Reliability Prediction for Component-based Software Systems with Architectural-level Fault Tolerance Mechanisms

Thanh-Trung Pham, 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*—This paper extends the core model of a recent component-based reliability prediction approach to offer an explicit and flexible definition of reliability-relevant behavioral aspects (i.e. error detection and error handling) of software fault tolerance mechanisms, and an efficient evaluation of their reliability impact in the dependence of the whole system architecture and usage profile. Our approach is validated with the reporting service of a document exchange server, by modeling the reliability, conducting a reliability prediction and sensitivity analyses, and demonstrating its ability to support design decisions.**

***Index Terms*—component-based reliability prediction, software fault tolerance mechanisms, error detection, error handling.**

I. Introduction

Reliability is one of the most important quality attributes of a software system. To improve the system reliability, fault tolerance mechanisms (FTMs) are often used. FTMs provide the ability to mask faults in systems, prevent them from leading to failures, and can be applied on different abstraction levels (e.g. source code level, architecture level) [1]. Analyzing the impact of architectural-level FTMs on the reliability of a component-based system is a challenge because: (1) FTMs can be employed in different parts of a system architecture, (2) usually, in a system architecture, there are multiple points which can be changed to create architecture variants, e.g. substituting components with more reliable variants, running components concurrently to improve performance, and (3) besides the reliability of its components, the system reliability depends on the system architecture and usage profile [2].

Existing reliability prediction approaches for component-based systems often do not allow modeling FTMs (e.g. [3]–[5]) or have limited expressiveness of FTMs (e.g. [6], [7]). These approaches lack flexible and explicit expressiveness of how error detection and error handling of FTMs influence the control and data flow within components. As a result, these approaches are limited in combining modeling FTMs with modeling the system architecture and the usage profile.

Further approaches provide more detailed analysis of individual FTMs (e.g. [8]–[10]). But these so-called non-architectural models do not reflect the system architecture and the usage profile, and therefore are not suitable when evaluating architecture variants under varying usage profiles.

Contribution: In this paper, we extend the core model (i.e. fundamental modeling steps and basic modeling elements) of our former work [11] to offer an explicit and flexible definition of reliability-relevant behavioral aspects (i.e. error detection and error handling) of software FTMs, and an efficient evaluation of their reliability impact in the dependence of the whole system architecture and usage profile.

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

II. Related Work

Our approach belongs to the field of architecture-based software reliability modeling and prediction, and is related to approaches on reliability modeling for FTMs.

The field has been surveyed by several authors [12]–[14]. One of the first approaches is Cheung's approach [2] that uses Markov chains. Recent work extends Cheung's work to combine reliability analysis with performance analysis [15], and to support compositionality [5], but does not consider FTMs. Further approaches such as Cheung et al. [16] focusing on the reliability of individual components, Zheng et al. [17] aiming at service-oriented systems, Cortellessa et al. [3] and Goseva et al. [4] applying UML modeling language, also do not consider FTMs.

Some approaches consider explicitly error propagation to relax the assumption that a component failure immediately leads a system failure [18]–[21]. To model the possibility of propagating component failures, they introduce error propagation probabilities. The complement of these probabilities can be used to express the possibility of masking component failures. However, FTMs (with error detection and error handling) cannot be considered explicitly by these approaches.

Several approaches take into consideration FTMs. Sharma et al. [6] allow modeling component restarts and component retries. Wang et al. [7] support different architectural styles including fault tolerance architectural style. However, these approaches do not consider the influences of both error detection and error handling of FTMs on the control and data flow within components. Brosch et al. [22] offer a flexible way to include FTMs but do not consider the influences

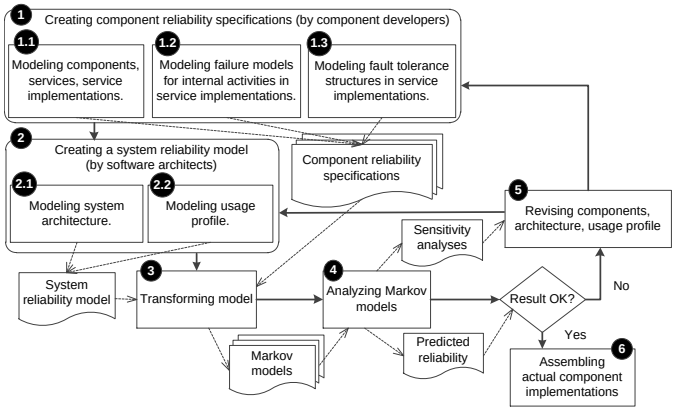

Fig. 1. Component-based reliability prediction.

of error detection of FTMs on the control and data flow within components. The same holds for our former work [11] which incorporates error propagation analysis for multiple execution models including primary-backup fault tolerance executions. Ignoring the influences of either error detection or error handling of FTMs on the control and data flow within components can lead to incorrect prediction results when the behaviors of FTMs deviate from the specific cases mentioned by the authors.

