A hybrid approach to quantify software reliability in nuclear safety systems

P. Arun Babu a, C. Senthil Kumar b,c, N. Murali a

a Indira Gandhi Centre for Atomic Research, Kalpakkam 603102, India
b Safety Research Institute, Atomic Energy Regulatory Board, Kalpakkam 603102, India

ARTICLE INFO

Article history:
Received 9 April 2012
Received in revised form 21 June 2012
Accepted 25 June 2012

Keywords:
Software reliability
Safety critical software
Software verification
Mutation testing
Test adequacy
Software licensing

ABSTRACT

Technological advancements have led to the use of computer-based systems in safety-critical applications. As computer-based systems are being introduced in nuclear power plants, effective and efficient methods are needed to ensure dependability and compliance to high-reliability requirements of systems important to safety. Even after several years of research, quantification of software reliability remains controversial and unresolved issue. Also, existing approaches have assumptions and limitations, which are not acceptable for safety applications.

This paper proposes a theoretical approach combining software verification and mutation testing to quantify the software reliability in nuclear safety systems. The theoretical results obtained suggest that the software reliability depends on three factors: the test adequacy, the amount of software verification, and reusability of verified code in the software. The proposed approach may help regulators in licensing computer-based safety systems in nuclear reactors.

© 2012 Elsevier Ltd. All rights reserved.

1. Introduction

Nuclear power plants (NPPs) are replacing analog equipment with modern digital equipment for their safety-critical functions. Since software failures in such systems can be catastrophic and life-threatening, the increase in use of computer-based controls for safety-critical operations demand for a systematic evaluation of software reliability. Documented software failures in the nuclear industry include: (i) Therac-25 radiation therapy machine delivered high radiation, doses to patients (Lewesdon, 1993); (ii) files become inaccessible to the accountants using nuclear material tracking software at Kurchatov institute, Russia (Blair, 2001); (iii) Slammer worm disabled safety parameter display system for 5 h at Davis-Besse nuclear power station (Poulson, 2003); (iv) computer resets the control system after software patching and reboots at Edwin I. Hatch nuclear power plant (Krebs, 2008) and (v) Stuxnet worm infects nuclear plants in Iran running software controlled SCADA systems (Nicolas Falliere and Chien, 2011).

Nevertheless, the current licensing procedure for computer-based systems is based on deterministic criteria. Unlike hardware reliability, the software reliability is not a pure function of time. Software failures are result of design faults which are often difficult to visualize, classify, detect, and debug (Lyu, 1995). Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment (ANSI/IEEE, 1991, 1995). For a risk-informed regulation, a procedure for software reliability estimation is not yet been satisfactorily developed (Chu et al., 2008, 2010). This is due to factors such as software complexity, difficulty in identifying suitable metrics, difficulty in exhaustive testing and difficulty in quantifying effectiveness of test cases.

An ideal way to demonstrate that the software meets a required reliability is through formal verification. Formal verification is a method of proving certain properties in the designed algorithm, with respect to its requirement specification written in mathematical language/notation. Approaches to formal verification include formal proof and model checking. Formal proof is a finite sequence of steps which proves or disproves a certain property of the software, whereas model checking achieves the same through exhaustive search of the state space of the model. Unfortunately, it is not always feasible to ensure complete formal verification of software due to the difficulties involved such as state space explosion and difficulties involved in application of formal methods (Stidolph and Whitehead, 2003). Also, a major assumption in formal verification is that the requirements specification captures all the desired properties correctly. If this assumption is violated, the formal verification becomes invalid. Reliability estimates based on software testing has been recommended, and has been adopted for decades. Repeated failure free execution of the software provides a certain level of confidence in the reliability estimate. However, software testing can only indicate the presence of faults and not its absence.

This paper proposes a theoretical approach which aims to be intuitive and with as realistic assumptions as possible to estimate software reliability in critical systems. The proposed approach
combines results of software verification and mutation testing. The rest of the paper is structured as follows: Section 2 reviews related work in software reliability estimation, and discusses associated problems. Section 3 describes the proposed approach. Section 4 analyzes the theoretical results obtained. Section 5 concludes the paper and gives further direction of research.

