What makes a Code Review Trustworthy?

Stacy Nelson[†] and Johann Schumann[‡]
[†]Nelson Consulting Company, NelsonConsult@aol.com
[‡]RIACS / NASA Ames, schumann@email.arc.nasa.gov

Abstract

Code review is an important step during the process of certifying safety-critical software because only code that passes review can be implemented. Reviews are performed by review boards composed of highly skilled and experienced computer scientists, engineers and analysts who generally rely upon a checklist of properties ranging from high-level requirements to minute language details. While many checklists and coding standards exist, the actual decision of which properties are most important is generally based on the experience of the person in charge.

This paper addresses the questions: How can code review ensure certification of trustworthy code? and Is code review trustworthy? We surveyed technical leaders at NASA and the Aerospace industry to find out which properties are most important during the code review. To make analyze easier, the most common properties have been classified along different "views", ranging from a standards-oriented view (defined as the properties needed to satisfy a specific standard) to a tool-oriented view.

In this paper, we present this classification together with a summary of findings and feed-back from the survey. We also discuss how a more uniform view on properties of code review and tool capabilities can result in increased trust for safety-critical software.

1 Introduction

Code review is an important step during the process of certifying safety-critical software. During this step, the code is manually (or automatically) inspected for a number of weaknesses and properties. The code only passes the review if it exhibits all required properties. These properties span a wide spectrum ranging from high-level requirements like "Are all software changes documented?" to nitty-gritty language details like the correct and safe use of parenthesis in C preprocessor macros.

Already in 1976, Fagan [7] described design and software walk-throughs which were carried out at IBM. Since then, a rich body of checklists, coding standards, and literature about this topic has been published. Despite the existing lists of properties, the actual decision of which properties are most important is usually based on the experience of the person in charge. In this paper, we address the question how code review can contribute to certification of trustworthy code.

We made a survey within NASA and the Aerospace industry regarding which of the properties are most important during the code review, and if there exists a subset of properties, that, when fulfilled, the code is considered to be trustworthy.

For a better presentation, we have developed a classification of the most common properties along different "views", for example, a standards-oriented view (defined as the properties needed to satisfy a specific standard like IEEE 12207, MIL STD 498, or DO-178B) or a tool-oriented view ("Does the code pass automated inspection by a respected tool?").

Why is trustworthiness of the code review important? Both NASA and the FAA rely upon results of review boards to detect and report errors. Code can only be implemented when the review board determines that the margin of error falls within an acceptable range. For safety-critical software this can mean zero-defects. Review boards are generally comprised of senior computer scientists, engineers and analysts with a reputation for excellence. During the review their primary concerns include correctness of the code with respect to the software requirements, correctness/robustness of the software architecture and conformance to applicable coding standards [6]. The team being reviewed must demonstrate their technical competence by answering tough questions about their code. Much of the review process relies upon the reviewer's ability to trust that the solution will work.

The paper proceeds as follows. In Section 2, we will introduce a classification of properties for code review according to different "views". We then will discuss some of the more important views in detail (Section 3). Section 4 and 5 presents a survey within NASA and the Aerospace industry and discusses findings and feed-back from the survey. In Section 6, we discuss future work and conclude.

2 Views on Code Review

2.1 Overview

When a program or a piece of code is subject to a code review, usually a team of experts has a close look at the source code and all other related artifacts that are produced during the software development process (e.g., documentation, log files). Where applicable, tools are used to support this process. Based upon the findings, the code review team makes recommendations about code improvements or required changes, or it concludes that the code successfully passed the review. In principle, there is a huge number of of different aspects and properties which could be inspected during a code review. These properties can range from high-level (managerial) aspects, like "Have all change-requests documented appropriately?" to details in the source code, like the correct use of parentheses in C-preprocessor macros. It is obvious that this list is open-ended. Thus, in reality, never all properties can be checked. Rather, the members of the code-review team decide on a subset of checks which then actually will be carried out.

This process of property selection, however, is often quite arbitrary and strongly depends on the expertise and experience of the person(s) doing code review. In some cases, code review has to be performed along the lines of given "check-lists" (sometimes being part of a software process) and anecdotal evidence. Only very few approaches are based upon statistically solid data (e.g., [22]). Thus the question arises: Is code review trustworthy? Or only, if you can just trust the expertise of the reviewer? In order to have a more objective metrics on the quality of a code review, there should be a "good" list of properties to be checked during a good review for trustworthy code. It should be mentioned, however, that this does not imply that the code is trustworthy, just because it passed a good code review. According to recent eWorkshops held by the Fraunhofer Center for Experimental Software Engineering [26] to capture expert knowledge about software development, peer reviews catch more than half (about 60%) of software defects. Therefore, code review can lead to desired results only in combination with other verification and validation (V&V) techniques.

