ARCHITECTURE-BASED SOFTWARE RELIABILITY ANALYSIS
INCORPORATING CONCURRENCY

REHAB A. EL KHARBOULTY, SWAPNA S. GOKHALE and REDA A. AMMAR
Dept. of Computer Science and Engineering, University of Connecticut
Storrs, Connecticut, 06269, USA

Received (received date)
Revised (revised date)

With the growing complexity of software applications and increasing reliance on the services provided by these applications, architecture-based reliability analysis has become the focus of several recent research efforts. Most of the prevalent research in this area does not consider simultaneous or concurrent execution of application components. Concurrency, however, may be common in modern software applications. Thus, reliability analysis considering concurrent component execution within the context of the application architecture is necessary for contemporary software applications. This paper presents an architecture-based reliability analysis methodology for concurrent software applications. Central to the methodology is a state space approach, based on discrete time Markov chains (DTMCs), to represent the application architecture taking into consideration simultaneous component execution. A closed form, analytical expression for the expected application reliability based on the average execution times, constant failure rates, and the average number of visits to the components is derived. The average number of visits to application components are obtained from the solution of the DTMC model representing the application architecture. The potential of the methodology to facilitate sensitivity analysis, identification of reliability bottlenecks, and an assessment of the impact of workload and component changes in addition to providing a reliability estimate is discussed. The methodology is illustrated with a case study. To enable the application of the methodology in practice, estimation of model parameters from different software artifacts is described. Finally, strategies to alleviate the state space explosion issue for an efficient application of the methodology are proposed.

1. Introduction

Software applications permeate many domains including telecommunications, finance, health care, avionics etc., and society today is increasingly reliant on the services provided by these applications. This growing dependence of our society on the services provided by software applications places a heavy premium on their reliable operation. Systematic, rigorous and disciplined approaches to analyze the reliability of software applications are thus necessary.

It is highly desirable that reliability analysis of a software application be conducted earlier in the software development life cycle, since this enables an understanding of the impact of design decisions on application reliability at a time when maximum latitude exists to take corrective action if it is discovered that the desired reliability targets cannot be met. In the early phases, such analysis may be based on the application architecture. A number of research efforts have focused on architecture-based reliability analysis. These research efforts predominantly consider sequential applications, where control resides in only one of the application components at a time. The assumption of sequential component execution was valid in the context of applications that were developed using the procedural programming paradigm. Many modern software applications, however, are built using the object-oriented and component-based software development paradigm, and
concurrency is very common in these applications. Reliability analysis of a software application considering concurrency within the context of application architecture is thus essential for contemporary software applications.

This paper presents a methodology to analyze the reliability of a concurrent software application considering its architecture. We propose a state space approach, based on discrete time Markov chains (DTMCs), to represent the architecture of a concurrent software application. We derive a closed form, analytical expression for application reliability using the average number of visits to components obtained from the solution of the DTMC model representing the application architecture and the average execution times and constant failure rates of the components. We then discuss the potential of the methodology to facilitate sensitivity analysis, identification of reliability bottlenecks, and an assessment of the impact of component and workload changes in addition to providing a reliability estimate. We illustrate the methodology using a case study. To allow the application of the methodology in practice, we discuss how model parameters may be estimated from different software artifacts. Finally, we propose strategies to alleviate the state space explosion issue to enable efficient application of the methodology to large software applications.

The rest of the paper is organized as follows: Section 2 provides an overview of DTMCs. Section 3 describes the state space approach to represent the architecture of a concurrent software application. Section 4 presents the reliability analysis methodology. Section 5 illustrates the methodology with a case study. Section 6 discusses model parameterization. Section 7 provides strategies to alleviate the state space explosion issue. Section 8 summarizes related research and places our effort in the context of prevalent work. Section 9 concludes the paper and outlines directions for future research.

2. Overview of DTMCs

In this section we provide a brief overview of discrete time Markov chains (DTMCs). A detailed treatment of DTMCs can be obtained from elsewhere.

A DTMC is characterized by its one-step transition probability matrix \( P = [p_{i,j}] \). All the elements in \( P \) are in the range \([0, 1]\). \( P \) is a stochastic matrix since all the elements in a row of \( P \) sum to one. In this paper we consider a terminating application whose architecture is described by an absorbing DTMC and hence we elaborate on absorbing DTMCs in this section.

