JAIST Repository
https://dspace.jaist.ac.jp/

Title	Modeling and Analyzing the Impact of Software Fault Tolerance Techniques on the Reliability of Component-based Systems
Author(s)	Pham, Thanh-Trung; Defago, Xavier
Citation	Research report (School of Information Science, Japan Advanced Institute of Science and Technology), IS-RR-2013-003: 1-12
Issue Date	2013-01-25
Type	Technical Report
Text version	publisher
URL	http://hdl.handle.net/10119/12084
Rights	
Description	リサーチレポート（北陸先端科学技術大学院大学情報科学研究科）

JAIST
JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Modeling and Analyzing the Impact of Software Fault Tolerance Techniques on the Reliability of Component-based Systems

Thanh-Trung Pham and Xavier Défago

School of Information Science,
Japan Advanced Institute of Science and Technology (JAIST),
Nomi, Ishikawa, Japan
Email: {thanhtrung.pham,defago}@jaist.ac.jp

***Abstract*—Reliability is one of the most important quality attributes of a software system. One way to improve the reliability of a software system is to use software fault tolerance techniques. However, existing reliability prediction approaches for component-based systems either do not allow modeling software fault tolerance techniques or do not support defining explicitly reliability-relevant behavioral aspects of software fault tolerance techniques. This limits the ability to apply the approaches in different application contexts. In this paper, we extend the core model of a recent component-based reliability prediction approach to take into consideration software fault tolerance techniques and analyze their impact on the overall reliability of component-based systems. We demonstrate the applicability of our approach by modeling the reliability of the reporting service of a document exchange server and conducting reliability predictions.**

***Index Terms*—component-based reliability prediction, software fault tolerance techniques, error detection, execution models.**

I. Introduction

Techniques for analyzing properties of a software design and a software system are useful for both functional properties (e.g. correctness) and quality properties (e.g. reliability, performance, security, etc.). Predicting quality properties of a software system based on design models can help not only to make the system more dependable but also to save costs, time and efforts significantly by avoiding implementing software architectures that do not meet the quality requirements.

Reliability[1] is one of the most important quality attributes of a software system. To improve the reliability of a software system, software fault tolerance techniques are often used. Software fault tolerance techniques provide the ability to mask faults in software systems, prevent them from leading to failures and can be applied on different abstraction levels (e.g. source code level, design pattern level, architecture level) [1]. However, existing component-based reliability prediction approaches either do not allow modeling software fault tolerance techniques (e.g. [2]–[4]) or do not support defining explicitly reliability-relevant behavioral aspects of software fault tolerance techniques (e.g. [5], [6]). This limits the ability to apply the approaches in different application contexts.

Furthermore, most existing reliability prediction approaches [6]–[8] only use a failure/non-failure domain, regardless of multiple failure types of a component service or different failure types of different component services and therefore cannot model complicated reliability-related behaviors. Besides, these approaches also accept an assumption of sequential component executions and therefore cannot model concurrent executions.

We extend the core model of the approach of Pham et al. [9], which incorporates error propagation analysis for different execution models, to takes into consideration software fault tolerance techniques and analyze their impact on the overall reliability of component-based systems. Our approach allows defining explicitly reliability-relevant behavioral aspects of software fault tolerance techniques and therefore offers a flexible way to model different software fault tolerance techniques. Moreover, our approach supports modeling with models similar to UML activity diagrams and UML component diagrams that are transformed automatically into Markov chains for reliability predictions.

We demonstrate the applicability of our approach by considering the reporting service of a document exchange server. We make reliability predictions and sensitivity analyses of the overall system reliability to model parameters.

Contribution: The main contribution of this paper is twofold. First, we propose an extended model for the flexible modeling of software fault tolerance techniques in component-based systems. With two primitive fault tolerance structures, we allow modeling several classical software fault tolerance techniques including primary-backup, try-catch and try-retry. Second, we illustrate our approach by studying the reliability of the reporting service of a document exchange server. In particular, we are not only able to predict the reliability of the reporting service but also to identify the most sensitive components, the most effective software fault tolerance techniques for a given architecture of the reporting service.

Structure: The rest of this paper is organized as follows. Section II surveys related work. Section III describes our modeling foundation and the steps in our methodology. Section IV describes in details about modeling the component reliability specifications, modeling system reliability models and the transformation to create Markov models in order to predict reliability. Section V demonstrates our approach with a case study. Section VI discusses the assumptions and limitations of our approach and Section VII concludes the paper.

[1]The reliability can be defined as the complement of the probability of failure on demand.

II. RELATED WORK

The field of architecture-based software reliability modeling has been surveyed by several authors [7], [8], [10]. One of the first approaches to architecture-based software reliability modeling is Cheung's approach [11] that uses Markov chains. Recent work enhances such models to combine reliability analysis with performance analysis [12], to support compositionality [4], to take into consideration several factors related to components' reliability [13]. However, these approaches do not consider software fault tolerance techniques, multiple failure types and concurrent executions.

Other approaches in this field such as the approach of Cheung et al. [14] focuses on the reliability of individual components, the approach of Zheng et al. [15] aims at service-oriented systems, the approach of Cortellessa et al. [2] and the approach of Goseva et al. [3] apply UML modeling language, but also do not consider software fault tolerance techniques and multiple failure types. Moreover, these approaches, except the approach of Zheng et al., do not consider concurrent executions.

Some approaches consider explicitly error propagation such as the approach of Mohamed et al. [16] and the approach of Filieri et al. [17]. To model the possibilities of masking, transforming and propagating different component failures, they introduce several specific concepts such as multiple failure types and error propagation probabilities. These possibilities can be used to express a system's fault tolerance capabilities but software fault tolerance techniques cannot be modeled explicitly by these approaches. Besides, these approaches do not consider concurrent executions.

Some approaches take into consideration software fault tolerance techniques in modeling reliability. The approach of Sharma et al. [5] allows modeling component restarts and component retries. This approach does not consider the influences of both error detection and error handling in software fault tolerance techniques on control and data flow within components. Moreover, this approach also does not consider multiple failure types and concurrent executions. The approach of Wang et al. [6] supports different architectural styles including fault tolerance architectural style and parallel architectural style. However, similar to the approach of the Sharma et al., the approach of Wang et al. does not consider the influences of both error detection and error handling, and multiple failure types. The same holds for the approach of Pham et al. [9] which considers error propagation for multiple execution models including concurrent and primary-backup fault tolerance executions. The approach of Brosch et al. [18] provides modeling elements to express software fault tolerance techniques. Although the approach of Brosch et al. considers multiple failure types and the influence of error handling, it ignores the influence of error detection. Moreover, the approach of Brosch et al. also does not consider concurrent executions. Ignoring the influences of either error detection or error handling can lead to incorrect results when the behaviors of software fault tolerance techniques deviate from the specific cases mentioned by the authors.

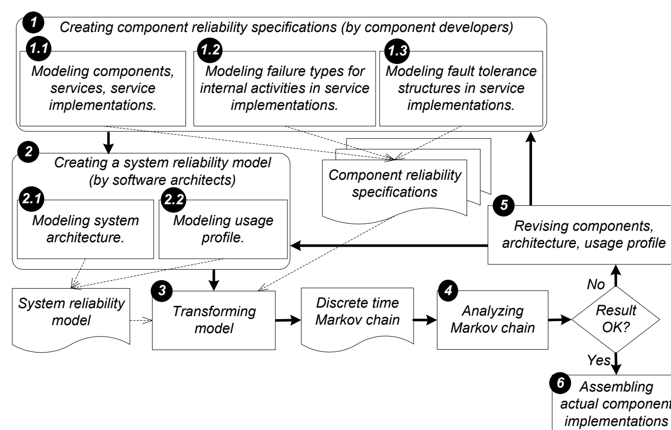

Fig. 1. Component-based reliability prediction.

