Doctoral Dissertation

Reliability of Component-based Software Systems: Modeling, Prediction, and Improvements

Thanh-Trung PHAM

Supervisor: Assoc. Prof. Xavier DÉFAGO

School of Information Science
Japan Advanced Institute of Science and Technology

June 2014

"With great power, comes great responsibility."

Stan Lee

Abstract

Software systems become increasingly complex to meet the increasing requirements for software support from many different areas. In this situation, it is a significant challenge to assure the system reliability, i.e. its ability to deliver its intended service to users. The reliability of a software system during its runtime is dependent not only on its implementation but also on its usage.

Approaches in the field of component-based software reliability modeling and prediction provide the ability to predict the reliability of software systems before their operations. They build on architectural models, denoting components, transitions of control flow between them, and reliability-relevant aspects. They evaluate the models either by analysis methods or simulations in order to obtain the predicted reliability of software systems. Because of being based on the system models rather than the systems, approaches in the field can be applied at early design stages when the systems are not yet available, supporting design decisions and assisting in identifying reliability-critical parts of the system architectures.

However, existing approaches in the field are limited in their applicability because they either neglect or have only basic expressiveness for modeling several factors which influence the system reliability: (1) error propagation, (2) software fault tolerance mechanisms, and (3) concurrently present errors. Neglecting these factors leads to inaccurate prediction results. Basic expressiveness for modeling these factors likely reduces the ability to reuse the models and the support when evaluating different design variants.

This dissertation proposes the RMPI (Reliability Modeling, Prediction, and Improvements) approach, a reliability modeling and prediction approach for component-based software system, which considers explicitly error propagation, software fault tolerance mechanisms, and concurrently present errors, and supports design decisions for reliability improvements. More concretely, the approach offers the following contributions:

- *Consideration of error propagation*: The approach allows modeling error propagation for multiple execution models, including sequential, parallel, and fault tolerance execution models. The approach considers how the error propagation affects the system execution with different execution models, and it derives the overall system reliability accounting for the error propagation impact.

- *Consideration of software fault tolerance mechanisms*: The approach offers enhanced fault tolerance expressiveness, explicitly and flexibly modeling how both error detection and error handling of fault tolerance mechanisms influence the control and data flow within components. These capabilities enable modeling comprehensively different classes of existing fault tolerance mechanisms and evaluating their impact on the system reliability.
- *Consideration of Concurrently Present Errors*: The approach is the first work to support modeling concurrently present errors. With this capacity, it is possible to cover system failures caused by the concurrent presence of errors, tending to obtain accurate prediction results.

The approach provides a reliability modeling language that captures comprehensively different reliability-influencing factors into a reliability model of the system under study. The language, implemented in the RMPI schema, offers a developer-friendly modeling notation, including modeling elements for provided/required services, components, component connectors, activities, structures, etc.

The approach offers an analysis method that evaluates the system reliability model to obtain a prediction result. The method has been implemented in the RMPI tool, offering an automated transformation of the system reliability model into discrete-time Markov chains, and a space-effective evaluation of these chains.

The RMPI approach has been validated in three case studies, by modeling the reliability, conducting reliability predictions and sensitivity analyses. Via these case studies, the approach has demonstrated its ability in supporting design decisions for reliability improvements and its reusability of modeling artifacts.

The approach and its contributions have been described in the *Science of Computer Programming* journal [PDH14] (currently accepted for publication and available in an online preprint version), the *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* [PBD14], and further peer-reviewed publications [PD12, PHD12, PD13].

Keywords Reliability modeling and prediction, error propagation, software fault tolerance mechanisms, concurrently present errors, component-based software systems.

Acknowledgements

I am deeply grateful to many people who have encouraged, guided, and supported me throughout the dissertation project. The aid of these people is important and essential to me during the last four and a half years.

First, I would like to thank my wife, Nguyen Thi Kieu Chinh, for her wonderful love and support. She has always encouraged me to continue, kept me grounded, and tolerated even the most stressful periods of my work. Very big thanks also go to my parents, Pham Binh Minh and Phung Minh Hai, and my sister, Pham Minh Thu, who have given me their full and unconditional love ever, and kept encouraging me throughout the dissertation project.

In many respects, my supervisor Assoc. Prof. Xavier Défago of Japan Advanced Institute of Science and Technology (JAIST) has paved the way for my dissertation project. He not only welcomed me to join his research lab but also taught me the standards and principles of good research. Moreover, he gave directions to my dissertation with valuable aid and advice. I also thank Assoc. Prof. Huynh Quyet Thang of Hanoi University of Science and Technology (HUST), who - at the early dissertation stages - guided my introduction to the foundation topics such as software reliability engineering and component-based software engineering, and gave me additional supervision for my minor research project.

Another thank goes to François Bonnet who provided me with valuable and insightful feedback until the final dissertation stages. He was my most inspiring and dedicated discussion partner and publication coauthor. Throughout the dissertation process, he made an increasing impact on my work and my progress, and I really enjoyed all of our discussions.

I am especially thankful to Prof. Tadashi Dohi of Hiroshima University, Assoc. Prof. Toshiaki Aoki, Prof. Mizuhito Ogawa, Assoc. Prof. Kazuhiro Ogata, and Prof. Ho Tu Bao of JAIST for reviewing parts of my dissertation and providing me with very helpful feedback.

Finally, I would like to acknowledge Vietnamese Ministry of Education and Training (MOET) for their financial support, HUST for their procedural aid, and JAIST for providing a top-ranked study and research environment.

Contents

List of Figures

List of Tables

To those whom I love *I know who you are*
and those who love me...

Chapter 1

Introduction

1.1 Motivation

Software systems become increasingly complex to meet the increasing requirements for software support from many different areas. The systems provide a potentially non-homogeneous set of services for their users and their architectures are potentially complex, with interconnected and hierarchically nested components. By providing processes, methods, and tools, the software engineering discipline tries to deal with important challenges in the development and engineering of such system.

Reliability, one of the most important quality attributes, is defined as "the ability of a system or component to perform its required functions under stated conditions for a specified period of time" [1]. The users of a system service expect it to perform all required processing steps, obtaining valid computation results, delivering all expected outputs, and not producing any unwanted side effects. A *failure* occurs if the system deviates from its intended service. Reliability becomes more important if the services provided by a system are mission-critical. In such systems, failure occurrences can lead to high financial loses, environment damage, or even loses of human lives. By numerous historical software system failures, reliability has demonstrated its critical role. As an example, during the Persian Gulf War, clock drift in the Patriot system caused it to miss a scud missile targeting an American barracks in Dhahran. As consequences, there were 29 people killed and 97 others injured. It was reported that the clock drift was caused by using two different and unequal representations of a value in the software [2].

There are various efforts to assure the system reliability. This dissertation focuses on the problems of the field of component-based software reliability modeling and prediction [3–5]. The motivation of the field comes from the observation that in many software

systems, the essential design decisions at the architecture level determine the reliability levels of the systems. This is true more than usual for systems with complex architectures such as business information systems or device control systems. Approaches in the field support design decisions by providing the answers for fundamental questions such as follows:

- What are the expected reliability impacts of failure possibilities in the system architecture?
- Which parts of the system architecture are most likely to cause failures, i.e. the most critical?
- Given a set of possible changes to the system architecture, which one is expected to bring the highest reliability improvement?
- Does a planed system architecture have a positive or negative effect on the expected system reliability?
- Given a set of possible system architecture alternatives, which one is expected to bring the highest system reliability?

To answer such questions, approaches in the field are based on a system model, i.e. the model of the system under study with probabilistic annotations representing failure possibilities in the system, to predict the system reliability. By conducting reliability predictions for possible design alternatives, the approaches can evaluate and rank the alternatives according their reliabilities. Because of being based on a system model rather than the system, the approaches can be applied at early design stages when the system is not yet available for an observation of its actual reliability.

However, approaches in the field also face important and unsolved challenges regarding their practical applicability. To this end, this dissertation focuses on a set of specific factors which are insufficiently captured by existing approaches in the field, namely error propagation, software fault tolerance mechanisms, and concurrently present errors.

1.2 Problem

The main problem that this dissertation tries to deal with is component-based software reliability modeling and prediction, taking into consideration comprehensively factors which influence the system reliability, and supporting design decisions. To solve this problem, an approach is developed that provides a modeling language for modeling the

reliability of component-based software systems and an analysis method for analyzing the models based on the language to obtain prediction results.

The goal of reliability modeling and prediction includes the ability to express reliability-influencing factors and evaluate their impacts on the system reliability. These factors may relate to each of the *error propagation*, *software fault tolerance mechanisms*, or *concurrent present errors*.

According to Avizienis et al. [6], a failure is defined as a deviation of the service delivered by a system from the correct service. An error is the part of the system state that could lead to the occurrence of a failure, and is caused by the activation of a fault. The deviation from correct service can be manifested in different ways, corresponding to different failure types of the system. In general, characterizing the failure types which may occur in a system is highly dependent on the specific system. For example, two basic failure types that can be identified are content and timing failures (where, respectively, content of system's output and delivery time deviate from the correct ones).

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 software 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.

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

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.

Software fault tolerance mechanisms are often included in a software system and constitute an important means to improve the system reliability. Fault tolerance mechanisms denote any capabilities of a software system to autonomously prevent the occurrence of system failures in the presence of faults that have already activated and resulted in errors within the system. Avizienis et al. have also outlined activities of a fault tolerance mechanism, including *error detection* and *system recovery* as the two main activities where the latter includes *error handling* and possibly *fault handling*. Fault tolerance mechanisms can be applied on different abstraction levels (e.g. source code level with exception handling, architecture level with replication) [7].

The reliability impact of a fault tolerance mechanism is not only dependent on its activities but also on the whole system architecture and usage profile. For example, if a fault tolerance mechanism is never executed under a certain usage profile, its reliability impact is considered as nothing. Analyzing the reliability impact of fault tolerance mechanisms becomes apparently a challenge when they are applied at architecture level, in a component-based software system because: (1) Fault tolerance mechanisms 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.

Situations involving multiple failures are frequently encountered. System failures are often turned out on later examination to have been caused by different errors [6]. 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.

Therefore, the modeling language is expected to capture all the these aspects and the analysis method should take them into account in order to obtain prediction results.

1.3 Existing Solutions

By comparing the state of the art of the existing approaches in the field of component-based software reliability modeling and prediction with the problem given above, it is possible to capture several drawbacks of existing approaches, which limit 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 [3].

Although error propagation is an important element in the chain that leads to a system failure, many approaches (e.g. [4, 8–12]) 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 [13]. On the other hand, approaches that do consider error propagation (e.g. [13, 14]) 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) [15], 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.

Many approaches (e.g. [9, 16, 17]) do not support modeling fault tolerance mechanisms. This forces modelers to implicitly model fault tolerance mechanisms 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 fault tolerance mechanisms and failure conditions (e.g. [15, 18]). They lack flexible and explicit expressiveness of how both error detection and error handling of fault tolerance mechanisms 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 fault tolerance mechanisms with modeling the system architecture and usage profile.

Further approaches provide more detailed analysis of individual fault tolerance mechanisms (e.g. [19–21]). 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 fault tolerance mechanisms employed in different parts of a system architecture influence the overall system reliability, especially when evaluating for architecture variants under varying usage profiles.

To the best of our knowledge, existing approaches do not support modeling concurrently present errors. Neglecting concurrently present errors can lead to inaccurate prediction results because there exist system failures that cannot be covered by existing approaches, which is confirmed by Hamill et al. [22] with two large, real-world case studies (GNU Compiler Collection (GCC) and NASA Flight Software).

Many approaches (e.g. [10, 15, 23]) use Markov models to conduct reliability predictions. They require the models to be directly created in the Markov-model notation, which is not aligned with the concepts and notations typically used in software engineering (e.g. UML or SysML). They map Markov states to software components (or their internal behavioral states) but they do not explicitly deal with other concepts of the software engineering domain (e.g. provided/required services, component connectors, etc.). In these approaches, the system is represented through a set of states and transition probabilities between them. Direct creation and interpretation of Markov models may discourage software developers who are not familiar with the Markov-model notation, especially when it is to be done repetitively during the development process.

Some approaches (e.g. [12, 14, 16, 17, 24, 25]) use UML or UML-like notation with reliability properties, such as failure probabilities. Such models can be transformed (manually or by tools) into Markov models. In these approaches, software developers can utilize existing design specifications to conduct reliability predictions and the complexity of the underlying analysis techniques is hidden from developers.

1.4 Contributions

The contribution of this dissertation is the RMPI (Reliability Modeling, Prediction, and Improvements) approach for component-based software reliability modeling and prediction that considers explicitly the discussed reliability-influencing factors, and supports design decisions for reliability improvements. More concretely, it offers the following contributions:

- *Consideration of error propagation*: The approach allows modeling error propagation for multiple execution models, including sequential, parallel, and fault tolerance execution models. The approach considers how the error propagation affects the system execution with different execution models, and it derives the overall system reliability accounting for the error propagation impact.
- *Consideration of software fault tolerance mechanisms*: The approach offers enhanced fault tolerance expressiveness, explicitly and flexibly modeling how both error detection and error handling of fault tolerance mechanisms influence the control and data flow within components. These capabilities enable modeling comprehensively different classes of existing fault tolerance mechanisms and evaluating their impact on the system reliability.
- *Consideration of Concurrently Present Errors*: The approach is the first work to support modeling concurrently present errors. With this capacity, it is possible to cover system failures caused by the concurrent presence of errors, tending to obtain accurate prediction results.

The approach provides a reliability modeling language that captures comprehensively different reliability-influencing factors into a reliability model of the system under study. The language, implemented in the RMPI schema, offers a developer-friendly modeling notation, including modeling elements for provided/required services, components, component connectors, activities, structures, etc.

The approach offers an analysis method that evaluates the system reliability model to obtain a prediction result. The method has been implemented in the RMPI tool, offering an automated transformation of the system reliability model into discrete-time Markov chains, and a space-effective evaluation of these chains.

1.5 Validation

This dissertation includes three case studies, which serve to demonstrate the applicability of the RMPI approach, including modeling the reliability, conducting reliability predictions and sensitivity analyses, and supporting design decisions. They are based on the reporting service of a document exchange server, the WebScan system, and the DataCapture system, giving evidence of the applicability of the approach on different kinds of software systems. The first two case studies (based on the reporting service and the WebScan system) include comparisons between prediction results and simulations, different sensitivity analyses and design decision supports, as well as introductions of fault tolerance mechanisms at both the service implementation level and the architecture level. The third case study features a prototype implementation of the DataCapture system, compares prediction results with measurements, and illustrates the approach's effectiveness for a family of related software systems. All these studies support the claim for the approach's validity.

In short, the RMPI approach accomplishes the goal regarding the target problem. It overcomes several important drawbacks of the existing approaches, and it offers a comprehensive and validated solution for supporting design decisions through reliability modeling and prediction for component-based software systems.

1.6 Outline

This section gives an outline of the dissertation's remaining chapters.

- Chapter 2 introduces the existing basics on which the RMPI approach builds.
- Chapter 3 presents the methodology and the reliability modeling capacities of the approach.
- Chapter 4 describes the analysis method provided by the approach for reliability predictions, and architectural changes supported by the approach for reliability improvements.
- Chapter 5 validates the approach in three case studies, by modeling the reliability, conducting reliability predictions and sensitivity analyses, and demonstrating its capability in supporting design decisions.
- Chapter 6 concludes the dissertation with a brief summary, and a discussion of limitations of the approach, and an examination of future research directions.

Chapter 2

Software Components and Reliability: Basics and State-of-the-Art

The RMPI approach introduced in the dissertation allows a comprehensive reliability modeling for component-based software systems and aims at increasing the reliability prediction accuracy for such systems and supporting design decisions for reliability improvements. This chapter introduces the basics on which the approach builds. Section 2.1 introduces the basic reliability concepts and existing approaches in the area of software reliability analyses. Section 2.2 discusses the state-of-the-art in deriving software reliability estimates, which are needed as the approach's inputs. Section 2.3 introduces Markov chains as the approach's underlying formalism. Section 2.4 introduces the field of component-based software reliability modeling and prediction, to which the approach belongs. Section 2.5 gives an overview of software fault tolerance mechanisms. Section 2.6 surveys most related work.

2.1 Software Reliability

2.1.1 Basic Concepts

A widely accepted basic concepts and taxonomy of dependable and secure computing has been defined by Avizienis et al. [6]. This dissertation utilizes a part of their definitions as a foundation of terminologies. In their work, the authors introduce a *system* as entity that interacts with its environment (i.e. other systems, including users) and delivers its *services* via a set of *service interfaces*. The system could be composed of a set of

components, where each component is a system itself. 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 service to deviate from correct service. *Reliability* is related to the system's ability to provide correct services. It is one of the *dependability attributes*, along with availability, safety, integrity, and maintainability. While the scope of these basic concepts and the taxonomy is very wide, this dissertation focuses on software system composed of software components.

Moreover, Avizienis et al. also include a classification of service failures which illustrates a wide range of different deviations of the delivered service from the intended service. From the failure domain viewpoint, deviations are classified as *content failures* if the service's output is not intended, as *timing failures* if the delivery time of the service is too early or too late. *Halt failures* are due to no service delivery and *erratic failures* result from inconsistent service deliveries. From the detectability viewpoint, a service failure is either *signaled failure* or *unsignaled failures*. By grading the consequences of failures for the environment of the system, failure severities can be defined, ranging from *minor failures* to *catastrophic failures*. Within its restricted scope, the RMPI approach in this dissertation also allows defining custom failure types, comparable to the range of the service failure types described above (see Section 3.2.1.2).

The authors further group the means to attain dependability into four major categories:

- *Fault prevention* aims to prevent the occurrence or introduction of faults by improving the quality of the development and engineering process.
- *Fault tolerance* aims to avoid service failures in the presence of faults. It is carried out via error detection (identifying the presence of errors) and system recovery (including error handling for eliminating errors from the system state and possibly fault handling for preventing faults from being activated again).
- *Fault removal* aims to reduce the number and severity of faults via detecting existing faults, using verification and validation methods, and eliminating the detected faults.
- *Fault forecasting* aims to estimate the current or future dependability attributes for the system under study, e.g. the present number, the future frequency, and the likely consequences of faults. Available analysis methods can roughly be classified as being qualitative (e.g. Failure Modes And Effects Analysis), quantitative (e.g. Markov chains, stochastic Petri nets), or mixed (e.g. reliability block diagrams, fault trees).

Each of these categories has its own importance, regardless of the dependability attribute under consideration is reliability, availability, safety, integrity, or maintainability. While fault prevention, fault tolerance, and fault removal aim to reduce the possibility of service failures as much as possible, fault forecasting accepts the fact that a certain possibility of failures remains in all cases and tries to estimate this possibility and its consequences. The RMPI approach in this dissertation belongs to the fourth category as a reliability modeling and prediction approach.

2.1.2 Software Reliability Analyses

The context of the RMPI approach is determined via the fraction of analysis methods specially tailored to reliability, and under the term *fault forecasting*. There are various methods which have been proposed and are widely accepted. However, they do not necessarily focus on software systems. Target metrics may be qualitative, e.g. identifying different failure types of a system, or quantitative, e.g. estimating failure probabilities or failure rates of a system. Examples of analysis methods include the Failure Modes and Effects Analysis, fault trees, reliability block diagrams, Markov-based analyses, reliability growth analyses. Usually, there are a number of variations for each method and multiple analyses can be applied on the system under study. The term *reliability engineering* has been coined to represent the systematic consideration of the reliability aspects throughout design and production processes (for a detailed overview, see [26]).

Even though the nature of software faults is different from that of hardware faults, there have been efforts on extending the classical reliability analysis methods from hardware to software, resulting in software-specific or combined software-hardware analyses (for an example of the Failure Modes and Effects Analysis adapted for software system, see [27] and for an overview, see [28]). The major drawbacks of such efforts are also obvious. While failures of hardware components are usually caused by physical deterioration and environmental influences, those of software components are usually due to the human design faults whose activation patterns may be complex and unique for each software component. Moreover, the reliability of a software component is highly dependent on the usage of the component, which, in turn, is dependent on the usage of the system in non-trivial ways. For example, a little change to the input parameter value of a software service may lead to an entirely different control and flow data throughout the system, activating different software faults. Therefore, such efforts are limited in their applicability to software systems with basic functionalities and static control and data flow. For more complex systems, the abstractions are either oversimplified or the analysis effort gets out of control. Reliability growth analyses are the only methods that have gone through a major evolution towards software systems. In their software-specific forms,

these methods focus on the process of testing software systems or software component, and removing detected faults. Software reliability growth models (for a recent overview, see [29]) allow for estimations of the reliability growth of a software system during further testing activities. Traditionally, software reliability growth models have been applied at the system level, without an attempt to consider software components and their reliability impact. Authors, e.g. Musa [30], have focused on software reliability growth models and have coined the term *software reliability engineering* to represent the software-specific evaluation of reliability engineering with software reliability growth models as a central constituent (for a more recent overview, see [31]).

However, when applied to modern component-based software systems, software reliability growth models are limited in their applicability. Because in a software reliability growth model analysis, the reliability impacts of software components with in the system are unclear, its results cannot be reused in a family of similar software systems. Also, in order to apply software reliability growth models, it is required to install and execute the complete software system under study. Therefore, software reliability growth models cannot be used easily to make comparisons between design alternatives of a software system, not at the early design stages, when the software system is not yet available. To solve the problem, the field of component-based software reliability modeling and prediction has emerged. Approaches in this field consider a software system as a composition of software components. They model the control and data flow between components in the system and provide a method to express the reliability of the system based on the individual component reliabilities. The approaches still face the challenge of estimating failure probabilities or failure rates of individual components. However, they can employ software reliability growth models at component level as well as other estimation methods (see Section 2.2).

While approaches in the field of component-based software reliability modeling and prediction establish a major advance in analyzing the reliability of component-based software systems, their applicability is still limited because of lacking support for expressing error propagation, software fault tolerance mechanisms, as well as concurrently present errors. The RMPI approach in this dissertation overcomes these drawbacks and provides a comprehensive reliability modeling and prediction for component-based software systems.

2.2 Software Reliability Estimation

This section discusses methods for software reliability estimation, mainly focusing on methods for estimating failure probabilities and failure rates of software components.

This is because approaches in the field of component-based software reliability modeling and prediction need such estimates as their inputs. Software reliability is modeled stochastically because of the reasons as follows:

- The knowledge regarding faults in the software system and their activation patterns is lacking.
- In order to reduce the modeling complexity, reliability models often include probabilistic abstractions from the actual behavior of the software system.
- Because the exact way (i.e. the exact input parameter value, sequence, and timing) users invoke the services of the system is unknown beforehand, the usage of the system cannot be clearly described.

For these reasons, estimating the reliability of a software component is apparently a significant challenge. A research field has emerged to address this problem. Here, the discussion focuses on main families of software reliability estimation methods, including software reliability growth models, software defect prediction models, and several other methods.

2.2.1 Software Reliability Growth Models

One of the most successful families of analysis methods in the software reliability engineering discipline are software reliability growth models [31]. Besides being applied at the system level, they can be employed to determine failure probabilities and failure rates of software components as inputs for approaches in the field of component-based software reliability modeling and prediction [3].

The general schema of a software reliability growth model is depicted in Fig. 2.1. A software reliability growth model observes a software system or a software component under test and records the increasing number of detected faults during the test. For terminating applications, the test time could be measured as the number of executed test runs. For continuous applications, the test time could be measured in sense of system or component execution time. From start of test t_S to the present test time t_P, a parameterized statistical mean value function *m(t)* is determined to fit the historical numbers of detected faults. This function can be used to predict the progress of the testing process, e.g. the total number of detected faults at the planned end of test t_E. Besides, many software reliability growth models include an estimation of the total number of faults contained in the system or component under test at t_S, showing the remaining number of faults during the operational phase (after t_E). Different mean value

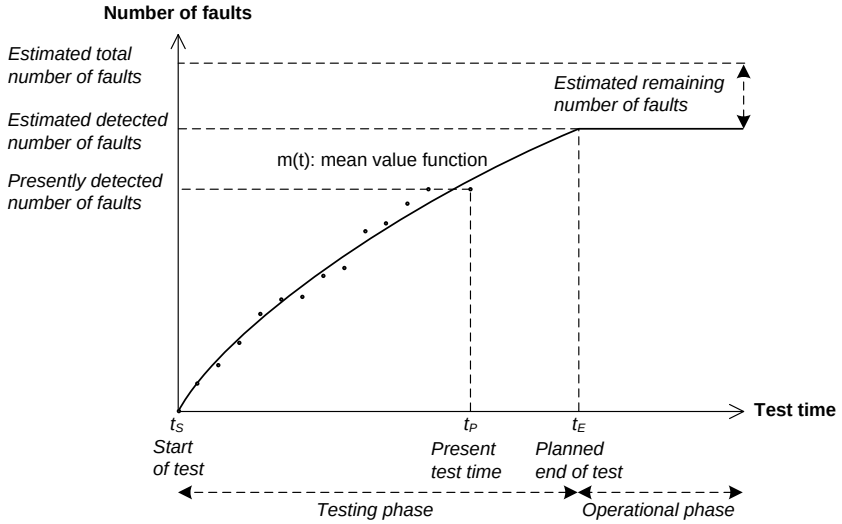


FIGURE 2.1: The general schema of a software reliability growth model

functions *m(t)* has been proposed by several authors, corresponding to different types of test progressions encountered in practice (for an overview, see [29]). The proposals are generally based on the assumption that the rate of fault detection decreases over time, leading to mean value functions with decreasing slopes. Moreover, most software reliability growth models share the common assumptions as follows:

- The removal of a detected fault is instantaneous.
- The correction of a detected fault never introduces new faults into the system or the component under test.

Moreover, software reliability growth models also reason about the failure rates of the systems or components under test. Under the assumption that during the test, each failure corresponds to one detected new fault, it is possible to derive a failure rate function $\lambda(t)$, representing the failure rate of the system or component at time t (i.e. $\lambda(t) = dm(t)/dt$). Therefore, the expected failure rate of the system or component at the planned end of test is $\lambda(t_E)$. As another possibility, in order to meet the given failure rate requirement, t_E can be selected in a dynamic way. Also, assuming that the usage profile of the system or component in the operational phase can be determined, and if test inputs are selected randomly according the usage profile, it is possible to predict the failure rate of the system or component in the operational phase.

Several authors (e.g. [3, 32]) have shown that software reliability growth models can be used to gather inputs for approaches in the field of component-based software reliability modeling and prediction. For software components which already exist and have undergone a certain amount of testing, their failure rates can be estimated by software

reliability growth models. Recently, Koziolek et al. [33] have illustrated this usage of software reliability growth models on a large industrial control system. However, Apel [34] points out several problems with regard to software reliability growth models, including model selection problem, lack of metrics for analyzing long-term predictability, and lack of empirical studies on prediction accuracy. Furthermore, when applying software reliability growth models in practice, their underlying assumptions are likely to be violated [35], e.g. the removal of a fault may be not instantaneous, the correction of a fault may introduce new faults.

When software reliability growth models are used to determine failure rates of software components, there are additional problems to consider. In case they are applied to each component in isolation, it is necessary to test each component according to its own usage profile within the whole system architecture. In case system-level test runs are used, it is necessary to determine how many times each component are invoked in each test run and when there is a failed test run, which component is to blame for.

There exists work to improve software reliability growth models and deal with their problems (for a summary, see [31]). Yet, the use of software reliability growth models to gather inputs for approaches in the field of component-based software reliability modeling and prediction needs further investigation regarding the challenges mentioned above.

2.2.2 Software Defect Prediction Models

Efforts related to estimating the number of faults (or *defects*) contained in software systems or component are known as software defect prediction models. *Defect count* (the number of defects) and *defect density* (the number of defect in a connection with the code size) are the target metrics. Software defect prediction models utilize different kinds of artifacts from different stages of the software development as information sources. It is assumed that the following factors have the most influence on the number of defects in a software system or component:

- *Size and complexity*: It is expected that the number of defects in a software system or component is proportional to its size and complexity. Code size can be measured in the number of *lines of code* (LOC), *code segments*, or *machine code instructions*. McCabe's *Cyclomatic Complexity* [36] (related to the number of decision statements in the code), Halstead's *Volume, Difficulty, and Effort* [37] (related to the number of operands and operators in the code), and Albrecht's *Function Points* [38] (related to the amount of functionality provided by a component or system, usually obtained from specifications rather than the code) are several examples of metrics for code complexity.

- *Test-related factors*: To estimate the total (or remaining) number of defects, it is possible to utilize an existing test history. Examples of metrics include the number of detected defects and the accomplished test coverage (can be statement coverage, branch coverage, etc.) [39], and the testability (i.e. the possibility that the test detects possible defects, usually determined via static code analyses) [40] of a system or component.

- *Process quality*: A high-quality development process is expected to produce software systems or components with less defects. The SEI Capability Maturity Model (CMM) [41], a process quality model, has been used to estimate defect densities.

Based on the existing data sets from the software development, many software defect prediction models attempt to obtain general formulas for the number of defects (e.g. [42, 43]). Fenton et al. [44] have pointed out flaws in such works, including a tendency towards oversimplification by focusing on a subset of the relevant factors, and incorrect use of statistical analyses with misleading results. However, this research field is still very active so far (for a recent review, see [45]). More advanced software defect prediction models have been developed using different formalisms, e.g. Capture-Recapture model [46], Bayesian networks [47]. From code metrics data, it is also possible to employ machine learning and data mining to estimate the number of defects (e.g. [48, 49]).

Although there are many existing software defect prediction models, using them to derive input information for approaches in the field of component-based software reliability modeling and prediction is a challenge. There is no straightforward relation between the number of faults in a software component and its failure rate. The component's failure rate is dependent on the possibility that existing faults are activated under a certain usage profile of the component. Recently, Zeimmermann et al. [50] have pointed out that it may be invalid to reuse results of software defect prediction models across multiple software development projects, whether or not these projects employ the same software development process model, or come from the same domain. However, in a software development process, software defect prediction models can be used to guiding decisions and further research efforts may fill the gap towards providing input information for reliability prediction.

2.2.3 Further Approaches to Software Reliability Estimation

There have been further efforts to estimate failure rates or failure probabilities of a software component or system. In order to assure required reliability levels, an operational test or validation test can be conducted at late testing stages [51, 52]. The component

or system is tested as if it executed according to its usage profile of the operational field. Then, it is possible to deduce the upper bounds for failure rates with certain levels of confidence from a certain amount of failure-free execution by applying frequentist inference [53]. It is also possible to apply methods and tools from model-based testing to support and automate partially the testing process [54–56]. As an example, JUMBL (J Usage Model Builder Library) [54] generates automatically test cases according to a usage profile specified as a Markov model, and then determines reliability estimates and confidence levels from the results of executed test cases. However, applying such methods on software systems or components with high reliability requirements is a major challenge because of extremely high testing efforts required [3].

There have also been efforts specially targeting component-level reliability estimation, known as component reliability models [11, 57]. Component states and transition probabilities between them are expressed by Markov models. Usually, there are two categories of states, namely normal operation and failure. Different information sources can be used to build component reliability models, e.g. component specifications, domain knowledge, use cases descriptions for components, simulations, as well as existing functionally similar components [11]. Then, it is possible to determine the failure probability or failure rate of the component by applying Markov theory. Component reliability models appear to be promising. Because of being based on a component model rather the component itself, they can be applied even if a component has not been implemented and executed under test, and they are not restricted by the level of reliability requirements. However, building such models is not straightforward and the problem of estimating reliability characteristics of a component is decomposed into a set of problems of estimating a component's internal properties, i.e the original problem is not completely resolved. When using component reliability models to gather input information for approaches of the field of component-based software reliability modeling and prediction, it is necessary to put each component reliability model in the component's usage profile within the whole system architecture. Further research efforts on component reliability models could increase their applicability to a more spread use.

