On Software Certification: We Need Product-Focused Approaches[⋆]

Alan Wassyng, Tom Maibaum, and Mark Lawford

Software Quality Research Laboratory
Department of Computing and Software
McMaster University, Hamilton, Canada L8S 4K1
wassyng@mcmaster.ca, tom@maibaum.org, lawford@mcmaster.ca

Abstract. In this paper we begin by examining the "certification" of a consumer product, a baby walker, that is *product-focused*, i.e., the certification process requires the performance of precisely defined tests on the product with measurable outcomes. We then review current practices in software certification and contrast the software regime's process-oriented approach to certification with the product-oriented approach typically used in other engineering disciplines. We make the case that *product-focused* certification is required to produce reliable software intensive systems. These techniques will have to be domain and even product specific to succeed.

1 Introduction

This paper deals briefly with the current state of software certification, why it is generally ill-conceived and some reasons for how (and why) we landed in this mess, and suggestions for improving the situation.

2 Motivation

A certification story: Let us start the discussion with an item that has little to do with software, but is typical of engineered artifacts - a baby walker. Consider a typical baby walker, as shown in Figure 1.

In recent years, there has been considerable concern regarding the safety and effectiveness of baby walkers. In reaction to this concern, we can now consider a certification process we may wish to advocate in order that we may regulate the sale of particular baby walkers. So, what should be the overall thrust of such a certification process? Well, humbly we may suggest that we model the process on certification processes that are common in our domain (software). What may such a process look like? Perhaps something like the list shown below:

1. Evaluate manufacturer's development process.
2. Evaluate list of materials used in manufacture of baby walker.

[⋆] Supported by the Natural Sciences and Engineering Research Council of Canada.

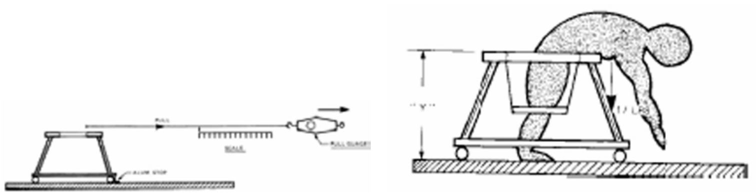
Fig. 1. A Typical Baby Walker

3. Evaluate manufacturer's test plan.
4. Evaluate manufacturer's test results.

Additionally, let us imagine what the manufacturer's submission to the regulators may contain:

1. Process.
   (a) Requirements
   (b) Requirements review
   (c) Design
   (d) Design review
   (e) Manufacturing process
   (f) Manufacturing process review
2. Materials.
   (a) List of materials for each design part
   (b) Safety analysis for each material used
3. Test plan.
4. Test results.

Perhaps a little more detail regarding the testing is required. The manufacturer decided to test two main problems. The first problem was related to quality of manufacture. In this regard, a number of tests were planned and performed regarding the uniformity of the production line and the degree to which the resulting baby walkers complied with the specified design. The second problem related to the tendency of the baby walker to tip. In this regard, two tests of stability were executed. The first test, shown in Figure 2(a), records the force required to tip the baby walker when it is stopped at an abutment. The second test, Figure 2(b), records the moment required to tip the baby walker - simulating a child leaning over.



(a) Stability Test 1 (b) Stability Test 2

Fig. 2. Stability Tests from ASTM Standard F977-00 [1]

So, what may we observe from this example? Our observations may include:

1. It is extremely unlikely that regulators of baby walkers are going to evaluate the manufacturer's development process.
2. The regulator may evaluate the manufacturer's tests, but will definitely run tests independently.
3. The regulator will examine the materials used in the baby walker and determine if they expose the baby to potential harm.
4. **Important:** The regulator is likely to publish and use tests specifically designed to test baby walkers.
   (a) For this example, a number of countries published very specific requirements for baby walkers. For example, *United States Standard ASTM F 977-00 - Standard Consumer Safety Specification for Infant Walkers* [1]. The tests mentioned in Figures 2(a), and 2(b) are not nearly sufficient. A number of dynamic tests have been added to those static tests. The static tests would have been completely useless in determining whether a baby in the walker would fall down unprotected stairs.
   (b) Product-focus is not a panacea either. Canada banned the use of all baby walkers, since Health Canada determined that baby walkers are (probably) not effective, and even product-focused standards may not guarantee safety when the product is ill-conceived [2]. The product-focused standard helped Health Canada arrive at these conclusions, since they could be confident that the products were designed and manufactured well enough to satisfy stated requirements, and so the problems were more fundamental.

Still, it seems strange to us that to certify a baby walker, a regulator devised a product-based standard and tests baby walkers to that standard, whereas, to certify a pacemaker (for example), regulators use generic software process-oriented standards and regulations!

Government oversight: The easiest software certification to motivate is where the government mandates licensing/certification anyway. In this case, we want to make the case to the regulators/certification authorities that product-focused certification will result in much more dependable systems than will process-based certification. We believe that this will improve the objectivity and measurability of evidence, thus improving the evaluation process, and thus making the certification process more predictable for all parties. It should also reduce the post-licensing failure rate and facilitate the identification of the cause. This would certainly, ignoring political issues, induce regulators to adopt a more product-focused approach.

Social expectations: Over the past three or four years, we have seen growing interest in software certification, driven in some cases by the public's dissatisfaction with the frailty of software-based systems. Online banking and trading systems have experienced failures that were widely publicized and have caused wide-spread chaos. Software driven medical devices have killed people. Security breaches in software systems have disrupted peoples' lives. There is no reason that software systems should not be certified as fit-for-use and safe - just as most other products are.

Market advantage: There is also a growing realization by commercial companies that if they can market their software with a warranty, that will give them a tremendous marketing edge. So, as soon as they can manufacture certified software at reasonable cost (and that is the difficulty right now), manufacturers will be driven to consider software certification through normal marketing forces.

Component assurance/qualification: Many industries have to use components manufactured by a myriad of different suppliers. For instance, auto manufacturers manufacture some components themselves and buy others from suppliers. These components are becoming more complex and have to work under stringent timing constraints. Product-focused standards and certification are going to be unavoidable if the components are going to be able to deliver dependable service.

Political considerations: Many software producers find the idea of software regulation anathema: witness the move in various jurisdictions (in the US and an abortive one in the European Union) to lower the liability of software manufacturers from even the abysmal levels in place today.

Governments are woefully ignorant of the dangers represented by the low or non-existent levels of regulation in some industries, such as those producing medical devices, cars and other vehicles, financial services, privacy and confidentiality issues in many information systems, etc.

However, the issue is much too large for us, as a society, to ignore any longer.

3 Current Practice

This section describes approaches to software certification in three different application domains. It presents one of our main hypotheses: current practice in software certification is primarily focused on the process used to develop the software, and that process-focused certification is never going to give us enough confidence that, if a software product is *certified*, then the product will be effective, safe and reliable.

The domains we are going to discuss are:

- medical systems (in the U.S.);
- security systems (primarily in Europe, Japan and North America);
- nuclear power safety systems (in Canada).

