Reliability Prediction for Component-based Software Systems

Thanh-Trung Pham[a], Xavier Défago[a], Quyet-Thang Huynh[b]

[a]*School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Nomi, Ishikawa, Japan*
[b]*School of Information and Communication Technology, Hanoi University of Science and Technology, Hanoi, Vietnam*

Abstract

One of the most important quality attributes of a software system beyond its functional attributes is its reliability. Techniques for predicting reliability of a software system based on the design models can help software architects in evaluating the impact of their design decisions on the system reliability. This can help to make the system more reliable and avoid costs for fixing the implementation. However, existing reliability prediction approaches for component-based software systems are limited in their applicability because they either neglect or do not support modeling explicitly several factors which influence the system reliability: (i) error propagation, (ii) software fault tolerance mechanisms, and (iii) concurrently present errors. In this paper, we present a reliability modeling and prediction approach for component-based software systems that considers explicitly these reliability-relevant factors. Our approach offers a reliability modeling schema whose models are automatically transformed by our reliability prediction tool into Markov models for reliability predictions and sensitivity analyses. We evaluate our approach in two case studies with reliability predictions and sensitivity analyses. Via these two case studies, we demonstrate its applicability in supporting design decisions.

Keywords: Reliability modeling and prediction, component-based software systems, error propagation, software fault tolerance mechanisms, error detection and error handling, concurrently present errors, multiple execution models.

1. INTRODUCTION

To meet the increasing requirements for software support from many different areas, software systems become increasingly complex. In this situation, to assure the system reliability, i.e. its ability to deliver its intended service to users, many classes of techniques in software reliability engineering have been deployed throughout the development process. One of such classes of techniques is the class of reliability prediction techniques based on the design models. These techniques can help to make the system more reliable by assisting software architects in evaluating the impact of their design decisions on the system reliability. This can help to save costs, time, and efforts significantly by avoiding implementing software architectures that do not meet the reliability requirements.

However, existing reliability prediction approaches for component-based software systems suffer from the following drawbacks and therefore are limited in their applicability and accuracy. In essence, these drawbacks are consequences of the assumption that components fail independently and each component failure leads to a system failure, which is common to most existing reliability models for component-based software systems [1].

1.1. Ignoring Error Propagation

According to Avizienis et al. [2], an error is defined as the part of a system's total 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

Email addresses: thanhtrung.pham@jaist.ac.jp (Thanh-Trung Pham), defago@jaist.ac.jp (Xavier Défago), thanghq@soict.hut.edu.vn (Quyet-Thang Huynh)

service to deviate from correct service. The deviation can be manifested in different ways, corresponding to the system's different failure types. For example, two failure types that could be defined are content failures (the content of a system service's output deviates from the correct one) and timing failures (the delivery time of a system service deviates from the correct one).

Errors can arise because of internal faults. For example, a bug in the code implementing a component is an internal fault. This fault causes an error in the internal state of the component if the code is executed. Errors can arise because of external faults. For example, an erroneous input appears as an external fault to a component and propagates the error into the component via its interface. Errors can also arise because of both internal faults and external faults, e.g. an erroneous input (an external fault) is also the application of an input (the activation pattern) to a component that causes the code with a bug (an internal fault) of the component to be executed.

However, not all errors in a component lead to component failures. A component failure occurs only when an error in a component propagates within the component up to its interface. Similarly, not all component failures lead to system failures. A component failure in a component-based system is an error in the internal state of the system. This error leads to a system failure only when it propagates through components in the system up to the system interface.

During this propagation path, an error can be detected,[1] and therefore stops from propagating, e.g. an erroneous input is detected by error detection of components. An error can be masked, e.g. an erroneous value is overwritten by the computations of component services before being delivered to the interface. An error can be transformed, e.g. a timing failure received from another component service may cause the current component service to perform computations with outdated data, leading to the occurrence of a content failure. An error can also be concurrently present with another error, e.g. a content failure received from another component service is also the activation pattern that causes the current component to perform unnecessary computations with corrupted data, leading to the concurrent presence of a content failure and a timing failure.

It is possible to see that the reliability of a component-based software system, defined as the probability that no system failure occurs, is strongly dependent on the error propagation path. The challenge of analyzing the reliability of a component-based software system becomes even more significant when the system embodies parallel and fault tolerance execution models. A parallel execution model has multiple components running in parallel, resulting in many concurrent error propagation paths. A fault tolerance execution model has a primary component and backup components, and the order of their executions is highly dependent on their error detection and error handling. This results in many different error propagation paths.

As an example, in a parallel execution model, an error in the input for the components running in parallel may be masked by the computations of a certain number of components while the computations of the other components may transform the error into multiple errors of different failure types, leading to a set of multiple errors of different failure types in the output of the execution model. As another example, in a fault tolerance execution model, an error of a certain failure type in the input for the primary component and backup components may be transformed into an error of other failure type by the primary component without being detected, leading to an error in the output of the execution model without activating the backup components.

Although error propagation is an important element in the chain that leads to a system failure, many approaches (e.g. [3, 4, 5, 6, 7, 8]) do not consider it. They assume that any error arising in a component immediately manifests itself as a system failure, or equivalently that it always propagates (i.e. with probability 1.0 and with the same failure type) up to the system interface [9]. On the other hand, approaches that do consider error propagation (e.g. [9, 10]) typically only consider it for a single sequential execution model. Since modern software systems often embody not just a single sequential execution model, but also parallel and fault tolerance execution models to achieve multiple quality attributes (e.g. availability, performance, reliability), ignoring the consideration of error propagation for these two latter execution models makes these approaches no more suitable for modeling complex software systems with different execution models.

[1]Software fault tolerance mechanisms, if any, can then provide error handling for the detected error.

1.2. Ignoring Software Fault Tolerance Mechanisms

Software Fault Tolerance Mechanisms (FTMs) are often included in a software system and constitute an important means to improve the system reliability. FTMs 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) [11]. Their reliability impact is highly dependent on the whole system architecture and usage profile. For example, if a FTM is never executed under a certain usage profile, its reliability impact is considered as nothing.

Analyzing the reliability impact of FTMs becomes apparently a challenge when they are applied at architecture level, in a component-based software system because: (1) FTMs can be employed in different parts of a system architecture, (2) In a system architecture, there are usually multiple changeable points to create architecture variants, e.g. substituting components with more reliable variants, running components concurrently to improve performance.

Many approaches (e.g. [7, 12, 13]) do not support modeling FTMs. This forces modelers to implicitly model FTMs of a software system, if any, via decreasing software failure probabilities. Some approaches step forward and offer basic fault tolerance expressiveness which are limited to specific FTMs and failure conditions (e.g. [14, 15]). They lack flexible and explicit expressiveness of how both 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 consequence, they 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. [16, 17, 18]). But these so-called non-architectural models do not reflect the system architecture and usage profile (i.e. component services, control flow transitions between them and sequences of component service calls). 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.

1.3. Ignoring Concurrently Present Errors

Situations involving multiple failures are frequently encountered. System failures are often turned out on later examination to have been caused by different errors [2]. For example, (1) failures of component services performing computations in parallel are concurrently present errors in the system, (2) a content failure received from another component service is also the application of an input (the activation pattern) that causes the current component to perform unnecessary computations with corrupted data, leading to the concurrent presence of a content failure and a timing failure.

However, to the best of our knowledge, existing approaches do not support modeling concurrently present errors. Neglecting concurrently present errors can leads to inaccurate prediction results because there exist system failures that cannot be covered by existing approaches, which is confirmed by Hamill et al. [19] with two large, real-world case studies (GNU Compiler Collection (GCC) and NASA Flight Software).

Contribution and Structure

The contribution of this paper is a novel approach of reliability modeling and prediction for component-based software systems that considers explicitly error propagation, software FTMs, and concurrently present errors. The approach supports modeling error propagation for multiple execution models, including sequential, parallel, and fault tolerance execution models. Via an explicit and flexible definition of reliability-relevant behavioral aspects (i.e. error detection and error handling) of FTMs, the approach offers an effective evaluation of their reliability impact in the dependence of the whole system architecture and usage profile. The approach accounts for concurrently present errors by introducing a hierarchical tree of multiple failure types. The approach offers a reliability modeling schema whose models are automatically transformed by our reliability prediction tool 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.

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 modeling component reliability specifications and system reliability models using 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

In contrast to software reliability growth models which treat software systems as black boxes, our approach belongs to the field of component-based software reliability modeling and prediction which treats software systems as a composition of software components. Several surveys in this field are available [1, 6, 20]. In the following, we survey the most related approaches with regard to the three gaps identified above. After the summary of the findings, we discuss our preliminary work and its relation to this paper.

2.1. Consideration of Error Propagation

Cheung's approach [5], one of the first approaches, expresses the control flow between components in a software system using an absorbing Discrete Time Markov Chain (DTMC). Some recent approaches extend Cheung's approach to support different architectural styles [15] and to combine reliability analysis and performance analysis [8] but do not consider error propagation. Further approaches building upon the Cheung's model such as the approach of Lipton et al. [21] which takes interface failures and network connection failures into account, the approach of Sharma et al. [14] which supports modeling component restarts and retries, also do not consider error propagation.

The approach of Reussner et al. [7] is based on Rich Architecture Definition Language (RADL) but employs the same underlying theory as Cheung's approach for reliability prediction. The approach of Brosch et al. [3] extends the approach of Reussner et al. to consider explicitly the influences of system usage profile and execution environment on the system reliability. However, these approaches do not consider the influence of error propagation on the system reliability.

The approach of Cheung et al. [4] uses hidden Markov models to determine component failure probabilities and does not include calls to other components, thus ignores error propagation. The approach of Sato et al. [22] combines a system model of interacting system services with a resource availability model but does not consider application-level software failures, thus also ignores error propagation. The approaches of Grassi [23] and Zheng et al. [24] aim at reliability prediction for Service-Oriented Architectures (SOA). The approach of Grassi considers recursively composed services, where each service may invoke multiple external services in order to complete its own execution. The approach of Zheng et al. employs a workflow description for composite services with sequential, looping, and parallel structures. However, these approaches neglect the impact of error propagation between services.

Scenario-based approaches such as the approach of Yacoub et al. [25] which constructs component dependency graphs from component sequence diagrams as a basic for reliability prediction, the approaches of Cortellessa et al. [12] and Goseva et al. [13] which employ UML diagrams annotated with reliability properties, the approach of Rodrigues et al. [26] which is based on message sequence charts, also do not consider error propagation.

The approaches [14, 15, 23, 27] that support modeling FTMs (see also Section 2.2) allow to express how FTMs prevent the occurrence of system failures in the presence of faults that have already been activated and resulted in errors within the system. However, in analogy with the approaches that do not consider error propagation, they assume that any error arising in a component always propagates (i.e. with probability 1.0 and with the same failure type) up to the system interface or until FTMs get involved to provide error handling. This is not always valid because the error can be masked or transformed by the computations of components during the propagation path, leading to imperfect error propagation (i.e. with probability less than 1.0 or with varying failure type).

Some approaches have proposed taking error propagation between components into account. The approach of Popic et al. [10] assumes that each error arising within a component always causes a system failure and at the same time, it can also propagate to other components to affect their reliability. According to us, the assumption of immediate failure conflicts with the reason of error propagation to other components. The approach of Cortellessa et al. [9] assumes that the internal failure probability and the error propagation probability of each component are independent of each other. As a consequence of this independence assumption, they argue that when a component fails, it always transmits an error to the next component irrespective of whether it has received or not an erroneous input from the previous component. This is not always valid because the failed computations of a component can overwrite the error from its erroneous input and therefore can produce a correct output. The approaches of Filieri et al. [28] and Mohamed et al. [29] support multiple failure types when considering error propagation. However, all these approaches consider error propagation only for a single sequential execution model, ignoring the consideration of error propagation for parallel and fault tolerance execution models which are often used by modern software systems.

2.2. Consideration of Software Fault Tolerance Mechanisms

Many approaches do not support modeling FTMs (e.g. [5, 7, 12]). The approaches [9, 10, 28, 29] that consider explicitly error propagation introduce error propagation probabilities to model the possibility of propagating component failures. The complement of an error propagation probability 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 take FTMs into account. The approach of Sharma et al. [14] supports modeling component restarts and component retries. The approach of Wang et al. [15] supports different architectural styles including fault tolerance architectural style. The approach of Grassi [23] introduces the OR completion model denoting the possibility that a composed service requires only 1 out of n invoked external services to be successful in order for its own execution to succeed. 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. The approach of Brosch et al. [27] extends Recovery Blocks (RB) to flexibly describe error handling of FTMs but still does not consider the influences of error detection of FTMs on the control and data flow within components. More concretely, these approaches assume that when there is an error of a certain failure type caused by a component failure, a FTM can always handle the error if it aims to handle errors of that failure type. This means that the FTM perfectly detects errors of that failure type (i.e. with error detection probability 1.0). However, in reality, error detection is not perfect and therefore, a FTM may let errors caused by component failures propagate to its output without activating its error handling, which in turns influences the control and data flow within the component service containing this FTM. 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. [16] 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. [18] evaluate RB and NVP using generalized stochastic Petri nets. Gokhale et al. [17] 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.

2.3. Consideration of Concurrently Present Errors

To the best of our knowledge, existing approaches do not support modeling concurrently present errors. In other words, they support only a single error at any time.

Table 1: Most Related Approaches.

Authors	**Year**	**Error propagation**	**Software FTMs**	**Concurrently present errors**
Grassi [23]	2004	-	(✓)	-
Popic et al. [10]	2005	(✓)	-	-
Wang et al. [15]	2006	-	(✓)	-
Sharma et al. [14]	2006	-	(✓)	-
Cortellessa et al. [9]	2007	(✓)	-	-
Mohamed et al. [29]	2008	(✓)	-	-
Filieri et al. [28]	2010	(✓)	-	-
Brosch et al. [27]	2011	-	(✓)	-
Pham et al. [This paper]	2013	✓	✓	✓

2.4. Summary of Findings

With regard to three gaps identified above, our findings on most related approaches can be summarized in Table 1. A hyphen mark means that an approach does not support the feature and a check mark in parentheses means that an approach supports the feature but is limited in several aspects. Error propagation are supported by some approaches but they introduce new assumptions which, according to us, deserves further investigation about their soundness, and/or consider error propagation only for a single sequential execution model. None of these approaches supports a combined consideration of error propagation for sequential, parallel, and fault tolerance execution models. Some approaches support modeling software FTMs but they lack flexible and explicit expressiveness of how both error detection and error handling of FTMs influence the control and data flow within components. Concurrently present errors are not supported by any approaches.

While our approach receives benefits from the experiences gained in the field by these approaches, it also presents unique features that enhance the state of the art, including (1) a combined consideration of error propagation for sequential, parallel, and fault tolerance execution models, (2) an explicit and flexible expressiveness of reliability-relevant behavioral aspects (i.e. error detection and error handling) of FTMs, and (3) the consideration of concurrently present errors.

2.5. Preliminary Work

Prevalent approaches in the field can be classified into main classes [1]: (i) path-based methods which consider explicitly the probabilities of possible component execution paths, and (ii) state-based methods which use probabilistic control flow graphs to model the usage of components. Our approach is in the class of state-based methods and for sequential executions, we assume that the control transitions between components have the Markov property.

In the first work [30], 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, primary-backup is the only FTM supported by the approach and the approach does not support modeling concurrently present errors.

In the second work [31], we extended the core model (i.e. fundamental modeling steps and basic modeling elements) of the first work to offer an explicit and flexible definition of 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. However, we neglected the impact of error propagation and did not consider concurrently present errors.

This paper goes beyond our former work through an extended analysis of error propagation with a hierarchical tree of multiple failure types for sequential, parallel, and different fault tolerance execution models, a support for modeling concurrently present errors, a more comprehensive validation, and a far more detailed description and discussion of our approach.

3. COMPONENT-BASED RELIABILITY PREDICTION

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment [32]. A component has its behavior defined in terms of provided and required interfaces. This information is sufficient to assemble components and check their interoperability. However, in order to predict the reliability of a component-based software architecture, additional information about each component is required.

Since there exists a strict separation between component developers and software architects in Component-Based Software Engineering (CBSE), it is necessary to consider these two roles when creating specifications (or models) to capture the additional information. Therefore, component developers implement components and provide not only component functional specifications but also component reliability specifications. Software architects use these component reliability specifications and provide additionally usage profiles in order to predict the reliability of planned system architectures. Later, they assemble the actual component implementations.

A component reliability specification needs to describe the behaviors of services provided by the component, i.e. how provided services of the component are related to required services and internal activities of the components in terms of frequencies and probabilities. From that, by assembling these specifications and providing additionally usage profiles, software architects create system reliability models reflecting the control and data flow throughout the whole planed system architectures for reliability predictions without referring to component internals. In Section 4, we introduce our reliability modeling schema that supports component developers to create component reliability specifications and software architects to create system reliability models.

Our approach follows repetitively six steps as depicted in Fig. 1. In Step 1, component developers provide component reliability specifications. A component reliability specification includes reliability-related probabilities (e.g. failure probabilities, error propagation probabilities) and call propagations to required services for each provided service of the component. How to determine these probabilities (e.g. [4, 33, 34]) is beyond the scope of this paper. For already implemented components, call propagations can be derived from static code analysis or dynamic monitoring.