III. COMPONENT-BASED RELIABILITY PREDICTION

In component-based software engineering (CBSE), there exists a strict separation between component developers and software architects. Component developers implement components; provide functional specifications and quality specifications (i.e. models). Functional specifications are sufficient to assemble components and check their interoperability. Quality specifications are required to reason about quality attributes such as reliability, performance, security of a component-based software architecture. Software architects use these specifications to reason about quality attributes of the proposed architectures and then assemble the actual component implementations.

Component developers create component quality specifications by describing how provided services of a component call required services in terms of probabilities, frequencies and parameter values. Software architects create a model of flow and data control throughout the entire architecture by incorporating these specifications without referring to component internals.

Fig. 1 shows six main steps in our approach. In Step 1, component developers create component reliability specifications. First, component developers model components, services and service implementations (Step 1.1). The results of Step 1.1 are enriched by different failure types with their occurrence probabilities for internal activities in service implementations (Step 1.2). Then, component developers can introduce different fault tolerance structures (FTSs) in service implementations, e.g. *RetryStructures* or *MultiTryCatchStructures* (Step 1.3). FTSs allow different configurations, for example, the number of times to retry for a *RetryStructure* or the number of replicated instances for handling certain failure types in a *MultiTryCatchStructure*. In Step 2, software architects create a system reliability model. First, software architects model the system architecture (Step 2.1) and then the usage profile (Step 2.2). In Step 3, the system reliability model, combined with the component reliability specifications, is transformed automatically into a discrete time Markov chain (DTMC). In

Step 4, by analyzing the DTMC, a reliability prediction and sensitivity analyses can be deduced. If the predicted reliability does not meet the reliability requirement, Step 5 is performed. Otherwise, Step 6 is performed. In Step 5, component developers can revise the components, for example, changing the configurations of FTSs. Software architects can revise the system architecture and the usage profile, for example, trying different system architecture configurations, replacing some key components with more reliable variants, or adjusting the usage profile appropriately. In Step 6, the modeled system is deemed to meet the reliability requirement; software architects assemble the actual component implementations following the system architecture model.

IV. Reliability Modeling

A. Basic Concepts

According to Avizienis et al. [19], an error is defined as the part of the system state that may lead to a failure. The cause of the error is called a fault. A failure occurs when the error causes the delivered service to deviate from correct service. The deviation can be manifested in different ways, corresponding to the system's different failure types.

In the same paper, the authors also describe clearly fault tolerance techniques. A fault tolerance technique is carried out through error detection and system recovery. Error detection is to determine the presence of an error. System recovery is to transfer a system state containing one or more errors and (possibly) faults to a state without detected errors and without faults that can be activated again. Error handling and fault handling together form system recovery. Error detection itself also has two different failure types: 1) signaling the presence of an error when no error has actually occurred, i.e. false alarm, 2) not signaling the presence of an error, i.e. an undetected error.

From that, it is necessary to deal with multiple failure types of a component service and different failure types of different component services for modeling complicated reliability-related behaviors. Moreover, it is important to take into consideration both error detection and error handling in modeling software fault tolerance techniques. However, to the best of our knowledge, no architecture-based reliability prediction approach takes into consideration error detection and its influences on control and data flow when modeling software fault tolerance techniques.

In the next section, we introduce our model for describing reliability-relevant characteristics of component-based software systems. Compared with MARTE-DAM profile [20], our model uses only a small set of necessary concepts to achieve our purposes such as reliability prediction, sensitivity analyses.

B. Component Reliability Specifications

1) Components, services and services implementations: Fig. 2 shows the modeling elements for component reliability specifications in our model (Step 1 in our approach). In Step 1.1, component developers model components, services and service implementations via modeling elements: *Component*, *Service* and *ServiceImplementation* , respectively. Components are associated with services via *RequiredService* and *ProvidedService*.

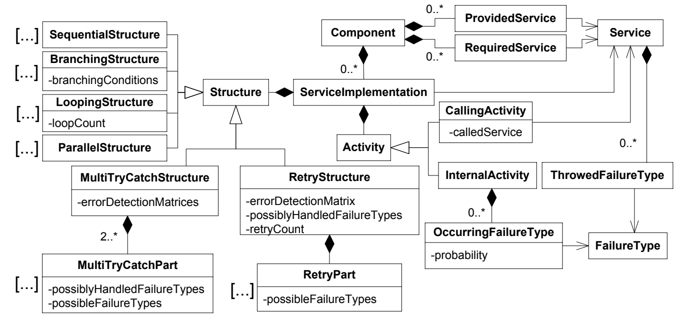

Fig. 2. Modeling elements for component reliability specifications.

In order to analyze reliability, component developers are required to describe the behavior of each service provided by a component, i.e. describe the activities to be executed when a service (*Service*) in the provided services of the component is called. Therefore, a component can contain multiple service implementations. A service implementation can include activities (*Activity*) and structures (*Structure*). There are two activity types: internal activities (*InternalActivity*) and calling activities (*CallingActivity*). An internal activity represents a component's internal computation. A calling activity represents a synchronous call to other components, that is, the caller blocks until receiving an answer. The called service of a calling activity is a service in the required services of the current component and this referenced required service can only be substituted by the provided service of other component when the composition of the current component to other components is fixed. Some structure types are sequential structures (*SequentialStructure*), branching structures (*BranchingStructure*), looping structures (*LoopingStructure*) and parallel structures (*ParallelStructure*). For branching structures, branching conditions are Boolean expressions. For looping structures[2], the number of loops is always bound, infinite loops are not allowed. Looping structures may include other looping structures but cannot have multiple entry points and cannot be interconnected. For parallel structures, parallel branches are supposed to be executed independently, i.e., their reliability behaviors are independent.

2) Failure Types: In Step 1.2, component developers model failure types for internal activities of service implementations via an association between *InternalActivity* and *FailureType*. This association models failure types occurring during a service execution with a probability. Different techniques [10], [14], such as growth reliability modeling, statistic testing or fault injection, can be used to determine these probabilities.

3) Fault Tolerance Structures:

a) Error detection: In software fault tolerance techniques, correct error detection is the prerequisite condition

[2]In our model, an execution cycle is also modeled by a looping structure with its depth of recursion as loop count.

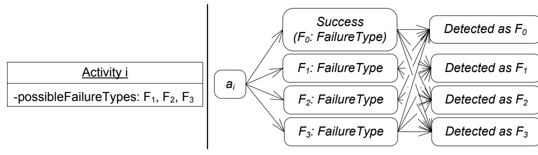

Fig. 3. Error detection semantics for an activity example.

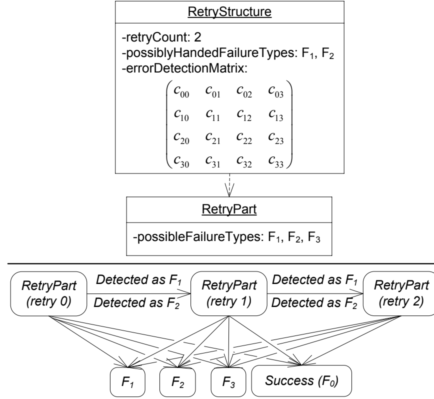

Fig. 4. Semantics for a *RetryStructure* example.

for a correct error handling. On the contrary, an undetected error leads to no error handling and a false alarm leads to an incorrect error handling.

Example 1: Fig. 3 shows an activity with three possible failure types: F_1, F_2 and F_3 (a new failure type, F_0, is introduced which corresponds to the correct service delivery). To provide error handling for the failure types, it is necessary to detect the failure types correctly. From that, for each failure type $F_i(i = 0, 1, 2, 3)$, fraction c_{ij} of being detected as failure type $F_j(j = 0, 1, 2, 3)$ needs to be provided. Therefore, the error detection can be described by the following matrix:

$$\begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix} \sum_j c_{ij} = 1$$

Rows (columns) 0, 1, 2 and 3 correspond to failure types F_0, F_1, F_2 and F_3, respectively. The elements of row 0 (except the first element c_{00}) correspond to false alarms. The elements of column 0 (except the first element c_{00}) correspond to undetected errors. The elements outside the diagonal of the matrix correspond to false signaling of failure type. In case of perfect error detection, the error detection matrix is an identity matrix.

b) RetryStructure: An effective technique to handle transient failures and therefore increase the service reliability is service re-execution. A *RetryStructure* is taken ideas from this technique. The structure contains a single *RetryPart* that can contain within itself different activity types, different structure types and even a nested *RetryStructure*. The first execution of the *RetryPart* models normal service execution while the following executions of the *RetryPart* model the service re-executions.