3.1 Medical Systems

As an example, we will consider, briefly, the regulatory requirements for medical systems in the U.S. The U.S. Federal Drug Administration (FDA) is responsible for regulating medical systems in the U.S., and they publish a number of *guidelines* on software *validation*, e.g., [3,4]. The FDA validation approach, as described in the FDA guidance documents falls short on describing objective criteria which the FDA would use to evaluate submissions. The documents do not do a good enough job of describing the artefacts that will be assessed. In particular, the targeted attributes of these artefacts are not mentioned, and approved ways of determining their values are never described. The focus of these documents is on the characteristics of a software development process that is likely to produce high quality software. It shares this approach and concern with almost all certification authorities' requirements (as well as those of standards organisations and approaches based on *maturity*, such as CMMI [5]).

3.2 Security Systems

The Common Criteria (CC) [6] for Information Technology Security Evaluation is an international standard for specifying and evaluating IT security requirements and products, developed as a result of a cooperation between many national security and standards organisations. Compared with the FDA's approach for medical systems, the CC has a more systematic and consistent approach to specifying security requirements and evaluating their implementation. The CC does fall into the trap of prescribing development process standards (ISO/IEC 15408) in detail, but, on the other hand, it does a much better job than the FDA guidelines of being measurement oriented.

One very good idea in the CC is that it defines seven levels of assurance, as shown below.

EAL1:	functionally tested
EAL2:	structurally tested
EAL3:	methodically tested and checked
EAL4:	methodically designed, tested and reviewed
EAL5:	semiformally designed and tested
EAL6:	semiformally verified design and tested
EAL7:	formally verified design and tested

It is interesting to note that formal methods are mandated, if only at the highest levels of assurance. Testing occurs at all levels, reinforcing that all certification regimes place a huge emphasis on testing. In keeping with a wide-spread movement in trying to make *software engineering* more of an engineering discipline, we see that CC has introduced the concept of "methodical" processes into their assurance levels. Our only cause for concern in this regard, is that the CC community does not seem to require the process to be both methodical and formal (or semi-formal). We do not agree with this, since formality really relates to the rigour of the documentation. It does not necessarily imply that the process is systematic/methodical.

The taxonomy of the CC describes Security Assurance Requirements (SARs) in terms of action elements for the developer, and for the *content and presentation* of the submitted evaluation evidence for the evaluator. Each evaluator action element corresponds to work units in the Common Evaluation Methodology [7], a companion document, which describes the way in which a product specified using the CC requirements is evaluated. Work units describe the steps that are to be undertaken in evaluating the Target of Evaluation (TOE), the Security Target (security properties of the TOE), and all other intermediate products. If these products pass the evaluation, they are submitted for certification to the certification authority in that country.

There are a number of important principles embedded in this approach: the developer targets an assurance level and produces appropriate evidence that is then evaluated according to pre-determined steps by the certifier; this undoubtedly helps in making the certification process more predictable; and the certification process is designed to accommodate third-party certification.

3.3 Canadian Nuclear Power Safety Systems

While proponents of formal methods have been advocating their use in the development and verification of safety critical software for over two decades [8,9,10], there have been few full industrial applications utilizing rigorous mathematical techniques. This is in part due to industry's perception that formal methods are difficult to use and fail to scale to "real" problems. To address these concerns, a method must supply integrated tool support to automate much of the routine mechanical work required to perform formal specification, design and verification.

There have been some notable industrial and military applications of tool supported formal methods, especially for the analysis of software systems requirements (e.g., [11,12,13,14]). Unfortunately, the formal methods advocates concerned, typically were not given the opportunity to fully integrate their techniques with the overall software engineering process. As a result these applications required at least some reverse engineering of existing requirements documents into the chosen formalism. A potential problem of this scenario is that two *requirements specifications* may result: the original, often informal, specification used by developers; and the formal specification used by verifiers.

An example of this problem occurred in 1988. Canadian regulators were struggling with whether to licence the new nuclear power station at Darlington in Ontario. At issue was the "certification" of the shutdown system - it was the first software controlled shutdown system in Canada. The regulators turned to Dave Parnas for advice, and his advice was to require formal proofs of correctness as well as other forms of verification and validation. The regulators and Ontario Hydro - OH (now Ontario Power Generation - OPG) worked together to agree on an approach to achieve this. Most of the time, OH prepared process documents and interacted with the regulator to obtain an agreement in principle. The correctness "proofs" were eventually delivered to the regulator (more than twenty large binders for the two shutdown systems), and a walkthrough was conducted for each of the shutdown systems [15,16]. The end result was a licence to operate the Darlington Nuclear Generating Station. However, the regulator also mandated a complete redesign of the software to enhance its maintainability.

As a result, OPG, together with Atomic Energy of Canada Limited (AECL), researched and implemented a new safety-critical software development process, and then used that process to produce redesigned versions of the two shutdown systems [17]. As a start, OPG and AECL jointly defined a detailed engineering standard to govern the specification, design and verification of safety-critical software systems. The CANDU Computer Systems Engineering Centre of Excellence Standard for Software Engineering of Safety Critical Software [18] states the following as its first fundamental principle:

> *The required behavior of the software shall be documented using mathematical functions in a notation which has well defined syntax and semantics.*

Not only was the software redesigned along information hiding principles, but new requirements documents were also produced. These requirements are formally described, primarily through the use of tabular expressions (function tables) [19]. In fact, the current implementation of the software engineering process makes extensive use of tool supported tabular expressions [20]. The process results in the production of a coherent set of documents that allows for limited static analysis of properties of the requirements. The process also includes a mathematical verification of the design, described in the Software Design Description (SDD), against the software requirements documented in the Software Requirements Specification (SRS). This project is then an example of one in which, from the start, the software development process itself was designed to use formal methods and associated tools to deliver evidence for the licensing (certification) of the resulting system.

A model of the process, including integrated tool support, that was applied to the Darlington Nuclear Generating Station Shutdown System One (SDS1) Trip Computer Software Redesign Project is shown in Figure 3.

The Darlington Shutdown Systems Redesign Project represents one of the first times that a production industrial software engineering process was designed, successfully, with the application of tool supported formal methods to specification and verification as a primary goal. As we have seen, this was

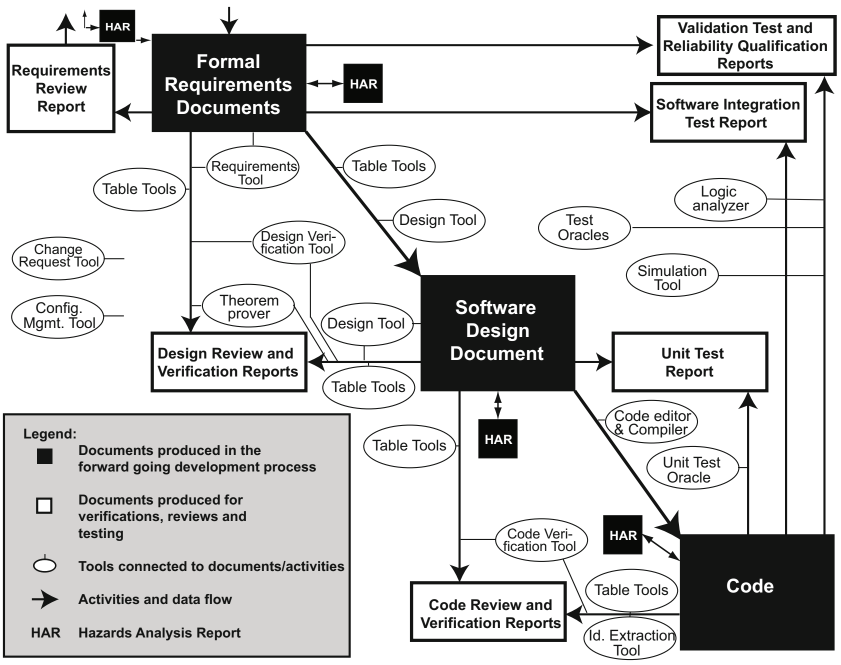

HAR
Formal Requirements Documents
Requirements Review Report
Validation Test and Reliability Qualification Reports
Software Integration Test Report
Requirements Tool
Table Tools
Design Tool
Design Verification Tool
Change Request Tool
Config. Mgmt. Tool
Theorem prover
Logic analyzer
Test Oracles
Simulation Tool
Software Design Document
Design Review and Verification Reports
Unit Test Report
Code editor & Compiler
Unit Test Oracle
Code Verification Tool
Code
Code Review and Verification Reports
Id. Extraction Tool
Legend:
■ Documents produced in the forward going development process
□ Documents produced for verifications, reviews and testing
Tools connected to documents/activities
Activities and data flow
HAR Hazards Analysis Report