Palviainen et al. [58] have summed up further efforts, known as heuristic reliability evaluation, to derive component reliability estimations, considering different reliability-influencing factors, including component size, complexity metrics and maturity levels, testing and operational data from existing similar components, level of experience of component developers or component vendors, etc. However, the authors have also pointed out that such efforts are not as strong as test-based approaches and component reliability models because their results may become invalid when applied across software development projects or companies.

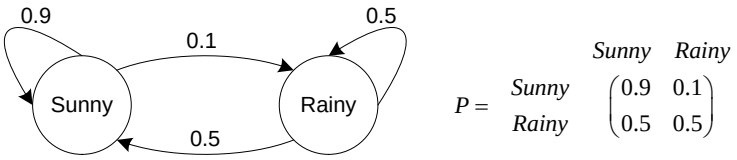


FIGURE 2.2: Example of DTMC

2.3 Markov Chains

Markov chains have been used as a fundamental modeling formalism of many approaches in the field of component-based software reliability modeling and prediction, including the RMPI approach in this dissertation. This section presents a brief introduction to Markov chains, limited to aspects relevant to the dissertation's context (see [59] for a detailed consideration).

A Markov chain is a stochastic process (or random process, i.e. its operation is described by probability distributions instead of being predetermined) which has a discrete (finite or countable) set of states (called state space) and the property that given the present, the future is conditionally independent of the past (called Markov property). A discrete-time Markov chain (DTMC) has transitions between its states at certain points in time, while a continuous-time Markov chain (CTMC) allows state transitions at any time. Markov chains have many applications as stochastic models of real-world processes, and different properties of a Markov chain can be examined by Markov theory.

Formally, a DTMC is described by a state space $\mathbf{S} = \{s_1, s_2, ..., s_n\}$ and transitions $\mathbf{T} = \{t_1, t_2, ..., t_m\}$ where each entry $t_k = (source\,(t_k)\,, target\,(t_k)\,, probability\,(t_k))\,, k \in [1, m]$ denotes the transition from source state $source\,(t_k) \in \mathbf{S}$ to target state $target\,(t_k) \in \mathbf{S}$ with transition probability value $probability\,(t_k) \in [0, 1]$. The DTMC can also be described by a $n \times n$ transition matrix $\mathbf{P}$, with each entry $p_{ij} \in [0, 1]\,, \forall i, j \in \{1, 2, ..., n\}$ representing the transition probability from s_i to s_j. The sum of entries in each row equals to 1: $\sum_{j=1}^{n} p_{ij} = 1, \forall i \in \{1, 2, ..., n\}$.

Example 2.1. *Fig. 2.2 shows an example of DTMC (with its transition matrix) representing the weather conditions. The weather in each day is one of states* $\mathbf{S} = \{Sunny, Rainy\}$. *The weather may stay the same or change between days. The transition matrix shows that a sunny day is 90% likely to be followed by another sunny days, and a rainy day is 50% likely to be followed by another rainy day. The columns of the transition matrix* $\mathbf{P}$ *can be labeled "Sunny" and "Rainy" and the rows of* $\mathbf{P}$ *can be labeled in the same order. As a result of the Markov property, the weather of tomorrow is only dependent on the weather of today, not on the weather history of days before today. Compared to reality, this is an assumption to make the model less complicated and therefore easier to analyze. Despite of its abstractions, as for certain purposes (e.g. predicting the weather,*

or steady state of the weather), the model may still be a capable representation of the corresponding real-world process.

In a DTMC, an absorbing state is a state that, once entered, cannot be left. If $s_i \in \mathbf{S}$ is an absorbing state, then $p_{ii} = 1, p_{ij} = 0 \; \forall i \neq j$. A DTMC is an absorbing DTMC if there is at least one absorbing state, and it is possible to go from any state to at least one absorbing state in a finite number of steps. In an absorbing DTMC, a state that is not absorbing is called transient. Let an absorbing DTMC with transition matrix $\mathbf{P}$ have t transition states and r absorbing states, then

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

where $\mathbf{Q}$ is a $t \times t$ matrix, $\mathbf{R}$ is a nonzero $t \times r$ matrix, $\mathbf{0}$ is an $r \times t$ zero matrix, and $\mathbf{I}_r$ is the $r \times r$ identity matrix. Thus, $\mathbf{Q}$ describes the probability of transitioning from some transient state to another while $\mathbf{R}$ describes the probability of transitioning from some transient state to some absorbing state. A basic property about an absorbing DTMC is the expected number of visits to a transient state s_j starting from a transient state s_i (before being absorbed), which is the (i, j)-entry of the fundamental matrix

$$\mathbf{N} = (\mathbf{I}_t - \mathbf{Q})^{-1},$$

where $\mathbf{I}_t$ is the $t \times t$ identity matrix. Another property is the probability of being absorbed in the absorbing state s_j when starting from transient state s_i, which is the (i, j)-entry of matrix $\mathbf{B} = \mathbf{NR}$.

Similar to a DTMC, a CTMC is also described by a state space $\mathbf{S} = \{s_1, s_2, ..., s_n\}$ and transitions $\mathbf{T} = \{t_1, t_2, ..., t_m\}$. However, each transition $t_k = (source\,(t_k), target\,(t_k), rate\,(t_k))$, $k \in [1, m]$ is associated with a transition rate value instead of a transition probability value. The CTMC can also be described by a transition rate matrix $\mathbf{A}$ with each entry $a_{ij} \in \mathbb{R}_0^+, \forall i, j \in \{1, 2, ..., n\}$ indicating that the transition rate from s_i to s_j. For each row, its diagonal entry is $a_{ii} = -\sum_{j=1, j\neq i}^{n} a_{ij}, \forall i \in \{1, 2, ..., n\}$. A transition rate $a_{ij} > 0$ indicates that transitions from s_i to s_j take place with frequencies specified by the exponential distribution with parameter $1/a_{ij}$. A zero transition rate $a_{ij} = 0$ indicates that transitions from s_i to s_j never take place. Different from DTMCs, each state s_i of a CTMC has a variable $\tilde{T}_i$ of sojourn time (or the amount of time between transition occurrences) according to a continuous time scale. The expected sojourn time of s_i is determined based on its transition rates: $E\left(\tilde{T}_i\right) = 1/-a_{ii}$.

DTMCs, CTMCs, and other related formalisms, e.g. semi-Markov process, constitute a powerful means for approaches in the field of component-based software reliability

modeling and prediction to represent the control and data flow between components throughout the whole system architectures. While other formalisms focus on the inputs, outputs, and internal progressions of the system (e.g. state charts [60], finite state machines [61]), Markov models capture various aspects of the system behavior (e.g. the system usage and its influence on the service execution via probabilistic annotations), resulting in a high-level representation of the system. Then, existing Markov theory can be applied to evaluate the created Markov models for reliability predictions. The RMPI approach in this dissertation exhibits novel methods to utilize absorbing DTMCs for a comprehensive reliability modeling and prediction (see Chapter 4).

2.4 Component-based Software Reliability Modeling and Prediction

As mentioned in Section 2.1.2, approaches in the field of component-based software reliability modeling and prediction aim to overcome the drawback of traditional reliability analysis methods with regard to component-based software systems. This section briefly introduces the field (for surveys, see [3–5]). Similar to several related analysis methods (e.g. fault trees, reliability block diagrams), approaches in the field assume that the system's overall failure possibilities can be determined from its components' failure possibilities. However, the ways to express the system structure and its components' relationships in traditional methods (e.g. AND/OR relationships in fault trees) are oversimplified to cover complex relations between components. From that, approaches in the field choose a more expressive formalisms to represent the data and control flow between components within the overall system architecture.

Cheung's approach [8] is one of the first approaches that consider the system reliability with respect to components utilization and their reliabilities. The approach has much influence on the development of the field of component-based software reliability modeling and prediction and serves as a fundamental model for a lot of approaches in the field. He introduces an absorbing DTMC to describe the control flow through the system architecture. In the DTMC, states represent software components and transitions represent the transfer of control between components when executing a certain system service. Each component is annotated with an independent failure probability, representing the possibility that this component fails to perform its function during a service execution. For reliability evaluation, two absorbing states are added, indicating successful service execution and service failure. Without loss of generality, one more initial state can be added such that service execution always starts in this state. By using Markov theory

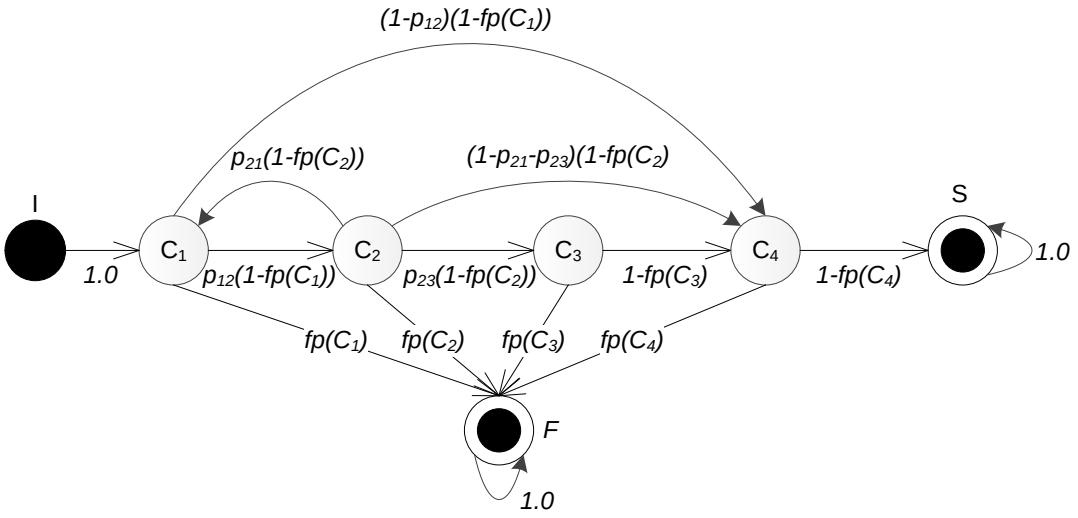


FIGURE 2.3: Example of a system architecture modeled by an absorbing DTMC.

(see Section 2.3), the probability of successful service execution, i.e. the system reliability, can be calculated as the probability of reaching the success state from the initial state. The approach assumes that a service execution finishes in either the success state or the failure state, i.e. an assumption of terminating applications. Because of being based on the system model and component failure probabilities, the approach can be applied at early design states when the system is not yet available, predicting the expected system reliability of the system implementation. However, applying the approach needs the estimates of required inputs even though sensitivity analyses can be conducted to assess the impact of uncertain input estimations.

Example 2.2. *Fig. 2.3 shows an example of a system architecture modeled by an absorbing DTMC. In addition to states representing components* C_1*,* C_2*,* C_3*, and* C_4*, an initial state* I*, a success state* S*, and a failure state* F *are added. In order to calculate the system reliability, i.e. the probability of reaching state* S *from state* I*, it is necessary to estimate component failure probabilities* $fp(C_1)$*,* $fp(C_2)$*,* $fp(C_3)$*, and* $fp(C_4)$*, as well as transition probabilities* p_{12}*,* p_{21}*, and* p_{23}*. This DTMC allows for expressing all possible control flow paths and their probabilities, e.g. the execution path* I-C_1-C_4-S *with probability* $1.0 \times (1 - p_{12})\,(1 - fp\,(C_1)) \times (1 - fp\,(C_4))$*. The model assumes that the control transitions between components have the Markov property. This assumption can lead to paths that are possible in the model but not in reality. For example, in reality, the number of cycles between components* C_1 *and* C_2 *may be limited by a maximum number* $max(n)$*, while the model allows for an arbitrary number of cycles before finishing in either success or failure states. However, if transition probabilities are chosen such that the expected number of cycles between components* C_1 *and* C_2 *corresponds to the average number of cycles in reality, the model can still provide sufficiently accurate results.*

Beyond absorbing DTMCs, other related formalisms (e.g. DTMCs, CTMCs, semi-Markov processes) have been used by many approaches in field of component-based software reliability modeling and prediction (e.g. [9, 10, 13–15, 17, 18, 24, 25, 62–64]) to represent the control and data flow throughout the system architecture and its failure possibilities (for a comprehensive survey, see [3]). Further categories of approaches include *path-based* and *addictive* approaches, using less related formalisms, but are still be considered as in the field by the survey:

- Path-based approaches calculate the system reliability explicitly considering the possible execution paths of the system. A sequence of components along different paths is obtained either experimentally by testing or algorithmically. The reliability of each path is calculated by multiplying the reliabilities of the component along the path. Then, the system reliability is calculated by averaging path reliabilities over all paths. These approaches are not affected by the Markov assumption. However, a comprehensive consideration of all execution paths may be impossible because of extremely high number of possible execution paths (e.g. when the system model contains loops). Therefore, they often consider only the most frequent paths.

- Addictive approaches assume that each component reliability can be modeled by non-homogeneous Poisson process (NHPP). Then, system failure process is also NHPP with cumulative number of failures and failure rate function that are the sums of the corresponding functions for each component. These approaches assume that each of the components is always visited during a service execution (i.e. not explicitly considering the system architecture).

Throughout the years, approaches in the field of component-based software reliability modeling and prediction offer several differences and extensions compared to Cheung's model, e.g. the consideration of execution environment (e.g. [14, 18, 24]), the inclusion of uncertainty analyses (e.g. [65, 66]), or the provision of design-oriented input modeling language (e.g. [9, 16, 17, 24, 25, 67]). Still, they suffer from several drawbacks in with regard to the consideration of reliability-influencing factors, namely error propagation, software fault tolerance mechanisms, and concurrently present errors (see Section 2.6), therefore are limited in their applicability and accuracy. Thus, by overcoming these drawbacks, the RMPI approach in this dissertation offers a comprehensive reliability modeling and prediction and constitutes further progress in the field.

2.5 Software Fault Tolerance Mechanisms

This section introduces the most important concepts related to software fault tolerance mechanisms, based on existing surveys and overviews [6, 7, 31, 68, 69]. Fault tolerance mechanisms denote any capabilities of a software system to autonomously prevent the occurrence of system failures in the presence of faults that have already activated and resulted in error within the system. It is obvious that fault tolerance mechanisms influence the probability of successful service execution. The RMPI approach in this dissertation supports modeling fault tolerance mechanisms and takes them into consideration for reliability prediction.

A key supporting concept for fault tolerance mechanisms is redundancy, that is, additional resources that would not be required if fault tolerance mechanisms were not being employed. Redundancy can take several forms: function, information, and time [7, 69]. Functional redundancy includes additional software implementations (e.g. components, objects) used in the system to support fault tolerance, introducing the concept of design diversity. Information or data redundancy includes the use of additional information with data and the use of additional forms of data to assist in fault tolerance, introducing the concept of data diversity. Temporal redundancy involves the use of additional time to perform operations required to accomplish fault tolerance. Fault tolerance mechanisms can also be differentiate between single-version software mechanisms (denoting the absence of functional redundancy), multiple version software mechanisms (employing functional redundancy), and multiple data representation mechanisms (employing data redundancy) [31].

Avizienis et al. [6] have described in detail the principle of fault tolerance mechanisms. A fault tolerance mechanism is carried out through error detection and system recovery. Error detection is to determine the presence of an error. It can be performed on demand (e.g. using an acceptance test to check the computation result) or in a preemptive manner (e.g. testing the system health regularly). Error handling possibly followed by fault handling together form system recovery. Error handling is to eliminate errors from the system state, e.g. using rollback to bring the system back to a saved state the existed prior to error occurrence, using rollforward to reach a new state without detected errors, or using compensation in case the erroneous state contains enough redundancy to enable error to be masked. Error handling can also performed on demand (e.g. after the acceptance test has identified an erroneous computation result) or in a preemptive manner (e.g. using software rejuvenation [70], aimed at removing the effects of software aging before they lead to failure). Fault handling is to prevent faults from being activated again and may include fault diagnosis, isolation, system or component reconfiguration, and reinitialization. Fault handling may be followed by corrective maintenance, aimed

at removing faults the were isolated by fault handling. Corrective maintenance requires the participation of an external agent and therefore is not included in fault tolerance mechanisms.

Many fault tolerance mechanisms have been proposed throughout the years, either focusing the ability to tolerate activated faults in an on-demand manner or they are carried out as periodic activities (i.e. in a preemptive manner). The former category includes, among others, Recovery Blocks, Retry Blocks, N-Version Programming, and N-Copy Programming. A Recovery Block consists of an acceptance test and primary and alternate try blocks. It first attempts to ensure the acceptance test (i.e pass a test on the acceptability of a result of an alternate) by using the primary alternate (or try block). If the primary alternate's result does not pass the acceptance test, then the remaining alternates are attempted until an alternate's result passes the acceptance test. If no alternates are successful, an error occurs. A Recovery Block employs functional redundancy (multiple alternates of the same functionality) and temporal redundancy (the overhead of executing alternates). Retry Blocks are similar to Recovery Blocks but employ data redundancy instead of functional redundancy. They execute multiple times the same behavior with logically equivalent variants of the input data (obtained by using data re-expression). Different from Recovery Blocks and Retry Blocks, N-Version Programming and N-Copy Programming execute all functional or data alternatives (or variants) concurrently. They do not conduct acceptance tests for explicitly detecting errors but rely on voters (which compare the results of variants to determine the correct result). Several variations of these fault tolerance mechanisms have been proposed, ranging from simple uses of different voters to combinations of multiple fault tolerance mechanisms (e.g. The Consensus Recovery Block mechanism which combines Recovery Block and N-Version Programing implementation mechanisms). Additionally, new fault tolerance mechanisms are often proposed to overcome the limitations associated with previous mechanisms, to provide fault tolerance for specific problem domains, or to apply new technologies to the needs of software fault tolerance, while attempting to maintain the strengths of the foundational mechanisms [7].

In traditional form, fault tolerance mechanisms have been employed at implementation levels of software systems. However, it is possible to apply fault tolerance mechanisms at the architectural levels of software systems [71], e.g. introducing new components or assigning special responsibilities to components for purposes of fault tolerance . In the field of component-based software reliability modeling and prediction, there are still very few approaches that explicitly consider fault tolerance mechanisms (see Section 2.6.2). The RMPI approach in this dissertation acknowledge that fault tolerance mechanisms can be employed at the architectural levels of software systems and may influence significantly

their reliabilities. The approach allows modeling explicitly fault tolerance mechanisms and evaluating their impact on the system reliability (see Section 3.2.1.3).

2.6 Related Work

The RMPI approach in this dissertation belongs to the field of component-based software reliability modeling and prediction (for surveys, see [3–5]). The covered related work in this section mainly belongs to the field but further reliability modeling approaches for individual fault tolerance mechanisms are also mentioned.

2.6.1 Error Propagation Modeling

One of the factors that make the RMPI approach unique is the way how the approach considers error propagation for multiple execution models, including sequential, parallel, and fault tolerance execution models. Although error propagation is an important element in the chain that leads to system failure, many approaches [8, 9, 11, 12, 16, 17, 23–25, 64, 72, 73] 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 [13]. However, some approaches [13, 14, 63, 74] have made a steps towards a consideration of error propagation. A closer investigation of those approaches reveals that they consider it only for the sequential execution model, and therefore, they do not match the consideration of error propagation for multiple execution models as done by the RMPI approach (see Chapter 3). In the following, existing approaches are surveyed with regard to the consideration of error propagation.

Cheung's approach [8], one of the first approaches, expresses the control flow between components in a software system using an absorbing DTMC. Some recent approaches extend Cheung's approach to support different architectural styles [15] and to combine reliability analysis and performance analysis [10] but do not consider error propagation. Further approaches building upon the Cheung's model such as the approach of Lipton et al. [73] which takes interface failures and network connection failures into account, the approach of Sharma et al. [18] which supports modeling component restarts and retries, also do not consider error propagation.

The approach of Reussner et al. [9] 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. [12] 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. [11] 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. [23] 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 [62] and Zheng et al. [64] 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.

Grassi et al. [72] reuse a number of concepts of the approach of Grassi [62] and propose the Kernal Language for Performance and Reliability Analysis (KLAPER). The aim of KLAPER is to capture the relevant information for the analysis of non-functional attributes (e.g. performance and reliability) of component-based systems. Then, an analysis model (e.g. queuing networks, Petri nets, Markov models, or PRISM models [75]) can be generated based on the information expressed in the language. However, error propagation characteristics have not been considered in KLAPER.

Scenario-based approaches such as the approach of Yacoub et al. [24] which constructs component dependency graphs from component sequence diagrams as a basic for reliability prediction, the approaches of Cortellessa et al. [16] and Goseva et al. [17] which employ UML diagrams annotated with reliability properties, the approach of Rodrigues et al. [25] which is based on message sequence charts, also do not consider error propagation.

The approaches [15, 18, 62, 72, 76] that support modeling software fault tolerance mechanisms (see also Section 2.6.2) also assume that any error arising in a component immediately manifests itself as a system failure if there is no fault tolerance mechanism in the system. Otherwise, they assume that any error arising in a component always propagates (i.e. with probability 1.0 and with the same failure type) until fault tolerance mechanisms get involved to provide error handling.

Some approaches have proposed taking error propagation between components into account. The approach of Popic et al. [14] 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. This assumption of immediate failure seems to conflict with the reason of error propagation to other components [13]. The approach of Cortellessa et al. [13] 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. [63] and Mohamed et al. [74] 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.

2.6.2 Software Fault Tolerance Mechanisms Modeling

Software fault tolerance mechanisms are commonly included in software systems (see Section 2.5) and constitute an important means to improve reliability. Therefore, the RMPI approach takes into consideration explicitly the capabilities of these mechanisms (see Section 3.2.1.3). The approach offers enhanced fault tolerance expressiveness, explicitly and flexibly modeling how both error detection and error handling of fault tolerance mechanisms influence the control and data flow within components. As a result, the approach allows modeling comprehensively different classes of fault tolerance mechanisms and evaluating their impact on the system reliability in the dependence of the whole system architecture and usage profile.

In contrast, many approaches [8, 9, 11, 12, 16, 17, 23–25, 64, 73] do not support modeling software fault tolerance mechanisms. The approaches [13, 14, 63, 74] 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, fault tolerance mechanisms with their error detection and error handling cannot be considered explicitly by these approaches.

Some approaches step forward and take fault tolerance mechanisms into account. The approach of Sharma et al. [18] 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 [62] 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 fault tolerance mechanisms on the control and data flow within components. The approach of Brosch et al. [76] extends Recovery Blocks to flexibly describe error handling of fault tolerance mechanisms but still does not consider the influences of error detection of fault tolerance mechanisms 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 fault tolerance mechanism can always handle the error if it aims to handle errors of that failure type. This means that the fault tolerance mechanism 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 fault tolerance mechanism 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 fault tolerance mechanism. Ignoring the influences of either error detection or error handling of fault tolerance mechanisms on the control and data flow within components can lead to incorrect prediction results when the behaviors of fault tolerance mechanisms deviate from the specific cases mentioned by the authors.

A great deal of past research effort focuses on reliability modeling of individual fault tolerance mechanisms. Dugan et al. [20] aim at a combined consideration of hardware and software failures for Distributed Recovery Blocks, N-version Programming, and N Self-checking Programming through fault tree techniques and Markov processes. Kanoun et al. [19] evaluate Recovery Blocks and N-version Programming using generalized stochastic Petri nets. Gokhale et al. [21] use simulation instead of analysis to evaluate Distributed Recovery Blocks, N-version Programming, and N Self-checking Programming. 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 fault tolerance mechanisms, 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.

2.6.3 Concurrently Present Errors Modeling

To the best of our knowledge, the RMPI approach is the first work to support modeling concurrently present errors (see Chapter 3), tending to obtain accurate prediction results. All existing approaches in the field support only a single error at any time, even though situations involving multiple failures are frequently encountered [6]. Neglecting

concurrently present errors can lead to inaccurate prediction results because there exist system failures that cannot be covered by existing approaches [22].

2.6.4 Further Modeling and Prediction Approaches

Besides the approaches discussed so far, several other works aim at system reliability or availability prediction but are different in their goals and scope from the RMPI approach [67, 77, 78].

Bernardi et al. [67] present the MARTE-DAM profile offering a comprehensive dependability modeling. The main focus of this work is on modeling rather than prediction. The authors demonstrate a transformation from the design model of a case study to a deterministic and stochastic Petri net and conduct availability prediction for the case study. However, they do not propose a transformation and prediction method for the general case.

Kharboutly et al. [78] has proposed an approach to analyze the reliability of concurrent component-based software systems using Stochastic Reward Nets as a variation of the stochastic-Petri-net formalism. They do not consider error propagation, software fault tolerance mechanisms, and concurrently present errors. Moreover, the approach targeted at reliability evaluation as a time-dependent probability that the considered software system "survives" from a defined start t_0 up to a point in time t without visiting any failure states, which is different from the goal of the RMPI approach to predict the probability of successful service execution at an arbitrary point in time.

The ABAS (Attribute Based Architecture Styles) approach [77] provides architecture styles or parterns with the modeling supports for the analysis of particular quality attributes (e.g. reliability, availability). However, currently, the only one style focusing on software reliability is the Simplex ABAS. The Simplex ABAS addresses the problem of how to take advantage of redundancy to increase reliability, and introduces the concepts of redundant components, acceptance tests and a decision and switch unit. Hence, the approach suffers from the same limitations as of the reliability modeling approaches for individual fault tolerance mechanisms (see Section 2.6.2).

2.6.5 The RMPI Approach and the Field of Component-based Software Reliability Modeling and Prediction

Table 2.1 summaries most related approaches with regard to the three gaps identified above. A hyphen mark means that an approach does not support the feature and a

TABLE 2.1: Most Related Approaches.

Authors	Year	Error propagation	Software fault tolerance mechanisms	Concurrently present errors
Grassi [62]	2004	-	✓	-
Popic et al. [14]	2005	✓	-	-
Wang et al. [15]	2006	-	✓	-
Sharma et al. [18]	2006	-	✓	-
Grassi et al. [72]	2007	-	✓	-
Cortellessa et al. [13]	2007	✓	-	-
Mohamed et al. [74]	2008	✓	-	-
Filieri et al. [63]	2010	✓	-	-
Brosch et al. [76]	2011	-	✓	-
Pham et al. [This dissertation]	2014	✓	✓	✓

check mark means that an approach supports the feature. Error propagation are supported by some approaches but they introduce new assumptions which deserve further investigation about their soundness, and/or consider error propagation only for a single sequential execution model. None of these approaches supports a consideration of error propagation for multiple execution models, including sequential, parallel, and fault tolerance execution models. Some approaches support modeling software fault tolerance mechanisms but they lack flexible and explicit expressiveness of how error detection and error handling of fault tolerance mechanisms influence the control and data flow within components. Concurrently present errors are not supported by any approaches.

While the RMPI approach in this dissertation receives benefits from the experiences gained in the field of component-based software reliability modeling and prediction by these approaches, it also presents unique features that enhance the state of the art, including (1) a consideration of error propagation multiple execution models, (2) a consideration of software fault tolerance mechanisms with explicit and flexible expressiveness of how both error detection and error handling of fault tolerance mechanisms influence

the control and data flow within components, and (3) a consideration of concurrently present errors.

Remark Related approaches in the field have presented other kinds of contributions. To this end, some approaches investigate alternative modeling formalisms such as Bayesian networks (e.g. [16]) or Markov reward models (e.g. [23]), focus on service-oriented architectures (e.g. [62, 64]), conduct reliability optimization (e.g. [63, 73]), offer combined predictions of multiple quality attributes (e.g. [10, 18], provide closed-form-formula considerations of input uncertainties and the corresponding sensitivity of analysis results (e.g. [66]), supply combined considerations of software and hardware failures (e.g. [12, 18, 62, 72]), and propose parameterized reliability prediction (e.g. [12, 62, 72]). In future work, the RMPI approach may receive benefits from adopting these contributions and integrating them with its existing achievements.

2.7 Summary

This chapter has presented the basics on which the RMPI builds. The discussion covers basic concepts related to software reliability, the overall context of reliability engineering and software reliability engineering, as well as the related field of component-based software reliability modeling and prediction. Further areas of discussion include existing methods for estimating software failure possibilities, an overview of software fault tolerance mechanisms, and a short introduction to Markov chains, on which the RMPI approach and many approaches in the field of component-based software reliability modeling and prediction build. Finally, this chapter reviewed existing approaches in the field of component-based software reliability modeling and prediction, pointing out that they provide limited support for modeling error propagation, fault tolerance mechanisms, and concurrently present errors.

Chapter 3

Methodology and Reliability Modeling

While the previous chapter has introduced the basics on which the RMPI approach builds, the individually discussed aspects are not yet connected and cannot be used directly for reliability modeling, prediction, and supporting design decisions for reliability improvements. This chapter combines the discussed aspects and fills the remaining gaps in order to explain the methodology of the RMPI approach and its reliability modeling capacities. Section 3.1 introduces the steps of the approach and explains the involved developer roles. Section 3.2 describes in detail modeling reliability of component-based software architectures using the reliability modeling schema of the approach. Section 3.3 concludes the chapter with a brief description of the implementation of the schema.

3.1 RMPI Methodology

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment [79]. 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

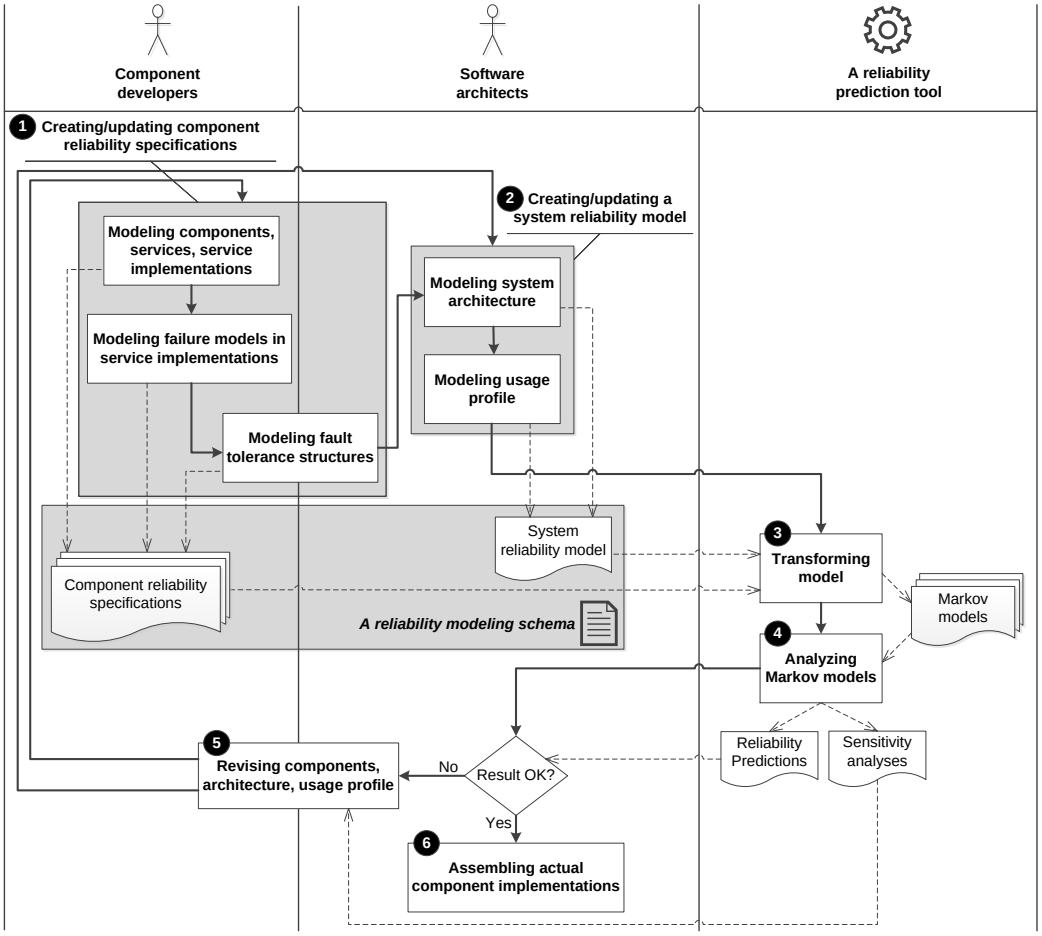


FIGURE 3.1: Reliability engineering process (modeling, prediction, and improvement).

