Journal of Computer Science 5 (4):311-322, 2009
ISSN 1549-3636

An Approach to Modeling Software Safety in Safety-Critical Systems

Ben Swarup Medikonda and Seetha Ramaiah Panchumarthy
Department of Computer Science and Systems Engineering
Andhra University, Visakhapatnam-530 003, India

Abstract: Software for safety-critical systems has to deal with the hazards identified by safety analysis in order to make the system safe, risk-free and fail-safe. Software safety is a composite of many factors. **Problem statement:** Existing software quality models like McCall's and Boehm's and ISO 9126 were inadequate in addressing the software safety issues of real time safety-critical embedded systems. At present there does not exist any standard framework that comprehensively addresses the Factors, Criteria and Metrics (FCM) approach of the quality models in respect of software safety. **Approach:** We proposed a new model for software safety based on the McCall's software quality model that specifically identifies the criteria corresponding to software safety in safety critical applications. The criteria in the proposed software safety model pertains to system hazard analysis, completeness of requirements, identification of software-related safety-critical requirements, safety-constraints based design, run-time issues management and software safety-critical testing. **Results:** This model was applied to a prototype safety-critical software-based Railroad Crossing Control System (RCCS). The results showed that all critical operations were safe and risk-free, capable of handling contingency situations. **Conclusion:** Development of a safety-critical system based on our proposed software safety model significantly enhanced the safe operation of the overall system.

Key words: Software safety, safety-critical system, software quality

INTRODUCTION

The notion of software safety was first mentioned in the Mil-Std-1574A[1] which required analysis of software to identify and eliminate software errors relating to safety critical commands and control functions of space and missile systems. Since then, the role of software has becoming increasingly important and is being used in many critical applications, such as avionics, vehicle control systems, medical systems, manufacturing, power systems and sensor networks[2,3].

A safety-critical system is one that has the potential to cause accidents. Software is hazardous if it can cause a hazard i.e., cause other components to become hazardous or if it is used to control a hazard. Software is deemed safe if it is impossible or at least highly unlikely that the software could ever produce an output that would cause a catastrophic event for the system that the software controls. Examples of catastrophic events include loss of physical property, physical harm and loss-of-life. Software engineering of a safety-critical system requires a clear understanding of the software's role in and interactions with, the system[4,5]. According to Dunn[6], dependable, seemingly safe, concepts and structures fail in practice for three primary reasons:

- Their originators or users have an incomplete understanding of what makes a system "safe
- fail to consider the larger system into which the implemented concept is to be embedded, or
- ignore single points of failure that will make the safe concept unsafe when put into practice

Application areas for safety-critical systems include the following-Military, e.g., weapon delivery systems and space programs. Industry, e.g., manufacturing control where toxic substances are involved and robots. Transportation, e.g., fly-by-wire systems on board aircraft, air traffic control, interlocking systems for trains, automatic train control and computer systems in cars. Communication, e.g., ambulance dispatch systems and the emergency call part of a telephone system. Medicine, e.g., radiation therapy machines, medical monitoring and medical robots. Nuclear power plant control. As is apparent from the above example areas, safety-critical systems are often real-time control systems. These systems

Corresponding Author: Ben Swarup Medikonda, Department of Computer Science and Systems Engineering, Andhra University, Visakhapatnam-530 003, India

require the utmost care in their specification, design, implementation, operation and maintenance, as they could lead to injuries or loss of lives and in-turn result in financial loss[7,8]. This is the type of system that will be considered in this study. Here are some concepts and terms relating to safety found in the literature relating to safety critical systems.

Safety-related terms:
Failure: An event where a system or subsystem component does not exhibit the expected external behavior. The expected system behavior and the environmental conditions under which it must be exhibited should be documented in the requirements specification.

Error: An incorrect internal system state.

Fault: A fault is anything that might cause an error. A fault may be a physical defect in hardware, a flaw in software or incorrect operator input. According to Nissanke[9], a fault may or may not cause an error and an error may or may not cause a failure. Faults can have their origin within the system boundaries (internal faults) or from without, namely, in the environment (external faults). In particular, an internal fault is said to be active when it produces an error and dormant (or latent) when it does not. A dormant fault becomes an active fault when activated by either its process or the environment. Fault latency is defined as either the length of time between the occurrence of a fault and the appearance of the corresponding error, or the length of time between the occurrence of a fault and its removal[10].

Hazard: A system state that might, under certain environmental conditions, lead to a mishap[11]. Hence, a hazard is a potentially dangerous situation.

Safety constraint: A hazard characterizes a system state that for safety reasons should not occur. If this is negated and some safety margins are included we get a safety constraint, i.e., a description of a property that the system should possess in order to be safe.

Safety-critical: Those software operations that, if not performed, performed out-of sequence, or performed incorrectly could result in improper control functions (or lack of control functions required for proper system operation) that could directly or indirectly cause or allow a hazardous condition to exist[12]. A real-time system is safety critical when its incorrect behavior can directly or indirectly lead to a state hazardous to human life[13]. Decisions which shape the software architecture for safety-critical, real-time systems are driven in part by three qualities; availability, reliability and robustness[13,14].

MATERIALS AND METHODS

Software quality models: There have been two notable models of software quality attributes viz. McCall's and Boehm's. There are others but these two illustrate the general purpose quality models. Both McCall and Boehm have described quality using a decompositional approach[15,16]. McCall's model of software quality (The GE Model, 1977) incorporates eleven criteria encompassing product operation, product revision and product transition. Boehm's model (1978) is based on a wider range of characteristics and incorporates nineteen criteria[17]. The criteria in these models are not independent; they interact with each other and often cause conflict, especially when software providers try to incorporate them into the software development process. ISO 9126 standard incorporates six quality goals, each goal having a large number of attributes[18].