A great deal of past research effort focuses on reliability modeling of individual FTMs. Dugan et al. [8] aim at a combined consideration of hardware and software failures for distributed recovery blocks (DRB), N-version programming (NVP), and N self-checking programming (NSCP) through fault tree techniques and Markov processes. Kanoun et al. [10] evaluate recovery blocks and NVP using generalized stochastic Petri nets. Gokhale et al. [9] use simulation instead of analysis to evaluate DRB, NVP, and NSCP. Their so-called non-architectural models do not reflect the system architecture and the usage profile. Therefore, although these approaches provide more detailed analysis of individual FTMs, they are limited in their application scope to system fragments rather than the whole system architecture (usually composed of different structures) and not suitable when evaluating architecture variants under varying usage profiles.

III. Component-Based Reliability Prediction

Fig. 1 shows six main steps in our approach. In Step 1, component developers create component reliability specifications. Component developers model components, services and service implementations (Step 1.1), and then failure models (i.e. different failure types with their occurrence probabilities) for internal activities in service implementations (Step 1.2). Component developers can introduce different fault tolerance structures (FTSs) in service implementations, e.g. *RetryStructures* or *MultiTryCatchStructures* (Step 1.3). FTSs support different configurations, e.g. 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. Software architects model the system architecture (Step 2.1) and then the usage profile (Step 2.2). Step 1 and 2 are supported by our reliability modeling schema including all necessary modeling elements (Section IV-B and IV-C).

In Step 3, the system reliability model, combined with the component reliability specifications, is transformed automatically into Markov models. In Step 4, by analyzing the Markov models, a reliability prediction and sensitivity analyses can be deduced. To support Step 3 and 4, we provide a reliability prediction tool whose transformation for reliability prediction is explained in Section IV-D.

If the predicted reliability does not meet the reliability requirement, Step 5 is performed. Otherwise, Step 6 is performed. In Step 5, there are several possible options: component developers can revise the components, e.g. changing the configurations of FTSs; software architects can revise the system architecture and the usage profile, e.g. trying different system architecture configurations, replacing some key components with more reliable variants, or adjusting the usage profile appropriately. Sensitivity analyses can be used as a guideline for these options, e.g. to identify the most critical parts of the system architecture which should receive special attention during revising. In Step 6, the modeled system is deemed to meet the reliability requirement, and software architects assemble the actual component implementations following the system architecture model.

IV. Reliability Modeling

A. Basic Concepts

According to Avizienis et al. [23], 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 describe in detail the principle of FTMs. A FTM 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 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, to model and predict better the reliability of component-based systems with architectural-level FTMs, it is necessary to support multiple failure types of a component service and different failure types of different component services, and to consider both the influences of error detection and error handling of FTMs on the control and data flow within components.

In the next section, we introduce our reliability modeling schema for describing reliability-relevant characteristics of

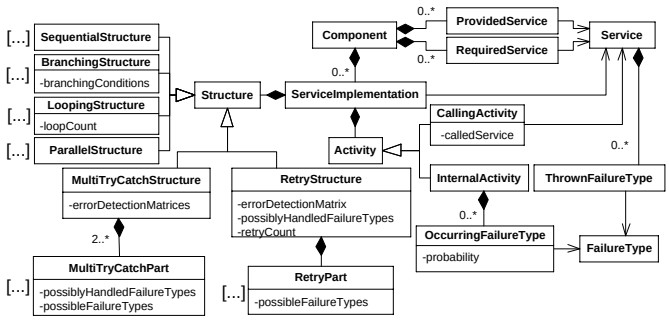

Fig. 2. Modeling elements for component reliability specifications.

component-based systems. With regard to our specific purposes, our schema is more suitable than UML extended with MARTE-DAM profile [24]. This profile focuses on modeling rather than prediction. Its authors do not propose a transformation for prediction for the general case. According to us, the complexity and the semantic ambiguities of UML make it hard to provide an automated transformation from UML to analysis models. On the contrary, our schema uses concepts needed for the reliability modeling and prediction, and therefore our approach can support an automated transformation for reliability prediction for the general case.

B. Component Reliability Specifications

1) Components, services and service implementations: Fig. 2 shows an extract of our reliability modeling schema with modeling elements for component reliability specifications. 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*.