Let \( P \) be the transition probability of an absorbing DTMC with \( s \) absorbing states and a total of \( n \) states. Without loss of generality we assume that the transient states are labeled \( 1, \ldots, n-s \) and the absorbing states are labeled \( n-s+1, \ldots, n \). The transition probability matrix of an absorbing DTMC can be partitioned as:

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

where \( Q \) is a \((n-s) \times (n-s)\) substochastic matrix (with at least one row sum less than 1), \( I \) is a \( s \times s \) identity matrix, \( 0 \) is an \( s \times (n-s) \) matrix of zeros, and \( C \) is an \((n-s) \times s\) matrix. The \( k \)-step transition probability matrix \( P^k \) has the form:

\[
P^k = \begin{bmatrix}
Q^k & C' \\
0 & 1
\end{bmatrix}
\]

where the entries of matrix \( C' \) are not relevant. The \((i, j)\)-th entry of matrix \( Q^k \) denotes the probability of arriving in transient state \( j \), starting from transient state \( i \) in exactly \( k \) steps. It can be shown that \( \sum_{k=0}^{\infty} Q^k \) converges as \( t \) approaches infinity. This implies that the inverse matrix \((I - Q)^{-1}\), called the fundamental matrix \( M \), exists, and is given by:

\[
M = (I - Q)^{-1} = I + Q + Q^2 + \ldots = \sum_{i=0}^{\infty} Q^i
\]  

(1)
The \((i, j)\)-th entry of the fundamental matrix provides the expected number of visits to state \(j\) starting from state \(i\). Thus, if \(\nu_j\) denotes the expected number of visits to state \(j\) assuming that the process starts at state \(i\), then \(\nu_j\) is given by:

\[
\nu_j = m_{i,j}
\]  

(2)

3. Architecture representation

In this section we describe the state space approach to represent the architecture of a concurrent software application.

In a sequential application only a single component is executing at any given time. However, in a concurrent application, several different components may be executing simultaneously. We propose, that at any given time, the application state may be represented by the list of components under execution at that time. The state of the application evolves as components begin and suspend execution. To represent the architecture of such an application using a state space model, a state would correspond to a list of components under execution at a given time. As components begin and suspend execution, this list changes dynamically and unique states along with the transitions among these states can be identified. We assume that the execution of components is independent of one another, that is, the beginning and suspension of one component will not have any impact on the other components.

We explain our approach with the example of a concurrent application with three components. We let \(C_i\) denote component \(i\). Figure 1 shows the complete state space model of the application. The application starts in initial state labeled \(<\text{START}\>\). In this state none of the components are under execution. Three possible events may occur in this state. If component \(C_1\) begins execution, then the application transitions to state \(<\text{C}_1>\). Similarly, the initiation of components \(C_2\) or \(C_3\) transition the application to states \(<\text{C}_2>\) or \(<\text{C}_3>\) respectively. In state \(<\text{C}_1>\), either component \(C_2\) or \(C_3\) can start execution or component \(C_1\) can terminate. If \(C_2\) begins, then the application transitions to state \(<\text{C}_1, \text{C}_2>\), if \(C_3\) begins then the application transitions to state \(<\text{C}_1, \text{C}_3>\), and if \(C_1\) terminates then the application completes and transitions to state \(<\text{END}>\). Reasoning in a similar manner, all the possible execution paths from the initial state to the final state can be obtained. In Table 1 state labels and their descriptions are presented, while Table 2 presents the events that cause transitions among the states.

We let \(p_{i,j}\) denote the transition probability from state \(i\) to state \(j\). We assume that the state space model may be treated as a DTMC, where the transitions among the states follow a Markov process. This implies that the list of components under execution in the next state only depends on the current state and does not depend on the past history of execution.
4. Analysis methodology

The architecture-based reliability analysis methodology, along with the different types of analyses it facilitates is described in this section.

We consider a terminating application, that is, an application which operates on demand. For such an application, it is possible to distinguish between consecutive software runs. We let $n$ denote the number of components in the application and $k$ denote the number of states in the DTMC model representing the application architecture. We assume that the execution time of each component is exponentially distributed and that the failure rate of a component is constant under different operating conditions. For component $i$, we let $1/\mu_i$ denote the mean execution time and $\lambda_i$ denote the constant failure rate. Further, we assume that components fail independently of one another, and the failure of any component causes application failure.