McCall software quality model: This framework is useful for its integrated approach to quality. In this framework, software quality attributes are classified into a hierarchy of three levels as shown in Fig. 1. At the top level are the so-called "quality factors" from a customer or user perspective: correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability and interoperability. At the second level, are the "quality criteria," which represent technical concepts. At the third level, are the "quality metrics," which measure the attributes of software products.

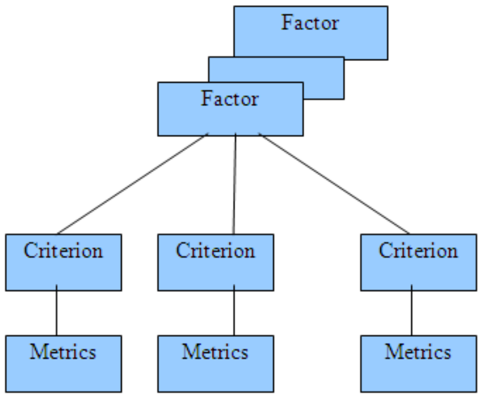

Fig. 1: McCall's software quality model

The last two levels are from engineering perspectives. McCall suggests these application steps:

- Deduce quality factors based on the characteristics of the system
- Trade-off and prioritize the quality factors based on the needs of the customers/users
- Deduce related quality criteria and metrics using the framework; and
- Base specification, design, coding and testing on the deduced factors, criteria and metrics

The original eleven quality factors in McCall's Software Quality Model are: Usability, Integrity, Efficiency, Correctness, Reliability, Maintainability, Testability, Flexibility, Reusability, Portability, Interoperability.

The modified McCall's quality framework applied to software safety: Raghu Singh has proposed a modified framework to address software safety[19]. The four factors relating to software safety in his model which are part of the original McCall model are: Correctness, efficiency, reliability, testability. To these four quality factors, a new factor-responsiveness was introduced to account for the real time performance. For each factor the corresponding criteria (attributes from the developer point of view) are derived as shown in Table 1. It is argued that determination and application of specification, design, coding and testing methods in a project should be based on the metrics derived from the criteria in order to "ensure" software safety.

All these quality models-McCall's, Boehm's and ISO 9126 and the modified model by Raghu Singh do not directly address the specific issues of software safety but emphasize the general quality attributes. They have the following limitations. First, many of the factors suggested by these models are not directly related to the specific issue of hazards contributed by the malfunction modes of software. Second, they assume that the concepts of reliability and safety are equivalent whereas a system can be reliable and still be not safe. Making a system more reliable is not sufficient if it has unsafe functions. This translates to having a system that reliably functions to cause unsafe conditions. Finally**,** these models seem to focus on non-safety critical systems where the emphasis is more on efficiency and other quality attributes and less on the safety issues of hazards and mishaps that can endanger human life and property. To overcome these limitations, a new model is proposed that captures the major issues specifically related to software safety.

Table 1: Factors and Criteria

Factors	Criteria
Correctness	Completeness, consistency, traceability
Efficiency	Execution efficiency storage efficiency
Reliability	Accuracy, consistency fault tolerance, simplicity
Responsiveness	Execution adequacy throughput adequacy
Testability	Instrumentation, modularity, self-descriptiveness, test completeness

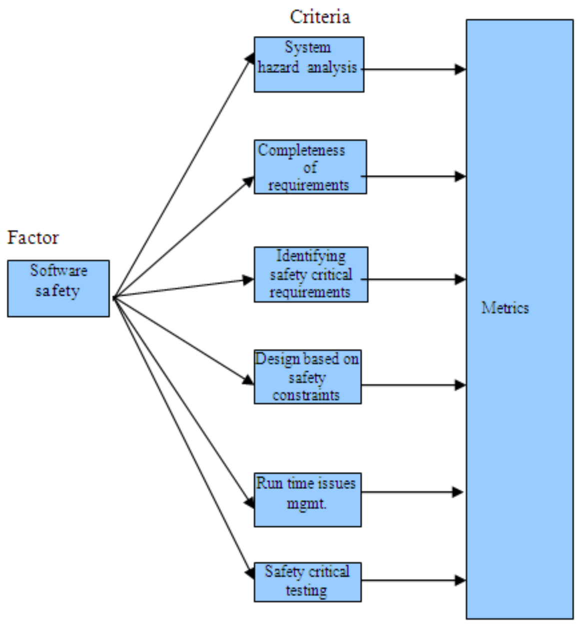
Criteria
Factor
Software safety
System hazard analysis
Completeness of requirements
Identifying safety critical requirements
Design based on safety constraints
Run time issues mgmt.
Safety critical testing
Metrics

Fig. 2: Software safety model

Proposed model for software safety: The proposed model for software safety based on the factor, criteria and metric approach is shown in Fig. 2.

The quality factor software safety may be decomposed into six quality criteria as listed below:

- System hazard analysis
- Completeness of requirements
- Identification of safety critical requirements
- Design based on safety constraints
- Run-time issues management
- Safety critical testing

Each criteria may be further decomposed into a set of lower level quality metrics, which are directly measurable. Each proposed criteria of software safety is briefly explained as follows:

System hazard analysis: While developing a framework for software safety is the focus of this study it is important to note that no software works in isolation. The entire system must be designed to be safe. The system contains the software, hardware, the users and the environment. All must be given consideration when developing software. All parts of the system must be safe. Functional and operational safety starts at the system level. Safety cannot be assured if efforts are focused only on software. The software can be totally free of 'bugs' and employ numerous safety features, yet the equipment can be unsafe because of how the software and all the other parts interact in the system. Hazards at the system level include: hardware hazards, software hazards, procedural hazards, human factors, environmental hazards and interface hazards[20].

