A Methodology for Software Reliability Risk Assessment

Xu Jian, Yan Han, Li Qianmu
Institute of Computer Science and Technology, Nanjing University of Science and Technology, Nanjing Jiangsu, China, 210094, dolphin.xu@mail.njust.edu.cn
{yanhan,qianmu}@mail.njust.edu.cn

Abstract

Software reliability risk assessment based on its architecture is an important technology used to assure the quality of software products early in the software lifecycle. In this paper, we expand the work of “A methodology for architecture level reliability risk analysis” by S. Yacoub and et al. to present a more simple methodology for risk assessment at the design stages of the software development lifecycle. Our methodology adopts the same definition for component risk as the above work, namely component risk is a combination of two factors: the probability of component failure (complexity) and the consequence of component failure (severity). However, we focus on the severity factor of risk, and estimate it by using the ripple effect of component failures. We combine complexity and severity factors to develop risk factors of components. We also develop a risk analysis algorithm that aggregates risk factors of components to the architectural level. Using the risk aggregation algorithm and the risk analysis model, we show how to analyze the overall risk factor of the system as the function of the risk factors of its constituting components. A case study of the pacemaker is used to illustrate the application of the methodology. The results show that the proposed methodology can be used to identify critical components and to investigate the sensitivity of the system risk factor to variations in the heuristic risk factors of components. Moreover, in order to validate the proposed component risk analysis method, we compare the predicted results derived from our method with those derived from the simulation model for the same case study.

Keywords: Risk Assessment, UML Specification, Software Architecture, Dynamic Metrics

1. Introduction

Many companies develop and maintain different types of large-scale critical software systems. The impact of failure of these systems would be enormous. It is therefore very important to identify any defects in these software systems in advance to reduce the occurrence of failures. However, testing all modules in these types of large-scale systems to identify defects is very expensive. Software risk assessment is an essential process to ensure high quality software products by identifying complex modules that require detailed inspection and estimating potentially troublesome modules. Risk assessment at the early design phase of the software is more feasible and more beneficial than assessment at later development phases. By performing risk assessment at architecture level, we can identify complex and high risk software modules, thus enables us to guide software development, testing, and maintenance process.

According to NASA-STD-8719.13A [1], risk is a function of: the possible frequency of occurrence of an undesired event, the potential severity of resulting consequences, and the uncertainties associated with the frequency and severity. The STD-8719.13A standard defines several types of risks, for example, cost risk, performance risk, reliability risk, etc. In this study, our interest is reliability risk, which depends on the probability that the software product will fail in the operational environment and the adversity of that failure.

The Unified Modeling Language (UML) is emerging as an industry standard for software architecture specification. UML models capture not only the application static aspects, but also dynamic ones. Concepts like use cases, scenarios, actions and events are not only used to discover the relevant domain objects, but also to model the interactions between these objects (e.g. in collaboration diagram), and thus to shape the overall architecture of the system. To track the quality of software
products, there is an increasing need for an automated method that can automatically collected and analyzed measurable parameters in the early software design phase based on UML artifacts. Further, when calculating the metrics from architectural descriptions based on UML models, we achieve independency of languages and human factors.

The research [7] presented reliability risk assessment approach which covers during analysis and design phase of software development. This work defines risk as a combination of two factors: the probability of malfunctioning (failure) and the consequence of failure (severity). Its assessment starts from defining a software architecture model which is realized and modeled using UML specifications, mainly sequence diagrams and state machine diagrams. The dynamic complexity metrics are used to define complexity factors for the architecture elements (components). And the dynamic coupling metrics are used to consider the behavior of interaction complexity between objects. The whole risk assessment process is strongly based on simulation of architecture models. Due to simulation, the severity analysis is performed by running the simulator, injecting faults and observing simulation outputs and reports. However, this process a high cost one. Firstly, a simulator need be developed for each architecture model. Then, the different type software faults need be injected to simulate software failures. Finally, the analyst need analyze simulation outputs and reports to evaluate the severity level of software failures, to calculate the CDG (Component Dependence Graph) parameters, such as probability of transitions and average execution time of a component, and to develop the CDGs manually.

We expand the work [7] and adopt the same risk definition. We also develop a risk assessment methodology to assess the risk of software systems in the early software design phase based on UML artifacts. However, we focus on the severity factor of risk, and estimate it by developing an architectural level severity assessment technique based on the ripple-effect of component failures. Moreover, we make adaptations to the risk analysis model and risk aggregation algorithm so as to measurable parameters that can be automatically collected, and the risk factor of a scenario or system that can be evaluated without simulation.