The application reliability, denoted $R$, is given by:

$$R = \prod_{i=1}^{n} R_i$$  (3)

where $R_i$ is the expected reliability of component $i$ in a typical run of the application. $R_i$ is given by:

$$R_i = e^{-\lambda_i T_i}$$  (4)

where $T_i$ is the expected execution time in component $i$ during a typical execution of the application. $T_i$ is given by:
In Equation (5), $\nu_j$ is the expected number of visits to state $j$ which can be obtained by solving the DTMC model of the application architecture. $I_{i,j}$ is an indicator function, where $I_{i,j} = 1$ if component $i$ is active (under execution) in state $j$ and zero otherwise. $t_{i,j}$ is the expected execution time of component $i$ in state $j$. Since the component execution times are exponentially distributed, due to the memoryless property, the distribution of the remaining execution time of a component as the application transitions among states also follows an exponential distribution with the same mean as the initial distribution of the execution time. Thus, $t_{i,j} = 1/\mu_i$ in all the states. $R_i$ is thus given by:

$$R_i = e^{-\lambda_i \mu_i \sum_{k=1}^{k} \nu_j I_{i,j}}$$

(6)

From Equation (6), application reliability $R$ can be written as:

$$R = e^{-\sum_{i=1}^{n} \frac{\lambda_i}{\mu_i} \sum_{j=1}^{k} \nu_j I_{i,j}}$$

(7)

The reliability analysis methodology described above is hierarchical or two-step – In the first step, the state space model representing the application architecture is solved to obtain the average number of visits to the application components. In the second step, these visit statistics are combined with mean execution times and constant failure rates of the components to obtain an analytical reliability function. The analytical reliability function can be used to estimate the application reliability for specific values of failure rates and mean execution times of components, and transition probabilities of the application architecture. In addition, it could also be used for the following purposes:

- **Identification of reliability bottlenecks:** Component $i$, where $\max_i \{ \frac{\lambda_i}{\mu_i} \sum_{j=1}^{k} \nu_j I_{i,j} \}$ holds is the reliability bottleneck, since the expected component reliability in a typical execution of the application is the lowest for this component. Thus, we note that the component with the highest failure rate may not be the reliability bottleneck. In fact, the reliability bottleneck is determined by a component’s failure rate, its mean
execution time, and its architectural context which provides the expected number of visits to the component during a typical execution of the application. Further, if all the components have identical mean execution times and failure rates, then the bottleneck component is the one which is visited the highest number of times. Thus, under these circumstances, the bottleneck component is solely determined by the application architecture.

• **Impact of component changes:** A component may change if one or both of its parameters, namely, constant failure rate and mean execution time change. If the component preserves its interactions with the other components, which are determined by the transition probabilities in the application architecture, then the expected application reliability can be obtained using Equation (8).

\[
R = e^{-\sum_{i=1}^{\infty} \lambda_i c - \mu c \sum_{j=1}^{b} \nu_j I_{i,j}} e^{-\sum_{i=1}^{\infty} \lambda_i c - \mu c \sum_{j=1}^{b} \nu_j I_{c,j}}
\]

In Equation (8), \(\lambda_c^*\) and \(\mu_c^*\) are respectively the new failure rate and the mean execution time of the changed component \(c\). Thus, the expected application reliability when a component changes can be obtained by simply estimating the parameters of the changed component.

• **Sensitivity analysis:** In the design phase, exact values of the component parameters may not be known. In this case, it is necessary to assess the sensitivity of the application reliability to component parameters. Sensitivity analysis requires the computation of application reliability at several intermediate values of each parameter over its entire range of variation. The analytical reliability function in Equation (7) can easily enable such repeated reliability computations, thereby facilitating sensitivity analysis.

The hierarchical nature of the analysis methodology, wherein the architectural model is solved separately regardless of the component parameters can also facilitate an analysis of the impact of workload changes on application reliability. A specific application workload is given by the set of transition probabilities in the application architecture, which are determined by the operational profile of the application. Initially, reliability analysis may be conducted for a specific operational profile. However, a number of factors may cause variations in the operational profile that is used for initial reliability analysis. These factors include a difference between the actual and the anticipated usage of the software and upgrades and evolution which introduce new features. As a result, the sensitivity of the application reliability to variations in the operational profile needs to be determined. To achieve this, the architectural model can be solved for different sets of transition probabilities corresponding to different operational profiles, to obtain the the average number of visits to the components. Each set of the average number of visits can then be used in Equation (7) to compute the application reliability for different operational profiles.