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

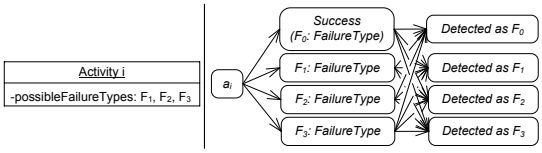

Fig. 3. Error detection semantics for an activity example.

structures[1], 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 Models: In Step 1.2, component developers model failure models (i.e. different failure types with their occurrence probabilities) for internal activities of service implementations via an association between *InternalActivity* and *FailureType*. Different techniques such as fault injection, statistic testing, or growth reliability modeling can be used to determine these probabilities [12], [16].

3) Fault Tolerance Structures:

a) Error detection: In FTMs, correct error detection is the prerequisite condition 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, corresponding to the correct service delivery). To provide error handling for these failure types, it is necessary to detect them correctly. From that, for each F_i, fraction c_{ij} of being detected as F_j 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 F_0, F_1, F_2 and F_3, respectively. The elements of row 0 (except c_{00}) correspond to false alarms. The elements of column 0 (except 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 is service re-execution. A *RetryStructure* is taking ideas from this technique. The structure contains a single *RetryPart* which, in turn, can contain different activity types, 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.

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

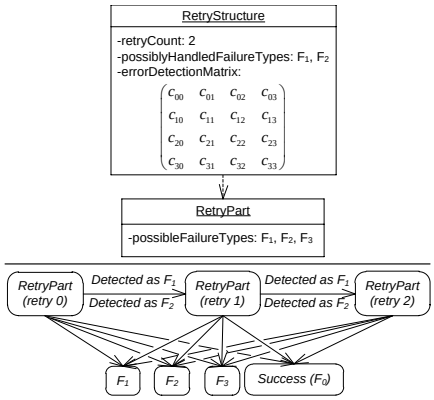

Fig. 4. Semantics for a *RetryStructure* example.

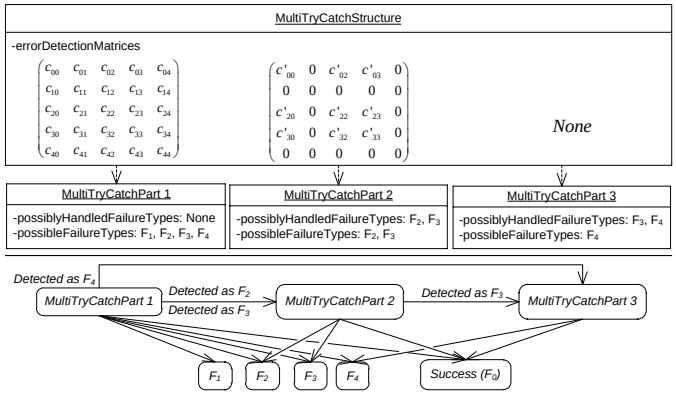

Fig. 5. Semantics for a *MultiTryCatchStructure* example.

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 handled 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 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.

c) MultiTryCatchStructure: A *MultiTryCatchStructure* is taking ideas from the exception handling in object-oriented programming. The structure consists of two or more *MultiTryCatchParts*. Each *MultiTryCatchPart* can contain different activity types, 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 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 F_2 and F_3 lead to *MultiTryCatchPart 2*, the failure types of *MultiTryCatchPart 1* that are detected as 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 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 impossible failure types of *MultiTryCatchPart 2*:

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

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*. Notice that error detection cannot prevent failures and it should be followed by an error handling, therefore, in this case, an error detection matrix for *MultiTryCatchPart 3* is not required because there is no *MultiTryCatchPart* to handle failures of *MultiTryCatchPart 3*.

Remark: FTSs can be employed in different parts of the system architecture and are quite flexible to model FTMs because their inner parts (*RetryPart*, *MultiTryCatchParts*) are able to contain different activity types, structure types, and even nested FTSs, e.g. if a *RetryPart* or a *MultiTryCatchPart* contains a *CallingActivity*, errors from the provided service of the called component can also be handled.

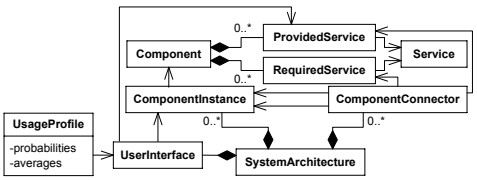

Fig. 6. Modeling elements for system reliability models.

C. System Reliability Models

1) System Architecture: Fig. 6 shows an extract of our reliability modeling schema with modeling elements for system reliability models. Software architects model system architecture via modeling element *SystemArchitecture*. Software architects create component instances (*ComponentInstance*) and assemble them through component connectors (*ComponentConnector*) 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.

D. Transformation for Reliability Prediction

Our transformation[2] is to derive the reliability for the service provided to users via the user interface from the system reliability model and the component reliability specifications. It is implemented in our reliability prediction tool in order to offer an automated transformation for reliability prediction.