2. Related work and motivation

2.1. Software in mission and safety critical systems

Software built for mission and safety critical applications are different from business-critical or general purpose systems. Generally, software in critical systems are: (i) smaller and focused; (ii) rugged and have fault tolerant features; (iii) designed with defense in depth; (iv) expected to have lower failure rates; (v) meant to fail in fail-safe mode; (vi) not expected to rely on human judgment or intervention to initiate safety action and (vii) written in safe subsets of programming languages such as: Motor Industry Software Reliability Association (MISRA) C/C++ (MISRA, 2004, 2008), Joint Strike Force (JSF)+++ (JSF, 2005) and SPARK Ada (Barnes, 2003). In addition, the software undergoes rigorous review, static analysis, dynamic analysis and independent verification and validation processes.

2.2. Software in nuclear reactors

Systems in a nuclear reactor can be classified into three categories based on safety (AERB, 2003):

1. Safety critical: systems important to safety, provided to assure that under anticipated operational occurrences and accident conditions, the safe shutdown of the reactor followed by heat removal from the core and containment of any radioactivity is satisfactorily achieved.

2. Safety related: systems important to safety which are not included in safety critical systems and which are required for the normal functioning of the safety systems.


For each category, the International Atomic Energy Agency (IAEA) as well as the atomic energy regulator in the respective countries issue guidelines on best practices in software requirement analysis, defense in depth design, safe programming practices, verification and validation processes, etc. The regulators expect a formal systematic review of the software and its associated hardware, using requirement specifications and independent reviews. However, the difficulty arises in trying to implement the high level guidance (IAEA, 2009) and establishing a working consensus (Chu et al., 2008). Deterministic analysis such as hazard analysis and formal methods are generalization of the design basis accident methodology used in the nuclear industry. Probabilistic analysis is considered more appropriate as software faults are by definition design faults.

As safety systems in a nuclear power plant are categorized based on their importance to safety, for computer based systems, the IEC standards give requirements in the form of Safety Integrity Levels (SILs) (IEC-61508-5, 1998). SIL is specified in terms of a number from one to four based on the probability of failure. SIL-1 represent the lowest safety integrity level with target average probability of failure on demand (PFD) between 10^{-2} and 10^{-1}. SIL-4 is the highest with PFD between 10^{-5} and 10^{-2}. Common safety functions in NPPs are governed by defense in depth principles: reactivity control, maintenance of fuel integrity, control of pressure boundary, continuation of core cooling and prevention of radioactivity release. In view of the inherent complexity in such control software, it is difficult to assess the failure probability of software and quantify its influence during an accident situation.

2.3. Reliability growth and defect prediction models/methods

There exists two broad categories of methods for estimation of the reliability of software systems: white-box and black-box models. The group of white-box models consists of models that work based on the knowledge of systems’ internal structure and processes within them. This knowledge may be expressed by different kinds of models such as architecture models and test case models. On the other hand, the group of black-box models encompasses much larger number of models that treat the software as a monolithic whole, i.e. as a black-box. While using a software reliability growth model, it is assumed that the test profile reflects the actual operational profile of the system being analyzed. However, the use of a particular software component will vary due to the characteristics of the process controlled. This brings out the significance of operational profile in software reliability estimation. (Pasquin et al., 1996) proposed a method to perform sensitivity analysis of operational profile in software reliability estimation by providing different operational profile estimates to the reliability model and comparing with the actual measurements. The study emphasized that the accuracy of the model and the effect of operational profile has strong relation with the number of test cases executed. This is because, when more number of operational profiles are used, a high degree of coverage is reached. Work such as (Chandran et al., 2010) show that the reliability models are sensitive to different factors such as test coverage and time between failures.

Considerable research (Gokhale, 2007, 2004, 2001, 2005) has been performed in the field of software reliability ranging from reliability growth models, architecture based models, application of techniques such as software rejuvenation, fault injection and formal approaches. After several years of research, it is widely accepted that quantification of safety critical software reliability is infeasible using statistical models. Even today, a widely acceptable method to quantify software reliability is not available. Most of the existing software reliability modeling and defect prediction methods have assumptions, which are questionable in safety and mission critical applications (Wood, 1997, Goel, 1985). For example:

1. There are fixed number of faults in the software being tested and whenever a fault is found, it is removed instantaneously, without inducing new fault.
2. Each fault has the same contribution to the unreliability of the software.
3. The probability that two or more software failures occur simultaneously is negligible.
4. Enough and accurate software failure data is available for analysis.
5. The execution time between failures is assumed to be having a particular distribution.
6. The hazard rate for a single fault is constant.
7. The tests conducted represent the operational profile.