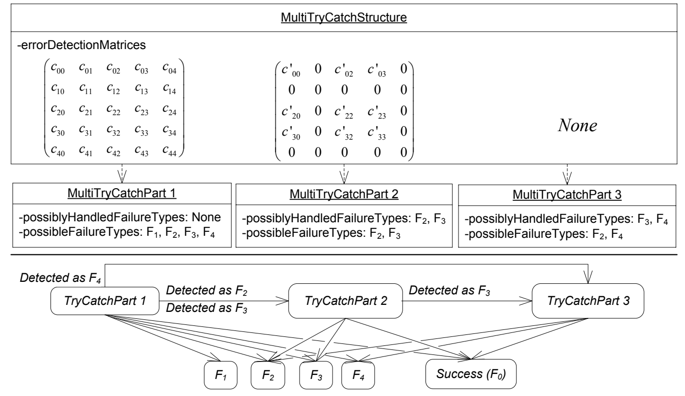

Fig. 5. Semantics for a *MultiTryCatchStructure* example.

Example 2: Fig. 4 shows a *RetryStructure* with a single *RetryPart* having three possible failure types: F_1, F_2 and F_3, an error detection matrix (as in Fig. 4), two possibly handed failure types: F_1 and F_2 and retry count: 2. During the execution of the *RetryPart*, all failure types F_1, F_2 and F_3 can occur. The field *possiblyHandledFailureTypes* of this structure shows that only failure types that are detected as failure types F_1 and F_2 lead to retry the *RetryPart*. This is repeated with the number of times equal to the field *retryCount* of the structure. Because *RetryPart (retry 2)* is the last retry, it is not necessary to apply error detection modeling for it in our reliability modeling.

c) MultiTryCatchStructure: A *MultiTryCatchStructure* is taken ideas from the exception handling in object-oriented programming. The structure consists of two or more *MultiTryCatchParts*. Each *MultiTryCatchPart* can contain different activity types, different structure types and even a nested *MultiTryCatchStructure*. Similar to try and catch blocks in exception handling, the first *MultiTryCatchPart* models the normal service execution while the following *MultiTryCatchParts* handle certain failure types by launching alternative activities.

Example 3: Fig. 5 shows a *MultiTryCatchStructure* with three *MultiTryCatchParts* and four possible failure types F_1, F_2, F_3 and F_4. During the execution of the first *MultiTryCatchPart*, all failure types can occur. Because the possibly handled failure types of *MultiTryCatchPart 2* are F_2, F_3 and of *MultiTryCatchPart 3* are F_3, F_4, only failure types of *MultiTryCatchPart 1* that are detected as failure types F_2, F_3 and F_4 lead to finding *MultiTryCatchParts* to handle detected failure types. In particular, the failure types of *MultiTryCatchPart 1* that are detected as failure types F_2 and F_3 lead to *MultiTryCatchPart 2*, the failure types of *MultiTryCatchPart 1* that are detected as failure type F_4 lead to *MultiTryCatchPart 3*. During the execution of *MultiTryCatchPart 2*, possible failure types are F_2 and F_3. Therefore, rows and columns corresponding to failure types F_1 and F_4 of the error detection matrix for *MultiTryCatchPart 2* contain all 0. The error detection matrix for *MultiTryCatchPart 2* can be simplified by eliminating the rows and columns corresponding to the

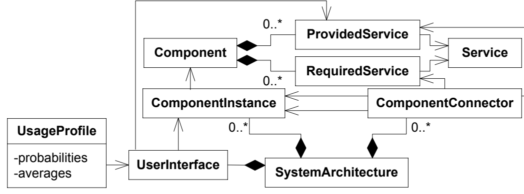

Fig. 6. Modeling elements for system reliability models.

impossible failure types of *MultiTryCatchPart 2* to be as follows:

$$\begin{pmatrix} c'_{00} & c'_{02} & c'_{03} \\ c'_{20} & c'_{22} & c'_{23} \\ c'_{30} & c'_{32} & c'_{33} \end{pmatrix}$$

The former error detection matrix for *MultiTryCatchPart 2* can be restored by using the possible failure types of *MultiTryCatchPart 2*. Because the possibly handled failure types of *MultiTryCatchPart 3* are F_3 and F_4, only failure types of *MultiTryCatchPart 2* that are detected as failure type F_3 lead to *MultiTryCatchPart 3*. Because there is no *MultiTryCatchPart* to handle failures of *MultiTryCatchPart 3*, an error detection matrix for it is not necessary in our reliability modeling.

Because a *MultiTryCatchPart* can contain different activity types (*InternalActivity*, *CallingActivity*) and different structure types (*SequentialStructure*, *BranchingStructure*, *LoopingStructure*, *ParallelStructure* and even nested FTSs like *RetryStructure* and *MultiTryCatchStructure*), it is quite flexible in modeling the different fault tolerance techniques. For example, if a *MultiTryCatchPart* contains within itself a *CallingActivity*, errors from the provided service of the called component also can be handled.

For FTSs, we don't include state recovery error because the design of recovery mechanism is sufficiently simple enough that standard design practices can ensure that there are no residual faults in its design [21].

C. System Reliability Models

1) System Architecture: Fig. 6 shows the modeling elements for system reliability models in our model (Step 2 in our approach). In Step 2.1, software architects model system architecture via modeling element *SystemArchitecture*. The software architects create component instances (*ComponentInstance*) and assemble them through *ComponentConnectors* to realize the required functionality. Users can access this functionality through a user interface (*UserInterface*).

2) Usage Profile: After modeling system architecture, software architects model a usage profile for the user interface of the required functionality. A usage profile (*UsageProfile*) includes probabilities and averages. The usage profile must include sufficient information to determine the branching probabilities of branching structures and the average number of loops for each looping structure.

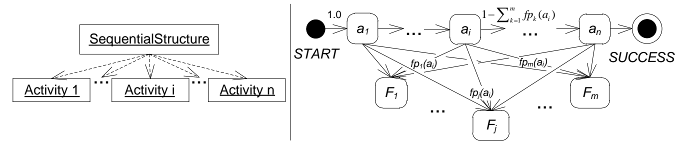

Fig. 7. Sequential structure and its Markov model.

D. Transformation for Reliability Prediction

This transformation is to build a Markov model that reflects all the possible execution paths throughout the entire system architecture and their corresponding probabilities. After that, the system reliability can be calculated from this Markov model.

The transformation starts with a user interface that is actually a reference to a service implementation. Because a service implementation can contain a structure of any structure types or an activity of any activity types, a structure can contain different structure types and different activity types, and a calling activity can refer to another service implementation, the transformation, in essence, is a recursive procedure of transforming for the structure types and the activity types.

1) Internal Activity: The probabilities of different failure types of an internal activity (abbreviated as *ia*) are provided as a direct input to the model: $fp_j(ia)$. The success probability of an internal activity can be calculated by $sp(ia) = fp_0(ia) = 1 - \sum_{k=1}^{m} fp_k(ia)$ where m is the number of failure types.

2) Sequential Structure: Fig. 7 shows a sequential structure and its Markov model. The sequential structure includes *n* activities numbered from *1* to *n*; each activity has *m* different failure types numbered from *1* to *m*. Its Markov has *n+m+2* states, *n* states corresponding to *n* activities, *m* states corresponding to *m* failure types and *2* states: *START* and *SUCCESS*. The transition probability between state a_i and state F_j is the probability of failure type *j* of activity *i*: $fp_j(a_i)$. The transition probability between state a_i and a_{i+1} is the non-failure probability of activity *i*: $1 - \sum_{k=1}^{m} fp_k(a_i)$.