This paper is organized as follows. Section 2 proposes a new methodology to perform reliability risk assessment at the architecture level based on the UML artifacts, where we mainly focus on the quantification of the complexity factors and severity factors, and the description of the risk analysis model and the risk aggregation algorithm. Section 3 is an application of the methodology to an illustrative case. Section 4 summarizes the related work. Finally we conclude this paper and discuss future extensions.

2. The risk assessment methodology

In this section we introduce a multi-hierarchy risk assessment methodology based on UML models. The proposed methodology is defined by the following steps:

1) Performing the dynamic complexity for components using the same method proposed by S.Yacoub and et al [7];
2) Performing severity analysis using the ripple effect of component failures;
3) Assessing risk factors of components and identifying critical components in a given scenario;
4) Developing CRDGs for risk assessment purpose;
5) Performing risk assessment and analysis using a risk aggregation algorithm.

2.1. Component Complexity Analysis

As mentioned above, we focus on the second factor in the risk definition [7]. Therefore, in this step, we use the same component complexity analysis method developed by S.Yacoub and et al [7] to evaluate the first factor in the risk definition, the probability of failure. Considering the integrity of the assessment process, we simply describe the complexity analysis method here.

The probability of failure depends on the probability of existence of a fault combined with the possibility of exercising that fault. During the early phases of the software lifecycle, it is difficult to estimate the probability of failure of software components; therefore they use quantitative factors that have been proven to be correlated with the probability of faults in the development of software
components, such as complexity factors [2,3]. Moreover, to account for the probability of a fault manifesting itself into a failure, they use dynamic metrics as indicators of the probability of failure of a software component. As components are invoked, they become active for a specific duration performing the requested functionalities. The most active set of components are sources of errors [4] because they execute more frequently and experience numerous state changes. Therefore, there is a higher probability that, if a fault exists in an active component, it will easily manifest itself into a failure.

McCabe cyclomatic complexity is a measure of program complexity. It is obtained from the control flow graph and defined as \( \text{cc} = e - n + 2 \), where \( e \) is number of edges and \( n \) is number of nodes. We use a measure of component complexity similar to McCabe’s cyclomatic complexity. However, in contrast to McCabe’s cyclomatic complexity which is based on the control flow graph of the source code, our metric for component’s dynamic complexity is based on the UML state charts that are available during the early stages of the software lifecycle. The state chart of each component \( C \) has a number of states and transitions between these states that describe the dynamic behavior of the component.

For each scenario \( S \), a subset of all states of component \( C \) is visited and a subset of all transitions is traversed. Let \( S' \) denote the subset of states for a component \( C \) visited in the scenario \( S \), and \( T' \) denote the subset of transitions traversed in the state chart of component \( C \) in the scenario \( S \). The subset of states \( S' \) and the corresponding transitions \( T' \) are mapped into a control flow graph. By analogy with McCabe’s cyclomatic complexity, we define the dynamic complexity of component \( C \) in the scenario \( S \) as

\[
\text{cdx}(C') = |T'| - |S'| + 2
\]

(1)

where \( |T'| \) and \( |S'| \) is number of edges and nodes in this graph respectively.

2.2. Severity Analysis

The complexity of a component is not a sufficient measure for assessing the risk associated with its failure. Some components could have low complexity measure, but they play a major role such that their failure could cause catastrophic failures. Moreover, it is known that a software fault and the resulting error in one component can be propagated to the other interacting components causing their failures. Component failures are, therefore, seldom independent. Thus, we propose a ripple-effect based method to study the second factor in our risk definition, the consequence of a failure (severity). The proposed methodology takes into consideration the severity associated with each component based on how its failures affect the rest of the system.

The following definitions are necessary to estimate the severity of component failures.

**Definition 1 (Dependence Relation).** Let \( X \) be a finite set of components, \( X = \{C_1, C_2, ..., C_n\} \), and \( R \) be a binary relation on the set \( X \). \( R \) can be denoted as \( R = (C_i \rightarrow C_j) \) if component \( C_i \) calls any methods of component \( C_j \), we say \( C_i \) is dependent on \( C_j \), denoted by \( C_i \rightarrow C_j \). Conversely, if component \( C_i \) never calls any methods of component \( C_j \), we say \( C_i \) is independent of \( C_j \), denoted by \( C_i \not\rightarrow C_j \). Furthermore, if \( C_i \rightarrow C_j \) and \( C_j \rightarrow C_i \), then \( C_i \leftrightarrow C_j \).