Critical reviews (Fenton and Neil, 1999, 1985) and controlled experiments (Kai-Yuan et al., 2008) have thrown light on assumptions, limitations and applicability of software reliability growth and defect prediction models. Choosing the right model that suits a particular situation/software is also considered a complex task (Stringfellow and Andrews, 2002, 2004). Also, some of the models have been reported to be less useful in certain development methodologies such as agile approach to software development (Beckhaus et al., 2009).
Reliability of a software system not only depends on the structure of the software but also on the runtime information such as frequency of component reuse and the interactions between the components. The design requirement and code of the application are reviewed for the software structure, but the simulation or the executions of the software provide the runtime characteristics. During the simulation process, when a fault is identified, the fault is corrected and the components experience the reliability growth (Gokhale and Lyu, 2005). It should be noted that faults are discovered only if that portion of the software, where it lies is executed. If a fault prevents the execution of some portion of the software, until it is removed, the faults downstream may not be identified. The limitation of the simulation approach is its inability to use a time dependent rate for event occurrences.

2.4. The need for a new approach

Traditional reliability models assume availability of accurate and adequate software failure data, which is often difficult to collect. Also, for a newly built plant with no failure history, the software reliability estimation methods do not apply. Even in cases, where such information were made available; the type of software, its associated faults and the frequency of faults in traditional systems are different from critical systems. Hence, it is unclear if traditional reliability growth models are suitable for critical applications. Studies such as (Butler and Finelli, 1993, Littlewood, 2000) suggests that the amount of time required in testing to demonstrate ultra-high reliability is infeasible (e.g.: to quantify $10^{-8}$/h failure rate requires at least $10^8$ h of software testing).

The principle findings of a US nuclear regulatory commission report (Chu et al., 2010) quotes: “most of the existing quantitative software reliability methods were not developed specifically for supporting quantification of software failure rates and demand failure probabilities to be used in reliability models of digital systems”; “all methods are based on assumed empirical formulas that are not applicable in all situations”. Hence, there is a need for a robust software reliability estimation method suitable for critical applications related to safety.

3. The proposed approach

This section describes an approach for quantifying software reliability using software verification and mutation testing.

3.1. Assumptions and limitations

1. Software for a system related to safety may be divided into five modules (Fig. 1):
   (a) A hardware-interface module, which can take inputs from sensors, and give outputs to actuators, blowers, heaters, etc.
   (b) A user-interface module, which interacts with the user.
   (c) A network-interface module, which can share soft outputs/inputs with other connected systems.
   (d) A diagnostic module, which checks the state of the system at regular intervals.
   (e) A core module which performs the system’s intended function.

2. The approach focuses on system failures due to pure software failures (the shaded portion in Fig. 2), and not on failures arising due to hardware or hardware-software interaction.

3. The software of the core module is written in portable C-programming language adhering to MISRA standards.

4. The software is assumed to be single-threaded and can run on bare hardware without any operating system support. This is due to the fact that most of the safety–critical software in nuclear reactors are simple and focused embedded systems. However, for complex safety–critical systems which require multi-tasking, multi-threading or nested-interrupt support; a trusted, safe and certified real-time operating system must be used. The reliability of such operating systems is assumed to be $\approx 1$.

5. The software is fused into ROM in order to prevent malwares from modifying it.

6. The output of the software depends only on the current inputs.

3.2. Prerequisites for the approach

As the proposed approach is based on mutation testing and software verification, the prerequisites for the approach are:

3.2.1. Set of test cases

Software in safety applications often require 100% Modified Condition Decision Coverage (MC/DC) (Hayhurst et al., 2001) and Linear Code Sequence and Jump (LCSAJ) coverage (Woodward et al., 1980). Achieving these criteria may require hundreds (in some cases, thousands) of test cases. Such large numbers of test cases are often difficult to hand code and may require automatic test case generation. The test cases may be generated through techniques such as model based testing (Utting and Legear, 2006), controlled random number generation (Duran and Ntafos, 1984, 2005), equivalence partitioning and boundary value analysis (Burnstein, 2003). The generated test suite must be reduced by removing redundant test cases which follow the same path of execution. This can be achieved through the coverage information from coverage analysis tools. One example of such tool is gcov (von Hagen, 2006) with – abcfu arguments. gcov is a test coverage program included with gcc (von Hagen, 2006), and – abcfu arguments implies show information about: all blocks, branch probabilities, branch counts, function summaries and unconditional branches respectively.