Therefore, the success probability of the structure is the probability of reaching state SUCCESS from state START:

$$sp = fp_0 = \prod_{i=1}^{n} (1 - \sum_{k=1}^{m} fp_k(a_i)) \tag{1}$$

The probability of failure type *j* ($j \neq 0$) of the structure is the probability of reaching state F_j from state *START*:

$$fp_j = \sum_{i=1}^{n} \left(\left(\prod_{k=1}^{i-1} \left(1 - \sum_{l=1}^{m} fp_l(a_k) \right) \right) fp_j(a_i) \right) \tag{2}$$

Equation (2) can be obtained from the following disjoint cases:

- When activity a_1 fails with failure type *j*. This case has probability $fp_j(a_1)$.
- When activity a_1 succeeds, activity a_2 fails with failure type *j*. This case has probability $(1 - \sum_{k=1}^{m} fp_k(a_1))\, fp_j(a_2)$.
- ...

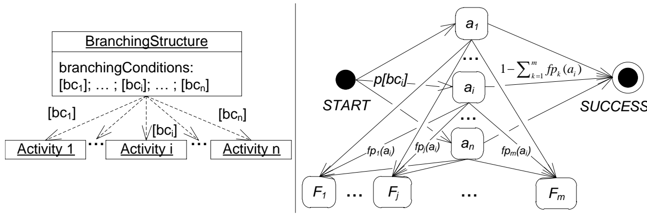

Fig. 8. Branching structure and its Markov model.

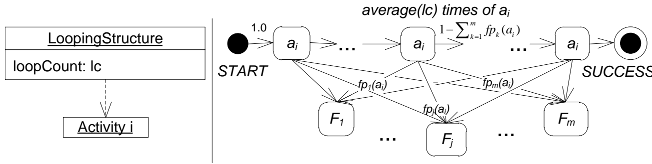

Fig. 9. Looping structure and its Markov model.

- When activities a_1, a_2, ..., a_{n-1} succeed, activity a_n fails with failure type *j*. This case has probability $\left(\prod_{i=1}^{n-1}\left(1-\sum_{k=1}^{m} fp_k(a_i)\right)\right) fp_j(a_n)$.

3) Branching Structure: Fig. 8 shows a branching structure and its Markov model. The branching structure includes *n* activities numbered from *1* to *n* in *n* branches; each activity has *m* different failure types numbered from *1* to *m*. Its Markov model has *n+m+2* states, *n* states corresponding to *n* activities, *m* states corresponding to *m* failure types and *2* states: *START* and *SUCCESS*. The transition probability between state a_i and state F_j is the probability of failure type *j* of activity *i*: $fp_j(a_i)$. The transition probability between state *START* and state a_i is the probability of the branching condition *i*: $p[bc_i]$. The transition probability between state a_i and state *SUCCESS* is the non-failure probability of activity *i*: $1-\sum_{k=1}^{m} fp_k(a_i)$.

Therefore, the success probability of the structure is the probability of reaching state *SUCCESS* from state *START*:

$$sp = fp_0 = \sum\nolimits_{i=1}^{n} p(bc_i)\left(1-\sum\nolimits_{j=1}^{m} fp_j(a_i)\right) \tag{3}$$

The probability of failure type *j* of the structure is the probability of reaching state F_j from state *START*:

$$fp_j = \sum_{i=1}^{n}\left(p(bc_i) fp_j(a_i)\right) \tag{4}$$

4) Looping Structure: Fig. 9 shows a looping structure and its Markov model. The looping structure includes an activity *i* which is looped *lc* times; this activity has *m* different failure types numbered from *1* to *m*. Its Markov model has *average(lc)* number of a_i states, *m* state corresponding to *m* failure types and *2* states: *START* and *SUCCESS*. The transition probability between state a_i and state F_j is the probability of failure type *j* of activity *i*: $fp_j(a_i)$. The transition probability between two consecutive a_i states is the non-failure probability of activity *i*: $1-\sum_{k=1}^{m} fp_k(a_i)$.

Therefore, the success probability of the structure is the probability of reaching state *SUCCESS* from state *START*:

$$sp = fp_0 = \left(1-\sum\nolimits_{k=1}^{m} fp_k(a_i)\right)^{average(lc)} \tag{5}$$

The probability of failure type *j* ($j \neq 0$) of the structure is the probability of reaching state F_j from state *START*:

$$fp_j = \sum_{i=1}^{average(lc)}\left(\left(\prod_{k=1}^{i-1}\left(1-\sum_{l=1}^{m} fp_l(a_i)\right)\right) fp_j(a_i)\right) \tag{6}$$

5) Parallel Structure: For a parallel structure, we assume its parallel branches fail independently. Furthermore, in order to avoid introducing additional failures types for the parallel structure when its parallel branches fail in different failure types, we assume that the failure types are sorted in a certain order (for example, according to their severity). Therefore, when its parallel branches fail in different failure types, the failure type of the parallel structure is the highest failure type of its parallel branches. Without loss of generality, we assume that the failures types are sorted in the following order: $F_1 \leq F_2 \leq \ldots \leq F_m$.

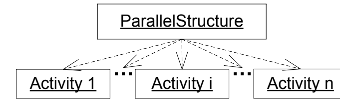

Fig. 10. Parallel structure.

Considering a parallel structure including *n* activities numbered from *1* to *n* as in Fig. 10, the parallel structure succeeds when all the parallel branches succeed:

$$sp = fp_0 = \prod_{i=1}^{n}\left(1-\sum\nolimits_{k=1}^{m} fp_k(a_i)\right) \tag{7}$$

The parallel structure fails with failure type *j* if at least one branch fails with failure type *j* and no parallel branch fails with failure type $k > j$:

$$fp_j = \sum_{i=1}^{n}\left(\begin{array}{l}\prod\limits_{k=1}^{i-1}\left(1-\sum\limits_{l=j}^{m} fp_l(a_k)\right) \times fp_j(a_i) \\ \times \prod\limits_{k=i+1}^{n}\left(1-\sum\limits_{l=j+1}^{m} fp_l(a_k)\right)\end{array}\right) \tag{8}$$

Equation (8) can be obtained from the following disjoint cases:

- When activity a_1 fails with failure type *j*, the remaining activities a_2, a_3, ..., a_n do not fail with failure type $k > j$. This case has probability $fp_j(a_1)\prod\limits_{i=2}^{n}\left(1-\sum\limits_{k=j+1}^{m} fp_k(a_i)\right)$.
- When activity a_1 fails with failure type $k < j$, activity a_2 fails with failure type *j*, the remaining activities a_3, a_4, ..., a_n do not fail with failure type $k > j$. This case has probability $\left(1-\sum\limits_{k=j}^{m} fp_j(a_1)\right) fp_j(a_2)\prod\limits_{i=3}^{n}\left(1-\sum\limits_{k=j+1}^{m} fp_k(a_i)\right)$.

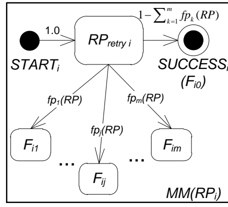

Fig. 11. Markov model for i-th retry.

- ...
- When activities a_1, a_2, ..., a_{n-1} fail with failure type $k < j$, activity a_n fails with failure type *j*. This case has probability $\left(\prod_{i=1}^{n-1}\left(1-\sum_{k=j}^{m} fp_j\left(a_i\right)\right)\right) fp_j(a_n)$.