**Definition 2 (Dependence Matrix).** Let \( X \) be a finite set of components, \( X = \{C_1, C_2, ..., C_n\} \), and \( R \) be a dependence relation on the set \( X \). A dependence matrix on the relation \( R \) can be denoted as \( M_R = (C_i \rightarrow C_j) \), where \( C_i \) denotes the dependence relationship between component \( C_i \) and \( C_j \), and \( n \) is the cardinality of the set \( X \). \( M_R \) is an \( n \times n \) matrix, where \( M_{ij} = 1 \) if \( C_i \rightarrow C_j \), \( M_{ij} = 0 \) otherwise.

**Definition 3 (Right Neighbor).** Let \( X \) be a finite set of components, \( X = \{C_1, C_2, ..., C_n\} \), and \( R \) be a dependence relation on the set \( X \). \( M_R = (C_i \rightarrow C_j) \) be a dependence matrix on the relation \( R \). If \( C_i \rightarrow C_j \), we say \( C_j \) is a right neighbor of \( C_i \). All right neighbors of component \( C_i \) is denoted by \( RN[C_i] \).

The greater the cardinality of \( RN[C_i] \) is, which denotes that component \( C_i \) may be a local center node and plays a major role, the more severe the severity of its failure is.
Definition 4 (Reachability Matrix). Let $X$ be a finite set of components, $X = \{C_1, C_2, \ldots, C_n\}$, and $R$ be a dependence relation on the set $X$. The transitive closure of $R$ is given by $R^+ = \bigvee_{i=1}^n R^i$. Therefore, the corresponding matrix for $R^+$ can be denoted as $M_{R^+} = M_R \cup M_R^2 \cup \ldots \cup M_R^n = \bigcup_{i=1}^n M_R^i$, where $M_R^0 = M_R$, $i=2,3,\ldots,n$. We say $M_{R^+}(C_j)$ is a reachability matrix for the system that are composed of components in the set $X$.

Definition 5 (Left Neighbor). Let $X$ be a finite set of components, $R$ be a dependence relation on the set $X$, and $M_{R^+}$ be a reachability matrix on the transitive closure $R^+$ of $R$. For $C_i, C_j \in X$, $(i \neq j)$, if $C_i$ can reach to component $C_j$ after $l$ steps ($0 \leq l \leq N$). Here, $N$ is a finite positive integer. We say $C_i$ is a left neighbor of $C_j$. All left neighbors of component $C_j$ is denoted by $LN[C_j]$.

The greater the cardinality of $LN[C_j]$ is, the more components which depend on component $C_j$ to perform specified tasks, the more severe the severity of its failure is.

In order to assess the consequence of a failure of component $C$, we take into consideration the severity based on how its failures affect its left and right neighbors. Let $RN[C]$ and $LN[C]$ be the set of right neighbors and the set of left neighbors, and $\alpha$ and $\beta$ be a weight factor of $RN[C]$ and $LN[C]$ respectively. Thus, the severity of a failure of component $C$ is estimated by

$$svrty(C) = \begin{cases} \alpha |RN[C]| + \beta (|LN[C] - RN[C] \cap LN[C]|) & \alpha \geq \beta \\ \alpha (|RN[C] - RN[C] \cap LN[C]|) + \beta |LN[C]| & \alpha < \beta \end{cases}$$

For example, Figure 1 shows a predigestion model of software architecture, which retains only direction semantic of connector to depict dependence relationships between components. Although this model is very simple, from the macro point of view, it provides enough information to study the consequence of a failure. The corresponding dependence matrix and reachability matrix are demonstrated via the following matrix $M_R$ and $M_{R^+}$ respectively, where $M_R = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ and $M_{R^+} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$. Supposed that the set of right neighbors has the same weight factor as the set of left neighbors, namely $\alpha=\beta=0.5$. If Component $C_5$ (Component 5 is abbreviated to $C_5$, and other components take the same notation) fails, $RN[C_5] = \emptyset, LN[C_5] = \{C_1, C_2, C_3, C_4\}$ and $svrty(C_5) = 2$. Similarly, if Component $C_4$ fails, $RN[C_4] = \{C_1\}, LN[C_4] = \{C_2, C_3\}$ and $svrty(C_4) = 1.5$.

