A Method for Selecting a Model to Estimate Software Reliability at the Design Phase of Component-Based Real-Time System Development

Mohit Garg, Richard Lai*
La Trobe University, Melbourne, Victoria, Australia.

* Corresponding author. Email: r.lai@latrobe.edu.au
Manuscript submitted February 10, 2018; accepted April 8, 2018.
doi: 10.17706/jsw.13.6.317-334

Abstract: The rapid growth of software-based functionalities has not only increased the complexity of Component-Based Real-Time Systems (CBRTS) but also made it difficult for designers to quantify its reliability. Architecture-Based Software Reliability Models (ABSRMs) are useful for estimating the architectural reliability of component-based systems so that the behavior of its software components can be examined at different development phases. However, due to the availability of a number of models spread over different ABSRM categories, it is difficult to know which one is suitable for a particular system. Traditional model selection methods have two limitations: (i) they can only be applied to identify a model from a specific ABSRM category; and (ii) they omit a well-defined and verified mechanism to identify the ABSRMs which are capable of addressing the reliability requirements at an early development phase. Reliability estimation at the design phase enables developers to analyze the impact of the software component's reliability and its interactions on a CBRTS, study the sensitivity of CBRTS reliability to the reliabilities of its software components and interfaces and guide the process of identifying potential troublesome software components which require more attention during the integration exercises. In this paper, we describe a method for selecting an ABSRM to estimate software reliability at the design phase of a CBRTS development. To demonstrate the usefulness of our proposed method, we apply it to an automotive Automatic Parking System.

Key words: Architecture-based model, software reliability estimation, component-based system, software reliability model selection.

1. Introduction

A Component-Based Real-Time System (CBRTS) consists of many third-party developed black-box type software components working together by interacting and exchanging data with each other spontaneously [8], [20], [27], [28], [41]. Rapidly increasing the need to accommodate newly developed software-based functionalities with more emphasis on reuse has not only made CBRTS significantly complex, but has made it difficult for designers to evaluate different quality-related aspects such as performability and modifiability [2], [9]. In order to examine these aspects from the design stage to implementation and final deployment, designers need to have a good amount of confidence that CBRTS and its software components have a certain level of reliability before such a system is constructed [13]. However, estimating the architectural reliability of a CBRTS is difficult as input space and test cases may be extremely large, thus

making it infeasible to extensively test each component, given the resource limitations [12], [22]. This problem becomes more challenging in the case where failure data is unavailable during early phases of software development [7], [39].

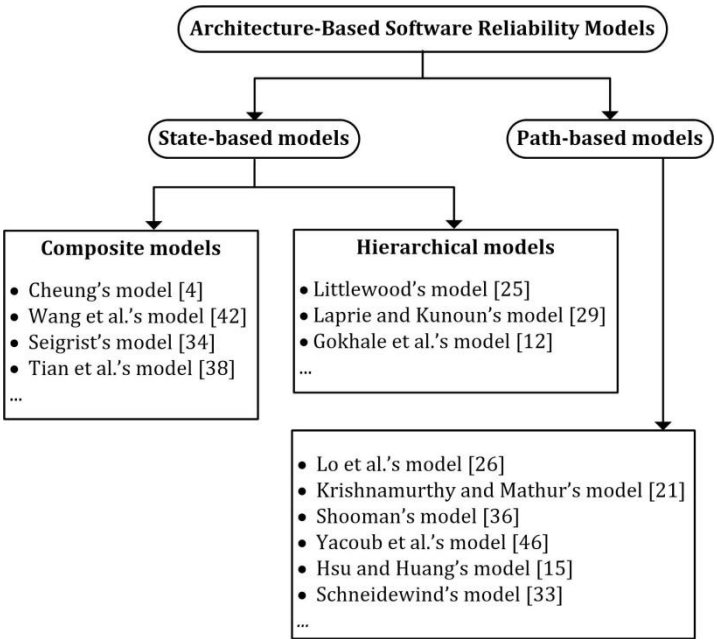

Fig. 1. ABSRMs developed using markov processes.

Software reliability estimation using a component-based system's architecture is achieved in two stages [15], [9]. In the first stage, the architectural description of the system and state transition probabilities for connecting software components are adaptively integrated using a variation of Markov processes, such as discrete time Markov chain [4], [40], continuous time Markov chain [12], or semi–Markov process [25], to compute the reliability of software components. During the second stage, architecture-based tools use the reliability values of the software components and either the state maintained by the software components or testing path information to estimate the software reliability of a component-based system.

An engineering tool commonly used for estimating the architectural reliability of component-based systems is called the Architecture-Based Software Reliability Model (ABSRM). Figure 1 show how architectural models, developed using different Markov processes, can be divided into two categories viz. state-based models and path-based models. In the literature, additive type models are also proposed which estimate the architectural reliability of a system as the sum of the reliabilities of its software components assuming software component failure intensities can be modelled by a Non-Homogeneous Poisson Process [5], [45]. Additive models are not suitable for component-based systems, whose software components operate in a real-time environment, as these models do not explicitly consider system architecture as the key criteria for architectural reliability estimation [35]. Therefore, for the architectural reliability estimation of CBRTSs, the focus is only on the state-based and path-based models.

State-based models are formed by combining the architecture of the system and the failure behavior of its software components. State-based models are further divided into composite models and hierarchical models. In most composite models, the architecture of the system is modeled using a discrete time Markov chain and the system's reliability is computed using a matrix of state transitions whereas, the hierarchical models account for the variance of the number of visits to each software component, and thus provide estimations closer to those provided by composite models. In path-based models, the system architecture and the failure behavior of software components is described using a path-based approach and the system's

reliability is computed using the approximation of testing path reliabilities where the reliability of a testing path relies on the sequence in which the software components are executed and the reliability estimates of each software component in the testing path [15], [26].

The rich literature on ABSRMs provides engineers with the knowledge to analyze the impact of the software component's reliability and its interactions on a system; study the sensitivity of the system's reliability to the reliabilities of its software components and interfaces; and guide the process of identifying potential troublesome software components which require more attention during the testing and integration exercises [9], [13], [15], [23], [24]. But when it comes to applying an ABSRM, it becomes difficult to select a suitable model as models that are good may not always be the best choice for a particular system. Traditional ABSRM selection methods have two limitations: (i) they can only be applied to identify a model from a specific ABSRM category [11], [31]; and (ii) they omit a well-defined and verified mechanism to identify ABSRMs which are capable of addressing the reliability requirements at an early development phase [1].