The results of test case generation is the test coverage which; to impart rigour, and to give weightage to large, complex, and frequently called functions; is calculated as a weighted average, given as:

$$\text{Test coverage} = \frac{\sum t_i w_i}{\sum w_i}$$

where $t_i$ is the conservative test coverage achieved for each function during system testing, and is defined as:

$$t_i = \min (\text{LC}SA, \text{MC/DC}, \text{Statement coverage})$$

and $w_i$ is the weight assigned to each function as:

$$w_i = \text{No. of statements} \times \text{Cyclomatic Complexity} \times \text{Frequency of function call}$$

3.2.2. A test oracle

The generated test cases are verified by checking against functional specification, invariants and safety properties. The test cases which satisfy these conditions are termed as verified test cases. It may not be always feasible to write complete functional specifica-
tion, safety properties, and invariants to verify all the test cases; in such cases, the test suite is partially verified. If a path in the program is proven or verified, then the reliability of the path is assumed to be \( \approx 1 \).

3.2.3. Set of mutants

Mutation testing (Jia and Harman, 2011) is a fault injection technique, where realistic faults are induced intentionally into the source code. The fault induced program is known as a mutant. The proposed approach requires a set of single fault (first order) mutants. The number of mutants that can be generated depends on the number of mutant operators and the size of the code. As the approach is statistical in nature, the number of generated mutants should be as large as feasible, to achieve the required accuracy. The result of mutation testing is the mutation score, defined as:

\[
\text{Mutation score} = \frac{K}{G - E}
\]

where \( K \) is the number of mutants killed by the test cases, \( G \) is the number of mutants generated and \( E \) is the number of equivalent mutants. Automatic detection of equivalent mutants is an undecidable problem, and several methods have been proposed to solve it (Schuler and Zeller, 2010, 1996, 1997) with certain amount of accuracy.

3.2.4. Test adequacy computation

Safety critical software undergoes rigorous testing, but it is impractical to expect that all possible execution paths in a program can be tested. Hence the rigour in software testing may be expressed as the adequacy of testing, and can be estimated using Eqs. (1) and (2) as:

\[
\text{Test adequacy} = \frac{\text{Mutation score}}{\text{Test coverage}}
\]

The computed test adequacy is in the range \([0,1]\); and is useful in achieving a realistic estimate of the reliability based on software testing approaches described in Sections 3.3 and 3.4.

3.3. Approach for reliability estimation – 1

The approach is similar to the Monte Carlo method (Kalos and Whitlock, 2008) of calculating the value of \( \pi \). In which, random darts are thrown at a square in which a circle is inscribed (Fig. 3). Similarly, a program under test may be visualized as a graph (Fig. 4) consisting of verified (indicated by \( \rightarrow \)) and un-verified (indicated by \( \leftarrow \)) paths; and randomly induced fault is the dart thrown at it. In the method of \( \pi \) value calculation, a random dart may either fall inside the circle or outside it; similarly, an induced fault may have three possible outcomes (i.e.: result of the mutant execution): (i) it fails at least one of the verified test cases, indicating that the fault has been induced in a verified path; (ii) passes all verified test case but fails at least one of the un-verified test cases, indicating that fault has been induced in an un-verified path; and (iii) does not fail any of the test cases (aka. the unkilled mutant), indicating that the induced fault may not have any effect on the program.

By generating such large number of mutants, and ignoring all the unkilled mutants, the reliability is estimated as:
...reliability estimate. Also, it is difficult to integrate operational profile into the approach. This approach is more suitable for non-safety applications, but may also be used for systems important to safety.