The transformation starts with the service implementation of the service provided to users. By design, in our reliability modeling schema: (1) a service implementation can contain a structure of any structure types or an activity of any activity types, (2) a structure type can contain structures of any structure types and activities of activity types, and (3) a calling activity is actually a reference to another service implementation. Therefore, the transformation is essentially a recursive procedure applied for structures and internal activities.

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

For each structure, the transformation transforms it into an equivalent *ia*.

1) Sequential Structure: With a sequential structure as in Fig. 7a, its equivalent *ia* has:

$$sp = fp_0 = \prod_{i=1}^{n} sp(a_i) \tag{1}$$

and for $1 \leq j \leq m$

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

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

- Activity a_1 fails with failure type j: $fp_j(a_1)$.
- Activity a_1 succeeds, activity a_2 fails with failure type j: $sp(a_1)fp_j(a_2)$.
- ...
- Activities a_1, a_2, ..., a_{n-1} succeed, activity a_n fails with failure type j: $\left(\prod_{k=1}^{n-1} sp(a_k) \right) fp_j(a_n)$.

2) Branching Structure: For a branching structure as in Fig. 7b, its equivalent *ia* has:

$$sp = fp_0 = \sum_{i=1}^{n} p(bc_i) sp(a_i) \tag{3}$$

and for $1 \leq j \leq m$

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

where $p(bc_i)$ is the probability of the branching condition bc_i which is obtained from the usage profile.

[2]We argue for the correctness of the transformation as "by construction".

3) Looping Structure: For a looping structure as in Fig. 7c, it is can be seen as a sequential structure of activity a_1 appearing lc times, therefore, its equivalent *ia* has:

$$sp = fp_0 = sp(a_1)^{average(lc)} \tag{5}$$

and for $1 \leq j \leq m$

$$fp_j = \sum_{i=1}^{average(lc)} sp(a_1)^{i-1} fp_j(a_1) \tag{6}$$

where $average(lc)$ is the average number of lc which is also obtained from the usage profile.

4) Parallel Structure: For a parallel structure, 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 (e.g. 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: $F_1 \leq F_2 \leq \ldots \leq F_m$. Considering a parallel structure as in Fig. 7d, its equivalent *ia* has:

$$sp = fp_0 = \prod_{i=1}^{n} sp(a_i) \tag{7}$$

and for $1 \leq j \leq m$

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

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

- Activity a_1 fails with failure type j, activities a_2, a_3, ..., a_n do not fail with failure type $l > j$: $fp_j(a_1) \prod_{k=2}^{n} \left(1 - \sum_{l=j+1}^{m} fp_l(a_k) \right)$.
- Activity a_1 fails with failure type $l < j$, activity a_2 fails with failure type j, activities a_3, a_4, ..., a_n do not fail with failure type $l > j$: $\left(1 - \sum_{l=j}^{m} fp_l(a_1) \right) fp_j(a_2) \prod_{k=3}^{n} \left(1 - \sum_{l=j+1}^{m} fp_l(a_k) \right)$.
- ...
- Activities a_1, a_2, ..., a_{n-1} fail with failure type $l < j$, activity a_n fails with failure type j: $\left(\prod_{k=1}^{n-1} \left(1 - \sum_{l=j}^{m} fp_l(a_k) \right) \right) fp_j(a_n)$.

5) RetryStructure and MultiTryCatchStructure: For a *RetryStructure* or a *MultiTryCatchStructure*, the transformation builds a Markov model reflecting all of its possible execution paths and their corresponding probabilities, and then derives the equivalent *ia* from this Markov model.

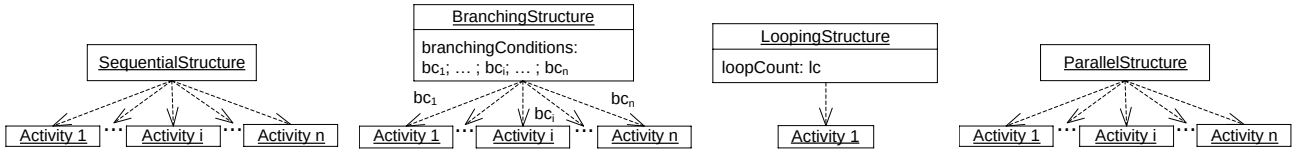

Fig. 7. Example of structures: (a) Sequential structure, (b) Branching structure, (c) Looping structure, and (d) Parallel structure.