In [6], we reported a preliminary version of our ABSRM selection method for a CBRTS. The method is based on model selection criteria discussed in an existing algorithm [1] and the use of a degree of influence weighing mechanism. Since the conference paper publication, we have improved our method. This paper aims to demonstrate the process of solving the model selection problem at the design phase of a CBRTS development by: (i) presenting an improved version of our ABSRM selection method; and (ii) discussing the application of method to an automotive Automatic Parking System in order to demonstrate the usefulness of our method when used in a CBRTS. Our method is different from traditional architecture-based model selection approaches in two ways. In the pre-selection phase, we describe that the intensity of model selection problem can be reduced by identifying an applicable category of ABSRMs, that is, composite type state-based models, hierarchical type state-based models or path-based models, by analyzing distinct characteristics of models in each category with attributes of specific model selection criteria which are not bound to any specific category of ABSRMs. In the selection phase, the likelihood of obtaining more than one suitable model from the applicable category of ABSRMs is minimized by assigning degree of influence weights to each attribute of the different model selection criteria. The selected model can then be applied for the purposes of estimating reliability of CBRTS at an early development phase and identification of those software components that are critical from a reliability perspective.

2. Related Work

Previous research efforts have been focused on: (i) the classification of the existing ABSRMs into three categories viz. state-based [4], [12], [25], [29], [34], [38], [42], path-based [15], [21], [26], [33], [36], [46] and additive [5], [45]; or (ii) describing different mathematical and statistical techniques such as u plot, y plot likelihood ratio, etc. for the selection of software reliability models at different development phases. These model classification schemes and techniques only help in using the architectural models instead of ranking the models.

In the literature, very few research attempts are reported to select an architectural model. Authors in [11] recommended that architectural model selection should be based on the software artefacts that are available for estimating the input parameters of a model at the design phase. However, this approach can only be applied to find a state-based model (that is either a composite [4], [34], [38], [42] or a hierarchical model [12], [25], [29]), and no suggestions were given about the usefulness of this approach for path-based models.

Authors in [31] proposed SREPT (Software Reliability Estimation and Prediction Tool), in which architectural models can be incorporated for predicting software reliability at the design phase of a

software product development. The tool supports architecture-based software reliability evaluation using discrete-event simulation to capture the architecture of a system and utilizes the metrics data directly to represent the model parameters. Similar to Gokhale and Trivedi's approach [11], this tool is applicable to only the state-based models and not path-based models, thus its ability to satisfactorily select an architectural model for large-scaled software systems such as CBRTS with many transient states is limited.

In another attempt, authors in [1] proposed to identify candidate models using model selection criteria and then ranked the shortlisted models using criterion and applicability weights. Although inspiring, the applicability of this approach to find an ABSRM for a CBRTS is limited as it lacks a well-defined and verified technique to assign and compute these weights and thereby increases the possibility of obtaining more than one model with similar results. Hence, the development of a robust ABSRM selection method becomes infeasible to characterize the expected behaviour of CBRTS and its software components at different development phases and during run-time, using an ABSRM.

3. An Overview of the Method

The ABSRM selection method described in this paper comprises five steps spread over two phases, namely the pre-selection phase and selection phase. The pre-selection phase focuses on the reliability estimation of individual software components that constitute a CBRTS as these values are imperative to enable the use of an ABSRM [9], [25], [42]. In this phase, we first define the architecture of the CBRTS using appropriate architectural styles to address the encapsulated behaviours and constraints. Second, state transition probabilities for each software component are determined to construct a state machine for the CBRTS. Then, the architectural and state transition probability information is used to estimate the parameters of a reliability function in order to compute the reliability of the software components of a CBRTS. During the selection phase, an architectural model for estimating the system-level reliability of a CBRTS is selected. For this, we first identify an applicable category of ABSRMs using specific model selection criteria [1] and then apply degree of influence weights [37], [43], [44] to obtain the best model. Fig. 2 depicts an overview of our method and the sequence in which the steps are performed.

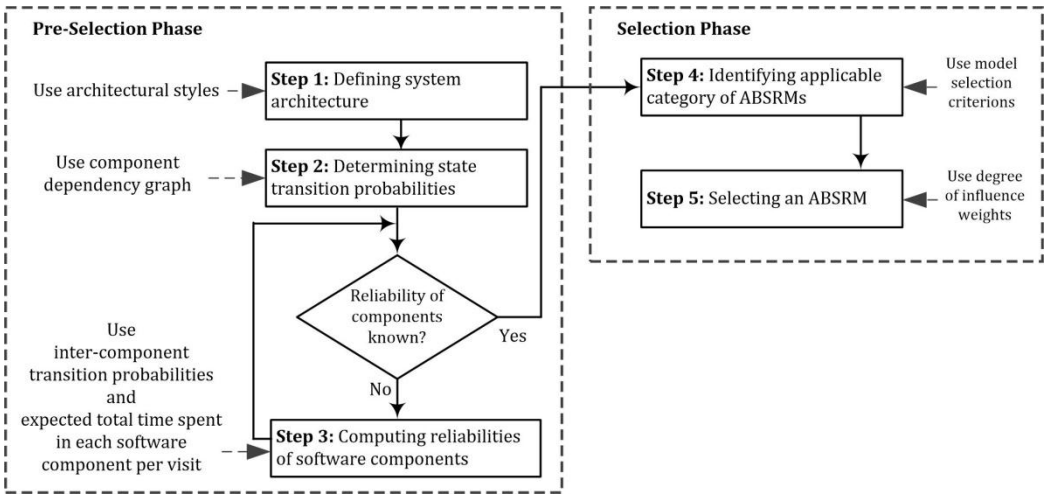

Fig. 2. Overview of the ABSRM selection method.

3.1 Pre-selection Phase

Step 1: Defining system *architecture*

In the first step of the pre-selection phase, the CBRTS is described using an architectural style that has well-defined characteristics to analyse a component-based system's architecture and develop a state machine based on the behaviour encapsulated. Different architectural styles, such as batch-sequential, parallel/pipe-filter, fault-tolerant, call-and-return, sequence, branch and loop can be used to define the architecture of a system [15], [46]. In batch-sequential and sequence styles, the components execute in sequence and are independent of each other. Parallel/pipe-filter style is useful in scenarios where components commonly run simultaneously to fulfil a task. In fault-tolerant style, components compensate for the failure of the primary component. In call-and-return style, a calling component may request services provided by the called components. Branch style is useful for representing a system consisting of several conditional statements (i.e., if-then-else or switch-case). Loop style is useful for representing a system having iterative statements (i.e., do-while or for-loop). Of these architectural styles, the batch-sequential style is most commonly used due to its ease of representing the organization of a system, that is, the transitions from one software component to another, in sequence. One of the instances of this style is modelled in Fig. 3 where C_1, $C_2 ... C_N$ are independent software components of a CBRTS. Since only one software component is executed at a time, an incomplete execution of a software component will not proceed to the next software component. Without loss of generality, it is taken that the execution begins with a single software component and terminates on a single software component.

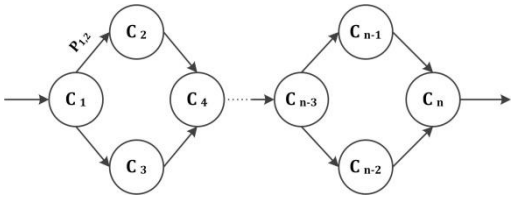

Fig. 3. Batch-sequential style.