In Step 2, software architects create a system reliability model by assembling component reliability specifications following a planed system architecture and providing additionally a usage profile for the complete system (i.e. interacting directly to users or other systems).

In Step 3, from the system reliability model, it is possible to describe the control flow throughout the whole system architecture by propagating requests at the system boundary to individual components. Because each component reliability specification includes call propagations to required services of the component, this method works recursively. The resulting model can be transformed into Markov models.

In Step 4, by analyzing the Markov models, a reliability prediction for each provided services at the system boundary can be derived, based on the reliability-related probabilities of components inside the system architecture. To support Step 3 and Step 4, we provide a reliability prediction tool whose transformation for reliability prediction is described in detail 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.

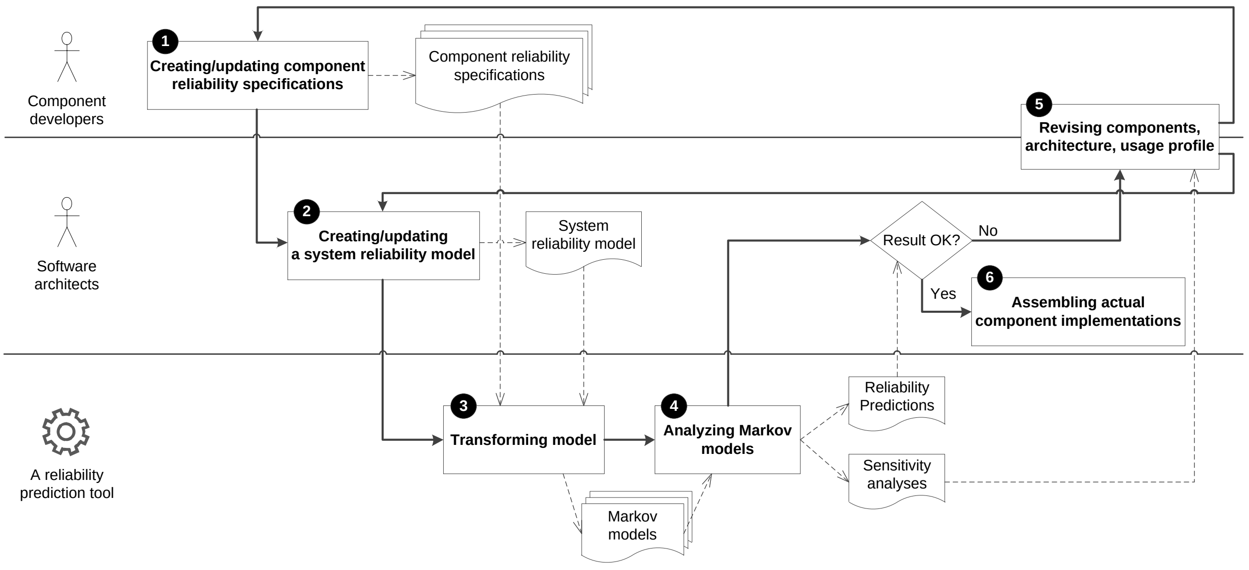

Component developers
Software architects
A reliability prediction tool
1 Creating/updating component reliability specifications
Component reliability specifications
5 Revising components, architecture, usage profile
2 Creating/updating a system reliability model
System reliability model
Result OK?
No
Yes
6 Assembling actual component implementations
3 Transforming model
4 Analyzing Markov models
Reliability Predictions
Sensitivity analyses
Markov models

Figure 1: Component-based reliability prediction.

If the prediction results show that given reliability requirements cannot be meet, 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 (e.g. the number of retries or replicated instances) of FTMs; 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 requirements, and software architects assemble the actual component implementations following the system architecture.

4. RELIABILITY MODELING

In this section, we describe our reliability modeling schema which supports component developers to create component reliability specifications and software architects to create system reliability models. 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[2] [35] 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.

4.1. Component Reliability Specifications

4.1.1. Services, components, and service implementations

In our approach, component developers are required to provide component reliability specifications. Fig. 2[3] shows an extract of our reliability modeling schema with modeling elements which supports component developers to create component reliability specifications. Component developers model components,

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

[3]Refer to our project website [36] for the full reliability modeling schema.

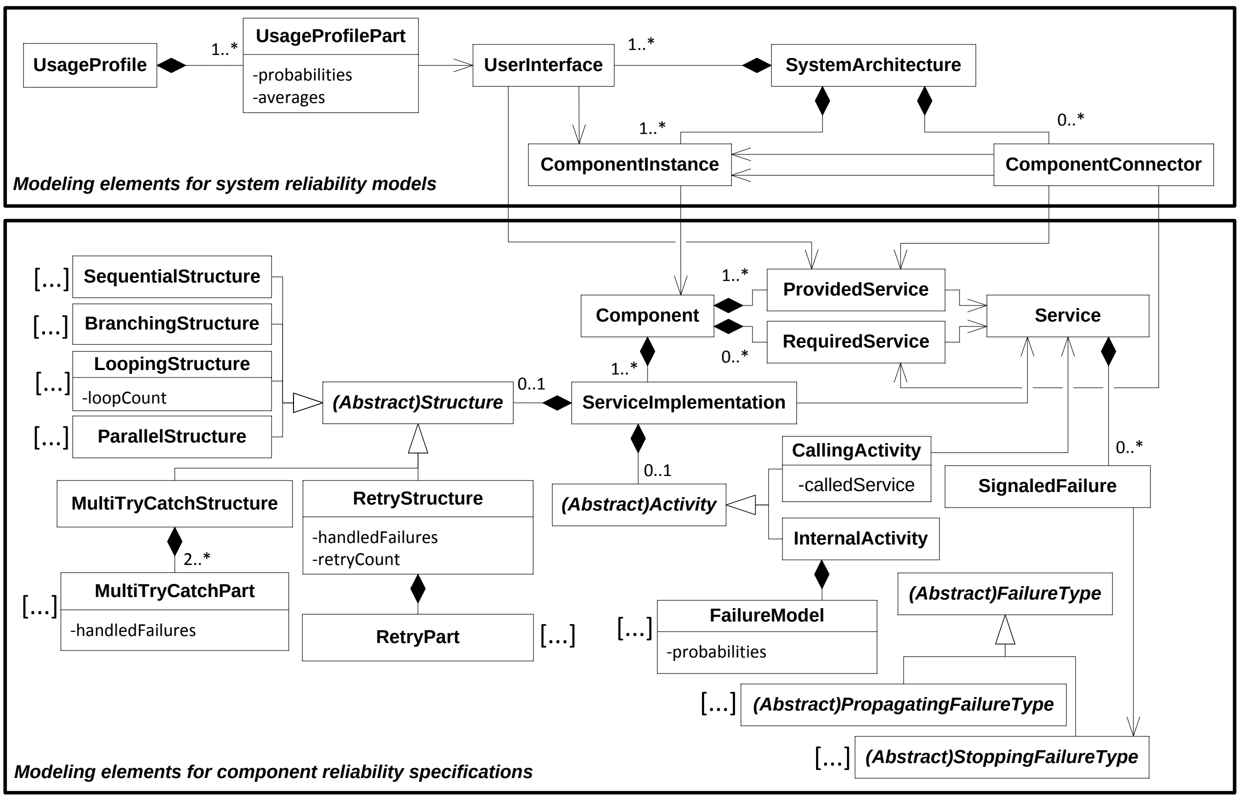


Figure 2: Modeling elements in our reliability modeling schema.

services, and service implementations via modeling elements: *Component*, *Service*, and *ServiceImplementation*, respectively. Components are associated with services via *RequiredService* and *ProvidedService*.

A service implementation (*ServiceImplementation*) is used to describe the behavior of each service provided by a component, i.e. describe the activities to be executed when a service (*Service*) in the provided services of the component is called. Therefore, a component can contain multiple service implementations. A service implementation can include activities (*Activity*) and control flow structures (*Structure*).

There are two activity types, namely internal activities and calling activities.

- An internal activity (*InternalActivity*) represents a component's internal computation.
- A calling activity (*CallingActivity*) 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.

There are four standard types of control flow structures supported by our reliability modeling schema, including sequential structures, branching structures, looping structures and parallel structures (Fig. 3).

- In a sequential structure (*SequentialStructure*), sequential parts (*SequentialPart*) are executed sequentially, i.e. only a single part is executed at any time. The control is transferred to one (and only one) of its successors upon the completion of a part. The selection of the succeeding part is always deterministic.

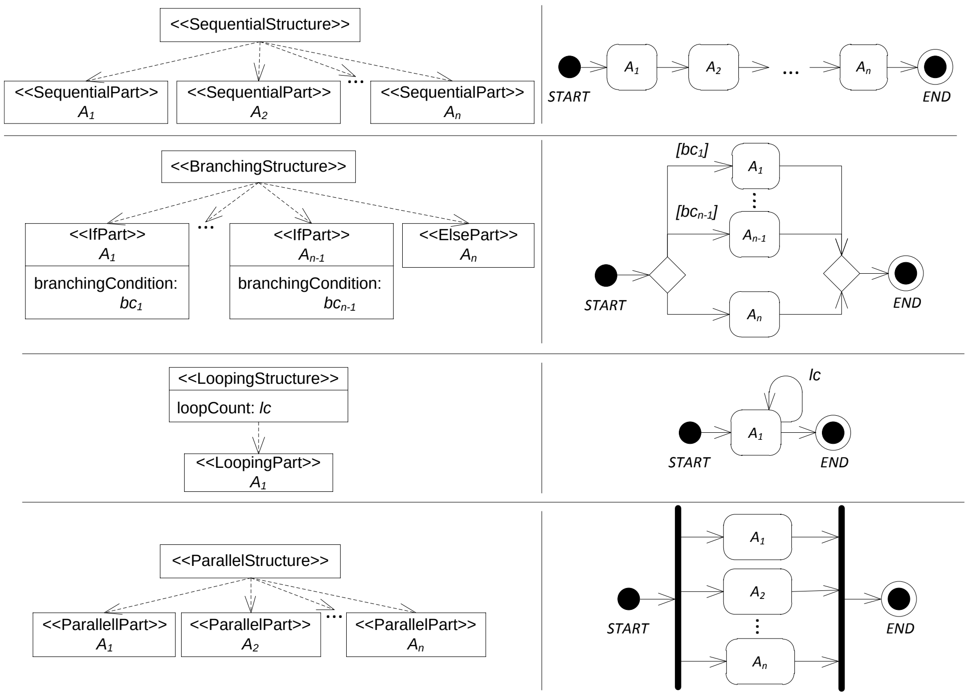


Figure 3: Supported control flow structures and their execution semantics: Sequential structure, branching structure, looping structure, and parallel structure.

- A branching structure (*BranchingStructure*) inherits the characteristics of a sequential structure. The difference is that the selection of the succeeding part (*IfPart* or *ElsePart*) depends on branching conditions (i.e. Boolean expressions).

- In a looping structure[4] (*LoopingStructure*), there is a single looping part (*LoopingPart*) which is repeated the loop count times. Infinite loop count are not allowed. Looping structures can include other looping structures but cannot have multiple entry points and cannot be interconnected.

- Parallel structures (*ParallelStructure*) are commonly used in concurrent execution environments, in which a set of parallel parts (*ParallelPart*) is usually executed simultaneously to improve performance. In Fig. 3, parallel parts *ParallelPart* A_1, *ParallelPart* A_2, ..., *ParallelPart* A_n are running in parallel. These parts cooperatively work on the structure's input and synchronously release the control to end the structure's execution.

Example 1. *Fig. 4 shows an example of component reliability specification. The component* C_2 *provides two services:* S_1 *and* S_2 *and requires three services:* S_3, S_4, S_5.

- *Service implementation for provided service* S_1 *is a sequential structure executing an internal activity, a branching structure and another internal activity in sequence. The branching structure either leads to a parallel structure, if* $[Y = true]$, *or to a calling activity to call required service* S_5 *otherwise. The parallel structure executes two calling activities to call required services* S_3 *and* S_4 *in parallel.*

- *Service implementation for provided service* S_2 *is a looping structure executing an internal activity* Z *times.*

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

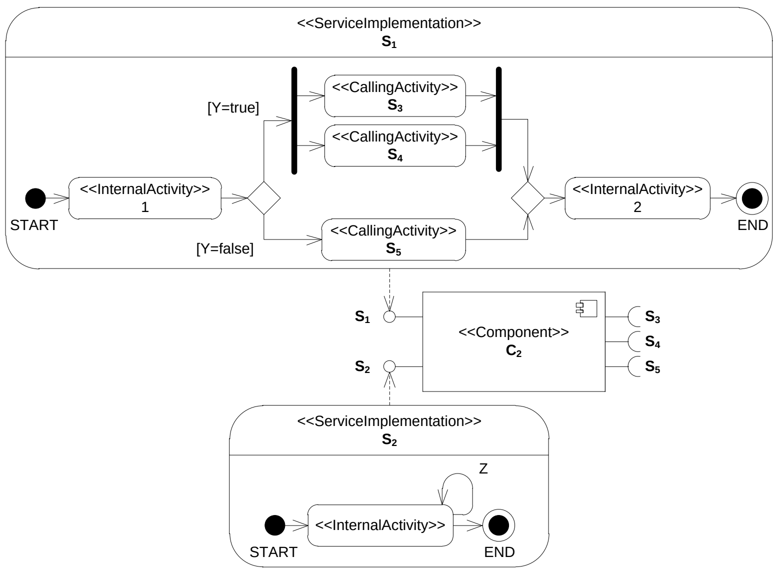


Figure 4: An example of component reliability specification.

Remark. A service implementation in our model is an abstraction of the behavior of a service provided by a component. Control flow structures are included only when they influence calls to required services. A single internal activity modeled with a failure model may represent thousands of line of code. This abstraction focuses on the necessary properties for a component-based reliability prediction (i.e., failure probabilities, error propagation probabilities, and call propagations).

4.1.2. Failure models

In this section, we revisit the definitions of terms "error", "fault", and "failure", and reexamine the "pathology of failure: relationship between faults, errors, and failures" on the abstraction level of our approach in order to create a definition of failure models for internal activities.

In their taxonomy, Avizienis et al. [2] define an error as the part of a system's total 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 general, characterizing the failure types which may occur in a system is highly dependent on the specific system. For example, two failure types that could be defined are content failures (the content of a system service's output deviates from the correct one) and timing failures (the delivery time of a system service deviates from the correct one).

Errors can arise because of internal faults. For example, a bug in the code implementing an internal activity of a component is an internal fault. This fault causes an error in the internal state of the component if the code is executed. Errors can arise because of external faults. For example, an erroneous input appears as an external fault to a component and propagates the error into the component via its interface. Errors can also arise because of both internal faults and external faults, e.g. an erroneous input (an external fault) is also the application of an input (the activation pattern) to a component that causes the code with a bug (an internal fault) of the component to be executed.

However, not all errors in a component lead to component failures. A component failure occurs only when an error in a component propagates within the component up to its interface. Similarity, not all component failures lead to system failures. A component failure in a component-based system is an error in the internal state of the system. This error leads to a system failure only when it propagates through components in the system up to the system interface.

During this propagation path, an error can be detected and therefore stops from propagating, e.g. an erroneous input is detected by error detection of internal activities. An error can be masked, e.g. an erroneous

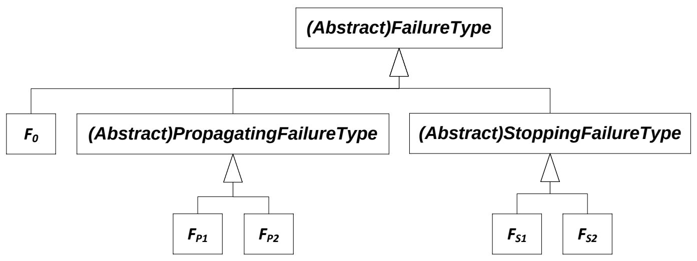

(Abstract)FailureType
F0
(Abstract)PropagatingFailureType
(Abstract)StoppingFailureType
FP1
FP2
FS1
FS2

Figure 5: An example of failure types.

value is overwritten by the computations of internal activities before being delivered to the interface. An error can be transformed, e.g. a timing failure received from another component service may cause the current component service to perform computations with outdated data, leading to the occurrence of a content failure. An error can also be concurrently present with another error, e.g. (1) a content failure received from another component service is also the activation pattern that causes the current component to perform unnecessary computations with corrupted data, leading to the concurrent presence of a content failure and a timing failure, (2) failures of component services performing computations in parallel are concurrently present errors in the system.

In order to take into consideration explicitly the whole set of factors identified above, component developers are required to model different failure types and failure models for internal activities of service implementations. A failure model for an internal activity captures the possibilities for errors after the internal activity's execution, including the possibility of being detected, the possibility of being masked, the possibility of being transformed, or the possibility of being concurrently present.

Component developers model different failure types (*FailureType*) by using the hierarchical tree of failure types (cf. Fig. 2). Except failure type F_0, a predefined failure type corresponding to the correct service delivery, component developers model a failure type by extending either *StoppingFailureType* or *PropagatingFailureType.* Failure types extending *StoppingFailureType* are related to errors that can be detected and signaled with a warning signal by the error detection of internal activities. When a failure type extending *StoppingFailureType* manifests itself after an internal activity's execution, this immediately leads to a signaled failure of this failure type. On the other hand, failure types extending *PropagatingFailureType* are related to errors that cannot be detected and signaled by the error detection of internal activities. When a failure type extending *PropagatingFailureType* manifests itself after an internal activity's execution, this propagates errors into another internal activity through an erroneous output of this failure type. For the sake of simplicity, failure types extending *StoppingFailureType* are called stopping failure types and failure types extending *PropagatingFailureType* are called propagating failure types.