Preliminary system safety analyses (e.g., Preliminary Hazard Analysis (PHA)), conducted during the system requirements phase when the role of software is being defined, begin to identify the hazards associated with a particular design concept and/or operation. These preliminary analyses and subsequent system and software safety analyses identify when software is a potential cause of a hazard or will be used to support the control of a hazard. This software shall be classified as safety-critical and shall be subjected to software safety analysis. The system safety analyses are the first place to identify software safety requirements necessary to support the development of the software requirements specification. These requirements shall be provided to the developer for inclusion into the software requirements document. Some examples of software safety requirements include limits (e.g., redlines, boundary values), sequence of events, timing constraints, interrelationship of limits, voting logic, hazardous hardware failure recognition, failure tolerance, caution and warning interfaces, hazardous commands, etc.,

The system safety analyses continue throughout the project life cycle. The software safety analysis process needs to continue to review the results of the systems analyses to assure that changes and findings at the system level are incorporated into the software as necessary. In addition, the software safety analyses provide input to the system safety analyses. The software safety analyses are a special portion of the overall system safety analyses and are not conducted in isolation.

The basis of sound design for a safety-related system is the identification, through systematic analysis, of the hazards which the system might encounter in operation. A number of techniques are well established for electrical and electronic systems but there has been much debate as to how relevant these techniques are when applied to software. The objectives for the software hazard analysis, as stated by the standards/guidelines include:

- Identifying critical system modules and program sections, i.e., those with most safety relevance
- Verifying that software required to handle the failure modes identified by systems/subsystems hazard analysis does so effectively
- Allowing more rigorous methods and controls to be selected and applied to areas of software which are most critical to the safety of the system
- Identifying and evaluating safety hazards associated with the software, with the aim of either eliminating them or assisting in the reduction of associated risks
- Identifying failure modes that can lead to an unsafe state and making recommendation for changes
- Determining the sequence of inputs which could lead to the software causing an unsafe state and making recommendations for changes

Approaches suggested include Failure Modes and Effects Analysis (FMEA), Fault Tree Analysis (FTA) and Hazard and Operability (HAZOP) technique.

Completeness of requirements: Completeness can be defined as the property that requirements are sufficient to distinguish the desired behavior of the program from that of any other undesired program that might be designed[21]. It should not be surprising then that most errors found in operational software can be traced to requirements flaws, particularly incompleteness. Completeness is a quality often associated with requirements but rarely defined. In addition, nearly all the serious accidents in which software has been involved in the past 20 years can be traced to requirements flaws, not coding errors. The software may reflect incomplete or wrong assumptions about the operation of the system components being controlled by the software or about the operation of the computer itself. The problems may also stem from unhandled controlled-system states and environmental conditions. Thus simply trying to get the software "correct" in terms of accurately implementing the requirements will not make it safer in most cases. Basically the problems stem from the software doing what the software engineer thought it should do when that is not what the original design engineer wanted. Integrated product teams and other project management schemes to help with this communication are being used, but the problem has not been solved[19].

Donald Firesmith[22] proposes seven different ways in which the phrase ‘requirements completeness’ could be interpreted. These include the completeness of:

- Requirements analysis models
- Individual requirements
- Metadata describing individual requirements

- Requirements repositories
- The set of requirements documents
- Individual requirements specification documents
- A requirements baseline

An individual requirement is complete if it contains all necessary information to avoid ambiguity and needs no amplification to enable proper implementation and verification. To avoid ambiguity, a requirement must express the entire need and state all conditions and constraints under which it applies[23]. Different kinds of requirements are specified differently. Therefore the following different kinds of requirements may be incomplete because different component parts of them are missing:

- Functional Requirements
- Data Requirements
- Interface Requirements
- Quality Requirements
- Constraints

Types of safety-related requirements: When engineering safety-related requirements, stakeholders must realize that these requirements come in four distinct types, which need to be analyzed and specified differently[24]. They are (i) Safety requirements (ii) safety-significant requirements (iii) Safety system requirements (iv) Safety constraints. They are explained as follows:

First of all, there are pure safety requirements, which are a kind of quality requirement that views safety as a quality factor within a quality model. As such, safety requirements are typically of the form of a quality criterion (a system-specific statement about the existence of a sub factor of safety) combined with a minimum or maximum required threshold along some quality measure. They directly specify how safe the system must be. Second are safety-significant requirements, which are normal functional, data, interface and non-safety quality requirements that are relevant to the achievement of the safety requirements. In other words, safety-significant requirements can lead to hazards and accidents when not implemented correctly. When most people think of safety-critical systems, they are thinking of systems, the required functionality of which makes them subject to serious accidents. Third are safety system requirements, which are the requirements for safety systems or safety components of safety-related systems. A canonical example of which would be requirements for the emergency core cooling system of a nuclear power plant. Requirements for an aircraft's fire detection and suppression system would also be safety system requirements. Finally, safety constraints are architecture or design constraints mandating the use of specific safety mechanism or safeguards. Many industries including petrochemicals, nuclear power and automated people movers have industry safety standards requiring specific safeguards.