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 3.2, a reliability modeling schema is introduced that supports component developers to create component reliability specifications and software architects to create system reliability models.

The RMPI approach in this dissertation follows repetitively six steps as depicted in Fig. 3.1. In Step 1, component developers create component reliability specifications.

A component reliability specification includes reliability-related properties (e.g. failure probabilities) and call propagations to required services for each provided service of the component. How to determine these probabilities (e.g. using methods described in Section 2.2) is beyond the scope of this dissertation. For already implemented components, call propagations can be derived from static code analysis (e.g. [80]) or dynamic monitoring (e.g. [81]).

More concretely, component developers model components, services and service implementations, and then failure models (i.e. different failure types with their occurrence probabilities) in service implementations. Component developers/software architects can include different fault tolerance structures, e.g. *RetryStructures*, *MultiTryCatchStructures*, or *MVPStructures* (see Section 3.2.1.3), either directly into service implementations of already modeled components or as additional components. Fault tolerance structures support different configurations, e.g. the number of times to retry in a *RetryStructure*, the number of replicated instances for handling certain failure types in a *MultiTryCatchStructure*, or the number of versions executed in parallel in a *MVPStructure*.

In Step 2, software architects create a system reliability model 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 failure probabilities of the components inside the system architecture. To support Step 3 and Step 4, the RMPI approach provides a reliability prediction tool whose transformation for reliability prediction is described in detail in Chapter 4. With the tool support, sensitivity analyses can also be derived, e.g. by varying reliability-related probabilities of components inside the system architecture to obtain corresponding reliability predictions.

If the 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 of fault tolerance structures; 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.

Remark In their taxonomy, Avizienis et al. [6] define a *service failure* of a system or component service as the transition from correct service to incorrect service, i.e. to be not in accordance with the expectation of the service users. They also state that an error becomes a failure when reaching the external state of a component or a system, where "external" means "perceivable at the service interface". However, a user of the system or component service does not necessarily perceive a failure from this definition. If there is no invocation of the service, the failure may be entirely unrecognized. From the other viewpoint, if the transition from correct service to incorrect service is permanent and there are multiple invocations of the service, the failure may be perceived multiple times. In his foundational approach in the field of component-based software reliability modeling and prediction, Cheung [8] states that "A failure is said to occur if, given the input values and specifications of the computations to be performed by the program, the output values are either incorrect or indefinitely delayed.". Here, Cheung focuses on the user-perceived effects of a transition from correct service to incorrect service, instead of the transition itself. Therefore, the RMPI approach, as well as many related approaches (e.g. [12, 13, 16, 63]), distinguishes between a *service failure* and a *failure on demand*, where the latter denotes user-perceived effects of the transition in terms of an undesired service invocation result, including delivery of incorrect outputs, mistimed delivery of outputs, or infinitely delayed processing.

3.2 Reliability Modeling

This section describes the reliability modeling schema of the RMPI approach 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 the RMPI approach upon UML. However, by introducing the reliability modeling schema, the approach avoids 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, the schema is more suitable than UML

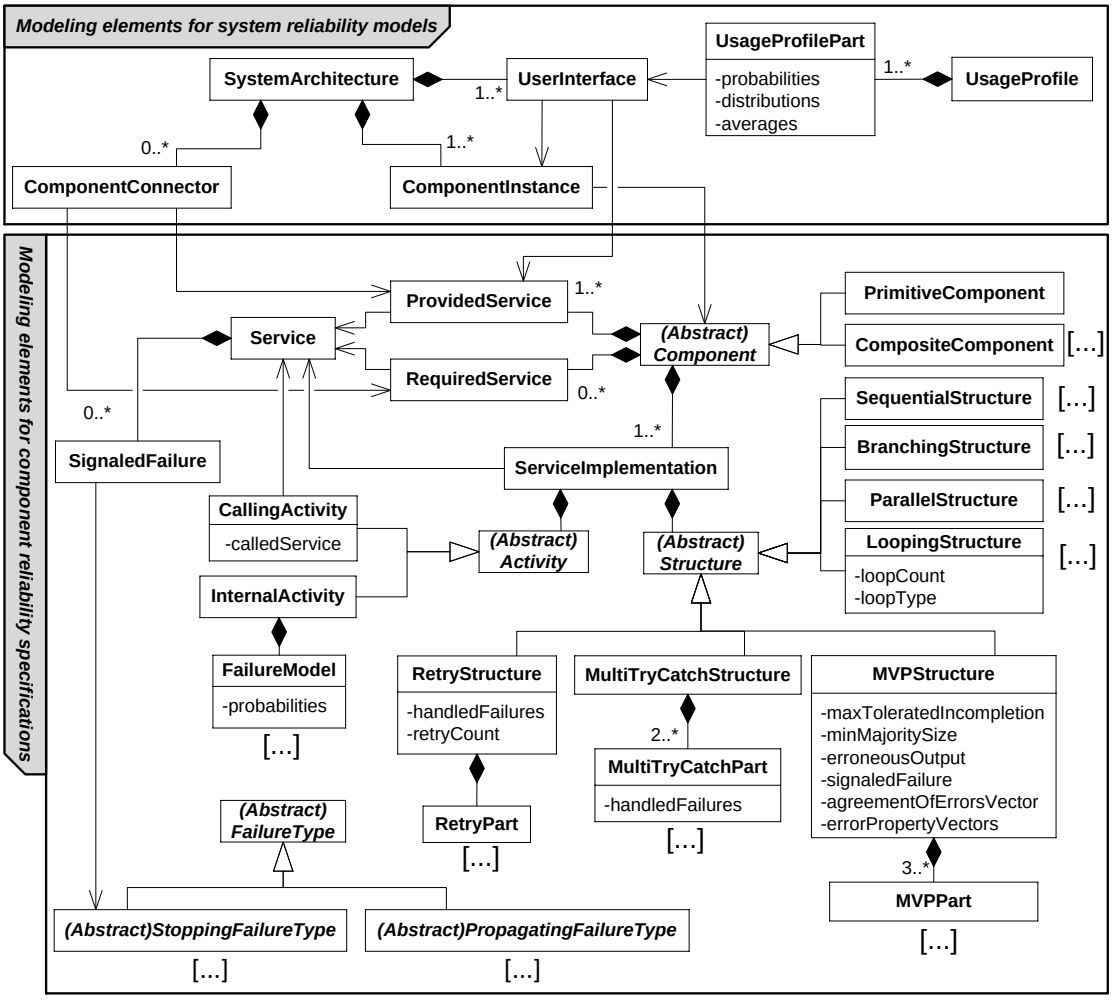


FIGURE 3.2: Modeling elements in the reliability modeling schema.

extended with MARTE-DAM profile[1] [67] because the schema is reduced to concepts needed for reliability prediction, and therefore the approach can support an automated transformation for reliability prediction for the general case.

3.2.1 Component Reliability Specifications

3.2.1.1 Services, Components and Service Implementations

In the RMPI approach, component developers are required to provide component reliability specifications. Fig. 3.2 shows an extract of the reliability modeling schema[2] with modeling elements which supports component developers to create component reliability specifications. Component developers model components and services via modeling elements: *Component* and *Service*, respectively. A component can be either a primitive

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

[2]For a full documentation, refer to our project website [82]

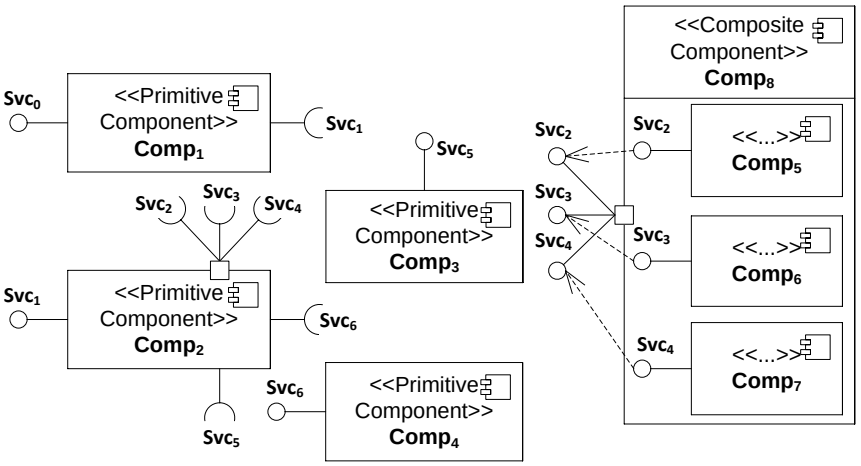


FIGURE 3.3: Example of components and services.

component (*PrimitiveComponent*) or a composite component (*CompositeComponent*) which is hierarchically structured with nested inner components. Components are associated with services via *RequiredService* and *ProvidedService*.

Example 3.1. *Fig. 3.3 shows an example of components and services, including seven services (from Svc_0 to Svc_6), one composite component ($Comp_8$) which contains three nested primitive component ($Comp_5$, $Comp_6$, and $Comp_7$), and four separated primitive components (from $Comp_1$ to $Comp_4$).*

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.

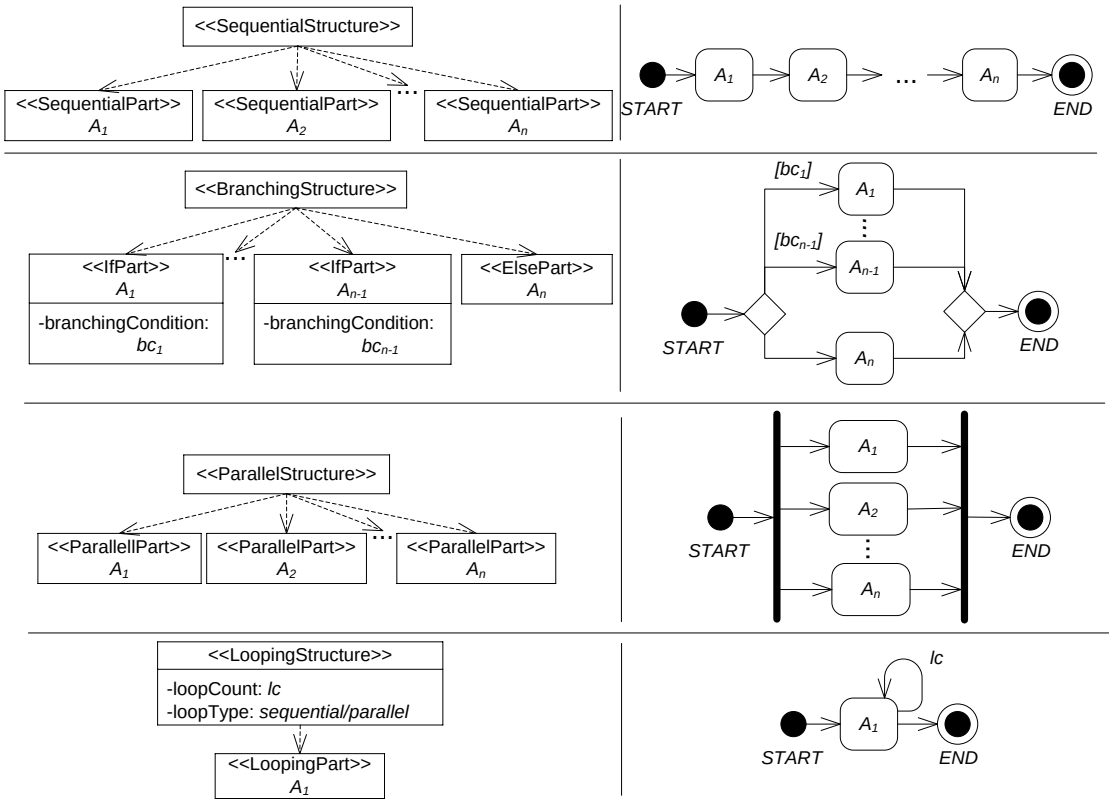


FIGURE 3.4: Supported control flow structures and their execution semantics: Sequential structure, branching structure, parallel structure, and looping structure.

There are four standard types of control flow structures supported by the reliability modeling schema, including sequential structures, branching structures, parallel structures, and looping structures (Fig. 3.4).

- 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.

- 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).

- 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. 3.4, 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.

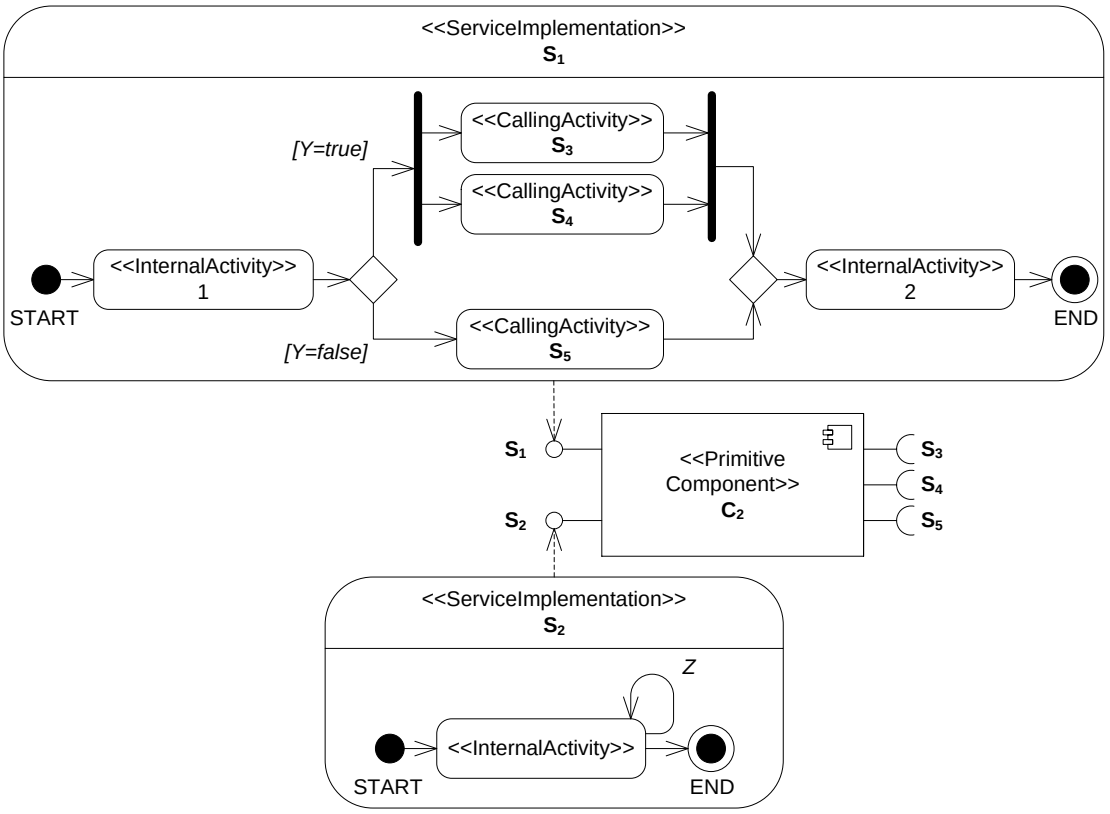


FIGURE 3.5: An example of service implementations.

- In a looping structure[3] (*LoopingStructure*), there is a single looping part (*LoopingPart*) which is repeated the loop count times. Infinite loop count is not allowed. Looping structures can include other looping structures but cannot have multiple entry points and cannot be interconnected. Furthermore, two types of looping structures are supported: sequential and parallel looping structures. In a sequential looping structure, the first iteration of *LoopingPart* works on the structure's input and the current iteration works on the output of the previous iteration. In a parallel looping structure, the iterations work on the structure's input.

Example 3.2. *Fig. 3.5 shows an example of service implementations. The primitive 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.*

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

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

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. An single internal activity modeled with a failure model (see Section 3.2.1.2) may represent thousands of line of code. This abstraction focuses on the necessary properties for a component-based software reliability prediction (i.e. failure probabilities and call propagations).

3.2.1.2 Failure Models

In order to take into consideration explicitly the whole set of factors mentioned in Section 1.2, 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. 3.2). 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 3.3. *Fig. 3.6 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.*

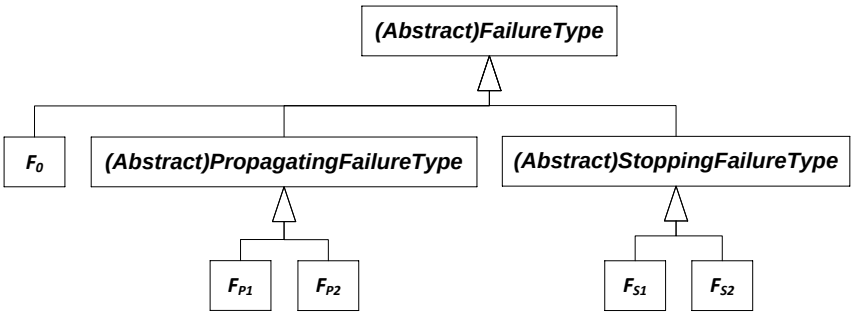


FIGURE 3.6: An example of failure types.

Component developers model a failure model (i.e. different failure types with their occurrence probabilities) for an internal activity via a composition between *InternalActivity* and *FailureModel*. In the literature, techniques for determining these probabilities (e.g. using methods discussed in Section 2.2) have been discussed extensively (also see Section 6.2 for more details) and are beyond the scope of this dissertation.

Definition 3.1. Failure Model[4]

- Let F_0 be a predefined failure type corresponding to the correct service delivery.
- Let $\mathcal{F}_\mathcal{S}$ be the set of all stopping failure types $\{F_{S1}, F_{S2}, ..., F_{Su}\}$.
- Let $\mathcal{F}_\mathcal{P}$ 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 $\{\{F_0\}\} \cup \left(2^{\mathcal{F}_\mathcal{P}} \setminus \emptyset\right)$.
- 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 the internal activity 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 the internal activity has received an input of failure types I. It holds that $\sum_{FO \in (\mathcal{AFS} \cup \mathcal{AIOS})} Pr_{IA}(I, FO) = 1$ for all $I \in \mathcal{AIOS}$.

The failure model of an internal activity can be used as a basis to define interesting reliability properties of the internal activity. Some examples of these properties are proposed as follows:

- **Reliability** is the probability $Pr_{IA}(\{F_0\}, \{F_0\})$.

[4]From here, the introduced symbols are utilized unless otherwise stated.

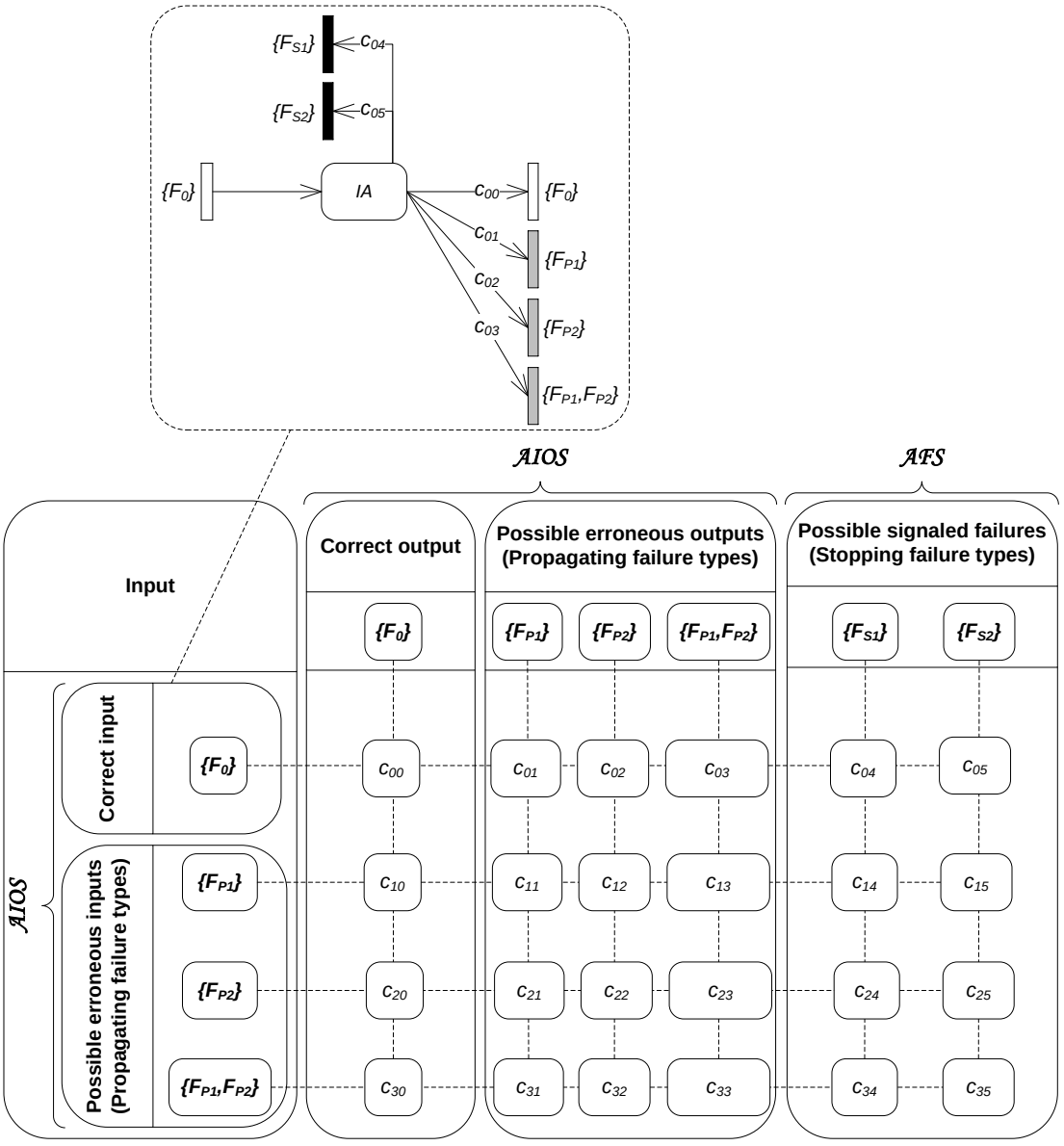


FIGURE 3.7: An example of failure model for an internal activity.

- **Internal failure probability** with respect to failure type FO (with $FO \in \mathcal{AFS} \cup \mathcal{AIOS} \setminus \{\{F_0\}\}$) is the probability $Pr_{IA}(F_0, FO)$.
- **Robustness** with respect to failure type I (with $I \in \mathcal{AIOS} \setminus \{\{F_0\}\}$) is the probability $Pr_{IA}(I, F_0)$.

Example 3.4. *Fig. 3.7 shows an example of failure model for an internal activity with* $\mathcal{F}_\mathcal{S} = \{F_{S1}, F_{S2}\}$, $\mathcal{F}_\mathcal{P} = \{F_{P1}, F_{P2}\}$, $\mathcal{AFS} = \{\{F_{S1}\}, \{F_{S2}\}\}$, *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* $c_{00} = 1 - \sum_{j=1}^{5} c_{0j}$*.*

- *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 [13, 63]. 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 fault tolerance mechanisms to handle it, immediately leads to a system failure.

Remark The RMPI approach supports modeling concurrently present errors via the concurrent presence of propagating failure types. It also allows the approach to support modeling error propagation for parallel structures (see Section 4.1.2.3). Distinguishing between stopping failure types and propagating failure types enables the approach to support modeling error propagation for software fault tolerance mechanisms (see Section 3.2.1.3). With the comprehensive failure model, the 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

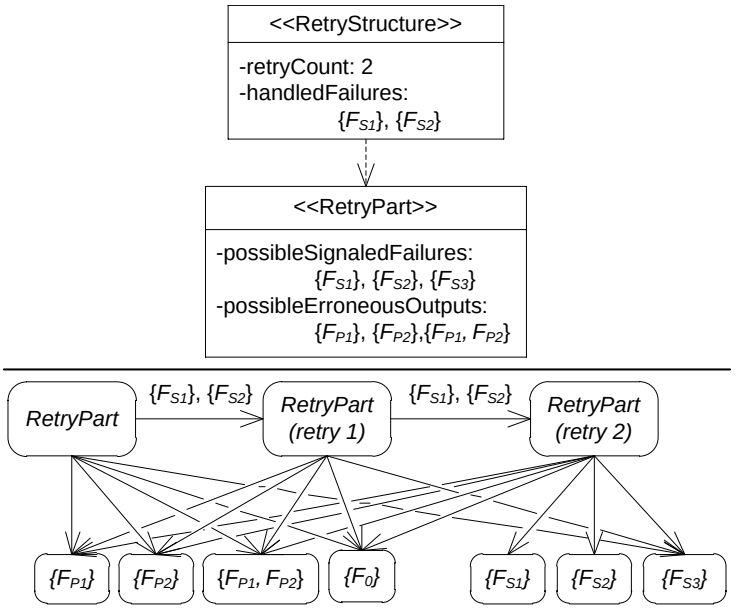


FIGURE 3.8: Semantics for a *RetryStructure* example.

erroneous input, the internal activity signals a signaled failure of an improper stopping failure type).

3.2.1.3 Fault Tolerance Structures

To support modeling fault tolerance mechanisms, the reliability modeling schema provides fault tolerance structures. Because in a fault tolerance mechanisms, error detection is a prerequisite for error handling and not all detected errors can be handled. Therefore, at most, a fault tolerance structure 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 3.5. *Fig. 3.8 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*

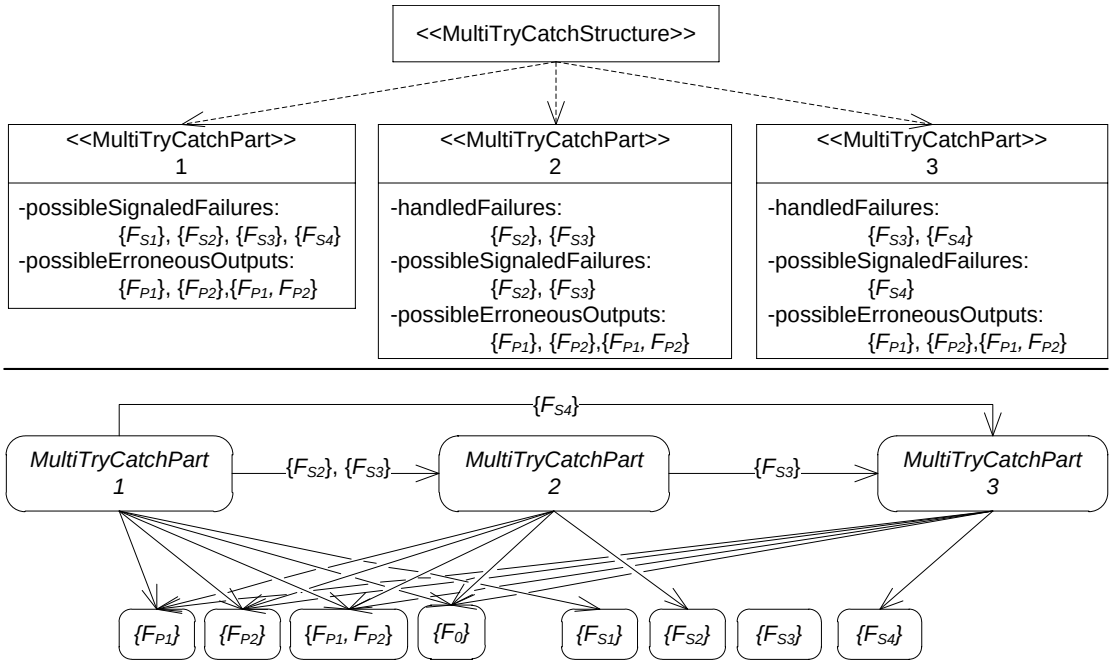


FIGURE 3.9: Semantics for a *MultiTryCatchStructure* example.

field handledFailures*). This means that the structure handles signaled failures of these stopping failure types and retries the* RetryPart*. Signaled failures of* $\{F_{S3}\}$ *cannot 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.*

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 3.6. *Fig. 3.9 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* $\{F_{S1}\}$

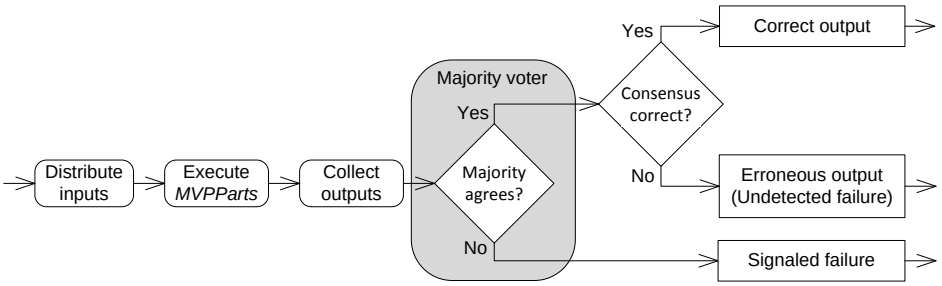


FIGURE 3.10: The operation of a *MVPStructure*.

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. Erroneous 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.*

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

The voter has to determine the decision output from a set of results. If there is no agreement of the majority results, the voter signals a signaled failure. Otherwise, the voter produces an output which is the result of the agreement (i.e. the consensus). The output of the voter is correct if the agreement is of the majority correct results, otherwise the output of the voter is erroneous. In analogy to N-version Programing, it

is assumed that the *MVPStructure* is not used in the situations that can have multiple distinct correct outputs. The operation of a *MVPStructure* is depicted in Fig. 3.10.

From the viewpoint of the majority voter, it distinguishes whether a *MVPPart* completes its execution with an output in time or not (respectively, a complete or incomplete execution, for the sake of simplicity). Therefore, when a *MVPPart* signals a signaled failure of any stopping failure type, the voter considers the *MVPPart*'s execution as an incomplete execution. Given that a *MVPPart* has produced an erroneous output of any propagating failure types, a fraction of the fact that it has not completed its execution in time needs to be provided. With all the possible erroneous outputs of a *MVPPart*, there is a vector of such fractions, called *errorPropertyVector*. For all *MVPParts* of a *MVPStructure*, there is a list of *errorPropertyVectors*.

