical knowledge related to how to build and evaluate software-dependent safety-critical systems. There is amazingly little research currently targeted directly at software certification, resulting in ample opportunity to target this for graduate research topics. Not only do we have to produce graduates who are competent to build and certify these systems, we also have to produce graduates who are capable of extending our knowledge in these areas.

*High-quality Case Studies:* Educational curricula should include real-life case studies (or sanitized, synthesized cases derived from real-life examples), for example, as done in leading business schools. Their classroom use should be supported with reusable shared resources, as mentioned above in the context of Capstone design project courses. This would also reinforce that software engineering involves the *engineering* of a software based *system*, even in the case of conventional software engineering [43, 55]. Some projects that have made an initial start in this direction include the Pacemaker Challenge [7] and the Open PCA Pump [45].

*Course Material Providing Domain Knowledge for Safety-critical Domains:* The time and effort currently needed to learn about a particular application domain should be reduced. For example, online learning resources could be prepared in cooperation with respective industry leaders in avionics; digital safety systems for nuclear power plants; life-critical medical devices; drive-by-wire automobiles, etc.

*Mock Standards and Compliance Evaluation Guidelines:* Given the cost barrier of purchasing relevant standards, mock standards (e.g., for development, evaluation, and assurance of safety-critical systems in various application domains) should be developed as resources shared across the educational community.

*Raising Awareness of Managers and Executives:* Since many managers and executives in industry are not aware of issues such as hidden hazards and costs in engineering complex safety-critical systems and are not aware of the competence required to engineer such systems, short courses, seminars, and workshops should be developed to raise their awareness and understanding (for an example, see [70]). These training resources should include case studies, showing the economic impact of these issues.

# 10. CONCLUSIONS AND FUTURE WORK

Increasingly, critical systems being deployed are growing in size and interactions with other systems, to the extent that these systems are not fully understood, much more difficult to verify, and almost impossible to validate to the level of confidence required. As a result they are becoming less assurable. This is exacerbated by the fact that current common practice in building and certifying software-dependent safety-critical systems falls short of the state of the art, and far short of what many believe is possible. In particular, *designed-in safety/security* seems to be a foreign concept to many companies building safety-critical systems. Designed-in safety/security is the single most important principle in the development of safety-critical software-dependent systems. Safety and security cannot be tested into the system – they have to be built in right from the beginning.

The gap between current practice and state of the art encompasses a wide variety of topics, from requirements validation to reliance on process as an indicator of quality in the case of certification. The most important barriers to closing these gaps are summarized below.

- Education seems to be at the heart of many of our problems. There are almost no courses in software engineering or computer science that deal with essential topics in building and/or certifying safety-critical systems. Section 9 presents a number of ways in which we could improve the education of the safety-critical software workforce.
- Another systemic problem we face is that the (safety-critical) software industry has been built on the premise that if developers comply with process based development standards, the products they build will be safe, secure and dependable. Section 4 describes the benefits of assurance cases as a framework for product focused certification, as well as how to

overcome some of the practical issues that may arise from their unconstrained use.

- The inadequacy of requirements in common practice is astounding. Requirements elicitation/gathering is a notoriously difficult activity. Documenting requirements so that they are complete, unambiguous, correct, understandable, and mitigate known hazards is not much easier. Section 5 discusses why validation of requirements is so crucial and so poorly implemented, and why mathematics based requirements are so advantageous – assuming we do not bury the 'intent' of the requirement in the rigour of the mathematics.

- Software tools are an indispensable aspect of safety-critical software development. Complexity of the systems we need to build (and certify), as well as the increasing trend towards model-driven development, has fundamentally changed our need for software tools. The need for tools, as well as the need for a more unified and effective approach to tool qualification is discussed in Section 7. Interestingly, almost all our current tools have been aimed at supporting development and verification of software. There are (almost) no tools specifically designed to aid the certifier/regulator.

- Compositional approaches to certification are needed as just one way of coping with the large, interconnected systems we are building. Compositional verification, and design by contract are well known techniques. However, certification is not verification – and safety is a global property. Section 6 shows why compositional certification, if it can be realized, would be so advantageous. It also highlights the problems we face on a regulatory level, as well as the technical hurdles we must overcome.