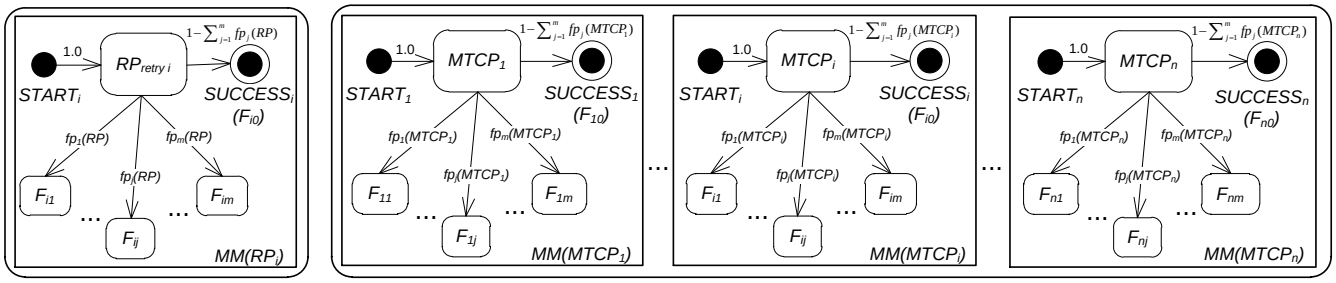

Fig. 8. Markov models: (a) for i-th retry and (b) for *MultiTryCatchParts*.

a) RetryStructure: Considering a *RetryStructure*, let rc be the number of times to retry, $\mathcal{F}_{\mathcal{H}}$ be the set of possibly handled failure types, $\mathbf{C}$ be the error detection matrix represented by $\{c_{rs}\}$ with $r, s = 0, 1, \ldots m$. The *RetryPart* (abbreviated as *RP*) has m different failure types numbered from 1 to m, and failure type j with probability $fp_j\,(RP)$.

The *i-th* retry is represented by a Markov model $MM\,(RP_i)$ as in Fig. 8a. Markov model $MM\,(RP_i)$ 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 $RP_{retry\,i}$ is $fp_0(RP) = 1 - \sum_{j=1}^{m} fp_j(RP)$. With the number of times to retry rc, there are $rc+1$ Markov models $MM\,(RP_i)$, i from 0 to rc. The problem is how to connect these Markov models $MM\,(RP_i)\,(i = 0, 1, \ldots rc)$ 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 transitions:

- From $START$ to $START_0$ with probability 1.
- For $MM\,(RP_{rc})$ (i.e. the Markov model of the last retry), for all $0 \leq j \leq m$: from F_{rcj} to F_j with probability 1.
- For the other Markov models, i.e. $MM\,(RP_i)$ with i from 0 to $rc - 1$, for all $0 \leq j \leq m$: from F_{ij} to $START_{i+1}$ with probability $\sum_{F_k \in \mathcal{F}_{\mathcal{H}}} c_{jk}$; from F_{ij} to F_j with probability $1 - \sum_{F_k \in \mathcal{F}_{\mathcal{H}}} c_{jk}$

As the resulting Markov model is an absorbing Markov chain, the success probability of the equivalent *ia*, which is the probability of reaching $SUCCESS$ from $START$, and the failure probability of failure type j of the equivalent *ia*, which is the probability of reaching F_j from $START$, can be calculated [25].

b) MultiTryCatchStructure: For a *MultiTryCatchStructure*, let n be the number of *MultiTryCatchParts*, $\mathcal{F}_{\mathcal{H}i}$ be the set of possibly handled failure types of *MultiTryCatchPart* i $(i = 1, 2, \ldots n)$, $\mathbf{C}^i$ be the error detection matrix for *MultiTryCatchPart i* which is represented by $\{c^i{}_{rs}\}$ with $r, s = 0, 1, \ldots m$. The *MultiTryCatchPart i* (abbreviated as $MTCP_i$) has *m* different failure types, numbered from 1 to m, failure type j with probability $fp_j\,(MTCP_i)$.

MultiTryCatchParts are represented by Markov models as in Fig. 8b. Markov model $MM\,(MTCP_i)$ has a start state $START_i$, a success state $SUCCESS_i$ (or F_{i0}) and failure states F_{ij} for each failure types j. The probability of reaching F_{i0} from $MTCP_i$ is $fp_0(MTCP_i) = 1 - \sum_{j=1}^{m} fp_j(MTCP_i)$. To connect these Markov models $MM\,(MTCP_i)$ $(i = 1, 2, \ldots n)$ into one Markov model representing the whole structure, 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 transitions:

- From $START$ to $START_0$ with probability 1.
- For $MM\,(MTCP_n)$ (i.e. the Markov model of the last *MultiTryCatchPart*), for all $0 \leq j \leq m$: from F_{nj} to F_j with probability 1.
- For other Markov models, i.e. $MM\,(MTCP_i)$ with i from 1 to $n - 1$, for all $0 \leq j \leq m$: from F_{ij} to $START_x$ with probability $\sum_{F_k \in \mathcal{F}_{\mathcal{H}ix}} c^i_{jk}$ where $\mathcal{F}_{\mathcal{H}ix} = \mathcal{F}_{\mathcal{H}x} - \bigcup_{i<y<x} \mathcal{F}_{\mathcal{H}y}$; from F_{ij} to F_j with probability $1 - \sum_{i<x\leq n} \left(\sum_{F_k \in \mathcal{F}_{\mathcal{H}ix}} c^i_{jk} \right)$.