Example 2. *Fig. 5 shows an examples of failure types: F_0 is the predefined failure type, F_{P1} and F_{P2} are propagating failure types, and F_{S1} and F_{S2} are stopping failure types.*

Component developers model a failure model (i.e. different failure types with their occurrence probabilities) for a internal activity via a composition between *InternalActivity* and *FailureModel.* In the literature, techniques for determining these probabilities have been discussed extensively (see Section 7 for more details) and are beyond the scope of this paper.

Definition 1. ***Failure Model***[5]

- *Let F_0 be a predefined failure type corresponding to the correct service delivery.*
- *Let $\mathcal{F_S}$ be the set of all stopping failure types $\{F_{S1}, F_{S2}, ..., F_{Su}\}$.*

[5]From here, we utilize the introduced symbols unless otherwise stated.

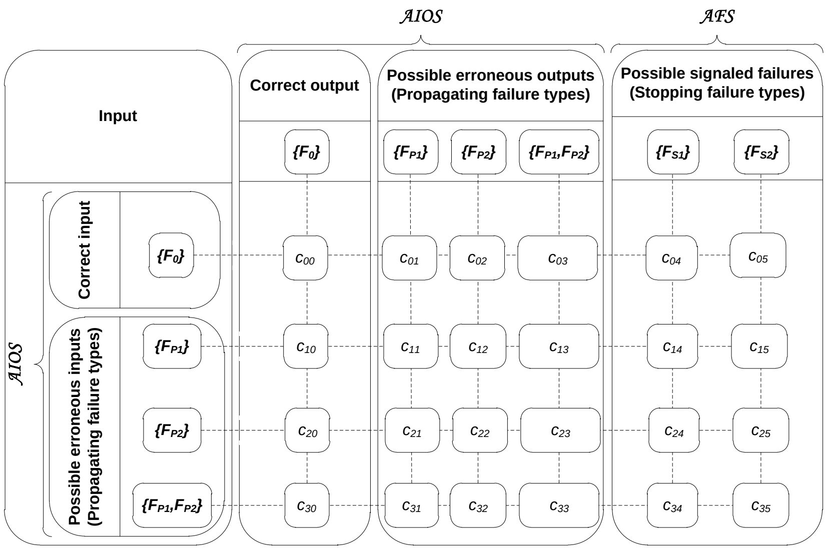


Figure 6: An example of failure model for an internal activity.

- *Let $\mathcal{F_P}$ be the set of all propagating failure types $\{F_{P1}, F_{P2}, ..., F_{Pv}\}$.*
- *Let $\mathcal{AIOS}$ be the Set of All sets of failure types for an internal activity's Input or Output $\{\{F_0\}\} \cup \left(2^{\mathcal{F_P}} \setminus \emptyset\right)$.*
- *Let $\mathcal{AFS}$ be the Set of All sets of failure types for an internal activity's signaled Failures $\{\{F_{S1}\}, ..., \{F_{Su}\}\}$.*
- *Then, a failure model (*FailureModel*) for an internal activity (*IA *for short) is defined by probabilities: $Pr_{IA}(I, FO)$, $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$, where $Pr_{IA}(I, FO)$ is the probability that* IA *signals a signaled failure of a failure type FO (when $FO \in \mathcal{AFS}$) or produces an output of failure types FO (when $FO \in \mathcal{AIOS}$) given that* IA *has received an input of failure types I. It holds that $\sum\limits_{FO \in (\mathcal{AFS} \cup \mathcal{AIOS})} Pr_{IA}(I, FO) = 1$ for all $I \in \mathcal{AIOS}$.*

Example 3. *Fig. 6 shows an example of failure model for an internal activity with $\mathcal{F_S} = \{F_{S1}, F_{S2}\}$, $\mathcal{F_P} = \{F_{P1}, F_{P2}\}$, $\mathcal{AFS} = \{\{F_{S1}\}, \{F_{S2}\}\}$, and $\mathcal{AIOS} = \{\{F_0\}, \{F_{P1}\}, \{F_{P2}\}, \{F_{P1}, F_{P2}\}\}$. It is possible to understand the internal activity's execution to follow its failure model as follows:*

- *The internal activity can receive a correct input: $\{F_0\}$. In this case, errors can arise because of the activity's internal faults. When these errors are detected and signaled with a warning signal by the error detection of the activity, then a signaled failure of a stopping failure type occurs: $\{F_{S1}\}$ with probability c_{04} or $\{F_{S2}\}$ with probability c_{05}. Otherwise, the activity produces an erroneous output of different propagating failure types: $\{F_{P1}\}$ with probability c_{01}, $\{F_{P2}\}$ with probability c_{02}, or $\{F_{P1}, F_{P2}\}$ (the concurrent presence of F_{P1} and F_{P2}) with probability c_{03}. In case there is no error during the activity's execution, the activity produces a correct output: $\{F_0\}$ with probability $c_{00} = 1 - \sum\limits_{j=1}^{5} c_{0j}$.*
- *The internal activity can receive an erroneous input of different propagating failure types: $\{F_{P1}\}$, $\{F_{P2}\}$, or $\{F_{P1}, F_{P2}\}$. In this case, beside the errors from the erroneous input, errors can arise because of the activity's internal faults. If the error detection of the activity detects and signals these errors with a warning signal, this leads to a signaled failure of a stopping failure type: $\{F_{S1}\}$ with probability c_{i4} or $\{F_{S2}\}$ with probability c_{i5} (with $i \in \{1, 2, 3\}$ when the erroneous input is $\{F_{P1}\}$, $\{F_{P2}\}$, or $\{F_{P1}, F_{P2}\}$, respectively). Otherwise, an erroneous output of different propagating failure*

types is produced by the activity: $\{F_{P1}\}$ with probability c_{i1}, $\{F_{P2}\}$ with probability c_{i2}, or $\{F_{P1}, F_{P2}\}$ with probability c_{i3}. In case these errors are masked by the activity's execution, there is a correct output: $\{F_0\}$ with probability $c_{i0} = 1 - \sum_{j=1}^{5} c_{ij}$

In our model, it is assumed that an internal activity receives both data and control transfer through its input and produces both data and control transfer through its output [9, 28]. A correct or erroneous output (of any propagating failure types), when received by an internal activity, becomes its correct or erroneous input (of the same propagating failure types), respectively. A signaled failure (of any stopping failure type), without any software FTMs to handle it, immediately leads to a system failure.

Remark. Our approach supports modeling concurrently present errors via the concurrent presence of propagating failure types. It also allows our approach to support modeling error propagation for parallel structures (see Section 5.1.4). Distinguishing between stopping failure types and propagating failure types enables our approach to support modeling error propagation for software FTMs (see Section 4.1.3). With the comprehensive failure model, our approach is able to model explicitly and flexibly error detection via internal activities, including correct error detection (e.g. with an erroneous input, the internal activity signals a signaled failure of a proper stopping failure type), a false alarm (e.g. with a correct input, the internal activity signals a signaled failure), as well as a false signaling of failure type (e.g. with an erroneous input, the internal activity signals a signaled failure of an improper stopping failure type).

4.1.3. Fault Tolerance Structures

In their paper [2], Avizienis et al. describe in detail the principle of FTMs. A FTM is carried out via error detection and system recovery. Error detection is to identify 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.

To support modeling FTMs, our reliability modeling schema provides Fault Tolerance Structures (FTSs), namely *RetryStructure* and *MultiTryCatchStructure*. Because in a FTM, error detection is a prerequisite for error handling and not all detected errors can be handled. Therefore, at most, a *RetryStructure* or a *MultiTryCatchStructure* can provide error handling only for signaled failures, which are consequences of errors that can be detected and signaled by error detection.

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. 7 shows a* RetryStructure *with a single* RetryPart*. After the* RetryPart*'s execution, possible signaled failures of stopping failure types $\{F_{S1}\}$, $\{F_{S2}\}$, or $\{F_{S3}\}$ (the field* possibleSignaledFailures*), or possible erroneous outputs of propagating failure types $\{F_{P1}\}$, $\{F_{P2}\}$, or $\{F_{P1}, F_{P2}\}$ (the field* possibleErroneousOutputs*) can occur. The* RetryStructure *can handle only signaled failures of $\{F_{S1}\}$ or $\{F_{S2}\}$ (the field* handledFailures*). This means that the structure handles signaled failures of these stopping failure types and retries the* RetryPart*. Signaled failures of $\{F_{S3}\}$ can not be handled, and therefore lead to signaled failures of the whole structure. Erroneous outputs of the* RetryPart*, which are consequences of errors that cannot be detected and signaled by error detection, lead to erroneous outputs of the whole structure. This procedure is repeated the number of times equal to the field* retryCount *(2 times in this example). For the last retry, signaled failures of $\{F_{S1}\}$, $\{F_{S2}\}$, or $\{F_{S3}\}$ all lead to signaled failures of the whole structure.*

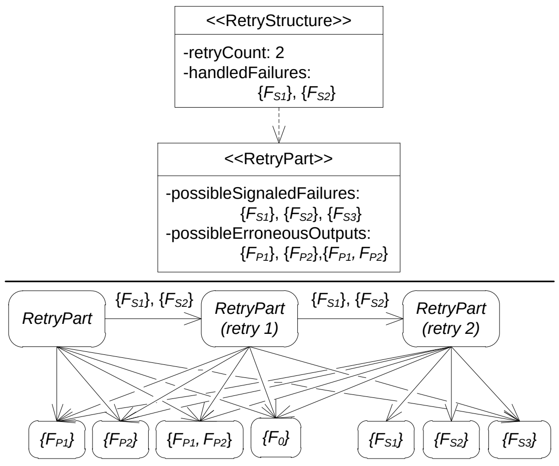
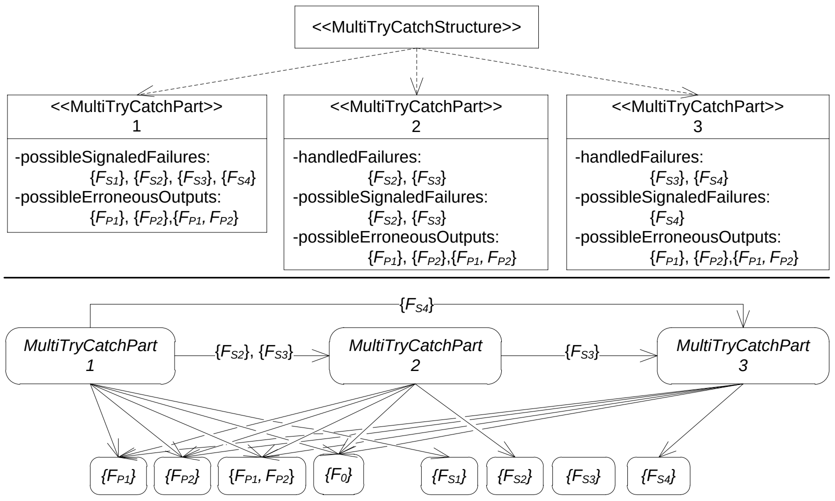


Figure 7: Semantics for a *RetryStructure* example.



<<MultiTryCatchStructure>>
<<MultiTryCatchPart>> 1
-possibleSignaledFailures: {FS1}, {FS2}, {FS3}, {FS4}
-possibleErroneousOutputs: {FP1}, {FP2},{FP1, FP2}
<<MultiTryCatchPart>> 2
-handledFailures: {FS2}, {FS3}
-possibleSignaledFailures: {FS2}, {FS3}
-possibleErroneousOutputs: {FP1}, {FP2},{FP1, FP2}
<<MultiTryCatchPart>> 3
-handledFailures: {FS3}, {FS4}
-possibleSignaledFailures: {FS4}
-possibleErroneousOutputs: {FP1}, {FP2},{FP1, FP2}
{FS4}
MultiTryCatchPart 1
{FS2}, {FS3}
MultiTryCatchPart 2
{FS3}
MultiTryCatchPart 3
{FP1}
{FP2}
{FP1, FP2}
{F0}
{FS1}
{FS2}
{FS3}
{FS4}

Figure 8: Semantics for a *MultiTryCatchStructure* 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 failures of stopping failure types and launch alternative activities.

Example 5. *Fig. 8 shows a* MultiTryCatchStructure *with three* MultiTryCatchParts*. After the execution of* MultiTryCatchPart 1*, possible signaled failures of stopping failure types* $\{F_{S1}\}$*,* $\{F_{S2}\}$*,* $\{F_{S3}\}$*, or* $\{F_{S4}\}$*, or possible erroneous outputs of propagating failure types* $\{F_{P1}\}$*,* $\{F_{P2}\}$*, or* $\{F_{P1}, F_{P2}\}$ *can occur. Signaled failures of* $\{F_{S1}\}$ *cannot be handled by any following* MultiTryCatchParts *(*MultiTryCatchPart 2, MultiTryCatchPart 3*) and therefore lead to a signaled failures of the whole structure.* MultiTryCatchPart 2 *handles signaled failures of* $\{F_{S2}\}$ *or* $\{F_{S3}\}$*.* MultiTryCatchPart 3 *handles signaled failures of* $\{F_{S4}\}$*. Erroneous outputs of* MultiTryCatchPart 1 *lead to erroneous outputs of the whole structure.*

Similarly, for MultiTryCatchPart 2*, signaled failures of* $\{F_{S2}\}$ *cannot be handled by any following* MultiTryCatchParts *(*MultiTryCatchPart 3*) and therefore lead to signaled failures of the whole structure. Erro-*

neous outputs of MultiTryCatchPart 2 *lead to erroneous outputs of the whole structure.* MultiTryCatchPart 3 *handles signaled failures of* $\{F_{S3}\}$*.*

For the last MultiTryCatchPart *(*MultiTryCatchPart 3*), because there is no following* MultiTryCatchPart *to handle its signaled failure, all of its signaled failures lead to signaled failures of the whole structure. Erroneous outputs of* MultiTryCatchPart 3 *lead to erroneous outputs of the whole structure.*

Remark. FTSs can be employed in different parts of the system architecture and are quite flexible to model FTMs because their inner parts (*RetryPart*, *MultiTryCatchParts*) are able to contain different activity types, structure types, and even nested FTSs. They support enhanced fault tolerance expressiveness in several aspects, including different recovery behaviors in response to occurrences of signaled failures, as well as multi-type and multi-stage recovery behaviors. They allow modeling different classes of existing FTMs, including exception handling, restart-retry, primary-backup, and recovery blocks. If a *RetryPart* or a *MultiTryCatchPart* contains a *CallingActivity*, signaled failures 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.

4.2. System Reliability Models

In our approach, software architects obtain components and their reliability specifications from public repositories, assemble them to realize the required functionality. After that, they provide a usage profile for the complete system to form a system reliability model.

Fig. 2 shows an extract of our reliability modeling schema with modeling elements for system reliability models. Software architects model a 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 user interfaces (*UserInterface*).

After modeling system architecture, software architects model a usage profile for the user interfaces. 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 average number of loops for each looping structure.

Example 6. *Continuing with Example 1, Fig. 9 shows an example of system reliability model. The system architecture includes instances of components* C_1*,* C_2*,* C_3*, and* C_4*. They are connected via component connectors. Provided service* S_0 *of* C_1*'s component instance is exposed as a user interface for users.*

The usage profile includes two usage profile parts with probabilities 0.7 and 0.3. This means that with probability 0.7, users access with usage profile part 1 and with probability 0.3, users access with usage profile part 2. Each usage profile part contains probabilities and averages to determine the branching probabilities of branching structures and the average number of loops for each looping structure.