## 10.1 Future Directions

The above discussion centered on discovering and filling gaps between current practice and state of the art. State of the art reflects current knowledge and capability, and we would be remiss if we set our sights that low. There is another set of gaps – and these are barriers related to lack of knowledge and/or capability.

- A hot research topic and of real importance to certification is the notion of confidence. It is most often discussed within the context of both the argument and evidence in assurance cases. We need to be able to 'measure' it; determine if it is cumulative in some way; and determine what an acceptable minimum might be. Related to this, we need a theory of coverage that enables us to combine multiple verification techniques in such a way that we can show that we have more confidence in this case, compared with having a single verification technique.

- A related topic is that of evidence. We need to decide what constitutes evidence, and what evidence is essential to the evaluation of safety, dependability, and security. If we can achieve this, we should also be able to build tools that directly support certification.

- Having a requirements specification in a form that is understandable, amenable to analysis, and precise enough to drive the development and verification is also essential. Domain specific languages (DSLs) will likely be an effective way of achieving this, for both traditional (formal) software development as well as model-driven approaches.

- Effective ways of performing compositional certification must cope with the global nature of safety and security. This will have to include compositional approaches to hazard analysis, for example. It may also be facilitated by separation of concerns – the idea of a safety system that monitors a system of (control) systems.

- We need a more uniform and effective way of specifying timing requirements, both functional timing requirements and performance timing requirements. In addition, we need to be able to generate design envelopes that would satisfy those timing requirements, and have analysis methods and tools that verify compliance with the timing requirements.

- Current verification analysis techniques are not practical enough in more than one respect. We normally single out our inability to verify very large, complex systems, because our methods often do not scale adequately. Just as important is the fact that it is unrealistic to expect a refinement step to comply exactly with its specification. Verification needs to show compliance within a pre-determined tolerance.

- Suites of tools could be developed to ease the burden of qualification of some of those tools. For example, generating a proof is sometimes very difficult. Checking the proof is a much easier task. Placing a higher qualification level on the checking tool may be possible to achieve and as long as both tools are used, could result in much a much less onerous qualification for the tool that generated the proof.

- Recognition that certification is a socio-technical endeavor raises difficult questions. An obvious concern is how to factor into the confidence measure, the competence of the developers and the certifiers. Another may be how we get industry consensus on mandating the use of formal requirements. Formal methods must be made more amenable to practitioners, and the tools in which they are implemented, must scale to deal with realistic problems.

The bottom-line: We have not yet reached the time when we can say that we truly do understand how to evaluate fundamental properties in software products. *Software development* has really outstripped both *computer science* and *software engineering*. Software products are a mainstay of safety-critical systems, and we have built these systems through sweat and tears, with rapidly acquired knowledge related to practical, carefully thought-out processes, and some use of sophisticated mathematical tools. However, the true nature of software still eludes us. If we cannot agree on what the essential attributes of a software product are, with regard to 'quality', and if we cannot measure the degree to which we achieve those attributes, we have not yet built an adequate software science. It also means that software engineers then lack essential knowledge on what to base their methods. This is the fundamental challenge that we face.

# 11. ACKNOWLEDGEMENTS

The authors wish to thank Tom Maibaum (McMaster Centre for Software Certification, McMaster University) for his comments on earlier versions of this paper. The authors also deeply appreciate the support and patience of the FoSE 2014 organizers Matt Dwyer and Jim Herbsleb during the preparation of this paper.

# 12. REFERENCES

[1] Ministry of defence defence standard 23-09 – generic vehicle architecture, aug 2010.

[2] ACM and IEEE Computer Society. Software engineering 2004, curriculum guidelines for undergraduate degree programs in software engineering. http://sites.computer.org/ccse/SE2004Volume.pdf, 2004.

[3] Associated Press. Okla. jury: Toyota liable in sudden acceleration crash. http://www.cbsnews.com/news/okla-jury-toyota-liable-in-sudden-acceleration-crash/, 2013.