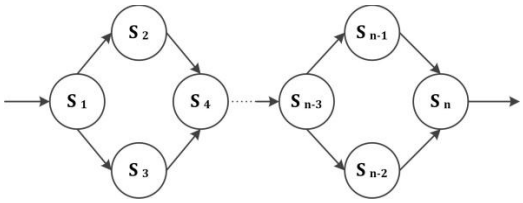

Fig. 4. State transition diagram.

For a CBRTS, the objective is to construct a state machine which is capable of addressing its heterogeneous behaviour. In the past, many computer-aided software engineering tools or modelling languages [9], [13], [46] have been proposed to construct a graphical representation for a component-based software system, such as directed graph, control flow graph, component dependency graph, etc. While nodes in a directed graph and control flow graph perform certain calculations, outputs, or error handlings [5], a component dependency graph incorporates component and interaction usage probabilities, as well as their estimated reliabilities [46]. Contrary to directed graph and control flow graph, where Markov chains show the system states and transitions from one state to another, component dependency graphs are useful in representing the system in terms of the composition of software components. According to the authors in [42], a state is a set of circumstances characterizing a system at a given condition, where condition is an instance of a set of components from receiving the control to the release of the control and a circumstance is an event that activates one of the components retaining the control. For discrete time Markov chains, both state space and parameter space are discrete (finite or countably infinite). Discrete time Markov chains can have an irreducible state or an absorbing state. In irreducible state, each state can be reached from other state in a finite number of steps whereas in an absorbing state there is at least one state with no outgoing transition. Figure 4 depicts the control structure of the system developed using a directed graph where the state of every software component is represented by S_i and a directed branch (S_i, S_j) represents a possible transfer of control from S_i to S_j. A system with n components (refer to Figure 3) will have n mapping states where state S_i has component C_i activated, $1 \leq i \leq n$. The state S_1 is the initial state and state S_n is the absorbing state of discrete time Markov chains.

Step 2: Determining state transition probabilities

In the next step, state transition probabilities of connecting software components are determined. The inter-component transition probabilities are generally estimated using a random input generator or could also be obtained from the occurrence probabilities of various scenarios based on the operational profile of the system, as illustrated in [46]. For discrete time Markov chains based system, the transitions occur either at discrete-time intervals or at discrete events, and the transition probabilities exhibit a discrete distribution. The inter-component transition probabilities are described by one-step transition probability matrix P of the discrete time Markov chain [40] such that the transition probability p_{ij} from state S_i to S_j (refer to Figure 4) of the directed branch (S_i, S_j) is $P = [p_{ij}]$. In [10], the authors described that the transition probability matrix P of an absorbing discrete time Markov chains can be partitioned as:

$$P = \begin{bmatrix} Q & C \\ 0 & 1 \end{bmatrix}$$

where Q (i.e., infinitesimal generator matrix of Markov chains) is a *(N - 1)* by *(N – 1)* sub-stochastic matrix (with at least one row sum < 1) describing the probabilities of transition only between transient states, N is the number of states, C is the column vector and *0* is a row vector of *(n - 1)* zeros.

In [46], [47], a scenario is described as a set of component interactions triggered by a specific input stimulus. The dynamic behaviour of a CBRTS is specified using a set of scenarios. Each scenario depicts an event at run-time. The interaction between components in the system, such as a system depicted in Figure 3, described using batch-sequential architectural style can be analysed using n scenarios, each triggered by an event. Authors in [46] described that for a scenario R_k from the set of the application scenarios R, $R_k \epsilon R$ where $k = 1...|R|$, represents a sequence of component interactions. Each scenario is assigned the probability of execution. pR_k is the execution frequency of scenario k with respect to all other scenarios. The execution probabilities of all scenarios R_k, $k = 1...|R|$, should sum to unity. In [46], the authors gave an equation to compute the inter-component transition probabilities from the number of interactions between two components:

$$p_{ij} = \sum_{k=1}^{|R|} pR_k * \left(\frac{|interact\ (S_i,S_j)|}{|interact\ (S_i,S_j)|_{i=1...|N|}} \right)_{S_i,S_j,S_1 in\ R_k} \quad (1)$$

where $|N|$ is the number of components, and $|interact(S_i,S_j)|$ is the number of times S_i interacts with S_j in scenario R_k.

Step 3: Computing reliabilities of software components

In this step, reliabilities of individual software components are computed. At architectural level, the failure behaviour can be superimposed on to the solution of the architectural model to obtain reliability predictions. In the cases where only the information regarding the architecture is available but the failure behaviour of the components is unknown, the failure behaviour of software components can be obtained by conducting coverage testing on each component [12]. Failure behaviour of software components can also be obtained by coverage measurements made on the basis of possibilities of execution of the application and a static complexity metric-based approach to predicting the number of fault-related items in components, such as fault density in lines of code, complexity of code in terms of control statements and the number of interferences with other components [12], [13], [48]. In [12], an inter-component dependency approach-based reliability function is described for computing the reliabilities of software components in a component-based system. This function employs the statement coverage approach to compute the expected total time $V_i t_i$ spent on a component i per visit. The statement coverage approach

uses the trace data produced by the coverage analysis tool to determine the expected number of times V_i the component i visits state j, and the expected time t_i spent on the software component i per visit. The expected number of visits to component i, denoted by V_i, is computed as shown in [50] by solving the following system of linear equations:

$$V_i = q_i + \sum_{j=1}^{n} V_j p_{ji} \tag{2}$$

where q_i denotes the initial state probability vector for component i that is the probability of first calling component i, p_{ji} is the probability of calling component j after executing component i, and V_j is the expected number of visits to component j.

The expected total time spent on each software component per visit information is then used with the failure behaviour of software components, described by a time-dependent failure intensity function [9], to compute the reliabilities of components. Time-dependent failure intensity function refers to the variation in rate at which faults occur during the testing of a component and it is proportional to the product of the time-dependent failure occurrence rate per fault and the expected number of remaining faults [49].The time-dependent failure intensity function $\lambda(t)$ in terms of the time spent on each component visit V_i is computed using the equation given in [49]:

$$\lambda(t) = [a - m(t)] \frac{\acute{c}(t)}{1 - c(t)} \tag{3}$$

where a is the expected number of remaining faults, $m(t)$ is the expected number of faults detected by time t, $\acute{c}(t)$ is the rate at which the remaining potential faults are covered during testing, and $1\text{-}c(t)$ denotes the fractional population of uncovered potential faults.

The component reliabilities are estimated considering the time-dependent failure rates $\lambda_i(t)$ and the utilization of the components through cumulative expected time spent in the component per execution $V_i t_i$. The reliability of a component is computed using the equation given in [12]:

$$R_i = e^{-\int_0^{V_i t_i} \lambda_i(t) dt} \tag{4}$$

where R_i denotes the reliability of a software component i, $\lambda_i(t)$ is the time-dependent failure intensity and $V_i t_i$ is the expected total time spent in the component i per visit. From a reliability point of view, the time to failure is exponentially distributed for a component-based system.