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

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

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

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

Example 3.7. *Fig. 3.11 shows a* MVPStructure *with three* MVPParts*. After the execution of* MVPPart 1*, signaled failures of stopping failure type* $\{F_{S1}\}$ *or erroneous outputs of propagating failure type* $\{F_{P1}\}$ *can occur. The error property vector for* MVPPart 1 *(the first elements of the field* errorPropertyVectors*) shows that given that* MVPPart 1 *has produced an erroneous output of propagating failure type* F_{P1}*,* MVPPart 1 *has not completed its execution in time with probability* $d_{\{F_{P1}\}}$*. In case erroneous outputs of*

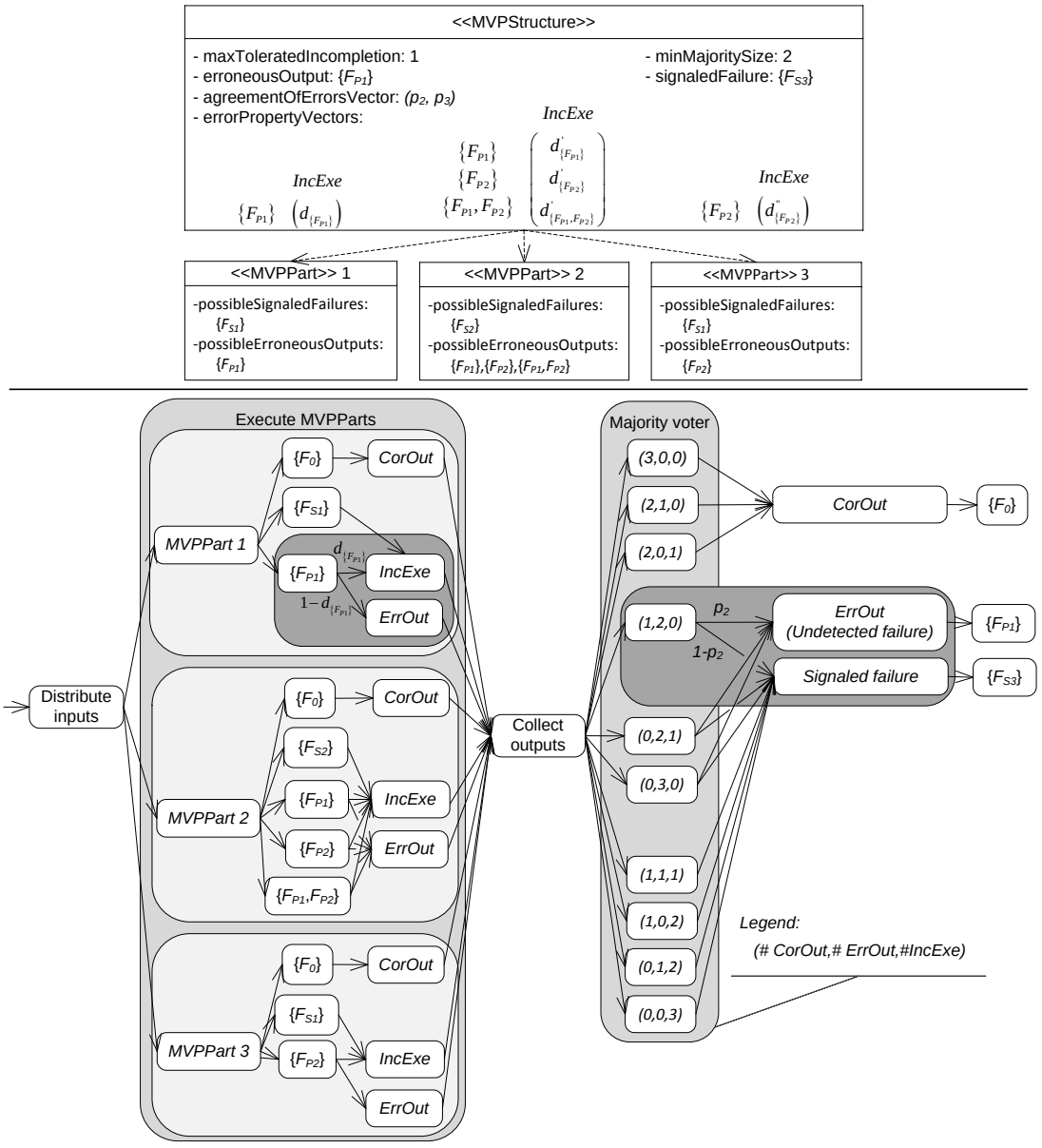


FIGURE 3.11: Semantics for a *MVPStructure* example.

$\{F_{P1}\}$ *are content failures (i.e. the content of a system service's output deviates from the correct one),* $d_{\{F_{P1}\}} = 0$*; in case erroneous outputs of* F_{P1} *are late timing failures (i.e. the delivery time of a system service is too late from the correct one),* $d_{\{F_{P1}\}} = 1$*. Similarly, there is an error property vector for each of the remaining* MVPParts*.*

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

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

- *Possibilities with* $\#IncExe > 1$ *cause the voter to signal a signaled failure of* F_{S3} *(the field* signaledFailure*).*
- *Possibilities with* $\#IncExe \leq 1$ *and* $\#CorOut \geq 2$ *make the voter to produce a correct output (*$\{F_0\}$*).*
- *Possibilities with* $\#IncExe \leq 1$*,* $\#CorOut < 2$*, and* $\#ErrOut < 2$ *also cause the voter to signal a signaled failure of* F_{S3}*.*

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

- *For possibilities with* $\#IncExe \leq 1$*,* $\#CorOut < 2$*, and* $\#ErrOut = 2$*, the voter produces an erroneous output of* F_{P1} *(the field* erroneousOutput*) with probability* p_2*, or signal a signaled failure of* F_{S3} *with probability* $1 - p_2$*.*
- *Similarly, for possibilities with* $\#IncExe \leq 1$*,* $\#CorOut < 2$*, and* $\#ErrOut = 3$*, the voter produces an erroneous output of* F_{P1} *with probability* p_3*, or signal a signaled failure of* F_{S3} *with probability* $1 - p_3$*.*

Remark Fault tolerance structures can be employed in different parts of the system architecture and are quite flexible to model fault tolerance mechanisms because their inner parts (*RetryPart*, *MultiTryCatchParts*, *MVPParts*) are able to contain different activity types, structure types, and even nested fault tolerance structures. 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 fault tolerance mechanisms, including exception handling, restart-retry, primary-backup, recovery blocks, N-version programing, and consensus recovery blocks. If a *RetryPart*,

MultiTryCatchPart, or *MVPPart* 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 Chapter 5 show different possible usages of fault tolerance structures.

3.2.2 System Reliability Models

In RMPI 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. 3.2 shows an extract of the reliability modeling schema with modeling elements for system reliability models.

3.2.2.1 System Architecture

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*).

3.2.2.2 Usage Profile

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 discrete probability distributions (or the average values) of the loop counts of looping structures.

Example 3.8. *Continuing with Example 3.2, Fig. 3.12 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 a distribution (or an average) to determine the branching probabilities of branching

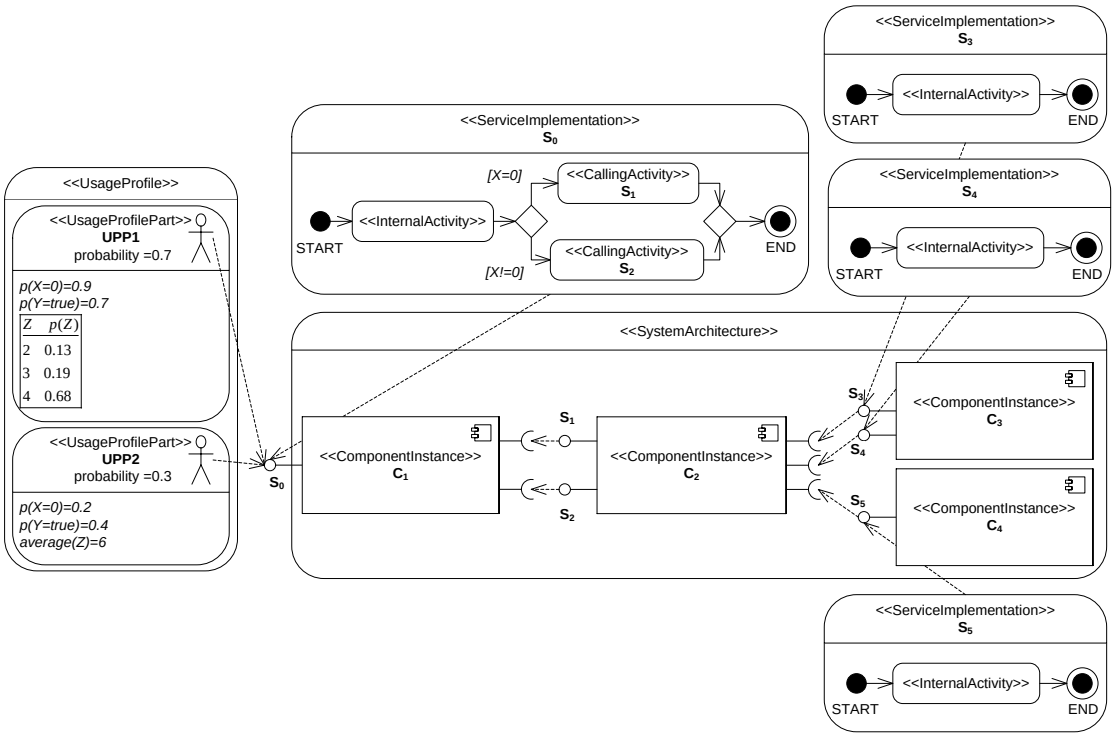


FIGURE 3.12: An example of system reliability model.

structures and the discrete probability distribution (or the average value) of the loop count of the looping structure.

3.2.2.3 System Reliability

Taking into account comprehensively different reliability-influencing factors described above, our goal is to predict the probability of successful execution of the usage scenarios given by the system reliability model. The aimed probability of successful execution under a given usage scenario is the probability that the system completes its execution and produces a correct output, given that the input of the system is correct (namely as defined by its specification). Notice that, under a given usage scenario, the aimed probability is also the direct counterpart of the probability of failure on demand ($POFOD$): $1-POFOD$. Then, the overall system reliability over all the usage scenarios is weighted by the usage scenario probabilities.

The combined consideration of error propagation, software fault tolerance mechanisms, and concurrently present errors enables the reflection of their interplay in the context of the overall system architecture and system usage profile. Only an integrated analysis can provide an accurate view on the relations between them all (see Chapter 4).

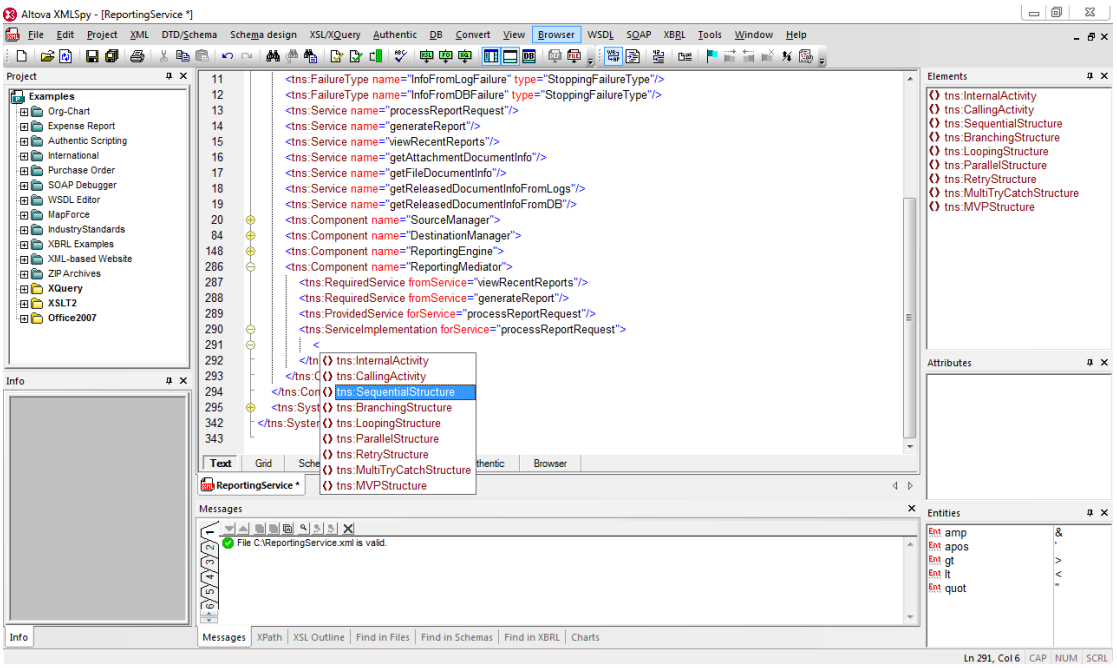


FIGURE 3.13: Reliability modeling environment.

3.3 Implementation

This section shortly describes the implementation of the reliability modeling schema. The implementation is based on the well-known XML Schema Definition (XSD)[83] for expressing a set of predefined semantic constraints (e.g. the total probability of all usage profile parts must be 1) to which system reliability models must conform in order to be considered "valid" according to the schema. Thanks to this industrial base, the user can create system reliability models through XML (eXtensible Markup Language) editors as in Fig 3.13. An analysis method to evaluate the created models in order to obtain reliability predictions is described in Chapter 4.

The implementation of the reliability modeling schema is open source and available at our project website [82].

3.4 Summary

This chapter has developed the methodology of the RMPI approach, based on the basics given in the previous chapter. It then focused on the reliability modeling capabilities of the approach. More concretely, it described how to model component-based software architectures using the reliability modeling schema, including modeling services, components and service implementations, modeling failure models of different failure types with their occurrence probabilities, modeling software fault tolerance mechanisms, as

well as modeling system architectures and usage profiles. Finally, it shortly introduced the implementation of the reliability modeling schema.

Chapter 4

Reliability Prediction and Improvements

After software architects have assembled component reliability specifications to realize the required functionality and specified a usage profile to form a system reliability model, the RMPI approach can predict the reliability for the complete system. This chapter presents in detail the prediction process of the approach (Section 4.1), including an investigation of the complexity of the algorithm, and then describes shortly the implementation of the algorithm for tool support (Section 4.2). Section 4.3 concludes the chapter with an overview of model changes supported by the approach for reliability improvements.

4.1 Reliability Prediction

The prediction process of the RMPI approach starts with the system reliability model and the component reliability specifications, and ends with the system reliability prediction output. The main result of the process is the probability of successful (in other words, failure-free) execution of the given usage scenarios. In analogy to related approaches, the RMPI approach uses discrete-time Markov chains (DTMCs) to represent the system under study and to predict its reliability. DTMCs are a well-established means for the field of component-based software reliability modeling and prediction (see Section 2.3). However, while other approaches use DTMCs to represent software components and the transfer of control between them (see Section 2.4), the RMPI approach additionally reflects the transfer of control and data flow at the intra-component (high-level) and inter-component level, the user behavior, as well as multiple failure types via DTMCs. As a result, this representation allows the RMPI approach to support a more

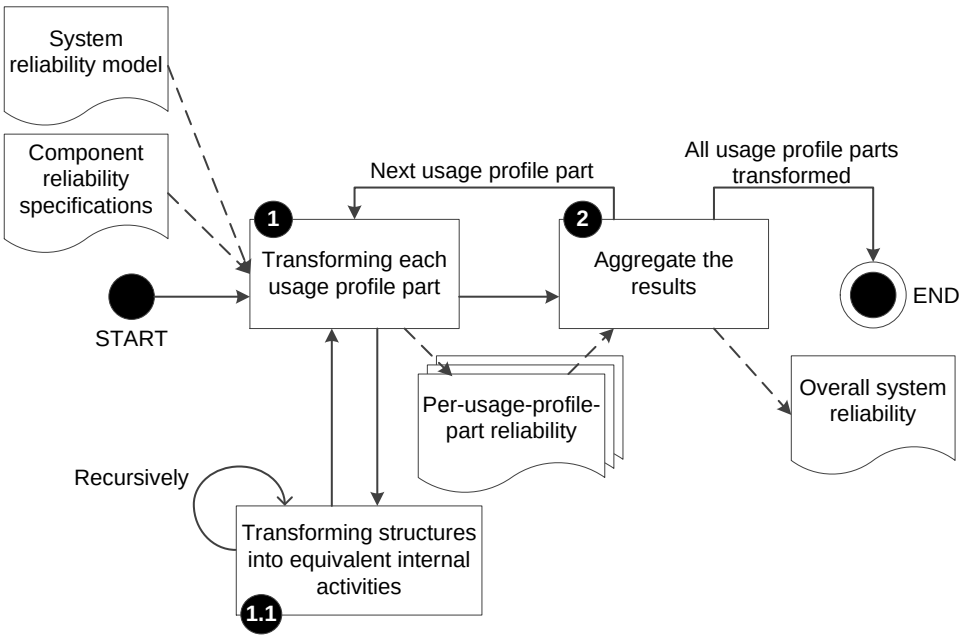


FIGURE 4.1: RMPI prediction process overview.

comprehensive analysis, but it also leads to significantly large DTMCs. Therefore, to ensure the feasibility of the process of creating and solving DTMCs, the RMPI approach takes the following measures:

- Software developers do not need to go through the labored and error-prone process of manually creating DTMCs as stated by a given set of rules. Instead, the RMPI approach automatically transforms the system reliability model into DTMCs.
- The transformation is realized by the RMPI approach through a space-effective transformation algorithm. The algorithm utilizes the specific structural properties of a given system reliability model and produces compact DTMCs as its result.

4.1.1 RMPI Prediction Process Overview

Fig. 4.1 gives an overview of the RMPI prediction process. The process starts with the *System reliability model* and the *Component reliability components* as its input, and finishes with the *Overall system reliability* as its output. As described in Section 3.2.2.2, the usage profile in a system reliability model can contain multiple usage profile parts, and therefore, the process includes the *Transforming each usage profile part* (see Section 4.1.2) to obtain the *Per-usage-profile-part reliability*, and the *Aggregate the results* (see Section 4.1.3) over all usage profile parts of the usage profile.

4.1.2 Transformation for Each Usage Profile Part

The transformation is to derive the reliability for the provided service to which the current usage profile part refers. It starts with the service implementation of this provided service. By design, in the reliability modeling schema of the RMPI approach: (1) a service implementation can contain a structure of any structure type or an activity of any activity type, (2) a structure's inner part (i.e. *SequentialPart*, *IfPart*, *ElsePart*, *ParallelPart*, *LoopingPart*, *RetryPart*, *MultiTryCatchPart*, or *MVPPart*) 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 a structure, the transformation transforms it into an equivalent internal activity (*IA*, for short), and then it uses the equivalent internal activity to transform the outer structure (cf. Fig. 4.1). Details of the transformation for structure types are given in Sections 4.1.2.1, 4.1.2.2, ..., and 4.1.2.7. Section 4.1.2.8 gives details of the transformation to derive the reliability for the provided service to which the current usage profile part refers.

Example 4.1. *Fig. 4.2 shows an example of* Transformation for each usage profile part *for the only usage profile part in the system reliability model at the top of the figure.*

1. *The transformation starts with the service implementation of* Service 1 *and its goal is to obtain the reliability of* Service 1 *under the current usage profile part's information.*

2. *As the service implementation of* Service 2 *is a single internal activity, the calling activity to call* Service 2 *in the service implemetation of* Service 1 *is directly replaced by the internal activity.*

3. *As the service implementation of* Service 3 *is a looping structure executing an internal activity Y times, the looping structure is transformed into an equivalent internal activity, and then the calling activity to call* Service 3 *in the service implemetation of* Service 1 *is replaced by the equivalent internal activity.*

4. *The branching structure in the service implemetation of* Service 1 *is transformed into an equivalent internal activity.*

5. *The sequential structure in the service implemetation of* Service 1 *is transformed into an equivalent internal activity. Finally, the transformation derives the reliability for* Service 1 *from the failure model of the equivalent internal activity.*

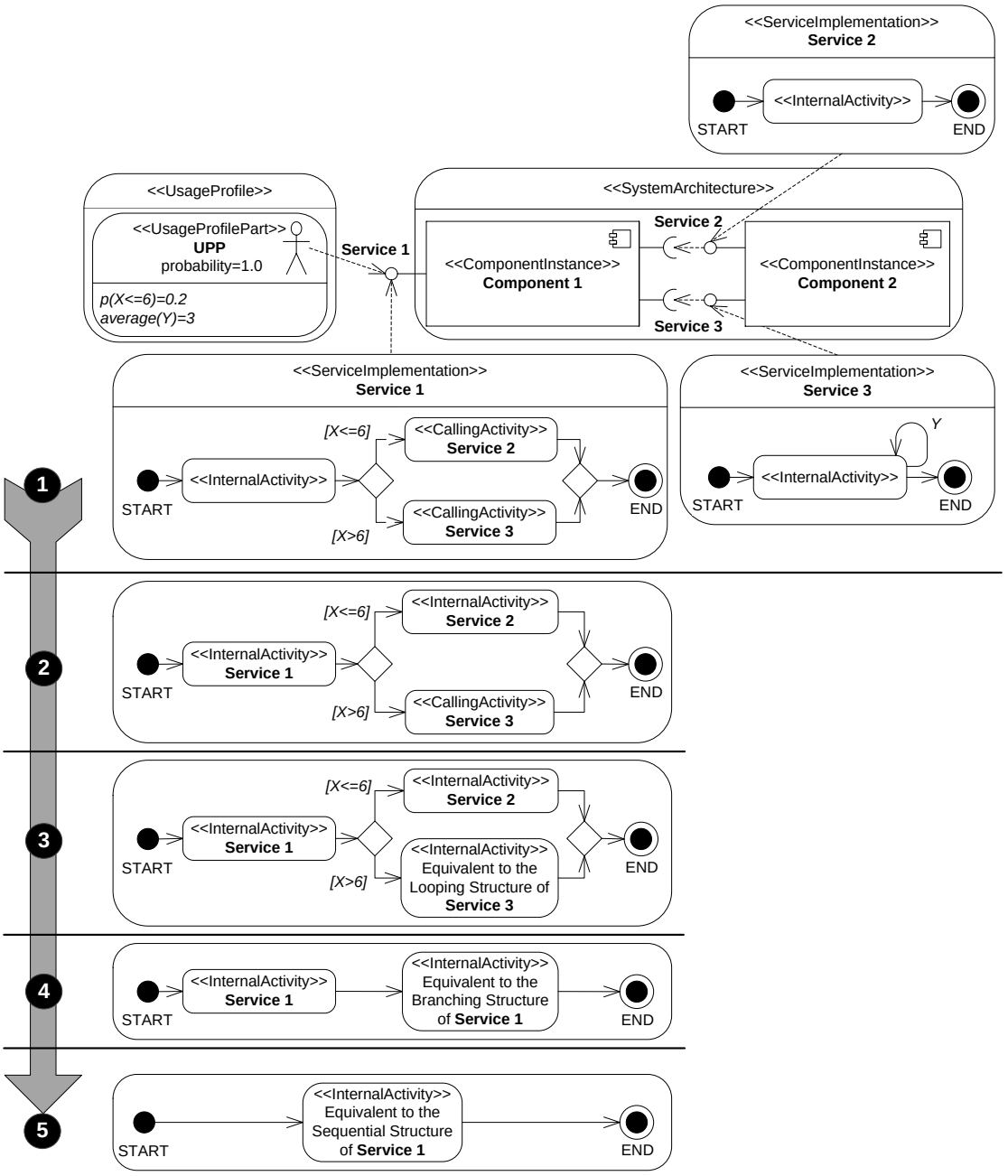


FIGURE 4.2: Example of transformation for each usage profile part.

4.1.2.1 Sequential Structure

Considering a sequential structure with n sequential parts A_1, A_2, ..., A_n, its usage inputs and outputs is shown in Fig. 4.3. The structure's input is the input for sequential part A_1; the output of sequential part A_1 is the input for sequential part A_2; ...; the output of sequential part A_n is the structure's output.

The transformation transforms the sequential structure into an equivalent internal activity in an accumulative manner, that is, it transforms the two sequential parts A_1 and

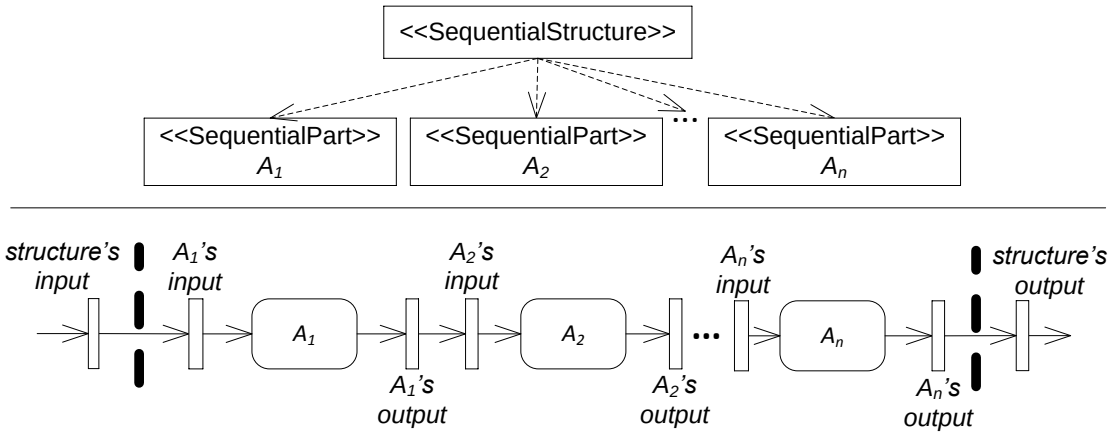


FIGURE 4.3: Using inputs and outputs in a sequential structure.

A_2 into an equivalent internal activity A_{12}, then transforms A_{12} and the sequential part A_3 into an equivalent internal activity A_{123}, and so forth until all the sequential parts of the sequential structure are transformed into an equivalent internal activity $A_{12...n}$.

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 internal activity of the first k (with $k < n$) sequential parts. Fig. 4.4 shows the Markov skeleton that reflects the execution paths for $A_{12...k}$ and A_{k+1} in the sequential structure with a given input $I \in \mathcal{AIOS}$, a given signaled failure $F \in \mathcal{AFS}$, a given (correct or erroneous) output $O' \in \mathcal{AIOS}$, and a given erroneous output $O \in \mathcal{AIOS} \setminus \{\{F_0\}\}$. The complete Markov chain can be obtained by expanding the Markov skeleton for all possible values of I, F, O', and O. The Markov skeleton includes the following elements:

- A state labeled "$I, START$" ($[I, START]$, for short) as the global initial state.
- A Markov block $MB(I, A_{12...k})$ that reflects $A_{12...k}$'s execution paths for a signaled failure F and an (correct or erroneous) output O' . It contains a state $[I, A_{12...k}]$ as the local initial state, a state $[A_{12...k}, F]$ as the state of signaled failure F, and a state $[A_{12...k}, O']$ as the state of output O'. The probability of reaching state $[A_{12...k}, F]$ from state $[I, A_{12...k}]$ is $Pr_{A_{12...k}}(I, F)$, and the probability of reaching state $[A_{12...k}, O']$ from state $[I, A_{12...k}]$ is $Pr_{A_{12...k}}(I, O')$.
- A Markov block $MB(O', A_{k+1})$ that reflects A_{k+1}'s execution paths for a signaled failure F, an erroneous output O, and a correct output $\{F_0\}$. It contains a state $[O', A_{k+1}]$ as the local initial state, a state $[A_{k+1}, F]$ as the state of signaled failure F, a state $[A_{k+1}, O]$ as the state of erroneous output O, and a state $[A_{k+1}, \{F_0\}]$ as the state of correct output $\{F_0\}$. The probability of reaching state $[A_{k+1}, F]$ from state $[O', A_{k+1}]$ is $Pr_{A_{k+1}}(O', F)$, the probability of reaching state $[A_{k+1}, O]$ from

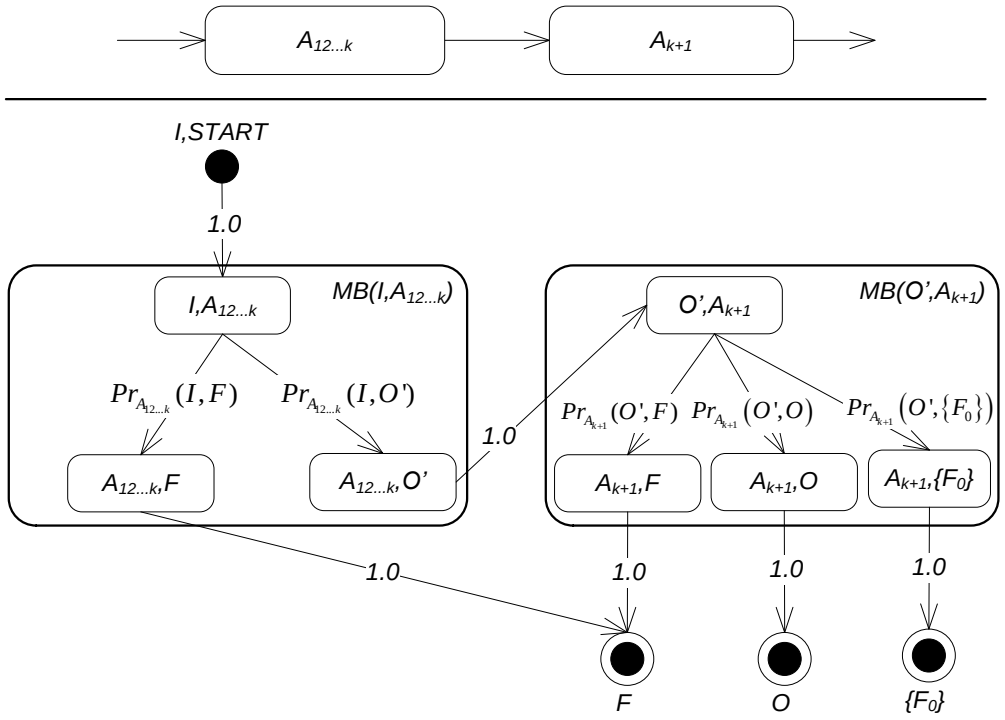


FIGURE 4.4: Markov skeleton for $A_{12...k}$ and A_{k+1} in a sequential structure.

state $[O', A_{k+1}]$ is $Pr_{A_{k+1}}(O', O)$, and the probability of reaching state $[A_{k+1}, \{F_0\}]$ from state $[O', A_{k+1}]$ is $Pr_{A_{k+1}}(O', \{F_0\})$.

- A state $[F]$ as the global state of signaled failure F, a state $[O]$ as the global state of erroneous output O, and a state $[\{F_0\}]$ as the global state of correct output $\{F_0\}$.
- A transition from state $[I, START]$ to state $[I, A_{12...k}]$ with probability 1.0
- A transition from state $[A_{12...k}, O']$ to state $[O', A_{k+1}]$ with probability 1.0.
- A transition from state $[A_{12...k}, F]$ to state $[F]$ with probability 1.0, a transition from state $[A_{k+1}, F]$ to state $[F]$ with probability 1.0, a transition from state $[A_{k+1}, O]$ to state $[O]$ with probability 1.0, and a transition from state $[A_{k+1}, \{F_0\}]$ to state $[\{F_0\}]$ with probability 1.0.

Then, the failure model for the equivalent internal activity of the first $k+1$ sequential parts is calculated 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:

$$Pr_{A_{12\ldots k+1}}(I,\{F_0\}) = \sum_{O'\in\mathcal{AIOS}} Pr_{A_{12\ldots k}}(I,O')\, Pr_{A_{k+1}}(O',\{F_0\}) \tag{4.1}$$

- 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:

$$Pr_{A_{12\ldots k+1}}(I,F) = Pr_{A_{12\ldots k}}(I,F) + \sum_{O'\in\mathcal{AIOS}} Pr_{A_{12\ldots k}}(I,O')\, Pr_{A_{k+1}}(O',F) \tag{4.2}$$

- 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:

$$Pr_{A_{12\ldots k+1}}(I,O) = \sum_{O'\in\mathcal{AIOS}} Pr_{A_{12\ldots k}}(I,O')\, Pr_{A_{k+1}}(O',O) \tag{4.3}$$

By using Equations (4.1), (4.2), and (4.3), the transformation recursively calculates the failure model for the equivalent internal activity of all n sequential parts (i.e. the failure model for the equivalent internal activity of the sequential structure): $Pr_{IA}(I,FO) = Pr_{A_{12\ldots n}}(I,FO)$, $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS}\cup\mathcal{AIOS})$. Notice that the goal of the Markov skeleton as in Fig. 4.4 is just to support our argumentation in building the equations, and the transformation directly uses the equations in its calculation.

4.1.2.2 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, its usage of inputs and outputs is shown in Fig. 4.5. The structure's input is the input for all if parts and else part, and the structure's output is the output of a if part or else part.