From the above discussion, it is obvious, that for any realistic code review, not all possible properties can be reviewed and checked. In order to facilitate the selection of the essential checks, we define each property within a metric spanned by *importance* and *difficulty*. Difficulty indicates, how much effort needs to be spent to perform the review item under consideration. For example, the property "Compiles without warnings" can be checked easily and in relatively short time, whereas an accurate check on pointer arithmetic can be very difficult and time-consuming.

The notion of "importance" is more difficult to define, because it involves various aspects, like risk and frequency of occurrences. For example, the check for a concise documentation layout is definitely important, and violations will occur frequently. However, there is no immediate risk of an imminent major program failure. On the other hand, an array-bounds violation is found much less frequent, but if there is one, consequences can be devastating. A typical example of such a failure are security vulnerabilities due to buffer overflow [18].

Therefore, we use a metric which is common in traditional risk analysis, namely that of frequency and risk (or cost in case of failure). There, the space is divided into four quadrants, along the lines low risk, high risk, and low frequency vs. high frequency incidents. When we combine the metrics, we obtain a diagram as shown in Figure 1. Here the axes are labeled according to "difficulty" and "frequency/risk" and the location of several properties is shown. Often, code reviewers are taught to find errors effectively, so "it is prudent to condition them to seek the high-occurrence, high-cost error types" [7].

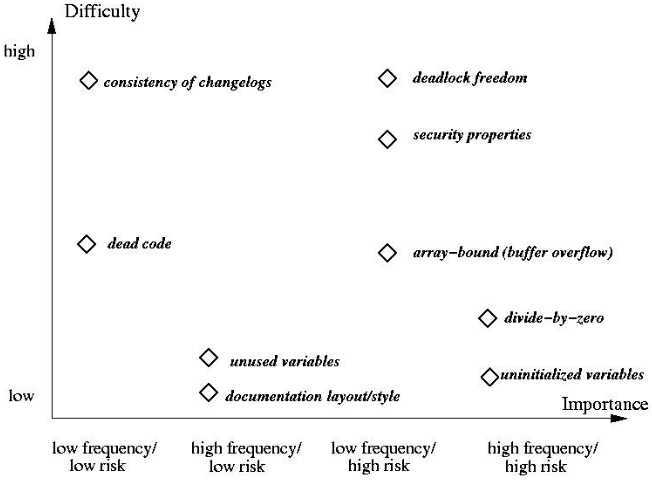

Figure 1. Metrics for Code Review Properties.

There is yet another dimension in this problem, namely that of *coverage*. Although it is clear that even in the hypothetical case that all conceivable code review tasks are performed, the program is not necessarily 100% correct. So, for example, a review of the requirements or tests with actual data are not part of a code review. Nevertheless, a good and trustworthy code review has to be based on a list of properties which, combined, provide "good coverage". With this coverage we do not only mean the percentage of source lines (and lines of documentation) which is being reviewed, but also that a broad variety of different issues are investigated. For example, a code review, solely focusing on array-bound violations is most certainly not sufficient, even if all source lines are investigated. A much better coverage can, for example, be obtained, when variable initialization, array-bounds, and parameter handling are checked at the most important ”hot spots”.

In order to cover these issues, we looked at the various approaches to the source-code analysis required for V&V or certification as it is found in the literature. Often, these approaches focus on certain aspects (e.g., following a standard, analyzing C++ code). We identified six different “views” on code review that are representative of the most commonly documented approaches. In the following, we briefly characterize these views. An in-depth discussion of some of these views follows in Section 3.

2.2 The Views

Process/standards-oriented View. This view describes properties about concurrence of applicable standards (i.e., IEEE, RTCA DO-178B, etc.) and processes. A typical standards related property in this category could be “Has the SDD been implemented correctly with respect to the given standard?”. An example more on the process-oriented side would be a property like “code must compile without producing error or warning messages”.

This view can be further subdivided into a *phase-product-oriented view* which splits up the relevant properties according to the artifacts and the phases of the software life-cycle during which they are generated or modified. Products have the following four categories:

Documentation relates to sufficient documentation in accordance with applicable standards, as well as adequate comments in code.

Code considers whether code compiles cleanly without warnings and also considers whether the code adheres to coding standards.

Software revisions judges whether revisions are documented appropriately including rationale and approval for revision.

Maintainability/Reuse relates to using good coding techniques for easily decipherable code and competent configuration management techniques to ensure that source and object code are always synchronized.

For each of these categories and for each phase, specific properties should be fulfilled. For example, typical properties for the phase ”Implementation” could be: ”Does it compile?” (code), ”Are enough comments in the code?” (documentation), ”Are all versions of the implemented code stored/documented carefully?” (revisions), and ”Are sufficient provisions in the implementation for code maintenance or re-use?” (maintainability).