3.4. Approach for reliability estimation – 2

The approach is similar to the approach – 1; and is based on the principle that, if in a given program, reliability of an execution path $p$ is known, then other paths in the program sharing code with the path $p$ also share the reliability of path $p$. For example: in Fig. 4, a program has four paths $p_1$, $p_2$, $p_3$ and $p_4$; and the paths $p_3$ and $p_4$ share reliability of $p_2$. If the reliability of path $p_2$ (i.e.: $R_2$) is known, then the reliability of any path $p_i$ (i.e.: $R_i$) can be estimated by:

$$R_i = R_2 \times (\text{Fraction of code shared between } p_i \text{ and } p_2)$$

The fraction of code shared between paths is estimated statistically through mutation testing, by injecting faults in paths for which reliability is unknown (e.g.: path $p_3$). For example: in (Fig. 5) the first injected fault causes the test cases running through paths $p_2$, $p_3$, and $p_4$ to fail; while as the second injected fault fails test case running through path $p_2$. If several such single fault (first order) mutants are generated, and are tested against the test cases, then the fraction of code shared between paths $p_i$ and $p_2$ may be estimated by:

$$\text{Fraction of code shared between } p_i \text{ and } p_2 = \frac{F_{i2}}{F_{22}}$$

Fig. 4. Paths in a program (→ indicates a path whose reliability is known).

Test adequacy

\[
\text{No. of times at least one of the verified test cases failed} \times \frac{\text{Total no. of mutants killed}}{}
\]

The advantage of this approach is its simplicity, but its results could be biased when estimating reliability for a highly verified software, i.e.: if the mutation testing is not effective enough, then large number of verified test cases may incorrectly lead to a higher reliability estimate. Also, it is difficult to integrate operational profile into the approach. This approach is more suitable for non-safety applications, but may also be used for systems important to safety to get an initial/quick approximate estimate of the reliability.

3.4.1. Pseudo-code of the approach

1. let $T = \{t_1, t_2, \ldots, t_N\}$ be the set of $N$ generated test cases, where $t_i$ represents an unique path $p_i$ in the program. And let $\text{adequacy}(T)$ represent the adequacy of the test cases $T$ calculated using Eq. (3).
2. let $V_i$ represent the number of times at least one of the verified test case in $T$ kills a mutant, given that a fault is induced in the path $p_i$.
3. let $U_i$ represent the number of times an un-verified test case $t_i$ in $T$ kills a mutant, given that a fault is induced in the path $p_i$ ($U_i > 0$).
4. let $M$ be the set of mutants generated for the program.
5. let $I_m$ represent the set of un-verified test case indices in $T$, which can kill the mutant $m$.
6. let $F_i$ represent the fraction of code the path $p_i$ shares with other verified paths.
7. for each mutant $m$ in $M$:
   (a) $I_m = \phi$;
   (b) for each un-verified test case $t_i$ in $T$ if $t_i$ kills the mutant $m$ then:
      $U_i \leftarrow U_i + 1$
      $I_m \leftarrow I_m \cup \{i\}$
   (c) if $I_m = \phi$ then ignore the mutant $m$ and continue with next mutant in step 7.
      else if $\exists t_i$ in $T$ such that: $t_i$ is a verified test case and kills the mutant $m$ then
      $V_i \leftarrow V_i + 1$
   end if
   end if
8. $F_i = \{\frac{1}{V_i} \text{ if path } p_i \text{ is verified} \}$ otherwise
9. Reliability =

$$\text{adequacy}(T) \times \sum_{i=1}^{n} \frac{F_i}{n}$$

if all paths are equally likely to be executed.

$$\text{adequacy}(T) \times \sum_{i=1}^{n} \frac{(1 - V_i \times O_i)}{n}$$

if path $p_i$ has a probability $O_i$ of execution (the operational profile).

Unlike approach – 1, the approach – 2 allows integration of the operational profile in the reliability estimate. Also, it ensures that un-verified test cases fail during mutation testing; thus eliminating any bias present due to large number of verified test cases. This property makes the reliability estimate realistic, and more suitable for systems important to safety.

4. Theoretical results

The two approaches described in Section 3 provides a framework for assessing software failure probability to support the licensing process. When little or no information on the operation...
profile of the software is available (e.g.: during commissioning of a new plant), the proposed approaches can be adopted for a initial estimation of software reliability.