**Figure 1.** Predigestion Model of Software Architecture
2.3. Develop Reliability Risk Factors for Components

In this step, we calculate a risk factor for each component in the architecture based on its complexity and severity factor as shown in the formula (2)

$$ \text{Risk}(C_i) = \text{cpx}(C_i) \times \text{svrty}(C_i) $$

where $\text{cpx}(C_i)$ and $\text{svrty}(C_i)$ is the dynamic complexity and the severity of the failure of component $C_i$.

To obtain the component risk factors, we propose a component risk assessment algorithm. This algorithm takes UML artifacts in the early software design phrase, such as state charts, sequence diagrams and class diagrams, as inputs. Before describing the algorithm in detail, we provide a formal definition of state chart and sequence diagram with states.

**Definition 6** (State Chart, SC). A SC is defined as follows: $SC = (S, E, s_0)$, where $S$ is set of nodes used to represent all states in the whole lifecycle of component, $E$ is set of directed edges used to represent all transitions between these states, and $s_0$ is the start node, i.e., $S = \{s_0, s_1, \ldots, s_i\}$, $E = \{e_1, e_2, \ldots, e_k\}$, $e_i = (s_n, s_m)$, where $s_n$ and $s_m$ is the source and target state of transition $e_i$ respectively, in case of $e_i$ is a method name, transition $e_i$ is triggered by the external invocation; in another case of $e_i$ is empty, transition $e_i$ is triggered by time.

A scenario is a set of component interactions triggered by specific input stimulus. In UML models, one main way to model scenarios is using sequence diagrams. In order to depict the states visited and state changes for a component in a given scenario, we extend sequence diagrams with state notations.

**Definition 7** (Sequence Diagram with states, SDWS). A SDWS is defined as follows: $SDWS = (C, \text{Msg})$, where $C$ is set of components, $C = \{C_1, C_2, \ldots, C_i\}$, and $\text{Msg}$ is set of interactions between these components, $\text{Msg} = \{ \langle (C_i, s'_i, s''_i), \text{msgName}, (C_j, s'_j, s''_j) \rangle | C_i, C_j \in C \}$, where $\text{msgName}$ is the name of method invocation, $C_i$ and $C_j$ is the source and target component for a invocation respectively, $s'_i$ and $s''_i$ is the state of component $C_i$ before and after invocation, $s'_j$ and $s''_j$ is similar to $s'_i$ and $s''_i$.

The proposed algorithm for component risk assessment is described in Figure 2.

![Figure 2. The component risk analysis process](image-url)

2.4. A Risk Aggregation Algorithm

So far we have developed risk factors for components. To assess a risk factor for the system or for individual subsystems, in case of a hierarchical system, that are composed of components, we need to define a risk aggregation algorithm. The system or individual subsystems risk factor is obtained from
aggregating the risk factors of individual components. In this section, we first extend CDGs model [8] to represent scenarios, subsystems and system for the purpose of risk analysis, and then give a risk aggregation algorithm to assess the risk of the overall system or individual subsystems based on the extended model respectively.

**Definition 8 (Component Risk Dependency Graph, CRDG).** A CRDG is defined as follows: CRDG = \( (N, E, n_s, n_t) \), where \( N \) is set of components, \( E \) is set of edges used to represent the dependency relations between components, and \( n_s \) and \( n_t \) is the start and termination node respectively. i.e., \( N = \{ n \}, E = \{ e \} \).

- \( n = (C_i, cpx_i, svt_yi) \), where \( C_i \) is the identifier of the \( i \)-th component, \( cpx_i \) is the dynamic complexity of component \( C_i \), and \( svt_yi \) is the severity level of the results caused by the failure of component \( C_i \).