3.2 Selection Phase

Step 4: Identifying applicable category *of ABSRMs*

In this step, we identify an applicable category of ABSRMs from composite type state-based models, hierarchical type state-based models and path-based models. For this, we use the model selection criteria proposed by authors in [1]. Of the nine model selection criteria, four criteria can be applied to determine an applicable category of ABSRMs viz. input required by the model, the nature of the project, the validity of assumptions according to the data and the output desired by the user.

- Input required by the model criteria helps in verifying if the input parameters required by the architectural models can be estimated from the available CBRTS artefacts.
- The nature of the project criteria aids in identifying whether the models in a particular category of ABSRMs are capable of addressing: (i) the heterogeneous behavior of the CBRTS; (ii) the architectural

styles which show the transfer of control among software components; and (iii) the terminating nature of the CBRTS.

- The validity of assumptions according to the data criteria is useful for validating whether the assumptions of models in a particular category of ABSRMs suit the reliability-related requirements of the CBRTS.
- The output desired by the user criteria is helpful in checking if the models in a particular category of ABSRMs are able to satisfactorily provide the outputs specified by the CBRTS designers.

The above-mentioned criteria help in finding an applicable category of ABSRMs by analysing the distinguishing characteristics of each model category using the information provided by the CBRTS under study and the requirements set by its designers.

Table 1. Basic Assumptions of Architectural Models

Assumption	Description
System is component-based	System can be designed in a structured way so that composition or decomposition into its constituent software components is simple [16], [32], [36]. The exchange of control among software components can be related to functional structures or logical sequences, performing a certain function.
All software components are physically independent of each other	This assumption implies that each software component in a complex system can be independently designed, implemented, and tested [9], [13]. That is, software components have high cohesion and low coupling [16], [18], [32].
Modification of one software component has no impact on others	The functionalities embedded in a software component have little in common, and the integration of software components may require less effort due to the decreased inter-component dependency [7]. Thus, the faults in each software component are also physically independent from each other in different software components. Moreover, each detected fault can be easily isolated and fixed in a specified software component during fault detection and correction processes.
The transfer of control among software components can be described by a Markov process	The Markov process depends on the structure of the software system so that it is helpful in modelling the execution between the functional software components and branching characteristics [9].

Step 5: Selecting an *ABSRM*

1 *Initialization*: $VAL_{total} = 0$, $ATT_{total} = 0$, $CRIT_{total} = 0$
2 for each MSC
3 for each ATT_{crit}
4 assign DI_{weight} (0 to 5)
5 while $DI_{weight} \neq 0$ do
6 if ABSRM satisfies condition ATT_{crit};
7 VAL_{total} ++;
8 else
9 $VAL_{total} = VAL_{total}$;
10 endif
11 end while
12 $ATT_{total} = DI_{weight} \times VAL_{total}$;
13 end for
14 $CRIT_{total} = \sum ATT_{total}$;
15 end for

Fig. 5. Algorithm to compute total score of a model selection criterion

In the last step, a model from an applicable category of ABSRMs is selected which can be suitably applied to estimate the architectural reliability of a CBRTS. For this, we compute the total score of each model selection criterion for the candidate models and then rank the models based on their cumulative score. Each model selection criterion mentioned in the previous step has a different set of attributes. For example,

in Table 1, we have listed some of the basic assumptions commonly made for architectural models [13], [15]. Each of these assumptions can be termed as an attribute for the validity of assumptions according to the data criterion. Figure 5 shows an algorithm for computing the total score of a model selection criterion.

To identify a model, we first determine the influence of the attributes on the respective model selection criteria (*MSC*). The influence on an attribute of the model selection criteria can be determined in two ways: (i) through research and consultation with experts who are intimately familiar with the system; and/or (ii) based on the historical test data collected from a prior release of the system or another similar system [37], [43], [44]. Depending on the influence on the model selection criteria weights (DI_{weight}) are assigned to each attribute. The definitions of the weights which correspond to the different degrees of influence are: (i) 0 for no presence/influence; (ii) 1 for insignificant influence; (iii) 2 for moderate influence; (iv) 3 for average influence; (iv) 4 for significant influence; and (v) 5 for strong influence throughout. Once the degree of influence weights are assigned to all attributes (ATT_{crit}) in a model selection criterion, the next task is to analyse whether the candidate models belonging to the applicable category of ABSRMs satisfy the conditions of the attributes in a model selection criterion. For this, the value (VAL_{total}) one is assigned if: (i) the candidate model satisfies the condition of an attribute; and/or (ii) the attribute is not applicable to the candidate model, else zero. Then, the respective assigned values to the candidate model for an attribute and the degree of influence weight of the attribute are multiplied for each attribute and this number is summed (ATT_{total}) to obtain the total score ($CRIT_{total}$) for the candidate model for a particular model selection criterion.

Table 2. Computing Total Score for Candidate Models

Attribute	DI	Candidate model A	Candidate model B
X	…	X's DI x (1 or 0) = …	X's DI x (1 or 0) = …
Y	…	Y's DI x (1 or 0) = …	Y's DI x (1 or 0) = …
Total score of Z		…	…

Table 2 describes the process of computing total score of a model selection criterion for candidate models. In the table, *X* and *Y* are the attributes for a model selection criterion *Z*, *DI* is the degree of influence assigned to each attribute, and candidate models *A* and *B* are the models under study from an applicable category of ABSRMs. In the end, the total score of each model selection criterion for the candidate model are added to obtain the cumulative score. Similarly, the cumulative score for other candidate models is obtained. The model with the highest cumulative score is selected for architectural reliability estimation of a CBRTS.

4. An Application

The application of the ABSRM selection method is carried out on an automotive Automatic Parking System (APS). According to the models described in [14], [19], the main function of the software components in an APS is to find the optimal path for parking by measuring the distance between the car and its surrounding objects, and to drive the car automatically without the driver's intervention.

The specifications of the APS use ten software components for sensors, actuators, set point generators, and logic types as shown in Table 3. Three sensor software components C_1, C_3 and C_4 constantly monitor the vehicle's speed, the video scene, and the distances during automatic parking, and the four actuator software components C_6, C_7, C_8 and C_9 control the engine throttle, transmission mode (either forward or backward), steering direction, and brakes. Two logic software components C_2 and C_5 find the optimal parking path and execute the parking by using sensor and actuator software components. The set point generator software component C_{10} allows the driver to select the various automatic parking options. The execution of the software components begins with C_1 and terminates upon the execution of C_{10}. The

software components differ from each other in complexity, size, and the function they perform etc. and can be tested independently during the testing phase of software development. From the view point of the CBRTS designers, two outcomes are required: first, to compute the reliability of the APS based on its architecture, the time spent (response time) in each software component per visit and the reliability values of individual software components which take into account the variance of the number of visits of each software component; and second, to identify a software component that is critical from the reliability perspective. Next, we explain the applicability of the ABSRM selection method in obtaining a model for these purposes.