A slightly different approach to classification of important properties along the different phases of the Software Life Cycle is proposed in [28] and is referred to as a *QA-oriented view* (Quality Assurance view). A number of abstract properties like “completeness”, “consistency”, “correctness” etc., are identified. Then, for each phase and each kind of review, a detailed and tailored definition of the property is provided. In the standards [13] and [27], these abstract properties are called *quality attributes*. For example (from [28], B.2.1), the ”robustness” property applies to software requirements, software design and code review phases, but it is defined differently for each phase. For the Software Requirements Review, ”robustness” means ”are there requirements for fault tolerance and graceful degradation?”. The ”robustness” property for the Software Design Review means ”are all SRS requirements related to fault tolerance and graceful degradation addressed in the design?”. Finally, ”robustness” during source code review means ”does the code protect against detectable run-time errors (e.g., range array index values)?”. The QA view will be discussed in more detail in Section 3.1.

Programming-language Specific View. Most check-lists for source code analysis (e.g. [16, 2, 25]) describe properties which are (very) specifically tailored toward a selected programming language. Usually, these lists are structured according to the syntactic elements of the programming language (declarations, initialization, assignment, flow control, etc). Most of these lists provide great detail about specific language constructs that are particularly error-prone and thus require special care during code review. These lists provide the most extensive view on individual properties that can be automatically checked. On the other hand, because of their size, such lists are usually of limited use, unless a very experienced person can select a subset of the most important properties. A more detailed description of that view will be presented in Section 3.2.

Application/Generic View. This view distinguishes between two different kinds of properties: generic or language specific properties as discussed above, and domain or application specific properties. For example, the property ”are all array indices within the correct range?” clearly belongs to the first category. Consistency of physical units (e.g., ”are all lengths measured in meters?”) belongs to the second category. The Mars Climate Orbiter (MCO) incident is a typical example for such a violated property: different development teams for that space-probe used different units (the metric versus the English). Due to this discrepancy, the MCO probe was lost. Another example, which shows up regularly is a mismatch between expressing angles in degrees and radians. These properties can be checked during code review (actually, such a property should already be in the requirements review). However, a concise checking of application specific properties in the code can be quite tricky and time-consuming. In many applications (e.g., in guidance, navigation and control GN&C), an entire set of physical variables with different units (e.g., position, speed, forces, etc.) are packed into a single vector of floating point numbers, the so-called state vector. This representation enables the use of matrix operations in the algorithms, but obscures the physical units. For approaches to overcome that problem see e.g., [15].

Architectural View. Whereas the language specific view focuses on properties which must hold uniformly over the entire code, the architectural view classifies properties according to the structure or architecture of the code. Typically, this view distinguishes between properties for the environment (e.g., ”are we using the right version of the operating system?”), the interface between components (e.g., ”are all parameters passed by reference?”, or ”are all ports of the component attached to some other component?”), and the core, the actual code inside the building blocks.

Computer Science View. This view, which also could be called *formal methods‘ view*, structures the properties according to the topics of *safety properties, resource properties, liveness properties, security properties*, and *functional equivalence (properties)* and their techniques for analysis and verification. This view will be discussed in Section 3.3.

Tool-oriented View. This view classifies the properties according to the capabilities of the analysis tool which is used during source code analysis. Typical tools in this area are the compiler, lint (an old C checker), tools based upon static analysis (e.g., PolySpace [20]), and, in few cases classical verification tools (e.g., Model Checkers). We will discuss this view in Section 3.4.

3 Selected Views in more Detail

3.1 Quality Assurance View

Reviewing properties from a quality assurance perspective includes examining the software quality assurance results (including configuration management and the results of verification and validation) to ensure that the product was developed according to its specification. This review may also help detect whether or not QA and V&V activities were performed in accordance with their respective plans.

The QA view focuses on two important aspects: Software Requirements Review and Software Design Review. The following sections provide an overview of properties to consider when reviewing both the software product and plans for quality assurance activities.

3.1.1 Software Requirements Review

Requirements are the cornerstone of any software development endeavor because they describe the product being constructed. They must be compatible, complete, consistent, correct, feasible, modifiable, robust, traceable, understandable, and verifiable and testable. (These properties are presented in alphabetical order because each property is equally important.) An overview of these properties appears below and is based on review of these references: [12, 9, 27, 21, 14, 11, 3, 1].

Compatibility - defined as ensuring that interface requirements enable hardware and software or various software products to work together.

Completeness - necessary to make sure that all software requirements have been identified and documented including nominal functionality, performance, constraints, safety functioning; abnormal operating situations; temporal aspects of all functions; time-critical functions and their associated time criteria; any anticipated future changes; and normal environmental variables for all operating modes (normal, abnormal and disturbed).