- \( e = (t_{ij}, p_{ij}) \), where \( t_{ij} \) is the identifier of the transition from component \( C_i \) to \( C_j \), hence \( C_i \) and \( C_j \) is the source and target component respectively, \( p_{ij} \) is the probability of transition \( t_{ij} \) being executed, and it is estimated from the percentage of the number of messages that the source component \( C_i \) sends to the target component \( C_j \) to the total number of messages that the source component \( C_i \) sends to all other components in the architecture as shown in (4)

\[
p_{ij} = \frac{\text{Interact}(C_i, C_j)}{\sum_{k=1}^{n} \text{Interact}(C_i, C_k)} \quad (4)
\]

To assess a risk value for a specified scenario, we construct a CRDG model from the sequence diagram which depicts this scenario according to definition 8. Similarly, to assess a risk value for a subsystem or for a system, we merge all CRDGs at scenario level into a new CRDG model at subsystem level or at system level according to their weight factors.

The proposed risk aggregation algorithm can assess scenarios or architecture risk factors depending on whose input is a CRDG model at scenario level or at subsystem level. The algorithm expands all branches of a CRDG starting from the start node and terminating at the end node. All branches represent logical “or” paths. The depth of each path represents a sequential execution of components. Hence, the scenario or architecture risk factor is obtained from aggregating the risk factors of all paths. Further, the path risk factor is obtained from aggregating the risk factors of components along this path. Assume that a CRDG model can be divided into \( m \) different paths, and the length of the \( n \)-th path is \( L_n \).

Therefore, the risk factor for the scenario or architecture represented by that CRDG is given by:

\[
hrf = 1 - \sum_{n=1}^{m} \prod_{i=1}^{L_n} (1 - hrf_i) \prod_{j=1}^{L_n-1} p_{ij}, \quad j = i + 1 \quad (5)
\]

where \( p_{ij} \) is the probability of the transition from the \( i \)-th component to the \( j \)-th one, and \( hrf_i \) is a normalized risk value of the \( i \)-th component.

Due to the probabilistic nature of the dependency graph, several loops might exist in traversing the graph. These loops can lead to a deadlock. To evaluate the risk value in case of loops, we provide a maximum length based method to remove the loops in all execution paths. This method gives a threshold of the maximum length for all paths according to project experience. In the following texts, we depict the removal process in detail by an example. \( a(bc)^{*}d(ef)^{*}g \) is an infinite execution path due to the presence of two loops. Assume that the threshold of the maximum length is 12.

1) We remove all loops in this path and get a new path \( a(bc)^{1}d(ef)^{1}g \).

2) We traverse path \( a(bc)^{*}d(ef)^{*}g \) from left to right. Once located at some loop, try adding one to the numbers of the current loop, if the length of the path after change is not more than the threshold, the above operation is valid and go on traversing; otherwise the operation is cancelled and the traverse is finished.

3) If the length of current path is still not more than the threshold, go back to step (2). In case of the first traverse for the example path, \( a(bc)^{1}d(ef)^{1}g \) is changed into \( a(bc)^{1}d(ef)^{1}g \) firstly, and then \( a(bc)^{2}d(ef)^{2} \) will be changed into \( a(bc)^{2}d(ef)^{2} \). In case of the second traverse, we try changing

\[
\text{...}
\]
A Methodology for Software Reliability Risk Assessment
Xu Jian, Yan Han, Li Qianmu
Journal of Convergence Information Technology, Volume 6, Number 4. April 2011

a(bc)^2d(ef)^2 into a(bc)^3d(ef)^2, however the length of a(bc)^3d(ef)^2 is 13, more than the threshold 12, the changing operation is cancelled. Hence the resulting of the removal process is a(bc)^2d(ef)^2.

We provide an approximate estimate of scenario or architecture risk factor when the application architecture has infinite path due to the presence of loops using the threshold-based method. The evaluated approximate risk value is less than the real one in case of loops, according to the formula (5). The input of the proposed algorithm is a CRDG model representing a certain scenario or system. The detailed steps of are as follows:

1) \( \forall n_i \in N, n_i.hrf = n_i.cpx \cdot n_i.svrtv \cdot \text{normalized}(n_i.hrf) = \frac{n_i.hrf}{\sum_{i=1}^{N} n_i.hrf} \)

2) \( \forall n_i, n_j \in N, \text{if } \exists e_1, e_2 \in E \text{ satisfying } n_i \xrightarrow{e_1} n_j \text{ and } n_j \xrightarrow{e_2} n_i \text{, delete edge } e_1 \text{ and } e_2 \text{ and add a new edge } e, n_i \xrightarrow{e} n_j \text{ and } e.p_0 = e_1.p_0 + e_2.p_0 \)

3) To expand all branches of the CRDG starting from the start node \( n_s \) and terminating at the end node \( n_t \), and delete the node \( n_s \) and \( n_t \) in each path. Suppose that there is \( m \) paths, denoted by \( L[1], L[2], \ldots, L[m] \). We adopt a maximum length based method to remove the loops in the above paths.