5. RELIABILITY PREDICTION

After software architects have assembled component reliability specifications to realize the required functionality and specified a usage profile to form a system reliability model, we can predict the reliability for the complete system. The prediction process starts with the system reliability model and the component reliability specifications, and ends with the system reliability prediction output. It includes the 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 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 type or an activity

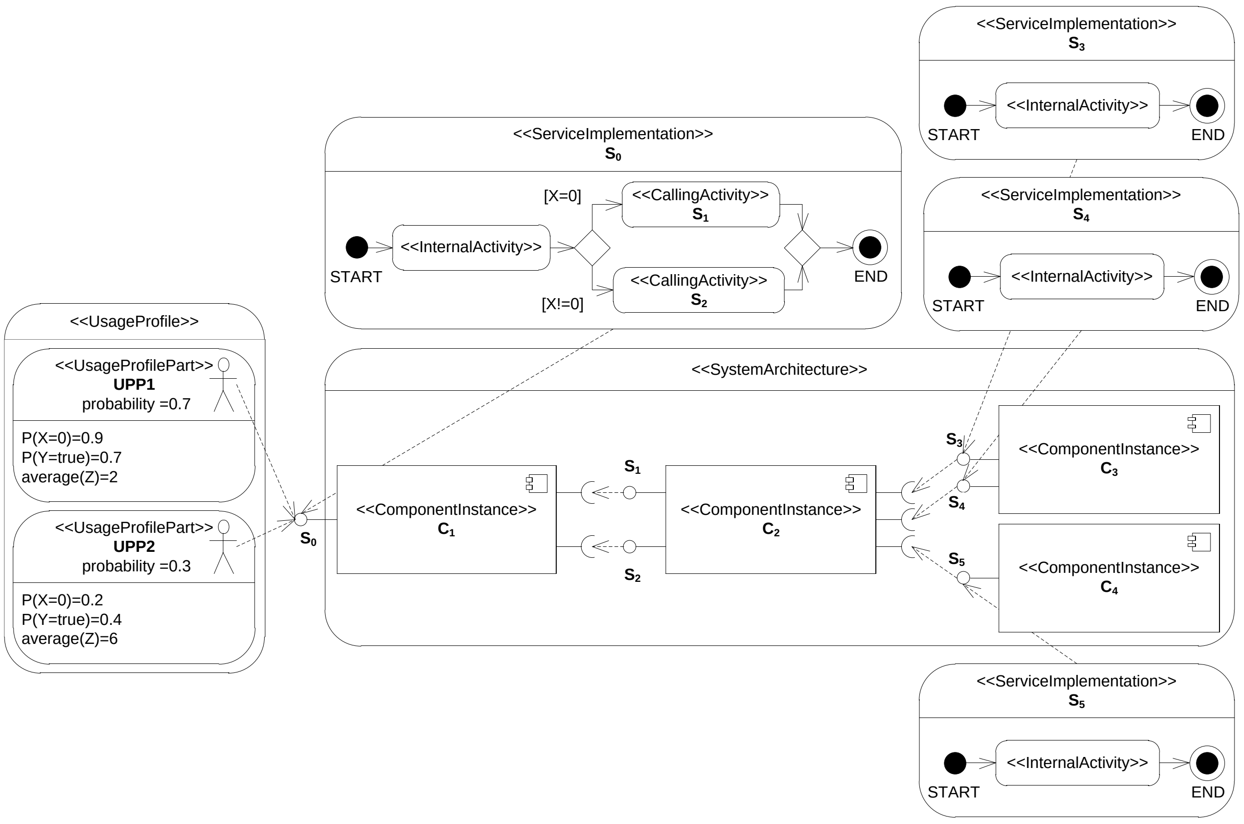
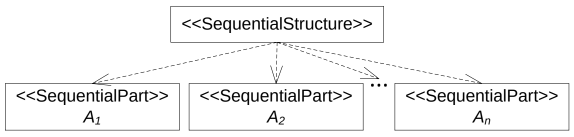
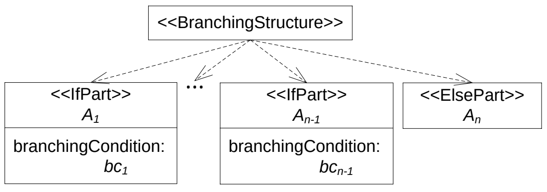


Figure 9: An example of system reliability model.



(a)



(b)

Figure 10: Example of structures: (a) Sequential structure and (b) Branching structure.

of any activity type, (2) a structure's inner part (i.e. *SequentialPart*, *IfPart*, *ElsePart*, *LoopingPart*, *ParallelPart*, *RetryPart*, *MultiTryCatchPart*) can contain a structure of any structure type or an activity of any activity type, and (3) a calling activity is actually a reference to another service implementation. Therefore, the transformation is essentially a recursive procedure applied for structures.

For each structure, the transformation transforms it into an equivalent internal activity (*IA*, for short).

5.1.1. Sequential Structure

Considering a sequential structure with n sequential parts A_1, A_2, ..., A_n as in Fig. 10a, let $Pr_{A_{12...k}}(I, FO)$, $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$ be the failure model for the equivalent IA of the first k sequential parts, then the failure model for the equivalent IA of the first $k+1$ sequential parts is computed as follows.

- The first $k+1$ sequential parts produce a correct output if the first k sequential parts produce an output (correct or erroneous) and after receiving this output as its input, the $(k+1)-th$ sequential part produces a correct output:

$$Pr_{A_{12...k+1}}(I, \{F_0\}) = \sum_{O' \in \mathcal{AIOS}} Pr_{A_{12...k}}(I, O')\, Pr_{A_{k+1}}(O', \{F_0\}) \tag{1}$$

- The first $k+1$ sequential parts signal a signaled failure of stopping failure type F (with $F \in \mathcal{AFS}$) if either (1) the first k sequential parts signal a signaled failure of stopping failure type F or (2) the first k sequential parts produce an output (correct or erroneous) and after receiving this output as its input, the $(k+1)-th$ sequential part signals a signaled failure of stopping failure type F:

$$Pr_{A_{12...k+1}}(I, F) = Pr_{A_{12...k}}(I, F) + \sum_{O' \in \mathcal{AIOS}} Pr_{A_{12...k}}(I, O')\, Pr_{A_{k+1}}(O', F) \tag{2}$$

- The first $k+1$ sequential parts produce an erroneous output of propagating failure types $O \in \mathcal{AIOS} \setminus \{\{F_0\}\}$ if the first k sequential parts produce an output (correct or erroneous) and after receiving this output as its input, the $(k+1)-th$ sequential part produces an erroneous output of propagating failure types O:

$$Pr_{A_{12...k+1}}(I, O) = \sum_{O' \in \mathcal{AIOS}} Pr_{A_{12...k}}(I, O')\, Pr_{A_{k+1}}(O', O) \tag{3}$$

By using Equations (1), (2), and (3), the transformation recursively computes the failure model for the equivalent IA of all n sequential parts (i.e. the failure model for the equivalent IA of the sequential structure): $Pr_{IA}(I, FO) = Pr_{A_{12...n}}(I, FO)$, $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$.

5.1.2. Branching Structure

Considering a branching structure with $n-1$ if parts A_1, A_2, ..., A_{n-1} and a single else part A_n as in Fig. 10b, its equivalent IA has the failure model as follows (with $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$):

$$Pr_{IA}(I, FO) = \sum_{i=1}^{n-1} p(bc_i) Pr_{A_i}(I, FO) + \left(1 - \sum_{i=1}^{n-1} p(bc_i)\right) Pr_{A_n}(I, FO) \tag{4}$$

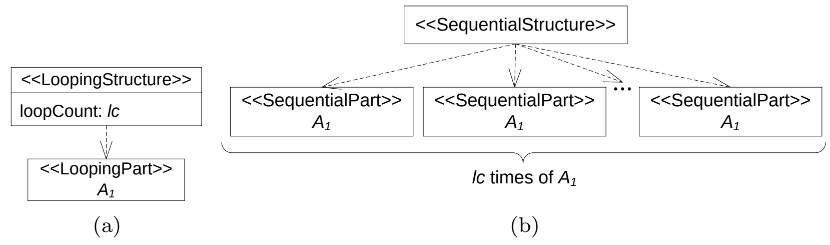


Figure 11: Unrolling a looping structure: (a) Looping structure and (b) Its equivalent sequential structure.

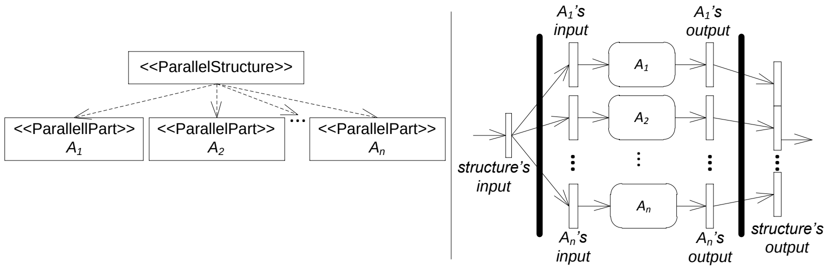


Figure 12: Using inputs and outputs in a parallel structure.

where $p(bc_i)$ (with $i = 1, 2, ..., n-1$) is the probability of the branching condition bc_i (i.e. the execution probability of the if part A_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. 11a, this looping structure can be unrolled to a sequential structure with lc sequential parts A_1 (as in Fig. 11b). Then, with the average of loop count, $average\,(lc)$, obtained from the current usage profile part, the failure model for the equivalent IA of the looping structure can be computed by applying the same transformation, as for a sequential structure, on the equivalent sequential structure of the looping structure. However, because all sequential parts of the equivalent sequential structure are the same A_1, the transformation also employs the exponentiation by squaring[6] for fast transforming.

5.1.4. Parallel Structure

For a parallel structure, the transformation transforms it into an equivalent IA based on the following arguments:

- The parallel structure (therefore the equivalent IA) signals a signaled failure if at least one parallel branch has a signaled failure.
- The parallel structure (therefore the equivalent IA) produces a correct output if all parallel branches produce correct outputs.
- The parallel structure (therefore the equivalent IA) produces an erroneous output if no parallel branch has a signaled failure and at least one parallel branch produces an erroneous output.

and the following assumptions:

[6]http://en.wikipedia.org/wiki/Exponentiating_by_squaring

- Reliability-related behaviors of parallel branches are independent[7].
- In case a parallel structure receives an erroneous input of certain propagating failure types, each of its parallel branch receives an erroneous input of the same propagating failure types. And in case a parallel structure produces an erroneous output, the propagating failure types of the parallel structure's erroneous output is a union of propagating failure types of the parallel branches' erroneous outputs. Fig. 12 shows a usage[8] of inputs and outputs that satisfies the assumption for a parallel structure: each A_k receives the whole input of the parallel structure as its input, and all outputs of $A_k(s)$ are joined to form the structure's output.
- When parallel branches signal signaled failures of different stopping failure types, the stopping failure type of the signaled failure of the whole parallel structure is the stopping failure type of the signaled failure of the lowest index parallel branch[9].

Considering a parallel structure with n parallel branches A_1, A_2, ..., A_n as in Fig. 12, let $Pr_{A_{12\ldots k}}\left(I, FO\right)$, $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$ be the failure model for the equivalent IA of the first k parallel branches, then the failure model for the equivalent IA of the first $k+1$ parallel branches is computed as follows.

- The first $k+1$ parallel branches produce a correct output if the first k parallel branches produce a correct output and the $(k+1)-th$ parallel branch produces a correct output:

$$Pr_{A_{12\ldots k+1}}\left(I, \{F_0\}\right) = Pr_{A_{12\ldots k}}\left(I, \{F_0\}\right) Pr_{A_{k+1}}\left(I, \{F_0\}\right) \tag{5}$$

- The first $k+1$ parallel branches signal a signaled failure of stopping failure type F (with $F \in \mathcal{AFS}$) if either (1) the first k parallel branches signal a signaled failure of stopping failure type F or (2) the first k parallel branches produce an output (correct or erroneous) and the $(k+1)-th$ parallel branch signals a signaled failure of stopping failure type F:

$$Pr_{A_{12\ldots k+1}}\left(I, F\right) = Pr_{A_{12\ldots k}}\left(I, F\right) + \left(\sum_{O' \in \mathcal{AIOS}} Pr_{A_{12\ldots k}}\left(I, O'\right)\right) Pr_{A_{k+1}}\left(I, F\right) \tag{6}$$

- The first $k+1$ parallel branches produce an erroneous output of propagating failure types $O \in \mathcal{AIOS} \setminus \{\{F_0\}\}$ if (1) the first k parallel branches produce a correct output and the $(k+1)-th$ parallel branch produces an erroneous output of propagating failure types O, or (2) the first k parallel branches produce an erroneous output of propagating failure types O and the $(k+1)-th$ parallel branch produces a correct output, or (3) the first k parallel branches produce an erroneous output of propagating failure types $O_1 \in \mathcal{AIOS} \setminus \{\{F_0\}\}$ and the $(k+1)-th$ parallel branch produces an erroneous output of propagating failure types $O_2 \in \mathcal{AIOS} \setminus \{\{F_0\}\}$ such that $O_1 \cup O_2 = O$:

$$\begin{array}{rcl} Pr_{A_{12\ldots k+1}}\left(I, O\right) & = & Pr_{A_{12\ldots k}}\left(I, \{F_0\}\right) Pr_{A_{k+1}}\left(I, O\right) + Pr_{A_{12\ldots k}}\left(I, O\right) Pr_{A_{k+1}}\left(I, \{F_0\}\right) \\ & + & \sum\limits_{\substack{O_1 \cup O_2 = O \\ O_1, O_2 \in \mathcal{AIOS} \setminus \{\{F_0\}\}}} \left(Pr_{A_{12\ldots k}}\left(I, O_1\right) Pr_{A_{k+1}}\left(I, O_2\right)\right) \end{array} \tag{7}$$

[7]Our method does not explicitly consider errors caused by shared resource access or thread interaction, which can be removed by existing techniques before the analysis [37], or implicitly included in probabilities of the failure models for *IA(s)* in parallel branches.

[8]This is one of the most common scenarios in parallel executions. Our method for transforming parallel structures can be extended to include other common scenarios in parallel executions.

[9]We could have supported modeling the concurrent presence of stopping failure types caused by parallel branches signaling signaled failures of different stopping failure types (using the same method as for propagating failure types). However, the fact that in practice, FTMs, if any, to handle errors of parallel executions are often put inside each parallel execution could make the support useless in modeling FTMs. Whereas, supporting modeling the concurrent presences of both stopping failure types and propagating failure types could increase quickly the danger of state-space explosion for our method. Moreover, using the stopping failure types of the signaled failure of the lowest index parallel branch is simply our design choice to avoid introducing the concurrent presence of stopping failure types. Another possible design choice could be using the highest stopping failure type among different stopping failure types of signaled failures of parallel branches given that the stopping failure types are sorted in a certain order (e.g. according to their severities).

Table 2: An Example of Transformation Results.

Execution Results		**Transformation Results**	
$\mathbf{A_1}$	$\mathbf{A_2}$	**Result**	**Occurrence Probability (with $I \in \mathcal{AIOS}$)**
$F \in \mathcal{AFS}$	-	F	$Pr_{A_1}(I, F)$
$O \in \mathcal{AIOS}$	$F \in \mathcal{AFS}$	F	$Pr_{A_1}(I, O)\, Pr_{A_2}(I, F)$
$\{F_0\}$	$O \in (\mathcal{AIOS} \setminus \{\{F_0\}\})$	O	$Pr_{A_1}(I, \{F_0\})\, Pr_{A_2}(I, O)$
$O \in (\mathcal{AIOS} \setminus \{\{F_0\}\})$	$\{F_0\}$	O	$Pr_{A_1}(I, O)\, Pr_{A_2}(I, \{F_0\})$
$\{F_0\}$	$\{F_0\}$	$\{F_0\}$	$Pr_{A_1}(I, \{F_0\})\, Pr_{A_2}(I, \{F_0\})$
$O_1 \in (\mathcal{AIOS} \setminus \{\{F_0\}\})$	$O_2 \in (\mathcal{AIOS} \setminus \{\{F_0\}\})$	$O_1 \cup O_2$	$Pr_{A_1}(I, O_1)\, Pr_{A_2}(I, O_2)$

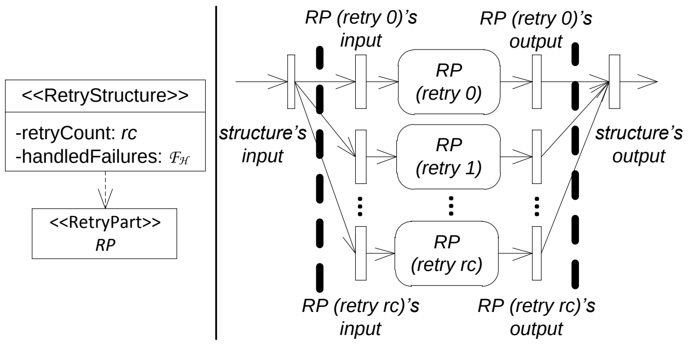


Figure 13: Using inputs and outputs in a *RetryStructure*.

By using Equations (5), (6), and (7), the transformation recursively computes the failure model for the equivalent *IA* of all n parallel branches (i.e. the failure model for the equivalent *IA* of the parallel structure): $Pr_{IA}(I, FO) = Pr_{A_{12\ldots n}}(I, FO)$, $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$.

Example 7. *Assume we have a parallel structure with two parallel branches A_1 and A_2. Each parallel branch has a failure model as in Example 3. Table 2 shows the transformation results. From this table, the transformation can build up the failure model for the equivalent* IA *of the parallel structure. For example, $Pr_{IA}(I, \{F_{S1}\}) = Pr_{A_1}(I, \{F_{S1}\}) + \left(\sum_{O \in \mathcal{AIOS}} Pr_{A_1}(I, O)\right) Pr_{A_2}(I, \{F_{S1}\})$ for all $I \in \mathcal{AIOS}$ (as in Equation (6)).*

5.1.5. RetryStructure

Considering a *RetryStructure*, let rc be the retry count, $\mathcal{F_H} \subseteq \mathcal{AFS}$ be the set of handled failures, $Pr_{RP}(I, FO)$ for all $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$ be the failure model of *RetryPart* (abbreviated as *RP*).

Fig. 13 shows the usage of inputs and outputs in a *RetryStructure*. The structure's input is the input for all *RetryPart*'s executions and the structure's output is the output of a *RetryPart*'s execution. For the sake of uniformity, we consider the first execution of the *RetryPart* as *RP (retry 0)*.

