Reliability Prediction for Component-based Software Systems with Architectural-level Fault Tolerance Mechanisms (Extended Version)[*]

Thanh-Trung Pham[1†], François Bonnet[1], Xavier Défago[1,2]
[1] *School of Information Science, JAIST, Nomi, Ishikawa, Japan*
[2] *I3S, UNS, CNRS, Inria, Sophia Antipolis, France*
{thanhtrung.pham, f-bonnet, defago}@jaist.ac.jp

Abstract

Reliability, one of the most important quality attributes of a software system, should be considered early in the development. Software fault tolerance mechanisms (FTMs) are often included in a software system to improve the system reliability. Their reliability impact highly depends on the application context. Existing reliability prediction approaches for component-based software systems either do not support modeling FTMs or have a limited expressiveness of FTMs. In this paper, we present a novel extension built upon 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 FTMs, and an efficient evaluation of their reliability impact in the dependence of the whole system architecture and usage profile. Our approach is validated in two case studies, by modeling the reliability, conducting reliability predictions and sensitivity analyses, and demonstrating its ability to support design decisions.

Keywords: component-based reliability prediction, software fault tolerance mechanisms, error detection, and error handling.

1 Introduction

Techniques for analyzing properties of a software design or 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, one of the most important quality attributes of a software system, can be defined as the probability of failure-free operation in given time span. Software fault tolerance mechanisms (FTMs) are often included in a software system and constitute an important means to improve the system reliability. 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 with exception handling, architecture level with replication) [2]. Analyzing the impact of architectural-level FTMs on the reliability of a component-based software system is a challenge because:

- FTMs can be employed in different parts of a system architecture.

[*]This paper is an extended version of the work originally presented at the 8th International Conference on Availability, Reliability and Security (ARES'13), Regensburg, Germany, September 2013 [1].

[†]Corresponding author: 1-1 Asahidai, Nomi, Ishikawa Prefecture 923-1211, Japan, Tel: +81-761-51-1111

- 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.

- Besides the reliability of its components, the system reliability depends on the system architecture and usage profile (i.e. component services, control flow transitions between them and sequences of component service calls) [3]. For example, under a certain usage profile, if faulty code is never executed, then no failures occurs, and the system is perceived as reliable by its users.

Existing reliability prediction approaches for component-based systems often do not allow modeling FTMs (e.g. [4–6]) or have limited expressiveness of FTMs (e.g. [7, 8]). These approaches lack flexible and explicit expressiveness of how error detection and error handling of FTMs influence the control and data flow within components. For example, an undetected error from a component's provided service leads to no error handling, which in turn influences the control and data flow within component services using this provided service. As a result, these approaches are limited in combining modeling FTMs with modeling the system architecture and usage profile.

Further approaches provide more detailed analysis of individual FTMs (e.g. [9–11]). But these so-called non-architectural models do not reflect the system architecture and usage profile. As a consequence, they are not suitable when analyzing how individual FTMs employed in different parts of a system architecture influence the overall system reliability, especially when evaluating for architecture variants under varying usage profiles.

Contribution: The contribution of this paper is a novel extension built upon the core model (i.e. fundamental modeling steps and basic modeling elements) of our former work [12] 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. Our approach offers a reliability modeling schema with developer-friendly modeling elements (e.g. provided/required services, components, connectors, etc.). We provide a reliability prediction tool for an automated transformation from models based on the schema into Markov models for reliability predictions and sensitivity analyses. We validate our approach in two case studies and demonstrate its applicability in supporting design decisions.

Structure: The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 describes the steps in our approach. Section 4 describes in detail our reliability modeling schema. Section 5 describes the transformation to create Markov models for reliability predictions. Section 6 demonstrates our approach with case studies. Section 7 discusses our assumptions and limitations and Section 8 concludes the paper.

2 Related Work

Our approach belongs to the field of architecture-based software reliability modeling and prediction which treats software systems as a composition of software components. It is related to approaches on architectural-level fault tolerance modeling and reliability modeling of individual FTMs.

The field of *architecture-based software reliability modeling and prediction* has been surveyed by several authors [13–15]. One of the first approaches is Cheung's approach [3] that uses Markov chains. Recent work extends Cheung's work to combine reliability analysis with performance analysis [16], and to support compositionality [6], but does not consider FTMs. Further approaches such as Cheung et al. [17] focusing on the reliability of individual components, Zheng et al. [18] aiming at service-oriented systems, Cortellessa et al. [4] and Goseva et al. [5] applying UML modeling language, also do not consider FTMs.

Several approaches in the field consider explicitly error propagation to relax the assumption that a component failure immediately leads to a system failure [19–22]. 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 their error detection and error handling cannot be considered explicitly by these approaches.

Some approaches step forward and deal with the problem of incorporating *architectural-level FTMs* into architecture-based reliability prediction models. Sharma et al. [7] allow modeling component restarts and component retries. Wang et al. [8] 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. [23] offer a flexible way to include FTMs but do not consider the influences of error detection of FTMs on the control and data flow within components. 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. [9] 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. [11] evaluate recovery blocks and NVP using generalized stochastic Petri nets. Gokhale et al. [10] 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.

Preliminary Work

In [12], we presented a reliability prediction approach for component-based software systems that considers error propagation for different execution models including sequential, parallel, and primary-backup fault tolerance executions. However, our support for fault tolerance modeling was limited to primary-backup FTM, while in this paper, we are able to support modeling large classes of existing FTMs (e.g. exception handling, restart-retry, primary-backup, recovery blocks, etc.).

Furthermore, this paper goes beyond our work published in [1] through extended fault tolerance modeling support for multi-version programming FTM, supports for modeling composite components and looping structures with discrete probability distributions of loop counts, a more extensive validation, and a far more elaborate detailed description and discussion of the approach.

3 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 and provide component functional specifications and component quality specifications (i.e. models). Component functional specifications are sufficient for software architects to assemble components and check their interoperability. However, in order to reason about quality attributes such as reliability, performance, or security of a component-based software architecture, software architects need to use component quality specifications.

In our approach, component developers are required to create component reliability specifications,

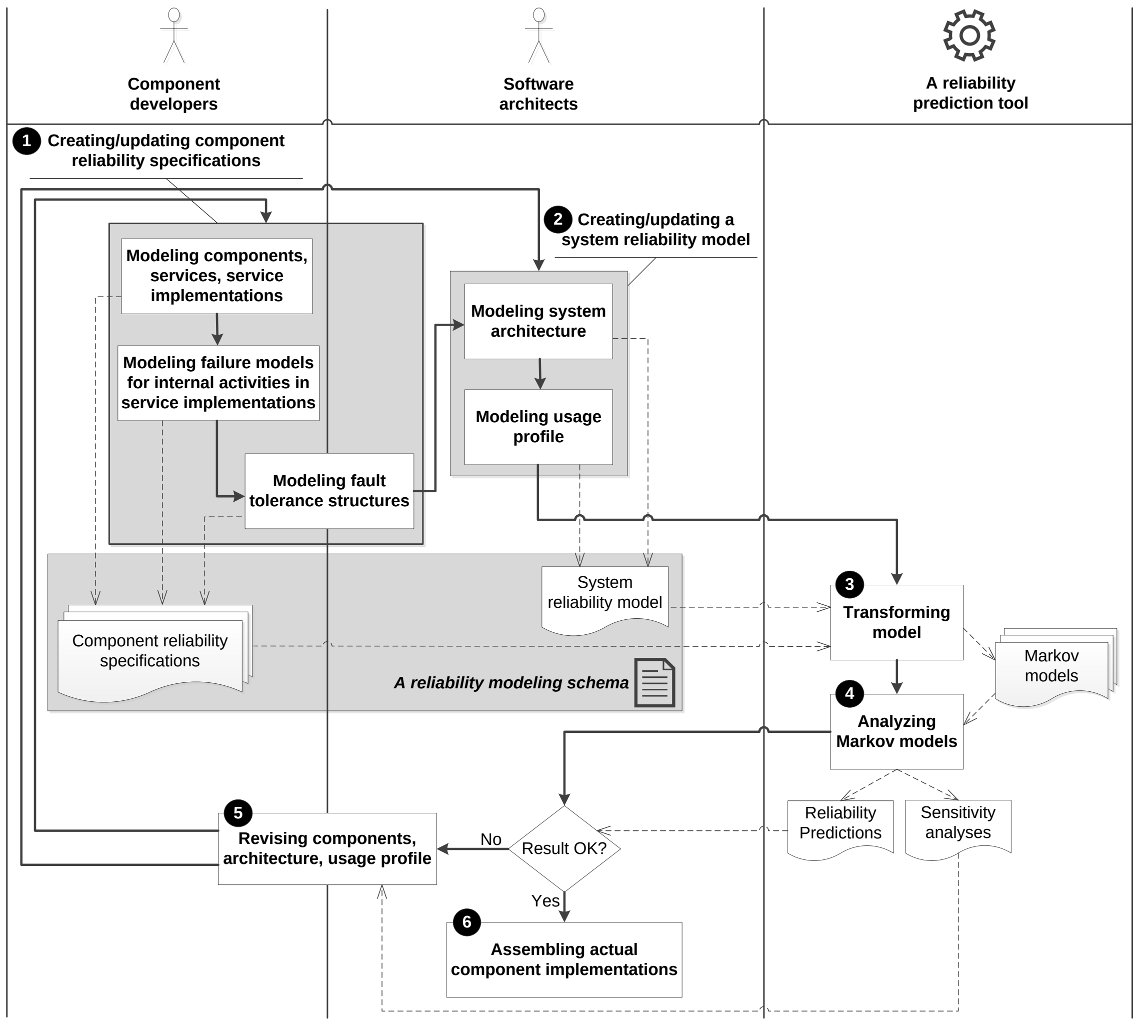

Component developers
Software architects
A reliability prediction tool
1 Creating/updating component reliability specifications
Modeling components, services, service implementations
Modeling failure models for internal activities in service implementations
Modeling fault tolerance structures
2 Creating/updating a system reliability model
Modeling system architecture
Modeling usage profile
System reliability model
Component reliability specifications
A reliability modeling schema
3 Transforming model
Markov models
4 Analyzing Markov models
Reliability Predictions
Sensitivity analyses
Result OK?
No
Yes
5 Revising components, architecture, usage profile
6 Assembling actual component implementations

Figure 1: Component-based reliability prediction.

by describing how provided services of a component call required services in terms of probabilities, frequencies, and parameter values. From that, software architects can create a model of flow and data control throughout the entire architecture for reliability predictions, by simply incorporating these specifications, without referring to component internals.

Fig. 1 shows six steps in our approach. In Step 1, component developers create component reliability specifications. Component developers model components, services and service implementations, and then failure models (i.e. different failure types with their occurrence probabilities) for internal activities in service implementations. Component developers/software architects can include different fault tolerance structures (FTSs), e.g. *RetryStructures*, *MultiTryCatchStructures*, or *MVPStructures* (see Section 4.2.3), either directly into service implementations of already modeled components or as additional components. FTSs support different configurations, e.g. the number of times to retry in a *RetryStructure*, the number of replicated instances for handling certain failure types in a *MultiTryCatchStructure*, or the number of versions executed in parallel in a *MVPStructure*.

In Step 2, software architects create a system reliability model. Software architects model the system architecture and then the usage profile. In Section 4, we introduce our reliability modeling schema that supports creating component reliability specifications and system reliability models.

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 5. With the tool support, sensitivity analyses can also be derived, e.g. by varying reliability-related probabilities of components inside the system architecture to obtain corresponding reliability predictions.

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.

4 Reliability Modeling

4.1 Basic Concepts

According to Avizienis et al. [24], 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. Error handling followed by fault handling together form system recovery. Error handling is to eliminate errors from the system state, e.g. by bringing the system back to a saved state that existed prior to error occurrence. Fault handling is to prevent faults from being activated again, e.g. by either switching in spare components or reassigning tasks among non-failed components. 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 component-based systems. It would have been possible for us to build our approach upon UML. However, by introducing our reliability modeling schema, we avoid the complexity and the semantic ambiguities of UML which make it hard to provide an automated transformation from UML to analysis models. With regard to our specific purposes, our schema is more suitable than UML extended with MARTE-DAM profile [25][1] because our schema is reduced to concepts needed for reliability prediction, and therefore our approach can support an automated transformation for reliability prediction for the general case.

[1]This profile provides a very comprehensive reliability modeling but its authors do not target an automated transformation for reliability prediction for the general case.