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

Finally, the reliability of the service provided to users is

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

the success probability of the equivalent *ia* of the service implementation of this service.

***Remark**:* By transforming structures to equivalent *ia(s)*, the transformation no longer needs these structures' information, but can efficiently transform the outer structure using these equivalent *ia(s)*. Because of the recursive nature, the transformation transforms a structure as soon as its inner parts have been transformed into equivalent *ia(s)*, therefore, can reduce the possibility of state-space explosion.

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 [11]. The 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 create the system reliability model of the reporting service as in Fig. 9 using our reliability modeling schema. At the architecture level, the reporting service consists of four components: *ReportingMediator*, *ReportingEngine*, *SourceManager* and *DestinationManager*. 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. 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). 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*). 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

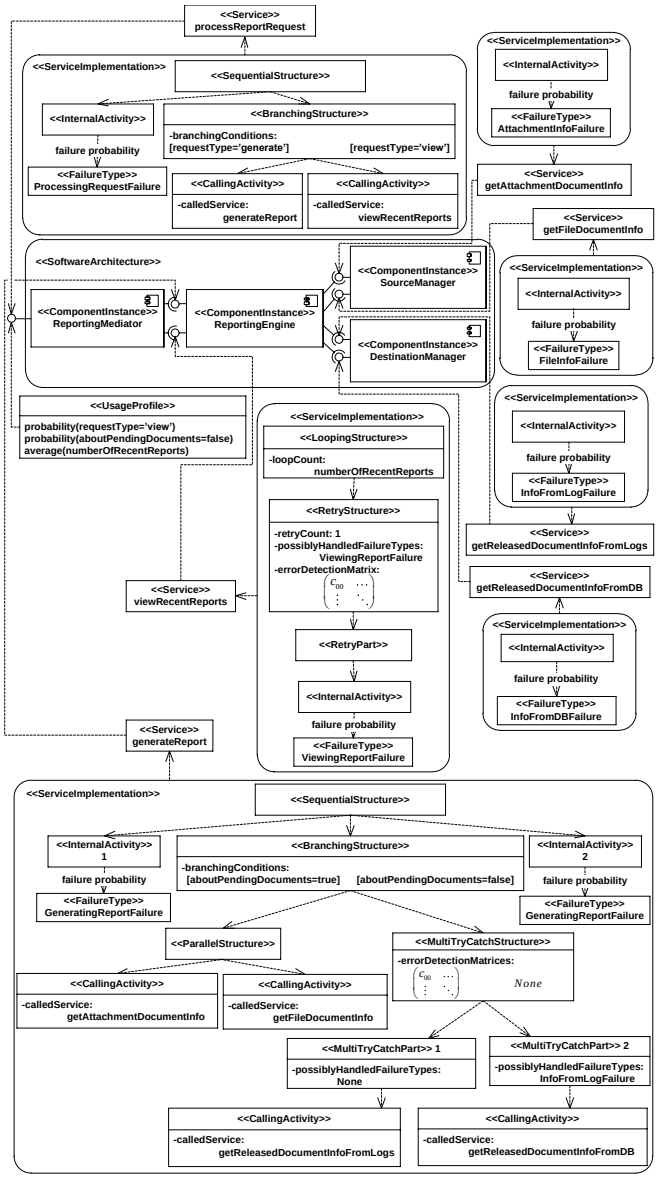

Fig. 9. The system reliability model of the reporting service.

service. For example, a *ProcessingRequestFailure* may occur during processing client requests in service *processReportRequest*; bugs in the code of service *generateReport* may lead to a *GeneratingReportFailure*. Table I shows different failure types and their symbols.

In the system reliability model, there are two FTSs. A *RetryStructure* in the implementation of service *viewRecentReports*. This structure has the ability to retry in case there is a *ViewingReportFailure* (with *retryCount*=1). And a *MultiTryCatchStructure* in the implementation of service *generateReport*. This structure has the ability to handle a *InfoFromLogFailure* of service *getReleasedDocumentInfoFromLogs* by redirecting calls to service *getReleasedDocumentInfoFromDB*.

The current version of 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

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

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 estimated the input parameters of the model. With the estimated input parameters, the system reliability model of the reporting service is complete to be transformed to compute the predicted reliability. The predicted reliability was then compared 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 failure type *j* ($F_j, j = 1, 2, ...7$) of internal activity a_i ($i = 1, 2, \ldots 8$) as: $fp_j(a_i) = f_{ji}/n_i$ 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.