4) \( R_{graph} = 0; \)
   For \( i=1,i<=m,i++ \) {\n     \( R_{graph} = PathRisk(L[i]); \text{ //call procedure } PathRisk \)
   }\n   \( R_{graph} = 1-R_{graph}; \)
   procedure PathRisk\n   double PathRisk(Path path){\n     double riskValue = 0;
     if path.length > 2{\n       Path subPath = path-path[1];
       riskValue = (1-path[1].hrf)*p_{12}*PathRisk(subPath);
     }else if path.length = 2{\n       riskValue = (1-path[1].hrf)*p_{12}*(1-path[2].hrf);
     }\n     Return riskValue;
   }

3. Case study

In order to compare with the simulation model based method in [7], we have studied the same case, a pacemaker device [6], to illustrate how the proposed methodology works. The pacemaker is a critical real-time application. An error in the software operation of the device can cause loss of the patient’s life. We use the UML notions to model the pacemaker and conduct risk analysis using the UML artifacts.

3.1. System description and architecture modeling

A cardiac pacemaker is an implanted device that assists cardiac functions when the underlying pathologies make the intrinsic heartbeats low. We do not consider the heart and programmer as a part of our software system, but consider them as external actors. Figure 3 shows the pacemaker architecture model. It consists of the following components:

- Reed_Switch (RS). A magnetically activated switch that must be closed before programming the device. The switch is used to avoid accidental programming by electric noise.
- Coil_Driver (CD). Receives/sends pulses from/to the device programmer. These pulses are counted and then interpreted as a bit of value zero or one. These bits are then grouped into bytes and sent to the communication gnome.
- Communication_Gnome (CG). Receives bytes from the coil driver, verifies these bytes as commands, and sends the commands to the Ventricular and Atrial models.
Ventricular_Model (VT) and Atrial_Model (AR). These two components are similar in operation. They both could pace the heart and/or sense heartbeats.

The pacemaker runs in either a programming mode or in one of five operational modes. During programming, the programmer specifies the type of the operation mode in which the device will work. The operation mode depends on whether the Atrium (A), Ventricle (V), or both are being monitored or paced. The programmer also specifies whether the pacing is inhibit (I), triggered (T), or dual (D). The use case diagram of the pacemaker application is given in Figure 4. It presents the six use cases and the two actors: doctor programmer and patient’s heart. Each use case is realized by at least one sequence diagram (i.e., scenario).

Domain experts determine probabilities of occurrence of use cases and the scenarios within each use case. For the pacemaker example, according to [7], the inhibit modes are more frequently used than the triggered mode. Also, the programming mode is executed significantly less frequently than the regular usage of the pacemaker in any of its operational modes. Hence, we assume the following scenarios profile: Programming = 0.01, AVI = 0.29, AAI =0.20, VVI = 0.15, AAT= 0.15, VVT = 0.20. Figure 5 shows a sequence diagram of a scenario from the AVI use case in which the VT senses the heart and the AR paces the heart when a heart beat is not sensed. As in all scenarios, a refractory period is then in effect after every pace. For the pacemaker example described here, only one scenario is available for each use case.
3.2. Component risk factor

For the AVI scenario example, we conduct risk assessment for each component in this scenario by using the component risk analysis process described in Figure 2.

First, we obtain relation matrix \( M_g = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \) and reachability matrix \( M_r = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \) from the system class diagram according to the definition 2 and definition 3 respectively, where rows and columns are indexed by components according to the order: RS, CG, CD, AR and VT. Next, to evaluate complexity factors of components from the AVI scenario, we extract local state charts and map them into the corresponding control flow graphs. As an illustration, the control flow graph of the component AR and VT in the AVI scenario is presented in part (a) and part (b) of Figure 6 respectively. The dynamic complexity of this graph is evaluated using (1). Last, we calculate component risk factors using (3). The detailed results of components risk assessment in the AVI scenario are also shown in Table 1.