Fig. 3. SDS1 lifecycle phases, documents and tools

necessitated by regulatory requirements, a situation that is becoming increasingly common for industries utilizing software in safety-critical applications. The major factors considered in choosing the particular formal methods for the Redesign Project were: (i) learning curve and ease of use and understanding of the formal specifications, (ii) ability to provide tool support, and (iii) previous history indicating the ability to successfully scale to industrial applications. We now address these three points in more detail.

Since tables are frequently used in many settings and provide important information visually, they are easily understood by domain experts, developers, testers, reviewers and verifiers. From the original Darlington licensing experience, and a trial example of the same verification procedure applied to a smaller scale Digital Trip Meter System [21], OPG had strong evidence that a verification procedure using tabular methods would meet the requirements of the Redesign Project. OPG's confidence in the use of tabular expressions was re-affirmed by domain experts working on SDS1 being able to read and understand the formal requirements specifications, documented almost exclusively by tabular expressions. Also, tabular expressions provide a mathematically precise notation with a formal semantics [19]. Other methods such as VDM or Z utilize unfamiliar notation and special languages with a significant learning curve [22]. The OPG Systematic Design Verification (SDV) procedure avoids this problem through the use of tabular notation in both the requirements and design documents utilized by all project team members. To create the tabular specifications, custom "light-weight" formal methods tools (in the sense of [23,24]) are used to help create and debug the tables from within a standard word processor. To perform the verification these tools then extract the tables from the documents and generate input files for SRI's Prototype Verification System (PVS) automated proof assistant [25].

Tabular methods are well suited to the documentation of the Shutdown System's control functions that typically partition the input domain into discrete modes or operating regions. Some of the other major benefits of this, and other, tool supported formal methods, include:

- Independent checks which are unaffected by the verifier's expectations,
- Domain coverage through the use of tools that can often be used to check *all* input cases – something that is not always possible or practical with testing,
- Detection of implicit assumptions and ambiguous/inconsistent specifications,
- Additional capabilities such as the generation of counter-examples for debugging, type checking, verifying whole classes of systems, etc.

The creation of the specialized tools that allowed verification to be done with the help of PVS played a large role in making the methods feasible for the larger Redesign Project. A further reason for the adoption of tabular methods is that they have been successfully applied to a wide variety of applications. In particular, they have been used successfully with PVS on problems such as the verification of hardware division algorithms similar to the one that caused the Pentium floating point bug [26].

There are some important points to note about the licensing of the redesigned Darlington Shutdown Systems. Compared with the licensing process for the original system, the redesign licensing process progressed remarkably smoothly. A major contributing factor was that the manufacturer (OPG) had asked the regulator to comment ahead of time on the deliverables for the licensing process. It is true that the regulator wanted to understand (and comment on) the software development process that was to be used in the project. However, the regulator's primary role was to evaluate the agreed upon set of deliverables. The evaluation was more in the form of an audit, in that *post factum*, the regulator specified a slice through the system for which a guided walkthrough was held. The regulator also reviewed major project documents.

3.4 Software Engineers Get It Wrong Again!

The aim of certification is to ascertain whether the product, for which a certificate is being sought, has appropriate characteristics. Certification should be a measurement based activity, in which an objective assessment of a product is made in terms of the values of measurable attributes of the product, using an agreed upon objective function.

Given the choice between focusing on process or product as a means of assessing whether software intensive systems possess the appropriate characteristics,

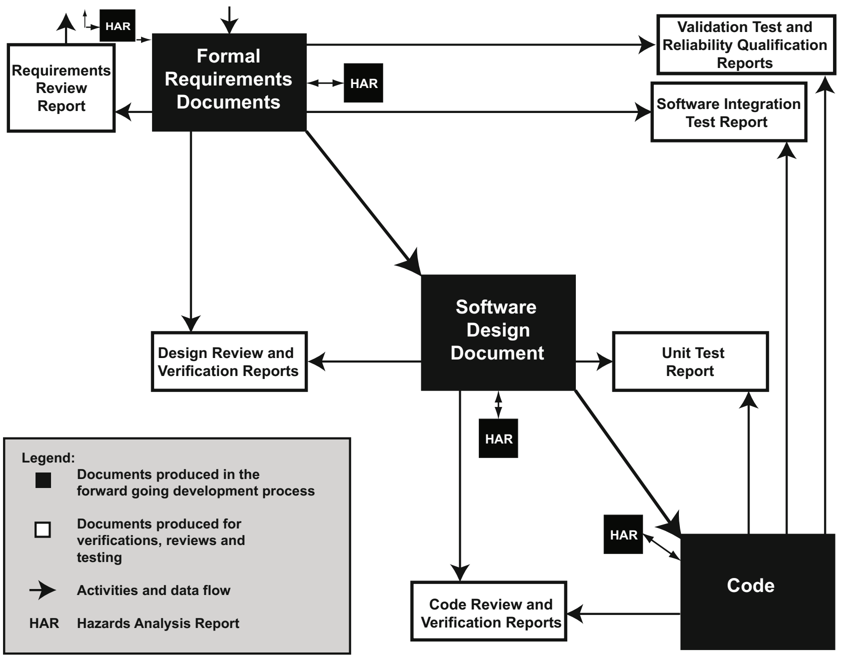

HAR
Formal Requirements Documents
Requirements Review Report
HAR
Validation Test and Reliability Qualification Reports
Software Integration Test Report
Software Design Document
Design Review and Verification Reports
Unit Test Report
HAR
Legend:
Documents produced in the forward going development process
Documents produced for verifications, reviews and testing
Activities and data flow
HAR Hazards Analysis Report
HAR
Code
Code Review and Verification Reports

Fig. 4. Idealized software development process

Software Engineers have again made the wrong choice. Classical engineers invariably use product-focused assessment and measurement in evaluating manufactured products. Software products are typically evaluated using process-focused standards. This is tantamount to trying someone on a murder charge - based solely on circumstantial evidence! The process based guarantee is a statistical one over all products, not a guarantee of an individual product.

The focus on CMM (and now CMMI) and other process-oriented standards was (perhaps) necessary to force companies to begin adopting proper engineering methods, but CMMI, as an example, has not progressed to the point where it achieves this.