Table 3. Automatic Parking System Specification

Software component	Description
C_1	Camera (Start)
C_2	Path Finder
C_3	Speed Sensor
C_4	Distance Sensor
C_5	Parking Region Analyzer
C_6	Steering Controller
C_7	Brake Controller
C_8	Transmission Controller
C_9	Throttle Controller
C_{10}	Parking Executor (End)

3.3 Pre-Selection Phase

Step 1: Defining system architecture

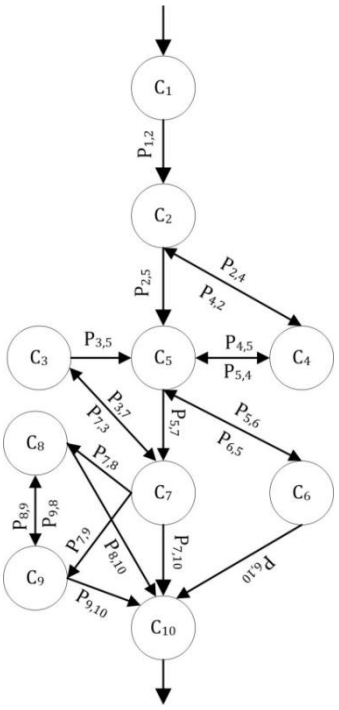

Fig. 6. Software architecture of automatic parking system.

In the first step of the pre-selection phase, we define the architecture of the APS in terms of the control structure of its software components. Figure 6 shows the APS architecture defined using a component

dependency graph [46]. In Fig. 6, there are some double ended arrows which are equivalent to two arrows in opposite directions and their probability values may be different. The initial state of the discrete time Markov chain is C_1 and the exit state or the absorbing state of the discrete time Markov chain is C_{10}. The architectural description also shows that the APS is a heterogeneous and terminating (single-input and single-output) system with sequential and branching architectural styles.

Step 2: Determining state transition probabilities

In the next step, state transition probabilities of software components in an APS are determined. The inter-component transition probabilities can be estimated based on the experimentation described in [10] or the probability of execution of each scenario described in [46] where the frequency of execution of each scenario is estimated relative to all other scenarios. We adopt Gokhale and Trivedi's method [10] in MATLAB for analysing the operational behaviour of the APS in terms of its workflow and software component interfaces for the estimation of the inter-component transition probabilities. We varied transitions such as $P_{3,5}$ from 0.10 to 0.90, while maintaining $P_{3,5} + P_{3,7} = 1.0$. We also restored $P_{3,5}$ back to its initial value, and varied $P_{4,2}$ from 1.0 to 0.90, maintaining $P_{4,2} + P_{4,5} = 1.0$. Table 4 presents the non-zero inter-component transition probabilities in the form of the transition probability matrix P for the APS architecture.

Table 4. Inter-component Transition Probabilities

$P_{1,2} = 1.00$			
$P_{2,4} = 0.68$	$P_{2,5} = 0.32$		
$P_{3,5} = 0.31$	$P_{3,7} = 0.69$		
$P_{4,2} = 0.36$	$P_{4,5} = 0.64$		
$P_{5,4} = 0.18$	$P_{5,6} = 0.51$	$P_{5,7} = 0.31$	
$P_{6,5} = 0.43$	$P_{6,10} = 0.57$		
$P_{7,3} = 0.11$	$P_{7,8} = 0.06$	$P_{7,9} = 0.35$	$P_{7,10} = 0.48$
$P_{8,9} = 0.23$	$P_{8,10} = 0.77$		
$P_{9,8} = 0.39$	$P_{9,10} = 0.61$		

Step 3: Computing reliabilities of software components

In this step, the reliability values of individual software components are determined using the inter-component transition probabilities and the expected total time spent on each software component per visit information. For the APS, we assume that each software component has a finite number of faults on the basis that every component executes as intended and produces the correct output and transfers control to the next component correctly. The failure behavior of the software components is therefore characterized using the time-dependent failure intensity function of the Gokhale et al.'s model [12].

The values of the expected number of times V_i the software component i visits state j computed using equation (2), the expected time t_i spent by the software component i per visit, the expected total time $V_i t_i$ spent on a software component i per visit, the model parameter a which is the total number of faults expected to be detected given infinite testing time and $c(t)$ which is complete test coverage, the time-dependent failure intensity function $\lambda_i(t)$, and the reliability R_i computed using equation (4) for each software component of APS are given in Table 5. The expected total time $V_i t_i$ spent on a software component i per visit and the time-dependent failure intensity function, $\lambda_i(t)$, for each software component are obtained using SAS Predictive Analytics software. Each calculation took between 1 and 3 milliseconds on a PC with a 2.3 GHz Intel Core i7 processor and 8 GB RAM. There may be a slight difference in the results in Table 5 and the results obtainable using a different computer, such as an automotive electronic control unit due to the reasons: (i) embedded systems have varied hardware configurations and system load requirements which may influence the component execution times; and (ii) the assumptions made by us in

this study: such as components having a finite number of faults may not be applicable to other APS designed for different models and makes of cars.

Table 5. Reliabilities of Software Components

Software component	Expected number of visits (V_i)	Expected execution time (t_i) in ms	$V_i t_i$	a	$c(t)$	$\lambda_i(t)$	R_i
C_1	1.0000	1.00	1.0000	218	0.9736	0.00146	0.956448
C_2	0.9077	1.62	1.4717	290	0.5321	0.00955	0.999765
C_3	0.9107	1.45	1.3254	231	0.4129	0.00174	0.946873
C_4	0.4184	1.40	0.5829	105	0.7276	0.00252	0.997941
C_5	0.3831	2.52	0.9669	305	0.8152	0.00271	0.982608
C_6	0.2510	1.26	0.3173	260	0.6321	0.00267	0.980764
C_7	0.6155	2.83	1.7433	312	0.6542	0.00167	0.949212
C_8	0.8737	1.50	1.3155	245	0.7965	0.00543	0.995611
C_9	1.3504	1.47	1.9784	286	0.3697	0.00429	0.999883
C_{10}	1.0000	1.00	1.0000	279	0.3175	0.00331	0.969578

4.2 Selection Phase

Step 4: Identifying applicable category of ABSRMs

In this step, we identify an appropriate category of ABSRMs from composite type state-based models, hierarchical type state-based models and path-based models, which can be suitably applied for system-level architectural reliability estimation and identification of the critical software components. For this, we analyse the information given in the APS application and the requirements set by its designers with the four model selection criteria [1] viz. input required by the model, the nature of the project, validity of assumptions according to the data and output desired by the user.