![Figure 6](image.png)

Figure 6. Part (a) is the control flow graph of the AR component and part (b) is the one of the VT component in the AVI scenario

<table>
<thead>
<tr>
<th>Component</th>
<th>CG</th>
<th>AR</th>
<th>VT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Severity Factor</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Complexity Factor</td>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Risk Factor</td>
<td>2</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>Normalized Risk Factor</td>
<td>0.056</td>
<td>0.5</td>
<td>0.444</td>
</tr>
</tbody>
</table>

Risk factors of all components involved in the pacemaker scenarios are given in Table 2. We conclude that the risk factor of the component AR and VT are significantly higher than those for other components in more than one scenario. This is due to the fact that those two components are active and executing most of the time.

![Table 1](table1.png)

Table 1. The results of component risk analysis in the AVI scenario

<table>
<thead>
<tr>
<th>Scenario Name</th>
<th>Component Risk Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming</td>
<td>RS</td>
</tr>
<tr>
<td>(0.01)</td>
<td>0</td>
</tr>
<tr>
<td>AVI(0.29)</td>
<td>2</td>
</tr>
<tr>
<td>AAI(0.20)</td>
<td>2</td>
</tr>
<tr>
<td>AAT(0.15)</td>
<td>0</td>
</tr>
<tr>
<td>VVI(0.15)</td>
<td>2</td>
</tr>
<tr>
<td>VVT(0.20)</td>
<td>2</td>
</tr>
<tr>
<td>architecture level</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 2. Risk factors for the pacemaker components

In order to validate the proposed component risk analysis method, we compare the predicted results derived from our method with those derived from the simulation model built in [7] for the same case study. Table 3 shows the comparison between the predicted system level risk factors for the software components of Pacemaker example, obtained from our proposed methodology and the results from the.
A Methodology for Software Reliability Risk Assessment
Xu Jian, Yan Han, Li Qianmu
Journal of Convergence Information Technology, Volume 6, Number 4. April 2011

All risk factors are the normalized ones. The normalized risk factor of a component is obtained by normalizing its risk factor with respect to the sum of risk factors for all components at system level. The results strongly correlate with each other and the correlation factor is 0.9943.

<table>
<thead>
<tr>
<th>Component</th>
<th>RS</th>
<th>CD</th>
<th>CG</th>
<th>AR</th>
<th>VT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Model Result</td>
<td>0.00286</td>
<td>0.01690</td>
<td>0.01300</td>
<td>0.49406</td>
<td>0.47578</td>
</tr>
<tr>
<td>Proposed Model Result</td>
<td>0</td>
<td>0.00188</td>
<td>0.00759</td>
<td>0.43259</td>
<td>0.48967</td>
</tr>
<tr>
<td>Correlation=0.9943(p=0.0005&lt;0.05)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.3. Scenarios or subsystem risk factor

Using the algorithm and the graph to aggregate risk factors is useful for analyzing the risks in complex systems. The functionalities of such systems are usually depicted by several scenarios. Each scenario is composed of a number of components. We can apply the proposed methodology to scenarios, which have their own CDRGs, and obtain a risk factor for a scenario using risk factors of its individual components.

First, we develop CRDGs model for the pacemaker with six scenarios (Programming, AVI, VVI, VVT, AAI, and AAT). For the AVI scenario example, Figure 7(a) shows a CRDG model representing this scenario. Suppose that the threshold of the maximum length for all paths is five. Taking this CRDG model as input of the proposed risk aggregation methodology in section 2.4, we obtain the risk factor of 0.9360 for this scenario. Similarly, we can construct CRDGs for other five scenarios and evaluate scenarios risk factors. Due to space limitations, we decide not to describe any longer. The results of scenarios risk analysis are shown in Tab.4. Several observations are made from Table 4. First, all scenarios from the operational mode have lower risk factors than the programming scenario. Although the programming scenario has a high risk factor, it is just used to set the mode of the pacemaker. The failure of this scenario only results in failing to program the pacemaker. Next, it is obvious that the AVI scenario has the largest scenario risk factor (0.9360) among the operational scenarios (AVI, AAI, VVI, AAT, and VVT) and a relative high execution probability, which means that this is the most critical scenario in the pacemaker case study.