Fig. 4.6 shows the Markov skeleton that reflects the execution paths for A_k (with $1 \leq k < n$) and A_n in the branching structure with a given input $I \in \mathcal{AIOS}$, a given signaled failure or (correct or erroneous) output $FO \in (\mathcal{AFS}\cup\mathcal{AIOS})$. The complete Markov chain can be obtained by expanding the Markov skeleton for all possible values of k, I, and FO. The Markov skeleton includes the following elements:

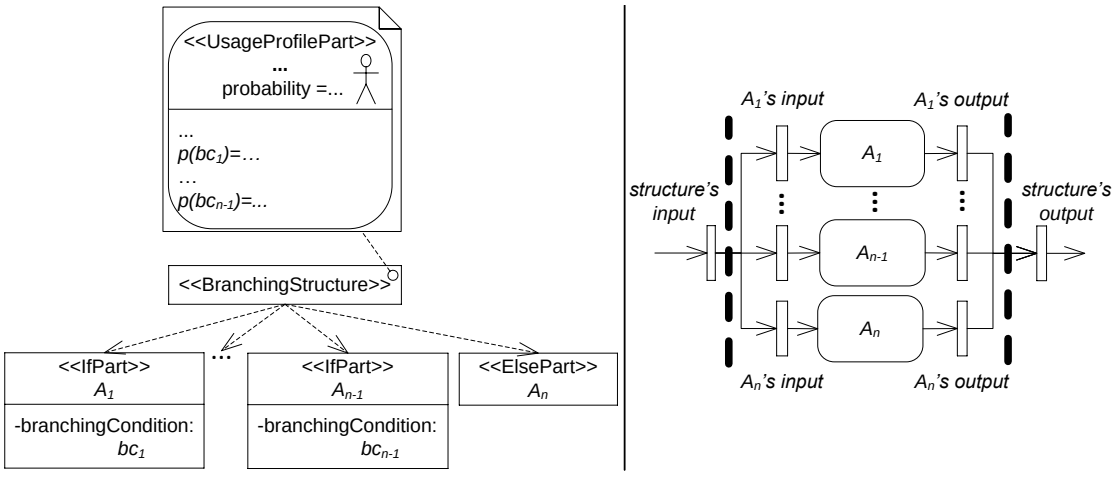


FIGURE 4.5: Using inputs and outputs in a branching structure.

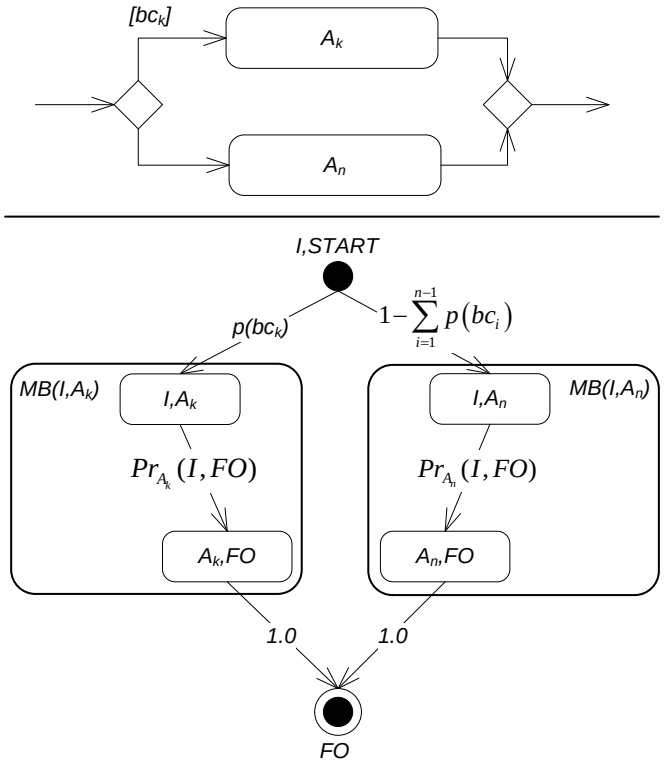


FIGURE 4.6: Markov skeleton for $A_{12...k}$ and A_n in a branching structure.

- A state $[I, START]$ as the global initial state.
- A Markov block $MB\,(I, A_k)$ that reflects A_k's execution paths for a signaled failure or (correct or erroneous) output FO. It contains a state $[I, A_k]$ as the local initial state, a state $[A_k, FO]$ as the state of signaled failure or (correct or erroneous) output FO. The probability of reaching state $[A_k, FO]$ from state $[I, A_k]$ is $Pr_{A_k}(I, FO)$.
- A Markov block $MB\,(I, A_n)$ that reflects A_n's execution paths for a signaled failure or (correct or erroneous) output FO. It contains a state $[I, A_n]$ as the local initial state, a state $[A_n, FO]$ as the state of signaled failure or (correct or erroneous) output FO. The probability of reaching state $[A_n, FO]$ from state $[I, A_n]$ is $Pr_{A_n}(I, FO)$.
- A state $[FO]$ as the global state of signaled failure or (correct or erroneous) output FO.
- A transition from state $[I, START]$ to state $[I, A_n]$ with probability $1 - \sum_{i=1}^{n-1} p\,(bc_i)$ which is the execution probability of A_n, 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 A_i) which is obtained from the current usage profile part.
- A transition from state $[I, START]$ to state $[I, A_k]$ with probability $p\,(bc_k)$.
- A transition from state $[A_k, FO]$ to state $[FO]$ with probability 1.0, and a transition from state $[A_n, FO]$ to state $[FO]$ with probability 1.0.

Then, the equivalent internal activity of the structure has the failure model as follows (with $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$):

$$Pr_{IA}\,(I, FO) = \sum_{i=1}^{n-1} p(bc_i) Pr_{A_i}\,(I, FO) + \left(1 - \sum_{i=1}^{n-1} p(bc_i)\right) Pr_{A_n}\,(I, FO) \tag{4.4}$$

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.

4.1.2.3 Parallel Structure

Considering a parallel structure with n parallel branches A_1, A_2, ..., A_n as in Fig. 4.7, the transformation transforms it into an equivalent internal activity based on the following arguments:

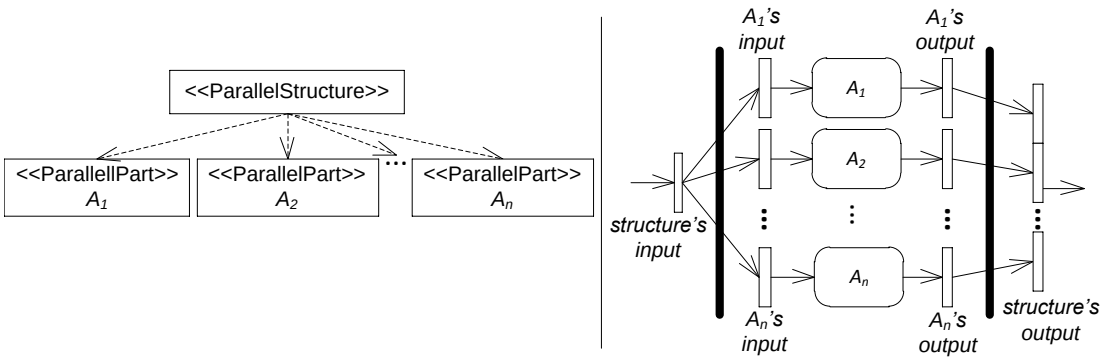


FIGURE 4.7: Using inputs and outputs in a parallel structure.

- The parallel structure (therefore the equivalent internal activity) signals a signaled failure if at least one parallel branch has a signaled failure.
- The parallel structure (therefore the equivalent internal activity) produces a correct output if all parallel branches produce correct outputs.
- The parallel structure (therefore the equivalent internal activity) 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:

- Reliability-related behaviors of parallel branches are independent.[1]
- 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. 4.7 shows a usage[2] 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.

[1] 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 [84], or implicitly included in probabilities of the failure models for internal activities in parallel branches.

[2] 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.

- 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.[3]

Under all the stated assumptions, the transformation transforms the parallel structure into an equivalent internal activity in an accumulative manner, that is, it transforms the two parallel parts A_1 and A_2 into an equivalent internal activity A_{12}, then transforms A_{12} and the parallel part A_3 into an equivalent internal activity A_{123}, and so forth until all the parallel parts of the parallel structure are transformed into an equivalent internal activity $A_{12...n}$.

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 internal activity of the first k (with $k < n$) parallel branches. Fig. 4.8 shows the Markov skeleton that reflects the execution paths for $A_{12...k}$ and A_{k+1} in the parallel structure with a given input $I \in \mathcal{AIOS}$, a given signaled failure $F \in \mathcal{AFS}$, a given (correct or erroneous) output $O' \in \mathcal{AIOS}$, a given erroneous output $O \in \mathcal{AIOS} \setminus \{\{F_0\}\}$, a given erroneous output $O_1 \in \mathcal{AIOS} \setminus \{\{F_0\}\}$, a given erroneous output $O_2 \in \mathcal{AIOS} \setminus \{\{F_0\}\}$, and a given signaled failure or (correct or erroneous) output $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$. The complete Markov chain can be obtained by expanding the Markov skeleton for all possible values of I, F, O', O, O_1, O_2, and FO. The Markov skeleton includes the following elements:

- A state labeled $[I, START]$ as the global initial state.
- A Markov block $MB(I, A_{12...k})$ that reflects the execution paths of $A_{12...k}$ and A_{k+1} for a signaled failure F, an (correct or erroneous) output O', an erroneous output O, an erroneous output O_1, an erroneous output O_2, a signaled failure or (correct or erroneous) output FO, and a correct output $\{F_0\}$. It contains the following states and transitions:
  - A state $[(I, A_{12...k}), (I, A_{k+1})]$ as the local initial state.
  - A state $[(A_{12...k}, \{F_0\}), (A_{k+1}, \{F_0\})]$ as the state that $A_{12...k}$ produces $\{F_0\}$ and A_{k+1} produces $\{F_0\}$. The probability of reaching state $[(A_{12...k}, \{F_0\}),$

[3] It would have been possible for us to support 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, fault tolerance mechanisms, if any, to handle errors of parallel executions are often put inside each parallel execution could make the support useless in modeling fault tolerance mechanisms. 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).

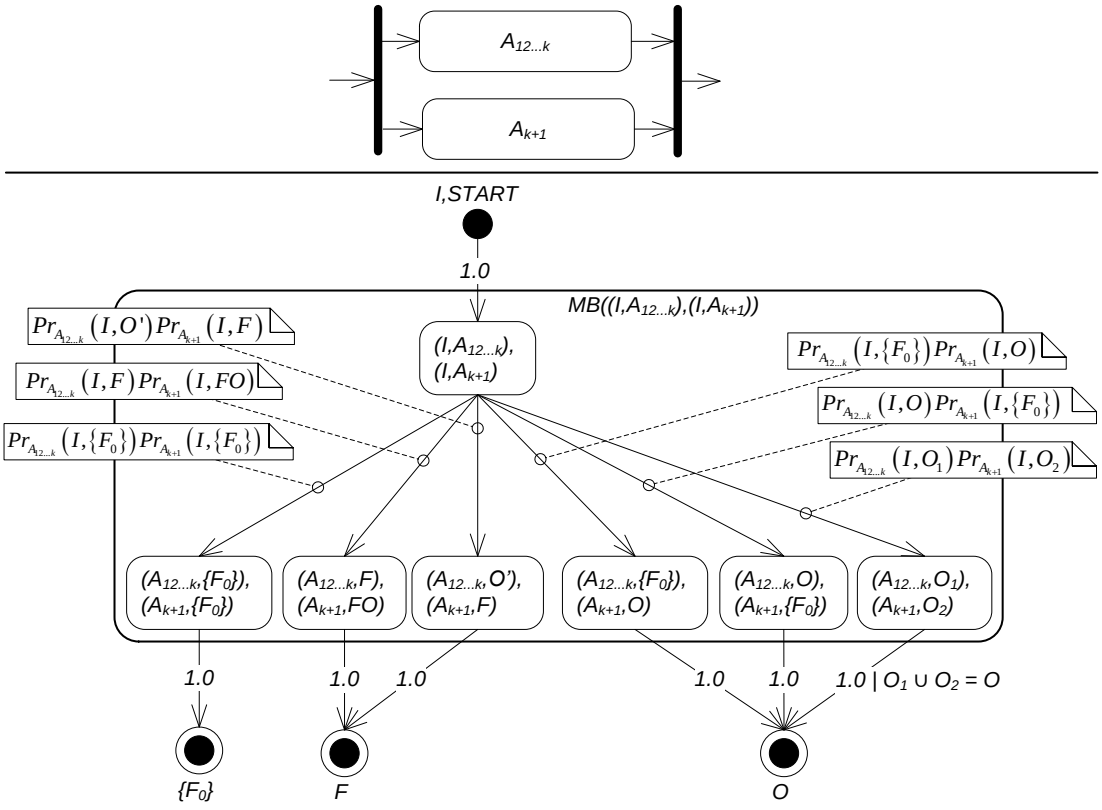


FIGURE 4.8: Markov skeleton for $A_{12...k}$ and A_{k+1} in a parallel structure.

$(A_{k+1}, \{F_0\})]$ from state $[(I, A_{12...k}), (I, A_{k+1})]$ is $Pr_{A_{12...k}}(I, \{F_0\})Pr_{A_{k+1}}(I, \{F_0\})$.

- A state $[(A_{12...k}, F), (A_{k+1}, FO)]$ as the state that $A_{12...k}$ signals F and A_{k+1} signals or produces FO. The probability of reaching state $[(A_{12...k}, F), (A_{k+1}, FO)]$ from state $[(I, A_{12...k}), (I, A_{k+1})]$ is $Pr_{A_{12...k}}(I, F)\, Pr_{A_{k+1}}(I, FO)$.
- A state $[(A_{12...k}, O'), (A_{k+1}, F)]$ as the state that $A_{12...k}$ produces O' and A_{k+1} signals F. The probability of reaching state $[(A_{12...k}, O'), (A_{k+1}, F)]$ from state $[(I, A_{12...k}), (I, A_{k+1})]$ is $Pr_{A_{12...k}}(I, O')Pr_{A_{k+1}}(I, F)$.
- A state $[(A_{12...k}, \{F_0\}), (A_{k+1}, O)]$ as the state that $A_{12...k}$ produces $\{F_0\}$ and A_{k+1} produces O. The probability of reaching state $[(A_{12...k}, \{F_0\}), (A_{k+1}, O)]$ from state $[(I, A_{12...k}), (I, A_{k+1})]$ is $Pr_{A_{12...k}}(I, \{F_0\})Pr_{A_{k+1}}(I, O)$.
- A state $[(A_{12...k}, O), (A_{k+1}, \{F_0\})]$ as the state that $A_{12...k}$ produces O and A_{k+1} produces $\{F_0\}$. The probability of reaching state $[(A_{12...k}, O), (A_{k+1}, \{F_0\})]$ from state $[(I, A_{12...k}), (I, A_{k+1})]$ is $Pr_{A_{12...k}}(I, O)Pr_{A_{k+1}}(I, \{F_0\})$.
- A state $[(A_{12...k}, O_1), (A_{k+1}, O_2)]$ as the state that $A_{12...k}$ produces O_1 and A_{k+1} produces O_2. The probability of reaching state $[(A_{12...k}, O_1), (A_{k+1}, O_2)]$ from state $[(I, A_{12...k}), (I, A_{k+1})]$ is $Pr_{A_{12...k}}(I, O_1)Pr_{A_{k+1}}(I, O_2)$.

- A state $[\{F_0\}]$ as the global state of correct output $\{F_0\}$, a state $[F]$ as the global state of signaled failure F, and a state $[O]$ as the global state of erroneous output O.

- A transition from state $[I, START]$ to state $[(I, A_{12...k}), (I, A_{k+1})]$ with probability 1.0

- A transition from state $[(A_{12...k}, \{F_0\}), (A_{k+1}, \{F_0\})]$ to state $[\{F_0\}]$ with probability 1.0.

- A transition from state $[(A_{12...k}, F), (A_{k+1}, FO)]$ to state $[F]$ with probability 1.0, and a transition from state $[(A_{12...k}, O'\}), (A_{k+1}, F)]$ to state $[F]$ with probability 1.0.

- A transition from state $[(A_{12...k}, \{F_0\}), (A_{k+1}, O)]$ to state $[O]$ with probability 1.0, a transition from state $[(A_{12...k}, \{F_0\}), (A_{k+1}, O)]$ to state $[O]$ with probability 1.0, and a transition from state $[(A_{12...k}, O_1), (A_{k+1}, O_2)]$ to state $[O]$ with probability 1.0 if $O_1 \cup O_2 = O$.

Then, the failure model for the equivalent internal activity of the first $k+1$ parallel branches is calculated 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:

$$Pr_{A_{12...k+1}}(I, \{F_0\}) = Pr_{A_{12...k}}(I, \{F_0\})\, Pr_{A_{k+1}}(I, \{F_0\}) \tag{4.5}$$

- 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:

$$Pr_{A_{12...k+1}}(I, F) = Pr_{A_{12...k}}(I, F) + \left(\sum_{O' \in \mathcal{AIOS}} Pr_{A_{12...k}}(I, O') \right) Pr_{A_{k+1}}(I, F) \tag{4.6}$$

- 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

TABLE 4.1: 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	$Pr_{A_1}(I, O)\, Pr_{A_2}(I, \{F_0\})$
$\{F_0\}$	$\{F_0\}$	$\{F_0\}$	$Pr_{A_1}(I, \{F_0\})\, Pr_{A_2}(I, \{F_0\})$
$O_1 \in (\mathcal{AIOS} \setminus \{\{F_0\}\})$	$O_2 \in (\mathcal{AIOS} \setminus \{\{F_0\}\})$	$O_1 \cup O_2$	$Pr_{A_1}(I, O_1)\, Pr_{A_2}(I, O_2)$

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 $O_1 \cup O_2 = O$:

$$\begin{aligned} Pr_{A_{12\ldots k+1}}(I, O) \;=\; & Pr_{A_{12\ldots k}}(I, \{F_0\})\, Pr_{A_{k+1}}(I, O) \\ +\; & Pr_{A_{12\ldots k}}(I, O)\, Pr_{A_{k+1}}(I, \{F_0\}) \\ +\; & \sum_{\substack{O_1 \cup O_2 = O \\ O_1, O_2 \in \mathcal{AIOS} \setminus \{\{F_0\}\}}} \left(Pr_{A_{12\ldots k}}(I, O_1)\, Pr_{A_{k+1}}(I, O_2)\right) \end{aligned} \tag{4.7}$$

By using Equations (4.5), (4.6), and (4.7), the transformation recursively calculates the failure model for the equivalent internal activity of all n parallel branches (i.e. the failure model for the equivalent internal activity 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 4.2. *Assuming a parallel structure with two parallel branches A_1 and A_2, each parallel branch has a failure model as in Example 3.4. Table 4.1 shows the transformation results. From this table, the transformation can build up the failure model for the equivalent internal activity 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 (4.6)).*

4.1.2.4 Looping Structure

Considering a looping structure with a single looping part A_1, in case the current usage profile part contains the average value of the loop count, i.e. $average(lc) = v_L$ as shown in Fig. 4.9, it can be seen as either a sequential structure with v_L sequential parts A_1 (if it

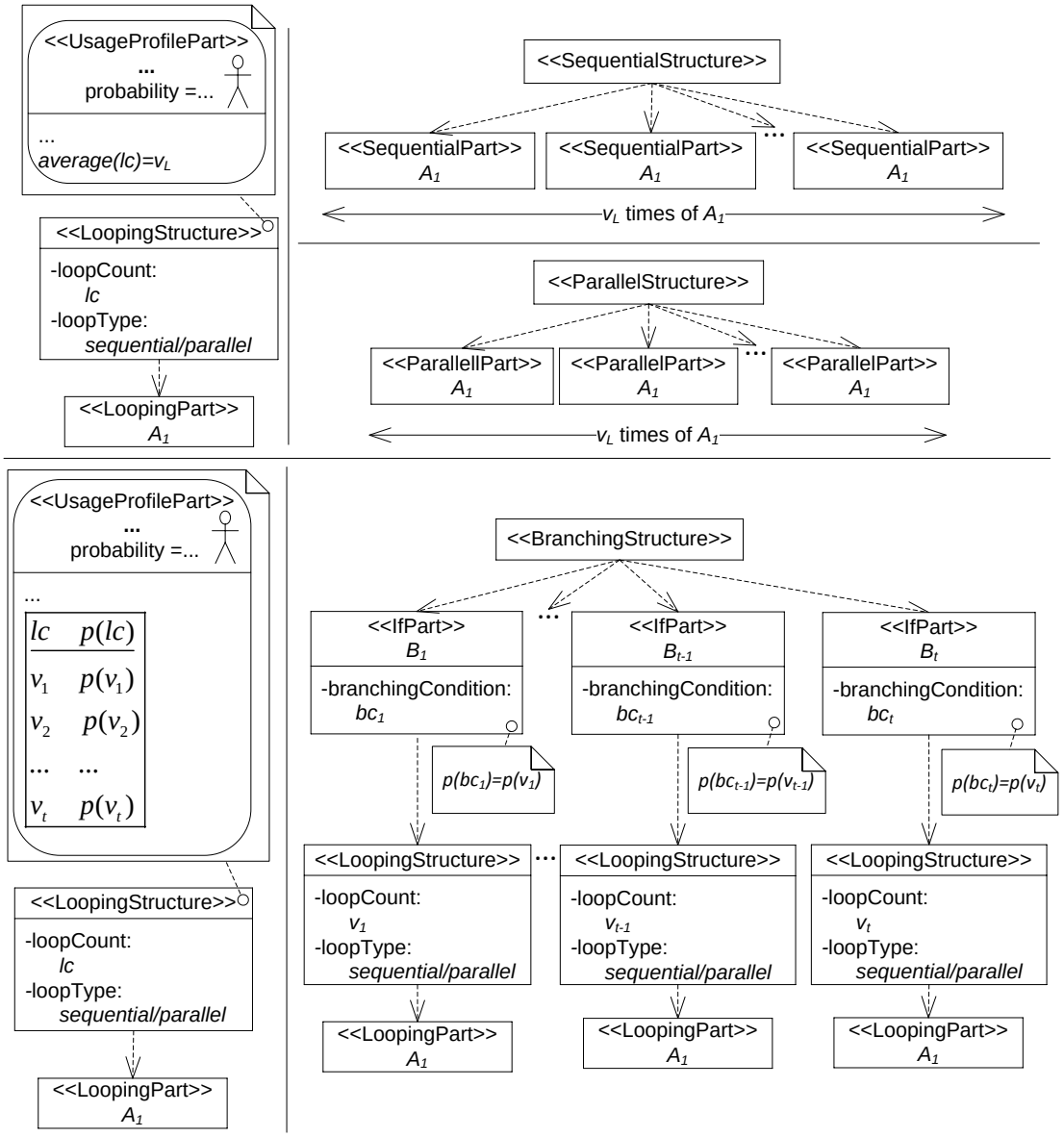


FIGURE 4.9: Looping structures and their equivalent structures.

is a sequential looping structure) or a parallel structure with v_L parallel parts A_1 (if it is a parallel looping structure). Then, the failure model for the equivalent internal activity of the looping structure can be calculated by applying the same transformation, as for a sequential structure (or a parallel structure), to the equivalent structure of the looping structure. Moreover, because all parts of the equivalent structure are the same A_1, the transformation also employs the exponentiation by squaring[4] for fast transforming.

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

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

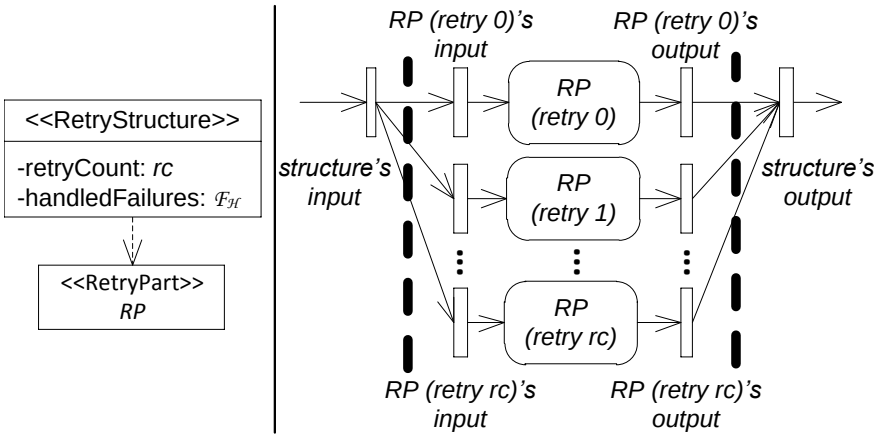


FIGURE 4.10: Using inputs and outputs in a *RetryStructure*.

structure can be seen as a branching structure of if parts as in Fig. 4.9. Each if part B_i, $i \in \{1, 2, ..., t\}$, has its execution probability $p(bc_i) = p(v_i)$ and contains a looping structure with the loop count value v_i. Then, the failure model for the equivalent internal activity of the looping structure can be calculated by applying (1) the same transformation, as in case of an average value of the loop count, to the inner looping structures, and then (2) the same transformation, as for a branching structure, to the outer branching structure.

4.1.2.5 RetryStructure

Considering a *RetryStructure*, let rc be the retry count, $\mathcal{F}_{\mathcal{H}} \subseteq \mathcal{AFS}$ 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. 4.10 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, the first execution of the *RetryPart* is considered 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 internal activity 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. 4.11). It includes a state labeled "I, RP_i" as an initial state, states $[RP_i, F]$ for all

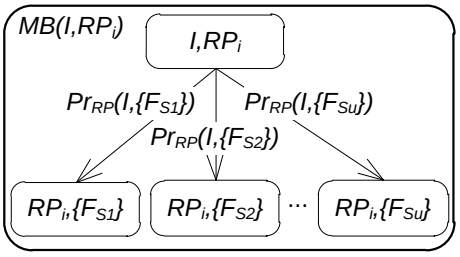


FIGURE 4.11: Markov block for *i-th* retry.

$F \in \mathcal{AFS}$ 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 (following the semantics as illustrated as in Fig. 3.8).

- 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 $MB(I, RP_i)$ with $i \in \{0, 1, ..., rc\}$, add transitions from state $[I, RP_i]$ to state $[O]$ with probability $Pr_{RP}(I, O)$ 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 $MB(I, RP_{rc})$ (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. $MB(I, RP_i)$ 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 $F \in \mathcal{F}_{\mathcal{H}}$, or otherwise to (2) state $[F]$ with probability 1.0 for all $F \in \mathcal{AFS}$, .

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

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

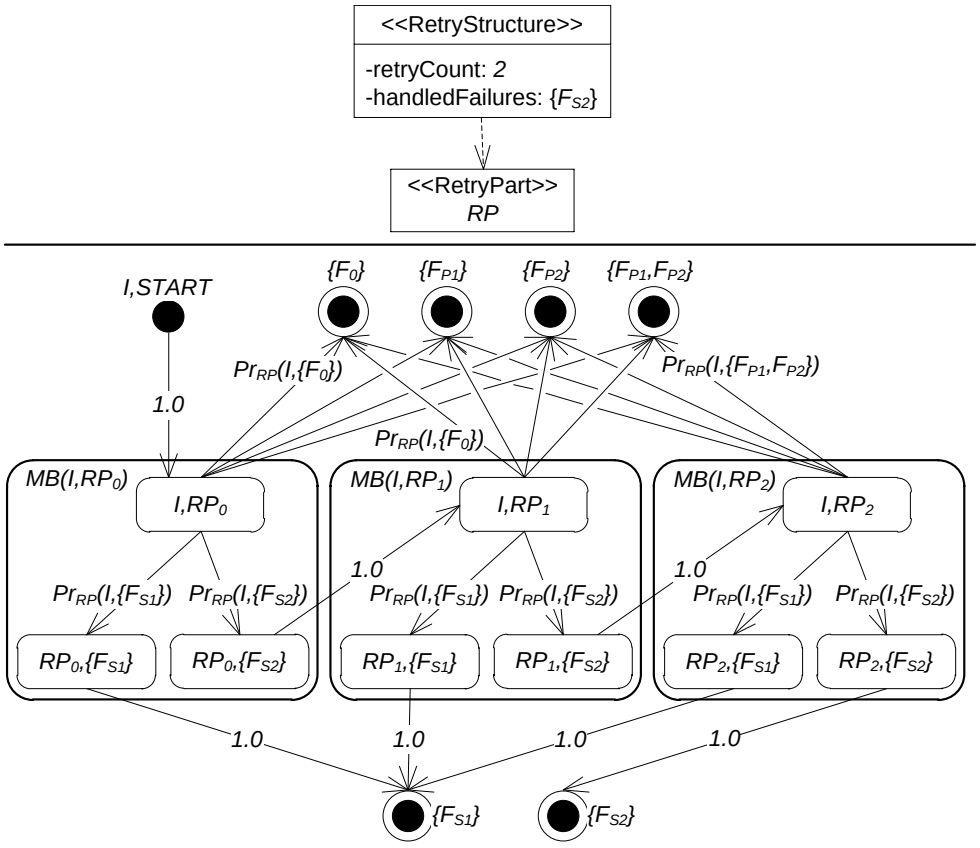


FIGURE 4.12: An example of transformation for a *RetryStructure*.

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 $\mathbf{B} = (\mathbf{I} - \mathbf{Q})^{-1}\mathbf{R}$ 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 [59]. Thus, the failure model of the equivalent *IA* can be obtained from the matrix $\mathbf{B}$.

Example 4.3. *Fig. 4.12 shows an example of transformation for a* RetryStructure *(several transition probabilities are omitted for the sake of clarity). In this example, it is assumed that the* RetryPart *has a failure model as in Example 3.4. Therefore,* $\mathcal{AFS} = \{\{F_{S1}\}, \{F_{S2}\}\}$ *and* $\mathcal{AIOS} = \{\{F_0\}, \{F_{P1}\}, \{F_{P2}\}, \{F_{P1}, F_{P2}\}\}$. *rc of the*

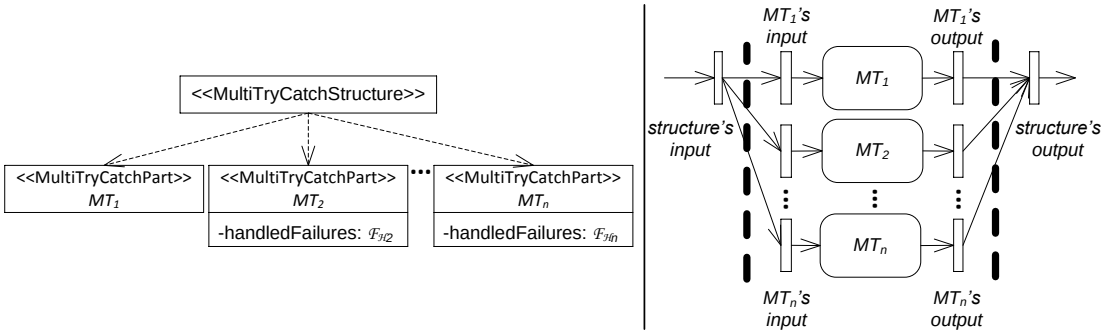


FIGURE 4.13: Using inputs and outputs in a *MultiTryCatchStructure*.

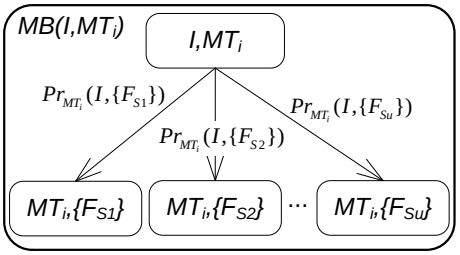


FIGURE 4.14: Markov block for *MultiTryCatchPart i*.

RetryStructure *is 2 and* $\mathcal{F}_{\mathcal{H}}$ *of the* RetryStructure *is* $\{\{F_{S2}\}\}$. *From the resulting Markov model, the failure model for the equivalent internal activity 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* $[I, START]$.

4.1.2.6 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, $Pr_{MT_i}(I, FO)$ for all $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$ be the failure model of *MultiTryCatchPart* i (abbreviated as MT_i).

Fig. 4.13 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 internal activity from this Markov model.

Step 1, the transformation builds a Markov block for each *MultiTryCatchPart*. The Markov Block for the *MultiTryCatchPart* i ($MB\,(I, MT_i)$) reflects its possible execution paths for signaled failures (Fig. 4.14). It includes a state $[I, MT_i]$ as an initial state, states $[MT_i, F]$ for all $F \in \mathcal{AFS}$ as states of signaled failures. The probability of reaching state $[MT_i, F]$ from state $[I, MT_i]$ is $Pr_{MT_i}\,(I, F)$ 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 (following the semantics as illustrated as in Fig. 3.9).