Consistency - needed to alleviate any contradictions in the requirements. Consistency promotes use of standard terminology and supports ensuring the requirements are compatible with the operational environment (both hardware and software) and internal consistency exists between specified models, algorithms, and numerical techniques.

Correctness - essential to ensure that all aspects of the software are accurate including checking that: algorithms are supported by scientific or other applicable literature; evidence exists that vendors have correctly applied appropriate regulations; all expected types of errors and failure modes have been identified by the hazard analysis; functional requirements were analyzed to check that all abnormal situations are properly covered by system functions; adequacy of requirements for the man-machine interface; valid rationale for each requirement; suitable justification for the design/implementation constraints and that requirements conform to standards.

Feasibility - needed to make sure that the design, operation, and maintenance of software is practicable including ensuring that specified models, numerical techniques and algorithms are appropriate for the problem being solved and are based on generally accepted practice for the industry.

Modifiability - defined as making sure that requirements are organized with adequate structure and cross referencing to make modification possible while ensuring that each requirement is unique

Robustness - necessary to ensure that requirements exist for fault tolerance and graceful degradation

Traceability - important to make sure the product is being constructed according to the requirements. Also promotes flagging of safety functions or computer security functions for special review.

Understandability - imperative to ensure that every requirement has only one interpretation

Verifiability and Testability - vital to make sure that software can be checked to see whether requirements have been fulfilled

3.1.2 Software Design Review

Software Design is the focal point of software development because without proper design, development projects can fail to meet expectations and/or overrun budgets. Software design must be consistent with and meet the requirements. Numerical techniques and algorithms should be appropriate for the problem being solved. In a similar way to the requirements, the design must be complete, consistent, correct, feasible, modifiable, modular, predictable, robust, structured, traceable, and verifiable and testable. However, the properties are defined differently for the software design review (based on [27, 12, 14, 1]).

Completeness as it relates to software design means that the design fulfills all the requirements and there is enough data (logic diagrams, algorithms, storage allocation charts, etc.) to ensure design integrity. To achieve this one must determine that algorithms and equations are adequate and accurate; interfaces are described in sufficient detail; the operational environment is defined; the design takes into account all expected situations and conditions and gracefully handles unexpected or improper inputs and other anomalous conditions and finally that programming standards exist and are followed.

Consistency as it relates to software design signifies that the design lacks internal contradictions. To accomplish this the following should be used: standard terminology; a change control system to ensure the integrity of changes; compatible interfaces; models, algorithms, and numerical techniques that are mathematically compatible; consistent input and output formats; and identical units of measure throughout the same computation.

Correctness as it relates to software design indicates that the design logic is sound and the software will do what is intended in the operational environment. **Feasibility** as it relates to software design means that the specified design (models, algorithms, numerical techniques) is based on generally accepted practices for the target industry and can be implemented in operational environment with the available resources. **Modifiability** as it relates to software design signifies that software modules are organized such that changes in the requirements only require changes to a small number of modules; functions and data structures likely to change have standardized interfaces; data structure access, database access and I/O access from the application software occurs via the use of data access objects (globally accessible data is not used); and functionality is partitioned into objects to maximize the internal cohesion and to minimize coupling.

Modularity is defined as a design structured so that it comprises relatively small, hierarchically related objects or sets of objects, each performing a particular, unique function. **Predictability** means that computer resources are scheduled in a primarily deterministic and predictable manner and the design contains objects which provide the required response to identified error conditions. **Robustness** as it relates to software design means that all requirements related to fault tolerance and graceful degradation are addressed in the design. **Survivability** is closely related to robustness, with its main focus, however, on requirements to preserve essential functionalities even in the case of failures or outside attacks. Often, survivability also covers methods of software self-repair. **Structured-ness** means that the design uses a logical hierarchical control structure.

Traceability as it relates to software design means a mapping and complete coverage of all requirements and design constraints exists in the SRS; functions outside the scope of the SRS are identified; all functions can be uniquely referenced by the code; a revision history exists documenting all modifications to the design and the rationale for these changes; and safety and computer security functions been flagged. **Understandability** as it relates to software design indicates that the design is unambiguous and devoid of unnecessarily complexity. Finally, **Verifiability/Testability** as it relates to software design means that the design itself and each function in the design can be verified and tested.

3.2 Programming-language Specific View

Programming language specific view looks at properties from the language perspective to make sure code constructs are correctly implemented. For each programming language, such a list usually is very lengthy and detailed. Also, different authors also have a different view on certain issues, e.g., on the use of pointer arithmetic or dynamically allocated memory. In this paper, we describe two the check lists for C and C++. Lists for other languages (e.g., Java) contain similar properties.