5. Case study

This section illustrates the potential of the reliability analysis methodology described in Section 4 with a case study.

We consider an application comprising of three components with the architecture shown in Figure 1. The nominal values of the transition probabilities of the DTMC model of the application architecture are summarized in Table 3. The average number of visits to the components obtained from the solution of the DTMC model of the architecture are listed in Table 4. If the mean execution times and failure rates of all the components were to be identical, then components #1 and #3 would be the reliability bottlenecks, as solely determined by the mean number of visits to each component obtained from the application architecture.
Table 3: Transition probabilities for the architectural model

<table>
<thead>
<tr>
<th>State #</th>
<th>Transition probabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( p_{0,1} = 0.1 )  ( p_{0,2} = 0.4 )  ( p_{0,3} = 0.5 )</td>
</tr>
<tr>
<td>1</td>
<td>( p_{1,5} = 0.7 )  ( p_{1,6} = 0.2 )  ( p_{1,9} = 0.1 )</td>
</tr>
<tr>
<td>2</td>
<td>( p_{2,4} = 0.7 )  ( p_{2,7} = 0.2 )  ( p_{2,9} = 0.1 )</td>
</tr>
<tr>
<td>3</td>
<td>( p_{3,5} = 0.9 )  ( p_{3,6} = 0.05 ) ( p_{3,9} = 0.05 )</td>
</tr>
<tr>
<td>4</td>
<td>( p_{4,1} = 0.7 )  ( p_{4,2} = 0.2 )  ( p_{4,8} = 0.9 )</td>
</tr>
<tr>
<td>5</td>
<td>( p_{5,1} = 0.1 )  ( p_{5,3} = 0.5 )  ( p_{5,8} = 0.4 )</td>
</tr>
<tr>
<td>6</td>
<td>( p_{6,2} = 0.6 )  ( p_{6,3} = 0.1 )  ( p_{6,8} = 0.3 )</td>
</tr>
<tr>
<td>7</td>
<td>( p_{7,4} = 0.25 ) ( p_{7,5} = 0.25 )  ( p_{7,6} = 0.5 )</td>
</tr>
</tbody>
</table>

Table 4: Average visits to components for probabilities in Table 3

<table>
<thead>
<tr>
<th>Component #</th>
<th>Average visits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.9</td>
</tr>
<tr>
<td>2</td>
<td>3.8</td>
</tr>
<tr>
<td>3</td>
<td>4.9</td>
</tr>
</tbody>
</table>

For the purpose of illustration, we use the nominal values of component parameters summarized in Table 5. The expected application reliability for these nominal parameter values is 0.414. The expected component reliabilities during a typical run of the application are summarized in Table 6. The results in Table 6 indicate that component #2 which has the lowest expected reliability is the reliability bottleneck. Although component #2 has the lowest mean execution time and mean number of visits of all the components as indicated in Tables 6 and 5, its failure rate is significantly higher than the other two components, making it the bottleneck component.

Table 5: Component parameters

<table>
<thead>
<tr>
<th>Component #</th>
<th>Mean execution time (1/( \mu_i ))</th>
<th>Failure rate (( \lambda_i ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20 msec.</td>
<td>0.0016/msec.</td>
</tr>
<tr>
<td>2</td>
<td>6.5 msec.</td>
<td>0.0278/msec.</td>
</tr>
<tr>
<td>3</td>
<td>75 msec.</td>
<td>0.0001/msec.</td>
</tr>
</tbody>
</table>

Table 6: Expected component reliabilities in a typical run

<table>
<thead>
<tr>
<th>Component #</th>
<th>Expected rel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.969</td>
</tr>
<tr>
<td>2</td>
<td>0.834</td>
</tr>
<tr>
<td>3</td>
<td>0.992</td>
</tr>
</tbody>
</table>

Through the next set of experiments we seek to illustrate the value of the reliability analysis methodology to facilitate different types of analyses described in Section 4.