6) RetryStructure: For a *RetryStructure*, let *rc* be the number of times to retry, F_H be the set of possibly handed failure types, *C* be the error detection matrix represented by $\{c_{rs}\}\, r, s = 0, 1, \ldots m$. The *RetryPart* (abbreviated as *RP*) has *m* different failure types numbered from *1* to *m*, failure type *j* has probability $fp_j\left(RP\right)$. Therefore, the *i-th* retry can be represented by a Markov model $MM\left(RP_i\right)$ as in Fig. 11.

Markov model $MM\left(RP_i\right)$ has a start state $START_i$, a success state $SUCCESS_i$ (or F_{i0}) and a failure state F_{ij} for each failure type *j*. The probability of reaching F_{i0} from state $RP_{retry\ i}$ is $fp_0(RP) = 1 - \sum_{k=1}^{m} fp_k(RP)$. With the number of times to retry *rc*, there are *rc +1* Markov model $MM\left(RP_i\right)$, *i* from *0* to *rc*. The problem is how to connect these Markov models $MM\left(RP_i\right)(i = 0, 1, \ldots rc)$ into one Markov model representing the whole structure. To solve the problem, we add *m+2* states, namely one state *START*, one state *SUCCESS* (or F_0) and states F_j for failure types $(j = 1, 2, \ldots m)$, and the following transition:

- The transition from state *START* to state $START_0$ having probability *1.0*.
- For Markov model $MM\left(RP_{rc}\right)$ (i.e. the Markov model of the last retry): transition from state F_{rcj} to state F_j having probability *1.0* $(j = 0, 1, 2, \ldots m)$
- For the other Markov models, i.e. $MM\left(RP_i\right)$ with *i* from *0* to *rc-1*: transition from state F_{ij} to state $START_{i+1}$ having probability $\sum_{\forall F_k \in F_H} c_{jk}\, (j = 0, 1, 2, \ldots m)$; transition from state F_{ij} to state F_j having probability $1 - \sum_{\forall F_k \in F_H} c_{jk}\, (j = 0, 1, 2, \ldots m)$

As the resulting Markov model is an absorbing Markov chain, the success probability of the whole structure, which is the probability of reaching state *SUCCESS* from state *START*, and the probability of failure type *j* of the structure, which is the probability of reaching state F_j from state *START*, can be calculated [22].

7) MultiTryCatchStructure: For a *MultiTryCatchStructure*, let *n* be the number of *MultiTryCatchParts*, F_{Hi} be the set of possibly handled failure types of *MultiTryCatchPart* *i* $(i = 1, 2, \ldots n)$, *C[i]* be the error detection matrix for *MultiTryCatchPart i* which is represented by $\left\{{c^i}_{rs}\right\} r, s = 0, 1, \ldots m$. The *MultiTryCatchPart i* (abbreviated as $MTCP_i$) has *m* different failure types, numbered from *1* to *m*, failure type *j* having probability $fp_j\left(MTCP_i\right)$. Therefore, *MultiTryCatchParts* can be represented by Markov models as in Fig. 12.

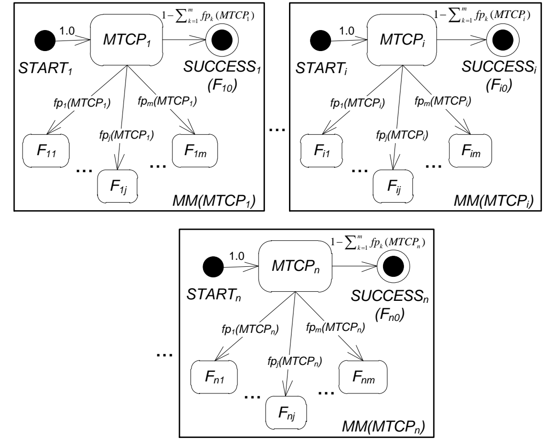

Fig. 12. Markov models for *MultiTryCatchParts*.

Markov model $MM\left(MTCP_i\right)$ has a start state $START_i$, a success state $SUCCESS_i$ (or F_{i0}) and failure states F_{ij} for failure types *j* $(j = 1, 2, \ldots m)$. The probability of reaching state F_{i0} from state $MTCP_i$ is $fp_0(MTCP_i) = 1 - \sum_{k=1}^{m} fp_k(MTCP_i)$. The problem is how to connect these Markov models $MM\left(MTCP_i\right)$ $(i = 1, 2, \ldots n)$ into one Markov model representing the whole structure. To solve the problem, we add *m+2* state, namely one state *START*, one state *SUCCESS* (or F_0) and states F_j for failure types $(j = 1, 2, \ldots m)$, and the following transition:

- The transition from state *START* to state $START_0$ having probability *1.0*.
- For Markov model $MM\left(MTCP_n\right)$ (i.e. the Markov model of the last *MultiTryCatchPart*): transition from state F_{nj} to state F_j having probability *1.0* $(j = 0, 1, 2, \ldots m)$.
- For other Markov models, i.e. $MM\left(MTCP_i\right)$ with *i* from *1* to *n-1*: transition from state F_{ij} to state $START_x$, $x \in \{i+1, i+2, ...n\}$ having probability $\sum_{\forall F_k \in F_{Hix}} c_{jk}^i$ where $F_{Hix} = F_{Hx} - \bigcup_{i<y<x} F_{Hy}$ $(j = 0, 1, 2, \ldots m)$; Transition from state F_{ij} to state F_j $(j = 0, 1, 2, \ldots m)$ having probability $1 - \sum_{i<x\leq n} \sum_{\forall F_k \in F_{Hix}} c_{jk}^i$.

With the Markov model representing the whole structure, the probability of reaching state *SUCCESS* from state *START* is the success probability of the structure and the probability of reaching state F_j from state *START* is the probability of failure type *j* of the structure.

Finally, based on all the calculations described above, the success probability of the service implementation corresponding to the user interface can be calculated in a recursive way, giving the reliability of the requested functionality with the given usage profile.

TABLE I
DIFFERENT FAILURE TYPES AND THEIR SYMBOLS.

Failure Type	Symbol
ProcessingRequestFailure	F_1
ViewingReportFailure	F_2
GeneratingReportFailure	F_3
AttachmentInfoFailure	F_4
FileInfoFailure	F_5
InfoFromLogFailure	F_6
InfoFromDBFailure	F_7

V. CASE STUDY

A. Description of the Case Study

The program chosen for the case study is the reporting service of a document exchange server. The document exchange server is an industrial system which was designed in a service-oriented way. Its reporting service allows generating reports about pending documents or released documents. This service was written in Java and consists of about 2,500 lines of code.

By analyzing the code, it was possible to get the system reliability model of the reporting service as in Fig. 13. At the architecture level, the reporting service consists of four components: *ReportingMediator*, *ReportingEngine*, *SourceManager* and *DestinationManager*. The component *SourceManager* provides two services to get information about pending documents: *getAttachmentDocumentInfo* to get information about pending documents attached in emails and *getFileDocumentInfo* to get information about pending documents stored in file systems. The component *DestinationManager* provides two services to get information about released documents: *getReleasedDocumentInfoFromLogs* to get the information from the logs, *getReleasedDocumentInfoFromDB* to get the information from the database (DB). The component *ReportingEngine* provides two services: *generateReport* to generate a new report (either about pending documents (*aboutPendingDocuments*=true) or about released documents (*aboutPendingDocuments*=false)) and *viewRecentReports* to view recently generated reports (with the number of reports specified by *numberOfRecentReports*). The component *ReportingMediator* provides the service *processReportRequest* for handling incoming report request from clients. An incoming report request can be about generating a new report (*requestType*=generate) or viewing recently generated reports (*requestType*=view).

There are different types of failures which may occur in the component instances during the operation of the reporting service. For example, a *ProcessingRequestFailure* may occur during processing report requests of clients in the service *processReportRequest*; bugs in the code of the service *generateReport* may lead to a *GeneratingReportFailure*. Table I shows different failure types of the reporting service and their symbols.