Identification of software-related safety-critical requirements: A safety critical software requirement may be understood as a software requirement identified as essential to the safe system operation or use[25]. Specifically, a safety critical software requirement performs one or more of the following functions:

- Controls or directly influences the functioning of safety critical hardware
- Controls or directly influences hazardous systems
- Monitors the state of the system for purposes of ensuring its safety
- Senses hazards and/or displays information, concerning the protection of the system
- Handles or responds to fault detection priorities
- Disables or enables interrupt processing software
- Generates output that displays the status of safety critical hardware
- Computes safety critical data

The above listed functions are based on the functions presented in STANAG 4404[26]. Safety critical computer system functions are essentially those software features that are used to monitor, control, or provide data for the safety-critical functions. Once the safety-critical computer system functions have been identified, the safety engineer should perform analyses to assess the risks associated with each identified safety-critical requirement. In software-intensive systems, mishaps often occur because of a combination of factors, including component failure and faults, human error, environmental conditions, procedural deficiencies, design inadequacies and software and computing system errors. In such systems software often cannot be divorced from the system where it resides. Software and computing system safety analyses should consider safety aspects of the following items:

- Computer system hardware, which includes physical devices that assist in the transfer of data and perform logic operations. Examples include Central Processing Units (CPU), busses, display screens, memory cards and peripherals

- Computer system firmware, which is resident software that controls the CPU’s basic functioning
- Computer system software, including operating system software and applications programs

In addition, because software safety is a systems issue, software and computing systems must be considered with respect to other aspects of the system, such as the following:

- Physical entities whose function and operation are being monitored or controlled, often called the application
- Sensors (thermocouples, pressure transducers)
- Effectors that take an instruction from the computing system and impart an action on the system (valves, actuators)
- Data communication to other computers
- Humans who will interact with the system

Safety is enhanced through the use of layers of protection that include both software- and hardware-specific safety measures. The output from the software-specific hazard analysis process includes design-level safety requirements based on safety measures developed to mitigate hazards. These design-level requirements could include specific hardware mitigation measures (such as redundant functionality using hardware) or coding requirements that must be implemented. Design-level requirements are statements that can be translated into code without interpretation, or specific mitigations that must be implemented.

Design based on safety-constraints: The first step in the safety-constraint centered design approach is the specification of safety constraints[27]. In hardware systems, redundancy and diversity are the most common ways to reduce hazards. Hardware detection and control includes mechanisms such as fail-safe designs, self-tests, exception handling, warnings to operators or users and reconfigurations. For software intensive safety-critical systems, software design must enforce safety constraints. Reviewers should be able to trace from requirements to code and vice versa. In addition to the specific safety constraints developed for the system being designed, the design should incorporate basic safety design principles. Safety, like any quality, must be built into the system design. Software represents or *is* the system design[13]. The most effective way to ensure that a system will operate safely is to build safety in from the start, which means that system operation must not lead to a violation of the constraints on safe operation.

System accidents result from interactions among components that lead to a violation of these constraints. In other words, from a lack of appropriate enforcement of constraints on the interactions. Because software often acts as a controller in complex systems, it embodies or enforces the constraints by controlling the components and their interactions. Software, then, can contribute to an accident by not enforcing the appropriate constraints on behavior or by commanding behavior that violates the constraints.

The requirement for software to be safe is not that it never "fails" but that it does not cause or contribute to a violation of any of the system constraints on safe behavior. This observation leads to the suggested approach to handling software in safety-critical systems, i.e., first identify the constraints on safe system behavior and then design the software to enforce those constraints.

The software-specific analysis should provide specific mitigation approaches for each potential hazard identified. The recommended order of precedence for eliminating or reducing risk in the use of software and computing systems is the same as that for hardware, as follows:

- Design for minimum risk
- Incorporate safety devices
- Provide warning devices
- Develop and implement procedures and training

Mitigation measures can include, but are not limited to, approaches such as the following[28]:

- Software fault detection (for example, built-in tests, incremental auditing)
- Software fault isolation (for example, isolating safety-critical functions from non-safety-critical functions)
- Software fault tolerance (for example, recovery blocks that use multiple software versions of progressively more reliable construction should faults occur)
- Hardware and software fault recovery (for example, incremental reboots, exception handling)

After the designers have applied measures to mitigate mishap risk to a basic system, they must determine if the modified system design meets an acceptable level of mishap risk. They can use three analytical techniques to make this determination. In Failure Modes And Effects Analysis (FMEA), the designer or analyst looks at each component in the system, considers how that component can fail, then determines the effects each failure would have on the system[29,30]. This analysis seeks first to verify that there is no mishap-producing single point of failure in the system because such a potential point of failure would nullify the benefits of applying mitigation measures elsewhere in the system.

Fault Tree Analysis (FTA) reverses this process by starting with an identified mishap and working downward to identify all the components that can cause a mishap and all the safety devices that can mitigate it[32,33]. This downward decomposition process builds a graphical structure called a fault tree. In contrast to FMEA and FTA, which are both qualitative methods, Risk Analysis (RA) is a quantitative measure that yields numerical probabilities of mishap[29,30]. To perform RA, the analyst must determine the component failure probabilities for the hardware, software and operator components in the fault tree[29-31]. In accordance with standards such as Mil-Std-882D[33] and IEC 61508[34] designers usually estimate failure probabilities on a per-hour basis.

If the system consists of redundant components, designers calculate its unreliability-the probability that it will not operate over the span of one hour. Next, they determine mitigation failure probabilities for the fault tree’s hardware, software and operator safety devices. If a mitigation device includes redundant components, designers determine its unavailability-the probability that it will not mitigate if required. The designers assign these component- and mitigation- failure probabilities to elements in the fault tree, then propagate them upward to yield a figure for mishap risk. If this results in an unacceptable figure, they must implement additional mitigation measures. As a side benefit, the fault tree shows where to add these measures in the system. If, on the other hand, the risk calculation yields an acceptable result, the design is ready for additional validation steps[28] such as in-depth risk assessment, testing and field trials to assure that the system, when implemented, will be safe. Although it may seem obvious, a developer’s concerns about a safety-critical system’s continuing safety do not end with design and implementation. Indeed, a vigorous system safety program must be in place throughout the system’s operational life to ensure that mishap risk is maintained at or below the level achieved in the original design[33,34].