[4] AUTomotive Open System ARchitecture. AUTOSAR. https://www.autosar.org, 2014.

[5] J. Barnes. *High Integrity Software—the SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[6] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre. The MILS component integration approach to secure information sharing. In *Proceedings of the 27th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Oct. 2008.

[7] Boston Scientific. PACEMAKER system requirements specification. http://sqrl.mcmaster.ca/pacemaker.htm, 2007.

[8] Carnegie Mellon. Entertainment Technology Center. http://www.etc.cmu.edu/site/, 2014.

[9] R. N. Charette. This Car Runs on Code. *IEEE Spectrum*, Feb. 2009.

[10] R. DeLong and J. Rushby. A common criteria authoring environment supporting composition. In *Proceedings of the 8th International Common Criteria Conference*, 2007.

[11] B. Dion. FACE, ARINC, DO-178C avionics standards help U.S. DoD's vision of reusable technology to take off. *Military Embedded Systems*, 9(2):36–39, mar 2013.

[12] M. B. Dwyer and S. G. Elbaum. Unifying verification and validation techniques: relating behavior and properties through partial evidence. In *Proceedings of the Workshop on Future of Software Engineering Research (FoSER 2010)*, pages 93–98, 2010.

[13] C. A. Ericson. *Hazard Analysis Techniques for System Safety*. Wiley-Interscience, 2005.

[14] European Committee for Electrical Standardization (CENELEC). Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems. *CENELEC Standard 50128*, 2011.

[15] P. Feiler. *Architecture Analysis and Design Language (AADL) Annex Volume 3: Annex E: Error Model V2 Annex*. Number SAE AS5506/3 (Draft) in SAE Aerospace Standard. SAE International, 2013.

[16] P. Feiler and D. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language*. Addison-Wesley, 2012.

[17] E. Feliz. Engineering pipeline under pressure. *Avionics Today*, December 2007.

[18] N. Gershenfeld, R. Krikorian, and D. Cohen. The internet of things. *Scientific American*, Oct. 2004.

[19] S. Grigorova and T. Maibaum. Taking a page from the law books: Considering evidence weight in evaluating assurance case confidence. In *ISSRE (Supplemental Proceedings)*, pages 387–390, 2013.

[20] C. Hagen, S. Hurt, J. Sorenson, and D. Wall. Software: The brains behind us defense systems. A.T. Kearney Whitepaper, Nov 2012.

[21] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[22] J. Hatcliff, M. Heimdahl, M. Lawford, T. Maibaum, A. Wassyng, and F. Wurden. A Software Certification Consortium and its Top 9 Hurdles. *Electronic Notes in Theoretical Computer Science*, 238(4):11–17, Sept. 2009.

[23] J. Hatcliff, A. King, I. Lee, A. Fernandez, J. Goldman, A. McDonald, M. Robkin, E. Vasserman, and S. Weininger. Rationale and architecture principles for medical application platforms. In *Proceedings of the 2012 International Conference on Cyberphysical Systems*, 2012.

[24] J. Hatcliff, A. Wassyng, T. Kelly, and C. Comar. Certifiably safe software-dependent systems: Challenges and directions (paper web site / extended version). http://santoslab.org/pub/papers/fose14-certification/.

[25] R. Hawkins, K. Clegg, R. Alexander, and T. Kelly. Using a software safety argument pattern catalogue: two case studies. In *Computer Safety, Reliability, and Security*, pages 185–198. Springer, 2011.

[26] R. Hawkins, I. Habli, T. Kelly, et al. Principled construction of software safety cases. In *Proceedings of Workshop SASSUR (Next Generation of System Assurance Approaches for Safety-Critical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security*, 2013.

[27] M. P. Heimdahl. Safety and software intensive systems: Challenges old and new. In *2007 Future of Software Engineering*, pages 137–152. IEEE Computer Society, 2007.

[28] J. Hillebrand, P. Reichenpfader, I. Mandic, H. Siegl, and C. Peer. Establishing confidence in the usage of software tools in context of iso 26262. In *Computer Safety, Reliability, and Security - 30th International Conference (SAFECOMP 2011)*, pages 257–269, 2011.