- *Input required by the model:* Models in all three categories of ABSRMs require reliability values of individual software components and/or inter-component-transition probabilities as input parameters for estimating the parameters of the system-level architectural reliability function [9], [10], [13], [21], [42], [46]. In the pre-selection phase, we have shown how these input parameters are obtainable using different artefacts of the APS. For APS, the inter-component transition probabilities are obtainable by analysing its software architecture, that is, each component can be examined to find out which components it can or cannot interact with. The reliabilities of individual software components of APS can be obtained experimentally using a reliability function given in [12]. Since most architectural models assume reliability values of software components and their inter-component-transition probabilities are known irrespective of how these input parameters were obtained, the state-based models (including composite and hierarchical models) as well as path-based models can be applied to address the needs of APS. Hence, this criterion is not of much help in identifying an applicable category of ABSRM.
- *Nature of the project:* From the architectural description it is evident that the APS: (i) is heterogeneous in nature, that is, the software components have different workloads and failure behavior; (ii) is a terminating system, that is, execution begins with a single software component and terminates on a single software component; and (iii) can be described using sequential and branching architectural styles. Research shows that models exist in all three categories of ABSRMs which are suitable for heterogeneous and terminating systems [13], [15], [42]. However, unlike the composite type state-based models and path-based models, the hierarchical type state-based models are not developed to address the different architectural styles used for describing a system [13]. Therefore, we exclude the hierarchical type state-based category of ABSRMs and proceed with composite type state-based models and path-based models.

- *Validity of assumptions according to the data:* Identifying an applicable category of ABSRMs based only on the assumptions discussed in Table 1 may be difficult as these assumptions are common in most architectural models. The models in each category make distinct assumptions in order to increase their applicability in addressing different reliability requirements at the design phase [9], [16], [18], [32], [36]. Both the basic and distinct assumptions can be used for identifying a category of architectural models and/or a model amongst candidate models. At the design phase, the failure behavior of a software component is determined, based on the information collected during the testing of its previous version(s). The APS does not provide any information regarding the historical test data of its software components', thus making it difficult for developers to determine which fault detection/removal process is to be adopted during the testing phase. Many architectural models assume perfect fault removal process during the testing phase, that is, the faults are removed instantly after detection without introducing any new faults [15], [49]. Architectural models making such a distinct assumption can be useful for APS. Such models exist in both the composite type state-based model and the path-based model categories; and therefore this criterion is not of much help in identifying an applicable category of ABSRMs.
- *Output desired by the user:* In the application, the requirement of the designers is to estimate the system-level architectural reliability of the APS and to identify software components which are critical from the reliability perspective. Although composite type state-based models provide function for the estimation of system-level architectural reliability, their ability to perform sensitivity and predictive analysis, which is required for the identification of critical software components, is found to be limited under certain circumstances [3], [9], [35]. Since the failure behavior of the software components in the APS is represented using the time-dependent failure intensity function and the architecture is described using absorbing discrete time Markov chains, we cannot use composite type state-based models. The models in the path-based category of ABSRMs provide a system-level architectural reliability estimation function and are proven to be suitable for performing sensitivity and predictive analysis [15], [21]. Therefore, the path-based category of ABSRMs seems to be a suitable choice for the APS.

Step 5: Selecting an ABSRM

In the next step of the selection phase, we find the best model from the path-based category of ABSRMs to estimate the architectural reliability of the APS. The first task in the achievement of this goal is to assign the degree of influence weights to each attribute, based on the attribute's influence on the model selection criteria. Each attribute of the model selection criteria represents a condition. In a case where more than one condition has an equal influence on the model selection criteria, it is possible to assign the same degree of influence weights to the attributes of a specific model selection criterion [37]. For example, all ABSRMs assume that the software components are independent and the modification of a software component has no impact on the others, therefore these attributes of the validity of assumptions according to the data criteria have equal importance and are assigned the same degree of influence weights.

Table 6. Computing Total Score for the Candidate Models

Attributes	DI	Krishnamurthy and Mathur's model	Shooman's model	Hsu and Huang's model
Input required by the model				
i. Reliability values of software components	3	3 x 1	3 x 1	3 x 1
ii. Inter-component transition probabilities	2	2 x 1	2 x 1	2 x 1
Total score		5	5	5
Nature of the project				
i. Sequential and branch architectural styles	4	4 x 0	4 x 0	4 x 1
ii. Terminating application	3	3 x 1	3 x 1	3 x 1
iii. Heterogeneous application	2	2 x 1	2 x 0	2 x 1
Total score		5	3	9

Validity of assumptions according to the data				
i. System is component-based	5	5 x 1	5 x 1	5 x 1
ii. All software components are physically independent of each other	4	4 x 1	4 x 1	4 x 1
iii. Modification of one software component has no impact on others	4	4 x 1	4 x 1	4 x 1
iv. The transfer of control among software components can be described by a Markov process	2	2 x 1	2 x 1	2 x 1
v. Faults are removed perfectly without introducing any new faults	2	2 x 0	2 x 1	2 x 1
Total score		15	17	17
Output desired by the user				
i. Estimate reliability of the system	5	5 x 1	5 x 1	5 x 1
ii. Identification of critical software components	4	4 x 1	4 x 0	4 x 1
Total score		9	5	9

In Table 6, we list the attributes of each of the four model selection criteria and their corresponding degree of influence weights. It is important to note here that we have assumed the degree of influence weights listed in Table 6 merely to demonstrate the application of the ABSRM selection method. After assigning the degree of influence weights, the next task is to analyse whether the models belonging to the path-based category of ABSRMs satisfy the conditions of the attributes in different model selection criteria. For this, we use three path-based models as candidate models viz. Krishnamurthy and Mathur's model [21], Shooman's model [36] and Hsu and Huang's model [15]. The respective assigned values to the candidate model are multiplied with the degree of influence weight of the attribute and then these numbers are summed to obtain the total score for a candidate model for a particular model selection criterion. Table 6 shows that in contrast to the other two candidate models, Hsu and Huang's model [15] satisfactorily fulfils all conditions of each model selection criterion and thus obtains the highest total score for each model selection criteria. Hsu and Huang's model [15] is therefore selected for estimating the architectural reliability of the APS.

4.3 Results and Discussion

In Table 6, the total scores for the candidate models are presented for each of the four model selection criteria. Since Shooman's model [36] does not: (i) consider the branching type of architectural style; (ii) consider the heterogeneous nature of software; and (iii) facilitate sensitivity and predictive analysis to identify critical software components in a real-time assembly, a zero value is assigned to these attributes. Similarly, Krishnamurthy's and Mathur's model [21] does not: (i) consider the branching type of architectural style; and (ii) assume that the fault detection process is perfect debugging, a zero value is assigned to these attributes. It is important to note that a value of one is given for the attributes: (i) require inter-component transition probabilities; and (ii) the transfer of control among software components can be described using a Markov process, as these attributes are not applicable to both Shooman's model [36] and Krishnamurthy's and Mathur's model [21].