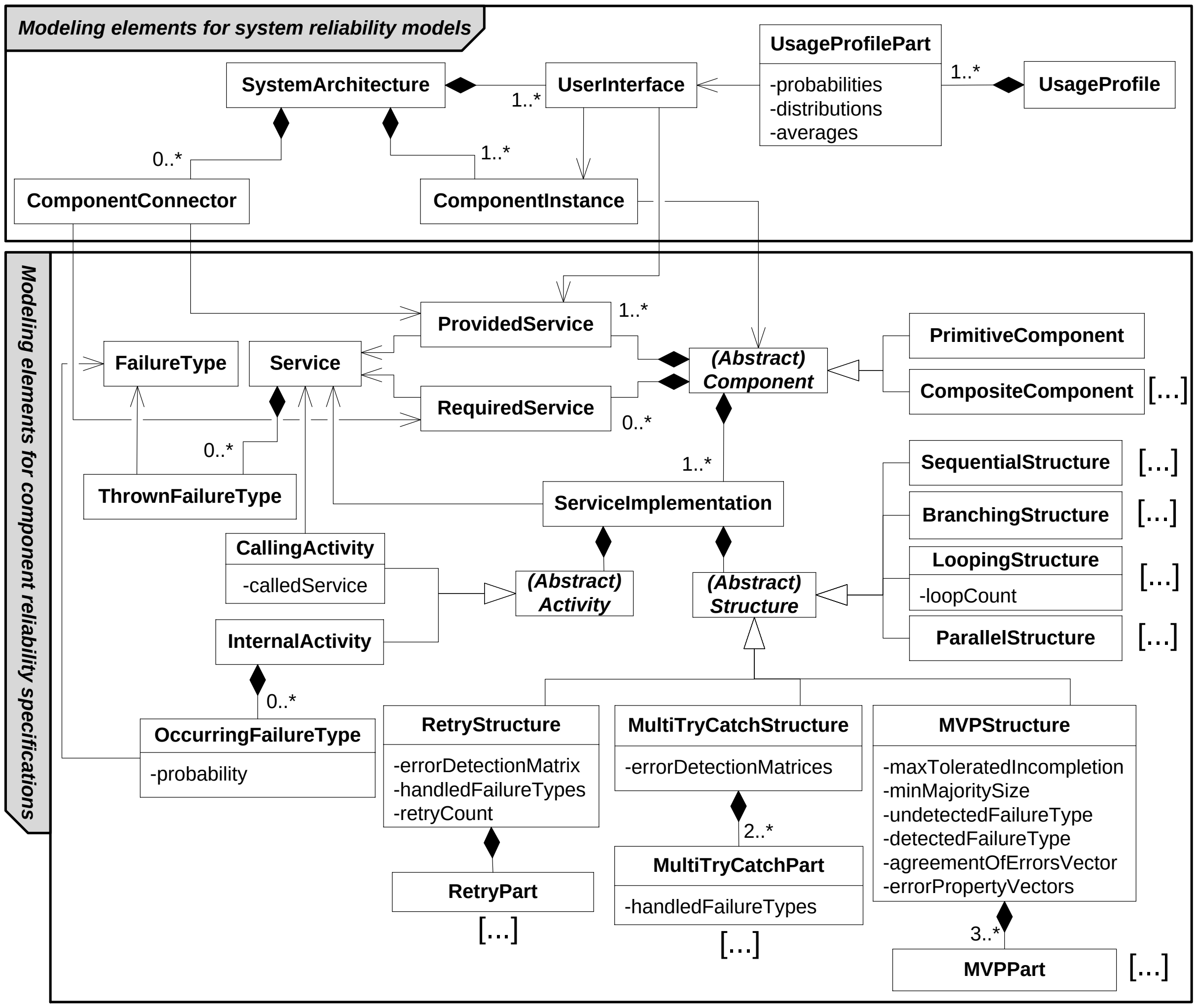


Figure 2: Modeling elements in our reliability modeling schema.

4.2 Component Reliability Specifications

4.2.1 Components, services, and service implementations

In our approach, component developers are required to provide component reliability specifications. Fig. 2 shows an extract of our reliability modeling schema with modeling elements that supports component developers to create component reliability specifications. Component developers model components and services via modeling elements: *Component* and *Service*, respectively. A component can be either a primitive component (*PrimitiveComponent*) or a composite component (*CompositeComponent*) which is hierarchically structured with nested inner components. Components are associated with services via *RequiredService* and *ProvidedService*.

Example 1. *Fig. 3 shows an example of components and services, including seven services (from S_0 to S_6), one composite component (C_8) which contains three nested primitive component (C_5, C_6, and C_7), and four separated primitive components (from C_1 to C_4).*

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

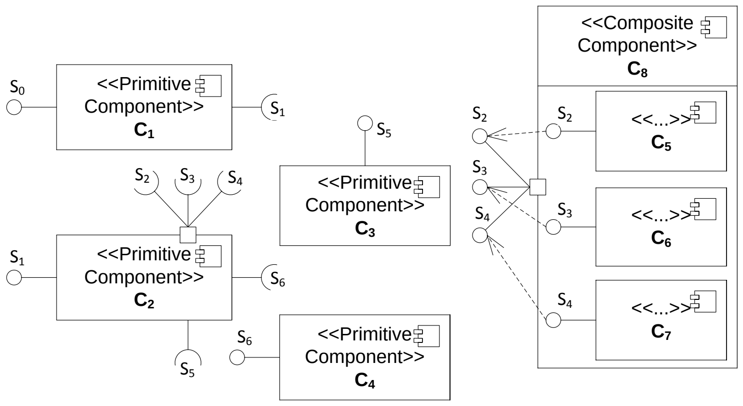


Figure 3: Example of components and services.

provided services of the component is called. Therefore, a component can contain multiple service implementations. A service implementation (*ServiceImplementation*) 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. Four standard types of control flow structures 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.

4.2.2 Failure Models

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 [13, 17].

Example 2. *Fig. 4 shows an example of service implementations. The service implementation* Svc_1 *includes one internal activity. The failure model of the internal activity shows that during the execution of the internal activity, failure type* F_2 *can occur with probability 0.001617.*

The service implementation Svc_2 *contains two internal activities (with their failure models), four calling activities (to call required services:* Svc_3*,* Svc_4*, and* Svc_5*), one branching structure (with branching conditions:* $[Y = true]$ *and* $[Y = false]$*), one looping structure (with loop count: Z).*

4.2.3 Fault Tolerance Structures

Error detection To support modeling FTMs, our reliability modeling schema provides fault tolerance structures (FTSs). In FTMs, correct error detection is the prerequisite condition for correct error han-

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

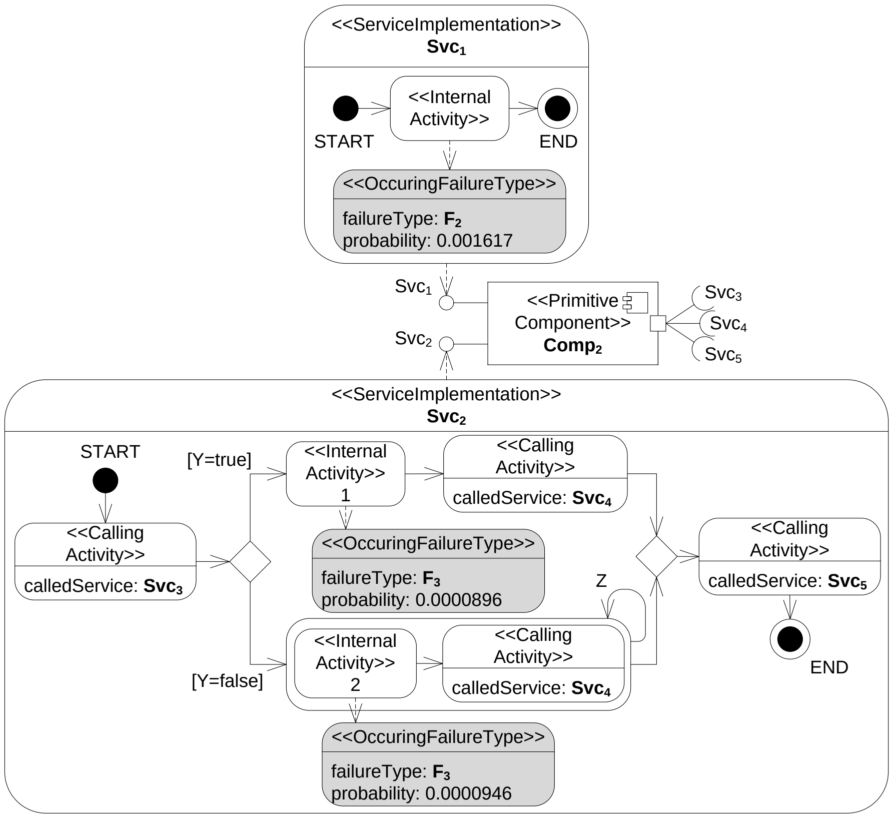


Figure 4: Example of service implementations.

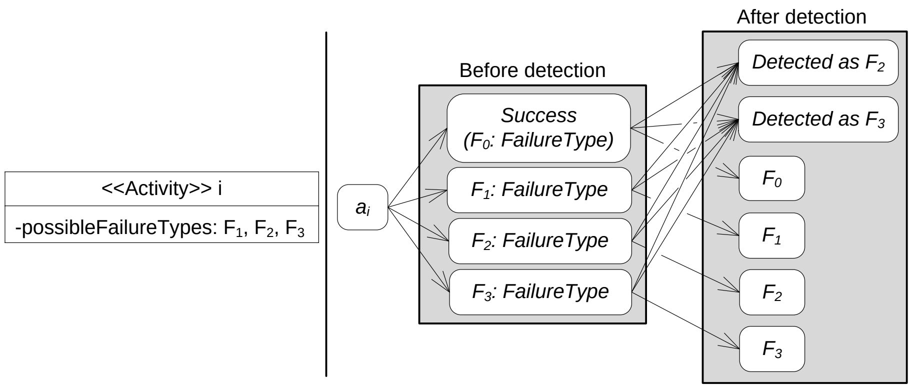


Figure 5: Error detection semantics for an activity example.

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

Example 3. *Fig. 5 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 certain failure types, e.g. F_2 and F_3, it is necessary to detect them correctly. From that, for each F_i with $i \in \{0,1,2,3\}$, fraction c_{ij} of being detected as F_j with $j \in \{2,3\}$ needs to be provided. Therefore, the*

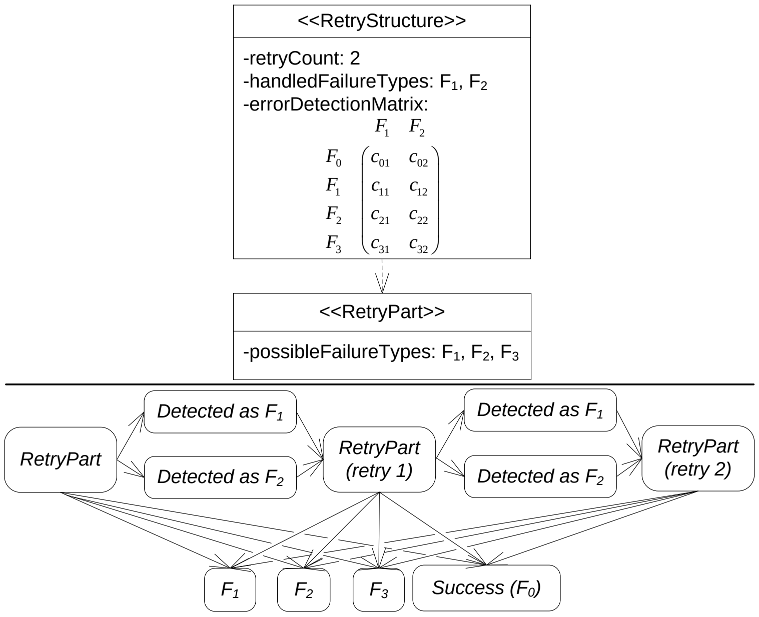


Figure 6: Semantics for a *RetryStructure* example.

error detection can be described by the following matrix:

$$\begin{array}{c} \\ F_0 \\ F_1 \\ F_2 \\ F_3 \end{array} \begin{array}{c} \begin{array}{cc} F_2 & F_3 \end{array} \\ \begin{pmatrix} c_{02} & c_{03} \\ c_{12} & c_{13} \\ c_{22} & c_{23} \\ c_{32} & c_{33} \end{pmatrix} \end{array}, \sum_j c_{ij} \leq 1$$

Elements c_{0j} correspond to false alarms. Elements c_{ij} with $i \neq j$ correspond to false signaling of failure type. In case of perfect error detection, the error detection matrix has $c_{jj} = 1$ and $c_{ij} = 0$ for $i \neq j$.

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.

Example 4. *Fig. 6 shows a* RetryStructure *with a single* RetryPart*. During the execution of the* RetryPart*, failure types F_1, F_2, and F_3 can occur (the field* possibleFailureTypes*). The field* handledFailureTypes *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 *(2 times in this example).*

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 and launch alternative activities.