- 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 $[I, MT_1]$ with probability 1.0.
- For all Markov blocks $MB\,(I, MT_i)$ with $i \in \{1, 2, ..., n\}$, add transitions from state $[I, MT_i]$ to state $[O]$ with probability $Pr_{MT_i}\,(I, O)$ 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 $MB\,(I, MT_n)$ (i.e. the Markov block of the last *MultiTryCatchPart*), add transitions from state $[MT_n, F]$ to state $[F]$ with probability 1.0 for all $F \in \mathcal{AFS}$.
- For other Markov blocks, i.e. $MB\,(I, MT_i)$ with $i \in \{1, 2, ..., n-1\}$, add transitions from state $[MT_i, F]$ to (1) state $[I, MT_x]$ 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, with the resulting Markov chain, the failure model for the equivalent internal activity is built up as follows.

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

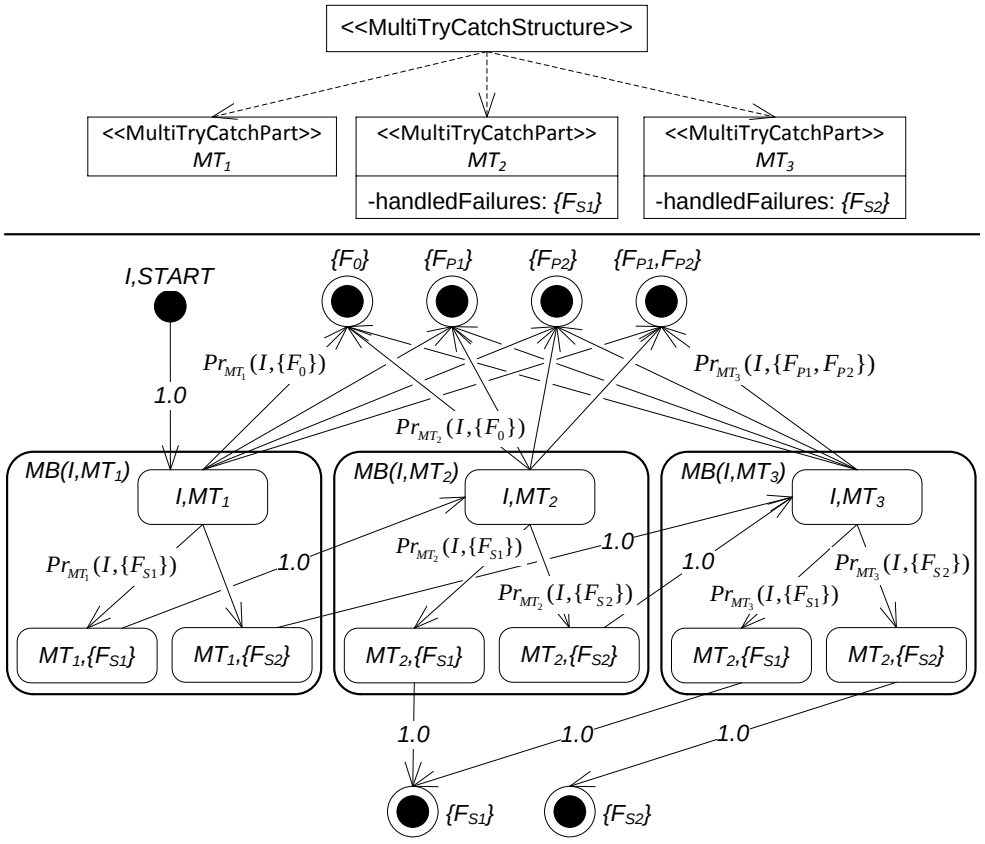


FIGURE 4.15: An example of transformation for a *MultiTryCatchStructure*.

Example 4.4. *Fig. 4.15 shows an example of transformation for a* MultiTryCatchStructure *(several transition probabilities are omitted for the sake of clarity). In this example, it is assumed that each* MultiTryCatchPart *has a failure model as in Example 3.4. Therefore,* $\mathcal{AFS} = \{\{F_{S1}\}, \{F_{S2}\}\}$ *and* $\mathcal{AIOS} = \{\{F_0\}, \{F_{P1}\}, \{F_{P2}\}, \{F_{P1}, F_{P2}\}\}$. $\mathcal{F}_{\mathcal{H}2}$ *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 internal activity 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* $[I, START]$.

4.1.2.7 MVPStructure

Considering a *MVPStructure*, let n be the number of *MVPParts*, $maxTI$ be the value of the field *maxToleratedIncompletion*, $minMS$ be the value of the field *minMajoritySize*, $F_{sF} \in \mathcal{AFS}$ be the value of the field *signaledFailure*, $F_{eO} \in \mathcal{AIOS} \setminus \{\{F_0\}\}$ be the value of the field *erroneousOutput*, $\mathbf{EPV}^i$ be the error property vector for *MVPPart* i with $i \in \{1, 2, ..., n\}$ which is represented by $\{d^i_{EP}\}$ with $EP \in \mathcal{AIOS} \setminus \{\{F_0\}\}$, **AEV** be the value of the field *agreementOfErrorsVector* which is represented by $\{p_y\}$ with

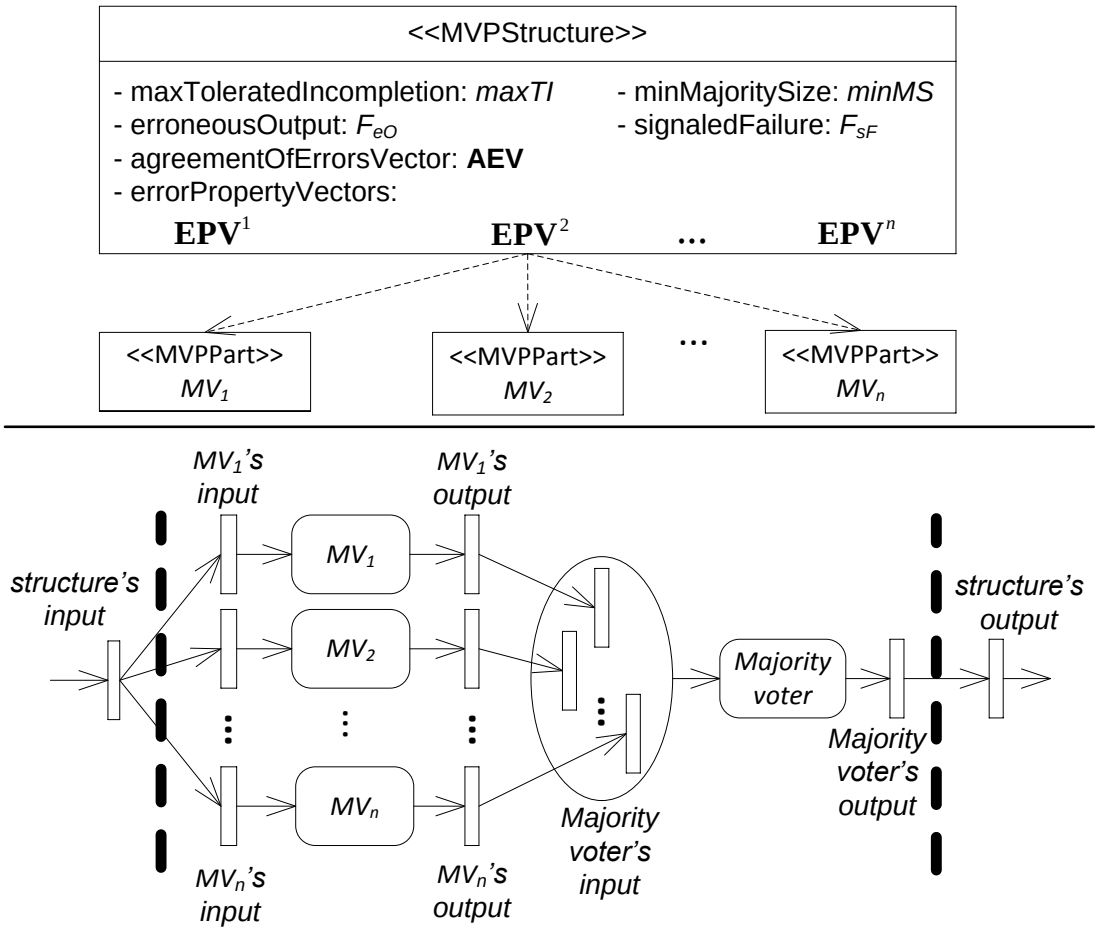


FIGURE 4.16: Using inputs and outputs in a *MVPStructure*.

$minMS \leq y \leq n$, $Pr_{MV_i}(I, FO)$ for all $I \in \mathcal{AIOS}$, $FO \in (\mathcal{AFS} \cup \mathcal{AIOS})$ be the failure model of *MVPPart* i (abbreviated as MV_i).

Fig. 4.16 shows the usage of inputs and outputs in a *MVPStructure*. The structure's input is the input for all *MVPParts*' executions and the structure's output is the output of the majority voter of the *MVPStructure*.

In order to transform a *MVPStructure* into an equivalent internal activity, for each possible input $I \in \mathcal{AIOS}$ of the *MVPStructure*, the transformation calculates the probabilities of the possibilities for the set of results of *MVPParts*' executions (Step 1), and then the probabilities for the voter to signal a signaled failure or to produce a correct or erroneous output (Step 2) (following the semantics as illustrated in Fig. 3.11). After that, the failure model for the equivalent internal activity can be derived from the probabilities for the voter by the transformation (Step 3).

Step 1, the transformation calculate the probabilities of the possibilities for the set of results of *MVPParts*' executions in an accumulative manner. After the executions of the first k *MVPParts*, let $(x, y, z)_k$ be a possibility for the set of results of these

MVPParts' executions where x is the number of correct outputs, y is the number of erroneous outputs, and z is the number of incomplete executions such that $x+y+z=k$, let $p\left((x,y,z)_k\right)$ be the probability of the possibility $(x,y,z)_k$. Therefore, there are $(k+2)(k+1)/2$ possibilities and the same number of probabilities. At the beginning, there is one possibility $(0,0,0)_0$ with probability $p\left((0,0,0)_0\right)=1$.

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

$$
\begin{aligned}
p\left((x,y,z)_{k+1}\right) \;=\; & p\left((x-1,y,z)_k\right) Pr_{MV_{k+1}}\left(I,\{F_0\}\right) \Big| \; x>0 \\
+ \; & p\left((x,y-1,z)_k\right) \sum_{O'\in\mathcal{AIOS}\setminus\{\{F_0\}\}} Pr_{MV_{k+1}}\left(I,O'\right)\left(1-d_{O'}^{k+1}\right) \Big| \; y>0 \\
+ \; & p\left((x,y,z-1)_k\right) \sum_{O'\in\mathcal{AIOS}\setminus\{\{F_0\}\}} Pr_{MV_{k+1}}\left(I,O'\right) d_{O'}^{k+1} \Big| \; z>0
\end{aligned}
\tag{4.8}
$$

By using Equation 4.8, the transformation recursively calculates the probabilities for all the possibilities of the set of results of n *MVPParts*' executions.

Fig. 4.17 shows the Markov chain that supports our argumentation. It starts with the state $[(0,0,0)_0]$, and ends with states $[(n,0,0)_n]$, $[(n-1,1,0)_n]$, ..., $[(0,n,0)_n]$,..., $[(0,1,n-1)_n]$, $[(0,0,n)_n]$. From state $[(x,y,z)_k]$, there are there transitions, including a transition to state $[(x+1,y,z)_k]$ with probability $Pr_{MV_{k+1}}(I,\{F_0\})$ (i.e. the probability that the $(k+1)-th$ *MVPPart* produces a correct output), a transition to state $[(x,y+1,z)_k]$ with probability $\sum_{O'\in\mathcal{AIOS}\setminus\{\{F_0\}\}} Pr_{MV_{k+1}}\left(I,O'\right)\left(1-d_{O'}^{k+1}\right)$ (i.e. the probability that the the $(k+1)-th$ *MVPPart* produces an erroneous output), and a transition to state $[(x,y,z+1)_k]$ with probability $\sum_{O'\in\mathcal{AIOS}\setminus\{\{F_0\}\}} Pr_{MV_{k+1}}\left(I,O'\right) d_{O'}^{k+1}$ (i.e. the probability that the $(k+1)-th$ *MVPPart* does not complete its execution in time).

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

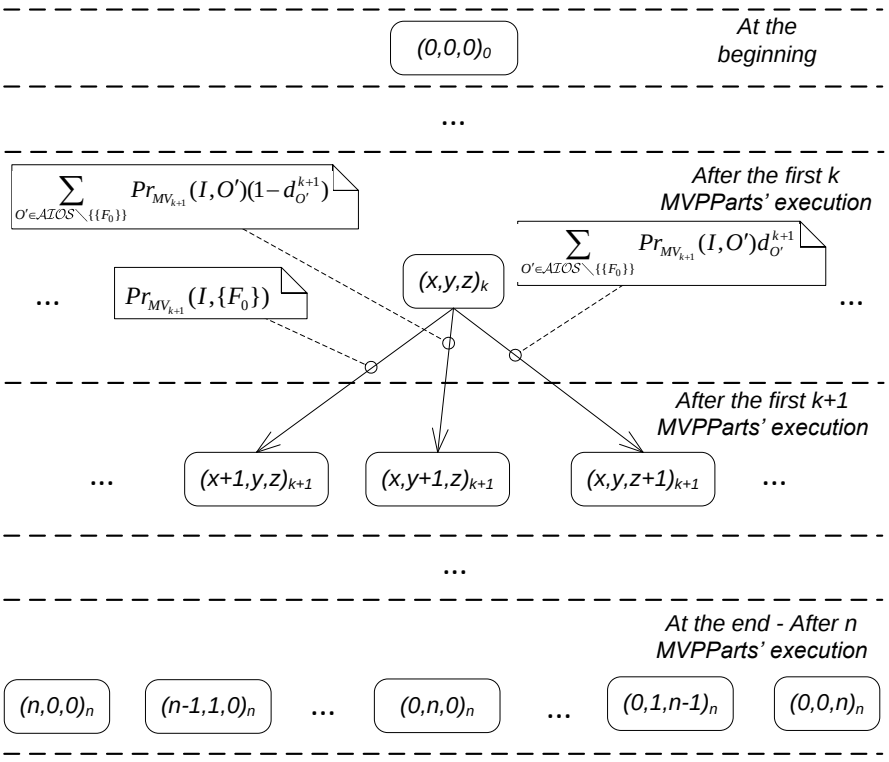


FIGURE 4.17: Markov chain in a *MVPStructure*.

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

$$p\left(\{F_0\}\right) = \sum_{(x,y,z)_n} p\left((x,y,z)_n\right) | z \leqslant maxIT, x \geqslant minMS \tag{4.9}$$

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

$$p(F_{eO}) = \sum_{(x,y,z)_n} p\left((x,y,z)_n\right) p_y | z \leqslant maxIT, x < minMS, y \geqslant minMS \tag{4.10}$$

- The voter signals a signaled failure of F_{sF} with probability:

$$p\left(F_{sF}\right) = 1 - p\left(\{F_0\}\right) - p\left(F_{eO}\right) \tag{4.11}$$

Step 3, the failure model for the equivalent internal activity can be derived from the probabilities for the voter as follows:

$$\begin{aligned} Pr_{IA}\left(I, \{F_0\}\right) &= p\left(\{F_0\}\right) \\ Pr_{IA}\left(I, F_{eO}\right) &= p\left(F_{eO}\right) \\ Pr_{IA}\left(I, F_{sF}\right) &= p\left(F_{sF}\right) \end{aligned} \tag{4.12}$$

4.1.2.8 The Reliability under the Usage Profile Part

As described in Section 3.2.2.3, under the current usage profile part, the reliability for a provided service is $1 - POFOD$, where $POFOD$ is the probability of failure on demand, given that the input of the service provided by the system is correct (i.e., as defined by its specification). Therefore, the reliability for the provided service to which the current usage profile part refers is the probability that this service produces a correct output given that it has received a correct input: $Pr_{IA}\left(\{F_0\}, \{F_0\}\right)$ of the failure model for the equivalent internal activity of the service implementation of this service. Notice that, from this failure model, other reliability properties of the provided service, e.g. *Internal failure probabilities* or *Robustness*, can also be derived (see Section 3.2.1.2).

4.1.3 Aggregation of Results

The results of the reliability of provided services to which the usage profile parts in the usage profile refer 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:

$$R = \sum_{j=1}^{m} P_j \; R(UPP_j) \tag{4.13}$$

Example 4.5. *Continuing with Example 3.8, the overall system reliability is determined as* $R = 0.7R(UPP_1) + 0.3R(UPP_2)$.

4.1.4 Complexity

Regarding space-effectiveness, by transforming a structure into an equivalent internal activity, the transformation algorithm no longer needs to store the structure with its

TABLE 4.2: 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	$O\left(n_B \lvert\mathcal{AIOS}\rvert \left(\lvert\mathcal{AIOS}\rvert + \lvert\mathcal{AFS}\rvert\right)\right)$
Parallel structure	$O\left(n_P \lvert\mathcal{AIOS}\rvert \left(\lvert\mathcal{AIOS}\rvert^2 + \lvert\mathcal{AFS}\rvert\right)\right)$
Sequential looping structure	$O\left(log_2\left(v_L\right) \lvert\mathcal{AIOS}\rvert^2 \left(\lvert\mathcal{AIOS}\rvert + \lvert\mathcal{AFS}\rvert\right)\right)$
Parallel looping structure	$O\left(log_2\left(v_L\right) \lvert\mathcal{AIOS}\rvert \left(\lvert\mathcal{AIOS}\rvert^2 + \lvert\mathcal{AFS}\rvert\right)\right)$
RetryStructure	$O\left(rc^2 \lvert\mathcal{AIOS}\rvert \lvert\mathcal{AFS}\rvert^2 \left(rc \lvert\mathcal{AFS}\rvert + \lvert\mathcal{AIOS}\rvert\right)\right)$
MultiTryCatchStructure	$O\left(n_{MT}^2 \lvert\mathcal{AIOS}\rvert \lvert\mathcal{AFS}\rvert^2 \left(n_{MT} \lvert\mathcal{AFS}\rvert + \lvert\mathcal{AIOS}\rvert\right)\right)$
MVPStructure	$O\left(n_{MV} \lvert\mathcal{AIOS}\rvert \left(\lvert\mathcal{AFS}\rvert + \lvert\mathcal{AIOS}\rvert + n_{MV}^2\right)\right)$

inner parts in the memory, but can efficiently transform the outer structure using the equivalent internal activity. Due to its recursive nature, the algorithm transforms a structure as soon as its inner parts have been transformed into equivalent internal activities, 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 internal activities of its inner parts, apart from the fact that the algorithm requires an additional amount of memory for (1) a Markov chain in case of a *RetryStructure* or a *MultiTryCatchStructure*, or (2) the possibilities of the set of results of *MVPParts*' executions and their probabilities in case of a *MVPStructure*. The aggregation of results over all usage profile parts in the usage profile can be calculated one after another, without the need to store each result separately.

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 (4.1), (4.2), ..., (4.12), it is possible to obtain the running times of the algorithm for the sequential, branching, parallel, looping, and *MVPStructure* 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 4.1.2.5 or 4.1.2.6, respectively). Table 4.2 shows the running times of the algorithm for structure types given that their inner parts have been transformed into equivalent internal activities. Abbreviations used in the table are as follows:

- $|\mathcal{AFS}|$: cardinality of $\mathcal{AFS}$, equal to u which is the number of stopping failure types;
- $|\mathcal{AIOS}|$: 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;
- v_L: value of the loop count of a looping structure;
- n_P: number of parallel parts of a parallel structure;
- rc: retry count of *RetryStructure*;
- n_{MT}: number of *MultiTryCatchParts* of a *MultiTryCatchStructure*.
- n_{MV}: number of *MVPParts* of a *MVPStructure*.

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 v_L. The fact that the algorithm for a *RetryStructure* (resp. *MultiTryCatchStructure*) involves calculations on matrices (i.e. matrix subtraction, inversion, and multiplication)[5] leads to a cubic time of the algorithm in rc (resp. n_{MT}). The running time of the algorithm for a *MVPStructure* is also cubic time in n_{MV}. 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 the RMPI approach. Therefore, scalability considerations are included in the case study (see Chapter 5 for more details).

4.2 Implementation

The transformation algorithm has been implemented in the reliability prediction tool of the RMPI approach. The tool receives a system reliability model as an input, validates this input against a set of predefined semantic constraints in the reliability modeling

[5]It is 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 $O(mpn)$.

schema of the approach (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.

As a part of the tool, a reliability simulator has also been implemented. 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, a method is implemented 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 $Pr_{IA}(I, O)$. A method is also implemented 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, it can be used 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 Chapter 5 for more details).

Fig. 4.18 shows a screenshot of the reliability prediction tool with its command-line user interface. The tool is open source and available at our project website [82].

4.3 Reliability Improvements with RMPI

This section gives an overview of possible model changes for reliability improvements, and describes how to reflect these changes in the RMPI approach.

If the prediction result shows that the given reliability goal cannot be met, it is possible to apply model changes or architectural tactics [85] to improve the system reliability (cf. Fig. 3.1). These changes usually come with extra costs and likely downgrade other quality attributes of the system. Therefore, software architects are responsible to

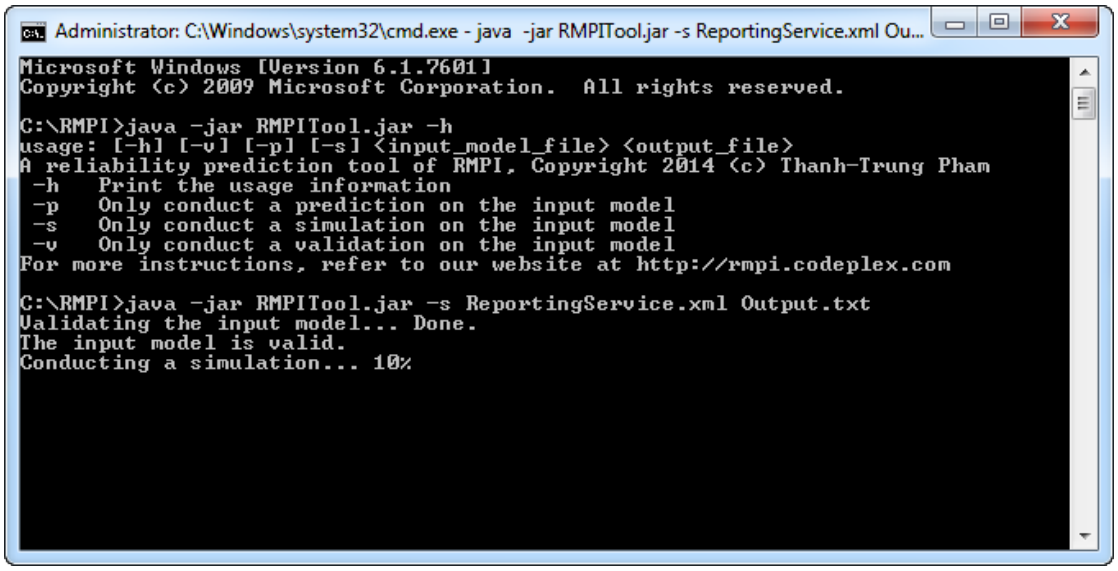


FIGURE 4.18: Reliability prediction tool support.

TABLE 4.3: Reliability Improvements Collection.

Name	**Type**	**Short Description**	**Non-reliability Impacts**	**Refections in Modeling**
Change component connectors	Topological	Change component connectors in such a way that services are provided by less reliability-sensitive components	Impacts on other quality attributes, e.g. performance	Change the system architecture model
High reliability components	Scalar	Apply high quality development processes to components for higher reliability	More testing and implementation efforts	Decrease failure probabilities of internal activities
Design diversity	Scalar/ Topological	Recovery Blocks, N-version programming	Extra costs for multiple designs, impacts on performance because of redundant computations	Decrease directly failure probabilities of internal activities/Introduce *MultiTryCatchStructures*, *MVPStructures* as additional components
Restart and retry techniques	Scalar/ Topological	Restart and retry components to ensure high reliability	Impacts on performance because of restarts and retries	Decrease directly failure probabilities of internal activities/Introduce *RetryStructures* as additional components

evaluate different possible solutions and decide an optimal trade-off between all existing goals of quality attributes and costs.

There are two types of model improvements supported by the RMPI approach, namely, topological improvements and scalar improvements. Topological improvements change the structure of the system architecture for improved reliability, e.g. changing component connectors in such a way that services are provided by less reliability-sensitive components. Scalar improvements cover changes to values of the input model parameters, e.g. changing failure probabilities of internal activities. It is possible to apply changes independently.

Table 4.3 shows a collection of model improvements, including their short descriptions, possible impacts on other quality attributes and costs, and reflections in modeling. The type and number of changes reflected in modeling are dependent on the concrete model improvements. For example, considering a replacement of a component by n redundant components for improved reliability, this replacement can be reflected in modeling by directly decreasing the failure probabilities of the internal activities of the original component (i.e. a scalar improvement) or by modeling the redundant components and then introducing a fault tolerance structure, e.g. a *MultiTryCatchStructure* or *MVPStructure*, as an additional component (i.e. a topological improvement). However, not every change to the model is an improvement, e.g. replacing a component with an alternative can have both positive and negative influence on the system reliability, depending on the specific system architecture. The case studies in Chapter 5 illustrate the application of different model improvements.

Being based on a system model rather than the actual system, the RMPI approach allows evaluating the influence of changes on the system reliability, without reimplementation, reconfiguration, and execution of the actual system. The task of software architects is to evaluate possible changes and choose the most beneficial one. The task is repeated gradually from the initial system model until a system model satisfying the existing goals of quality attributes and costs. Notice that the influence of a single change on the system reliability may depend the order of applying changes, e.g. it is more beneficial to introduce fault tolerance structures for a component after decreasing the failure probabilities of propagating failure types of its internal activities.

4.4 Summary

This chapter has described the analysis method provided by the RMPI approach for the reliability evaluation. More concretely, it described how to conduct transformations for usage profile parts and for structure types, and how to aggregate the transformation results for the reliability evaluation. It also investigated the complexity of the algorithm. It briefly introduced the implementation of the algorithm for tool support. Finally, it presented an overview of model changes supported by the approach for reliability improvements.

Chapter 5

Case Study Evaluation

5.1 Goals and Settings

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

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. [12, 16–18, 23, 63]). 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 [86] because the necessary number of tests and the necessary time for this are prohibitive. Therefore, in this dissertation, we validate prediction results of the RMPI approach 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 for the reporting service (Section 5.2), the WebScan system (Section 5.3), and the DataCapture system (Section 5.4), and present the scalability of the RMPI approach (Section 5.5).

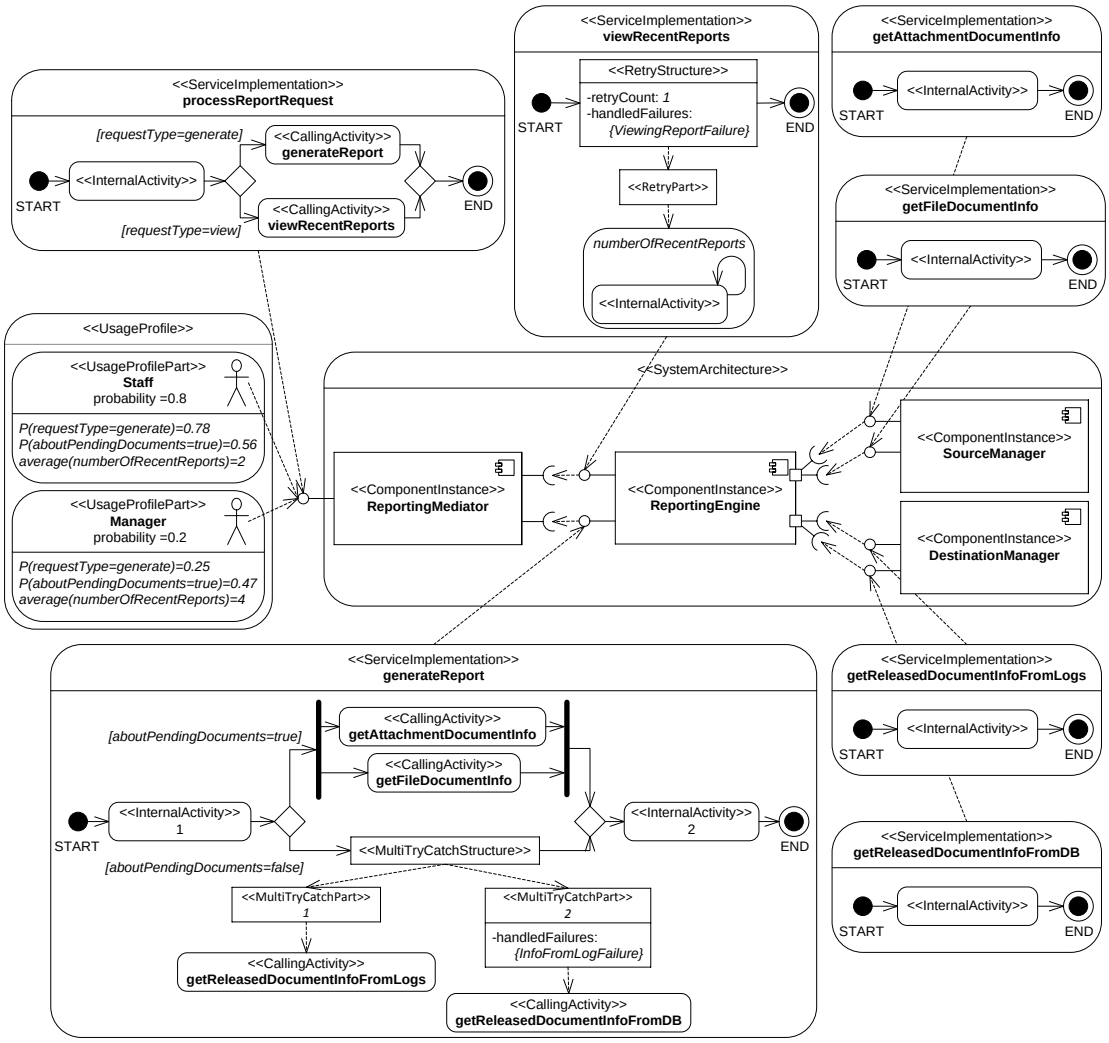


FIGURE 5.1: The system reliability model of the reporting service (overview).

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

5.2.1 Description of the Case Study

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

The system reliability model of the reporting service[1] is shown in Fig. 5.1 using the reliability modeling schema of the RMPI approach. At the architecture level, the reporting

[1]The model can be retrieved from our project website [82].

TABLE 5.1: Reporting Service: Different Propagating Failure Types and their Symbols.

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

TABLE 5.2: 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}

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. 5.1. 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.

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

TABLE 5.3: 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}

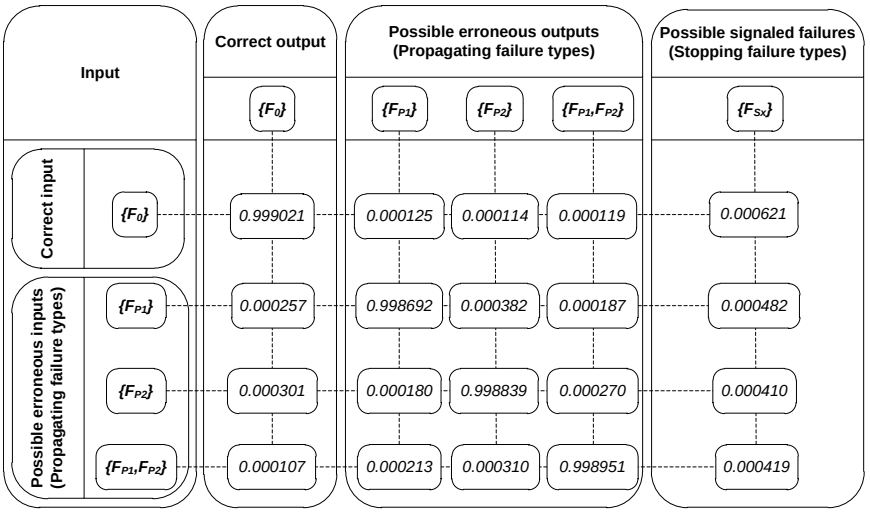


FIGURE 5.2: Reporting service: Failure model for internal activity a_i.