We are advocating a *product-focused* approach to software certification - we are not saying, however, that software certification regimes should completely ignore the software process. We believe we will always need some notion of an *idealized software development process* in the software certification process. The idea is similar to Parnas and Clement's exhortation to "fake it" [27], in that there has to be agreement on mandatory documents produced during the software development process. For example, a simplified version of the SDS1 development process (Figure 3), could describe a mandated idealized process (see Figure 4 for example), and the certifiers could then evaluate product evidence such as documents and the application itself, without any consideration given to the quality of the development process that actually was followed.

4 Evaluating Process Is Easier

An obvious question arises: "Why did we (software engineers) turn to evaluating process rather than evaluating the final product(s) directly"? The answer, as usual in multi modal disasters, is complicated.

Evaluating the software development process is much easier than evaluating the software product itself.

- We have no real consensus on absolutely essential metrics for products.
- Ironically, even if we did have consensus on essential metrics, what metrics would help us evaluate the dependability of software products directly?
- It is widely accepted that testing software products completely is not possible. One of the major differences between software products and more traditional, physical products, is that the principle of continuity does not apply to software products. Since software engineers felt that even a huge number of test cases could not guarantee the quality of the product, we turned to supportive evidence, hoping that layers of evidence will add up to more tangible *proof* of quality/dependability.

Other disciplines introduced an emphasis on process. From general manufacturing to auditing, the world started putting more and more emphasis on process. We should be clear - there is a huge difference between the manufacture of a product and the certification of that product. We need good manufacturing processes and we also need effective certification processes. We have no hesitation in agreeing that a company needs a good software development process. When we discuss the difference between *process-focus* and *product-focus*, we are really looking at where the certification process should place its emphasis. The world-wide emphasis on manufacturing process made it easy for software certifiers/regulators to concentrate on evaluating a manufacturer's software development process and thus appear to be achieving something worthwhile in terms of certifying the manufacturer's products. We think that certification standards like CMMI and ISO 9000 tell us about the care and competence with which a company manufactures its products. It tells us very little directly about a specific product manufactured by the company.

5 Engineering Methods

We want to make the distinction between a proper engineering method, on the one hand, and having a well defined process as usually understood in software engineering, on the other hand. We begin by discussing the nature of engineering as a discipline. Over the years, engineering has been defined in a number of ways. A useful definition of engineering is the one used by the American Engineers' Council for Professional Development (AECPD). It defines Engineering as: "The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation and safety to life and property." Within this context, we need to consider what it means to use "engineering methods". A classic description of the Engineering Method was presented by B.V. Koen [28] in 1985. Koen presents the view that the Engineering Method is "the use of engineering heuristics to cause the best change in a poorly understood situation within the available resources". When one reads the explanation of this, it becomes clear that Koen is talking about a systematic process, that uses a set of *state-of-the-art* heuristics to solve problems. He also makes the point that the state-of-the-art is time-dependent. Before we discuss software engineering in particular, we lay the foundation by exploring concepts in the epistemology of science and engineering, and how they fit into the framework envisaged by the AECPD. We begin below by outlining the difference between engineering method and the use of craftsmanship principles based on intuition, but not proper science.

5.1 Engineering Intuition

(Sections 5 and 6 are heavily based on extracts from [29,30,31].) We typically think of *intuition* as the ability to know something without having to reason about it, or without being able to give a proper explanation, in the sense of science or engineering, of it. We have "intuitive" people in all walks of life - including engineering. In fact, we would go so far as to say that one role of a university engineering education is to try to foster "engineering intuition". However, we also claim that engineering intuition is not sufficient for the solution of engineering problems. We believe that engineering intuition guides the engineer in the choice of heuristics to try in the current problem. The engineering method, on the other hand, constrains the engineer to apply and test those heuristics in a very systematic way. So, intuition is not the difference between solving the problem and not solving it. Rather, it affects the speed with which the engineer arrives at a solution.

Exacerbating this intuition-science based engineering gap, it is our observation that there is a fundamental confusion between the scientifically and mathematically based practice of engineers and the day-to-day use of mathematics in the engineering *praxis*. This confusion results in discussions about "formal" versus "rigorous" (e.g., in the formal methods community), as if the dichotomy being explored was that between science/mathematics, on the one hand, and engineering, on the other. Actually, this difference resides completely in the science/mathematics camp. Only 'good" mathematicians and scientists are capable of doing rigorous mathematics. They know when they can leave out steps in proofs because they know or they are confident (they have good intuition) that the gaps can be filled. More typical mathematicians, scientists and engineers are not so good at doing this and have to rely more on not leaving such big gaps, or any at all. Thus less skilled mathematicians and scientists are capable of using only the formal, formulaic versions. Engineers use quite different scientific principles and mathematical techniques in their daily work [32]. It is with respect to these practical uses of mathematics and science that engineers develop "intuitions". (The hydraulic engineer called in to resolve a knotty problem certainly recalled Bernoulli's equations when his intuition told him that the relation between the diameter of the tube and its effect on water flow is not linear, a recollection that enabled him to explain certain water shortages at Neuschwanstein castle in upper Bavaria, a shortage that the operatic stage designer who designed the castle could not explain.) In [29], Haeberer and Maibaum formulated a number of ad hoc principles that we would like to put forward and discuss. We will do this in the context of some ideas from epistemology that we believe can provide a framework for discussions of the nature of (Software) Engineering and for forming critical judgments of contributions to research and practice in the subject. The principles are:

1. Intuition is a necessary but not sufficient basis for engineering design.
2. Intuitions are very difficult to use in complex situations without well-founded abstractions or mental models. (The term "mental model" is used here as a synonym of a somewhat vague abstraction of a /emphframework in the Carnapian sense; see below.)
3. An engineer has our permission to act on intuition only when:
   - Intuitions can be turned into mathematics and science, and
   - Intuitions are used in the context of normal design processes (see below).
4. The abstractions, mental models, cognitive maps, ontologies used by engineers are not the same as those used by mathematicians and scientists.

5.2 Carnap's Statement View

Carnap's Statement View of Scientific Theories provides a setting for discussing these issues [33,34,35]. The primary motivation for the Statement View was to explain the language (and practice) of science. Haeberer and Maibaum, [29,30], adapted it to engineering and provided a framework to discuss issues such as intuition, method, and mathematics. According to Carnap, a scientific theory, relating some theory to observable phenomena, always has two disjoint subtheories: a theoretical one, not interpreted in terms of observable entities, and a purely observational one, related by a set of correspondence rules (often measurement procedures), which connect the two subtheories. According to Carnap's metaphilosophy, when we state some theory (or set of theories) to explain a set of observations stated in the observational language, therefore constructing an instance of the Statement View, we are putting in place a framework by making some ontological commitments. Once a framework is established, we automatically divide our (scientific) questions into two disjoint subsets, so-called internal questions (e.g., is it true that $E = mc^2$, and is it true that the halting problem is undecidable in the classical computability framework?) and so-called external questions (e.g., does Church's thesis provide a useful model of computation or not?). To assert that something is of utility, we must have in mind some task for which it is to be of utility.

Engineering, perhaps unlike science, is a normative subject. In our case, we are interested in discussing software engineering as a proper engineering discipline and use it as a basis for certifying its artefacts. That is, we want to ensure that the framework (in the Carnapian sense) is of utility in accomplishing the stated or intended purposes of engineering, generally, and software engineering, in particular. According to Vincenti [32], the day-to-day activities of engineers consist of *normal design*, as comprising the improvement of the accepted tradition or its application under 'new or more stringent conditions' ". He goes on to say: "The engineer engaged in such design knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task". Note the relationship to the definition of engineering above and Koen's view of engineering.