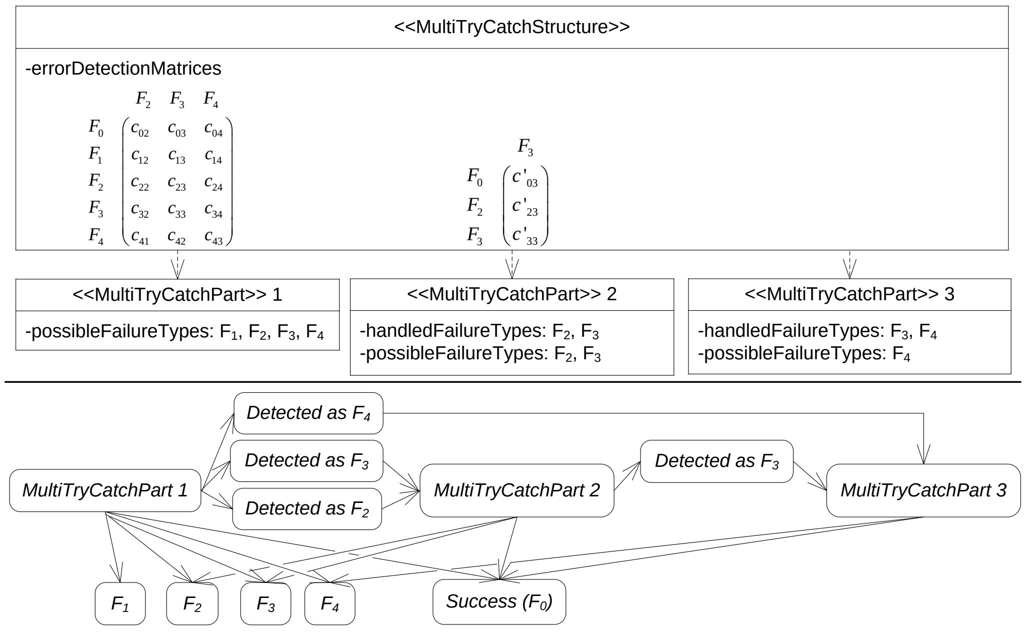

<<MultiTryCatchStructure>>
-errorDetectionMatrices
$\begin{array}{c|ccc} & F_2 & F_3 & F_4 \\ F_0 & c_{02} & c_{03} & c_{04} \\ F_1 & c_{12} & c_{13} & c_{14} \\ F_2 & c_{22} & c_{23} & c_{24} \\ F_3 & c_{32} & c_{33} & c_{34} \\ F_4 & c_{41} & c_{42} & c_{43} \end{array}$
$\begin{array}{c|c} & F_3 \\ F_0 & c'_{03} \\ F_2 & c'_{23} \\ F_3 & c'_{33} \end{array}$
<<MultiTryCatchPart>> 1
-possibleFailureTypes: F_1, F_2, F_3, F_4
<<MultiTryCatchPart>> 2
-handledFailureTypes: F_2, F_3
-possibleFailureTypes: F_2, F_3
<<MultiTryCatchPart>> 3
-handledFailureTypes: F_3, F_4
-possibleFailureTypes: F_4
MultiTryCatchPart 1
Detected as F_4
Detected as F_3
Detected as F_2
MultiTryCatchPart 2
Detected as F_3
MultiTryCatchPart 3
F_1
F_2
F_3
F_4
Success (F_0)

Figure 7: Semantics for a *MultiTryCatchStructure* example.

Example 5. *Fig. 7 shows a* MultiTryCatchStructure *with three* MultiTryCatchPart(s)*. During the execution of* MultiTryCatchPart 1*, failure types F_1, F_2, F_3, and F_4 can occur. Because the field* handledFailureTypes *of* MultiTryCatchPart 2 *includes F_2, F_3 and that of* MultiTryCatchPart 3 *includes 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*.*

Similarly, during the execution of MultiTryCatchPart 2*, possible failure types are F_2 and F_3. Moreover, because the field* handledFailureTypes *of* MultiTryCatchPart 3 *includes F_3 and F_4, only failure types of* MultiTryCatchPart 2 *that are detected as 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*.*

MVPStructure Based on the concept of N-version programing (NVP) with majority voting decision, we build our *MVPStructure*. A *MVPStructure* consists of three or more *MVPParts*. Each *MVPPart* can contain different activity types, structure types, and even a nested *MVPStructure*. Similar to variants (or versions) in NVP, these *MVPParts* are executed in parallel in the same environment: each of them receives identical inputs and each produces its version of the outputs. The outputs are then collected by the structure's majority voter and the results of the majority are assumed to be the correct output used by the system.

The voter has to determine the decision output from a set of results. If there is no agreement of the majority results, the voter signals a detected failure. Otherwise, the voter produces an output which is the result of the agreement (i.e. the consensus). The output of the voter is correct if the agreement is of the majority correct results, otherwise the output of the voter is erroneous. In analogy to NVP, we assume that the *MVPStructure* is not used in the situations that can have multiple distinct correct outputs. The operation of a *MVPStructure* is depicted in Fig. 8.

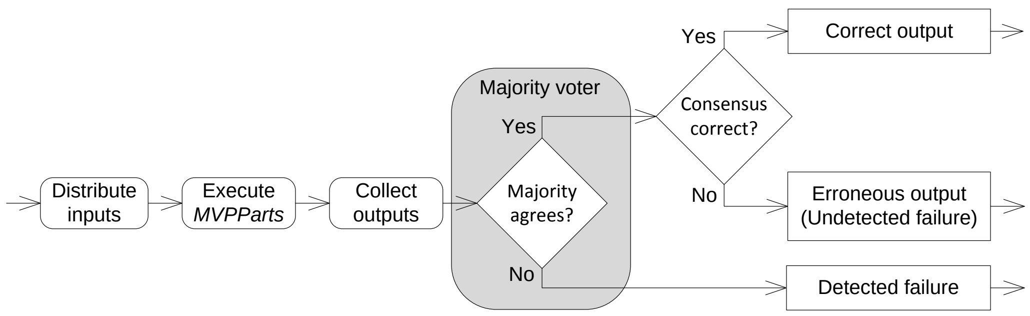


Figure 8: The operation of a *MVPStructure*.

From the viewpoint of the majority voter, it distinguishes the result of a *MVPPart*'s execution in terms of complete execution with an output (correct or erroneous) or incomplete execution. Therefore, given that a *MVPPart* has failed with a given failure type, a fraction of the fact that it has not completed its execution needs to be provided. With all the possible failure types of a *MVPPart*, there is a vector of such fractions, called *errorPropertyVector*. And for all *MVPParts* of a *MVPStructure*, there is a list of *errorPropertyVectors*.

The operation of the *MVPStructure* can also be configured via the following properties:

- *maxToleratedIncompletion*: the maximum number of incomplete executions of *MVPParts* the voter can tolerate.
- *minMajoritySize*: the minimum number of the results of the executions of *MVPParts* required to agree for the voter to produce an output (correct or erroneous).
- *detectedFailureType*: the failure type of detected failures for the voter to signal.
- *undetectedFailureType*: the failure type of erroneous outputs of the structure.

Let $n_{MV} \geq 3$ be the number of *MVPParts* of a *MVPStructure*, *minMS* be the value of *minMajoritySize*, then it is required that $minMS \geq \lceil (n_{MV}+1)/2 \rceil$.

Moreover, when there are at least *minMajoritySize* erroneous results in the set of the results of *MVPParts*' executions, in order to distinguish whether the voter signals a detected failure or produces an erroneous output, a fraction of the fact that there is an agreement of the majority erroneous results also needs to be provided. With all the possible number of erroneous results, there is a vector of such fractions, called *agreementOfErrorsVector*.

Example 6. *Fig. 9 shows a* MVPStructure *with three* MVPParts. *The execution of* MVPPart 1 *can either succeed (F_0) or fail with failure type F_1. The error property vector for* MVPPart 1 *(the first elements of the field* errorPropertyVectors*) shows that given that* MVPPart 1 *has failed with failure type F_1,* MVPPart 1 *has not completed its execution with probability d_1. In case failures of F_1 are content failures (i.e. the content of a system service's output deviates from the correct one), $d_1 = 0$; in case failures of F_1 are late timing failures (i.e. the delivery time of a system service is too late from the correct one), $d_1 = 1$. Similarly, there is an error property vector for each of the remaining* MVPParts.

Based on the set of the results of the MVPParts' *executions, the majority voter of the* MVPStructure *has to determine the decision output. Different possibilities for the set of the results are represented by $(\#CorOut, \#ErrOut, \#IncExe)$ with $\#CorOut + \#ErrOut + \#IncExe = 3$ where $\#CorOut$ is the number of correct outputs, $\#ErrOut$ is the number of erroneous outputs, and $\#IncExe$ is the number of incomplete executions. The voter has been configured to tolerate at most one incomplete execution from* MVPParts

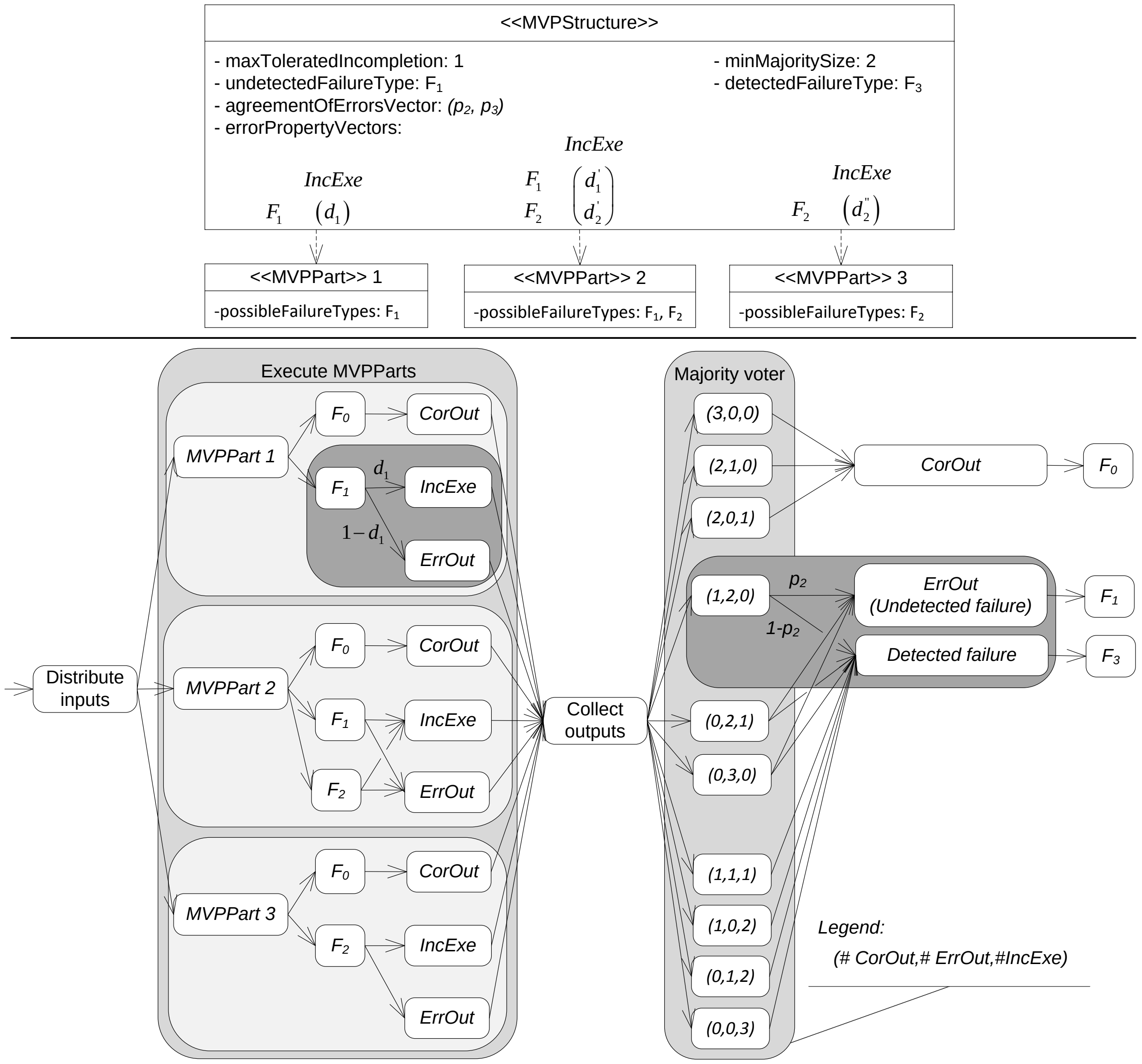

<<MVPStructure>>
- maxToleratedIncompletion: 1
- undetectedFailureType: F1
- agreementOfErrorsVector: (p2, p3)
- errorPropertyVectors:
- minMajoritySize: 2
- detectedFailureType: F3
IncExe
<<MVPPart>> 1
-possibleFailureTypes: F1
<<MVPPart>> 2
-possibleFailureTypes: F1, F2
<<MVPPart>> 3
-possibleFailureTypes: F2
Execute MVPParts
Majority voter
Distribute inputs
MVPPart 1
MVPPart 2
MVPPart 3
CorOut
IncExe
ErrOut
Collect outputs
ErrOut (Undetected failure)
Detected failure
Legend:
(# CorOut,# ErrOut,#IncExe)

Figure 9: Semantics for a *MVPStructure* example.