3.2.1 A traditional Checklist

The following list of properties for this view reads much like the table of contents for a C/C++ textbook and was derived from various sources including [10, 2, 16, 25]. For better readability, we try to group these language-specific properties into a number of groups, like memory-related issues, control-flow issues, and such. Each group has between

2 and more than 10 properties. Although this grouping is definitely not exhaustive, we hope that it adds structure to this view.

Generic properties usually talk about (strategic and tactical) comments in the code. Strategic comments describe what a function or section of code is intended to do. Tactical comments explain the purpose of a single line of code [10, 25]. A large number of properties belong to the group, **Variables, Data Types, Memory**. Here, we find properties about variable declarations, constants, and variable initialization. **Object-Oriented Constructs** in C++ give raise to another group of properties (e.g., classes, inheritance, virtual functions, operator overloading, etc.).

Data Usage properties include sizing of data, dynamic memory allocation issues, null-termination of strings, pointers, and casting and type conversion. **Computation** properties concern the calculation of values (numerical), and update of variables. There are numerous properties which deal with **flow control** aspects of the code. A typical example is that all case statements should have a default-case. Also the evaluation of conditions is addressed here. For example, by always using inclusive lower limits and exclusive upper limits, off-by-one errors are usually eliminated (e.g., instead of $x \geq 23$ and $x \leq 42$, use $x \geq 23$ and $x < 43$).

Argument Passing is one of the areas where many software errors are made. Properties in this group talk about declaration of arguments (call by value, call by reference), consistency, the construction of temporary objects, and return values.

File-oriented properties concern requirements for reading and writing a file, security aspects as well as file name conventions. **Error conditions:** All probable error conditions and exceptions must be handled gracefully so the code provides for recovery from error conditions.

The property groups of **security** and **multiple-threads** are usually restricted to specific kinds of programs, but they contain a number of important (and very difficult) properties, like absence of deadlocks (correct use of mutex-locks). Finally, **miscellaneous** properties concern, among others, deprecated language features, dead code, and execution times.

3.2.2 A JPL Checklist

This classification of defects in software is due to P. Gluck, JPL (personal communication, 2002). Here, the individual properties are grouped into properties/issues of *Concurrency* (race conditions, deadlocks), *Misuse* (e.g., array-out-of bounds, mis-alignment of pointers), *Initialization* when no or an incorrect initial value is assigned, *Assignment* (e.g., wrong value, type mismatch), *Computation* (e.g., using a wrong equation), *Undefined Operations* (floating-point errors (e.g., tan($\pi/2$)), arithmetic errors (e.g., divide by zero)), *Omission* (case/switch statements without defaults), *Scoping* (global variables that should be local and vice versa; static variables that should be dynamic and vice versa), *Arg Mismatches* (e.g., missing arguments, too many arguments, wrong types, uninitialized arguments), and *Finiteness* (with underflow and overflow errors).

3.3 Computer Science View

.

While many mechanisms and tools for verifying program properties have been published, especially for distributed systems, relatively little attention has been paid to the properties themselves. The related work in this area is usually concerned with computer security [23]. An initial taxonomy of "useful" properties has been made in [24]. There, a first distinction is drawn between *functional* and *property-based* verification. Functional verification is necessary to show that a program correctly implements a high-level specification. Property-based verification, on the other hand, ensures that the programs have desirable features (e.g., absence of certain runtime errors), but does not show program correctness in the traditional sense. Rather, property-based verification has strong similarities to code review; in a sense, property verification can be seen as "code review in the extreme".

These properties can be grouped into four categories: safety, resource-limit, liveness, and security properties. *Safety properties* prevent the program from performing illegal or nonsensical operations. Within this category, we further subdivide into five different aspects of safety:

Memory safety properties assert that all memory accesses involving arrays and pointers are within their assigned bounds.

Type safety properties assert that a program is "well typed" according to a type system defined for the language. This type system may correspond to the standard type system for the language, or may enforce additional obligations, such as ensuring that all variables representing physical quantities have correct and compatible units and dimensions [15].

Numeric safety properties assert that programs will perform arithmetic correctly. Potential errors include: (1) using partial operators with arguments outside their defined domain (e.g., division by zero), (2) performing computations that yield results larger or smaller than are representable on the computer (overflow/underflow), and (3) performing floating point operations which cause an unacceptable loss of precision.

Exception handling properties ensure that all exceptions that can be thrown are actually handled.