[29] C. M. Holloway. Making the implicit explicit: Towards an assurance case for do-178c. In *Proceedings of the 31st International System Safety Conference (ISSC)*, 2013.

[30] IEEE Computer Society. Software engineering body of knowledge. http://www.computer.org/portal/web/swebok/html/contents, 2004.

[31] Institute of Electrical and Electronics Engineers (IEEE). IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations . *IEEE Standard 7-4.3.2*, 2010.

[32] International Electrotechnical Commission. Medical device software – Software life cycle processes. *IEC Standard 62304 edition 1.0*, 2006.

[33] International Electrotechnical Commission. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. *IEC Standard 61508 edition 2.0*, 2010.

[34] International Organization for Standardization. Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. *ISO Standard 25000*, 2005.

[35] International Organization for Standardization. Application of Risk Management to Medical Devices. *ISO Standard 14971*, 2007.

[36] International Organization for Standardization. Road Vehicles – Functional Safety. *ISO Standard 26262*, 2011.

[37] International Organization for Standardization. Systems and software engineering – Systems and software assurance – Part 2: Assurance case. *ISO Standard 15026 part 2*, 2011.

[38] ISO, IEC, IEEE. IEEE Standard 24765: Systems and software engineering - Vocabulary, 2010.

[39] ISO/IEC. ISO/IEC 17000: Conformity assessment – vocabulary and general principles, 2004.

[40] ISO/IEEE. 11073-x Medical Health Device Communication Standards family.

[41] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.

[42] T. Kelly. Using software architecture techniques to support the modular certification of safety-critical systems. In *Proceedings of the Eleventh Australian Workshop on Safety Critical Systems and Software - Volume 69*, SCS '06, pages 53–65, 2006.

[43] J. C. Knight. Focusing software education on engineering. *ACM SIGSOFT Software Engineering Notes*, 30(2):3–5, 2005.

[44] J. C. Knight and N. G. Leveson. Software and higher education. *Communications of the ACM*, 49(1):160–160, 2006.

[45] B. Larson and J. Hatcliff. Open PCA Pump project website. `http://openpcapump.santoslab.org`.

[46] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, Dec. 2009.

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

[48] N. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, 2012.

[49] R. R. Lutz. Software engineering for safety: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 213–226. ACM, 2000.

[50] T. Maibaum and A. Wassyng. A product-focused approach to software certification. *Computer*, 41(2):91–93, 2008.

[51] Mathew J. Schwartz. Hacked Medical Device Sparks Congressional Inquiry. `http://www.informationweek.com/security/vulnerabilities-and-threats/hacked-medical-device-sparks-congressional-inquiry/d/d-id/1099726`, 2011.

[52] E. McKenna. Embedded overall. *Avionics Today*, Nov 2008.

[53] Naval Postgraduate School. Department of Systems Engineering: Programs of Study. `http://www.nps.edu/Academics/Schools/GSEAS/Departments/SE/Academics/ProgramsofStudy.html`, 2014.

[54] Open platform for evolutionary certification of safety-critical systems. `www.opencoss-project.eu`.

[55] D. L. Parnas. Software engineering programs are not computer science programs. *Software, IEEE*, 16(6):19–30, 1999.

[56] D. L. Parnas, A. J. van Schouwen, and S. P. Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, 1990.

[57] Patrick J. Graydon, C. Michael Holloway. AESSCS 2014 Workshop. Planning the Unplanned Experiment: Assessing the Efficacy of Standards for Safety Critical Software. `http://www.idt.mdh.se/AESSCS_2014/`, 2013.

[58] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[59] A. Rae, M. Nicholson, and R. Alexander. The state of practice in system safety research evaluation. In *5th IET International Conference on System Safety 2010*, pages 1–8, Oct 2010.

[60] RTCA. Software Considerations in Airborne Systems and Equipment Certification. *RTCA Standard DO-178C*, 2012.

[61] J. Rushby. Modular certification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, US, September 2001.

[62] J. Rushby. Composing safe systems. In *Proceedings of Formal Aspects of Component Software (FACS 2011)*, volume 7253 of *Lecture Notes in Computer Science*, pages 3–11, 2012.