(the field maxToleratedIncompletion*) and to require at least two results of the executions of* MVPParts *to agree in order to produce an output (correct or erroneous) (the field* minMajoritySize*). Therefore, the voter can determine the decision output for the following possibilities:*

- *Possibilities with #IncExe > 1 cause the voter to signal a detected failure of F_3 (the field* detected-FailureType*).*
- *Possibilities with #IncExe ≤ 1 and #CorOut ≥ 2 make the voter to produce a correct output (F_0).*
- *Possibilities with #IncExe ≤ 1, #CorOut < 2, and #ErrOut < 2 also cause the voter to signal a detected failure of F_3.*

For the remaining possibilities with #IncExe ≤ 1, #CorOut < 2, and #ErrOut ≥ 2, the field agreementOfErrorsVector *shows that when there are two erroneous outputs, with probability p_2 there is an* *agreement of the majority erroneous outputs, and when there are three erroneous outputs, with probability p_3 there is an agreement of the majority erroneous outputs. In case the executions of* MVPParts *always produce distinct erroneous outputs when they fail, $p_2 = p_3 = 0$; in case the output domain of the executions of* MVPParts *is a boolean domain (i.e. true/false), $p_2 = p_3 = 1$. Therefore, the voter can determine the decision output for the remaining possibilities:*

- *For possibilities with #IncExe ≤ 1, #CorOut < 2, and #ErrOut $= 2$, the voter produces an erroneous output of F_1 (the field* undetectedFailureType*) with probability p_2, or signal a detected failure of F_3 with probability $1 - p_2$.*
- *Similarly, for possibilities with #IncExe ≤ 1, #CorOut < 2, and #ErrOut $= 3$, the voter produces an erroneous output of F_1 with probability p_3, or signal a detected failure of F_3 with probability $1 - p_3$.*

Remark FTSs support enhanced fault tolerance expressiveness in several aspects, including different recovery behaviors in response to occurrences of failures, as well as multi-type and multi-stage recovery behaviors. They can be employed in different parts of the system architecture and are quite flexible to model FTMs because their inner parts (*RetryPart*, *MultiTryCatchParts*, and *MVPParts*) are able to contain different activity types, structure types, and even nested FTSs. They allow modeling different classes of existing FTMs, including exception handling, restart-retry, primary-backup, and recovery blocks, N-version programming, and consensus recovery blocks. If a *RetryPart*, *MultiTryCatchPart*, or *MVPPart* contains a *CallingActivity*, errors from the provided service of the called component (and any other component down the call stack) can be handled. The case studies in Section 6 show different possible usages of FTSs.

4.3 System Reliability Models

4.3.1 System Architecture

In our approach, software architects are required to provide a system reliability model. Fig. 2 shows an extract of our reliability modeling schema with modeling elements that supports software architects to create a system reliability model. 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*).

4.3.2 Usage Profile

After modeling system architecture, software architects model a usage profile for the user interface of the required functionality. A usage profile (*UsageProfile*) contains usage profile parts (*UsageProfilePart*) with different probabilities which model different usage scenarios of the system. A usage profile part must include sufficient information to determine the branching probabilities of branching structures, and the discrete probability distributions (or the average values) of the loop counts of looping structures.

Example 7. *Continuing with Example 2, Fig. 10 shows an example of system reliability model. The system architecture includes instances of components $Comp_1$, $Comp_2$, and $Comp_3$. They are connected via component connectors. Provided service Svc_0 of $Comp_1$'s component instance is exposed as a user interface for users.*

The usage profile contains two usage profile parts with probabilities 0.4 and 0.6. This means that with probability 0.4, users access with usage profile part 1 and with probability 0.6, users access with usage

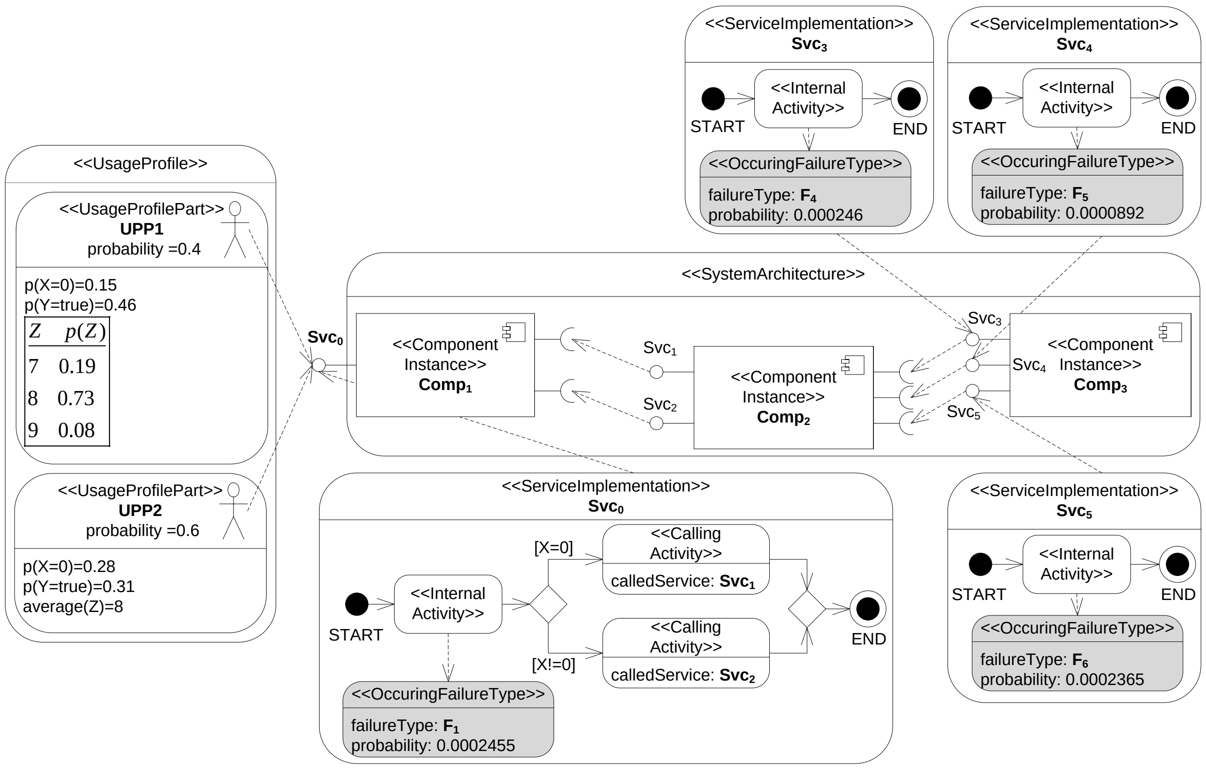


Figure 10: Example of system reliability model.

profile part 2. Each usage profile part contains probabilities and a distribution (or an average) to determine the branching probabilities of the branching structures and the discrete probability distribution (or the average value) of the loop count of the looping structure.

5 Reliability Prediction

After the software architects provide a system reliability model, we can predict the system reliability under the specified usage profile. The prediction process starts with the system reliability model and the component reliability specifications, and ends with the system reliability prediction output. It includes transformation for each usage profile part and an aggregation of results.

5.1 Transformation for each usage profile part

The transformation is to derive the reliability for the provided service for users which the current usage profile part refers to. It starts with the service implementation of this provided service. 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's inner parts 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.

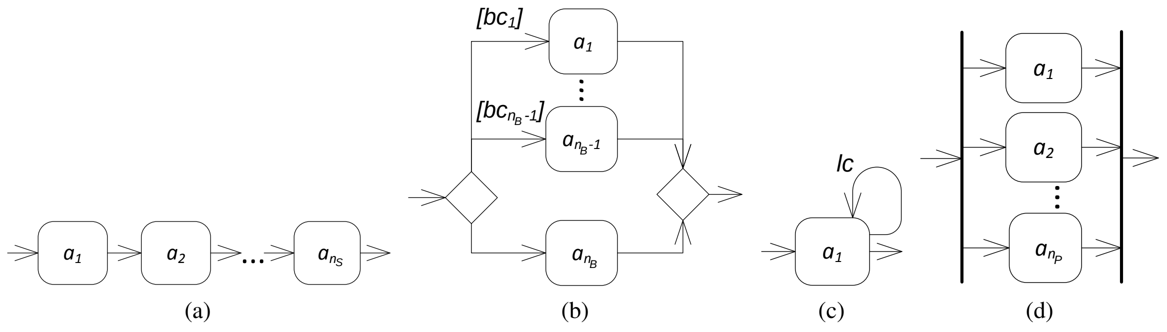


Figure 11: Example of structures: (a) Sequential structure, (b) Branching structure, (c) Looping structure, and (d) Parallel structure.

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

5.1.1 Sequential Structure

Considering a sequential structure with n_S sequential parts a_1, a_2, ..., a_{n_S} as in Fig. 11a, its equivalent *ia* has:

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

and for $1 \leq j \leq m$

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

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

- Part a_1 fails with failure type j: $fp_j(a_1)$.
- Part a_1 succeeds, part a_2 fails with failure type j: $sp(a_1)fp_j(a_2)$.
- ...
- Parts a_1, a_2, ..., a_{n_S-1} succeed, part a_{n_S} fails with failure type j: $\left(\prod_{k=1}^{n_S-1} sp(a_k)\right) fp_j(a_{n_S})$.

5.1.2 Branching Structure

Considering a branching structure with $n_B - 1$ *if* parts a_1, a_2, ..., a_{n_B-1} and a single *else* part a_{n_B} as in Fig. 11b, its equivalent *ia* has:

$$sp = fp_0 = \sum_{i=1}^{n_B-1} p(bc_i) sp(a_i) + \left(1 - \sum_{i=1}^{n_B-1} p(bc_i)\right) sp(a_{n_B}) \tag{3}$$

and for $1 \leq j \leq m$

$$fp_j = \sum_{i=1}^{n_B} p(bc_i) fp_j(a_i) + \left(1 - \sum_{i=1}^{n_B-1} p(bc_i)\right) fp_j(a_{n_B}) \tag{4}$$

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

5.1.3 Looping Structure

Considering a looping structure with a single looping part a_1 as in Fig. 11c, it can be seen as a sequential structure of part a_1 appearing lc times. Therefore, in case the current usage profile part contains the average value of the loop count, i.e. $average(lc) = v$, the equivalent ia of the looping structure has:

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

and for $1 \leq j \leq m$

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

In case the current usage profile part contains the discrete probability distribution of the loop count, i.e. all possible values for the loop count $\{v_1, v_2, ..., v_t\} \subseteq \mathbf{N}_0$ and their occurrence probabilities $\{p(v_1), p(v_2), \ldots, p(v_t)\}$ such that $\sum_{i=1}^{t} p(v_i) = 1$, the equivalent ia of the looping structure has:

$$sp = fp_0 = \sum_{i=1}^{t} p(v_i) sp(a_1)^{v_i} \tag{7}$$

and for $1 \leq j \leq m$

$$fp_j = \sum_{i=1}^{t} p(v_i) \sum_{k=1}^{v_i} sp(a_1)^{k-1} fp_j(a_1) \tag{8}$$

5.1.4 Parallel Structure

Considering a parallel structure with n_P parallel parts a_1, a_2, ..., a_{n_P} as in Fig. 11d, to avoid introducing additional failures types when the parallel parts fail in different failure types, it is assumed that the failure types are sorted in a certain order (e.g. according to their severity). Therefore, when the parallel parts fail in different failure types, the failure type of the parallel structure is the highest failure type of its parallel parts. Without loss of generality, the failures types are assumed to be sorted in the following order: $F_1 \leq F_2 \leq \ldots \leq F_m$, then the equivalent ia of the parallel structure has:

$$sp = fp_0 = \prod_{i=1}^{n_P} sp(a_i) \tag{9}$$

and for $1 \leq j \leq m$

$$fp_j = \sum_{i=1}^{n_P} \left(\prod_{k=1}^{i-1} \left(1 - \sum_{l=j}^{m} fp_l(a_k) \right) \times fp_j(a_i) \times \prod_{k=i+1}^{n_P} \left(1 - \sum_{l=j+1}^{m} fp_l(a_k) \right) \right) \tag{10}$$

Equation (10) can be obtained from the following disjoint cases:

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

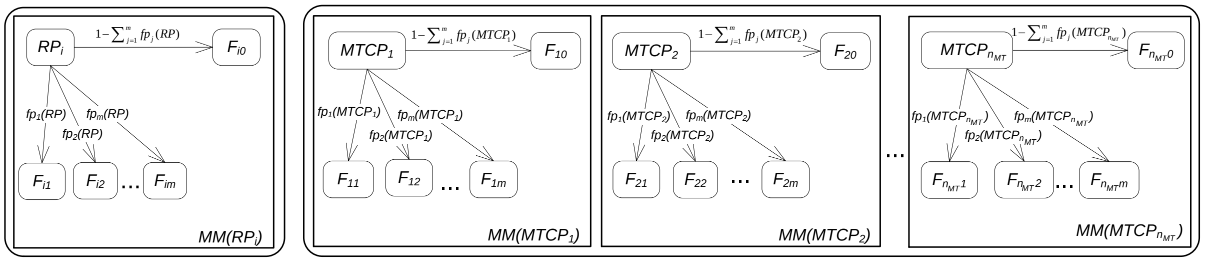


Figure 12: Markov models: (a) for i-th retry and (b) for *MultiTryCatchParts*.