For each possible input $I \in \mathcal{AIOS}$ of a *RetryStructure*, the transformation builds a Markov model that reflects all the possible execution paths of the *RetryStructure* with the input I and their corresponding probabilities, and then build up the failure model for the equivalent *IA* from this Markov model.

Step 1, the transformation builds a Markov block for each retry. The Markov Block for the *i-th* retry ($MB(I, RP_i)$) reflects its possible execution paths for signaled failures (Fig. 14). It includes a state labeled "I, RP_i" ($[I, RP_i]$, for short) as an initial state, states $[RP_i, F]$ for all $F \in \mathcal{AFS}$ as states of signaled failures. The probability of reaching state $[RP_i, F]$ from state $[I, RP_i]$ is $Pr_{RP}(I, F)$ for all $F \in \mathcal{AFS}$.

Step 2, the transformation assembles these Markov blocks into a single Markov model that reflects all the possible execution paths of the *RetryStructure* with the input $I \in \mathcal{AIOS}$ as follows.

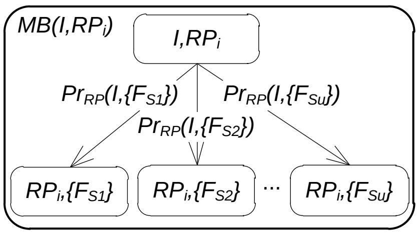


Figure 14: Markov block for *i-th* retry.

- Add a state $[I, START]$.
- Add states $[F]$ for all $F \in \mathcal{AFS}$.
- Add states $[O]$ for all $O \in \mathcal{AIOS}$
- Add a transition from state $[I, START]$ to state $[I, RP_0]$ with probability 1.0.
- For all Markov block $MB\left(I, RP_i\right)$ with $i \in \{0, 1, ..., rc\}$, add transitions from state $[I, RP_i]$ to state $[O]$ with probability $Pr_{RP}\left(I, O\right)$ for all $O \in \mathcal{AIOS}$. This is because a correct (resp. erroneous) output of the *RetryPart*'s execution leads to a correct (resp. erroneous) output of the whole *RetryStructure*.
- For Markov block $MB\left(I, RP_{rc}\right)$ (i.e. the Markov block of the last retry), add transitions from state $[RP_{rc}, F]$ to state $[F]$ with probability 1.0 for all $F \in \mathcal{AFS}$.
- For other Markov blocks, i.e. $MB\left(I, RP_i\right)$ with $i \in \{0, 1, ..., rc - 1\}$, add transitions from state $[RP_i, F]$ to (1) state $[I, RP_{i+1}]$ with probability 1.0 if $F \in \mathcal{F_H}$, or otherwise to (2) state $[F]$ with probability 1.0 for all $F \in \mathcal{AFS}$, .

Step 3, after the transformation generated the Markov model, the failure model for the equivalent *IA* is built up as follows.

- For all $F \in \mathcal{AFS}$, $Pr_{IA}\left(I, F\right)$ is the probability of reaching absorbing state $[F]$ from transient state $[I, START]$.
- For all $O \in \mathcal{AIOS}$, $Pr_{IA}\left(I, O\right)$ is the probability of reaching absorbing state $[O]$ from transient state $[I, START]$.

The transition matrix for the generated Markov chain has the following format:

$$\mathbf{P} = \begin{pmatrix} \mathbf{Q} & \mathbf{R} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}$$

where the upper left transition matrix $\mathbf{Q}$ is a square matrix representing one-step transitions between transient states $[I, START]$, $[I, RP_i]$, and $[RP_i, F]$ for all $F \in \mathcal{AFS}$ (with $i \in \{0, 1, ..., rc\}$), the upper right transition matrix $\mathbf{R}$ represents one-step transitions from the transient states to absorbing states $[F]$ for all $F \in \mathcal{AFS}$ and $[O]$ for all $O \in \mathcal{AIOS}$, $\mathbf{I}$ is an identify matrix with size equal the number of the absorbing states.

Let $\mathbf{B} = (\mathbf{I} - \mathbf{Q})^{\mathbf{-1}}\mathbf{R}$ be the matrix computed from the matrices $\mathbf{I}$, $\mathbf{Q}$ and $\mathbf{R}$. Because this is an absorbing Markov chain, the entry b_{ij} of the matrix $\mathbf{B}$ is the probability that the chain will be absorbed in the absorbing state s_j if it starts in the transient state s_i [38]. Thus, the failure model of the equivalent *IA* can be obtained from the matrix $\mathbf{B}$.

Example 8. *Fig. 15 shows an example of transformation for a* RetryStructure *(Several transition probabilities are omitted for the sake of clarity). In this example, we assume that the* RetryPart *has a failure model*

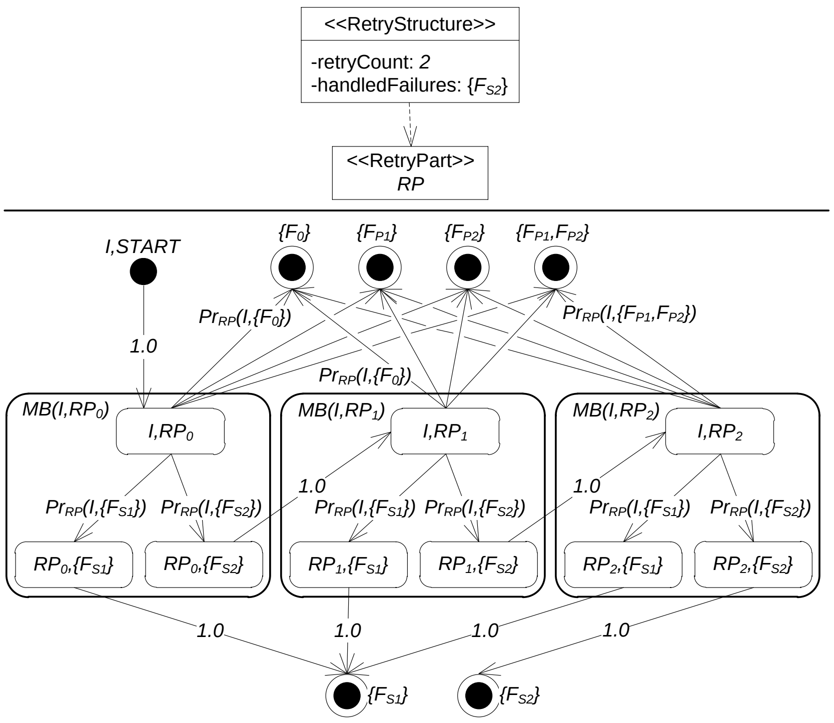


Figure 15: An example of transformation for a *RetryStructure*.

as in Example 3. Therefore, $\mathcal{AFS} = \{\{F_{S1}\}, \{F_{S2}\}\}$ *and* $\mathcal{AIOS} = \{\{F_0\}, \{F_{P1}\}, \{F_{P2}\}, \{F_{P1}, F_{P2}\}\}$. *rc of the* RetryStructure *is 2 and* $\mathcal{F_H}$ *of the* RetryStructure *is* $\{\{F_{S2}\}\}$. *From the resulting Markov model, the failure model for the equivalent* IA *of the* RetryStructure *can be built up, e.g.* $Pr_{IA}(I, \{F_0\})$ *is the probability of reaching absorbing state* $[\{F_0\}]$ *from transient state* $[I, START]$.

5.1.6. *MultiTryCatchStructure*

Considering a *MultiTryCatchStructure*, let n be the number of *MultiTryCatchParts.* For each $i \in \{1, 2, ..., n\}$, let $\mathcal{F}_{\mathcal{H}i} \subseteq \mathcal{AFS}$ be the set of handled failures of *MultiTryCatchPart i*, $Pr_{MP_i}(I, FO)$ for all $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$ be the failure model of *MultiTryCatchPart i* (abbreviated as MP_i).

Fig. 16 shows the usage of inputs and outputs in a *MultiTryCatchStructure.* The structure's input is the input for all *MultiTryCatchParts*' executions and the structure's output is the output of a *MultiTryCatchPart*'s execution.

Similar to the case of *RetryStructures*, for each possible input $I \in \mathcal{AIOS}$ of a *MultiTryCatchStructure*, the transformation builds a Markov model that reflects all the possible execution paths of the *MultiTryCatchStructure* with the input I and their corresponding probabilities, and then build up the failure model for the equivalent *IA* from this Markov model.

Step 1, the transformation builds a Markov block for each *MultiTryCatchPart.* The Markov Block for the *MultiTryCatchPart i* ($MB(I, MP_i)$) reflects its possible execution paths for signaled failures (Fig. 17). It includes a state $[I, MP_i]$ as an initial state, states $[MP_i, F]$ for all $F \in \mathcal{AFS}$ as states of signaled failures. The probability of reaching state $[MP_i, F]$ from state $[I, MP_i]$ is $Pr_{MP_i}(I, F)$ for all $F \in \mathcal{AFS}$.

Step 2, the transformation assembles these Markov blocks into a single Markov model that reflects all the possible execution paths of the *MultiTryCatchStructure* with the input $I \in \mathcal{AIOS}$ as follows.

- Add a state $[I, START]$.
- Add states $[F]$ for all $F \in \mathcal{AFS}$.

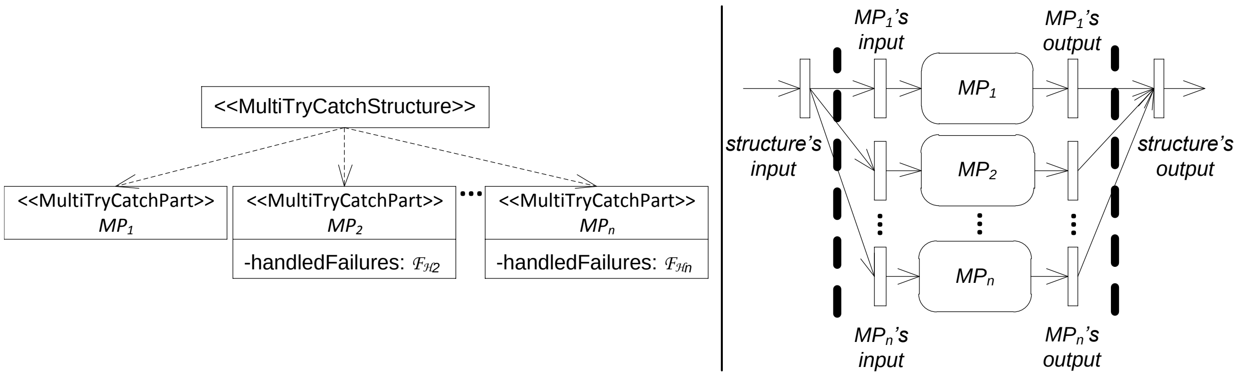

<<MultiTryCatchStructure>>
<<MultiTryCatchPart>> MP1
<<MultiTryCatchPart>> MP2
-handledFailures: $\mathcal{F}_{\mathcal{H}2}$
<<MultiTryCatchPart>> MPn
-handledFailures: $\mathcal{F}_{\mathcal{H}n}$
MP1's input
MP1's output
structure's input
structure's output
MP1
MP2
MPn
MPn's input
MPn's output

Figure 16: Using inputs and outputs in a *MultiTryCatchStructure*.

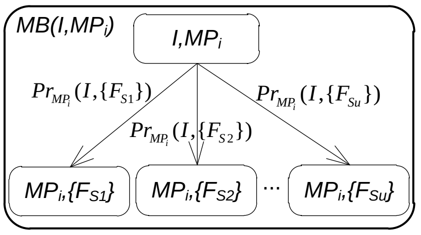

MB(I,MPi)
I,MPi
$Pr_{MP_i}(I,\{F_{S1}\})$
$Pr_{MP_i}(I,\{F_{S2}\})$
$Pr_{MP_i}(I,\{F_{Su}\})$
MPi,{FS1}
MPi,{FS2}
MPi,{FSu}

Figure 17: Markov block for *MultiTryCatchPart i*.

- Add states $[O]$ for all $O \in \mathcal{AIOS}$
- Add a transition from state $[I, START]$ to state $[I, MP_1]$ with probability 1.0.
- For all Markov blocks $MB\left(I, MP_i\right)$ with $i \in \{1, 2, ..., n\}$, add transitions from state $[I, MP_i]$ to state $[O]$ with probability $Pr_{MP_i}\left(I, O\right)$ for all $O \in \mathcal{AIOS}$. This is because a correct (resp. erroneous) output of a *MultiTryCatchPart*'s execution leads to a correct (resp. erroneous) output of the whole *MultiTryCatchStructure*.
- For Markov block $MB\left(I, MP_n\right)$ (i.e. the Markov block of the last *MultiTryCatchPart*), add transitions from state $[MP_n, F]$ to state $[F]$ with probability 1.0 for all $F \in \mathcal{AFS}$.
- For other Markov blocks, i.e. $MB\left(I, MP_i\right)$ with $i \in \{1, 2, ..., n-1\}$, add transitions from state $[MP_i, F]$ to (1) state $[I, MP_x]$ with probability 1.0 where $x \in \{i+1, i+2, ..., n\}$ is the lowest index satisfying $F \in \mathcal{F}_{\mathcal{H}x}$, or to (2) state $[F]$ with probability 1.0 if no such index $x \in \{i+1, i+2, ..., n\}$ satisfying $F \in \mathcal{F}_{\mathcal{H}x}$ for all $F \in \mathcal{AFS}$.

Step 3, because the resulting Markov model is an absorbing Markov chain, the failure model for the equivalent *IA* is built up as follows.

- For all $F \in \mathcal{AFS}$, $Pr_{IA}\left(I, F\right)$ is the probability of reaching absorbing state $[F]$ from transient state $[I, START]$.
- For all $O \in \mathcal{AIOS}$, $Pr_{IA}\left(I, O\right)$ is the probability of reaching absorbing state $[O]$ from transient state $[I, START]$.

Example 9. *Fig. 18 shows an example of transformation for a* MultiTryCatchStructure *(Several transition probabilities are omitted for the sake of clarity). In this example, we assume that each* MultiTryCatchPart *has a failure model as in Example 3. Therefore,* $\mathcal{AFS} = \{\{F_{S1}\}, \{F_{S2}\}\}$ *and* $\mathcal{AIOS} =$

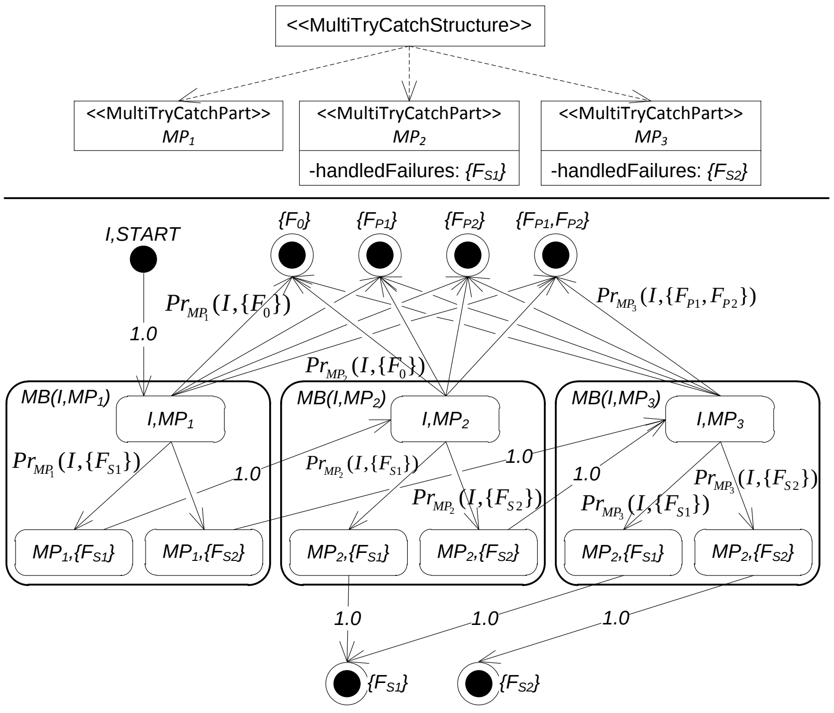


Figure 18: An example of transformation for a *MultiTryCatchStructure*.

$\{\{F_0\},\{F_{P1}\},\{F_{P2}\},\{F_{P1},F_{P2}\}\}$. *$\mathcal{F}_{\mathcal{H}2}$ of the* MultiTryCatchPart 2 *is* $\{\{F_{S1}\}\}$ *and* $\mathcal{F}_{\mathcal{H}3}$ *of the* MultiTryCatchPart 3 *is* $\{\{F_{S2}\}\}$. *From the resulting Markov model, the failure model for the equivalent* IA *of the* MultiTryCatchStructure *can be built up, e.g.* $Pr_{IA}(I,\{F_0\})$ *is the probability of reaching absorbing state* $[\{F_0\}]$ *from transient state* $[I, START]$.

Finally, in our approach, we define the reliability as $R = 1 - POFOD$, where $POFOD$ is the probability of failure on demand, given that the input of a service provided by the system is correct (i.e., as defined by its specification). Therefore, the reliability for the provided service which the current usage profile part refers to is the probability that this service produces a correct output given that it has received a correct input: $Pr_{IA}(\{F_0\},\{F_0\})$ of the failure model for the equivalent IA of the service implementation of this service.