Table 7. Cumulative Scores for the Candidate Models

Criteria	Krishnamurthy and Mathur's model	Shooman's model	Hsu and Huang's model
Input required by the model	5	5	5
Nature of the project	5	3	9
Validity of assumptions according to data	15	17	17
Output desired by the user	9	5	9
Cumulative score	34	30	40

Table 7 presents the cumulative scores of all three candidate models. The result of this analysis shows that Hsu and Huang's model [15] is the best choice for the APS. This model computes the architectural reliability of a CBRTS using the approximation of testing path reliabilities where the reliability of a testing path depends on the sequence in which the software components are executed and the reliability estimates of each software component in the testing path. It utilizes discrete time Markov chains and is capable of calculating the reliability of a testing path representing a sequential structure and testing path depicting a mixture of sequential as well as branching structures. When more testing paths are available as testing proceeds, this model recalibrates the estimated accuracy. It also allows designers to assess the impact of an individual software component on the overall behaviour of the APS and identify the critical software component from a reliability perspective through sensitivity analysis. The critical software components usually involve complex interactions (i.e., in/out-branch edges) with each other [10], [13]. Through traversing the critical software components, a critical path can be obtained, and the path selection thus depends on this critical path. Considering that testing resources are limited, using software component's criticality information, the test managers can develop an effective testing strategy so that more resources can be allocated to test the critical software components or paths instead of non-critical ones [4], [10], [17], [26]. Also, the priority of different requests for correcting failures in software components can be arranged in the order of their criticality with respect to the software reliability [30].

5. Conclusions and Future Work

In this paper, we have presented an ABSRM selection method to solve the model selection problem at the design phase of a CBRTS development. The method is based on the model selection criteria discussed in an existing algorithm [1] and the use of a degree of influence weighing mechanism. Our method is different from traditional architecture-based model selection approaches in two ways. First, the method provides flexibility in terms of identifying an applicable category of ABSRMs, that is, composite type state-based models, hierarchical type state-based models or path-based models, by analysing the distinct characteristics of models in each category with attributes of specific model selection criteria which reduce the intensity of ABSRM selection and are not bound to any specific category of ABSRMs. Second, the method minimizes the likelihood of obtaining more than one suitable model from the applicable category of ABSRMs by assigning degree of influence weights to each attribute of the different model selection criteria.

We have demonstrated the usefulness of the ABSRM selection method by applying it to an automotive Automatic Parking System. The application shows that the method provides a logical way of finding a model in the space of all categories of ABSRMs viz. state-based models (including composite models and hierarchical models) and path-based models. The application results are very much in line with the notion that the path-based models are better than the state-based models, in terms of estimating the architectural reliability of large scale and complex component-based systems which can have hundreds of possible states, as these models do not face the state space explosion problem [9], [46] and gives an encouraging indication as to the applicability of path-based modelling approaches for CBRTSs.

A limitation of our ABSRM selection method is that assigning appropriate degree of influence weights to the attributes of different model selection criteria becomes a challenging task in the case of a new component, for which neither there is any information regarding its prior release nor do the designers have experience in dealing with a similar type of system. In such cases, it may be possible that more than one applicable model should be obtained for the analysis. Thus, a future work will be an empirical validation of our ABSRM selection method using different types of CBRTSs. Since the degree of influence weights are machine readable and the properties of different ABSRMs can be analysed using a computer program, automating the ABSRM selection method would be another future work.

Acknowledgment

The financial support from Australia's Cooperative Research Centre for Advanced Automotive Technology (AutoCRC) for supporting this research work is hereby acknowledged by the authors.

References

[1] Asad, C. A., Ullah, I. M., & Muhammad, J. R. (2004). An approach for software reliability model selection. *Proceedings of the 28th Annual International Computer Software and Applications Conference* (pp. 534-539). IEEE.

[2] Bozga, M., Graf, S., & Mounier, L. (2002). A validation environment for component-based real-time systems. *Computer Aided Verification: Lecture Notes in Computer Science*.

[3] Brosch, F., Koziolek, H., Buhnova, B., & Reussner, R. (2010). Parameterized reliability prediction for component-based software architectures. *Proceedings of the 6th International Conference on the Quality of Software Architectures: Lecture Notes in Computer Science No. 6093* (pp. 36-51). Berlin, Heidelberg: Springer - Verlag.

[4] Cheung, R. C. (1980). A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, *6(2)*, 118-125.

[5] Everett, W. (1999). Software component reliability analysis. *Proceedings of the Symposium on Application-Specific Systems and Software Engineering Technology* (pp. 204-211).

[6] Garg, M., & Lai, R. (2014). Software reliability model selection for component-based real-time systems. *Proceedings of the International Conference on Data and Software Engineering* (pp. 1-6).

[7] Garg, M., Lai, R., & Huang, S. J. (2011). When to stop testing: a study from the perspective of software reliability models. *IET Software, 5(3)*, 269-273. IET Publishing Group.

[8] Garg, M., Lai, R., & Kapur, P. K. (2013). A method for selecting a model to estimate the reliability of a software component in a dynamic system. *Proceedings of the 22nd Australian Conference on Software Engineering* (pp. 40- 50).

[9] Gokhale, S. S. (2007). Architecture-based software reliability analysis: overview and limitations. *IEEE Transactions on Dependable and Secure Computing, 4(1)*, 32-40.

[10] Gokhale, S. S., & Trivedi, K. S. (2002). Reliability prediction and sensitivity analysis based on software architecture. *Proceedings of the 13th International Symposium on Software Reliability Engineering* (pp. 64 – 75).

[11] Gokhale, S. S., & Trivedi, K. S. (2006). Analytical models for architecture-based software reliability prediction: a unification framework. *IEEE Transactions on Reliability, 55(4)*, 578 - 590.

[12] Gokhale, S. S., Wong, W. E., Trivedi, K. S., & Horgan, J. R. (1998). An analytical approach to architecture-based software reliability prediction. *Proceedings of the Computer Performance and Dependability Symposium* (pp. 13 – 22).

[13] Goseva-Popstojanova, K., & Trivedi, K. S. (2001). Architecture based approach to reliability assessment of software systems. *Performance Evaluation, 45(2)*, 179 - 204.

[14] Her, J. S., Choi, S. W., Cheun, D. W., Bae, J. S., & Kim, S. D. (2007). A component based process for developing automotive ECU software. *In J. Munch, & P. Abrahamsson (Eds.), PROFES: Lecture Notes in Computer Science No. 4589*.

[15] Hsu, C. J., & Huang, C. Y. (2011). An adaptive reliability analysis using path testing for complex component-based software systems. *IEEE Transactions on Reliability*, *60(1)*, 158 - 170.

[16] Hsu, C. J., Huang, C. Y., & Chang, J. R. (2005). Enhancing software reliability modeling and prediction through the introduction of time-variant fault reduction factor. *IEEE Transactions on Reliability, 54(4)*, 592 - 603.

[17] Huang, C. Y., & Lin, C. T. (2010). Analysis of software reliability modeling considering testing compression factor and failure-to-fault relationship. *IEEE Transactions on Computers, 59(2)*, 283 – 288.

[18] Immonen, A., & Niemelä, E. (2008). Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling, 7(1)*, 49 – 65.

[19] Jung, H. G., Kim, D. S., & Kim, J. (2010). Light-stripe-projection-based target position designation for intelligent parking-assist system. *IEEE Transactions on Intelligent Transportation Systems, 11(4)*, 942 - 953.