- ...
- Parts a_1, a_2, ..., a_{n_P-1} fail with failure type $l < j$, part a_{n_P} fails with failure type j: $\left(\prod_{k=1}^{n_P-1}\left(1-\sum_{l=j}^{m} fp_l\left(a_k\right)\right)\right) fp_j(a_{n_P})$.

5.1.5 RetryStructure

Considering a *RetryStructure*, let *rc* be the number of times to retry, $\mathscr{F}_H$ be the set of handled failure types, $\mathbf{C}$ be the error detection matrix represented by $\{c_{rs}\}$ with $r, s \in \{0, 1, ..., m\}$. The failure model of the *RetryPart* (abbreviated as *RP*) includes failure types $F_1, F_2, ..., F_m$ and their occurrence probabilities $fp_1\left(RP\right), fp_2\left(RP\right), ..., fp_m\left(RP\right)$.

In order to transform the *RetryStructure* into an equivalent *ia*, 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.

The *i-th* retry is represented by a Markov model $MM\left(RP_i\right)$ as in Fig. 12a. $MM\left(RP_i\right)$ has a state RP_i as an initial state, states F_{ij} with $j \in \{0, 1, ..., m\}$ as states of failure types. Therefore, with the number of times to retry *rc*, there are $rc+1$ Markov models $MM\left(RP_i\right)$, with $i \in \{0, 1, ..., rc\}$. The problem is how to connect these Markov models into one Markov model representing all the possible execution paths of the whole structure (following the semantics as illustrated in Fig. 6). To solve the problem, we add $m+2$ states, namely one state *START* and states F_j with $j \in \{0, 1, ..., m\}$, and the following transitions:

- A transition from *START* to RP_0 with probability 1.
- For $MM\left(RP_{rc}\right)$ (i.e. the Markov model of the last retry), transitions from F_{rcj} to F_j with probability 1 for all $j \in \{0, 1, ..., m\}$.
- For the other Markov models, i.e. $MM\left(RP_i\right)$ with $i \in \{0, 1, ..., rc-1\}$: transitions from F_{ij} to RP_{i+1} with probability $\sum_{F_k \in \mathscr{F}_H} c_{jk}$; transitions from F_{ij} to F_j with probability $1 - \sum_{F_k \in \mathscr{F}_H} c_{jk}$ for all $j \in \{0, 1, ..., m\}$.

As the resulting Markov model is an absorbing Markov chain, the success probability of the equivalent *ia*, which is the probability of reaching F_0 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 [26].

5.1.6 MultiTryCatchStructure

For a *MultiTryCatchStructure*, let n_{MT} be the number of *MultiTryCatchParts*, $\mathscr{F}_{H_i}$ be the set of handled failure types of *MultiTryCatchPart i* with $i \in \{1,2,...,n_{MT}\}$, $\mathbf{C}^i$ be the error detection matrix for *MultiTryCatchPart i* which is represented by $\{c^i_{rs}\}$ with $r,s \in \{0,1,...,m\}$. The failure model of the *MultiTryCatchPart i* (abbreviated as $MTCP_i$) includes failure types $F_1,F_2,...,F_m$ and their occurrence probabilities $fp_1(MTCP_i),fp_2(MTCP_i),...,fp_m(MTCP_i)$.

Similar to the case of *RetryStructures*, to transform the *MultiTryCatchStructure* into an equivalent *ia*, 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.

MultiTryCatchParts are represented by Markov models as in Fig. 12b. $MM(MTCP_i)$ has a state $MTCP_i$ as an initial state, states F_{ij} with $j \in \{0,1,...,m\}$ as states of failure types. To connect these Markov models into one Markov model representing all the possible execution paths of the whole structure (following the semantics as illustrated in Fig. 7), we add $m+2$ states, namely one state *START* and states F_j with $j \in \{0,1,...,m\}$, and the following transitions:

- An transition *START* to $MTCP_1$ with probability 1.
- For $MM(MTCP_{n_{MT}})$ (i.e. the Markov model of the last *MultiTryCatchPart*): transitions from F_{nj} to F_j with probability 1 for all $j \in \{0,1,...,m\}$.
- For other Markov models, i.e. $MM(MTCP_i)$ with $i \in \{1,2,...,n_{MT}-1\}$: transitions from F_{ij} to $MTCP_x$ with probability $\sum\limits_{F_k \in \mathscr{F}_{H_{ix}}} c^i_{jk}$ where $\mathscr{F}_{H_{ix}} = \mathscr{F}_{H_x} - \bigcup\limits_{i<y<x} \mathscr{F}_{H_y}$ is the set of failure types that can not be handled by any $MTCP_y$ for $i<y<x$ but can be handled by $MTCP_x$; transitions from F_{ij} to F_j with probability $1 - \sum\limits_{i<x\leq n_{MT}} \left(\sum\limits_{F_k \in \mathscr{F}_{H_{ix}}} c^i_{jk} \right)$ for all $j \in \{0,1,...,m\}$.

With the Markov model representing all the possible execution paths of the whole structure, the probability of reaching F_0 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*.

5.1.7 MVPStructure

Considering a *MVPStructure*, let n_{MV} be the number of *MVPParts*, $maxTI$ be the value of the field *maxToleratedIncompletion*, $minMS$ be the value of the field *minMajoritySize*, $F_{udFT} \in \{F_1,F_2,...,F_m\}$ be the value of the field *undetectedFailureType*, $F_{dFT} \in \{F_1,F_2,...,F_m\}$ be the value of the field *detectedFailureType*, $\mathbf{D^i}$ be the error property vector for *MVPPart i* with $i \in \{1,2,...,n_{MV}\}$ which is represented by $\{d^i_r\}$ with $r \in \{1,2,...,m\}$, $\mathbf{E}$ be the value of the field *agreementOfErrorsVector* which is represented by $\{p_y\}$ with $minMS \leq y \leq n_{MV}$. The failure model of the *MVPPart i* (abbreviated as $MVPP_i$) includes failure types $F_1,F_2,...,F_m$ and their occurrence probabilities $fp_1(MVPP_i),fp_2(MVPP_i),...,fp_m(MVPP_i)$.

In order to transform the *MVPStructure* into an equivalent *ia*, the transformation calculates the probabilities of the possibilities for the set of results of *MVPParts*' executions (Step 1), and then the probabilities for the voter to signal a detected failure or to produce a correct or erroneous output (Step 2) (following the semantics as illustrated in Fig. 9). After that, the equivalent *ia* can be derived from the probabilities for the voter by the transformation (Step 3).

Step 1, after the executions of the first n *MVPParts*, let $(x,y,z)_n$ be a possibility for the set of results of these *MVPParts*' executions where x is the number of correct outputs, y is the number of erroneous outputs, and z is the number of incomplete executions such that $x+y+z=n$, let $p((x,y,z)_n)$ be the probability of the possibility $(x,y,z)_n$. Therefore, there are $(n+2)(n+1)/2$ possibilities and the same number of probabilities. At the beginning, there is one possibility $(0,0,0)_0$ with probability $p((0,0,0)_0)=1$.

After the executions of the first $n+1$ *MVPParts*, the set of results of these *MVPParts*' executions is $(x',y',z')_{n+1}$ if (1) $x'>0$, the set of results of the executions of the first n *MVPParts* is $(x'-1,y',z')_n$ and the $(n+1)-th$ *MVPPart* produces a correct output, or (2) $y'>0$, the set of results of the executions of the first n *MVPParts* is $(x',y'-1,z')_n$ and the $(n+1)-th$ *MVPPart* produces an erroneous output, or (3) $z'>0$, the set of results of the executions of the first n *MVPParts* is $(x',y',z'-1)_n$ and the $(n+1)-th$ *MVPPart* does not complete its execution. Therefore, the probability $p\left((x',y',z')_{n+1}\right)$ is calculated as follows:

$$\begin{aligned} p\left((x',y',z')_{n+1}\right) &= p\left((x'-1,y',z')_n\right)\left(1-\sum_{j=1}^{m} fp_j\left(MVPP_{n+1}\right)\right)\Bigg|x'>0 \\ &+ p\left((x',y'-1,z')_n\right)\left(\sum_{j=1}^{m} fp_j\left(MVPP_{n+1}\right)\left(1-d_j^{n+1}\right)\right)\Bigg|y'>0 \\ &+ p\left((x',y',z'-1)_n\right)\left(\sum_{j=1}^{m} fp_j\left(MVPP_{n+1}\right)d_j^{n+1}\right)\Bigg|z'>0 \end{aligned} \tag{11}$$

By using Equation 11, the transformation recursively calculates the probabilities for all the possibilities of the set of results of n_{MV} *MVPParts*' executions.

Step 2, with the probabilities $p\left((x'',y'',z'')_{n_{MV}}\right)$ for all the possibilities of the set of results of n_{MV} *MVPParts*' executions, the transformation calculates the probabilities for the voter as follows:

- The voter produces a correct output (F_0) if in a possibility for the set of the results of *MVPParts*' executions, the number of incomplete executions is at most *maxTI* and the number of correct outputs is at least *minMS*:

$$p(F_0)=\sum_{(x'',y'',z'')_{n_{MV}}} p\left(\left(x'',y'',z''\right)_{n_{MV}}\right)\Big|z''\le maxIT, x''\ge minMS \tag{12}$$

- The voter produces an erroneous output of F_{udFT} if in a possibility for the set of the results of *MVPParts*' executions, the number of incomplete executions is at most *maxTI*, the number of correct outputs is less than *minMS*, the number of erroneous outputs is at least *minMS*, and there is an agreement of the majority erroneous outputs:

$$p(F_{udFT})=\sum_{(x'',y'',z'')_{n_{MV}}} p\left(\left(x'',y'',z''\right)_{n_{MV}}\right)p_{y''}\Big|z''\le maxIT, x''<minMS, y\ge minMS \tag{13}$$

- The voter signals a detected failure of F_{dFT} with probability:

$$p(F_{dFT})=1-p(F_0)-p(F_{udFT}) \tag{14}$$

Step 3, the equivalent *ia* derived from the probabilities for the voter by the transformation has, with $0\le j\le m$:

$$fp_j=\begin{cases} p(F_0) \text{ if } j=0 \\ p(F_{udFT}) \text{ if } j=udFT \\ p(F_{dFT}) \text{ if } j=dFT \\ 0 \text{ otherwise} \end{cases} \tag{15}$$

5.1.8 The reliability under the usage profile part

Finally, in our approach, we define the reliability as $R = 1 - POFOD$, where $POFOD$ is the probability of failure on demand. Therefore, based on the described transformations, the reliability for the provided service for users which the current usage profile part refers to is the success probability of the equivalent *ia* of the service implementation of this service.

5.2 Aggregation of Results

The results of the reliability of provided services for users which the usage profile parts in the usage profile refer to are aggregated as follows: Let $R(UPP_k)$ be the reliability of the provided service for users which usage profile part UPP_k refers to, ℓ be the number of usage profile parts in the usage profile, P_k be the probability that users access with usage profile part UPP_j such that $\sum_{k=1}^{\ell} P_k = 1$, then the overall system reliability can be determined as a weighted sum over all usage profile parts in the usage profile:

$$R = \sum_{k=1}^{\ell} P_k \, R(UPP_k) \tag{16}$$

Example 8. *Continuing with Example 7, the overall system reliability is determined as* $R = 0.4\ R(UPP_1) + 0.6\ R(UPP_2)$.