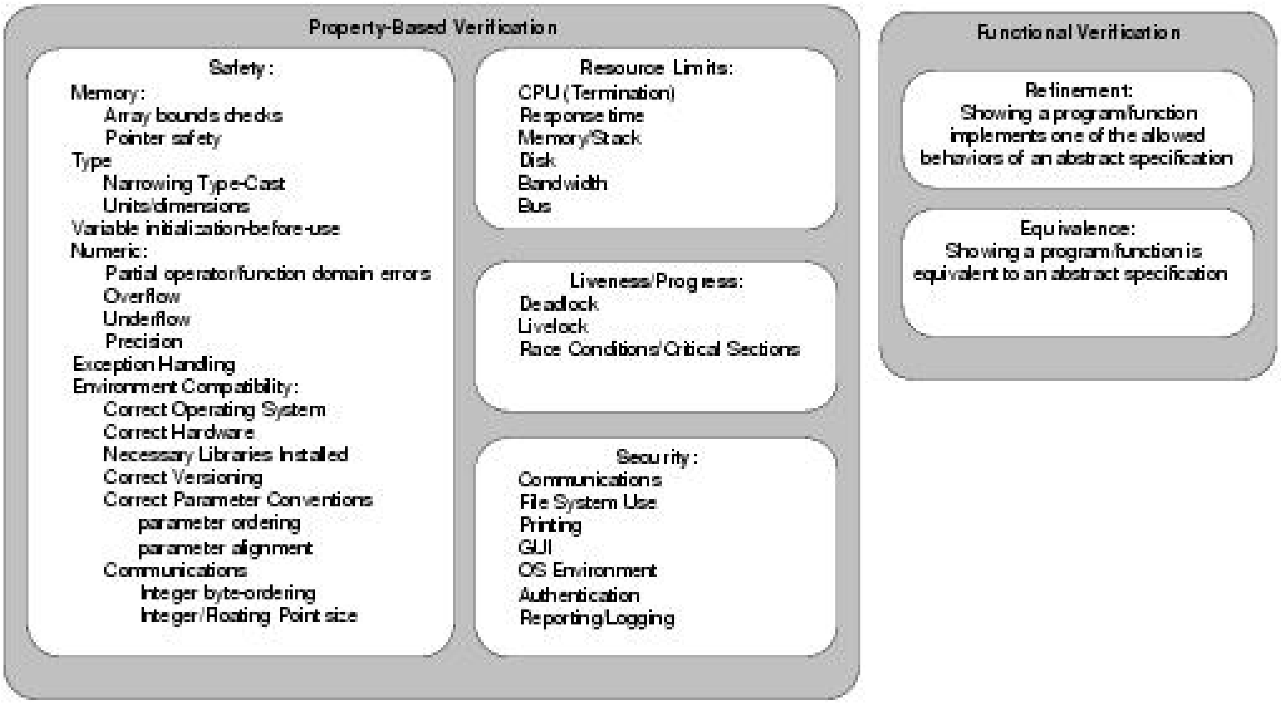

Figure 2. Property-based verification and functional verification [24].

Environment compatibility properties ensure that a program is compatible with its target environment. Compatibility constraints specify hardware, operating systems, and libraries necessary for safe execution. Parameter conventions define constraints on program communication and invocation.

Resource limit properties check that the required resources (e.g., stack size) for a computation are within some bound. *Liveness/progress properties* are used to show that the program will eventually perform some required activity, or will not be permanently blocked waiting for resources. *Security properties* prevent a program from accidental or malicious tampering with the environment. Security policies regulate access to system resources, and are often enforced by authentication procedures, which determine the identity of the program or user involved.

3.4 Tool-oriented View

This view on code review classifies properties according to the capabilities of an analysis tool. Here, the success criteria always is: Does the piece of code under review pass the analysis of a given tool *X*? It is obvious that the use of tools during code review can save a lot of time and effort; on the other hand, accuracy and coverage of the tool as well as trust in the tool (can we trust tool *X*?) plays an important role. The FAA requires tool qualification before a tool can be trusted to analyze or verify software. The objective of tool qualification is to ensure that the tool provides confidence at least equivalent to that of the process(es) eliminated, reduced or automated [6].

Traditionally, a compiler is a common tool to be used during code reviews. The most basic property of any code review obviously is: “Does the code compile without errors?”. A more restricted version of this property (e.g., used in some groups at JPL) is that during compilation no warning messages may show up. For this test, the compiler should be set to be most restrictive with respect to language features (e.g., `-pedantic` on the GNU C-compiler). Although a large number of language-specific properties are being checked by the compiler, an absence of warnings and errors does not say much, because, usually, it is not known, which properties are actually being checked.

A somewhat better coverage is provided by tools like `lint` and its successors. These tools are very close to compilers, however, they check for properties like naming or portability issues. A real breakthrough in support tool for code review are tools which are based upon static analysis (see [19] for an overview). For example, the tool PolySpace [20] combines a number of analysis algorithms (for unreachable code, array bounds checks, overflows and underflows, and others) with a graphical user interface. After analysis, the source code is displayed in a color-coded schema: code which has definitely passed all properties is shown in green, possible violations are shown in orange, errors in red. This tool is being used by the European Airbus industries and other transport agencies for safety-critical code. However, this tool has severe limitations for some kinds of code, e.g., on multi-threaded programs and code with matrices.