Along with the reliability estimate, it is equally important to understand on what factors does the estimated reliability depend on, in order to achieve target reliability. Hence, if \( P \) represents the number of verified test cases, and assuming that all paths of the software are equally likely to be executed, then (from Step-9 in Section 3.4.1):

\[
\text{Reliability} = \frac{\text{adequacy} (T)}{C^2 \sum_{i=1}^{N} F_i/N}
\]

\[
\text{Reliability} = \text{adequacy} (T) \times \left( \frac{P + \sum_{i=1}^{N-P} F_i}{N} \right)
\]

\[
\text{Reliability} = \text{adequacy} (T) \times \left( \frac{P}{N} + \frac{\sum_{i=1}^{N-P} F_i \times (N-P)}{(N-P) \times N} \right)
\]

\[
\text{Reliability} = \text{adequacy} (T) \times \left( x + y \times \frac{(N-P)}{N} \right)
\]

\[
\text{Reliability} = \text{adequacy} (T) \times \left( x + y \times \left( 1 - \frac{P}{N} \right) \right)
\]

\[
\text{Reliability} = \text{adequacy} (T) \times (x + y - xy)
\]

**4.1. Factors affecting the estimated reliability**

Eq. (4) represents the estimated software reliability, which is a function of three variables: (i) adequacy (\( T \)), the test adequacy; (ii) \( x \), the fraction of verified test cases; and (iii) \( y \), the fraction of code shared between \((N-P)\) un-verified paths and \( P \) verified paths, which is an indication of the software reusability. The adequacy(\( T \)), \( x \), and \( y \) values are in range \([0,1]\). The case, where \( P = N \) or \( x = 1 \), implies that all the given test cases have been verified and there are no un-verified paths left (i.e.: \( y = 0 \)), hence \( \text{Reliability} = \text{adequacy}(T) \).

**4.2. Achieving target reliability**

The Eq. (4) helps in choosing the combination of \( x \) and \( y \) values required to achieve target reliability (Fig. 6). For a given reliability, as \( x \) increases, the requirement for \( y \) decreases. The decrease in required value of \( y \) exhibits linear to exponential behavior as the target reliability increases, and becomes a step function as \( \text{Reliability} \rightarrow \text{adequacy}(T) \) and \( x \rightarrow 1 \) (Fig. 4).

**4.3. Properties of the software**

For software having high test adequacy (as required by most of the safety applications), the values of \( x \) and \( y \) help in understanding the properties of the software, and in making further recommendations to the development/testing team. For example:

1. When \( x \approx 0 \) and \( y \approx 0 \):
   - The software requires rigorous software verification.

![Fig. 6. Combination of x and y values for various target reliability (0.05–0.99), when test adequacy = 0.99.](image-url)
2. When $x = 0$ and $y \approx 1$:
   The software has very high reusability. To improve the confidence on the reliability estimate, more software verification must be recommended for such systems.
3. When $x \approx 1$ and $y = 0$:
   Nearly all the generated paths have been verified, and very few groups of un-verified paths have been left out, which do not share much code with the verified paths.
4. When $x = 1$ and $y \approx 1$:
   An ideal scenario, where almost all the generated paths have been verified. Few small groups of un-verified paths have been left out, which share most of code with the verified paths.

4.4. Limitation of the approach

It is well known that software reliability of 100% can never be achieved. Hence, if the proposed approach estimates the reliability as 1, it only indicates that: the test suite which has resulted in 100% test coverage and mutation score $= 1$, has been verified; which implies that the software has very high reliability ($\approx 1$). The accuracy in the reliability estimate can be further improved by: (i) improving the effectiveness of the mutation testing by using good number of mutant operators which induce realistic faults into the software; and (ii) reducing the uncertainty in mutation score calculation by detecting equivalent mutants correctly.

5. Conclusion and future work

There is an urgent need to demonstrate the safety of computer based systems in nuclear plants. The lack of commonly accepted methods on the assessment of software reliability may hinder the licensing process of such safety systems.

This paper has shown how software verification can be combined with software testing to assess a realistic estimate of the software reliability. The approaches presented can be used by the regulators to estimate the software reliability, and to ensure safety and dependability. This paper also shows how the amount of software verification may be varied to achieve target reliability. The theoretical results obtained suggest that the test adequacy is a major factor in determining the software reliability in systems related to safety. Hence, such systems must have high test coverage and mutation score. Considering the fact that all safety-critical software undergo rigorous testing and verification to ensure correctness; the proposed approach is suitable for any safety-critical software.

Future work includes implementation of the proposed approach on various safety related software, and to study the relationship between estimated software reliability and the associated software metrics to better understand the properties of reliable software. Also, the effect of uncertainties in mutating testing on reliability estimate must be investigated.

References