6 What Makes Software Engineering an Engineering Discipline?

The ongoing debate on engineering versus intuition motivated Haeberer and Maibaum to investigate the epistemology of software engineering, the role of mathematics in the software engineering curriculum, and the engineering nature of software engineering. This section is very heavily based on portions of that work, [29]. Mathematics is undoubtedly an essential tool in engineering. There are software engineers who still claim that mathematics is not necessary for producing software. Luckily, fewer and fewer are willing to say this. The real problem here is not the fact that mathematics is necessary, but that people tend to associate the mathematics required with that of theoretical computer science, rather than some appropriate *engineering mathematics*. In addition, many software engineers underestimate the importance of the role of heuristics (see Koen) and systematic method (see Vincenti), used in engineering to guide and constrain intuition.

Vincenti [32] argues the case for engineering being different, in epistemological terms and, consequently as *praxis*, from science or even applied science: "In this view, technology, though it *may apply* science, is not the same as or entirely *applied* science". GFC Rogers [36] argues that engineering is indeed different from science. He argues this view based on what he calls "the teleological distinction" concerning the *aims* of science and technology: "In its effort to explain phenomena, a scientific investigation can wander at will as unforeseen results suggest new paths to follow. Moreover, such investigations never end because they always throw up further questions. The essence of technological investigation is that they are directed towards serving the process of designing and manufacturing or constructing particular things whose purpose has been clearly defined. [...] It is also more limited, in that it may end when it has led to an adequate solution of a technical problem." He makes a further claim: "Because of its limited purpose, a technological explanation will certainly involve *a level of approximation that is certainly unacceptable in science* (our emphasis)." Going back to the distinctions between the aims of science and engineering, we have, again from [36]: "We have seen that in one sense science progresses by virtue of discovering circumstances in which a hitherto acceptable hypothesis is falsified, and that scientists actively pursue this situation. Because of the catastrophic consequences of engineering failures - whether it be human catastrophe for the customer or economic catastrophe for the firm - engineers and technologists must try to avoid falsification of their theories. Their aim is to undertake sufficient research on a laboratory scale to extend the theories so that they cover the foreseeable changes in the variables called for by a new conception.

So science *is* different from engineering. Proceeding on this basis, we can ask ourselves what the *praxis* of engineering is (and ignore, at least for the moment, the specifics of scientific *praxis*). Vincenti defines engineering activities in terms of design, production and operation of artefacts. Of these, design and operation are highly pertinent to software engineering, while it is often argued that production plays a very small role, if any. In the context of discussing the focus of engineers' activities, he then talks about *normal design* as comprising "the improvement of the accepted tradition or its application under new or more stringent conditions' ". He goes on to say: "The engineer engaged in such design knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has good likelihood of accomplishing the desired task" (see [34].)

Another important aspect of engineering design is the organizing principle of hierarchical design: "Design, apart from being normal or radical, is also multilevel and hierarchical. Interesting levels of design exist, depending on the nature of the immediate design task, the identity of some component of the device, or the engineering discipline required." An implied, but not explicitly stated, view of engineering design is that engineers normally design *devices* as opposed to *systems.* A device, in this sense, is an entity whose design principles are well defined, well structured and subject to *normal* design principles. A system, the subject of *radical* design, in this sense, is an entity, which lacks some important characteristics making normal design possible. Examples of the former given by Vincenti are aeroplanes, electric generators, turret lathes; examples of the latter are airlines, electric-power systems and automobile factories. The software engineering equivalent of devices may include compilers, relational databases, PABXs, etc. Software engineering examples of systems may include air traffic control systems, mobile telephone networks, etc. It would appear that systems become devices when their design attains the status of being normal. That is, the level of creativity required in their design becomes one of systematic choice, based on well-defined analysis, in the context of standard definitions and criteria developed and agreed by engineers. This is what makes everyday engineering practice possible and reliable.

Let us now consider the particular characteristics of software engineering as a discipline. We want to address the question: "Is the knowledge used by software engineers different in character from that used by engineers from the conventional disciplines?" The latter are underpinned not just by mathematics, but also by some physical science(s) - providing models of the world in terms of which artefacts must be understood. (The discussion above illustrates this symbiosis.) We might then ask ourselves about the nature of the mathematics and science underlying software engineering. It is not surprising, perhaps, that a large part of the mathematics underlying software engineering is formal logic.

Logic is the mathematics of concepts and abstractions. Software engineering may be distinguished from other engineering disciplines because the artefacts constructed by the latter are physical, whereas those constructed by the former are conceptual. There are some interesting and significant differences between the two kinds of mathematics and engineering mentioned above. One of these is that the real world acts as a (physical) constraint on the construction of (physical) artefacts in a way which is more or less absent in the science and engineering of concepts and abstractions. There seems to be a qualitative difference in the dimensions of the design space for software engineering as a result.

What distinguishes the theoretical computer science and software engineering dependence on logic is the day-to-day invention of theories (models) by engineers and the problems of size and structure introduced by the nature of the artefacts with which we are dealing in software engineering. Now, the relationship between the mathematics of theoretical computer science and that of (formal methods and) software engineering should be analogous to the difference between conventional mathematics and its application and use in engineering. As an example, program construction from a specification has a well-understood underlying mathematics developed over the last 25 years. (We are restricting our attention to sequential programs. Concurrency and parallelism are much less mature topics.) We might expect to find a CAD tool for program construction analogous to the "poles and canvas" model used in electronics for the design of filters. Instead, what we find is just a relaxation on the exhaustiveness requirement, i.e., we can leave out mathematical steps (proofs of lemmas) on the assumption that they can be filled in if necessary, the so-called rigorous approach. Where is the abstract model (analogous to the "poles and canvas" one) that encapsulates the mathematics and constrains manipulation in a (mathematically / scientifically) sensible manner?

6.1 An Epistemological Framework for Software Engineering

As Carnap (and others) have pointed out, an ontological framework, cannot be said to be correct or incorrect, it can only be of some utility, or not. Hence, in discussing a framework for software engineering, we are left with the task of convincing our colleagues that the proposed framework will be of some utility. We outline some details of the software engineering framework we proposed in terms of Figure 5, illustrating that we can give the diagram a semantics. That is, all the objects and relationships and processes denoted by the diagram could be given exact, mathematical/scientific definitions. (We say "could" because some of the relationships are presently the subject of research!) Nor do we claim that this is the only framework of utility for software engineering. (We only induce the reader to think about it as to be the last word in software engineering frameworks