5.2. Aggregation of Results

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

$$R = \sum_{j=1}^{m} P_j \; R(UPP_j) \tag{8}$$

Example 10. *Continuing with Example 6, the overall system reliability is determined as* $R = 0.7R(UPP_1) + 0.3R(UPP_2)$.

Table 3: Running Times of the Transformation Algorithm for Different Structure Types.

Structure type	**Running time**
Sequential structure	$O\left(n_S\lvert\mathcal{AIOS}\rvert^2\left(\lvert\mathcal{AIOS}\rvert+\lvert\mathcal{AFS}\rvert\right)\right)$
Branching structure	$O\left(n_B\lvert\mathcal{AIOS}\rvert\left(\lvert\mathcal{AIOS}\rvert+\lvert\mathcal{AFS}\rvert\right)\right)$
Looping structure	$O\left(log_2\left(lc\right)\lvert\mathcal{AIOS}\rvert^2\left(\lvert\mathcal{AIOS}\rvert+\lvert\mathcal{AFS}\rvert\right)\right)$
Parallel structure	$O\left(n_P\left(\lvert\mathcal{AIOS}\rvert^3+\lvert\mathcal{AFS}\rvert\right)\right)$
RetryStructure	$O\left(\lvert\mathcal{AIOS}\rvert\left(rc^2\lvert\mathcal{AFS}\rvert^2\left(rc\lvert\mathcal{AFS}\rvert+\lvert\mathcal{AIOS}\rvert\right)\right)\right)$
MultiTryCatchStructure	$O\left(\lvert\mathcal{AIOS}\rvert\left(n_M^2\lvert\mathcal{AFS}\rvert^2\left(n_M\lvert\mathcal{AFS}\rvert+\lvert\mathcal{AIOS}\rvert\right)\right)\right)$

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 for a *RetryStructure* or a *MultiTryCatchStructure*, the algorithm requires an additional amount of memory for a Markov chain. 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.

Regarding time-effectiveness, it is assumed that the running time of the transformation algorithm is a function of the structure type, the number of stopping failure types, and the number of propagating failure types. Based on Equations (1), (2), (3), (4), (5), (6), and (7), it is possible to obtain the running times of the algorithm for the sequential, branching, looping, and parallel structure types. The running times of the algorithm for a *RetryStructure* or *MultiTryCatchStructure* can be obtained from the process of creating and solving Markov chains (see Section 5.1.5 or 5.1.6, respectively). Table 3 shows the running times of the algorithm for structure types given that their inner parts have been transformed into equivalent *IA(s)*. Abbreviations used in the table are as follows:

- $\lvert\mathcal{AFS}\rvert$: cardinality of $\mathcal{AFS}$, equal to u which is the number of stopping failure types;
- $\lvert\mathcal{AIOS}\rvert$: cardinality of $\mathcal{AIOS}$, equal to 2^v with v is the number of propagating failure types;
- n_S: number of sequential parts of a sequential structure;
- n_B: number of branching parts (i.e. if and else parts) of a branching structure;

- lc: loop count of a looping structure;
- n_P: number of parallel parts of a parallel structure;
- rc: retry count of *RetryStructure*;
- n_M: number of *MultiTryCatchParts* of a *MultiTryCatchStructure*.

The running time of the algorithm for any structure type is exponential time in the number of propagating failure types and polynomial time in the number of stopping failure types. For a sequential, branching, or parallel structure, the running time of the algorithm is linear time in n_S, n_B, or n_P, respectively. Thanks to exponentiation by squaring, the running time of the algorithm for a looping structure is logarithmic time in lc. The fact that the algorithm for a *RetryStructure* (resp. *MultiTryCatchStructure*) involves calculations on matrices (i.e. matrix subtraction, inversion, and multiplication)[10] leads to a cubic time of the algorithm in rc (resp. n_M). The aggregation of results over m usage profile parts in the usage profile has a running time of $O(m)$.

The complexity of the algorithm presents an issue regarding the scalability of our approach. Therefore, we include scalability considerations into our case study (see Section 6 for more details)).

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.

We also implemented a reliability simulator as a part of our tool. It also receives a system reliability model as an input. It has the abilities to control the execution of each internal activity to follow its failure model, and the execution of each provided service to follow its implementation and the provided usage profile. To simulate the failure model for an internal activity, we implemented a method as follows: (1) The method receives an input, returns an output and may throw exceptions, (2) If the method receives an input marked as $I \in \mathcal{AIOS}$, it throws an exception marked as $F \in \mathcal{AFS}$ with probability $Pr_{IA}(I, F)$ or returns an output marked as $O \in \mathcal{AIOS}$ with probability $Pr_{IA}(I, O)$. We also implemented a method to simulate each provided service of a component. This method also receives an input, returns an output and may throw exceptions. The body of this method includes statements directing the data and control flow according to the provided service's implementation and the provided usage profile. Finally, the simulator determines the system reliability as the ratio of successful service executions (starting with inputs marked as correct $\{F_0\}$ and ending with outputs marked as correct $\{F_0\}$) to the overall execution count.

Compared to our analytical method, the simulation is significantly slower and cannot be used as our main prediction method. However, we can use it for validation purposes. By comparing prediction results obtained by our analytical method with simulations of the systems, it is possible for us to provide evidence for the correctness of the transformation algorithm and the validity of prediction results (see Section 6 for more details).

Our reliability modeling schema and reliability prediction tool are open source and available at our project website [36].

6. CASE STUDY EVALUATION

The goal of the case study evaluation described in this section is (1) to assess the validity of our prediction results, and (2) to demonstrate the capabilities of our reliability modeling and prediction approach in supporting design decisions.

[10]We assumed that the running time for subtracting two $n \times n$ matrices is $O(n^2)$, the running time for inverting one $n \times n$ matrix is $O(n^3)$, and the running time for multiplying rectangular matrices (one $m \times p$ matrix with one $p \times n$ matrix) is $O(mnp)$.

Table 4: Reporting Service: Different Propagating Failure Types and Their Symbols.

Propagating Failure Type	**Symbol**
ContentPropagatingFailure	F_{P1}
TimingPropagatingFailure	F_{P2}

There are several aspects to validate a reliability prediction result. First, varying the input parameters should result in a reasonable change of the prediction result. Second, the accuracy of the prediction results should be validated, in an ideal manner, against measured values. However, validating prediction results against measured values is such a strong challenge that, in practice, validations of prediction results are much weaker and mostly are only done at a reasonable level (i.e. with sensitivity analyses, reliability simulations) (e.g. [3, 12, 13, 14, 22, 28]). The main reason lies in the difficulty of estimating reliability-related probabilities (e.g. failure probabilities, error propagation probabilities) for a software system. It is well known that setting tests to achieve a statistically significant amount of measurement on which the estimation can be based is non-trivial for high-reliability software systems [39] because the necessary number of tests and the necessary time for this are prohibitive. Therefore, in this paper, we validate our prediction results at a reasonable level, i.e. by comparing the prediction results against the results of reliability simulations and by conducting sensitivity analyses to variations in the input parameters.

In the following, we describe the predictions and sensitivity analyses for the reporting service of a document exchange server (Section 6.1) and the WebScan system (Section 6.3), and present the scalability of our approach (Section 6.2). Both case studies include comparisons between prediction results with simulations while different sensitivity analyses are presented by the case studies. The case study based on the WebScan system presents how to introduce a FTS as an additional component while in the case study based on the reporting service, FTSs are within the service implementations of the existing components.

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 first case study is the reporting service of a document exchange server [30]. The document exchange server is an industrial system which was designed in a service-oriented way. Its reporting service allows generating reports about pending documents or released documents.

The system reliability model of the reporting service[11] is shown in Fig. 19 using our reliability modeling schema. At the architecture level, the reporting service consists of four components: *ReportingMediator*, *ReportingEngine*, *SourceManager*, and *DestinationManager*. The component *SourceManager* provides two services to get information about pending documents: *getAttachmentDocumentInfo* to get information about pending documents attached in emails and *getFileDocumentInfo* to get information about pending documents stored in file systems. The component *DestinationManager* provides two services to get information about released documents: *getReleasedDocumentInfoFromLogs* to get the information from the logs, *getReleasedDocumentInfoFromDB* to get the information from the database (DB). The component *ReportingEngine* provides two services: *generateReport* to generate a new report (either about pending documents (*aboutPendingDocuments*=true) or about released documents (*aboutPendingDocuments*=false)) and *viewRecentReports* to view recently generated reports (with the number of reports specified by *numberOfRecentReports*). The component *ReportingMediator* provides the service *processReportRequest* for handling incoming report request from clients. An incoming report request can be about generating a new report (*requestType*=generate) or viewing recently generated reports (*requestType*=view).

In this case study, we are interested in validity of the predictions and sensitivity analyses. We set the usage profile for the reporting service as shown in Fig. 19. The usage profile contains two usage profile parts that present different usage scenarios of the service. Staffs use the service mainly for generating reports while managers use the service mainly for viewing recently generated reports.

[11]The model can be retrieved from our project website [36].

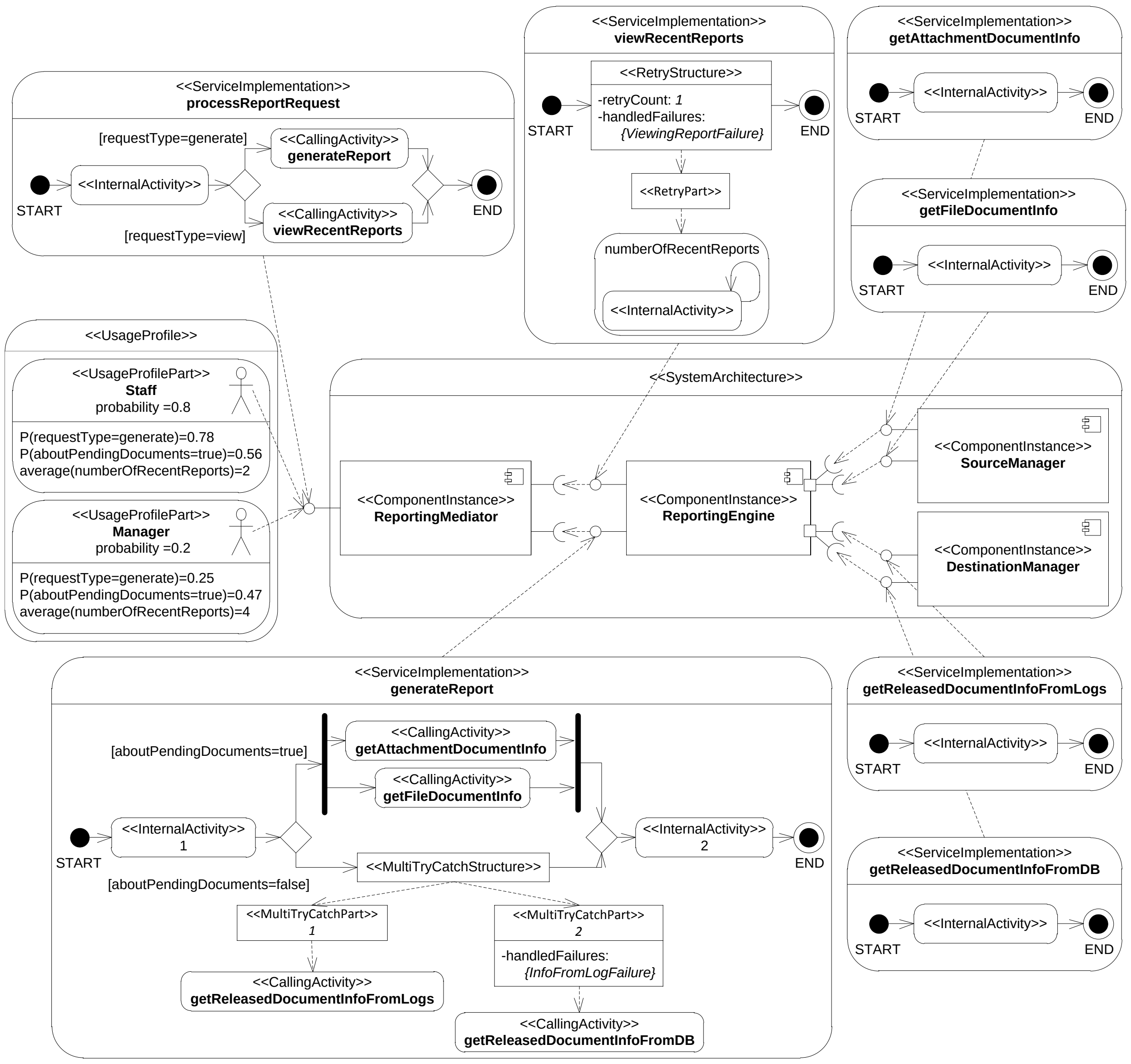


Figure 19: The system reliability model of the reporting service (overview).

Table 5: Reporting Service: Different Stopping Failure Types and Their Symbols.

Stopping Failure Type	**Symbol**
$ProcessingRequestFailure$	F_{S1}
$ViewingReportFailure$	F_{S2}
$GeneratingReportFailure$	F_{S3}
$AttachmentInfoFailure$	F_{S4}
$FileInfoFailure$	F_{S5}
$InfoFromLogFailure$	F_{S6}
$InfoFromDBFailure$	F_{S7}

Table 6: Reporting Service: Internal Activities, Their Symbols, and Involved Failure Types.

Symbol	Provided service/Internal activity (ia)	Involved Failure Types
a_1	*processReportRequest/ia*	F_{S1}, F_{P1}, F_{P2}
a_2	*viewRecentReports/ia*	F_{S2}, F_{P1}, F_{P2}
a_3	*generateReport/ia 1*	F_{S3}, F_{P1}, F_{P2}
a_8	*generateReport/ia 2*	F_{S3}, F_{P1}, F_{P2}
a_4	*getAttachmentDocumentInfo/ia*	F_{S4}, F_{P1}, F_{P2}
a_5	*getFileDocumentInfo/ia*	F_{S5}, F_{P1}, F_{P2}
a_6	*getReleasedDocumentInfoFromLogs/ia*	F_{S6}, F_{P1}, F_{P2}
a_7	*getReleasedDocumentInfoFromDB/ia*	F_{S7}, F_{P1}, F_{P2}

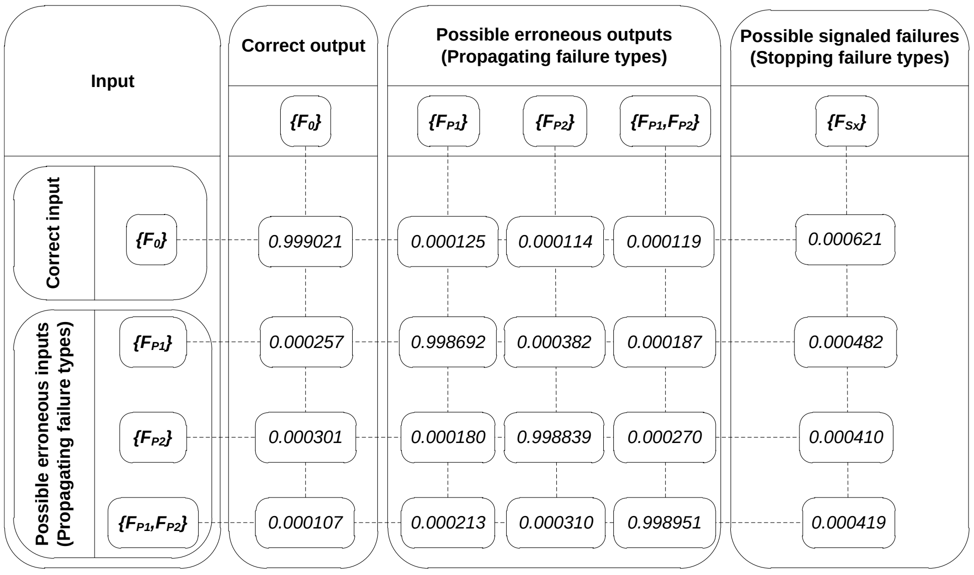

Input
Correct output
Possible erroneous outputs (Propagating failure types)
Possible signaled failures (Stopping failure types)
{F0}
{FP1}
{FP2}
{FP1,FP2}
{FSx}
Correct input
Possible erroneous inputs (Propagating failure types)
{F0} 0.999021 0.000125 0.000114 0.000119 0.000621
{FP1} 0.000257 0.998692 0.000382 0.000187 0.000482
{FP2} 0.000301 0.000180 0.998839 0.000270 0.000410
{FP1,FP2} 0.000107 0.000213 0.000310 0.998951 0.000419

Figure 20: Reporting service: Failure model for internal activity a_i.

There are different errors which may occur in the component instances during the operation of the reporting service. For example, during processing client requests in service *processReportRequest*, errors can arise because of its internal activity's faults. When these errors are detected and signaled with a warning signaled by the error detection of the internal activity, then a signaled failure of a stopping failure type occurs: $\{ProcessingRequestFailure\}$. Otherwise, the internal activity produces an erroneous output of different propagating failure types: $\{ContentPropagatingFailure\}$, $\{TimingPropagatingFailure\}$, or $\{ContentPropagatingFailure, TimingPropagatingFailure\}$. Different propagating (resp. stopping) failure types and their symbols are given in Table 4 (resp. Table 5). Table 6 shows internal activities, their symbols, and involved failure types.