Experiment I: Sensitivity to component execution times
The first experiment seeks to assess the sensitivity of the application reliability to the mean
execution times of the components. Towards this end, we vary the mean execution time of each component, one at a time, in the range of −15% to +15% around the original value. The percentage change in the application reliability as a function of the change in the mean execution time for each component is plotted in Figure 2. From the figure it can be observed that the application reliability is most sensitive to variations in the mean execution time of component #2 and least sensitive to variations in the mean execution time of component #3. Referring to the original mean execution times in Table 5, it can be seen that the mean execution time is lowest for component #2 and highest for component #3. Variations in the mean execution time of component #3, which is already significantly higher than the mean execution times of the other two components thus cause smaller variations in the application reliability. Contrary to component #3, the mean execution time of component #2 is significantly lower than the other two, due to which variations in the mean execution time of component #2 cause higher variations in the application reliability.

Experiment II: Sensitivity to component failure rates
Through the next experiment we seek to assess the sensitivity of the application reliability to the failure rates of the components. For this purpose, the failure rate of each component was varied in the range of −15% to +15%, one at a time, around its original value. The percentage change in application reliability as a function of the percentage change in the failure rates of the components is depicted in Figure 3. As can be seen from the figure, the application reliability is most sensitive to variations in the failure rate of component #2 and least sensitive to variations in the failure rate of component #3. This is due to the fact that originally component #2 has the highest value of the failure rate and component #3 has the lowest value, as indicated in Table 5. It is important to note that the observations of this experiment are contrary to the observations of Experiment I. In Experiment I, the application reliability exhibited highest sensitivity to variations in the mean execution time of the component with the lowest mean execution time, while in this experiment variations in the failure rate of the component with the highest failure rate has the strongest impact on the application reliability.

Experiment III: Sensitivity to application architecture
In the last experiment we assess the impact of architectural changes, manifested by the changes in the transition probabilities of the DTMC model representing the application architecture, on the application reliability. For the sake of illustration, the original transition
The probability matrix was modified as shown in Table 7. The mean number of visits to each component computed for the modified transition probabilities in Table 7 are reported in Table 8. Comparing the average visits to components in Tables 4 and 8, it can be observed that the average number of visits to the components under the modified architecture are much higher than the original average number of visits. This occurs because the modified application architecture is much more imbalanced or skewed compared to the original one.

If the mean execution times and failure rates of the components were to be identical, then component #3 emerges as the reliability bottleneck, solely based on the application architecture. Once again, this is different from the results obtained for the original application architecture where both components #1 and #3 were vying for the bottleneck spot. This highlights the important role played by the application architecture in identifying reliability bottlenecks.

The application reliability for the modified set of transition probabilities, using the nominal failure rates and mean execution times of components reported in Table 5 is 0.035. It is important to note that the application reliability under the modified architecture is significantly lower than the reliability under the original architecture (0.414) for the same values of the component parameters. The strong influence of application architecture on its reliability is highlighted further from a comparison of the reliabilities under the two architectures.

The expected component reliabilities under the modified architecture are reported in
Table 8: Average component visits for modified architecture

<table>
<thead>
<tr>
<th>Component #</th>
<th>Average visits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22.34</td>
</tr>
<tr>
<td>2</td>
<td>13.34</td>
</tr>
<tr>
<td>3</td>
<td>29.54</td>
</tr>
</tbody>
</table>

Table 9. These values indicate that component #2 is the reliability bottleneck even under the modified architecture. This is despite the fact that the difference in the average number of visits to component #2 and components #1 and #3 under the modified architecture is much higher than the difference in the average number of visits for these components under the original architecture.

Table 9: Expected component reliabilities for modified architecture

<table>
<thead>
<tr>
<th>Component #</th>
<th>Expected rel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.489</td>
</tr>
<tr>
<td>2</td>
<td>0.0896</td>
</tr>
<tr>
<td>3</td>
<td>0.799</td>
</tr>
</tbody>
</table>

To assess the sensitivity of the application reliability to variations in the mean execution times and failure rates of the components in conjunction with the modified architecture we repeated Experiments I and II for the modified transition probabilities in Table 7. Figure 4 depicts the percentage change in the application reliability as a function of the percentage change in the mean execution times of the components. Similar to Figure 2, variations in the mean execution time of component #2 cause the highest variations in application reliability. However, the magnitude of these variations is lower under the modified architecture than under the original architecture. This is because of the higher number of visits to component #2 under the modified architecture than under the original architecture as is evident from Tables 4 and 8. Figure 5 depicts the variation in the application reliability as a function of the variations in the failure rates of the components. It can be seen from Figure 3 that under the original architecture, variations in the failure rate of component #2 have a higher impact on the application reliability than the variations in the failure rates of components #1 and #2. However, under the modified architecture, the impact of variations in the failure rates of all three components on the application reliability is nearly the same. Thus, in this case the impact of the modified architecture is sufficiently significant that it masks the influence of the variations in the mean execution times and failure rates of the components.