In the system reliability model of the reporting service, there are two FTSs. The first is the *RetryStructure* in the implementation of the service *viewRecentReports*. This structure has the ability to retry in case there is a *ViewingReportFailure*. The number of times to retry of this structure is

TABLE II
NO. OF REINSERTED FAULTS INTO INTERNAL ACTIVITIES.

Symbol	Provided service/Internal activity (ia)	No. of reinserted faults
a_1	*processReportRequest/ia*	0
a_2	*viewRecentReports/ia*	2
a_3	*generateReport/ia 1*	0
a_8	*generateReport/ia 2*	1
a_4	*getAttachmentDocumentInfo/ia*	1
a_5	*getFileDocumentInfo/ia*	1
a_6	*getReleasedDocumentInfoFromLogs/ia*	2
a_7	*getReleasedDocumentInfoFromDB/ia*	1

TABLE III
FAILURE PROBABILITIES OF INTERNAL ACTIVITIES

Symbol	$fp_j\left(a_i\right)$
a_1	$fp_j\left(a_1\right) = 0 \forall j$
a_2	$fp_2\left(a_2\right) = 0.26087; fp_j\left(a_2\right) = 0 \forall j \neq 2$
a_3	$fp_j\left(a_3\right) = 0 \forall j$
a_8	$fp_3\left(a_8\right) = 0.0549451; fp_j\left(a_8\right) = 0 \forall j \neq 3$
a_4	$fp_4\left(a_4\right) = 0.111111; fp_j\left(a_4\right) = 0 \forall j \neq 4$
a_5	$fp_5\left(a_5\right) = 0.0277778; fp_j\left(a_5\right) = 0 \forall j \neq 5$
a_6	$fp_6\left(a_6\right) = 0.339286; fp_j\left(a_6\right) = 0 \forall j \neq 6$
a_7	$fp_7\left(a_7\right) = 0.0909091; fp_j\left(a_7\right) = 0 \forall j \neq 7$

1 (*retryCount*=1). The second is the *MultiTryCatchStructure* in the implementation of the service *generateReport*. This structure has the ability to handle a *InfoFromLogFailure* of the service *getReleasedDocumentInfoFromLogs* by redirecting calls to the service *getReleasedDocumentInfoFromDB*.

After the last fault removal, the reporting service has been used without having new failures. We used this gold version of the service as an oracle in our case study. We obtained a faulty version of the service by reinserting faults discovered during operational usage and integration testing (Table II shows the number of reinserted faults).

B. Parameter Estimation and Validity of Predictions

To validate the accuracy of our prediction approach, we first estimated the input parameters of the model, including failure probabilities of internal activities, branching probabilities of branching structures, the average number of loops for each looping structure and error detection matrices of FTSs; used the estimated input parameters to compute the predicted reliability by following the method in the Section IV.D and then compared the predicted reliability with the actual reliability of the reporting service. Notice that the goal of our validation is not to justify the input parameters of the model or to imply any accuracy in their estimates but to show that if the system reliability model is provided accurately, our method gives a reasonably accurate reliability prediction.

The faulty version of the reporting service and the oracle were executed on the same test cases for the reporting service. By comparing their outputs and investigating the executions of test cases, we were able to estimate the input parameters of the model. Faults have not been removed and the number of failures includes recurrences because of the same fault.

We estimate the failure probability of the failure type j $(F_j, j = 1, 2, ...7)$ of the internal activity a_i $(i = 1, 2, \ldots 8)$ as:

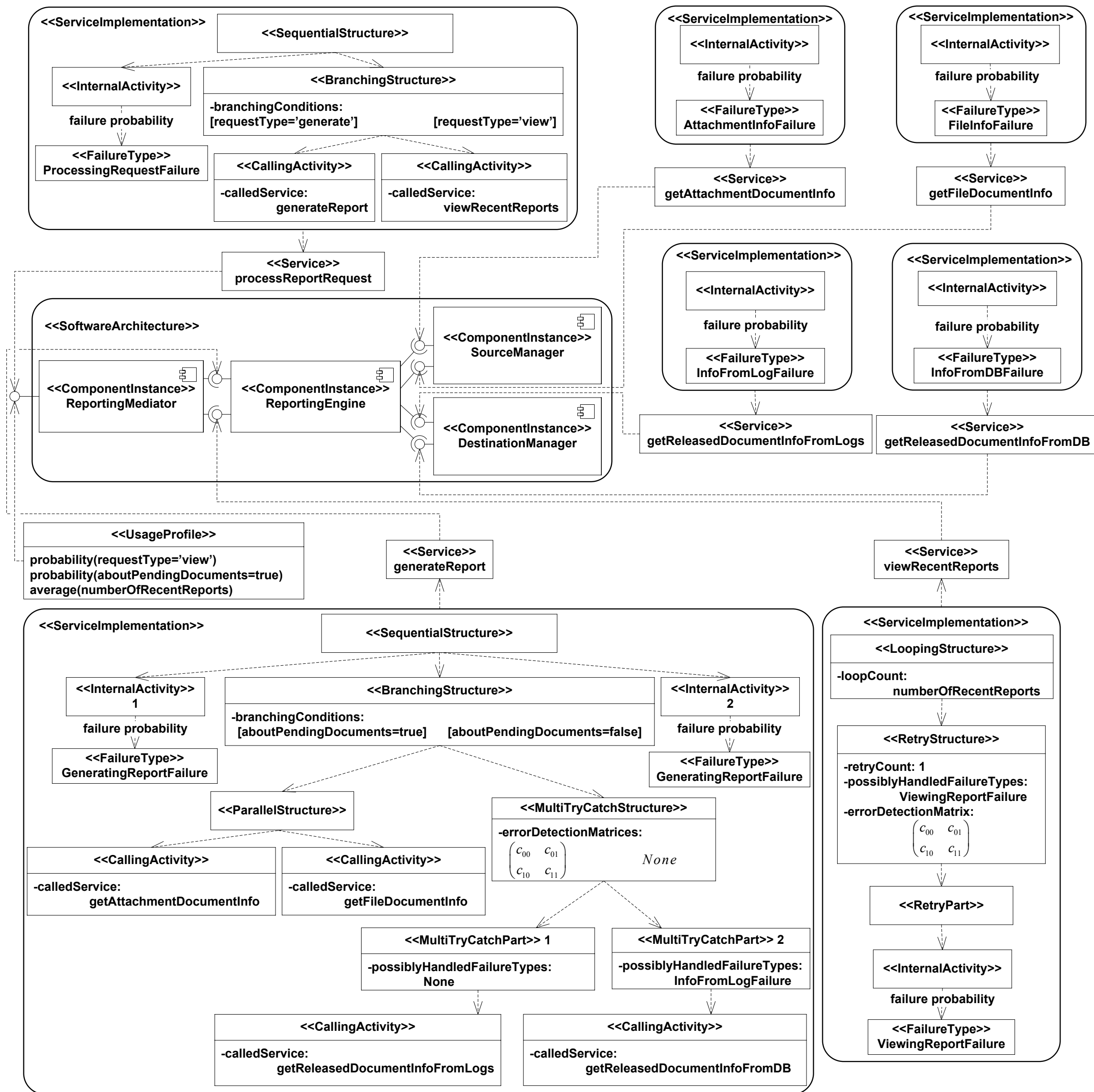

Fig. 13. The system reliability model of the reporting service.

TABLE IV
ERROR DETECTION MATRICES.

Provided service/FTS	Error detection matrix
viewRecentReports/RetryStructure	$\begin{pmatrix} 1.0 & 0.0 \\ 0.222222 & 0.777778 \end{pmatrix}$
generateReport/MultiTryCatchStructure	$\begin{pmatrix} 1.0 & 0.0 \\ 0.421053 & 0.578947 \end{pmatrix}$

$fp_j\,(a_i) = f_{ji}/n_i$ where f_{ji} is the number of failures of the failure type *j* of the internal activity a_i and n_i is the number of runs of the internal activity a_i in the set of test cases for the reporting service. Failure probabilities of different failure types of internal activities are given in Table III. Because no fault was injected into the two internal activities a_1 and a_3, their failure probabilities are assumed to be 0.