More powerful tools usually require in general additional information to be provided by the user in form of annotations (e.g., loop invariants) or a formal specification. ESC [8, 5] is a static analyzer which allows to enter annotations for tighter checking of properties. Proof-carrying code [17] also relies on checking of properties (e.g., memory safety). Using a theorem prover, these properties are formally proven for the given piece of code. In order to avoid tampering with the code, these proofs are bundled together with the code and are automatically checked when the program is loaded. Here, stronger checking of properties require additional annotations [4]. AutoBayes/CC [24] uses a similar mechanism, but all annotations are generated automatically during synthesis of the code. Other analysis approaches based on Model Checking or Rewriting usually require much more effort and cross the border between code review and verification.

However, in all approaches the question if a tool can be trusted, remains. In a very strict interpretation, a tool which is used for the development of safety-critical software (e.g., compiler, analysis tool, verification tool) needs to be certified to the same level of criticality as the software itself [6]. In many cases, however, "respectability" of a tool (has many users, has been on the market for several years, etc.) seems to be enough, in particular, when the tool's analysis is combined with manual review.

4 Questionnaire

The questionnaire in Table 1 is part of a document soliciting feedback to address the questions: Is code review trustworthy? How can a code review ensure certification of trustworthy code? To answer the first question, we refer to the Fraunhofer statistic [26] indicating that code reviews find about 60% of defects. Therefore, we can trust that a good code review will find defects.

However, what makes a good code review and will this review ensure certification of trustworthy code? In order to be effective a code review must cover salient software properties in an environment where participants have an incentive to provide candid feedback. While it is important to recognize that a trustworthy code review begins with management creating a suitable environment to promote team participation, discussion of such management techniques are outside the scope of this paper. Assuming a motivated team with incentive to succeed, we focused on providing a technique to ensure technical completeness of the code review by compiling a comprehensive list of properties across the software development life cycle and asking experts to prioritize this list based on lessons learned from prior projects. The entire questionnaire contains over 200 properties and was too long for inclusion in this paper. (To obtain a copy, please email `nelsonconsult@aol.com`) It was adopted at JPL for part of the verification and validation effort for the Mars Science Laboratory (MSL) rover project. Here research is underway to investigate automating relevant parts of this checklist in a workflow tool.

Because code reviews are generally conducted within specific time limits, this long list of properties must be prioritized to make sure that dangerous, difficult to find anomalies are uncovered as quickly as possible. Therefore, we asked survey participants to rank each property for importance and difficulty on a scale of 1 to 5 as defined below:

- Importance means how critical checking the property is to ensure human safety and/or mission success. 5 indicates high importance and 1 means less important.
- Difficulty is defined as how many resources it takes to check the property. For example: a complex algorithm or mathematical function may require special and rare expertise to ensure correctness or multi-threaded code may require a significant amount of time and labor. 5 indicates very difficult and 1 means less difficult.

Imp	Diff	Property
5	3	Divide by zero
5	3	Array index overrun
5	5	Mathematical functions sin, cos, tanh
5	1	Use of un-initialized variables or constants
3	3	No unused variables or constants
4	2	All variables explicitly declared
5	5	Proper synchronization in multi-threaded execution
4	4	Incorrect computation sequence
5	3	Loops are executed the correct number of times
5	3	Each loop terminates
3	2	All possible loop fall-throughs correct
4	3	Priority rules and brackets in arithmetic expression evaluation used as required to achieve desired results
5	5	Resource contention
5	2	Exception handling
5	5	The design implemented completely and correctly
4	2	No missing or extraneous functions.
5	1	Error messages and return codes used
5	1	Good code comments

Table 1. Sample Questionnaire

5 Findings and Feedback

The survey consisted of a questionnaire listing properties described in Table 1 and asking engineers and project managers to rank them based on importance and difficulty. In order to evaluate the survey information, various statistical methods were considered; however, due to the sparse or flat quantitative data and the richness of anecdotal information accompanying the questionnaire, a pure statistical model lacked the robustness to capture the true flavor of the responses. Therefore, we summarize the key issues described by NASA engineers at Dryden Flight Research Center, Ames Research Center and Jet Propulsion Laboratory:

- Code reviews catch about half of the defects. Additional V&V tools and techniques are required in concert with good code reviews to ensure trustworthy code. This is especially true for new types of adaptive and/or autonomy software.