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 5.1 (resp. Table 5.2). Table 5.3 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 dissertation. However, in order to make our model as realistic as possible, we aligned the probabilities with the remarks by Cortellessa et al. [13]: (1) With modern testing techniques, it is practically always possible 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. 5.2 where F_{Sx} is the involved stopping failure type for a_i.

TABLE 5.4: Reporting Service: Predicted vs. Simulated Reliability

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

In the system reliability model, there are two fault tolerance structures. 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 {*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*.

5.2.2 Validity of Predictions

To validate the accuracy of prediction results of the RMPI approach, we used the system reliability model of the reporting service as an input for the reliability prediction tool of the approach to get the reliability prediction result, then compared this prediction result to the result of a reliability simulation. Notice that the goal of the validation is not to justify the probabilities of the failure models for internal activities. Instead, we validate that the method of the approach 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, the 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 5.4 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 dissertation, the analytical method of the RMPI approach is sufficiently accurate.

5.2.2.1 Sensitivity Analyses and the Impacts of Fault Tolerance Structures

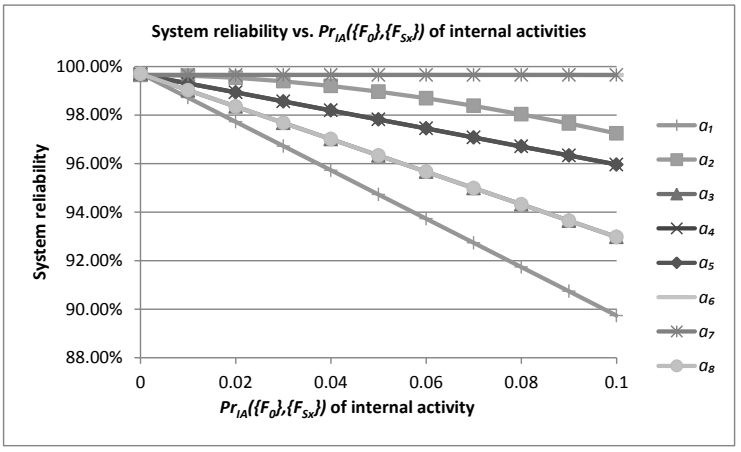
To demonstrate the capabilities of the RMPI approach in supporting design decisions, we present the results of sensitivity analyses of the reliability of the reporting service to

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.

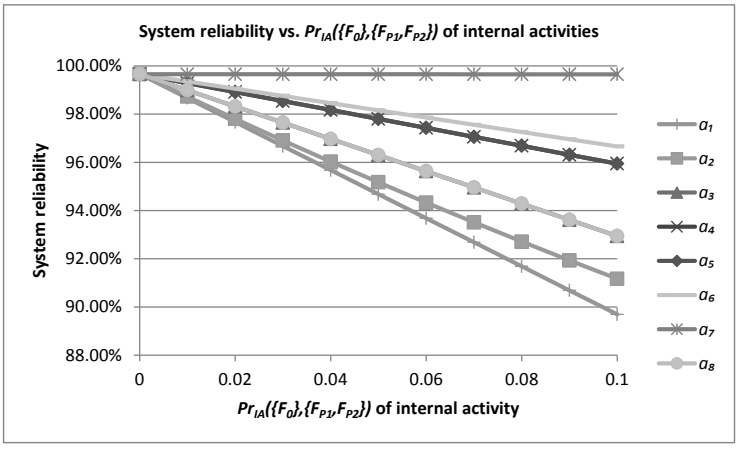
First, we conducted a sensitivity analysis modifying probabilities $Pr_{IA}(\{F_0\}, \{F_{Sx}\})$ of the internal activities (Fig. 5.3(a)). The reliability of the reporting service is most sensitive to the probability of internal activity a_1 of service *processReportRequest* provided by the component 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 fault tolerance structures, 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. 5.3(b)). 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 fault tolerance structures as in the first sensitivity analysis because the fault tolerance structures cannot provide error handling for erroneous outputs of propagating failure types $\{F_{P1}, F_{P2}\}$. Among 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 fault tolerance structures in the system.

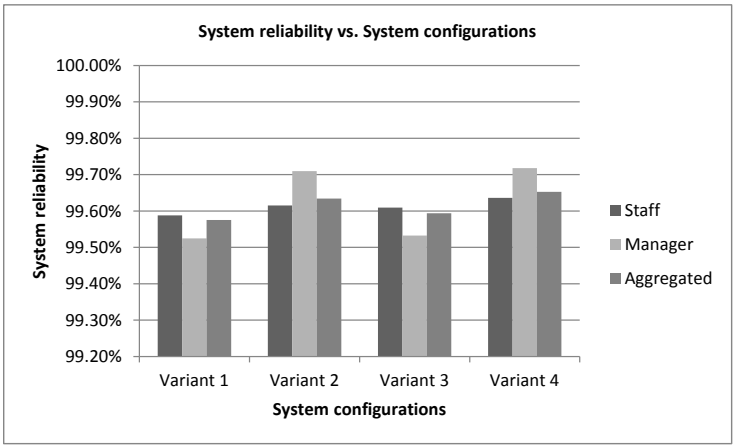
Third, we conducted an analysis of how the predicted reliability of the reporting service varies for fault tolerance variants. These variants include: without the fault tolerance structures (Variant 1), using only the *RetryStructure* (Variant 2), using only the *MultiTryCatchStructure* (Variant 3) and using both the fault tolerance structures (Variant 4) (Fig. 5.3(c)). In general, introducing fault tolerance structures brings increases in reliability for staffs, managers, or all (when aggregated). Variant 4 is predicted as being the most reliable. Comparing between Variant 2 and Variant 3 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 fault tolerance structures, increasing the number of



(a)



(b)



(c)

FIGURE 5.3: Reporting service: Sensitivity analyses.

TABLE 5.5: WebScan System: Propagating Failure Type and Its Symbol.

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

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 the RMPI approach for evaluating the reliability impacts of fault tolerance variants or system configurations. For Variant 3, 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 2, 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 1, the two above modifications are included.

5.3 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. 5.4 shows the system reliability model for this system[2].

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* 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.

[2]The model can also be retrieved from our project website [82].

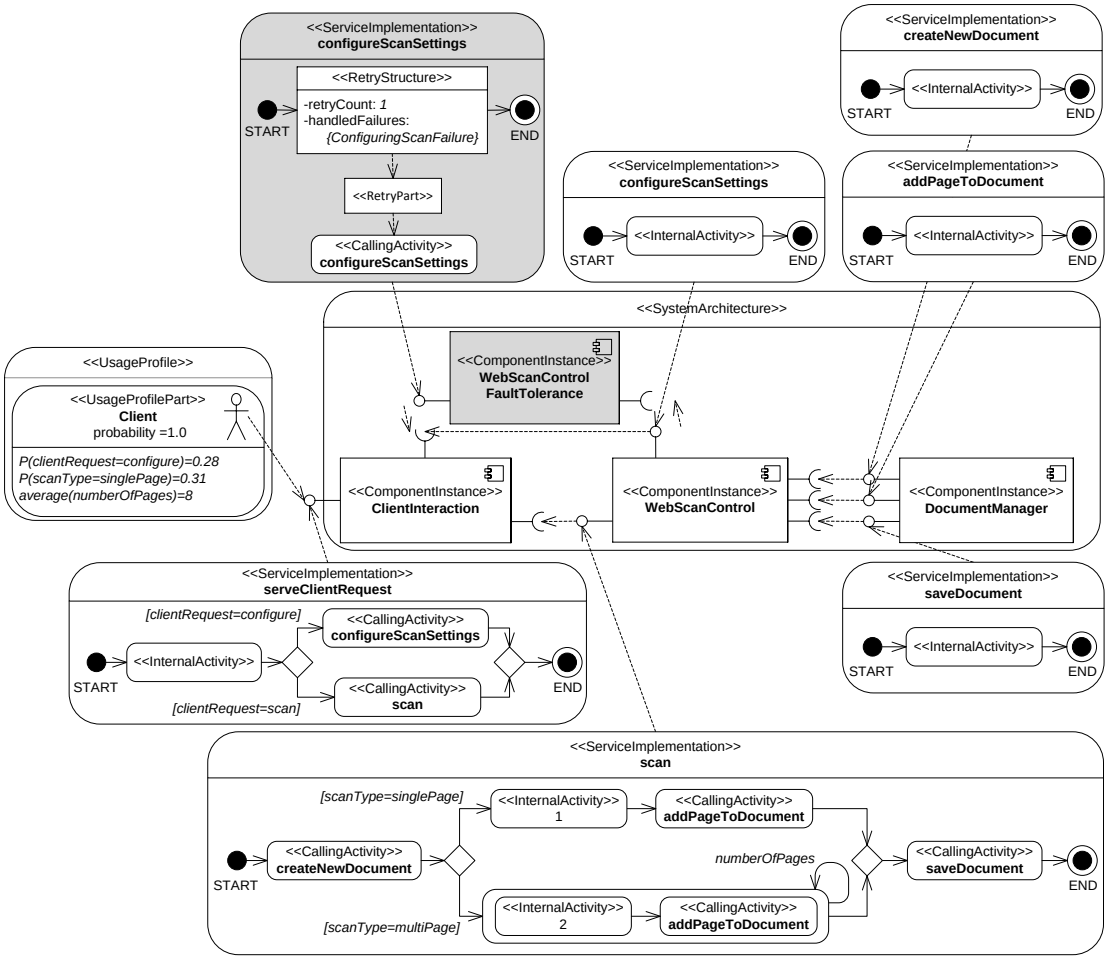


FIGURE 5.4: The system reliability model of the WebScan system (overview).

TABLE 5.6: 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}

TABLE 5.7: 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	*saveDocument/IA*	F_{S6}, F_{P1}

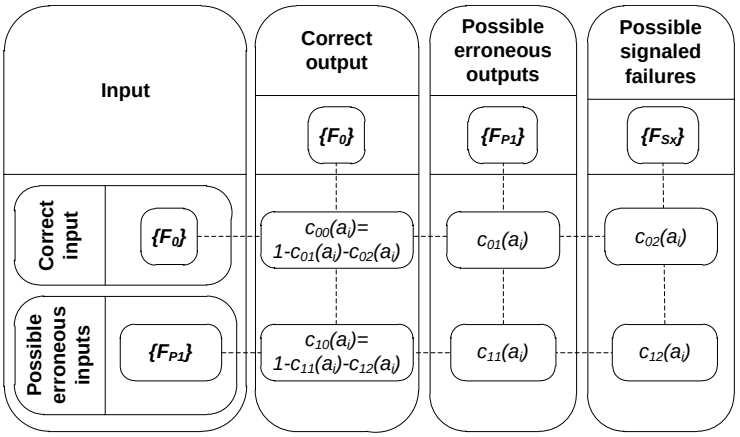


FIGURE 5.5: WebScan system: Failure model for internal activity a_i.

TABLE 5.8: 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

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 5.5 shows a propagating failure type and its symbol and Table 5.6 shows different stopping failure types and their symbols. Internal activities, their symbols, and involved failure types are given in Table 5.7.

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. 5.5 where F_{Sx} is the involved stopping failure type for a_i. Table 5.8 shows the specific values for the probabilities in the failure models of the internal activities.

TABLE 5.9: WebScan System: Predicted vs. Simulated Reliability

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

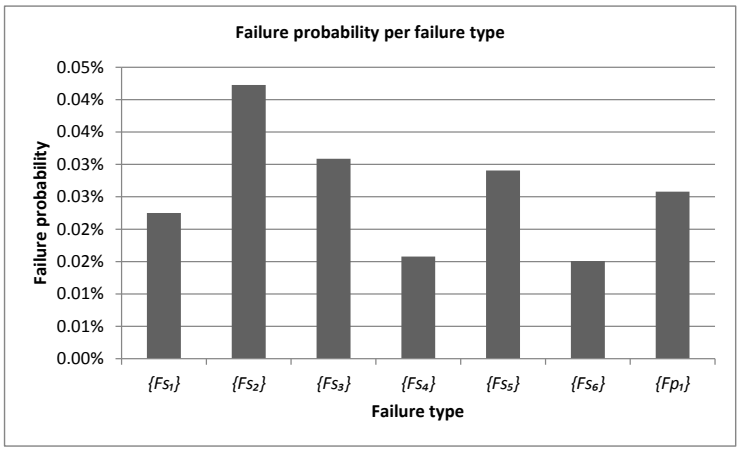
A fault tolerance structure can be optionally introduced into the WebScan System, in terms of an additional component which is shown in grey in Fig. 5.4. Component *WebScanControlFaultTolerance* can be put in the middle of component *WebScanControl* and component *ClientInteraction*. It has the ability 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 the reliability modeling tool of the RMPI approach predicted the system reliability as 0.998187. Table 5.9 compares the predicted system reliability and the simulated system reliability. This comparison gives evidence that the approach accurately predicts the system reliability in this case.

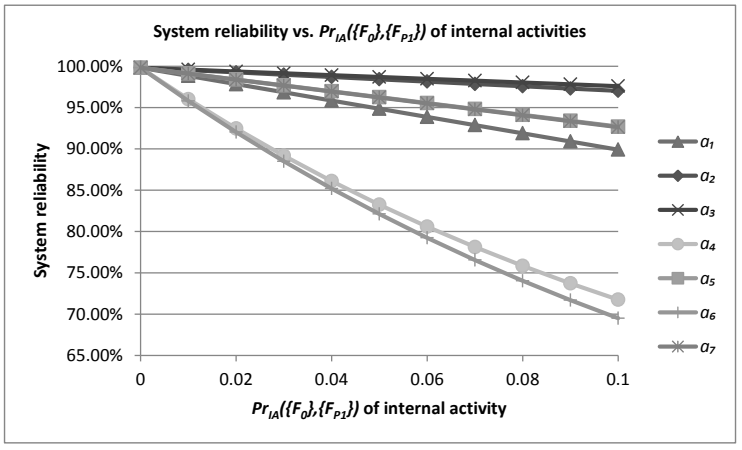
Fig. 5.6(a) provides more detail and shows the probability of a system failure due to a certain failure type. $\{F_{S2}\}$, $\{F_{S3}\}$), and $\{F_{S5}\}$ are the most frequent failure types. Thus, the software architect can recognize the need to introduce fault tolerance structures for these failures. 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. 5.4. 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 fault tolerance structure 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. 5.6(b) 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.

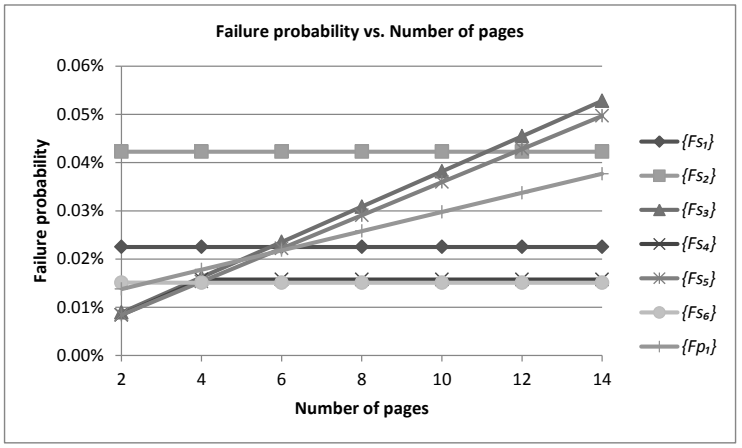
Fig. 5.6(c) shows the sensitivity of the failure probability per failure type to the number of pages (i.e., a change to the usage profile). As expected, only the failure probabilities for $\{F_{S3}\}$, $\{F_{S5}\}$, and $\{F_{P1}\}$ rise because they are the only failure types related to activities within the looping structure with loop count *numberOfPages*.



(a)



(b)



(c)

FIGURE 5.6: WebScan system: Sensitivity analyses.

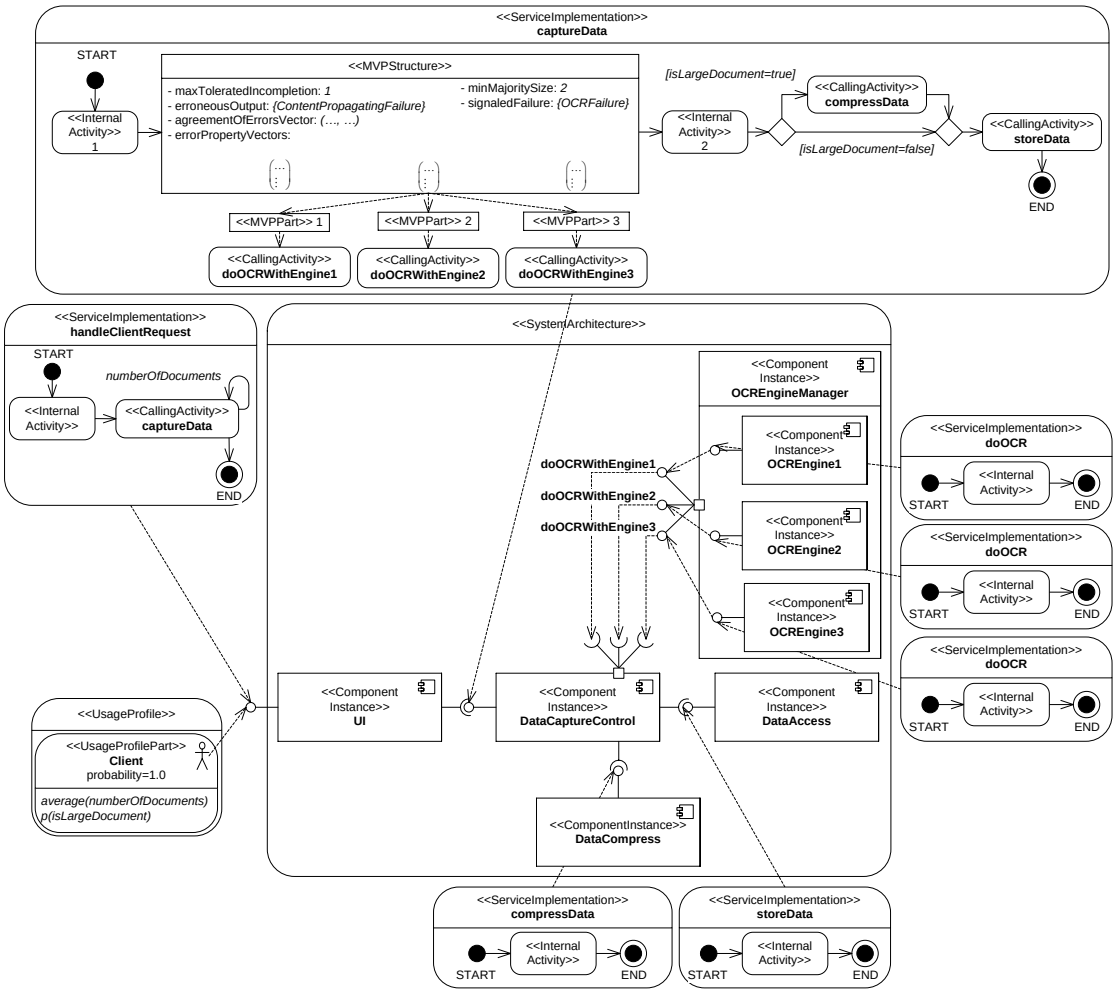


FIGURE 5.7: The system reliability model of the DataCapture system (overview).

TABLE 5.10: DataCapture System: Propagating Failure Type and Its Symbol.

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

5.4 Case Study III: DataCapture System

As the third case study, we analyzed the reliability of a DataCapture system. The system allows clients to capture data from printed texts such as documents, invoices, receipts, etc. using OCR (Optical Character Recognition) technology. Fig. 5.7 shows the system reliability model[3] for the standard system configuration with three OCR engines.

The system functionality is provided through four separated primitive components (*UI*, *DataCaptureControl*, *DataCompress*, and *DataAccess*) and one composite component (*OCREngineManager*) containing three nested primitive components (*OCREngine1*, *OCREngine2*, and *OCREngine3*). During the operation of the DataCapture system,

[3]The model can also be retrieved from our project website [82].

TABLE 5.11: DataCapture System: Different Stopping Failure Types and their Symbols.

Stopping Failure Type	**Symbol**
HandlingRequestFailure	F_{S1}
CapturingDataFailure	F_{S2}
DoingOCRFailure	F_{S3}
OCRFailure	F_{S4}
CompressingDataFailure	F_{S5}
StoringDataFailure	F_{S6}

TABLE 5.12: DataCapture System: Internal Activities, their Symbols, and Involved Failure Types.

Symbol	**Provided service/Internal activity (IA)**	**Involved Failure Types**
a_1	*handlingClientRequest/IA*	F_{S1}, F_{P1}
a_2	*captureData/IA 1*	F_{S2}, F_{P1}
a_3	*captureData/IA 2*	F_{S2}, F_{P1}
a_4	*OCREngine1's doOCR/IA*	F_{S3}, F_{P1}
a_5	*OCREngine2's doOCR/IA*	F_{S3}, F_{P1}
a_6	*OCREngine3's doOCR/IA*	F_{S3}, F_{P1}
a_7	*compressData/IA*	F_{S5}, F_{P1}
a_8	*storeData/IA*	F_{S6}, F_{P1}

TABLE 5.13: DataCapture System: Error Property Vectors.

Provided service/Fault tolerance structures	**Error property vector**
captureData/MVPStructure	$\{F_{P1}\}$ $\begin{matrix} IncExe \\ (0) \end{matrix}$

there are different errors which may occur in the involved component instances. For example, bugs in the code implementing the internal activity of service *doOCR* 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: {*DoingOCRFailure*}. Otherwise, an erroneous output of a propagating failure type is produced by the internal activity: {*ContentPropagatingFailure*}. Table 5.10 shows a propagating failure type and its symbol. Table 5.11 shows different stopping failure types and their symbols. Internal activities, their symbols, and involved failure types are given in Table 5.12.

There is a fault tolerance structure in the system, namely the *MVPStructure* in the implementation of service *captureData* of component *DataCaptureControl*. This structure tolerates at most one incomplete execution from three *doOCR* services and requires at least two results from these services to agree to produce an output. Besides a correct output, the structure can produce an erroneous output of {*ContentPropagatingFailure*}

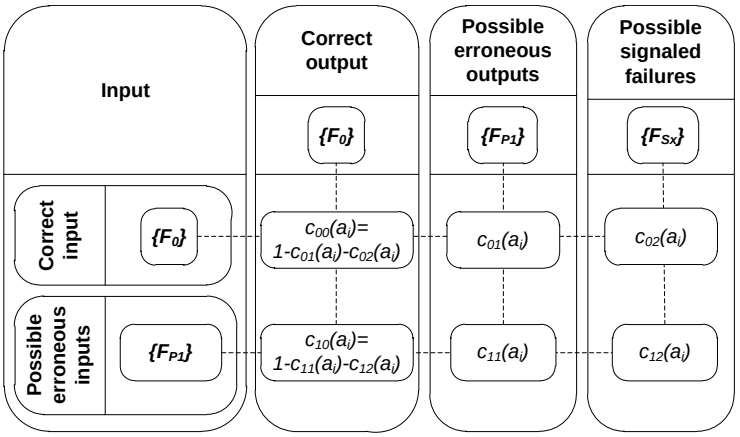


FIGURE 5.8: DataCapture system: Failure model for internal activity a_i.

or signal a signaled failure of $\{OCRFailure\}$. Because erroneous outputs of $\{Content\text{-}PropagatingFailure\}$ are content failures, the error property vectors for the three *MVP-Parts* are the same and given in Table 5.13.

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

To be able to conduct the measurements, several simplifications had to be included, compared with a real-world field experiment. First, the total number of scenario runs is limited to 12,000. Each scenario run consists of an average of 10 documents per call and a probability of 40% for the documents to be large, i.e. requiring compression before storing. Second, the system reliability is not measured due to real faults but rather to faults which have been injected in an artificial manner, with externally controlled occurrence probabilities.

By using a script, it is possible for us to determine the failure models for internal activities, the field *agreementOfErrorsVector* of the *MVPStructure*, and the measured system reliability from the execution traces. The probabilities of the failure model for the internal activity a_i (with $i \in \{1, 2, ..., 7\}$) are shown in Fig. 5.8 where F_{Sx} is the involved stopping failure type for a_i. The specific values for the probabilities in the failure models of the internal activities are shown in Table 5.14. Because in scenarios runs, documents used as inputs for the system were correct and no fault was injected into the two internal activities a_1 and a_2, the probabilities in their failure models are assumed to be 0. As a result, the inputs for the three internal activities a_3, a_4, and a_5 are always correct, and

TABLE 5.14: DataCapture 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	0	0	0
a_2	0	0	0	0
a_3	0.000988052	0.002527356	0.99437751	0.00562249
a_4	0.009980036	0.013947068	0	0
a_5	0.070123629	0.012995678	0	0
a_6	0.010028152	0.01903337	0	0
a_7	0.000795795	0.00207256	0.992041712	0.007135016
a_8	0.000399946	0.00200132	0.994810428	0.004456927

TABLE 5.15: DataCapture System: Predicted vs. Measured Reliability

Component Instance/ Provided service	**Predicted reliability**	**Measured reliability**	**Difference**	**Error (%)**
UI/handleClientRequest	0.886311	0.8811	0.005211	0.59142

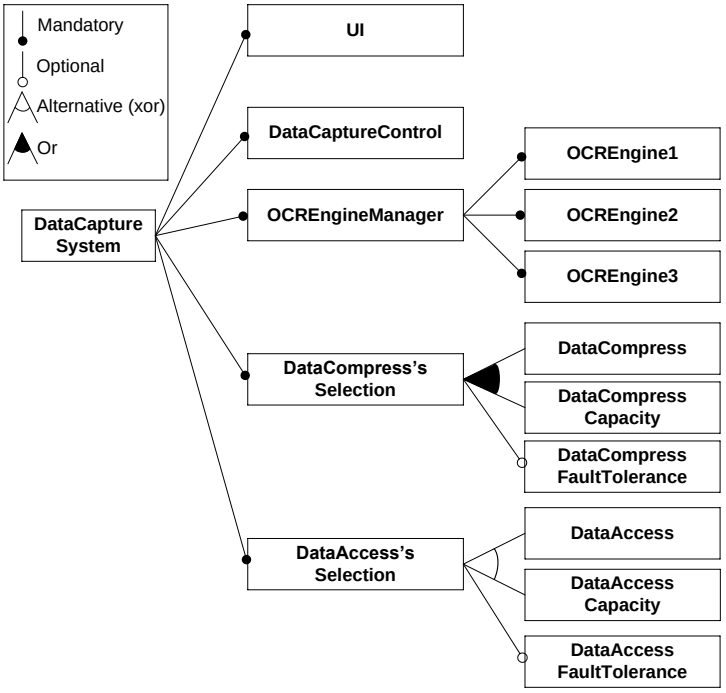


FIGURE 5.9: Feature model of variants of the DataCapture system .

therefore the probabilities $c_{11}(a_i)$ and $c_{12}(a_i)$ (with $i \in \{4,5,6\}$) are also assumed to be 0. The field *agreementOfErrorsVector* of the *MVPStructure* is $(p_2 = 0.1745, p_3 = 0.3014)$. For the predicted reliability of the system, a system reliability model is created with the support from the reliability modeling schema and then used as the input for the reliability prediction tool. Table 5.15 compares the predicted system reliability and the measured system reliability. This comparison gives evidence that the RMPI approach gives a reasonably accurate reliability prediction in this case.

Beyond the standard system configuration, different variants are possible for the system. Fig. 5.9 shows variants of the DataCapture system in terms of a feature model. They are made by using the alternative implementations for components *DataCompress* and *DataAccess*. The core functionality is provided via component types: *UI*, *DataCapture-Control*, *OCREngine1*, *OCREngine2*, *OCREngine3*, *DataCompress*, and *DataAccess* as the standard system configuration (cf. Fig. 5.7).

For the scope of this dissertation, we restricted the reliability analysis to the standard system configuration (Standard variant) and two further variants. Variant 1 is identical to the standard system configuration, except using component *DataCompressCapacity* instead of *DataCompress*. Variant 2 uses component *DataAccessCapacity* instead of *DataCompress*, and the other components are the same as in the standard system configuration.

Further fault tolerance structures can be optionally introduced into each DataCapture system variant, in terms of additional components which are shown in grey in Fig. 5.10. For example, Component *DataAccessFaultTolerance* may be put in the middle of component *DataCaptureControl* and component *DataAccess[Capacity]*. It has the ability to buffer *storeData* requests, to restart component *DataAccess[Capacity]*, and to retry the failed requests in case of signaled failures of {*StoringDataFailure*}. Component *DataCompressFaultTolerance* may be used to handle signaled failures of {*CompressingDataFailure*} of the main *DataCompress* component (i.e. component *DataCompress* in the Standard variant and Variant 2, or component *DataCompressCapacity* in Variant 1) by redirecting calls to the backup *DataCompress* component (i.e. component *DataCompressCapacity* in the Standard and Variant 2, or component *DataCompress* in Variant 1).

For illustrative purposes, we let a'_7 be the internal activity of service *compressData* of component *DataCompressCapacity*, its involved failure types and failure model are identical to those of internal activity a_7, except that $c_{01}(a'_7)$ and $c_{02}(a'_7)$ rise to 0.001194 and 0.003109, respectively, because of the more complex compression algorithm compared to the standard variant. Similarly, we let a'_8 be the internal activity of service *storeData* of component *DataAccessCapacity*, its involved failure types and failure model are identical to those of internal activity a_8, except that $c_{01}(a'_8)$ and $c_{02}(a'_8)$ fall to 0.00025 and 0.001251, respectively.