The error detection matrix of a FTS was estimated as $(c_{rs}) = (daf_{rs}/f_r); r, s = 0, 1, ...7$ where f_r is the number of failures of the failure type *r* (F_r) of the inner part of the FTS (i.e., *RetryPart* for a *RetryStructure* or *MultiTryCatchPart i* for a *MultiTryCatchStructure*) and daf_{rs} is the number of failures of the failure type *r* of the inner part detected as the failure type *s* (F_s). The simplified error detection matrices for the two FTSs are given in Table IV.

Branching probabilities of a branching structure was estimated as $p(bc_i) = n_i/n$ where n_i is the number of times control was transferred along the branch with branching condition bc_i and *n* is the total number of times control reached the branching structure.

The average number of loops of a looping structure was estimated as $average\,(lc) = nir/n$ where *nir* is the number of runs of the inner part of the looping structure, *n* is the number of times control reached the looping structure.

TABLE V
USAGE PROFILE.

Usage profile element	Value
p(*requestType*=view)	0.178571
p(*aboutPendingDocuments*=false)	0.608696
average(*numberOfRecentReports*)	2

TABLE VI
PREDICTED VS. ACTUAL RELIABILITY FOR THE FAULTY VERSION

Component Instance /Provided service	Predicted reliability	Actual reliability	Difference	Error (%)
ReportingMediator/ processReportRequest	0.800261	0.794643	0.005618	0.707

The usage profile including the branching probabilities of the branching structures and the average number of loops of the looping structure is given in Table V.

We estimate the actual reliability of the reporting service as $R = 1 - F/N$ where F is the number of failures of the reporting service in N test cases for the reporting service. Table VI shows the comparison between the predicted reliability and the actual reliability for the faulty version. From this comparison, we deem that for the system reliability model described in this paper, our analytical method is sufficiently accurate.

C. Sensitivity Analyses and the Impact of FTSs

In this subsection, we first present the results of sensitivity analyses of the reliability of the reporting service to changes of probabilities in the usage profile, to changes of failure probabilities of internal activities and to changes of error detection probabilities of FTSs. Then, we present the analysis of how the predicted reliability of the reporting service varies for different fault tolerance variants.

First, we conducted a sensitivity analysis modifying the usage probabilities (Fig. 14). The reliability of the reporting service is more sensitive to the portion of request types required by users (*requestType*=generate or *requestType*=view) because its corresponding curve has the steepest slope.

Second, we conducted a sensitivity analysis modifying failure probabilities of the internal activities (Fig. 15). The reliability of the reporting service is most sensitive to the failure probability of *ProcessingRequestFailure* (F_1) of the internal activity (a_1) of the service *processReportRequest* provided by the component instance *ReportingMediator* because its corresponding curve has the steepest slope. The reliability of the reporting service is most robust to the failure probabilities of the internal activities (a_2, a_6, a_7) of the services related to the two FTSs, namely the service *viewRecentReports* containing the *RetryStructure*; the service *getReleasedDocumentInfoFromLogs* and the service *getReleasedDocumentInfoFromDB* in the *MultiTryCatchStructure*. Based on this information, the software architect can decide to put more testing effort into the component *ReportingMediator*, to exchange the component with another component from a third party vendor, or run the component redundantly.

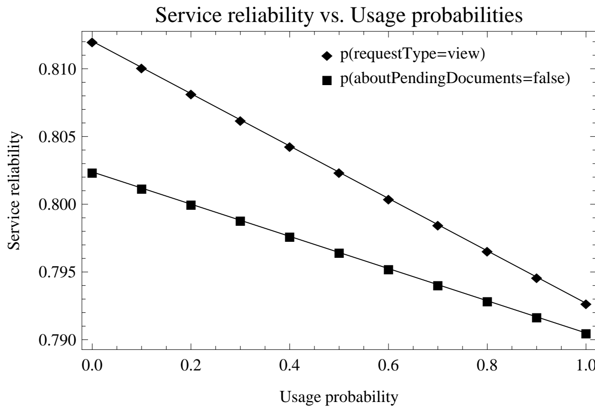

Fig. 14. Sensitivity to usage probabilities.

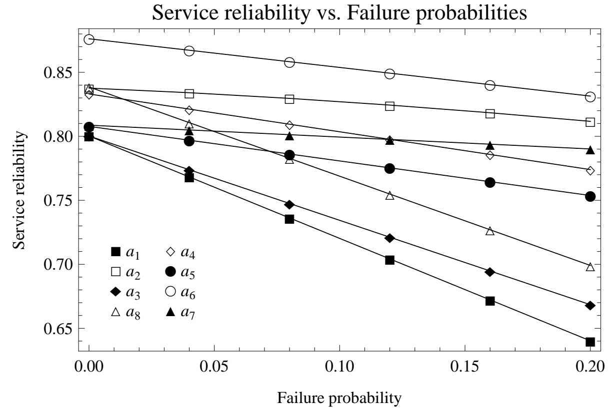

Fig. 15. Sensitivity to failure probabilities.

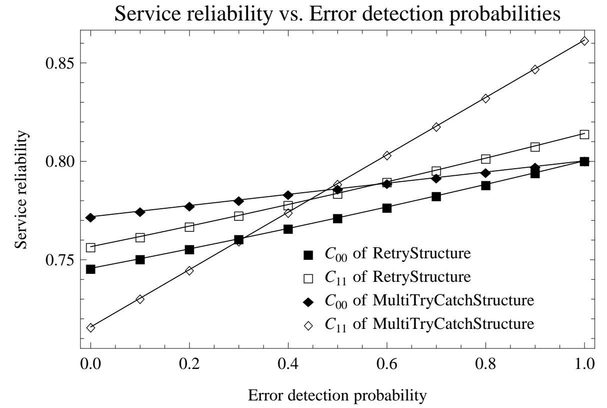

Fig. 16. Sensitivity to error detection probabilities.

Third, we conducted a sensitivity analysis modifying error detection probabilities of the two FTSs (Fig. 16). The reliability of the reporting service is most sensitivity to the element c_{11} of the error detection matrix of the *MultiTryCatchStructure* (i.e., the probability to detect correctly failures of *InfoFromLogFailure* (F_6) from the service *getReleasedDocumentInfoFromLogs*) because its corresponding curve has the steepest slope. This information may be valuable to the software

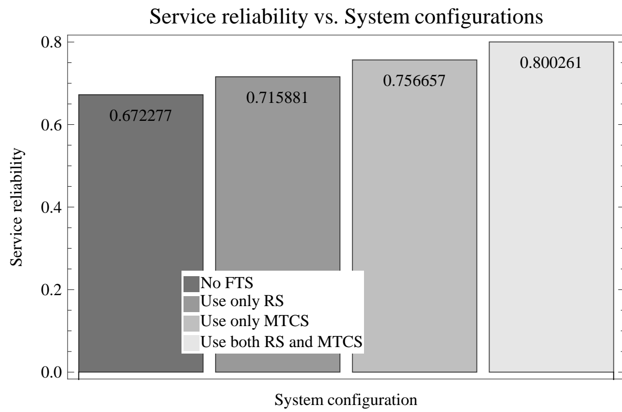

Fig. 17. Reliability comparison between different variants.

architect when considering putting more development effort in order to improve the correct error detections of the FTSs in the system.

Fourth, we conducted an analysis of how the predicted reliability of the reporting service varies for different fault tolerance variants. These variants include a configuration without the FTSs (*No FTS*), a configuration using only the *RetryStructure* (*Use only RS*), a configuration using only the *MultiTryCatchStructure* (*Use only MTCS*) and a configuration using both the FTSs (*Use both RS and MTCS*) (Fig. 17). The variant *Use both RS and MTCS* is predicted as being the most reliable. Comparing between the variant *Use Only RS* and the variant *Use Only MTCS* shows that using the *MultiTryCatchStructure* brings higher reliability impact than using the *RetryStructure* in this case. From the result of this type of analysis, the software architect can assess the impact on the system reliability of different fault tolerance variants and hence can decide whether the additional costs for introducing FTSs, increasing the number of retry times in a *RetryStructure*, adding replicated instances in a *MultiTryCatchStructure*... are justified.