5.3 Proving the correctness of the transformation algorithm

The transformation algorithm described above is a matter of complicated bookkeeping, generating all possible execution paths with their corresponding probabilities for a structure and computing the failure model for the equivalent *ia* of the structure via summation of multiplied probabilities over the available paths. Therefore, under all the stated assumptions, we argue for the correction of the algorithm as “by construction”. This means that once the underlying ideas are understood, anyone would agree that the algorithm can be made to work. A more formal proof could be given by induction on the size of the instance (here possibly the number of inner parts of a structure and the number of called and nested structures throughout the whole system model). However, we deem such a proof “uninformative”, i.e. it does not help in understanding the algorithm.

5.4 Complexity

Regarding space-effectiveness, by transforming a structure into an equivalent *ia*, the transformation algorithm no longer needs to store the structure with its inner parts in the memory, but can efficiently transform the outer structure using the equivalent *ia*. Due to its recursive nature, the algorithm transforms a structure as soon as its inner parts have been transformed into equivalent *ia(s)*, therefore, can efficiently reduce the possibility of state-space explosion.

At any point in time, the number of structures present in the memory is limited by the maximum depth of the stack of called and nested structures throughout the whole system model. The amount of memory required by the algorithm for a structure is almost equal to the amount of memory required to store the equivalent *ia(s)* of its inner parts, apart from the fact that the algorithm requires an additional amount of memory for (1) a Markov chain in case of a *RetryStructure* or a *MultiTryCatchStructure*, or (2) the possibilities of the set of results of *MVPParts*’ executions and their probabilities in case of a *MVPStructure*. The aggregation of results over all usage profile parts in the usage profile can be calculated one after another, without the need to store each result separately.

Table 1: Running Times of the Transformation Algorithm for Different Structure Types.

Structure type	**Running time**
Sequential structure	$O(mn_S)$
Branching structure	$O(mn_B)$
Looping structure	$O(mv)$ or $O\left(m\sum_{i=1}^{t} v_i\right)$
Parallel structure	$O\left(m^2 n_P^2\right)$
RetryStructure	$O\left(m^3 rc^3\right)$
MultiTryCatchStructure	$O\left(m^3 n_{MT}^3\right)$
MVPStructure	$O\left(n_{MV}\left(m + n_{MV}^2\right)\right)$

Regarding time-effectiveness, it is assumed that the running time of the transformation algorithm is a function of the structure type and the number of failure types. Based on Equations (1), (2),..., (15), it is possible to obtain the running times of the algorithm for the sequential, branching, looping, parallel, and *MVPStructure* structure types. The running times of the algorithm for a *RetryStructure* or a *MultiTryCatchStructure* can be obtained from the process of creating and solving Markov chains (see Section 5.1.5 or 5.1.6, respectively).

Table 1 shows the running times of the algorithm for structure types given that their inner parts have been transformed into equivalent *ia(s)*. The running time of the algorithm for any structure type is polynomial time in the number of failure types and in the number(s) representing the characteristics of the structure type, i.e. n_S, n_B, v (or $\{v_1, v_2, ..., v_t\}$), n_P, rc, n_{MT}, or n_{MV}. The aggregation of results over l usage profile parts in the usage profile has a running time of $O(l)$.

5.5 Implementation

We have implemented the transformation algorithm in our reliability prediction tool. The tool receives a system reliability model as an input, validates this input against a set of predefined semantic constraints in our reliability modeling schema (e.g. the total probability of all usage profile parts must be 1), and produces the system reliability prediction as an output. This output includes not only the predicted system reliability but also predicted failure probabilities of user-defined failure types.

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

6 Case Study Evaluation

6.1 Case Study I: Reporting Service of a Document Exchange Server

6.1.1 Description of the Case Study

The program chosen for the case study is the reporting service of a document exchange server [12]. 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. 13 using our reliability modeling schema. At the architecture level, the reporting service consists of four components: *ReportingMediator*, *ReportingEngine*, *SourceManager*, and *Destination-Manager*. Component *SourceManager* provides two services to get information about pending docu-

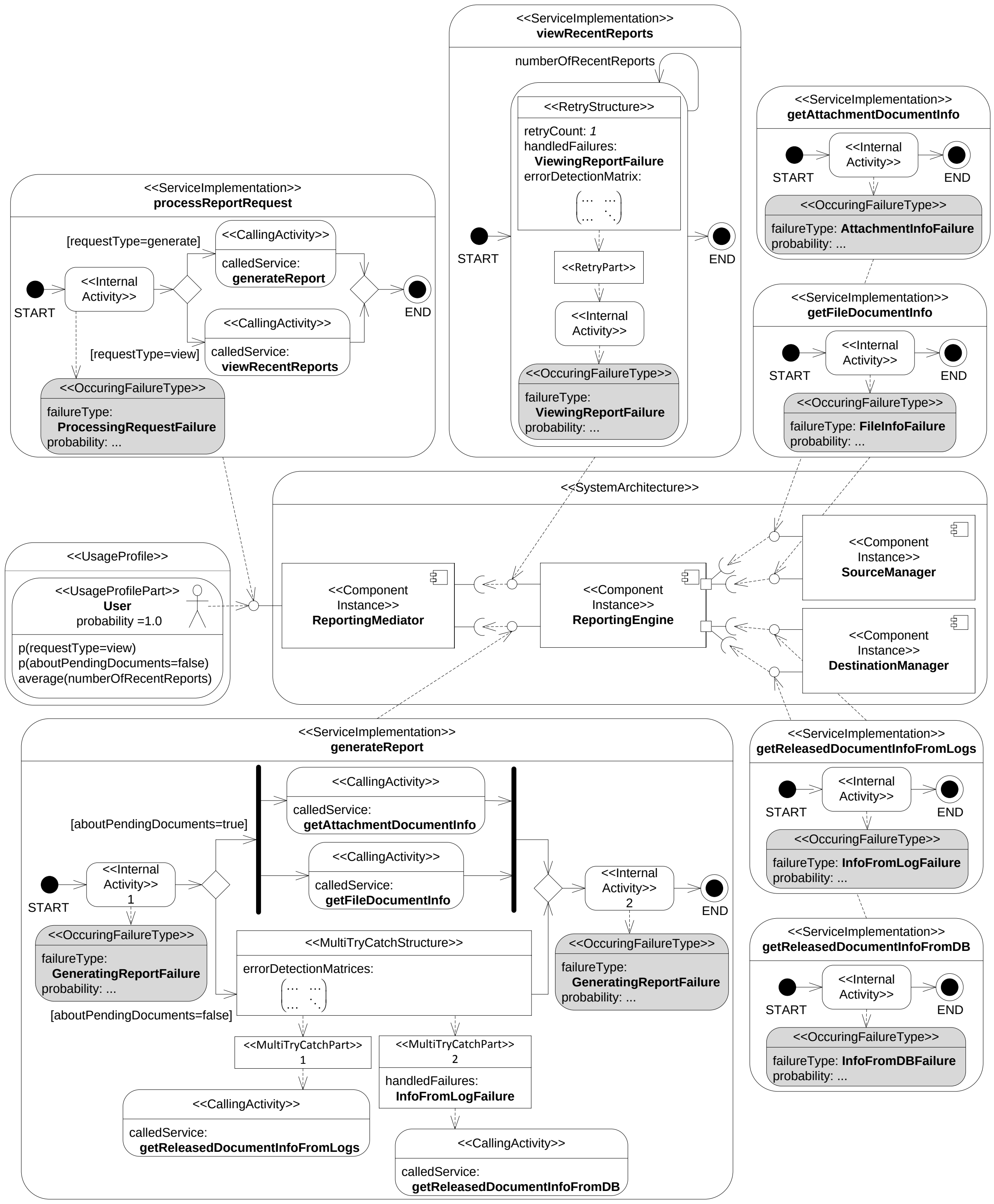

<<ServiceImplementation>>
viewRecentReports
numberOfRecentReports
<<RetryStructure>>
retryCount: 1
handledFailures: ViewingReportFailure
errorDetectionMatrix:
<<RetryPart>>
<<Internal Activity>>
<<OccuringFailureType>>
failureType: ViewingReportFailure
probability: ...
START
END
<<ServiceImplementation>>
processReportRequest
[requestType=generate]
[requestType=view]
<<CallingActivity>>
calledService: generateReport
calledService: viewRecentReports
failureType: ProcessingRequestFailure
<<ServiceImplementation>>
getAttachmentDocumentInfo
failureType: AttachmentInfoFailure
<<ServiceImplementation>>
getFileDocumentInfo
failureType: FileInfoFailure
<<SystemArchitecture>>
<<UsageProfile>>
<<UsageProfilePart>>
User
probability =1.0
p(requestType=view)
p(aboutPendingDocuments=false)
average(numberOfRecentReports)
<<Component Instance>> ReportingMediator
<<Component Instance>> ReportingEngine
<<Component Instance>> SourceManager
<<Component Instance>> DestinationManager
<<ServiceImplementation>>
generateReport
[aboutPendingDocuments=true]
[aboutPendingDocuments=false]
calledService: getAttachmentDocumentInfo
calledService: getFileDocumentInfo
<<Internal Activity>> 1
<<Internal Activity>> 2
failureType: GeneratingReportFailure
<<MultiTryCatchStructure>>
errorDetectionMatrices:
<<MultiTryCatchPart>> 1
<<MultiTryCatchPart>> 2
handledFailures: InfoFromLogFailure
calledService: getReleasedDocumentInfoFromLogs
calledService: getReleasedDocumentInfoFromDB
<<ServiceImplementation>>
getReleasedDocumentInfoFromLogs
failureType: InfoFromLogFailure
<<ServiceImplementation>>
getReleasedDocumentInfoFromDB
failureType: InfoFromDBFailure

Figure 13: The system reliability model of the reporting service.

Table 2: Reporting Service: 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

Table 3: Reporting Service: 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_4	*getAttachmentDocumentInfo/ia*	1
a_5	*getFileDocumentInfo/ia*	1
a_6	*getReleasedDocumentInfoFromLogs/ia*	2
a_7	*getReleasedDocumentInfoFromDB/ia*	1
a_8	*generateReport/ia 2*	1

ments: *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: *getReleased-DocumentInfoFromLogs* 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 requests 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 client requests in service *processReportRequest*; bugs in the code of service *generateReport* may lead to a *GeneratingReportFailure*. Table 2 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 version of the service by reinserting faults discovered during operational usage and integration testing (Table 3 shows the number of reinserted faults).

Table 4: Reporting Service: 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_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	$fp_7(a_7) = 0.0909091; fp_j(a_7) = 0\ \forall j \neq 7$
a_8	$fp_3(a_8) = 0.0549451; fp_j(a_8) = 0\ \forall j \neq 3$

Table 5: Reporting Service: Error Detection Matrices.

Provided service/FTS	**Error detection matrix**
viewRecentReports/RetryStructure	$\begin{array}{cc} & F_2 \\ F_0 & \left(\begin{array}{c} 0.0 \\ 0.777778 \end{array}\right) \\ F_2 & \end{array}$
generateReport/MultiTryCatchStructure	$\begin{array}{cc} & F_6 \\ F_0 & \left(\begin{array}{c} 0.0 \\ 0.578947 \end{array}\right) \\ F_6 & \end{array}$

6.1.2 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 and can 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 the 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 F_j (with $j \in \{1,2,...,7\}$) of internal activity a_i (with $i \in \{1,2,\ldots,8\}$) as: $fp_j(a_i) = f_{ji}/n_i$ where f_{ji} is the number of failures of failure type F_j of the internal activity a_i and n_i is the number of runs of the internal activity a_i. Failure probabilities of different failure types of internal activities are given in Table 4. 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 failure type F_r of the inner part of the FTS (i.e. *RetryPart* of the *RetryStructure* or *MultiTryCatchPart 1* of the *MultiTryCatchStructure*) and daf_{rs} is the number of failures of failure type F_r of the inner part detected as failure type F_s. The error detection matrices for the two FTSs are given in Table 5.

We considered the usage information obtained from the executions of test cases as being represented by a single usage profile part. Therefore, the usage profile includes one usage profile part with probability 1.0. Within the usage profile part, 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

Table 6: Reporting Service: The Usage Profile Part.

Element	**Value**
p(*requestType*=view)	0.178571
p(*aboutPendingDocuments*=false)	0.608696
average(*numberOfRecentReports*)	2

Table 7: Reporting Service: 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

condition bc_i and n is the total number of times control reached the branching structure; the average value of the 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 part including the branching probabilities of the branching structures and the average value of the number of loops of the looping structure is given in Table 6.

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 7[3] 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. The deviation comes from the modeling abstractions of our approach: (1) the Markov assumption, (2) the assumption that components fail independently, and (3) the assumption that a component failure, without FTMs to handle, leads immediately to a system failure (see Section 7 for more details).