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	$\begin{pmatrix} 1.0 & 0.0 \\ 0.421053 & 0.578947 \end{pmatrix}$

TABLE V
USAGE PROFILE.

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

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.

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.

Notice that different from our former work [11] which set the input parameters for illustrative purpose, in this paper, we estimated the input parameters by using the method above. Therefore, these estimates and the predicted reliability are for the faulty version. This means that our prediction result does not contradict the prediction result of our former work.

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

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

detection probabilities of FTSs. Then, we present the analysis of how the predicted reliability of the reporting service varies for fault tolerance variants.

First, we conducted a sensitivity analysis modifying the usage probabilities (Fig. 10a). The reliability of the reporting service is more sensitive to the portion of report types to generate (*aboutPendingDocuments*=true or false) because its corresponding curve has the steepest slope.

Second, we conducted a sensitivity analysis modifying failure probabilities of the internal activities (Fig. 10b). The reliability of the reporting service is most sensitive to the failure probability of *ProcessingRequestFailure* (F_1) of the internal activity (a_1) of service *processReportRequest* provided by 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 service *viewRecentReports* containing the *RetryStructure*; service *getReleasedDocumentInfoFromLogs* and service *getReleasedDocumentInfoFromDB* in the *MultiTryCatchStructure*. Based on this information, the software architect can decide to put more testing effort into component *ReportingMediator*, to exchange the component with another component from a third party vendor, or run the component redundantly.

Third, we conducted a sensitivity analysis modifying error detection probabilities of the two FTSs (Fig. 10c). The reliability of the reporting service is most sensitivity to the element c_{66} of the error detection matrix of the *MultiTryCatchStructure* (i.e. the probability to detect correctly *InfoFromLogFailure* failures (F_6) from service *getReleasedDocumentInfoFromLogs*) because its corresponding curve has the steepest slope. This information may be valuable to the software architect when considering putting more development effort 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 fault tolerance variants. These variants include: without the FTSs (*No FTSs*), using only the *RetryStructure* (*Only RS*), using only the *MultiTryCatchStructure* (*Only MTCS*) and using both the FTSs (*RS and MTCS*) (Fig. 10d). Variant *RS and MTCS* is predicted as being the most reliable. Comparing between variants *Only RS* and *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 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 variant *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 variant *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 variant *No FTSs*, the two above modifications are included.

VI. Assumptions and Limitations

Our approach assumes that the components fail independently. This means that the error propagation impact is neglected. We refer to our former work [11] for an analysis of error propagation for different execution models in reliability prediction of component-based systems.

Our approach assumes that control transitions between components have the Markov property. This means that operational and failure behaviors of a component are independent of its past. However, our approach can be adapted to any higher order Markov models to consider correlations between component executions because the problem of Markovian assumption in reliability modeling and prediction was treated deeply by Goseva et al. [13].

Another assumption lies in the estimation of failure probabilities for internal activities, error detection matrices for FTSs, and usage profile. 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 [26]. 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 [16]. 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 an explicit and flexible definition of reliability-relevant behavioral aspects (i.e. error detection and error handling) of software FTMs, and an efficient evaluation of their reliability impact in the dependence of the whole system architecture and usage profile. To apply our approach, component developers create component reliability specifications and software architects create a system reliability model using our reliability modeling schema. Then these artifacts are transformed automatically to Markov models for reliability prediction and sensitivity analyses by our reliability prediction tool[3]. Via a case study, we demonstrated our approach's applicability, especially, the ability to support design decisions and reuse modeling parts for evaluating architecture variants under the usage profile.

We plan to completely integrate with our former work [11], to extend with the more complex error propagation for concurrent executions, to include more software FTSs, and to validate further our approach.

[3]Our reliability modeling schema and prediction tool are open source and available at our project website [27].