Next, we construct an architecture-level CRDG model by merging the above CRDGs, as shown in Figure 7(b), and again use the proposed risk aggregation algorithm to obtain the risk factor of 0.7099 for the pacemaker.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Programming</th>
<th>AVI</th>
<th>AAI</th>
<th>AAT</th>
<th>VVI</th>
<th>VVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Risk Factor</td>
<td>0.9570</td>
<td>0.9360</td>
<td>0.6014</td>
<td>0.6129</td>
<td>0.6014</td>
<td>0.6129</td>
</tr>
</tbody>
</table>

3.4. Sensitivity Analysis

The reliability of a component-based software system is often higher through the improvement of some components in the system. Therefore, considering such a system we are often interested to know
which component is more important than others. Thus, the improvement of that important component will increase the system reliability more than others. Sensitivity analysis gives an approach to analyze the relative importance of components in determining which of the components affects the reliability of the system most. This type of analysis is also useful to study the effect of replacing one component with another of improved quality, i.e., lower risk factor.

Now we already have the individual component risk factors and the overall system or scenario risk factors. Next, we will analyze the sensitivity of the estimated system risk as a function of variations in the component risk factors to make sure which component can greatly reduce the system risk factor after improving quality and also to identify the critical point of system risk. Supposed that the variations of component risk factor do not result in the variations of transition probability among components, and we vary the risk factor of one component at a time.

Figure 8 illustrates the variation of the risk factor of the AVI scenario as a function of changes in risk factors of the active components in that scenario. The variation of the risk factor of the AR component introduces the biggest variation of this scenario risk factor (from 0.7513 to 1). This is the case because the AR component is the most active component in this scenario. On the other side, the variations of the risk factor of the VT and CG components have smaller effect on the variation the AVI scenario risk factor.

Figure 8. Sensitivity of the AVI scenario risk factor to the risk factors of the components

The variation of the overall system risk factor as a function of components risk factors is presented in Figure 9. It is clear that the risk factors of components CG, VT, and AR are most likely to affect the overall system risk. This is due to the fact that these components are active in scenarios that have high execution probabilities. In addition, the variation of the risk factors of components that are active only in the programming scenario (i.e., RS and CD) has almost no influence on the variation of the overall system risk factor because the execution probability of the programming scenario is one order of magnitude lower that the execution probabilities of other scenarios.

Figure 9. Sensitivity of the system risk factor to the risk factors of the components

4. Related work

Our work is in the field of software architecture assessment, and we mainly focus on the models and methods of software architecture reliability risk assessment. In the sequel, we summarize research work related to our work.
Nowadays, there are many risk management approaches proposed in the literature[5-12]. In [5], they developed a framework for model-based risk assessment of security critical systems. The framework integrates aspects of risk assessment methods by using UML. UML profile is refined into specialized UML profile in order to support model-based risk assessment process of CORAS project. This framework assesses risks at the functionality level by using specialized UML profile to identify, analyze and treat risks. But, it does not assess risks in the fine-grain level which has effect to system function. In [6] and [7], they proposed a methodology for analyzing risk in an architecture level which is performed during an early development phases. This methodology concerns risk factors of components by using a number of states and transitions in state machine and a number of messages in sequence diagram to consider complexity of component and their interaction respectively. However, this methodology does not concern the cause of changing state within a state machine which actually represents object behaviors. In [8], they proposed a methodology for risk assessment based on the UML specifications such as use cases and sequence diagrams that can be used in the early phases of the software life cycle. They construct a Markov model for estimation of each scenario risk factor and derive closed form exact solutions for the scenarios, use cases, and overall system risk factors. Hence, the fact that the risk assessment is entirely based on the analytical methods enables more effective risk assessment and sensitivity analysis. In [9], they proposed an approach of software risk assessment by expanding the work of [8] during analysis and design phase. The new idea introduced in this research is the risk assessment based on object behavior represented by a state machine. Test case generation from state machine is used to compute risk and identify risk factors. The obtained result can be used to manage and control risk at a fine-grain level.

5. Conclusion

This paper presents a risk analysis model and a risk aggregation algorithm for risk assessment of software architectures. The methodology is suitable for architectures whose analysis is based on UML artifacts. The proposed methodology has the following benefits: it is applicable at the architectural level and hence it is possible to identify critical components and scenarios early in the life cycle. The methodology uses ripple effect based method to analyze the severity of a failure, which relies a few on the subjective experience of assessors.

6. Acknowledgement

This work is funded in part by a grant from Specialized Research Fund for the Doctoral Program of Higher Education of China under grant No.20093219120024, and by a grant from NUST Research Funding under Grant No. 2010GJPY056.

7. References