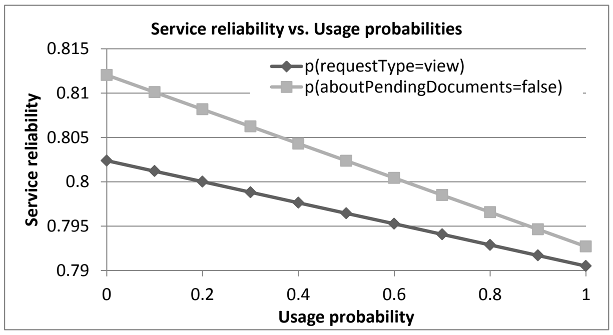
6.1.3 Sensitivity Analyses and the Impact of FTSs

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 fault tolerance variants.

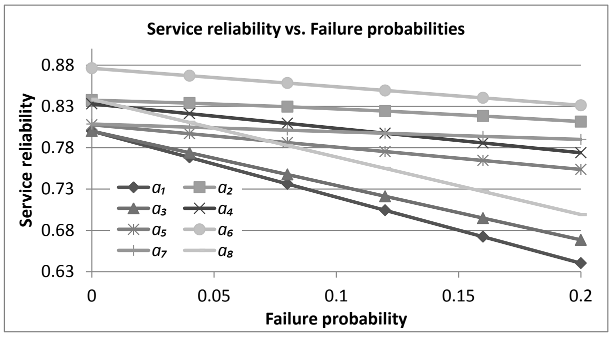
First, we conducted a sensitivity analysis modifying the usage probabilities (Fig. 14a). The reliability of the reporting service is more sensitive to the portion of report types to generate (*aboutPendingDocuments*=true or false) because its corresponding curve has the steepest slope.

Second, we conducted a sensitivity analysis modifying failure probabilities of the internal activities (Fig. 14b). 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.

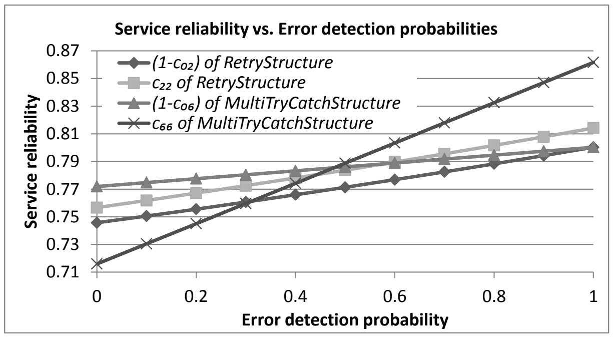
[3]Notice that different from our former work [12] which set the input parameters for illustrative purpose, in this paper, we estimated the input parameters by using the method above. Therefore, these estimates and the predicted reliability are for the faulty version. This means that our prediction result does not contradict the prediction result of our former work.



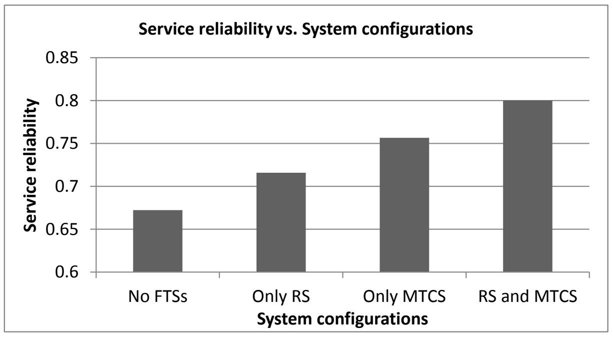
(a)



(b)



(c)



(d)

Figure 14: Reporting Service: Sensitivity analyses.

Third, we conducted a sensitivity analysis modifying error detection probabilities of the two FTSs (Fig. 14c). 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. 14d). 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 0 to the elements c_{66} the error detection matrix for the *MultiTryCatchPart 1* to disable the structure). For variant *No FTSs*, the two above modifications are included.

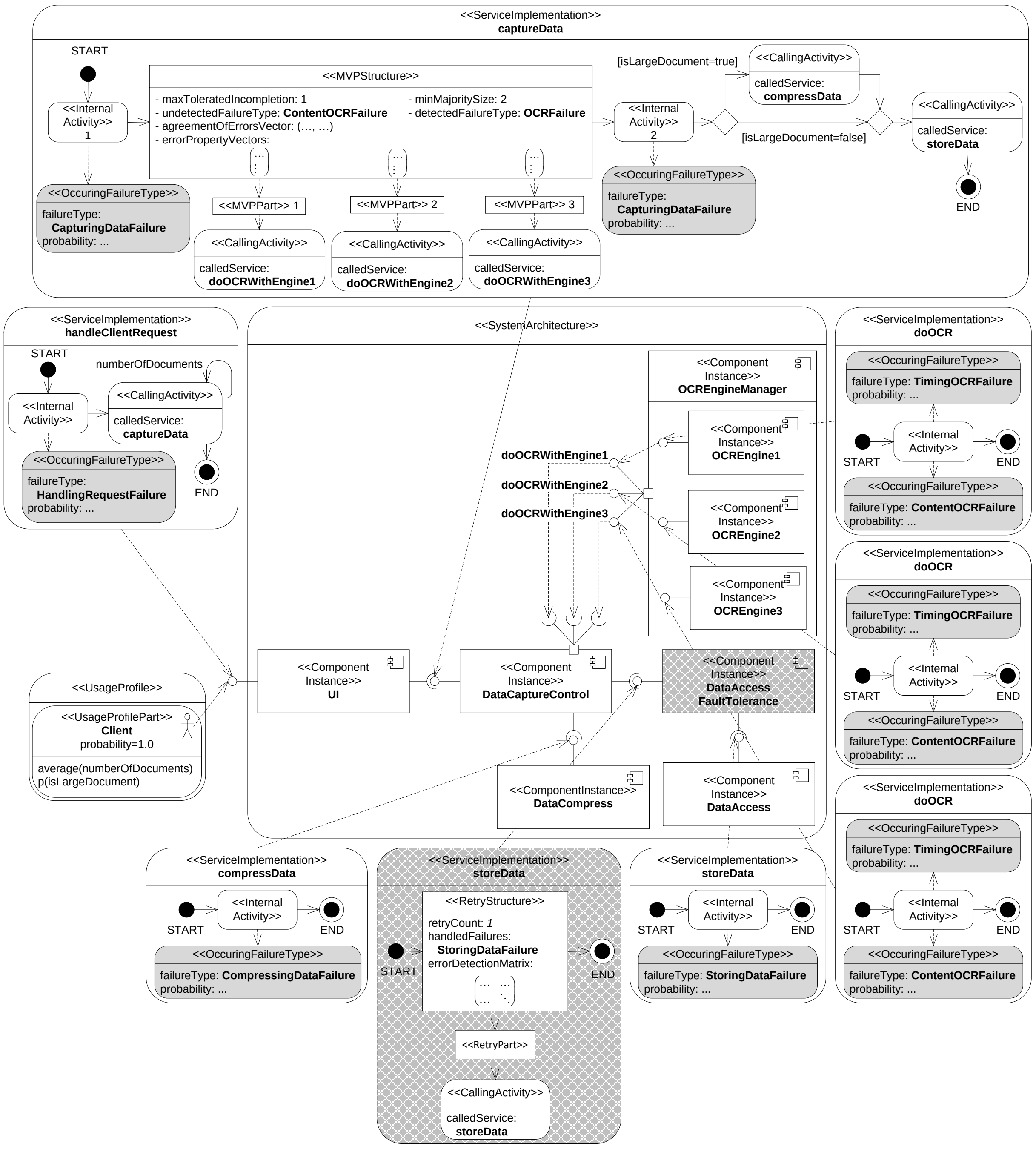

<<ServiceImplementation>>
captureData
START
<<Internal Activity>> 1
<<MVPStructure>>
- maxToleratedIncompletion: 1
- undetectedFailureType: ContentOCRFailure
- agreementOfErrorsVector: (…, …)
- errorPropertyVectors:
- minMajoritySize: 2
- detectedFailureType: OCRFailure
<<MVPPart>> 1
<<MVPPart>> 2
<<MVPPart>> 3
<<CallingActivity>> calledService: doOCRWithEngine1
<<CallingActivity>> calledService: doOCRWithEngine2
<<CallingActivity>> calledService: doOCRWithEngine3
<<OccuringFailureType>> failureType: CapturingDataFailure probability: ...
<<Internal Activity>> 2
[isLargeDocument=true]
[isLargeDocument=false]
<<CallingActivity>> calledService: compressData
<<CallingActivity>> calledService: storeData
END
<<ServiceImplementation>> handleClientRequest
numberOfDocuments
<<Internal Activity>>
<<CallingActivity>> calledService: captureData
<<OccuringFailureType>> failureType: HandlingRequestFailure probability: ...
<<SystemArchitecture>>
<<Component Instance>> OCREngineManager
<<Component Instance>> OCREngine1
<<Component Instance>> OCREngine2
<<Component Instance>> OCREngine3
doOCRWithEngine1
doOCRWithEngine2
doOCRWithEngine3
<<Component Instance>> UI
<<Component Instance>> DataCaptureControl
<<Component Instance>> DataAccess FaultTolerance
<<ComponentInstance>> DataCompress
<<Component Instance>> DataAccess
<<UsageProfile>>
<<UsageProfilePart>> Client probability=1.0
average(numberOfDocuments)
p(isLargeDocument)
<<ServiceImplementation>> doOCR
<<OccuringFailureType>> failureType: TimingOCRFailure probability: ...
<<Internal Activity>>
<<OccuringFailureType>> failureType: ContentOCRFailure probability: ...
<<ServiceImplementation>> compressData
<<OccuringFailureType>> failureType: CompressingDataFailure probability: ...
<<ServiceImplementation>> storeData
<<RetryStructure>>
retryCount: 1
handledFailures: StoringDataFailure
errorDetectionMatrix:
<<RetryPart>>
<<CallingActivity>> calledService: storeData
<<OccuringFailureType>> failureType: StoringDataFailure probability: ...

Figure 15: The system reliability model of the DataCapture system.

6.2 Case Study II: DataCapture System

As the second case study, we analyzed the reliability of a DataCapture system. The system allows clients to capture data from printed texts such as documents, invoices, receipts, etc. using OCR (Optical Character Recognition) technology. Different configurations are possible for the system by varying the number of activated OCR engines. Fig. 15 shows the system reliability model for the system configuration of three activated OCR engines.

The system functionality is provided through four separated primitive components (*UI*, *DataCap-*

Table 8: DataCapture System: Different Failure Types and Their Symbols.

Failure Type	**Symbol**
HandlingRequestFailure	F_1
CapturingDataFailure	F_2
ContentOCRFailure	F_3
TimingOCRFailure	F_4
OCRFailure	F_5
CompressingDataFailure	F_6
StoringDataFalure	F_7

Table 9: DataCapture System: Error Property Vectors.

Provided service/FTS	**Error property vector**
captureData/MVPStructure	$\begin{array}{cc} & IncExe \\ F_3 & \left(\begin{array}{c} 0 \\ 1 \end{array}\right) \\ F_4 & \end{array}$

tureControl, *DataCompress*, and *DataAccess*) and one composite component (*OCREngineManager*) containing three nested primitive components (*OCREngine1*, *OCREngine2*, and *OCREngine3*). During the system operation, different types of failures may occur in the involved component instances. For example, a *HandlingRequestFailure* may occur during handling client requests in component *UI*. A *ContentOCRFailure* or *TimingOCRFailure* may occur in component *OCREngine1* due to bugs in the code implementing the component. Bugs in the compressing algorithm of component *DataCompress* may cause a *CompressingDataFailure*. Different failure types and their symbols are given in Table 8.

There is a FTS in the system, namely the *MVPStructure* in the implementation of service *captureData* of component *DataCaptureControl*. This structure tolerates at most one incomplete execution from three *doOCR* services and requires at least two results from these services to agree to produce an output. Besides a correct output, the structure can produce an erroneous output of *ContentOCRFailure* or signal a detected failure of *OCRFailure*. Because failures of *ContentOCRFailure* are content failures and failures of *TimingOCRFailure* are late timing failures, the error property vectors for the three *MVPParts* are the same and given in Table 9. Another FTS can be optionally introduced into the system, in terms of additional component which is shown in cross-hatched grey in Fig. 15. Component *DataAccessFaultTolerance* can be put in the middle of component *DataCaptureControl* and component *DataAccess*. It has the ability to retry in case there is a failure of *StoringDataFailure*. The number of times to retry of this structure is 1 (*retryCount=1*).

To observe the accuracy of our prediction approach, we conduct a reliability measurement on a prototype implementation of the system, then compare the measurement to a reliability prediction to observe if there is a significant deviation caused by the modeling abstractions. The implementation is written in Java, using an Apache Derby database for storing the data and Java Native Access (JNA) wrappers for accessing native OCR engines. For the measurement, the system is executed in a testbed that triggers usage scenario runs and records the execution traces of all scenario runs.