6. Model parameterization

This section discusses how the parameters necessary for the application of the reliability analysis methodology could be estimated from different software artifacts.

The artifacts used for parameter estimation will depend on the phase of the life cycle in which the architecture-based reliability analysis methodology is to be employed. In the architecture/design phase, state space representation of the application architecture may be extracted from its description in a modeling language such as UML or its specification in a specification language such as SDL. The application description/specification may be simulated to generate profile data, from which the transition probabilities of the state space model can be estimated. It is rarely the case that accurate estimates of the constant failure rates of the components can be obtained in this phase. A promising approach which may provide some guidance on the estimates of component failure rates is based on design metrics. Additionally, expert opinion and failure data collected from similar systems
Figure 4: Sensitivity of application reliability to mean execution times (modified architecture)

Figure 5: Sensitivity of application reliability to failure rates (modified architecture)
or prior releases of the same system may be used to guide the selection of failure rates.

During the testing and operational phases, source and/or object code of the components and the application may be used for parameter estimation. Profile data generated during the execution of the application collected using tools such as Java Virtual Machine Tool Interface (JVMTI) and coverage measurement and analysis tools may be used to extract the state space model of the application architecture. The failure parameters of the components may be estimated from the failure data collected during the testing of the application, metrics-based approaches, fault injection, and statistical testing.

7. State space explosion

In this section we discuss the state space explosion issue that may arise when the analysis methodology is used to assess the reliability of practical software applications. We also propose strategies to alleviate this issue.

The maximum number of states in the state space representation of the application architecture is \(2^n\), where \(n\) is the number of components in the application. In practical software applications, additional restrictions and constraints on the components that can be simultaneously active may drive down the number of feasible states. In the worse case, however, the state space of the architecture representation will grow dramatically as a function of \(n\), which will make it infeasible to solve the DTMC model as is to obtain the expected number of visits to the application components. The state space explosion issue can be handled using the following strategies.

The first strategy consists of tolerating the state space explosion problem. Here it is assumed that the storage and the solution of the large model can be handled, but its manual specification is cumbersome, tedious and error-prone. In this case, high-level specification mechanisms such as Stochastic Reward Nets (SRNs) could be used for specification and the state space can be generated automatically using tools such as the Stochastic Petri Net Package (SPNP). The model specification constructs in the SRN modeling paradigm should offer sufficient expressive power to represent additional constraints among the components.

The second strategy consists of avoiding the state space explosion problem, which can be accomplished using the following methods. A consistent theme across all these methods is to aggregate a subset of the states in the architecture representation. The subset of states which are aggregated will depend on the application characteristics. For some applications, although a large number of states may exist in theory, in practice, the probability of transitioning to a subset of these states may be negligible. For example, it may be very rare that more than a certain number of components are simultaneously executing. In such applications, the occurrence probabilities of the states are imbalanced, and a subset of states is overwhelmingly favored over the others. The rare states can thus be aggregated into a single super state, reducing the number of states in the architecture representation. The other category consists of applications where all the states have similar probabilities of occurrence. For such applications, instead of keeping track of the list of components executing at a given time, we could simply keep track of the number of components that are executing. This will enable a significant reduction in the number of states, from \(2^n\) to \(n\). The strategy may also be tailored for applications with imbalanced state occurrence probabilities.

The state space explosion problem can also be avoided by employing a third strategy consisting of hierarchical model decomposition. In this case, the entire model is decomposed into pieces, which are then analyzed separately. The results obtained from an analysis of these individual pieces can then be combined to obtain the overall application reliability.