Run time issues management: There is always the risk that an a priori verified program behaves slightly differently-and faultily-at runtime. This may simply be the result of compiler bugs, or it may be due to mismatches between the expected and actual behavior of the execution environment, say with respect to timing issues or memory behavior.

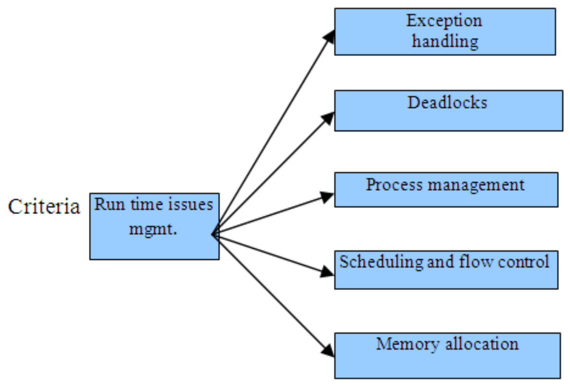

Fig. 3: A decomposition of run time issues criteria

An operating-system kernel and application programming interface often perform the most important role in a safety-critical system. Exception handling, deadlocks, process and stack management, scheduling and flow control and memory protection all have repercussions on the safety function and can be key elements of meeting safety-integrity requirements. Figure 3 shows the decomposition of the run time issues criteria into five sub-criteria or lower-level criteria which provide a basis for measurements.

Traditional testing techniques such as unit testing are ad hoc and informal. It is only a partial proof of correctness in that it does not guarantee that the system will operate as expected under untested inputs. In terms of its ability to guarantee software correctness, runtime verification is stronger than testing. Testing can only guarantee the correctness of a limited set of inputs at implementation time. As a result, undiscovered faults may result in failures at runtime and even allowing the system to propagate corrupted output because the failure was not detected. By always monitoring the software for correctness, such failures can be caught when they happen, for any input which causes them to occur.

Runtime verification is a lightweight verification technique that complements traditional techniques such as model checking and testing[35]. It checks whether the current execution of a system under scrutiny satisfies or violates a given correctness property. One of the main distinguishing features of runtime verification is that it is performed-as the name suggests-at runtime. This opens up the possibility not only to detect incorrect behavior of a software system but to react whenever incorrect behavior is encountered.

Checking whether an execution meets a correctness property is typically performed using a monitor. In its simplest form, a monitor decides whether the current execution satisfies a given correctness property by outputting either yes/true or no/false. More detailed assessments, like the probability with which a given correctness property is satisfied, can also be given. In runtime verification, monitors are typically generated automatically from some high-level specification. As runtime verification has its roots in model checking, often some variant of linear temporal logic is employed. Besides checking safety properties directly using the monitors generated from them, runtime verification can also be used with partially verified systems. Such partial correctness proofs often depend on assumptions made about the behavior of the environment. These can be easily checked using runtime verification techniques. Runtime verification itself deals (only) with detecting whether correctness properties are violated (or satisfied). Thus, if a violation is observed, it typically does not influence or change the program's execution, say by trying to repair the observed violation.

Safety critical testing: Testing of safety-critical systems follows two important strategies which are systematic rigorous testing and static analysis. While there is no substitute for rigorous testing at many levels: Unit, regression, functionality and integration testing, testing effectiveness depends on the quality of the test cases used. The best test suites are those that have good code coverage. Statement coverage and condition coverage are the most commonly used metrics. Full condition coverage is considered essential for safety-critical code, such as flight control software. Achieving full coverage can be exceedingly time-consuming and expensive. There are different kinds of coverage and the risk the code carries dictates which kind of coverage is required. In the DO-178B Standard for aviation, the riskiest code requires 100% Modified Condition/Decision Coverage (MCDC). The next two most risky classes require 100% decision coverage and statement coverage, respectively. The least risky code, such as the in-flight entertainment system, has no coverage requirements at all. Also, as all programmers know, just because a statement is executed in a successful test case does not mean it will always execute correctly. It may fail under an unusual combination of circumstances that the test cases did not explore.

Safety critical software functions provide the source of requirements to be tested. Testing shall be performed to verify correct incorporation of software safety requirements. Testing must show that hazards have been eliminated or controlled to an acceptable level of risk. Additional hazardous states identified during testing shall undergo complete analysis prior to software delivery or use. Software safety testing of Safety-Critical Computer Software Components (SCCSC) shall be included in the integration and acceptance tests. Acceptance testing shall verify correct operation of the SCCSCs in conjunction with system hardware and operators[36]. It shall verify correct operation during stress conditions and in the presence of system faults. It is important to tailor the safety-critical testing effort to emphasize the parts of the software that need additional analysis and testing. The greatest effort must be placed on the hazards posing the highest risk. We consider it adequate to divide the software into two risk groups for test purposes. Group one includes hazards that are catastrophic or critical. Group two includes hazards that are marginal or negligible as per the definitions in MIL-STD-882C. Software in the first group deserves special safety analysis and testing since the hazards pose a higher level of risk. The normal level of software analysis and testing performed for operational software is adequate for group two.

While traditional dynamic testing plays a fundamental role in producing high-quality software it is only as good as the test cases. To be effective, a great deal of effort must go into writing or generating good test cases and doing so can be very expensive. Recently, a new breed of static analysis tools has emerged that can find flaws without writing any test cases. These tools, which are also referred to as static testing tools, can find bugs that are difficult or impossible to find using standard testing methodologies[37]. They can locate serious flaws such as buffer overruns, null pointer dereferences, resource leaks and race conditions. Because they operate by analyzing the source code itself in detail, they can also highlight inconsistencies or contradictions in the code such as unreachable code, useless assignments and redundant conditions.