To provide evidence about the possible design decision support for different design alternatives, Fig. 5.11(a) shows the system reliability for each variant and fault tolerance alternative. Variant 1 has the lowest reliability, because of component *DataCompress-Capacity*. Variant 2 has the highest reliability, as a result of using component *DataAccessCapacity*. Employing component *DataAccessFaultTolerance* has the highest effect

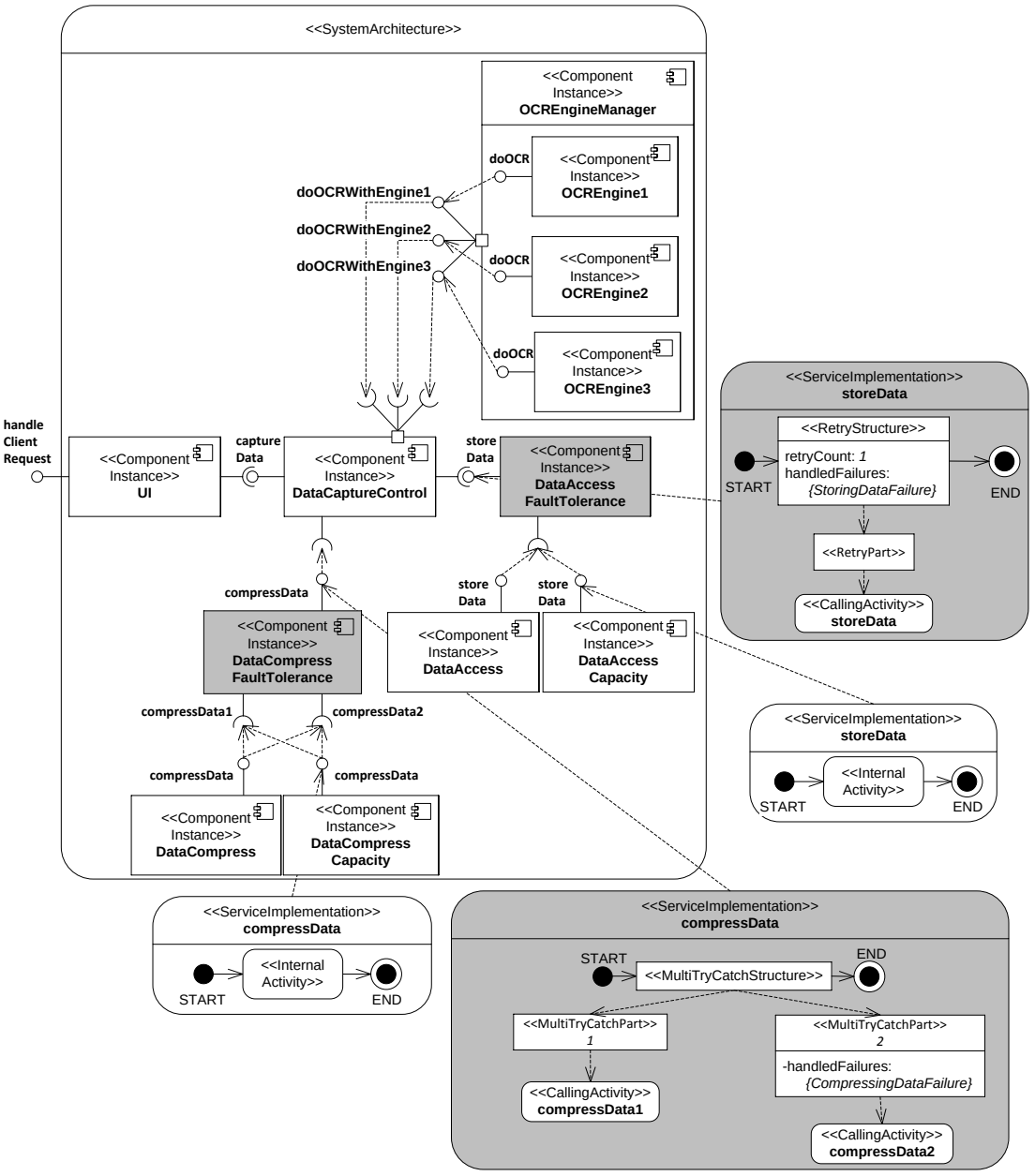
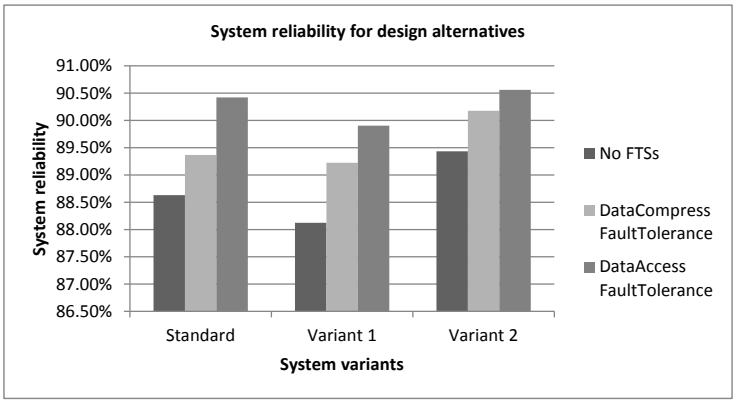


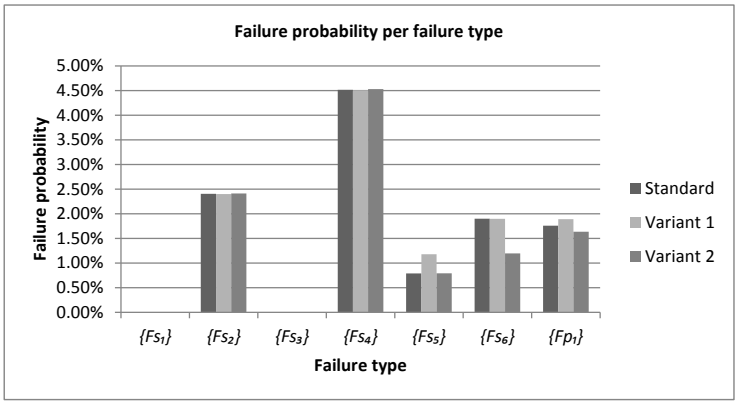
FIGURE 5.10: Variants of the DataCapture system.

compared to the design alternatives without fault tolerance. Notice that fault tolerance structures have different influences on variants, e.g. component DataCompressFaultTolerance is most effective for Variant 1.

Fig. 5.11(b) provides more detail and shows the probability of a system failure due to a certain failure type. Because the *MVPStructure* prevents signaled failures of $\{DoingOCRFailure\}$ ($\{F_{S4}\}$) of service *doOCR* of components *OCREngines* from manifesting as signaled failures of $\{DoingOCRFailure\}$ after the *MVPStructure*'s execution, the probability that the system fails with a signaled failures of $\{DoingOCRFailure\}$ is 0.



(a)



(b)

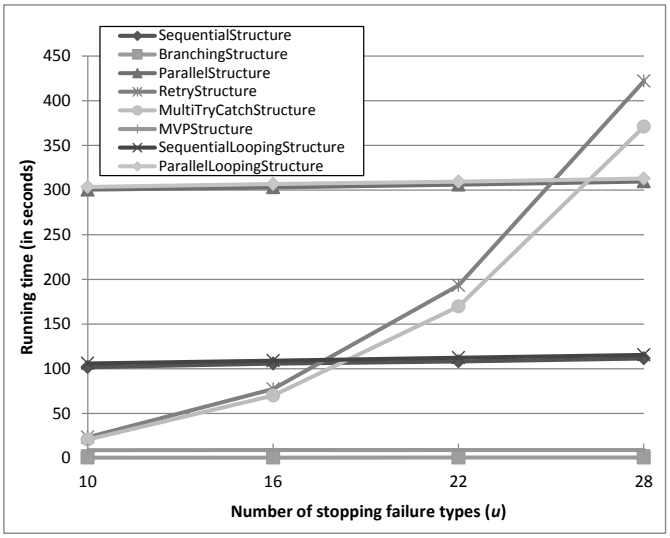
FIGURE 5.11: DataCapture system: Sensitivity analyses.

$\{F_{S2}\}$ and $\{F_{S4}\}$ are the most frequent failure types. Thus, the software architect can recognize the need to introduce fault tolerance structures for these failures.

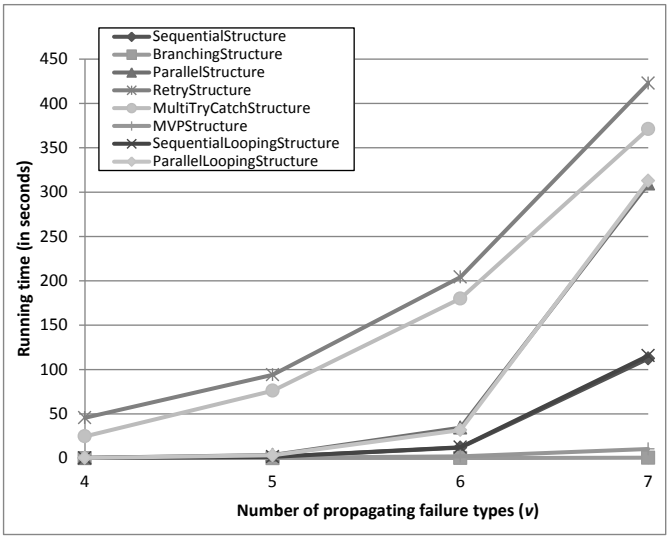
With this case study, it is also possible to see that nearly all modeling parts can be reused throughout variants and fault tolerance alternatives. Only some component instances need to be connected via additional component connectors, namely, component instance of components *DataCompressCapacity*, *DataAccessCapacity*, *DataCompressFaultTolerance*, and *DataAccessFaultTolerance*.

5.5 Scalability Analyses

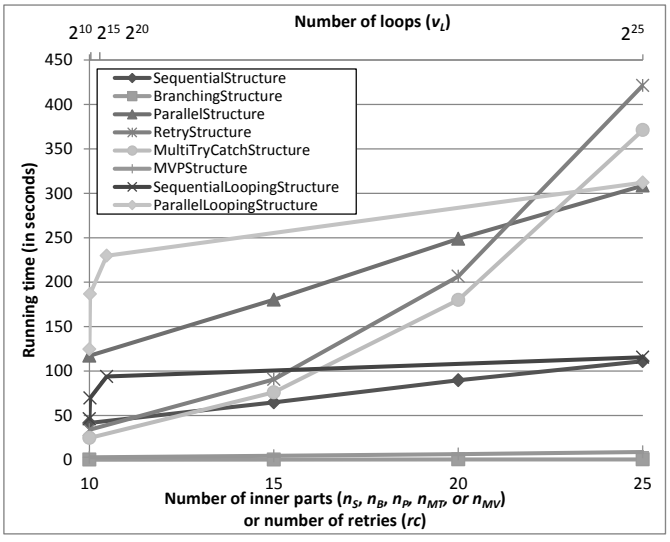
The scalability of the RMPI approach requires special attention. To examine it, for each structure type, we generated a number of simple system reliability models with different numbers of u, v, and n_S, n_B, v_L, n_P, rc, n_{MT}, or n_{MV} (cf. Section 4.1.4)), analyzed these system reliability models using the reliability prediction tool of the approach, and recorded the running times of the transformation algorithm. For example, for the



(a)



(b)



(c)

FIGURE 5.12: Scalability analyses.

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. 5.12(a) shows the running times of the algorithm for structure types with different numbers of u while $v = 7$, $n_S = n_B = n_P = rc = n_{MT} = n_{MV} = 25$, and $v_L = 2^{25}$. With the same numbers of n_S, n_B, n_P, rc, n_{MT}, n_{MV}, and v_L as above, in Fig. 5.12(b) 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_{MT}, n_{MV}, and v_L while $u = 28$ and $v = 7$ are shown in Fig. 5.12(c). These results provide evidence for the correctness of our complexity analysis for the transformation algorithm as in Section 4.1.4. For example, the running time of the algorithm for a sequential structure increases linearly with the number of stopping failure type (as in Fig. 5.12(a)), exponentially with the number of propagating failure types (as in Fig. 5.12(b)), and linearly with the number of its inner parts (as in Fig. 5.12(c)). The results 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.

Based on the transformation algorithm (see Sections 4.1.2.1, 4.1.2.2, ..., and 4.1.2.7), Table 5.16 shows the number of states of the equivalent underlying Markov chains for different structure types. The number of states in formulas are shown in Column 2 of the table. Column 3 of the table shows the number of states in values with $u = 28$, $v = 7$, $n_S = n_B = n_P = rc = n_{MT} = n_{MV} = 25$, and $v_L = 2^{25}$.

Therefore, we deem that the capacity of the RMPI 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

TABLE 5.16: Number of states of the Equivalent Underlying Markov Chains for Different Structure Types.

Structure type	**Number of states (Formulas)**	**Number of states (Values)**
Sequential structure	$(n_S + 2)\, 2^v + u$	3,484
Branching structure	$(n_B + 2)\, 2^v + u$	3,484
Parallel structure	$2^v\, ((n_P + 2)\, 2^v + u)$	445,952
Sequential looping structure	$(v_L + 2)\, 2^v + u$	4,294,967,580
Parallel looping structure	$2^v\, ((v_L + 2)\, 2^v + u)$	549,755,850,240
RetryStructure	$2^v\, (1 + (1 + u)\, (rc + 1) + u + 2^v)$	116,608
MultiTryCatchStructure	$2^v\, (1 + (1 + u)\, n_{MT} + u + 2^v)$	112,896
MVPStructure	$\frac{1}{6} 2^v\, (1 + n_{MV})\, (2 + n_{MV})\, (3 + n_{MV})$	419,328

systems (e.g. the WebScan system introduced in Section 5.3), 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.

5.6 Summary

This chapter has presented the case study evaluation of the RMPI approach. It sets up its validation goals and settings, and then described the predictions for three case studies, including the reporting service, the WebScan system, and the DataCapture system. These studies are served to demonstrate the reliability evaluation capabilities of the approach, to compare the obtained prediction results against simulations and a measurement conducted on a prototype implementation, as well as to provide evidence for the ability to support design decisions and for the reusability of modeling artifacts in evaluating different architecture variants under varying usage profiles. Finally, it investigated the scalability of the approach.

Chapter 6

Conclusions

6.1 Summary

This dissertation has presented the RMPI approach, a reliability modeling and prediction approach for component-based software systems, which considers explicitly error propagation, software fault tolerance mechanisms, and concurrently present errors, and supports design decisions for reliability improvements. The approach provides a reliability modeling language that captures comprehensively different reliability-influencing factors into a model of the system under study. Then, an analysis method evaluates the system model and obtains a prediction result as the probability of successful service execution.

With the support of the RMPI approach, software architects can assess the influence of possible changes in the system architecture and usage on the system reliability. They can identify the most critical parts in terms of reliability in the system architecture. They can assess possible design alternatives of the system and rank them according to their reliabilities. Being based on a system model rather than the actual system, the approaches can be applied at early design stages when essential design decisions at the architecture level are about to be made.

The RMPI approach belongs to the field of component-based software reliability modeling and prediction. While the approach receives benefits from the experiences gained in the field, the state of the art in the field is enhanced by the approach via its contributions as follows:

- *Consideration of error propagation*: The approach allows modeling error propagation for multiple execution models, including sequential, parallel, and fault

tolerance execution models. The approach considers how the error propagation affects the system execution with different execution models, and it derives the overall system reliability accounting for the error propagation impact.

- *Consideration of software fault tolerance mechanisms*: The approach offers enhanced fault tolerance expressiveness, explicitly and flexibly modeling how both error detection and error handling of fault tolerance mechanisms influence the control and data flow within components. These capabilities enable modeling comprehensively different classes of existing fault tolerance mechanisms and evaluating their impact on the system reliability.
- *Consideration of Concurrently Present Errors*: The approach is the first work to support modeling concurrently present errors. With this capacity, it is possible to cover system failures caused by the concurrent presence of errors, tending to obtain accurate prediction results.

The reliability modeling language of the RMPI approach has been implemented in the RMPI schema, offering a developer-friendly modeling notation. The language includes modeling elements for expressing the following aspects: (1) the structure of a software system in terms of its component instances and their interconnections, (2) the provided/required services of each software component, as well as its high-level internal control and data flow, (3) the system's usage profile in terms of a set of usage scenarios, where each scenario describes the sequences of invoked system services, (4) the failure possibilities that the system comprises, and (5) the system's capabilities to prevent the occurrence of system failures through software fault tolerance mechanisms.

The analysis method of the approach, that transforms the system reliability models based on the language into discrete-time Markov chains for reliability predictions and sensitivity analyses, has been implemented in the RMPI tool, allowing for a fully automated analysis and the display of the obtained results.

The RMPI approach has been validated in three case studies, by modeling the reliability, conducting reliability predictions and sensitivity analyses. Via these case studies, the approach has demonstrated its ability in supporting design decisions for reliability improvements and its reusability of modeling artifacts.

6.2 Assumptions and Limitations

In this section, we discuss assumptions and limitations of the RMPI approach, focusing on three central issues: (1) The provision of proper inputs for the approach, (2) the

Markovian assumption of the approach, and (3) the limitations in the expressiveness of the model.

6.2.1 Provision of Inputs

Perhaps, the most critical assumption lies in the provision of inputs for the RMPI approach. 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 [3, 4, 11, 86–88] 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.

The RMPI 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 [89]. 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 [3].

6.2.2 Markovian Assumption

The RMPI 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 the approach on different application domains. However, the

Markovian assumption has been proved to be valid at the component level for many software systems [8]. Moreover, the problem of the Markovian assumption in reliability modeling and prediction was treated deeply by Goseva et al. [3], 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, the approach can also be adapted to any higher order Markov chains, broadening the applicability of the approach to a large number of software systems.

6.2.3 Expressiveness of the Model

With regard to the expressiveness of the RMPI 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 [3–5]), we have restricted our approach to the most important concepts from our point of view (see Section 3.2). 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, these probability values are fixed constants. They cannot be adapted to take into consideration factors such as component state or system state at run-time.

6.3 Future Work

This section provides pointers for research extending the work conducted in this dissertation. It is divided into the following categories: (1) Enhanced methods for input estimations, (2) Extensions of modeling capabilities, (3) Extensions of analysis capabilities, and (4) Enhanced evaluation of prediction results.

6.3.1 Enhanced Methods for Input Estimations

Despite existing research efforts with regard to software reliability estimation, there are still important and unsolved challenges. Therefore, in order to provide adequate inputs at adequate granularity levels for the RMPI approach, new methods are required. They should consider the application phase of the approach (e.g. early design phase, system evolution) and the available sources of information in each phase. They should also focus

on the question how to collect relevant statistical failure data during the development process of the system and during the system operation, which can be used as a complete source of information for the required input estimations. To obtain reliable results, they should be validated in the development processes of real-world software systems.

6.3.2 Extensions of Modeling Capabilities

There are various directions to extend the existing modeling capabilities of the RMPI approach. In general, an extension to the modeling capabilities not only requires more input information but also enhanced analysis methods to deal with the extended modeling capabilities. Therefore, it is necessary to assess each possible extension with regard to the potentially involved modeling and analysis efforts. From our point of view, the following extensions could bring the most important benefits:

- *Parametric specifications for input model parameters*: Currently, the values for input model parameters of the RMPI approach are fixed constant, while in reality, the dependences between these input parameters do exist, e.g. a loop count may change dynamically within the inner part of a looping structure. Therefore, parametric specifications for input model parameters could extend the existing modeling capabilities of the RMPI approach to obtain higher expressiveness of the system behavior.

- *Stochastic dependences between failure possibilities*: The RMPI approach models failure possibilities as being independent although there exist interdependences between them in reality, e.g. during a scenarios run, multiple visits to the same components may be stochastically dependent, indicating the first visit may influence the success and failure probabilities of all subsequent visits in a very serious way. Although the approach could receive benefits from capturing such stochastic dependences, in order to avoid overstraining modelers, the corresponding extension should be done with caution.

- *Variance of input estimations*: The RMPI approach could be extended to take into account the uncertainty that exists in the estimates of its required inputs. Based on the involved variances, the approach could calculate the corresponding variances of the prediction results. With this capability, it could provide the ranking of design alternatives of the system with a degree of confidence. Although approaches in the field of component-based software reliability modeling and prediction provide uncertainty analyses, a new contribution to the field is still possible when extending these analyses with a combined consideration of error propagation for multiple execution models.

6.3.3 Extensions of Analysis Capabilities

The existing analysis capabilities of the RMPI approach could be extended to consider explicitly stochastic dependencies between multiple consecutive scenario runs, adding further value to the approach. Also, the analysis could be extended to take into account the possibility of multiple failure occurrences during a service execution, providing the number of occurred failures along with existing failure probabilities for multiple failure modes.

6.3.4 Enhanced Evaluation of Prediction Results

The case studies in this dissertation have shown that a single run of the analysis method may produce a high number of individual prediction results, and many input model parameters may exist whose values influence the results. It is apparently a challenge to find the most important parameters and then to derive solid interpretations of the results. Therefore, methods for automated selection of experiment runs and interpretation of the prediction results would provide improved assistance for answering the design questions concerning the system under study.

Author's Publications

[PBD14] Thanh-Trung Pham, François Bonnet, and Xavier Défago. Reliability prediction for component-based software systems with architectural-level fault tolerance mechanisms (Extended version). *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 5(1):4–36, 2014.

[PD12] Thanh-Trung Pham and Xavier Défago. Reliability prediction for component-based systems: Incorporating error propagation analysis and different execution models. In *Proceedings of the 12th International Conference on Quality Software (QSIC'12)*, pages 106–115, Xi'an, Shaanxi, China, 2012.

[PD13] Thanh-Trung Pham and Xavier Défago. Reliability prediction for component-based software systems with architectural-level fault tolerance mechanisms. In *Proceedings of the 8th International Conference on Availability, Reliability and Security (ARES'13)*, pages 11–20, Regensburg, Germany, 2013.

[PDH14] Thanh-Trung Pham, Xavier Défago, and Quyet-Thang Huynh. Reliability prediction for component-based software systems: Dealing with concurrent and propagating errors. *Science of Computer Programming*, 2014. (Accepted, Online preprint).

[PHD12] Thanh-Trung Pham, Quyet-Thang Huynh, and Xavier Défago. Making reliability modeling of component-based systems usable in practice (Fast abstract). In *Local Proceedings of The 18th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'12)*, Niigata, Japan, 2012.

Remark The RMPI approach and its contributions have been described in multiple peer-reviewed publications [PD12, PHD12, PD13, PBD14, PDH14]. The preliminary work of the approach has been developed in [PD12, PHD12]. The most significant work is an article in the *Science of Computer Programming* journal [PDH14], which is currently accepted for publication and available in an online preprint version. The

capabilities of the approach for the consideration of software fault tolerance mechanisms are specifically covered in [PD13, PBD14].

Bibliography

[1] IEEE Reliabbility Society. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[2] ACM SIGSOFT. RISKS Section. *Software Engeering Notes*, 18(1), 1993.

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

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

[5] Anne Immonen and Eila Niemelä. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, 2008.

[6] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

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

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

[9] Ralf H. Reussner, Heinz W. Schmidt, and Iman H. Poernomo. Reliability prediction for component-based software architectures. *J. Syst. Softw.*, 66(3):241–252, 2003.

[10] Vibhu Saujanya Sharma and Kishor S. Trivedi. Quantifying software performance, reliability and security: An architecture-based approach. *J. Syst. Softw.*, 80(4): 493–509, 2007.

[11] Leslie Cheung, Roshanak Roshandel, Nenad Medvidovic, and Leana Golubchik. Early prediction of software component reliability. In *Proceedings of the 30th international conference on Software engineering*, pages 111–120, Leipzig, Germany, 2008. ACM.

[12] Franz Brosch, Heiko Koziolek, Barbora Buhnova, and Ralf Reussner. Architecture-based reliability prediction with the Palladio Component Model. *IEEE Transactions on Software Engineering*, 38(6):1319–1339, 2012.

[13] Vittorio Cortellessa and Vincenzo Grassi. A modeling approach to analyze the impact of error propagation on reliability of component-based systems. In *CBSE*, pages 140–156, 2007.

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

[15] Wen-Li Wang, Dai Pan, and Mei-Hwa Chen. Architecture-based software reliability modeling. *J. Syst. Softw.*, 79(1):132–146, 2006.

[16] Vittorio Cortellessa, Harshinder Singh, and Bojan Cukic. Early reliability assessment of UML based software models. In *Proceedings of the 3rd international workshop on Software and performance*, pages 302–309, Rome, Italy, 2002. ACM.

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

[18] Vibhu Saujanya Sharma and Kishor 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*, pages 299–310. IEEE Computer Society, 2006.

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

[20] J. B. Dugan and M. R. Lyu. Dependability modeling for fault-tolerant software and systems. In M. R. Lyu, editor, *Software Fault Tolerance*, pages 109–138. John Wiley & Sons, 1995.

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

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

[23] N. Sato and 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*, pages 114–121, 2007.

[24] Sherif Yacoub, Bojan Cukic, and Hany H. Ammar. A scenario-based reliability analysis approach for component-based software. *IEEE Trans. on Reliability*, 53: 465–480, 2004.

[25] Genaína Rodrigues, David Rosenblum, and Sebastian 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*, pages 111–126, Berlin, Heidelberg, 2005. Springer-Verlag.

[26] Alessandro Birolini. *Reliability Engineering: Theory and Practice*. Springer, 6th edition, 2010.

[27] Haapanen Pentti and Helminen Atte. Failure mode and effects analysis of software-based automation systems. In *VTT Industrial Systems, STUK-YTO-TR 190*, 2002.

[28] P. Lakey and A. Neufelder. System and software reliability assurance notebook. In *Rome Lab, FSC-RELI*, 2002.

[29] K. Sharma, R. Garg, C.K. Nagpal, and R. K. Garg. Selection of optimal software reliability growth models using a distance based approach. *Reliability, IEEE Transactions on*, 59(2):266–276, 2010.

[30] John Musa. *Software Reliability Engineering: More Reliable Software Faster and Cheaper*. AuthorHouse, 2nd edition, 2004.

[31] M.R. Lyu. Software reliability engineering: A roadmap. In *Future of Software Engineering, 2007. FOSE '07*, pages 153–170, 2007.

[32] Swapna S. Gokhale, W. Eric Wong, J. R. Horgan, and Kishor S. Trivedi. An analytical approach to architecture-based software reliability prediction. In *IEEE International Computer Performance and Dependability Symposium (IPDS'98)*, pages 13–22, 1998.

[33] H. Koziolek, B. Schlich, and C. Bilich. A large-scale industrial case study on architecture-based software reliability analysis. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 279–288, 2010.

[34] S. Apel. Software reliability growth prediction - state of the art. In *Fraunhofer IESE, Kaiserslautern, Research Report*, 2005.

[35] A. Beckhaus, L.M. Karg, and G. Hanselmann. Applicability of software reliability growth modeling in the quality assurance phase of a large business software vendor. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, pages 209–215, 2009.

[36] T.J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, 1976.

[37] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[38] A J Albrecht. Measuring application development productivity. In *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, pages 83–92, 1979.

[39] A. Veevers and A. Marshall. A relationship between software coverage metrics and reliability. *Software Testing, Verification and Reliability (STVR)*, 4(1):3–8, 1994.

[40] J.M. Voas and K.W. Miller. Software testability: the new verification. *Software, IEEE*, 12(3):17–28, 1995.

[41] Michael Diaz and Joseph Sligo. How software process improvement helped motorola. *IEEE Softw.*, 14(5):75–81, 1997.

[42] John E. Gaffney. Estimating the number of faults in code. *Software Engineering, IEEE Transactions on*, 10(4):459–464, 1984.

[43] B. T. Compton and C. Withrow. Prediction and control of ada software defects. *J. Syst. Softw.*, 12(3):199–207, 1990.

[44] N.E. Fenton and M. Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675–689, 1999.

[45] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346–7354, 2009.

[46] Lionel C. Briand, Khaled El Emam, Bernd G. Freimut, and Oliver Laitenberger. A comprehensive evaluation of capture-recapture models for estimating software defect content. *IEEE Trans. Softw. Eng.*, 26(6):518–540, 2000.

[47] Norman Fenton, Martin Neil, William Marsh, Peter Hearty, Łukasz Radliński, and Paul Krause. On the effectiveness of early life cycle defect prediction with bayesian nets. *Empirical Software Engineering*, 13(5):499–537, 2008.

[48] Yan Ma, Lan Guo, and Bojan Cukic. Statistical framework for the prediction of fault-proneness. In *Advances in Machine Learning Applications in Software Engineering*. Idea Group, 2007.

[49] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[50] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.

[51] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability: Measurement, Prediction, Application (Professional Ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1990.

[52] K.W. Miller, L.J. Morell, R.E. Noonan, S.K. Park, D.M. Nicol, B.W. Murrill, and J.M. Voas. Estimating the probability of failure when testing reveals no failures. *Software Engineering, IEEE Transactions on*, 18(1):33–43, 1992.

[53] D.R. Cox. *Principles of Statistical Inference*. Cambridge University Press, 2006.

[54] S. J. Prowell. JUMBL: A tool for model-based statistical testing. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*. Society Press, 2003.

[55] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 392–401, New York, NY, USA, 2005. ACM.

[56] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, 2012.

[57] Deshan Cooray, Sam Malek, Roshanak Roshandel, and David Kilgore. Resisting reliability degradation through proactive reconfiguration. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 83–92, New York, NY, USA, 2010. ACM.

[58] Marko Palviainen, Antti Evesti, and Eila Ovaska. The reliability estimation, prediction and measuring of component-based software. *J. Syst. Softw.*, 84(6):1054–1070, 2011.

[59] Kishor Shridharbhai Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications, 2nd Edition.* Wiley-Interscience, 2nd edition, 2001.

[60] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[61] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[62] Vincenzo Grassi. Architecture-based reliability prediction for service-oriented computing. In *Architecting Dependable Systems III*, volume 3549 of *Lecture Notes in Computer Science*, pages 279–299. Springer Berlin Heidelberg, 2005.

[63] Antonio Filieri, Carlo Ghezzi, Vincenzo Grassi, and Raffaela 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, pages 1–20, 2010.

[64] Zibin Zheng and Michael R. Lyu. Collaborative reliability prediction of service-oriented systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 35–44, Cape Town, South Africa, 2010. ACM.

[65] Liang Yin, M.A.J. Smith, and K.S. Trivedi. Uncertainty analysis in reliability modeling. In *Reliability and Maintainability Symposium, 2001. Proceedings. Annual*, pages 229–234, 2001.

[66] Katerina Goseva-Popstojanova and Sunil Kamavaram. Assessing uncertainty in reliability of component-based software systems. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 307–320, Washington, DC, USA, 2003. IEEE Computer Society.

[67] Simona Bernardi, Merseguer José, and Dorina C. Petriu. A dependability profile within MARTE. *Softw. Syst. Model.*, 10(3):313–336, 2011.

[68] Michael R. Lyu. *Software Fault Tolerance.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[69] Jörg Kienzle. Software fault tolerance: An overview. In *Proceedings of the 8th Ada-Europe International Conference on Reliable Software Technologies*, pages 45–67, Berlin, Heidelberg, 2003. Springer-Verlag.

[70] Y. Huang, C. Kintala, N. Kolettis, and N.D. Fulton. Software rejuvenation: analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 381–390, 1995.

[71] Neil B. Harrison and Paris Avgeriou. Incorporating fault tolerance tactics in software architecture patterns. In *Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, SERENE '08, pages 9–18, New York, NY, USA, 2008. ACM.

[72] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Journal of Systems and Software*, 80(4):528–558, 2007.

[73] Michael W. Lipton and Swapna S. Gokhale. Heuristic component placement for maximizing software reliability. In *Recent Advances in Reliability and Quality in Design*, Springer Series in Reliability Engineering, pages 309–330. Springer London, 2008.

[74] Atef Mohamed and Mohammad Zulkernine. On failure propagation in component-based software systems. In *Proceedings of the 2008 The Eighth International Conference on Quality Software*, pages 402–411. IEEE Computer Society, 2008.

[75] Andrew Hinton, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Prism: A tool for automatic verification of probabilistic systems. In *TACAS*, Lecture Notes in Computer Science, pages 441–444. Springer, 2006.

[76] Franz Brosch, Barbora Buhnova, Heiko Koziolek, and Ralf 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*, pages 75–84, Boulder, Colorado, USA, 2011. ACM.

[77] Mark H. Klein, Rick Kazman, Leonard J. Bass, S. Jeromy Carrière, Mario Barbacci, and Howard F. Lipson. Attribute-based architecture styles. In *First Working IFIP Conference on Software Architecture (WICSA1)*, pages 225–244, 1999.

[78] Rehab El-Kharboutly and Swapna S. Gokhale. Architecture-based reliability analysis of concurrent software applications using stochastic reward nets. In *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011)*, pages 635–639, 2011.

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

[80] Thomas Kappler, Heiko Koziolek, Klaus Krogmann, and Ralf H. Reussner. Towards automatic construction of reusable prediction models for component-based performance engineering. In *Proceedings of Software Engineering 2008 (SE'08)*, pages 140–154, 2008.

[81] Michael Kuperberg, Klaus Krogmann, and Ralf Reussner. Performance prediction for black-box components using reengineered parametric behaviour models. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering (CBSE 2008)*, pages 48–63, 2008.

[82] RMPI. Reliability modeling, prediction, and improvement. http://rmpi.codeplex.com/, 2014.

[83] World Wide Web Consortium (W3C). Xml schema definition. http://www.w3.org/2001/XMLSchema, 2001.

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

[85] Suntae Kim, Dae-Kyoo Kim, Lunjin Lu, and Sooyong Park. Quality-driven architecture development using architectural tactics. *J. Syst. Softw.*, 82(8):1211–1231, 2009.

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

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

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

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