To be able to conduct the measurements, several simplifications had to be included compared with a real-world field experiment. First, the total number of scenario runs is limited to 4000. Each scenario run consists of an average of 10 documents per call and a probability of 40% for the documents to be large, i.e. requiring compression before storing. Second, the system reliability is not measured due to real faults but rather to faults which have been injected in an artificial manner, with externally controlled occurrence probabilities.

Table 10: DataCapture System: Failure Probabilities of Internal Activities.

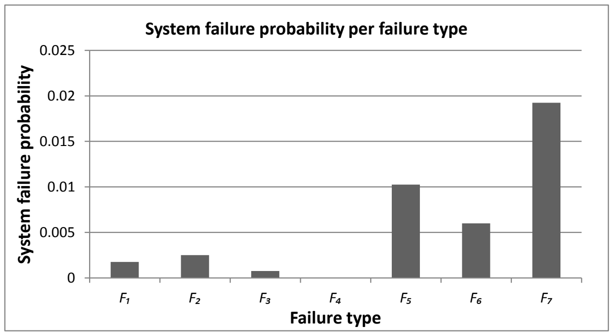
Provided service/Internal activity (ia)	fp_j
handleClientRequest/ia	$fp_1 = 0.00175$; $fp_j = 0\ \forall j \neq 1$
captureData/ia 1	$fp_2 = 0.00012747$; $fp_j = 0\ \forall j \neq 2$
captureData/ia 2	$fp_2 = 0.00012763$; $fp_j = 0\ \forall j \neq 2$
OCREngine1's doOCR/ia	$fp_3 = 0.01053007$; $fp_4 = 0.00724102$; $fp_j = 0\ \forall j \neq 3, j \neq 4$
OCREngine2's doOCR/ia	$fp_3 = 0.00813340$; $fp_4 = 0.00119834$; $fp_j = 0\ \forall j \neq 3, j \neq 4$
OCREngine3's doOCR/ia	$fp_3 = 0.00963769$; $fp_4 = 0.02771474$; $fp_j = 0\ \forall j \neq 3, j \neq 4$
compressData/ia	$fp_6 = 0.00151295$; $fp_j = 0\ \forall j \neq 6$
storeData/ia	$fp_7 = 0.00196689$; $fp_j = 0\ \forall j \neq 7$

Table 11: DataCapture System: Predicted vs. Measured Reliability

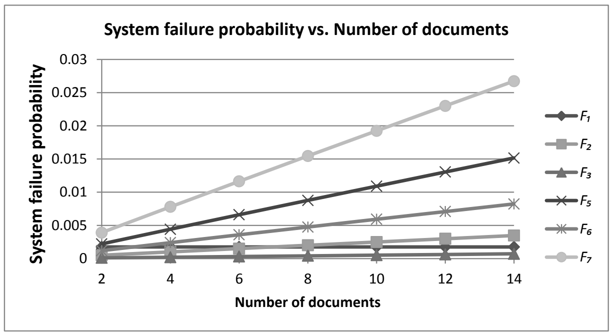
Component Instance/ Provided service	**Predicted reliability**	**Measured reliability**	**Difference**	**Error (%)**
UI/handleClientRequest	0.959148	0.9595	0.000352	0.037

By using a script, it is possible for us to calculate the failure probabilities of failure types for internal activities (as in Table 10), the *agreementOfErrorsVector* of the *MVPStructure*: $(p_2 = 0.2, p_3 = 0)$, and the measured system reliability from the execution traces. For the predicted reliability of the system, a system reliability model is created with the support from our reliability modeling schema and then used as the input for our reliability prediction tool. Table 11 compares the predicted system reliability and the measured system reliability. This comparison gives evidence that our approach give a reasonably accurate reliability prediction in this case.

Fig. 16a provides more detail and shows the probability of a system failure due to a certain failure type. Because the *MVPStructure* prevents *TimingOCRFailures* (F_4) of services *doOCR* of components *OCREngines* from manifesting as *TimingOCRFailures* after the *MVPStructure*'s execution, the probability that the system fails with a *TimingOCRFailure* is 0. F_7, F_5, and F_6 are the most likely failure types of a system failure. Thus, the software architect can recognize the need to introduce FTSs for these failure types. For example, the software architect puts an instance of component *DataAccessFaultTolerance* in the middle of the instance of component *DataCaptureControl* and the instance of component *DataAccess* as in Fig. 15. With this modification, the predicted system reliability increases by around 2%, from



(a)



(b)

Figure 16: DataCapture System: Sensitivity analyses.

0.959148 to 0.978182 under the assumption of perfect error detection for failures of *StoringDataFailure* (F_7). Via this example, it is possible to see that a FTS can be introduced into the system without modifying the existing service implementations and with just a few necessary changes while nearly all modeling parts can be reused.

Fig. 16b shows the sensitivity of system failure probability per failure type to the number of documents (i.e. a change to the usage profile). The curve corresponding to *TimingOCRFailure* (F_4) has not been plotted because the values for it are always 0. As expected, the system failure probability rises with the number of documents for all failure types excepting *HandlingRequestFailure* (F_1). This is because excepting *HandlingRequestFailure*, the other failure types are related to activities within the looping structure with loop count *numberOfDocuments*.

7 Assumptions and Limitations

Similar to some related approaches (e.g. [3,6,16,18], we assume that *components fail independently* and *a component failure, without FTMs to handle, leads to a system failure*. This means that the impact of the error propagation among components is neglected. We refer to our former work [12] for an analysis of the influence of the error propagation in reliability prediction of component-based software systems with different execution models, including sequential, parallel, and primary-backup fault tolerance executions.

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 execution history. This Markovian assumption limits the applicability of our approach in different application domains. However, many real-life applications have been proved to satisfy this Markovian assumption at the component-level [3]. Our approach can be adapted to any higher order Markov models to increase the applicability scope. We confirm this because the problem of Markovian assumption in reliability modeling and prediction was treated deeply by Goseva et al. [14]. 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, 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 [28]. 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 [17]. 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 [14].

Currently, the parameter values in our approach are fixed constants. They cannot be adapted to take into consideration factors such as component state or system state at run-time. Such considerations are left as a topic for future work.

8 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. Via two case studies, 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. This kind of helps can lead to more reliable software systems in a cost-effective way because potentially high costs for late life-cycle changes for reliability improvements can be avoided.

We plan to completely integrate with our former work [12], to extend with the more complex error propagation for concurrent executions, to include more software FTSs, and to validate further our approach. These extensions will further increase the applicability of our approach.

Acknowledgments

This work was supported by 322 FIVE-JAIST (Vietnam-Japan) program and JSPS KAKENHI Grant Number 23500060.

References

[1] T.-T. Pham and X. Défago, "Reliability prediction for component-based software systems with architectural-level fault tolerance mechanisms," in *Proc. of the 8th International Conference on Availability, Reliability and Security (ARES'13), Regensburg, Germany*. IEEE, September 2013, pp. 11–20.

[2] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*. Artech House Publishers, 2001.

[3] R. C. Cheung, "A user-oriented software reliability model," *IEEE Transactions on Software Engineering*, vol. 6, no. 2, pp. 118–125, March 1980.

[4] V. Cortellessa, H. Singh, and B. Cukic, "Early reliability assessment of UML based software models," in *Proc. of the 3rd International Workshop on Software and Performance (WOSP'02), Rome, Italy*. ACM, July 2002, pp. 302–309.

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

[6] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo, "Reliability prediction for component-based software architectures," *Journal of Systems and Software*, vol. 66, no. 3, pp. 241–252, June 2003.

[7] 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 International Symposium on Software Reliability Engineering (ISSRE'06), Raleigh, NC, USA*. IEEE, November 2006, pp. 299–310.

[8] W.-L. Wang, D. Pan, and M.-H. Chen, "Architecture-based software reliability modeling," *Journal of Systems and Software*, vol. 79, no. 1, pp. 132–146, January 2006.

[9] 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 and Sons, 1995, pp. 109–138.

[10] S. S. Gokhale, M. R. Lyu, and K. S. Trivedi, "Reliability simulation of fault-tolerant software and systems," in *Proc. of the Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS'97), Taipei, Taiwan*. IEEE, December 1997, pp. 167–173.

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

[12] T.-T. Pham and X. Défago, "Reliability prediction for component-based systems: Incorporating error propagation analysis and different execution models," in *Proc. of the 12th International Conference on Quality Software (QSIC'12), Xi'an, Shaanxi, China*. IEEE, August 2012, pp. 106–115.

[13] S. S. Gokhale, "Architecture-based software reliability analysis: Overview and limitations," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 1, pp. 32–40, January–March 2007.

[14] 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, October 2003.

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

[16] V. S. Sharma and K. S. Trivedi, “Quantifying software performance, reliability and security: An architecture-based approach,” *Journal of Systems and Software*, vol. 80, no. 4, pp. 493–509, April 2007.

[17] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik, “Early prediction of software component reliability,” in *Proc. of the 30th International Conference on Software Engineering (ICSE’08), Leipzig, Germany*. ACM, May 2008, pp. 111–120.

[18] Z. Zheng and M. R. Lyu, “Collaborative reliability prediction of service-oriented systems,” in *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE’10) Cape Town, South Africa*. ACM, May 2010, pp. 35–44.

[19] P. Popic, D. Desovski, W. Abdelmoez, and B. Cukic, “Error propagation in the reliability analysis of component based systems,” in *Proc. of the 16th International Symposium on Software Reliability Engineering (ISSRE’05), Chicago, IL, USA*. IEEE, November 2005, pp. 53–62.

[20] V. Cortellessa and V. Grassi, “A modeling approach to analyze the impact of error propagation on reliability of component-based systems,” in *Proc. of the 10th International Conference on Component-based Software Engineering (CBSE’07), Medford, MA, USA*. Springer-Verlag, July 2007, pp. 140–156.

[21] A. Mohamed and M. Zulkernine, “On failure propagation in component-based software systems,” in *Proc. of the 8h International Conference on Quality Software, (QSIC’08), Oxford, UK*. IEEE, August 2008, pp. 402–411.

[22] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, “Reliability analysis of component-based systems with multiple failure modes,” in *Proc. of the 13th International Conference on Component-Based Software Engineering (CBSE’10), Prague, Czech Republic*. Springer-Verlag, June 2010, pp. 1–20.

[23] F. Brosch, B. Buhnova, H. Koziolek, and R. Reussner, “Reliability prediction for fault-tolerant software architectures,” in *Proc. of the 7th International Conference on the Quality of Software Architectures (QoSA’11), Boulder, CO, USA*. ACM, June 2011, pp. 75–84.

[24] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, January 2004.

[25] S. Bernardi, J. Merseguer, and D. C. Petriu, “A dependability profile within MARTE,” *Software and Systems Modeling*, vol. 10, no. 3, pp. 313–336, July 2011.

[26] K. S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications (Second Edition)*. John Wiley and Sons, 2001.

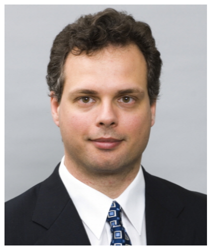
[27] “Reliability modeling and prediction,” http://reliabilitymodeling.codeplex.com, November 2013.

[28] M. R. Lyu, *Handbook of Software Reliability Engineering*. McGraw-Hill, 1996.

Author Biography

Thanh-Trung Pham is a PhD student in School of Information Science, Japan Advanced Institute of Science and Technology (JAIST). He received his M.Sc. in Information Technology in 2007 from Hanoi University of Science and Technology. His research interests include model-driven software development, service-oriented architectures, component-based software engineering, software reliability modeling and prediction.

François Bonnet is an assistant professor in the School of Information Science at the Japan Advanced Institute of Science and Technology (JAIST) since 2013. He obtained his M.S. from the ENS Cachan at Rennes, France in 2006 and his Ph.D. from the University of Rennes 1 in 2010. From 2011 to 2012, he was a JSPS postdoctoral fellow at JAIST. His research focuses mainly on theoretical distributed computing and robot computing.



Xavier Défago is an associate professor at the Japan Advanced Institute of Science and Technology (JAIST) since 2003. He obtained his Ph.D. in Computer Science in 2000 from the Swiss Federal Institute of Technology in Lausanne (EPFL, Switzerland). In addition, from 1995 to 1996, he also worked at the NEC C&C Research Labs in Kawasaki (Japan), and has been a researcher for the PRESTO program "Information and System" of the Japan Science and Technology Agency (JST) from 2002 to 2006. He was an invited researcher for CNRS in France, both at LIP6, UPMC, Paris, and at I3S, UNS, Inria Sophia Antipolis in 2013. He is a regular member of the IFIP working group 10.4 on dependable computing and fault-tolerance. He is also a member of ACM, IEEE, IPSJ, EATCS. His research interests include distributed algorithms, fault-tolerance and dependability, group communication and middleware, and cooperative autonomous mobile robot networks.