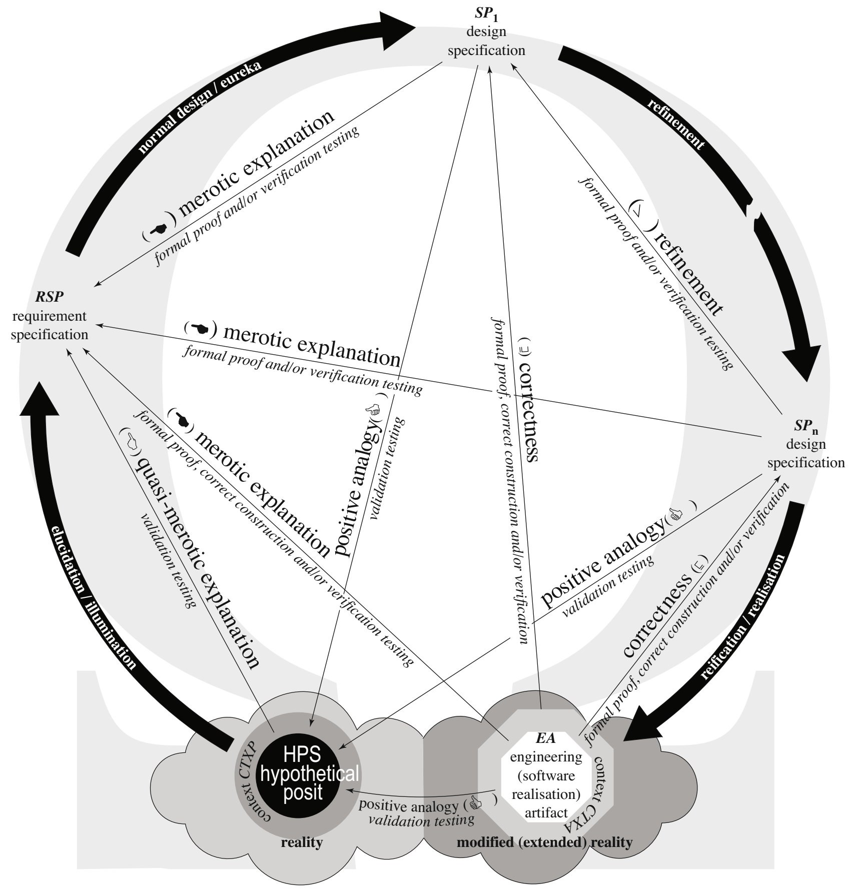

SP1 design specification
RSP requirement specification
SPn design specification
HPS hypothetical posit
context CTXP
reality
EA engineering (software realisation) artifact
context CTXA
modified (extended) reality
normal design / eureka
refinement
reification / realisation
elucidation / illumination
merotic explanation
formal proof and/or verification testing
refinement
formal proof and/or verification testing
merotic explanation
formal proof and/or verification testing
positive analogy
validation testing
correctness
formal proof, correct construction and/or verification
merotic explanation
formal proof, correct construction and/or verification testing
quasi-merotic explanation
validation testing
positive analogy
validation testing
correctness
formal proof, correct construction and/or verification
positive analogy
validation testing

Fig. 5. Carnapian Framework for Software Engineering from [29]

by means of the background omega letter!) There may be others, more or less detailed, that are of equal utility. Actually, to assert that something is of utility, we must have in mind some task for which it is to be of utility. In our case, we are interested in making software engineering a proper engineering discipline (see, e.g., [30]) and supporting the practice of certification. Superficially, the elements of the diagram (objects and relationships) are just a more or less detailed version of diagrams used to represent the development process of software systems from conception to final realization as an executable system. As examples to illustrate that the elements of the diagram can be formalized, we give the following definitions: *Correctness* is a relation between two constructed artefacts asserting that the properties required by one are preserved in the other. Preservation of properties may be mediated by translation (between ontologies). Also, preservation does not exclude the inclusion of new (consistent) properties. *Validation* is the activity of determining, by means of experiments (i.e., testing), whether or not we are constructing the appropriate positive analogy. *Positive analogy* is a relation between two entities (frameworks) consisting of a map between the two underlying ontologies (an interpretation between languages), which correlates positively (in the sense of essential and non-negative) properties of the source of the mapping with the positive properties of the target. We call the source an *iconic* model of the target. *Testing* is the application of tests. A *test* is an experiment to determine if some entity may have (can be assumed to have) some ground property (in the sense of logic).

We can use the framework to demonstrate the necessity of testing. We say that a relation is *epistemological* if it cannot, in principle, be formally (i.e., mathematically) corroborated. Hence, whether the relation holds or not is inherently contingent. That is, the existence of the relationship requires some form of testing (or experiment, in the sense of science) for its establishment. Despite its logical character, the truth of a logical relation is often checked by verification testing, in which case the character of this truth becomes contingent. The truth of an epistemological relation cannot be definitively established, just as a scientific theory cannot be "proved" once and for all.

6.2 Evidence and Measurement

Certification of any form requires evidence supporting the case for certification and judgements based on this evidence (the utility function mentioned above). The framework outlined above is intended to provide the foundation for building a framework for certification. The epistemological basis of science has established the principles and practice of using evidence in science. The adaptation to engineering ([29,30,31]), and software engineering in particular, enables us to apply informed judgements about proposals related to software development and certification. In particular, it provides a setting in which definitions of measurable attributes of software and their role in certification can be scientifically assessed. It also enables us to attempt assessments of their utility for the objectives of certification. It is on this basis that process-based approaches to certification should be rejected as insufficient to make certification judgements. The process-based assessment may well provide a statistical basis for confidence about the products of the process. But it does not provide sufficient levels of confidence about a particular product. The only way to obtain sufficient confidence about the product itself is to measure relevant product attributes and then make a judgement based on this evidence. Normally, in science, it is not sufficient for experiments to *usually* be successful in verifying some hypothesis about a theory. If an experiment fails to verify the hypothesis, there are only two possibilities: the experimental procedure was faulty, or the theory on which the hypothesis was based is false. (The former is probably the more usual cause for failure.) In the case of process-based predictions, there is a third possibility, namely that the process-based evidence was incorrect in relation to this particular product. Hence, the process-based approach does not pass the utility test: it fails to be reliable enough, and cannot, in principle, be "improved" to overcome this shortcoming. There is a lot more that could usefully be learned from the epistemology of science and engineering. In particular, the concept of explanation in science ([37]) might form a useful basis for assessing the evidence produced by a manufacturer to support the licensing of a product. A basic question that needs to be asked during the assessment of the evidence is: Does the evidence provide a sufficiently good explanation (in this technical sense borrowed from epistemology) of the effectiveness and safety of the product to be accepted as a guarantee warranting certification? However, we shall not pursue this interesting topic here.

7 The Certification Initiative

In mid-2005, a number of researchers in academia and industry decided to start working on a *Certification Initiative.* The initiative was spearheaded by members of the Software Quality Research Laboratory (SQRL) at McMaster University in Canada, primarily Alan Wassyng, Tom Maibaum, Mark Lawford and Ryszard Janicki. Within a very short time, a small group of "Founding Members" was formed:

- SQRL faculty - McMaster University (Canada)
- Jo Atlee, University of Waterloo (Canada)
- Marsha Chechik, University of Toronto (Canada)
- Jonathan Ostroff, York University (Canada)
- Stefania Gnesi, ISTI-CNR (Italy)
- Connie Heitmeyer, NRL (USA)
- Brian Larson, Boston Scientific (USA)

The idea was to put software certification on the primary research agenda, and a number of activities have since resulted directly from this initiative.