[63] Russell Sydnor, Sushil Birla, Michael Waterman. Gap Assessment of IEC and IEEE Standards for Safety Assurance of Digital Systems. Software Certification Consortium Meeting 9: `http://cps-vo.org/node/3472`, 2012.

[64] K. Sandler, L. Ohrstrom, L. Moy, and R. McVay. Killed by Code: Software transparency in implantable medical devices. Software Freedom Law Center Whitepaper, 2010.

[65] G. Schkade. SAE's Perspective on Connected Vehicle. `http://www.itu.int/dms_pub/itu-t/oth/06/5B/T065B0000020008PDFE.pdf`, 2011.

[66] O. Slotosch. Model-based tool qualification. `http://wiki.eclipse.org/Auto_IWG_WP5`.

[67] Society of Automotive Engineers (SAE). Arp4761: Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment, 1996.

[68] J. Stern. Google Self-Driving Car License Approved in Nevada. `http://abcnews.go.com/blogs/technology/2012/05/google-self-driving-car-license-approved-in-nevada/`, 2012.

[69] United Kingdom Ministry of Defence. Safety Management Requirements for Defence Systems. *Defence Standard 00-56. Issue 4.*, 2007.

[70] University of California, Los Angeles. UCLA Extension: Safety-Critical Software. `https://www.uclaextension.edu/pages/Course.aspx?reg=V9149`, 2014.

[71] U.S. Department of Defense. Mil-std-882e: Department of defense standard practice: System safety, 2012.

[72] U.S. Food and Drug Administration. Safety, Recalls, Market Withdrawals, & Safety Alerts, Background and Definitions. `http://www.fda.gov/Safety/Recalls/ucm165546.htm`, 2009.

[73] U.S. Nuclear Regulatory Commission. Research information letter (ril) 1101: Technical basis to review hazard analysis of digital safety systems.

[74] U.S. Nuclear Regulatory Commission. Research Information Letter 1001: Software-related uncertainties in the assurance of digital safety systems – expert clinic findings Part 1. `http://adamswebsearch2.nrc.gov/IDMWS/ViewDocByAccession.asp?\\AccessionNumber=ML111240017`, 2011.

[75] A. van Lamsweerde. Requirements engineering: From craft to discipline. In *FSE'2008: 16th ACM Sigsoft Intl. Symposium on the Foundations of Software Engineering (Invited Paper for the ACM Sigsoft Outstanding Research Award)*, pages 238–249, 2008.

[76] W. G. Vincenti. *What engineers know and how they know it: Analytical studies from aeronautical history*. The Johns

Hopkins University Press, 1990.

[77] A. Wassyng, , M. Lawford, and T. Maibaum. Separating safety & control systems to reduce complexity. In M. Hinchey and L. Doyle, editors, *Conquering Complexity*, pages 89–108. Springer, 2012.

[78] A. Wassyng and M. Lawford. Lessons learned from a successful implementation of formal methods in an industrial project. In K. Arakai, S. Gnesi, and D. Mandrioli, editor, *FME 2003: International Symposium of Formal Methods Europe Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 133–153. Springer-Verlag, 2003.

[79] A. Wassyng, T. Maibaum, M. Lawford, and H. Bherer. Software certification: Is there a case against safety cases? In R. Calinescu and E. Jackson, editors, *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, volume 6662 of *Lecture Notes in Computer Science*, pages 206–227. Springer Berlin Heidelberg, 2011.

[80] C. B. Weinstock, J. B. Goodenough, and A. Z. Klein. Measuring assurance case confidence using baconian probabilities. In *Proceedings of the 1st International Workshop on Assurance Cases for Software-Intensive Systems (ASSURE)*, 2013.

[81] M. Wildmoser, J. Philipps, and O. Slotosch. Determining potential errors in tool chains - strategies to reach tool confidence according to iso 26262. In *Computer Safety, Reliability, and Security - 31st International Conference (SAFECOMP 2012)*, pages 317–327, 2012.

[82] The Muen Separation Kernel. `http://muen.codelabs.ch/`.