Determining the probabilities of the failure models for the internal activities is beyond the scope of this paper. However, in order to make our model as realistic as possible, we aligned the probabilities with the remarks by Cortellessa et al. [9]: (1) With modern testing techniques, it is practically always to produce a software component with a failure probability lower than 0.001, and (2) It is very likely to find and build software components with values of error propagation probabilities very close to 1. For the sake of simplicity, we assumed the probabilities of the failure model for the internal activity a_i (with $i \in \{1, 2, ..., 8\}$) as in Fig. 20 where F_{Sx} is the involved stopping failure type for a_i.

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

Table 7: Reporting Service: Predicted vs. Simulated Reliability

Predicted reliability	**Simulated reliability**	**Difference**	**Error(%)**
0.996527	0.996652	0.000125	0.012542

$\{ViewingReportFailure\}$. The number of times to retry of this structure is 1 (*retryCount*=1). The second is the *MultiTryCatchStructure* in the implementation of service *generateReport*. This structure has the ability to handle a signaled failure of $\{InfoFromLogFailure\}$ of the service *getReleasedDocumentInfoFromLogs* by redirecting calls to the service *getReleasedDocumentInfoFromDB*.

6.1.2. Validity of Predictions

To validate the accuracy of our prediction results, we used the system reliability model of the reporting service as an input for our reliability prediction tool to get the reliability prediction result, then compared this prediction result to the result of a reliability simulation. Notice that the goal of our validation is not to justify the probabilities of the failure models for internal activities. Instead, we validate that our method produces an accurate system reliability prediction if the system reliability model is provided accurately.

With the system reliability model of the reporting service as an input, our reliability prediction tool predicted the system reliability as 0.996527 after 1 second on an Intel Core 2 Duo 2.26 GHz and 4 GB of RAM while the simulation took more than 30 minutes to run with overall execution count 1,000,000 and produced the simulated system reliability 0.996652.

Table 7 shows the comparison between the predicted reliability and the reliability from the simulation. From this comparison, we deem that for the system reliability model described in this paper, our analytical method is sufficiently accurate.

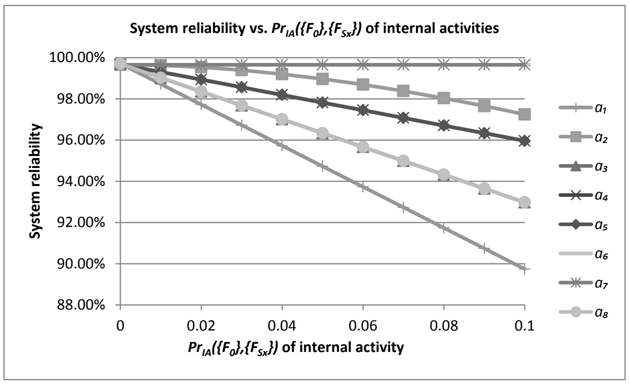
6.1.3. Sensitivity analyses and the Impacts of FTSs

To demonstrate the capabilities of our approach in supporting design decisions, we present the results of sensitivity analyses of the reliability of the reporting service to changes in probabilities of failure models of internal activities, and the analysis of how the predicted reliability of the reporting service varies for fault tolerance variants.

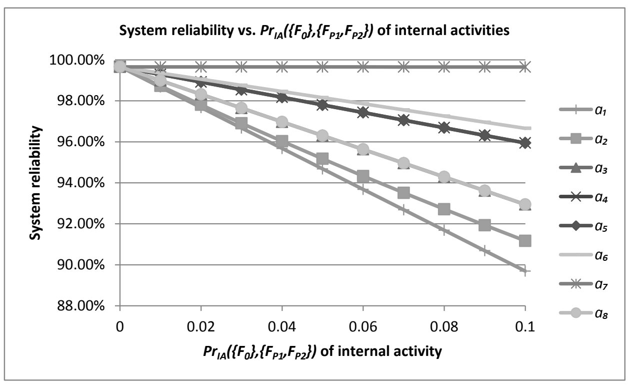
First, we conducted a sensitivity analysis modifying probabilities $Pr_{IA}(\{F_0\}, \{F_{Sx}\})$ of the internal activities (Fig. 21a). The reliability of the reporting service is most sensitive to the probability of internal activity a_1 of service *processReportRequest* provided by the component instance of *ReportingMediator* because its corresponding curve has the steepest slope. The reliability of the reporting service is most robust to the probabilities of 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.

Second, we conducted a sensitivity analysis modifying probabilities $Pr_{IA}(\{F_0\}, \{F_{P1}, F_{P2}\})$ of the internal activities (Fig. 21b). Again, the reliability of the reporting service is most sensitive to the probability of internal activity a_1 because its corresponding curve has the steepest slope. However, the reliability of the reporting service is not as robust to the probabilities of internal activities a_2, a_6, a_7 of the services related to the two FTSs as in the first sensitivity analysis because the FTSs cannot provide error handling for erroneous outputs of propagating failure types $\{F_{P1}, F_{P2}\}$. Between these three internal activities a_2, a_6, a_7, the reliability of the reporting service is most sensitive to the probability of internal activity a_2. This information may be valuable to the software architect when considering putting more development effort to improve the error detection (therefore limit the ability to produce erroneous outputs) of internal activities within the FTSs in the system.

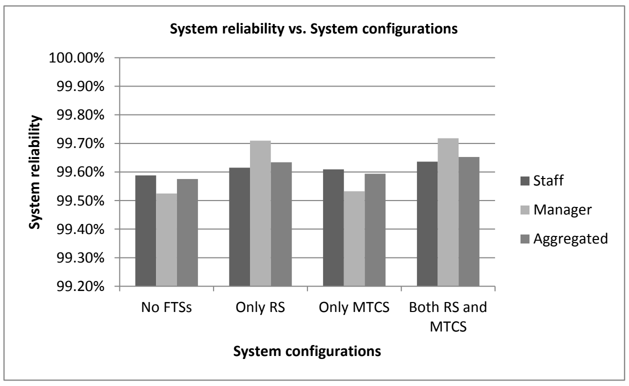
Third, we conducted an analysis of how the predicted reliability of the reporting service varies for fault tolerance variants. These variants include: without the FTSs (*No FTSs*), using only the *RetryStructure* (*Only RS*), using only the *MultiTryCatchStructure* (*Only MTCS*) and using both the FTSs (*Both RS and*



(a)



(b)



(c)

Figure 21: Reporting service: Sensitivity analyses.

MTCS) (Fig. 21c). In general, introducing FTSs brings increases in reliability for staffs, managers, or all (when aggregated). Variant *Both RS and MTCS* is predicted as being the most reliable. Comparing between variants *Only RS* and *Only MTCS* shows that using the *RetryStructure* brings higher reliability impact than using the *MultiTryCatchStructure* 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 *handledFailures* of the structure to $\emptyset$ or 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 *handledFailures* of the second *MultiTryCatchPart* to $\emptyset$ to disable the structure). For variant *No FTSs*, the two above modifications are included.

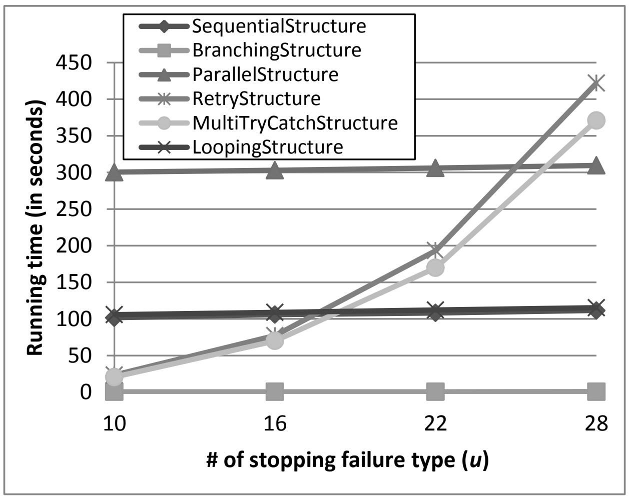
6.2. Scalability analyses

The scalability of our approach requires special attention. To examine it, for each structure type, we generated a number of simple system reliability models with different numbers of u, v, and n_S, n_B, lc, n_P, rc, or n_M (cf. Section 5.4)), analyzed these system reliability models using our reliability prediction tool, and recorded the running times of the transformation algorithm. For example, for the sequential structure type, with given numbers of u, v, and n_S, the generated system reliability model includes the following elements:

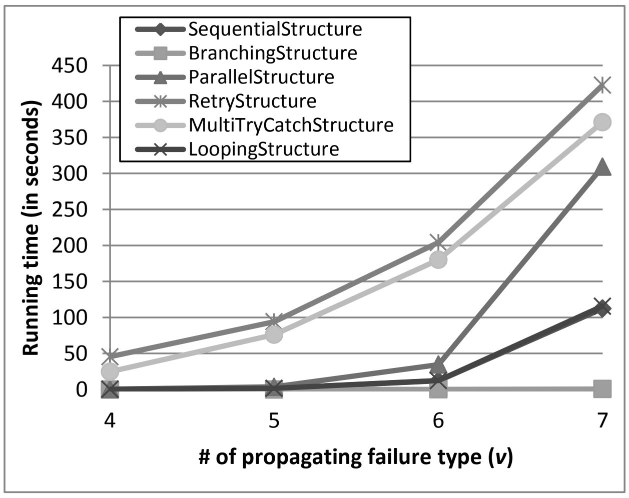
- u stopping failure types and v propagating failure types.
- One service.
- One component with a provided service referring the service. The service implementation for the provided service is a sequential structure of n_S sequential parts. Each of the sequential parts contains an internal activity with a failure model of random probabilities.
- One system architecture with a component instance of the component.
- One user interface referring to the provided service of the component instance.
- One usage profile with a usage profile part referring to the user interface.

Fig. 22a shows the running times of the algorithm for structure types with different numbers of u while $v = 7$, $n_S = n_B = n_P = rc = n_M = 25$, and $lc = 2^{25}$. With the same numbers of n_S, n_B, n_P, rc, n_M, and lc as above, in Fig. 22b are the running times of the algorithm for structure types with different numbers of v while $u = 28$. The running times of the algorithm for structure types with different numbers of n_S, n_B,n_P, rc, n_M, and lc while $u = 28$ and $v = 7$ are shown in Fig. 22c. These results not only provide evidence for the correctness of our complexity analysis for the transformation algorithm as in Section 5.4 but also indicate that the algorithm can analyze system reliability models with up to approximately 28 stopping failure types and 7 propagating failure types within 20 minutes.

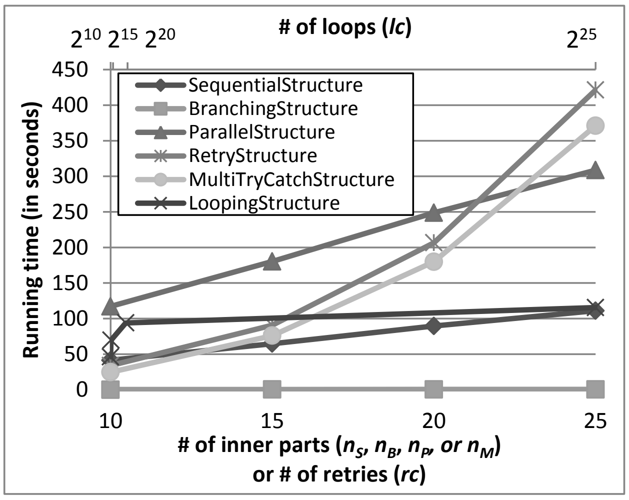
Therefore, we deem that the capacity of our approach is sufficient for typical small-sized and mid-sized software systems, including information systems (e.g. business reporting systems), e-commerce applications (e.g. online shops), device control systems (e.g. the WebScan system introduced in Section 6.3), as well as other types of software systems. A more effective strategy for large-scale software systems with more propagating failure types remains as a goal for future research. In the meantime, for large-scale software systems with large numbers of failure types, multiple failure types can be grouped together and aggregated into one failure type before the analysis.



(a)



(b)



(c)

Figure 22: Scalability analyses.

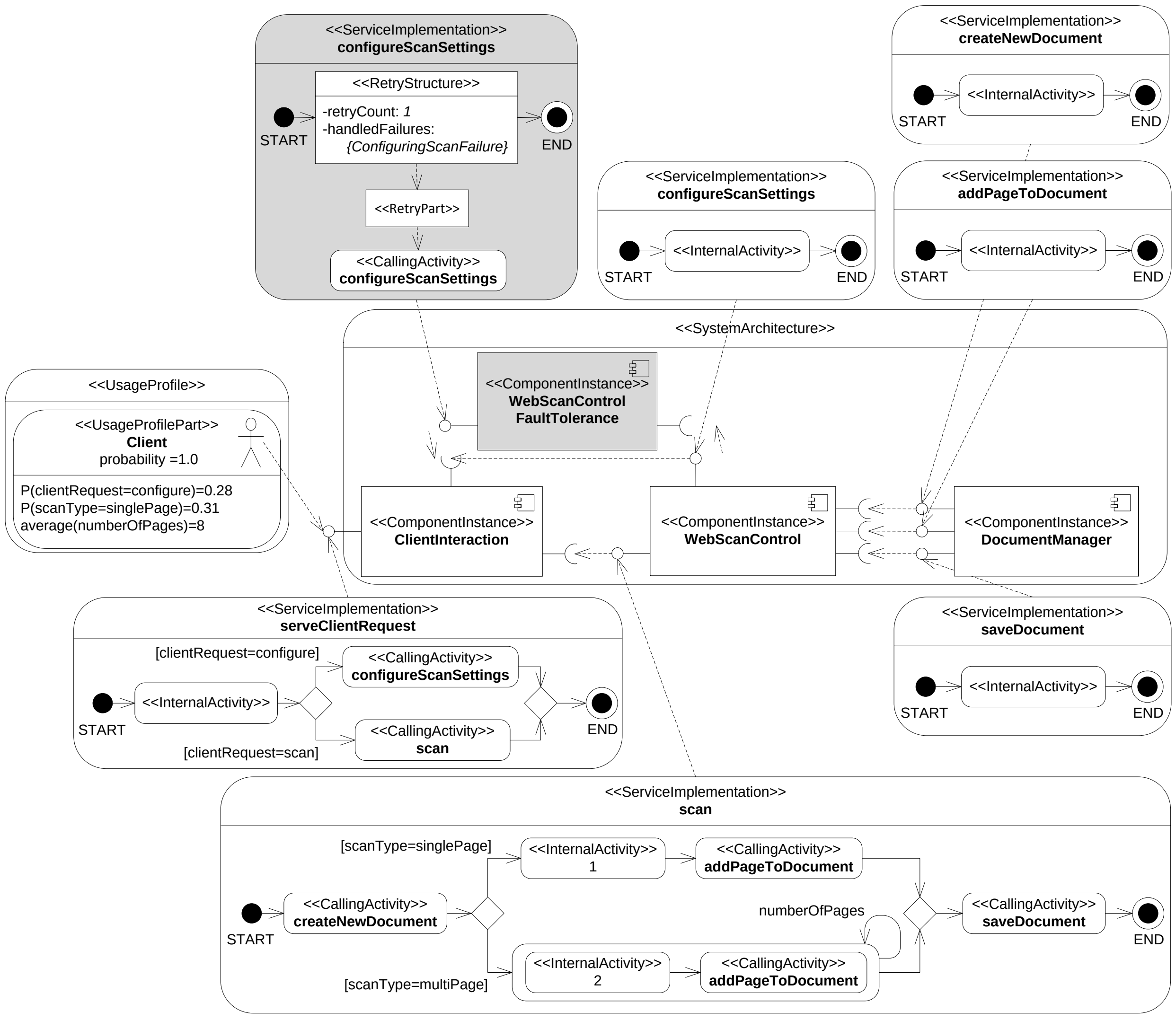


Figure 23: The system reliability model of the WebScan system (overview).

6.3. Case study II: WebScan system

As the second case study, we analyzed the reliability of a WebScan system. The system allows users at desktop to scan one or more images into a document management system using a browser, such as Internet Explorer, and a locally attached TWAIN scanner. Fig. 23 shows the system reliability model for this system[12].

The WebScan system can be accessed via provided service *serveClientRequest* of the instance of component *ClientInteraction*. An incoming request can be a request to configure the settings of the scanner (*clientRequest*=configure) or a request to scan (*clientRequest*=scan). With a request to scan, it can be a request to scan a single page (*scanType*=singlePage) or a request to scan multiple pages (*scanType*=multiPage). With a request to scan multiple pages, *numberOfPages* is to specify the number of pages.

About the system architecture, the system includes three core components, namely *ClientInteraction*, *WebScanControl*, and *DocumentManager*. Component *DocumentManager* provides services: *createNewDocument* to create a new document , *addPageToDocument* to add a page to a document, and *saveDocument*

[12]The model can also be retrieved from our project website [36].

Table 8: WebScan System: Propagating Failure Type and Its Symbol.

Propagating Failure Type	**Symbol**
ContentPropagatingFailure	F_{P1}

Table 9: WebScan System: Different Stopping Failure Types and Their Symbols.

Stopping Failure Type	**Symbol**
ServingRequestFailure	F_{S1}
ConfiguringScanFailure	F_{S2}
ScanningFailure	F_{S3}
CreatingDocumentFailure	F_{S4}
AddingPageFailure	F_{S5}
SavingDocumentFailure	F_{S6}

to save a document. All these three provided services are modeled through single internal activities. Component *WebScanControl* provides services: *configureScanSettings* to configure the settings of the scanner, and *scan* to scan.