The following illustrates some of the most important classes that static tools can detect. The first class is the most serious-bugs that either cause the program to terminate abnormally or result in highly unpredictable behavior. These include buffer overrun and under run, null pointer dereference, division by zero and use of uninitialized variables. Memory allocation errors are those that result from the misuse of malloc or new functions. These can be tricky to debug because the erroneous behavior may only show up long after the event that caused the error. Such errors include double free, use after free and memory leak. Concurrency bugs may be caused by misuse of the threads library. Double locks or unlocks, race conditions and futile attempts to lock are among the checks that are available.

A second class of check is for inconsistencies or redundancies. These are not bugs per se, but are often indicators that a programmer misunderstood something.

This class includes redundant conditions, useless assignments and checking whether a pointer is null after it has already been dereferenced. Holtzmann[38], in his list of ten rules for writing safety-critical code, explicitly specifies that advanced static-analysis tools should be used proactively all through the safety-critical development process.

Application of safety model to Railroad Crossing Control System (RCCS): Crossing gates on a full-size railroads are controlled by a complex control system that causes the gates to be lowered to prevent access to the crossing shortly before a train arrives and to be raised to allow access to resume after the train has departed. This requires the detection of approaching trains or the manual actuation of the crossing gates by an operator. RCCS is a prototype safety-critical railroad crossing control system of limited complexity. Figure 4 shows the laboratory prototype of RCCS consisting of several components listed below.

Components of RCCS: RCCS consists of the following main components: Train, Railway track, Sensors, Gates, Controller with a digital I/O card, Signals and a muscle-wire operated track-change lever. A brief description of each component is given below.

Train: The train is powered by a power supply relay. When the power is initially switched on, the train begins movement along the track when the metallic wheels of the train receive power. The train comes to a halt at the position where the power to the tracks is switched off. When a train approaches the gate crossing region, the train is detected by the sensor positioned near the gate crossing area. The sensor sends this information to the controller component. When a train completely passes the crossing section, it is detected by the sensor which is positioned after the gate crossing area. This information is sent to the controller.

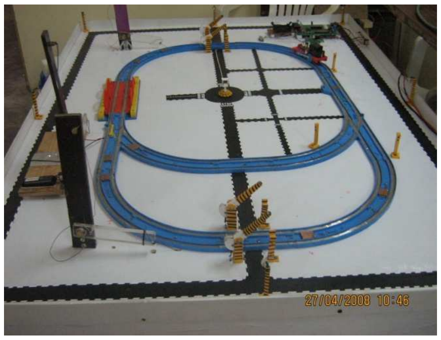

Fig. 4: Prototype of RCCS

Sensors: These are used to detect the location of the train on the tracks. Altogether RCCS employs nine sensors. Two pair of sensors detect the train position before and after the gates. A set of three sensors relate to track change where the track splits into two directions. A pair of sensors give the train position with reference to the platform, which is the starting point of the train movement. Information from each of the sensors is passed to controller.

Controller: The controller synchronizes the train activities with the gate. When the controller receives a message from sensor1, it sends a command to lower the gates. When it receives a message from sensor2, it sends a command to raise the gates. An IBM compatible PC is used as a controller for RCCS. RCCS software that controls the overall operation of the system is stored in the memory of the controller PC. A user interface is provided to operate the selections of the controller PC. A 48-line digital I/O (DIO) add-on card is plugged into an available slot in the controller PC for monitoring and controlling sensors and gate actuators. The DIO card receives the inputs from each of the nine sensors of RCCS. The eight output signals sent from DIO card control the following: the power supply to the train track, power supply to the two gate assemblies, power supply to muscle-wire based mechanism to change the track lever and four signal lights.

Gates: RCCS has two sets of gates on either side of the track layout. The gate receives signals from the controller component. When it receives lower, it moves down. When the gate receives raise, it moves up. The gates are operated by means of a muscle wire based mechanism. Muscle wire (Nitinol) is a nickel titanium alloy which contracts when current flows through it, for achieving motorless motion for gate movement and track change.

Signals: Railroad signals are provided to indicate to train operators whether the track is clear or occupied, or if certain precautionary measures should be taken while using the track, such as maintaining a reduced speed. RCCS contains three train signals, erected beside the track. One signal is at the platform to signal a halt at the platform. The other two signals are placed just before the point of convergence of the inner track and outer track, which lead to the platform. A signal head consists of one or more signal faces that can include solid red and green lights.

RESULTS AND DISCUSSION

Normal operation of RCCS: When RCCS is first switched on, the controller does a preliminary check of the normal working status of all the subsystems involved-the driver circuitry, the sensors, the gate assemblies and the train signals. If all the components are found to be in normal working condition, it executes the code related to normal operation. Figure 5 shows the partial block diagram of RCCS corresponding to the rail-road intersection. If the train passes Sensor1 positioned prior to gate, a signal is sent to the controller indicating the approaching train. The controller then sends a signal to the gates assembly, causing the gate arms on either side of the road to close. When the train finally has passed Sensor2, which is positioned just beyond the gate crossing section, a corresponding signal is sent to the controller, which in turn triggers both the gate arms to open simultaneously. If RCCS detects any abnormal situation or state during its normal mode of operation, perhaps due to an unexpected lightning strike or rainstorm that disrupts the circuitry of the gate assemblies, it executes the code relating to emergency situation causing the signal erected near the gates, to flash a red light continuously. This is an indicator to the public that the gate assembly is not in working condition and that they need to take necessary precaution in crossing the intersection.