7.1 The PACEMAKER Grand Challenge

With some encouragement from SQRL and Jim Woodcock, Brian Larson of Boston Scientific (Guidant), worked hard to release a natural language specification of a ten year-old model of a pacemaker. The specification forms the basis of a *Grand Challenge* to the software engineering community [38]. The PACEMAKER specification has also been used as a project for the first Student Contest in Software Engineering (SCORE) that is part of the 31^{st} International Conference on Software Engineering (ICSE 2009). A reference hardware platform was designed by students at University of Minnesota, supervised by Brian Larson, and Mark Lawford arranged to have 50 (slightly modified) PACEMAKER boards manufactured. They have been available through SQRL [39] for use in the PACEMAKER Grand Challenge, SCORE, and other academic endeavours.

The benefits we hope to realize from the PACEMAKER Grand Challenge and related activities are:

- Demonstrate the state-of-the-art in safety-critical software development.
- Provide a comparison of development methods.
- Develop product-focused certification methods.

7.2 The Software Certification Consortium

During 2007, SQRL researchers and Brian Larson spearheaded the formation of the Software Certification Consortium (SCC). Its purpose is to develop and promote an agenda for the certification of systems containing software (ScS), by forming a critical mass of industry, academic and regulatory expertise in this area. We held an inaugural meeting in August 2007, at SEI's Arlington location, and two further meetings in December 2007 (hosted by Mats Heimdahl, University of Minnesota) and late April 2008 (hosted by Austin Montgomery and Arie Gurfinkel, SEI). The current steering committee for SCC is:

- Richard Chapman (U.S. Federal Drug Administration)
- John Hatcliff (Kansas State University)
- Brian Larson (Boston Scientific)
- Insup Lee (University of Pennsylvania)
- Tom Maibaum (McMaster University)
- Bran Selic (Malina Software)
- Alan Wassyng (McMaster University)

A description of the goals of SCC, its objectives, and SCC's view of the major hurdles facing us in meeting those objectives was presented at SafeCert 2008 [40]. The goal of certification, SCC's goals and objectives are repeated below. The hurdles and their descriptions are also paraphrased below.

Goal of Certification - SCC: The *Goal of Certification* is to systematically determine, based on the principles of science, engineering and measurement theory, whether an artefact satisfies accepted, well defined and measurable criteria.

SCC Objectives:

(i) To promote the scientific understanding of certification for Systems containing Software (ScS) and the standards on which it is based;
(ii) To promote the cost-effective deployment of product-focused ScS certification standards;
(iii) To promote public, government and industrial understanding of the concept of ScS certification and the acceptance of the need for certification standards for software related products;
(iv) To investigate and integrate formal methods into ScS certification and development;
(v) To co-ordinate software certification initiatives and activities to further objectives i-iv above.

Goals to Achieve SCC Objectives: The *Primary Goals* are:

(i) Develop and document generic certification models that will serve as a framework for the definition of domain specific regulatory and certification requirements; and

(ii) Proof of concept: Develop and document software regulatory requirements that help both developers of the software and the regulators of the software in the development of safe, reliable software applications in specific domains.

A number of *Detailed Goals* were also identified: (i) Use existing software engineering and formal methods knowledge to develop appropriate evidence-based standards and audit points for critical software in specific domains, including hard real-time, safety-critical systems; (ii) Create software development methods that comply with the above standards and audit points for the development of critical software; and (iii) Research and develop improved methods and tools for the development of critical software.

Hurdles in Achieving Objectives: During the December 2007 SCC meeting, participants identified the following 9 hurdles. The first 4 of these were voted as the top 4 hurdles, in the order shown. The remaining 5 hurdles were not prioritized.

1. Clarity of regulator's expectation and method of communicating with the regulator. *Application developers do not know what to produce, and often have to pay consultants - who get it wrong.*
2. Lack of clear definition of evidence and how to evaluate it. *We know very little about the effectiveness of attributes and metrics related to dependability, and do not really understand how to combine different evidentiary artefacts.*
3. Poor documentation of requirements and environmental assumptions. *We need accurate and complete requirements in order to produce evidence of compliance. Poor requirements invariably lead to poor products.*
4. Incomplete understanding of the appropriate use of inspection, testing and analysis. *We do not know when to use inspection, testing and mathematical analysis to achieve specific levels of dependability.*

- No overarching theory of coverage that enables coverage to accumulate across multiple verification techniques. *In our opinion, this is the most important hurdle of all. It was not voted #1 simply because it was felt that we need to tackle easier hurdles first. We know of no single quality assurance technique that is solely sufficient for effective certification. Each of these techniques differs in strength of properties verified, types of behaviours covered, and the life-cycle stage in which they are most naturally applied. Sharing coverage across techniques via a single unified framework will enable the successes of one technique to reduce the obligations of associated techniques, and will clarify gaps in verification that must be filled by other techniques. The most convincing arguments of correctness will rely on being able to accurately state in quantitative ways how multiple verification techniques each contribute evidence of overall correctness.*
- Theories of coverage for properties like timing, tolerances as well as concurrent programs. *Structural coverage for testing plays a key role in development*

and certification of safety-critical software. Existing coverage measures fail to take into account properties such as timing and tolerance ranges for data values and the degree to which interleavings in concurrent computations are exercised As a result, even development efforts that succeed in achieving high levels of mandated coverage measures often fail to fully explore and validate common sources of program faults.

- Hard to estimate a priori the V&V and certification costs. *Currently, it is difficult to make a business case for the introduction of formal techniques, because it is difficult to estimate both the time required to carry out various forms of formal analysis and the reductions that can be obtained either in costs of the certification process itself or long-term costs associated with fewer defects found late in the development life-cycle, greater reuse in subsequent development of similar systems, fewer recalls of deployed systems, and decreased liability costs.*
- Lack of interoperable tools to manage, reason, and provide traceability.
- Laws, regulation, lawyers and politics. *Certification has legal implications, and as difficult as the technical problems may be, political considerations complicate the process immeasurably.*

8 Research Overview

Below, we list a number of broad research questions that we need to answer. In addition to these questions, the specific *hurdles* that were identified above in section 7.2 round out an initial research agenda for software certification.

- Is there a generic notion of certification, valid across many domains?
- What, if anything, needs to be adapted/instantiated in the generic model to make it suitable for use in a particular domain?
- What benefit do we achieve by creating product-specific software certification standards and processes?
- What simple process model is sufficient to enable the "faking" of real processes and providing a platform for evaluation by certification authorities?
- What is the difference between software quality, of a certain level, and certifiability?
- In what situations can we safely use process-based properties as a proxy for product qualities?
- If we have levels of certifiability, as in the Common Criteria, how does the mix of formal verification and testing change with the level?
- Since evaluating evidence about software is an onerous task, how can we assist evaluators to perform their tasks by providing tools? (Amongst examples of such tools may be proof checkers (to check proofs offered in evidence), test environments (to re-execute tests offered in evidence), data mining tools to find "interesting" patterns in artefacts, etc.)

9 Conclusions

Engineering methods are identifiable as those that are systematic, depend on theories and heuristics derived from relevant basic sciences, and rely on being able to measure relevant values in a repeatable way. Software engineering methods are moving - slowly - in that direction. Another important factor is the role of measurement in engineering and science. One of the major problems facing us is that we have not built or discovered adequate, meaningful metrics that can be used to measure attributes of software artefacts, either to support engineering methods, or more crucially, certification regimes. Unfortunately, existing software certification methods are primarily focused on evaluating the software development process that was used to develop the system being certified. This does not seem to qualify as an engineering approach to certification of software products. Almost all engineering certification regimes we have seen are product-focused. In any case, it seems that reliance on indirect evaluation of artefacts is a poor way of determining whether a product is effective, safe, and dependable.