8. Related research

In this section we summarize the related research and place the analysis methodology described in this paper in the context of prevalent work.
Architecture-based software reliability analysis has been the focus of several research efforts in the past few years. Prevalent architecture-based analysis techniques can be classified into three categories, namely, path-based, state-based, and additive as proposed by Goseva et al. In the path-based approaches, execution paths through the application architecture are enumerated either algorithmically, experimentally, or via simulation. The reliability of each path is computed as a product of the reliabilities of the components along the path. The application reliability is then obtained by averaging the path reliabilities of several paths. An important drawback of path-based approaches is that it produces only an approximate estimate of the application reliability when the application architecture has infinite paths due to the presence of loops. Further, it is cumbersome and expensive to conduct sensitivity analysis in the path-based approach since it requires a repeated enumeration of the paths through the application architecture.

In the state-based approaches, the application architecture is mapped to a state-space model which is then solved to obtain an estimate of the application reliability. Using analytical methods, state-based approaches can consider the impact of infinite loops and can also facilitate sensitivity analysis. Thus, state-based approaches can alleviate the drawbacks of path-based approaches. Additive models add the failure rates of the components to obtain the overall failure rate of the application. They do not consider the application architecture explicitly.

State-based approaches have received significantly higher degree of attention than the path-based and additive approaches. They can be further classified according to two dimensions. The first dimension is concerned with the application type considered, namely, terminating or continuously running. A terminating application is an application that operates on demand. For such an application, it is possible to distinguish between consecutive runs. On the other hand, a continuously running application operates incessantly until it is either stopped by intervention or due to a failure. The second dimension is concerned with the solution method employed to analyze the state-based model to obtain an estimate of application reliability. In the composite solution method, the state space model representing the application architecture is combined with the parameters representing the failure behavior of the application components into a single model called the composite model. The composite model is then solved to obtain an estimate of the application reliability. On the other hand, in the hierarchical method, the application reliability is obtained in two steps. In the first step, the state space model of the application architecture is solved to obtain the visit statistics of the components. These visit statistics may include the mean and the variance of the number of visits and the mean execution time of the components in a typical execution of the application. These visit statistics are combined with the parameters representing the failure behavior of the components to obtain an analytical function for application reliability. The analytical reliability function expresses application reliability in terms of the failure parameters and the visit statistics of the components. This analytical reliability function can form the basis of sensitivity and predictive analysis, bottleneck identification, an assessment of the impact of workload and component changes, and an exploration of architectural alternatives, in addition to providing an estimate of application reliability. These analyses could be extremely valuable to guide decisions about which components should be developed in-house, which can be developed contractually, and which can be picked off-the-shelf. It can also be useful in determining how resources should be allocated to components, so that the application reliability can be achieved/improved in a cost-effective manner. Due to its ability to enable different types of valuable analyses, the hierarchical method is more desirable compared to the composite method.

A vast majority of state-based approaches consider sequential applications, where exactly one application component is executing at any given time. However, modern software applications are invariably concurrent. A few efforts which address architecture-based reliability analysis for concurrent software applications can also be classified according to the two dimensions described above. This classification is shown in Figure 6. Kanoun et al. present a hierarchical approach for a continuously running application.
al. develop a composite solution approach for a terminating application. The analysis methodology described in this paper provides a hierarchical approach for a concurrent, terminating application. Due to its hierarchical nature the proposed methodology facilitates different types of analyses, which is a significant advantage of this research over the work described by Rodriguez et al., which is the most closely related.

Figure 6: Classification of architecture-based approaches for concurrent applications

9. Conclusions and future research

In this paper we presented a reliability analysis methodology for a concurrent software application based on its architecture. We derived a closed form, analytical expression which relates the application reliability to the mean execution times, constant failure rates, and the average number of visits to the components in a typical execution of the component. To enable the computation of average number of visits to the components, we proposed a state space approach to represent the application architecture. The potential of the methodology to facilitate the identification of reliability bottlenecks, sensitivity analysis and an assessment of the impact of workload and component changes was also discussed. We illustrated the value of the methodology to enable different types of analyses using a case study. To allow the application of the methodology to practical software applications, we described methods to estimate model parameters from different software artifacts and suggested strategies to alleviate the state space explosion issue.

Our future research consists of extending the methodology to consider non exponential component execution times. Developing a reliability analysis approach for continuously running concurrent applications is also a topic of future research. Methods to estimate model parameters from profile data will also be investigated in the future.

References

24. D. R. Jeske, X. Zhang, and L. Pham. “Adjusting software failure rates that are esti-