During the operation of the WebScan system, there are different errors which may occur in the involved component instances. For example, bugs in the code implementing the internal activity of service *addPageToDocument* may lead to errors. If the error detection of the internal activity detects and signals these errors with a warning message, this leads to a signaled failure of stopping failure type: $\{AddingPageFailure\}$. Otherwise, an erroneous output of a propagating failure type is produced by the internal activity: $\{ContentPropagatingFailure\}$. Table 8 shows a propagating failure type and its symbol and Table 9 shows different stopping failure types and their symbols. Internal activities, their symbols, and involved failure types are given in Table 10.

The usage profile consists of a single usage profile part with 28% of requests to configure the settings of the scanner, probability of 31% for scanning a single page per request to scan, an average of 8 pages per request to scan multiple pages.

For illustrative purpose, we set the probabilities of the failure model for the internal activity a_i (with $i \in \{1, 2, ..., 7\}$) as in Fig. 24 where F_{Sx} is the involved stopping failure type for a_i. Table 11 shows the specific values for the probabilities in the failure models of the internal activities.

A FTS can be optionally introduced into the WebScan System, in terms of an additional component which is shown in grey in Fig. 23. Component *WebScanControlFaultTolerance* can be put in the middle of component *WebScanControl* and component *ClientInteraction*. It has the ability to retry in case there is a signaled failure of $\{ConfiguringScanFailure\}$. The number of times to retry of this structure is 1 (*retryCount*=1).

For a comparison between predicted system reliability and simulated system reliability, we ran a simulation with execution count 1,000,000. The simulation produced the simulated system reliability 0.998041 while our reliability modeling tool predicted the system reliability as 0.998187. Table 12 compares the

Table 10: WebScan System: Internal Activities, Their Symbols, and Involved Failure Types.

Symbol	**Provided service/Internal activity (ia)**	**Involved Failure Types**
a_1	*serveClientRequest/ia*	F_{S1}, F_{P1}
a_2	*configureScanSettings/ia*	F_{S2}, F_{P1}
a_3	*scan/ia 1*	F_{S3}, F_{P1}
a_4	*scan/ia 2*	F_{S3}, F_{P1}
a_5	*createNewDocument/ia*	F_{S4}, F_{P1}
a_6	*addPageToDocument/ia*	F_{S5}, F_{P1}
a_7	*saveDocoument/ia*	F_{S6}, F_{P1}

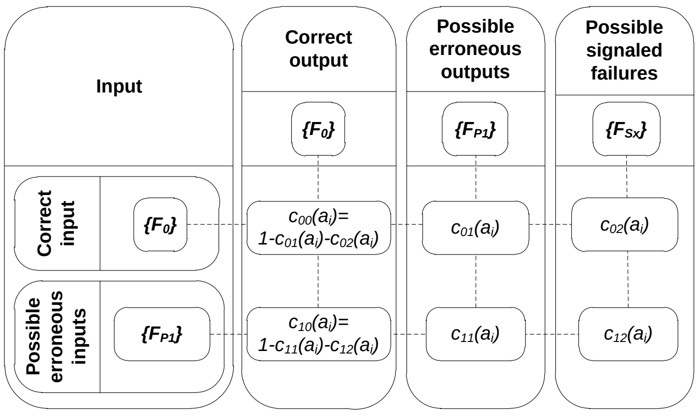

Input
Correct output
Possible erroneous outputs
Possible signaled failures
{F0}
{FP1}
{FSX}
Correct input
{F0}
c00(ai)= 1-c01(ai)-c02(ai)
c01(ai)
c02(ai)
Possible erroneous inputs
{FP1}
c10(ai)= 1-c11(ai)-c12(ai)
c11(ai)
c12(ai)

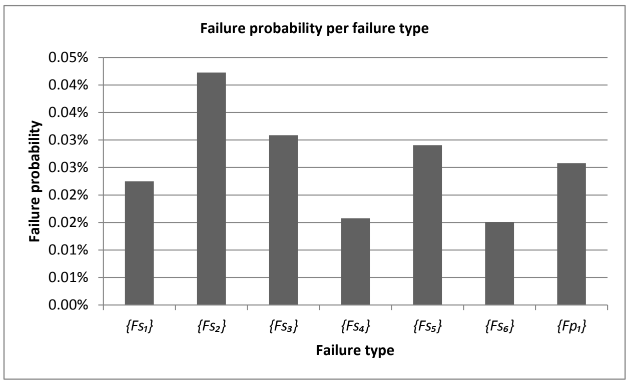
Figure 24: WebScan system: Failure model for internal activity a_i.

Table 11: WebScan System: Internal Activities and the Probabilities in Their Failure Models.

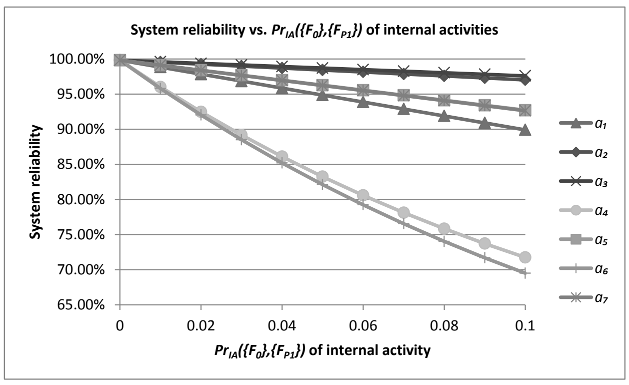
Internal activity	$c_{01}\ (a_i)$	$c_{02}\ (a_i)$	$c_{11}\ (a_i)$	$c_{12}\ (a_i)$
a_1	0.0000205	0.000225	0.99908	0.000119
a_2	0.000107	0.00151	0.99819	0.00171
a_3	0.0000183	0.0000713	0.9991	0.000125
a_4	0.0000209	0.0000737	0.9991	0.000114
a_5	0.000027	0.000219	0.99901	0.000221
a_6	0.0000199	0.0000693	0.99925	0.000101
a_7	0.0000265	0.00021	0.99914	0.000108

Table 12: WebScan System: Predicted vs. Simulated Reliability

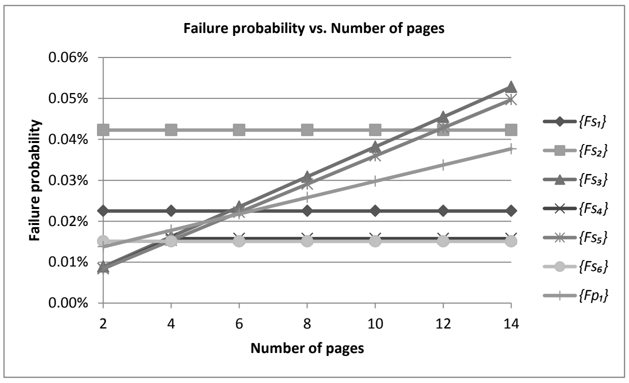
Predicted reliability	**Simulated reliability**	**Difference**	**Error(%)**
0.998187	0.998041	0.000146	0.014629



(a)



(b)



(c)

Figure 25: WebScan system: Sensitivity analyses.

predicted system reliability and the simulated system reliability. This comparison gives evidence that our approach accurately predicts the system reliability in this case.

Fig. 25a provides more detail and shows the probability of a system failure due to a certain failure type. $\{F_{S2}\}$, $\{F_{S3}\}$), and $\{F_{S5}\}$ are the most frequent failure types. Thus, the software architect can recognize the need to introduce FTSs for these failures. For example, the software architect puts an instance of component *WebScanControlFaultTolerance* in the middle of the instance of component *ClientInteraction* and the instance of component *WebScanControl* as in Fig. 23. With this modification, the predicted reliability of the WebScan system increases by 0.042277%, from 0.998187 to 0.998609. 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 changes necessary while nearly all modeling parts can be reused.

Fig. 25b shows the impact of different $Pr_{IA}\left(\{F_0\},\{F_{P1}\}\right)$ of the internal activities to the reliability of the WebScan system. The slopes of the curves indicate that the reliability of the WebScan system is most sensitive to the probabilities of internal activities: a_4 of service *scan* provided by the instance of component *WebScanControl* and a_6 of service *addPageToDocument* provided by the instance of component *DocumentManager*. Thus, it is most beneficial to focus on the improvements for these two services.

Fig. 25c shows the sensitivity of the failure probability per failure type to the number of pages (i.e., a change to the usage profile). As expected, only the failure probabilities for $\{F_{S3}\}$, $\{F_{S5}\}$, and $\{F_{P1}\}$ rise because they are the only failure types related to activities within the looping structure with loop count *numberOfPages*.

7. ASSUMPTIONS AND LIMITATIONS

In this section, we discuss assumptions and limitations of our approach. We focus on three central issues: (1) The provision of proper inputs for our method, (2) the Markovian assumption of our approach, and (3) the limitations in the expressiveness of the model.

7.1. Provision of inputs

Perhaps, the most critical assumption lies in the provision of inputs for our method. The predicted reliability can only be close to reality when inputs are provided accurately for the method.

Call propagations in a component reliability specification can be provided by component developers or determined through reverse engineering. Monitoring inputs and outputs of the component by running it as a black box in a test-bed can be used to determine call propagations in case the source code of the component is not available. Besides call propagations, probabilities of the failure models for internal activities also need to be given as an input for the method. Because failures and error propagations are rare events, and the exact circumstances of their occurrences are unknown, it is difficult to measure these probabilities. However, there are techniques [1, 4, 6, 33, 34, 39] that particularly target the problem of estimating these probabilities, such as fault injection, statistic testing, or software reliability growth models. Further sources of information can be expert knowledge or historical data from components with similar functionalities. In case these probabilities are only estimated roughly, our approach can be used in comparing alternatives of system architectures or determining acceptable ranges for probabilities.

Our approach assumes that software architects can provide a usage profile reflecting different usage scenarios of the system. Similar to the problem of estimating probabilities of the failure models for internal activities, no methodology is always valid to deal with the problem. In early phases of software development, the estimation can be based on historical data from similar products or on high level information about software architecture and usage obtained from specification and design documents [40]. 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 [1].

7.2. Markovian assumption

Our approach assumes that control transitions between components have the Markov property. This means that operational and reliability-related behaviors of a component are independent of its past execution history. This Markovian assumption limits the applicability of our approach on different application domains. However, the Markovian assumption has been proved to be valid at the component level for many software systems [5]. Moreover, the problem of the Markovian assumption in reliability modeling and prediction was treated deeply by Goseva et al. [1], where the authors took the execution histories of components into account by using higher order Markov chains and recalled that a higher order Markov chain can be mapped into a first order Markov chain. Therefore, our approach can also be adapted to any higher order Markov chains, broadening the applicability of our approach to a large number of software systems.

7.3. Expressiveness of the model

With regard to the expressiveness of our approach, we face a general trade-off between the model complexity and its suitability for real-world software systems. A more complex model not only increases the possibility of state-space explosion of the underlying analytical model but also requires more modeling efforts as well as more fine-grained inputs. Therefore, in analogy to related approaches (see [1, 6, 20]), we have restricted our approach to the most important concepts from our point of view (see Section 4). In particular, we do not distinguish between control flow and data flow, and assume that data errors always propagate through control flow. Moreover, we assume that probabilities of the failure models for internal activities are stochastically independent. Currently, input parameters 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 an extension for future work.

8. CONCLUSIONS

In this paper, we present a reliability modeling and prediction approach for component-based software systems that explicitly considers error propagation, software fault tolerance mechanisms, and concurrently present errors. Our approach supports modeling error propagation for multiple execution models, including sequential, parallel, and fault tolerance execution models. Via an explicit and flexible definition of reliability-relevant behavioral aspects (i.e. error detection and error handling) of FTMs, our approach offers an effective evaluation of their reliability impact in the dependence of the whole system architecture and usage profile. Our approach accounts for concurrently present errors by introducing a hierarchical tree of multiple failure types.

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 predictions and sensitivity analyses by our reliability prediction tool. Via case studies, we demonstrated the applicability of our approach, e.g. the ability of supporting design decisions and reusing 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 extend our approach with more complex error propagation for concurrent executions, to include more software FTSs, and to validate further our approach. We also plan to extend our reliability modeling schema and prediction tool for automated sensitivity analyses and design optimization. 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. We are grateful to François Bonnet for his invaluable comments that greatly helped improve this manuscript.

REFERENCES

[1] K. Goseva-Popstojanova, K. S. Trivedi, Architecture-based approaches to software reliability prediction, Computers and Mathematics with Applications 46 (7) (2003) 1023–1036.

[2] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Secur. Comput. 1 (1) (2004) 11–33.

[3] F. Brosch, H. Koziolek, B. Buhnova, R. Reussner, Architecture-based reliability prediction with the Palladio Component Model, IEEE Transactions on Software Engineering 38 (6) (2012) 1319–1339.

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

[5] R. C. Cheung, A user-oriented software reliability model, IEEE Trans. Softw. Eng. 6 (2) (1980) 118–125.

[6] S. S. Gokhale, Architecture-based software reliability analysis: Overview and limitations, IEEE Trans. Dependable Secur. Comput. 4 (1) (2007) 32–40.

[7] R. H. Reussner, H. W. Schmidt, I. H. Poernomo, Reliability prediction for component-based software architectures, J. Syst. Softw. 66 (3) (2003) 241–252.

[8] V. S. Sharma, K. S. Trivedi, Quantifying software performance, reliability and security: An architecture-based approach, J. Syst. Softw. 80 (4) (2007) 493–509.

[9] V. Cortellessa, V. Grassi, A modeling approach to analyze the impact of error propagation on reliability of component-based systems, in: CBSE, 2007, pp. 140–156.

[10] P. Popic, D. Desovski, W. Abdelmoez, B. Cukic, Error propagation in the reliability analysis of component based systems, in: 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005)., 2005, pp. 53–62.

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

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

[13] K. Goseva-Popstojanova, A. Hassan, A. Guedem, W. Abdelmoez, D. Nassar, H. Ammar, A. Mili, Architectural-level risk analysis using UML, IEEE Transactions on Software Engineering 29 (10) (2003) 946 – 960.

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

[15] W.-L. Wang, D. Pan, M.-H. Chen, Architecture-based software reliability modeling, J. Syst. Softw. 79 (1) (2006) 132–146.

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

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

[18] K. Kanoun, M. Kaaniche, C. Beounes, J.-C. Laprie, J. Arlat, Reliability growth of fault-tolerant software, IEEE Transactions on Reliability 42 (2) (1993) 205–219.

[19] M. Hamill, K. Goseva-Popstojanova, Common trends in software fault and failure data, IEEE Transactions on Software Engineering 35 (4) (2009) 484 –496.

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

[21] M. W. Lipton, S. S. Gokhale, Heuristic component placement for maximizing software reliability, in: Recent Advances in Reliability and Quality in Design, Springer Series in Reliability Engineering, Springer London, 2008, pp. 309–330.

[22] N. Sato, K. S. Trivedi, Accurate and efficient stochastic reliability analysis of composite services using their compact markov reward model representations, IEEE International Conference on Services Computing (2007) 114–121.

[23] V. Grassi, Architecture-based reliability prediction for service-oriented computing, in: Architecting Dependable Systems III, Vol. 3549 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 279–299.

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

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

[26] G. Rodrigues, D. Rosenblum, S. Uchitel, Using scenarios to predict the reliability of concurrent component-based software systems, in: Proceedings of the 8th international conference, held as part of the joint European Conference on Theory and Practice of Software conference on Fundamental Approaches to Software Engineering, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 111–126.

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

[28] A. Filieri, C. Ghezzi, V. Grassi, R. Mirandola, Reliability analysis of component-based systems with multiple failure modes, in: Proceedings of the 13th international conference on Component-Based Software Engineering, CBSE'10, 2010, pp. 1–20.

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

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

[31] T.-T. Pham, X. Défago, Reliability prediction for component-based software systems with architectural-level fault tolerance mechanisms, in: Proceedings of 8th International Conference on Availability, Reliability and Security (ARES13), in Regensburg, Germany, 2013, pp. 11–20.

[32] OMG, OMG Unified Modeling Language (OMG UML), Superstructure, v2.1.2, Object Management Group, Inc, 2008.

[33] W. Abdelmoez, D. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. Ammar, B. Yu, A. Mili, Error propagation in software architectures, in: In Proceedings of the 10th International Symposium on Software Metrics, 2004, pp. 384–393.

[34] M. Hiller, A. Jhumka, N. Suri, EPIC: profiling the propagation and effect of data errors in software, IEEE Transactions on Computers 53 (5) (2004) 512–530.

[35] S. Bernardi, M. José, D. C. Petriu, A dependability profile within MARTE, Softw. Syst. Model. 10 (3) (2011) 313–336.

[36] Reliability modeling, prediction, and improvements (2014).
URL `http://rmpi.codeplex.com/`

[37] R. H. Reussner, Automatic component protocol adaptation with the CoConut/J tool suite, Future Generation Computer Systems 19 (5) (2003) 627–639.

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

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

[40] J. A. Whittaker, J. H. Poore, Markov analysis of software specifications, ACM Trans. Softw. Eng. Methodol. 2 (1) (1993) 93–106.