We believe that software certification methods should be primarily product-focused. There are technical, social, commercial and political pressures being brought to bear on this movement. We also believe that there is growing agreement on this issue. We also think that there are good reasons why we should be examining whether or not we should be developing not just domain specific software certification methods, but even product specific, product-focused software certification methods. Interestingly, the FDA is also considering similar ideas [41], which would be a major change in direction for their certification regime.

The previous section briefly describes a research agenda that we believe will lead us to fundamental results that will aid in building new product-focused software certifications processes. In order to accomplish the goals of certification for software, we also have to undertake fundamental research on appropriate metrics for software and software design artefacts. We must develop significantly better engineering heuristics and methods, to make software development more reliable and repeatable, akin to classical engineering disciplines. Almost certainly, these heuristics and methods, and the accompanying certification regimes, will also be domain specific. This appears to be an inescapable attribute of engineering.

References

1. ASTM Standard F977: Standard Consumer Safety Specification for Infant Walkers. ASTM International, West Conshohocken, PA, USA (2000)
2. Regulatory Review and Recommendation Regarding Baby Walkers Pursuant to the Hazardous Products Act. Health Canada (April 2004)
3. General Principles of Software Validation; Final Guidance for Industry and FDA Staff. U.S. Dept. of Health and Human Services: FDA (January 2002)
4. Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices; Guidance for Industry and FDA staff. U.S. Dept. of Health and Human Services: FDA (May 2005)
5. `http://www.sei.cmu.edu/cmmi/` (March 2009)

6. Common Criteria for Information Technology Security Evaluation: Part 1: Introduction and general model, Version 3.1, Revision 1 (2006)
7. Common Criteria for Information Technology Security Evaluation: Evaluation methodology, Version 3.1, Revision 2 (2007)
8. Parnas, D.: The use of precise specifications in the development of software. In: IFIP Congress, pp. 861–867 (1977)
9. Heninger, K.L.: Specifying software requirements for complex systems: New techniques and their applications. IEEE Trans. on Soft. Engineering 6(1), 2–13 (1980)
10. Parnas, D.: Using Mathematical Models in the Inspection of Critical Software. In: Applications of Formal Methods, pp. 17–31. Prentice-Hall, Englewood Cliffs (1995)
11. Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D.: Requirements specification for process-control systems. IEEE Transactions on Software Engineering 20(9), 684–707 (1994)
12. Heimdahl, M.P.E., Leveson, N.G.: Completeness and consistency in hierarchical state-based requirements. IEEE Trans. on Soft. Eng. 22(6), 363–377 (1996)
13. Heitmeyer, C., Kirby Jr., J., Labaw, B., Archer, M., Bharadwaj, R.: Using abstraction and model checking to detect safety violations in requirements specifications. IEEE Transactions on Software Engineering 24(11), 927–948 (1998)
14. Crow, J., Di Vito, B.L.: Formalizing Space Shuttle software requirements: Four case studies. ACM Trans. on Soft. Eng. and Methodology 7(3), 296–332 (1998)
15. Archinoff, G.H., Hohendorf, R.J., Wassyng, A., Quigley, B., Borsch, M.R.: Verification of the shutdown system software at the Darlington nuclear generating station. In: International Conference on Control and Instrumentation in Nuclear Installations, Glasgow, UK, The Institution of Nuclear Engineers (May 1990)
16. Parnas, D.L., Asmis, G.J.K., Madey, J.: Assessment of safety-critical software in nuclear power plants. Nuclear Safety 32(2), 189–198 (1991)
17. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 133–153. Springer, Heidelberg (2003)
18. Joannou, P., et al.: Standard for Software Engineering of Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 1 (January 1995)
19. Janicki, R., Wassyng, A.: Tabular representations in relational documents. Fundamenta Informaticae 68, 1–28 (2005)
20. Wassyng, A., Lawford, M.: Software tools for safety-critical software development. Software Tools for Technology Transfer (STTT) 8(4-5), 337–354 (2006)
21. McDougall, J., Viola, M., Moum, G.: Tabular representation of mathematical functions for the specification and verification of safety critical software. In: SAFECOMP 1994, pp. 21–30. Instrument Society of America, Anaheim (1994)
22. Wassyng, A., et al.: Choosing a methodology for developing system requirements. Ontario Hydro/AECL SD-2 Study Report (November 1990)
23. Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., Hamilton, D.: Experiences using lightweight formal methods for requirements modeling. IEEE Transactions on Software Engineering 24(1), 4–14 (1998)
24. Heitmeyer, C., Kirby, J., Labaw, B., Bharadwaj, R.: SCR*: A toolset for specifying and analyzing software requirements. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 526–531. Springer, Heidelberg (1998)
25. Shankar, N., Owre, S., Rushby, J.M.: PVS Tutorial. In: Computer Science Laboratory, SRI International, Menlo Park, CA (February 1993)
26. Rueß, H., Shankar, N., Srivas, M.K.: Modular verification of SRT division. Formal Methods in Systems Design 14(1), 45–73 (1999)

27. Parnas, D., Clements, P.: A rational design process: How and why to fake it. IEEE Trans. Software Engineering 12(2), 251–257 (1986)
28. Koen, B.: Definition of the Engineering Method. ASEE (1985)
29. Haeberer, A.M., Maibaum, T.S.E.: Scientific rigour, an answer to a pragmatic question: A linguistic framework for software engineering. In: ICSE 2001 Proceedings, pp. 463–472. IEEE Computer Society, Washington (2001)
30. Maibaum, T.: Mathematical foundations of software engineering: a roadmap. In: ICSE 2000 Proceedings, pp. 161–172. ACM, New York (2000)
31. Maibaum, T.: Knowing what requirements specifications specify. In: PRISE 2004, Conference on the PRInciples of Software Engineering, Technical Report, University of Buenos aires, keynote address in memory of Armando Haeberer (2004)
32. Vincenti, W.G.: What Engineers Know and How They Know It: Analytical Studies from Aeronautical History. The Johns Hopkins University Press, Baltimore (1993)
33. Carnap, R.: Empiricism, semantics, and ontology. Revue Internationale de Philosophie 11, 208–228 (1950)
34. Carnap, R.: The Methodological Character of Theoretical Concepts. In: Minnesota Studies in the Philosophy of Science, vol. II, pp. 33–76. U. of Minnesota Press (1956)
35. Carnap, R.: Introduction to the Philosophy of Science. Dover Publications, New York (1995)
36. Rogers, G.: The Nature of Engineering. The Macmillan Press Ltd., Basingstoke (1983)
37. Hempel, C.: Aspects of Scientific Explanation and Other Essays in the Philosophy of Science. The Free Press, New York (1965)
38. `http://sqrl.mcmaster.ca/pacemaker.htm`
39. `http://www.cas.mcmaster.ca/wiki/index.php/Pacemaker`
40. Hatcliff, J., Heimdahl, M., Lawford, M., Maibaum, T., Wassyng, A., Wurden, F.: A software certification consortium and its top 9 hurdles. In: Proceedings of SafeCert 2008. ENTCS (2008) (to appear)
41. Arney, D., Jetley, R., Jones, P., Lee, I., Sokolsky, O.: Formal methods based development of a PCA infusion pump reference model: Generic infusion pump (GIP) project, pp. 23–33 (June 2007)