All the six criteria of the model were applied to RCCS. First, the system-level hazard analysis was done to identify possible hazardous failure conditions at the system level. The potential hazards identified are: Failure of Controller, Failure of Sensors, Failure of Driver Circuitry, Failure of Gate 1 and Gate 2, Failure of Train Signal, Failure of muscle-wire operated Track Change Lever in changing from outer to inner track. Next, the identified hazards were classified according to their severity. A hazard belongs to one of four levels-catastrophic, critical, marginal and negligible.

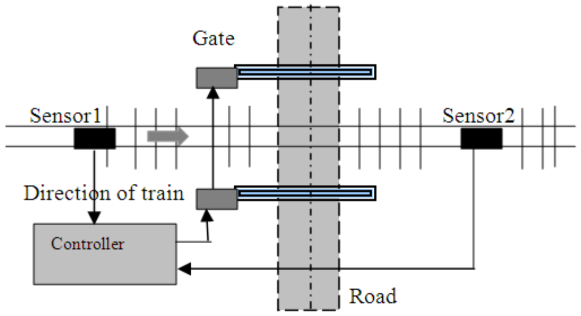

Fig. 5: RCCS partial block diagram showing railroad crossing intersection

For example, the failure of the controller may lead to both gates being permanently open, causing accidents, can be considered a catastrophic or severe hazard. Failure of the sensor that detects that the train has passed the gate crossing section, with the effect of the gates being permanently closed will not cause an accident but will violate the utility property of the gates, until the problem is rectified. Failure of the sensor that detects the approaching train can cause an accident as the controller will not close the gates keeping them open, which can lead to accidents as the road users are unaware of the approaching train. This is a catastrophic or severe hazard.

Second, completeness of requirements criteria was applied to check any missing or ambiguous specifications. This was done by peer review and manual checking rather than applying any formal methods. Third, all the safety-critical and non-safety critical requirements were identified. All requirements that directly or indirectly lead to incorrect operation of the gates are considered safety-critical. Fourth, a design that enforced the safety constraints was chosen for RCCS. The objective of the design was to eliminate or mitigate the hazards identified in the preliminary system-level hazard analysis. Another objective was to avoid the possibility of single point failure. This was achieved by using a additional redundant controller that takes over control of the system should the main controller fail unexpectedly. Implementation was done in Cyclone programming language which is a dialect of C language which includes several safety features not found in C. Fifth, run-time performance was monitored for problems relating to exceptions, deadlocks, memory related issues like buffer overruns. Lastly, safety critical testing of RCCS was done by separating the code into two risk groups. Group one includes hazards that are catastrophic or critical. Group two includes hazards that are marginal or negligible. More testing effort was spent on those code sections dealing with hazards related to group one. The preliminary results in applying the safety model in developing the safety-critical RCCS clearly demonstrate that the system is safe, risk-free and fail-safe when compared to a development methodology that does not take hazards and associated risks into consideration.

CONCLUSION

This study discussed the criteria relevant to software safety. A new model for software safety is proposed. A set of quality criteria that form the basis of software safety is presented. The proposed model is applied to a laboratory prototype of a software-based

Railroad Crossing Control System (RCCS) that includes safety-critical operations and observed satisfactory results. Using the experimental results of the proposed model with railroad crossing control system, work can be extended to address issues of development cost and development time in implementing this model to achieve software safety metrics. Rigorous work is needed to meet the complete requirements of software safety aspects that leads to standardization of model with safety metrics.