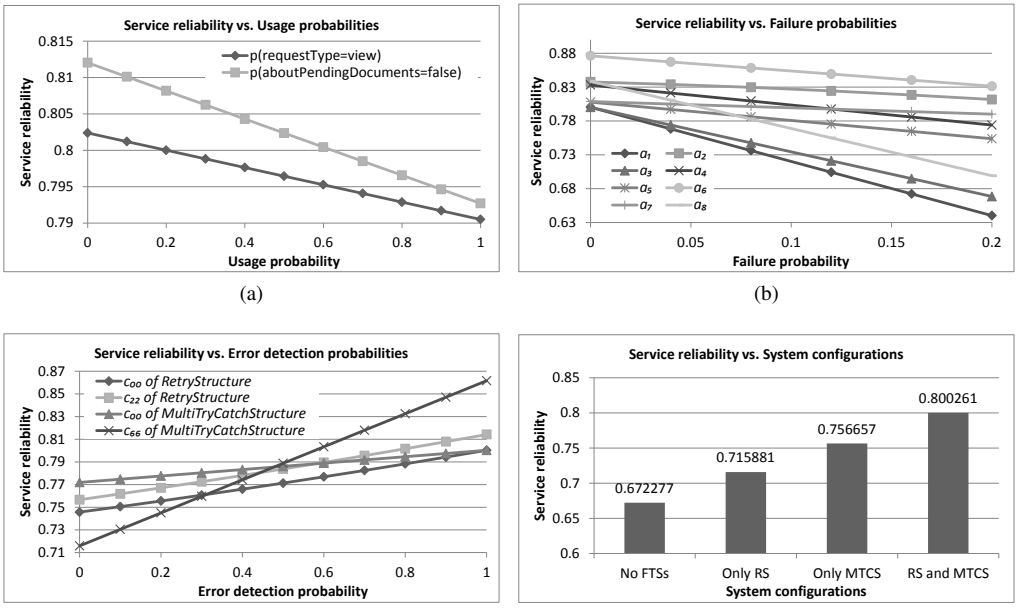

Fig. 10. Sensitivity analyses.

ACKNOWLEDGMENTS

This work was supported by 322 FIVE-JAIST (Vietnam-Japan) program and JSPS KAKENHI Grant Number 23500060. We are grateful to François Bonnet for his invaluable comments that greatly helped improve this manuscript.

REFERENCES

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

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

[3] V. Cortellessa, H. Singh, and B. Cukic, "Early reliability assessment of UML based software models," in *Proc. of the 3rd Intl. Workshop on Softw. and Performance*, 2002, pp. 302–309.

[4] K. Goseva-Popstojanova, A. Hassan, A. Guedem, W. Abdelmoez, D. Nassar, H. Ammar, and A. Mili, "Architectural-level risk analysis using UML," *IEEE Trans. on Softw. Eng.*, vol. 29, pp. 946–960, 2003.

[5] 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.

[6] V. S. Sharma and K. S. Trivedi, "Reliability and performance of component based software systems with restarts, retries, reboots and repairs," in *Proc. of the 17th Intl. Symp. on SRE*, 2006, pp. 299–310.

[7] 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.

[8] J. B. Dugan and M. R. Lyu, "Dependability modeling for fault-tolerant software and systems," in *Software Fault Tolerance*, M. R. Lyu, Ed. John Wiley & Sons, 1995, pp. 109–138.

[9] S. Gokhale, M. Lyu, and K. Trivedi, "Reliability simulation of fault-tolerant software and systems," in *Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS)*, 1997, pp. 167–173.

[10] K. Kanoun, M. Kaaniche, C. Beounes, J.-C. Laprie, and J. Arlat, "Reliability growth of fault-tolerant software," *IEEE Transactions on Reliability*, vol. 42, no. 2, pp. 205–219, Jun 1993.

[11] T.-T. Pham and X. Défago, "Reliability prediction for component-based systems: Incorporating error propagation analysis and different execution models," in *Proc. of 12th Intl. Conf. on Quality Softw.*, 2012, pp. 106–115.

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

[13] 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.

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

[15] 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.

[16] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik, "Early prediction of software component reliability," in *Proc. of the 30th Intl. Conf. on Softw. Engineering*, 2008, pp. 111–120.

[17] Z. Zheng and M. R. Lyu, "Collaborative reliability prediction of service-oriented systems," in *Proc. of 32nd ACM/IEEE Intl. Conf. on Softw. Eng. - Vol. 1*, 2010, pp. 35–44.

[18] P. Popic, D. Desovski, W. Abdelmoez, and B. Cukic, "Error propagation in the reliability analysis of component based systems," in *Proc. of 16th IEEE Intl. Symp. on Softw. Reliability Engineering*, 2005, pp. 53–62.

[19] V. Cortellessa and V. Grassi, "A modeling approach to analyze the impact of error propagation on reliability of component-based systems," in *Proc. of 10th Intl. Conf. on CBSE*, 2007, pp. 140–156.

[20] A. Mohamed and M. Zulkernine, "On failure propagation in component-based software systems," in *Proc. of the 8th Intl. Conf. on Quality Softw.*, 2008, pp. 402–411.

[21] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, "Reliability analysis of component-based systems with multiple failure modes," in *Proc. of 13th Intl. Conf. on CBSE*, 2010, pp. 1–20.

[22] F. Brosch, B. Buhnova, H. Koziolek, and R. Reussner, "Reliability prediction for fault-tolerant software architectures," in *Intl. Conf. on the Quality of Softw. Architectures - QoSA 2011*, 2011, pp. 75–84.

[23] 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.

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

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

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

[27] (2013) Reliability modeling and prediction. [Online]. Available: http://srmp.codeplex.com/