[20] Kopetz, H. (2011). *Real-Time Systems: Design Principles for Distributed Embedded Applications* (2nd ed.). Springer.

[21] Krishnamurthy, S., & Mathur, A. P. (1997). On the estimation of reliability of a software system using reliabilities of its components. *Proceedings of the 8th International Symposium on Software Reliability Engineering* (pp. 146 – 155).

[22] Kruchten, P., Obbink, H., & Stafford, J. (2006). The past, present, and future for software architecture. *IEEE Software*, *22(2)*, 22 - 30.

[23] Lai, R., & Garg, M. (2012). A detailed study of software reliability models (invited paper). *Journal of Software, 7(6)*, 1296 - 1306.

[24] Lai, R., Garg, M., & Kapur, P. K. (2011). A study of when to release a software product from the perspective of software reliability models. *Journal of Software, 6(4)*, 651 - 661.

[25] Littlewood, B. (1975). A reliability model for systems with Markov structure. *Applied Statistics, 24(2)*, 172 – 177. JSTOR.

[26] Lo, J. H., Huang, C. Y., Chen, I. Y., Kuo, S. Y. & Lyu, M. R. (2005). Reliability assessment and sensitivity analysis of software reliability growth modeling based on software modular structure. *Journal of Systems and Software, 76(1)*, 3 - 13. Elsevier.

[27] Mahmood, S., & Lai, R. (2013). RE-UML: A component-based system requirements analysis language. *Computer Journal, 56(7)*, 901 - 922. Oxford University Press.

[28] Meedeniya, I., Buhnova, B., Aleti, A., & Grunske, L. (2010). Architecture-driven reliability and energy optimization for complex embedded systems. In G. T. Heinemann, J. Kofron, & F. Plasil (Eds.) *Proceedings of the 6th International Conference on the Quality of Software Architectures: Lecture Notes in Computer Science No. 6093* (pp. 52 – 67). Berlin, Heidelberg: Springer - Verlag.

[29] Parnas, D. L. (1975). The influence of software structure on reliability. *Proceedings of the International Conference on Reliable Software* (pp. 358 – 362).

[30] Pietrantuono, R., Russo, S. & Trivedi, K. S. (2010). Software reliability and testing time allocation: an architecture-based approach. *IEEE Transactions on Software Engineering*, *36(3)*, 323 – 337.

[31] Ramani, S., Gokhale S., & Trivedi, K. S. (2002). SREPT: Software Reliability Estimation and Prediction Tool. *Performance Evaluation, 39*, 37 - 60. Elsevier.

[32] Reussner, R., Schmidt, H. W., & Poernomo, I. (2003). Reliability prediction for component-based software architectures. *Journal of Systems and Software, 66(3)*, 241 – 252. Elsevier.

[33] Schneidewind, N. (2009). Integrating testing with reliability. *Software Testing, Verification and Reliability, 19(3)*, 175 – 198. John Wiley and Sons.

[34] Seigrist, K. (1998). Reliability of systems with Markov transfer of control. *IEEE Transactions on Software Engineering, 14(7)*, 1049 – 1053.

[35] Sharma, V. S., & Trivedi, K. S. (2007). Quantifying software performance, reliability and security: an architecture-based approach. *Journal of Systems and Software, 80(4)*, 493 - 509. Elsevier.

[36] Shooman, M. L. (1976). Structural models for software reliability prediction. *Proceedings of the 2nd International Conference on Software Engineering* (pp. 268 – 280). IEEE.

[37] Symons, C. R. (1988). Function point analysis: difficulties and improvements. *IEEE Transactions on Software Engineering, 14(1)*, 2 - 11.

[38] Tian, J., Rudraraju, S., & Li, Z. (2004). Evaluating web software reliability based on workload and failure data extracted from server logs. *IEEE Transactions on Software Engineering, 30(11)*, 754 – 769.

[39] Tripathi, R., & Mall, R. (2005). Early stage software reliability and design assessment. *In Proceedings of the 12th Asia-Pacific Software Engineering Conference* (pp. 619 – 628). IEEE.

[40] Trivedi, K. S. (2001). Probability and statistics with reliability, queuing and computer science applications. John Wiley and Sons.

[41] Wang, S., Rho, S., Mai, Z., Bettati, R., & Zhao, W. (2005). Real-time component-based systems. *In Proceedings of the 11th Real Time and Embedded Technology and Applications Symposium* (pp. 428 – 437). IEEE.

[42] Wang, W., Pan, D., & Chen, M. (2006). Architecture-based software reliability modeling. *Journal of Systems and Software, 79(1)*, 132 - 146. Elsevier.

[43] Wijayasiriwardhane, T., & Lai, R. (2010). Component point: A system-level size measure for component-based software systems. *Journal of Systems and Software, 83(12)*, 2456 - 2470. Elsevier.

[44] Wijayasiriwardhane, T., Lai, R., & Kang, K. C. (2011). Effort estimation of component-based software development – a survey. *IET Software, 5(2)*, 216 - 228. IET Publishing Group.

[45] Xie, M., & Wohlin, C. (1995). An additive reliability model for the analysis of modular software failure data. *Proceedings of the 6th International Symposium on Software Reliability Engineering* (pp. 188 – 194). IEEE.

[46] Yacoub, S., Cukic, B., & Ammar, H. H. (2004). A scenario-based reliability analysis approach for component-based software. *IEEE Transactions on Reliability, 53(4)*, 465 – 480.

[47] Wesslen, A., Runesson, P., & Regnell, B. (2000). Assessing the sensitivity to usage profile changes in test planning. *Proceedings of the 11th International Symposium on Software Reliability Engineering* (pp. 317 – 326). IEEE.

[48] Lipow, M. (1982). Number of faults per line of code. *IEEE Transactions on Software Engineering,* SE - *8(4)*, 437 - 439, IEEE.

[49] Gokhale, S., Philip, T., Marinos, P. N., & Trivedi, K. S. (1996). Unification of finite failure non-homogeneous Poisson process model through test coverage. *In Proceedings of the 7th International Symposium on Software Reliability Engineering* (pp. 289 – 299). IEEE.

[50] P. Kubat (1989). Assessing reliability of modular software. *Operations Research Letters, 8(1)*, 35 - 41.

Mohit Garg is with the Department of Computer Science and Information Technology La Trobe University, Australia. He received his MSc (by research) in computer science from La Trobe University in 2009, and his PhD in computer science from La Trobe University in 2013. His research interests include software reliability estimation, software complexity measurement, and component-based real-time systems.

Richard Lai is with the Department of Computer Science and Information Technology at La Trobe University, Australia. Prior to joining La Trobe, he had spent more than 10 years in the computer and communications industry. He was ranked as the world's number one scholar in systems and software engineering consecutively for four years (1999–2002), according to an annual survey published in the Journal of Systems and Software. His current research interests include component-based software system, software measurement, requirements engineering, and global software development.