- During code reviews, human reviewers have difficulty finding race conditions and other anomalies associated with synchronization of multi-threaded code and use of multi-threaded code in general. Autonomous and adaptive software provide additional challenges for human reviewers.

- Code standards and informative comments are necessary to ensure consistent flow of information among teams over the life of the project. This is largely due to high rates of turn over when economic conditions create high demand for technical experts.

- Reviewers rely upon demonstrations and prototypes running selected scenarios to increase confidence that software will work in accordance with expectations. There is an opportunity to leverage increased confidence from use of formal methods, but these approaches must be scalable and the number of false positives must be minimized.

6 Discussion: Trustworthy Code Review

Key facets of a trustworthy code review include trust and thoroughness. In order for code to be certified, reviewers must believe, based on the facts presented, that the development team has the expertise and experience to complete the task and that they have thoroughly investigated every software component critical to human safety or mission success. The comprehensive list of properties presented in this paper provides a technical guideline based on lessons learned at NASA and the aerospace industry for reviewers to use, in conjunction with consideration of the reputation of the development team, to accomplish a trustworthy code review.

Since the process of code review has to live in close vicinity with code development, there is a strong human aspect there: if code reviews are to be trusted, they should be used to provide positive feed-back to the developers and programmers and, "they [results of code inspection] should not under any circumstances be used for programmer's performance appraisal" [7].

References

[1] ANSI/ANS-10.4 Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry, 1987.

[2] An abbreviated C++ code inspection checklist. URL: `http://www2.ics.hawaii.edu/~johnson/FTR/Bib/Baldwin92.html`, 1992.

[3] N. Birrell and M. Ould. *A Practical Handbook for Software Development*. Cambridge University Press, 1985.

[4] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, 2000.

[5] D. Detlefs, K. R. Leino, G. Nelson, and J. Saxe. Extended static checking. Technical Report 159, SRC Research Report, 1998.

[6] DO-178B: Software considerations in airborne systems and equipment certification. URL: `http://www.rtca.org`, 1992.

[7] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15:182–211, 1976.

[8] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. Oliveira and P. Zave, editors, *Proc. Intl. Symp. Formal Methods Europe 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lect. Notes Comp. Sci.*, pages 500–517. Springer, 2001.

[9] Starts Group. The Starts Purchasers' Handbook, 1987.

[10] M. Henricson and E. Nyquist. Programming in C++: rules and recommendations. `http://www.doc.ic.ac.uk/lab/cplus/c++.rules/`, 1992.

[11] Reviewer guidance for computer controlled devices undergoing 510(k) review, 2999.

[12] IEEE STD. 1028-1997 IEEE Standard for Software Reviews and Audits, 1997.

[13] IEEE 610.12 IEEE Standard Glossary of Software Engineering Terminology, 1990.

[14] IEEEP STD 1059 IEEE Guide for Software Verification and Validation Plans (draft), 1991.

[15] M. Lowry, T. Pressburger, and G. Rosu. Certifying domain-specific policies. In M. S. Feather and M. Goedicke, editors, *Proc. 16th Intl. Conf. Automated Software Engineering*, pages 118–125, IEEE Comp. Soc. Press, 2001.

[16] B. Marick. A question catalog for code inspections, 1991.

[17] G. C. Necula and P. Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104. IEEE Computer Society Press, 1998.

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

[19] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1998.

[20] Polyspace technologies.
`http://www.polyspace.com`.

[21] F. J. Redmill, editor. *Dependability of Critical Computer Systems 2; The European Workshop on Industrial Computer Systems Technical Committee 7 (EWICS TC7)*. Elsevier, 1989.

[22] I. Rus, V. Basili, and B. Boehm. Empirical evaluation of techniques and methods used for achieving and assessing software high dependability. In *Proc. DSN Workshop on Dependability Benchmarking*, 2002.

[23] F. B. Schneider. Enforceable security policies. Computer Science Technical Report TR98-1644, Cornell University, Computer Science Department, September 1998.

[24] J. Schumann, B. Fischer, M. Whalen, and J. Whittle. Certification support for automatically generated programs. In *In Proceedings of the Thirty-Sixth Annual Hawaii International Conference on System Sciences (HICSS-36)*. IEEE, 2003.

[25] A. Shostack. Security code review guidelines, 2000.
`http://www.homeport.org/ adam/review.html`

[26] F. Shull, V. R. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. V. Zelkowitz. What we have learned about fighting defects. In *In Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada*, pages 249–258. IEEE, 2002.

[27] Standard for Software Engineering of Safety Critical Software, 1990.

[28] D. R. Wallace, W. W. Peng, and L. M. Ippolito. Nistir 4909: Software quality assurance: Documentation and reviews.
`http://hissa.ncsl.nist.gov/publications/-nistir4909/`, 1995.