With this type of analysis, it is also possible to see the ability to reuse modeling parts of our approach for evaluating the reliability impacts of fault tolerance variants or system configurations. For the variant *Use Only MTCS*, only a single modification to the *RetryStructure* is necessary (namely, setting the *retryCount* of the structure to 0 to disable the structure). For the variant *Use Only RS*, also only a single modification to the *MultiTryCatchStructure* is necessary (namely, setting the value 1 to all the elements of the column 0 and the value 0 to all the elements of the other columns of the error detection matrix for the *MultiTryCatchPart 1* to disable the structure). For the variant *No FTS*, the two above modifications are included.

VI. Assumptions and Limitations

Similar to the approaches of Cheung, Reussner et al. and Sharma et al. [4], [11], [12], we assume the components fail independently and a component failure leads to a system failure (i.e. a stopping failure model). This means that the error propagation impact is neglected. For the analysis of the influence of error propagation in reliability prediction of component-based software systems with different execution models (including sequential executions, parallel executions, fault tolerance executions), we refer to the approach of Pham et al. [9].

Our approach assumes that for sequential executions, control transitions between components have the Markov property. This means that operational and failure behaviors of a component are independent of its past. This Markovian assumption limits the applicability of our approach. However, many real-life applications have been proved to satisfy this Markovian assumption [11]. Our approach can be adapted to any higher order Markov chains to increase the applicability scope. We confirm this because the problem of Markovian assumption was treated deeply by Goseva et al. [7]. In their paper, the authors point out that a higher order Markov chain (i.e. the next execution step depends not only on the last step but also on the previous n steps) can be mapped into a first order Markov chain.

Another assumption lies in the estimation of failure probabilities for internal activities and error detection matrices for FTSs. No methodology is always valid to deal with the problem. Most of the approaches are based on setting up tests to achieve a statistically significant amount of measurement which the estimation can be based on [23]. Besides, component reuse may allow exploiting the historical data which the estimation can be based on. In early design phases, the estimation can be based on the available specification and design documents of the system [14]. Similarly, the estimation of usage profile can be based on historical data from similar products or on high level information about software architecture and usage obtained from specification and design documents in early phases of software development. In the late phases of the software development, when testing or field data become available, the estimation can be based on the execution traces obtained using profilers and test coverage tools.

VII. Conclusion

In this paper, we presented our extended model for the flexible modeling of software fault tolerance techniques in component-based systems. Our approach allows defining explicitly reliability-relevant behavioral aspects of software fault tolerance techniques and analyzing their impact on the overall reliability of component-based systems. In order to apply our approach, component developers create component reliability specifications, software architects create system reliability models which are transformed automatically to discrete time Markov chains to give reliability predictions and sensitivity analyses. Via a case study, we demonstrated the applicability of our approach, in particular, its ability to support design decisions in early development stages. This can help to make a system more reliable in a cost-effective way because potentially high costs for late life-cycle fixings of a system can be avoided.

We plan to completely integrate with the approach of Pham et al. [9] and to extend with the more complex error propagation for concurrent executions, to include more software FTSs and to validate further our approach.

Acknowledgments

This research was supported 322 FIVE-JAIST program.

References

[1] L. Pullum, *Software fault tolerance techniques and implementation*. Artech House, 2001.

[2] V. Cortellessa, H. Singh, and B. Cukic, "Early reliability assessment of uml based software models," in *Proceedings of the 3rd international workshop on Software and performance*. Rome, Italy: ACM, 2002, pp. 302–309.

[3] K. Goseva-Popstojanova, A. Hassan, A. Guedem, W. Abdelmoez, D. Nassar, H. Ammar, and A. Mili, "Architectural-level risk analysis using uml," *Software Engineering, IEEE Transactions on*, vol. 29, no. 10, pp. 946 – 960, 2003.

[4] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo, "Reliability prediction for component-based software architectures," *J. Syst. Softw.*, vol. 66, no. 3, pp. 241–252, 2003.

[5] V. S. Sharma and K. S. Trivedi, "Reliability and performance of component based software systems with restarts, retries, reboots and repairs," in *Proceedings of the 17th International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2006, pp. 299–310.

[6] W.-L. Wang, D. Pan, and M.-H. Chen, "Architecture-based software reliability modeling," *J. Syst. Softw.*, vol. 79, no. 1, pp. 132–146, 2006.

[7] K. Goseva-Popstojanova and K. S. Trivedi, "Architecture-based approaches to software reliability prediction," *Computers and Mathematics with Applications*, vol. 46, no. 7, pp. 1023–1036, 2003.

[8] A. Immonen and E. Niemel, "Survey of reliability and availability prediction methods from the viewpoint of software architecture," *Software and Systems Modeling*, vol. 7, no. 1, pp. 49–65, 2008.

[9] T.-T. Pham and X. Défago, "Reliability prediction for component-based systems: Incorporating error propagation analysis and different execution models," in *Proceedings of 12th International Conference on Quality Software (QSIC12).* in Xian, China: IEEE Computer Society, 2012, pp. 106–115.

[10] S. S. Gokhale, "Architecture-based software reliability analysis: Overview and limitations," *IEEE Trans. Dependable Secur. Comput.*, vol. 4, no. 1, pp. 32–40, 2007.

[11] R. C. Cheung, "A user-oriented software reliability model," *IEEE Trans. Softw. Eng.*, vol. 6, no. 2, pp. 118–125, 1980.

[12] V. S. Sharma and K. S. Trivedi, "Quantifying software performance, reliability and security: An architecture-based approach," *J. Syst. Softw.*, vol. 80, no. 4, pp. 493–509, 2007.

[13] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Parameterized reliability prediction for component-based software architectures," in *Research into Practice Reality and Gaps*, ser. Lecture Notes in Computer Science, G. Heineman, J. Kofron, and F. Plasil, Eds. Springer Berlin / Heidelberg, 2010, vol. 6093, pp. 36–51.

[14] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik, "Early prediction of software component reliability," in *Proceedings of the 30th international conference on Software engineering*. Leipzig, Germany: ACM, 2008, pp. 111–120.

[15] Z. Zheng and M. R. Lyu, "Collaborative reliability prediction of service-oriented systems," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. Cape Town, South Africa: ACM, 2010, pp. 35–44.

[16] A. Mohamed and M. Zulkernine, "On failure propagation in component-based software systems," in *Proceedings of the 2008 The Eighth International Conference on Quality Software*. IEEE Computer Society, 2008, pp. 402–411.

[17] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, "Reliability analysis of component-based systems with multiple failure modes component-based software engineering," ser. Lecture Notes in Computer Science, L. Grunske, R. Reussner, and F. Plasil, Eds. Springer Berlin / Heidelberg, 2010, vol. 6092, pp. 1–20.

[18] F. Brosch, B. Buhnova, H. Koziolek, and R. Reussner, "Reliability prediction for fault-tolerant software architectures," in *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*. Boulder, Colorado, USA: ACM, 2011, pp. 75–84.

[19] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, 2004.

[20] S. Bernardi, M. Jos, and D. C. Petriu, "A dependability profile within marte," *Softw. Syst. Model.*, vol. 10, no. 3, pp. 313–336, 2011.

[21] S. Shrivastava and N. R. Project, *Reliable computer systems: collected papers of the Newcastle Reliability Project*. Springer-Verlag, 1985.

[22] K. S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications, 2nd Edition*, 2nd ed. Wiley-Interscience, 2001.

[23] M. Lyu, *Handbook of software reliability engineering*. IEEE Computer Society Press, 1996.