REFERENCES

1. MIL-STD-1574A (USAF)**,** 1979. System safety program for space and missile systems. http://store.mil-standards.com/index.asp?PageAction=VIEWPROD&ProdID=142
2. Wang, D., F.B. Bastani and I.L. Yen, 2005. Automated aspect-oriented decomposition of process-control systems for ultra-high dependability assurance. IEEE Trans. Software Eng., 31: 733-753. http://portal.acm.org/citation.cfm?id=1092850
3. Bhansali, P.V., 2005. Software safety: Current status and future directions. ACM SIGSOFT Software Eng. Notes, 30: 3. http://portal.acm.org/citation.cfm?doid=1039174.1039193
4. Lutz, R.R., 2000. Software engineering for safety: A roadmap. Proceedings of the Conference on The Future of Software Engineering Limerick, June 04-11, Ireland, pp: 213-226. http://portal.acm.org/citation.cfm?id=336556
5. Knight, J.C., 2002. Safety critical systems: Challenges and directions. Proceeding of the 24th International Conference on Software Engineering, May 19-25, IEEE Xplore Press, Orlando, Florida, pp: 547-550. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1007998
6. Dunn, W., 2003. Designing Safety Critical Computer Systems. IEEE-Computer, 36: 40-46. DOI: 10.1109/MC.2003.1244533
7. Herman, D.S., 2000. Software Safety and Reliability Basics. Software Safety and Reliability: Techniques, Approaches and Standards of Key Industrial Sectors. Wiley-IEEE Computer Society Press, ISBN: 978-0-7695-0299-1, pp: 520.
8. Schmid, D.C., 2002. Adaptive middleware: Middleware for real-time and embedded systems. Commun. ACM., 45: pp: 43-48. http://portal.acm.org/citation.cfm?id=508448.508472
9. Nissanke, N., 1997. Real-Time Systems. Prentice Hall International Series in Computer Science, Prentice Hall, London, ISBN: 0-13-651274-7.
10. Florio, V.D. and C. Blondia, 2008. A survey of linguistic structures for application-level fault tolerance. ACM Comput. Surveys, 40: 1-37. http://portal.acm.org/citation.cfm?id=1348246.1348249
11. Leveson, N.G., 1986. Software safety: Why, what and how. ACM Comput. Surveys, 18: 125-163. http://portal.acm.org/citation.cfm?id=7528
12. Software Safety, 1997. NASA Technical Standard, NASA-STD-8719.13A. http://satc.gsfc.nasa.gov/assure/distasst.pdf
13. Leveson, N., 1995. Safeware: System Safety and Computers. 1st Edn., Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN: 0201119722.
14. Bass, L., P. Clements and R. Kazman, 2003. Software Architecture in Practice. 2nd Edn., Addison-Wesley Publishing Company, Boston, Massachusetts, ISBN: 0-321-15495-9.
15. Fenton, N. and S. Pfleeger, 2003. Measuring External Product Attributes. Software Metrics-A Rigorous and Practical Approach. 2nd Edn., Thomson, pp: 337-359.
16. Rawashdeh, A. and B. Matalkah, 2006. A new software quality model for evaluating cots components. J. Comput. Sci., 2: 373-381. http://www.scipub.org/fulltext/jcs/jcs24373-381.pdf
17. Boehm, B., 1989. Software Risk Management. IEEE Computer Society Press, Los Alamitos, CA.
18. Kilidar, A.H., K. Cox and B. Kitchenham, 2005. The use and usefulness of iso/iec 9126 quality standard. Proceeding of the International Symposium on Empirical Software Engineering, Nov. 17-18, IEEE Xplore Press, Noosa Heads, Queensland, pp: 7-7. DOI: 10.1109/ISESE.2005.1541821
19. Singh, R., 1999. A systematic approach to software safety. Proceeding of the 6th Conference on Asia Pacific Software Engineering, Dec. 7-10, IEEE Xplore Press, Takamatsu, Japan, pp: 420-423. DOI: 10.1109/APSEC.1999.809632
20. Raheja, D.G. and M. Allocco, 2006. Assurance Technologies Principles and Practices. 2nd Edn., Wiley Inter Science, ISBN: 0471744913, pp: 472.
21. Jaffe, M.S. and N.G. Leveson, 1989. Completeness, robustness and safety in real-time software requirements specification. Proceeding of the 11th International Conference on Software Engineering, May 15-18, Pittsburgh, USA., pp: 302-311. http://portal.acm.org/citation.cfm?id=74587.74628

22. Firesmith, D.G., 2005. Are your requirements complete? J. Object Technol., 4: 27-43. http://www.jot.fm/issues/issue_2005_01/column3/
23. Young, R.R., 2004. The Requirements Engineering Handbook. Artech House, Norwood, MA., USA.
24. Firesmith, D.G., 2005. Engineering safety-related requirements for software-intensive systems. Proceeding of the 27th International Conference on Software Engineering, May 15-21, St. Louis, Missouri, USA., pp: 720-721. http://portal.acm.org/citation.cfm?id=1062455.1062635
25. MIL-STD-882C, 1993. System Safety Program Requirements, https://crc.army.mil/guidance/system_safety/882C.pdf
26. NATO, 1996. NATO standardization agreement STANAG 4404 safety design requirements and guidelines for munitions related safety critical computing systems.
27. Satish, R. *et al*., 1996. Run time assertion schemes for safety critical systems. Proceeding of the 9th IEEE Symposium on Computer Based Medical Systems, June 17-18, IEEE Xplore Press, Ann Arbor, Michigan, pp: 18-23. DOI: 10.1109/CBMS.1996.507119
28. Storey, N., 1996. Safety-Critical Computer Systems 1st Edn., Addison-Wesley, Boston, MA., USA., ISBN: 0201427877.
29. Dunn, W.R., 2002. Practical Design of Safety-Critical Computer Systems. Reliability Press, Solvang, CA., ISBN: 10: 0971752702.
30. Goble, W., 1998. Control Systems Safety Evaluation and Reliability. 2nd Edn., ISA Publisher, ISBN: 1556176368, pp: 739.
31. Bedford, T. and R. Cooke, 2001. Probabilistic Risk Analysis: Foundations and Methods. 1st Edn., Cambridge University Press, UK., ISBN: 0521773202, pp: 393.
32. Fault Tree Handbook, 1981. NUREG-0492, US NuclearRegulatoryCommission. http://www.nrc.gov/reading-rm/doc-collections/nuregs/staff/sr0492/sr0492.pdf
33. MIL-STD-882D, 1993. Standard practice for system safety, US department of defense, http://safetycenter.navy.mil/instructions/osh/milstd882d.pdf
34. ISA., 1998. Functional Safety of Electrical/Electronic/ Programmable Electronic Safety-Related Systems-part: General requirements. IEC-61508-1-1998. http://www.isa.org/Template.cfm?Section=Standards8&Template=/Ecommerce/ProductDisplay.cfm&ProductID=5764
35. Leucker, M., 2008. Checking and enforcing safety: Runtime verification and runtime reflection. http://ercim-news.ercim.org/content/view/459/699/
36. Zelkowithz, M. and I. Rus, 2001. Understanding IV and V in a safety critical and complex evolutionary environment: The NASA space shuttle program. Proceeding of the 23rd International Conference on Software Engineering, May 12-19, Toronto, Ontario, Canada, pp. 349-357. http://portal.acm.org/citation.cfm?id=381473.381510
37. Anderson, P., 2008. Detecting bugs in safety critical code. Dr. Dobbs J., February. http://www.ddj.com/development-tools/206104422
38. Holzmann, G.J., 2006 The power of ten: Rules for developing safety critical code. IEEE Comput., 39: 95-99. DOI: 10.1109/MC.2006.212