Reliability Prediction for Fault-Tolerant Software Architectures

Conference Paper · January 2011

DOI: 10.1145/2000259.2000274

4 authors:

Franz Brosch
FZI Forschungszentrum Informatik
10 PUBLICATIONS **144** CITATIONS
SEE PROFILE

Barbora Buhnova
Masaryk University
79 PUBLICATIONS **696** CITATIONS
SEE PROFILE

Heiko Koziolek
ABB
102 PUBLICATIONS **2,435** CITATIONS
SEE PROFILE

Ralf H. Reussner
Karlsruhe Institute of Technology
304 PUBLICATIONS **4,066** CITATIONS
SEE PROFILE

Reliability Prediction for Fault-Tolerant Software Architectures

Franz Brosch[1], Barbora Buhnova[2], Heiko Koziolek[3], Ralf Reussner[4]
[1]Research Center for Information Technology (FZI), Karlsruhe, Germany
[2]Masaryk University, Brno, Czech Republic
[3]Industrial Software Systems, ABB Corporate Research, Ladenburg, Germany
[4]Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

brosch@fzi.de, buhnova@fi.muni.cz, heiko.koziolek@de.abb.com, reussner@kit.edu

ABSTRACT

Software fault tolerance mechanisms aim at improving the reliability of software systems. Their effectiveness (i.e., reliability impact) is highly application-specific and depends on the overall system architecture and usage profile. When examining multiple architecture configurations, such as in software product lines, it is a complex and error-prone task to include fault tolerance mechanisms effectively. Existing approaches for reliability analysis of software architectures either do not support modelling fault tolerance mechanisms or are not designed for an efficient evaluation of multiple architecture variants. We present a novel approach to analyse the effect of software fault tolerance mechanisms in varying architecture configurations. We have validated the approach in multiple case studies, including a large-scale industrial system, demonstrating its ability to support architecture design, and its robustness against imprecise input data.

Categories and Subject Descriptors

D.2.11 [**Software**]: SOFTWARE ENGINEERING—*Software Architectures*; D.2.4.g [**Software**]: SOFTWARE ENGINEERING—*Software/Program Verification—Reliability*

General Terms

Software Engineering, Reliability, Design

Keywords

Component-Based Software Architectures, Reliability Prediction, Fault Tolerance, Software Product Lines

1. INTRODUCTION

Software fault tolerance (FT) mechanisms mask faults in software systems and prohibit them to result in a failure. FT mechanisms are established on different abstraction levels, such as exception handling on the source code level, watchdog and heart-beat as design patterns, and replication on the architecture level [20, 18]. FT mechanisms are commonly used to improve the reliability of software systems (i.e., the probability of failure-free operation in a given time span). The effect of a FT mechanism (i.e., the extent of such an improvement) is however non-trivial to quantify because it highly depends on the application context.

The challenge of assessing FT mechanisms in different contexts becomes particularly apparent on the architecture level, when evaluating different architectural configurations, and even more in the design of software product lines (SPL) [7]. An SPL has a core assembly of software components and variation points where application-specific software can be filled in. Thus, it can result in many different products all sharing the same core with different configuration at the variation points.

Existing approaches for reliability prediction [11, 13, 15] either do not support modelling FT mechanisms (e.g., [5, 8, 12, 21]) or do not allow for explicit definition and reuse of core modelling artefacts, and hence are difficult to apply in varying contexts, as with architecture design and SPLs (e.g., [24, 26]).

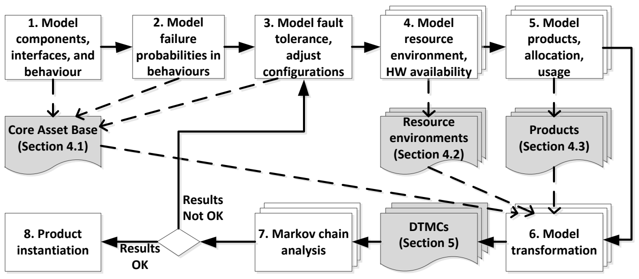
The contribution of this paper is an approach to analyse the effect of a software fault tolerance mechanism in dependence of the overall system architecture and usage profile. The approach is novel as it (i) takes software fault tolerance mechanisms explicitly into account, and (ii) reuses model parts for effective evaluation of architectural alternatives or system configurations. The approach is ideally suited for software product lines, which are used to formulate and illustrate the approach. It builds upon the Palladio Component Model and an associated reliability prediction approach [4], which includes both software reliability and hardware availability as influencing factors. Our tool support allows the architects to design the architecture with UML-like models, which are automatically transformed to Markov-based prediction models, and evaluated to determine the expected system reliability.

The remainder of this paper is structured as follows. Section 2 outlines our approach and explains the steps involved. Section 3 details the models used in our approach and then Section 4 explains how these models are formalised and analysed to predict the system reliability. Section 5 evaluates the approach on two case studies. Section 6 delimits our work from related approaches. Finally, Section 7 draws conclusions and sketches future directions.

2. PREDICTION PROCESS

This section outlines our reliability prediction approach for fault-tolerant software architectures, software families, and software product lines (SPL) in particular. According to Clements et al. [7], an SPL is defined as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way". Our modelling approach provides support only for the technical part of an SPL or software family as we assume that domain engineering and asset scoping have been performed before modelling the architecture and reusable components.

Our approach iteratively follows eight steps depicted in Fig. 1. First, the software architect creates a `CoreAssetBase`, which models interfaces, reusable software components, and their (abstract) behaviour (step 1). The `CoreAssetBase` is enriched with software failure probabilities of actions forming component (service) behaviour (step 2). Afterwards, the software architect can include different FT mechanisms, such as recovery blocks (explained in Section 3.2) or redundancy (step 3), either as additional components or directly into already modelled component behaviours. The FT mechanisms allow for different configurations, e.g., the number of retries or replicated instances.



1. Model components, interfaces, and behaviour
2. Model failure probabilities in behaviours
3. Model fault tolerance, adjust configurations
4. Model resource environment, HW availability
5. Model products, allocation, usage
Core Asset Base (Section 4.1)
Resource environments (Section 4.2)
Products (Section 4.3)
Results Not OK
8. Product instantiation
Results OK
7. Markov chain analysis
DTMCs (Section 5)
6. Model transformation

Figure 1: Process activities and artifacts

The software architect then creates a `ResourceEnvironment` to model hardware resources (step 4) and specific `Products` (step 5), including component allocation and system usage information. Combined with the `CoreAssetBase`, these models are transformed into multiple discrete-time Markov chains (step 6), from which system reliability predictions and sensitivity analyses can be deduced (step 7). If the prediction results do not satisfy the reliability requirements, the FT mechanisms can be reconfigured and/or the resource environment and products adjusted iteratively. Otherwise, the modelled products are deemed sufficient for the requirements, and the product can safely be instantiated from the core asset base (step 8). The following two sections describe the models and the predictions in detail.

3. RELIABILITY MODELLING

This section introduces our meta model for describing the reliability characteristics of software product lines, which are used to formulate our approach. The model and its reliability solver are implemented using the Eclipse Modeling Framework (EMF) and build upon the Palladio Component Model (PCM) [2]. An excerpt of our meta model is depicted in Fig. 2; for a full documentation, refer to our website [1].

[1]http://sdqweb.ipd.kit.edu/wiki/ReliabilityPrediction

The model allows to express variation points on different levels:

- architectural level: using composite components to encapsulate subsystems and enable their replacement
- component level: selecting different component implementations for a given specification
- component implementation level: using component parameters with values for specific product configurations
- resource level: using allocation references expressing different deployment schemes for product line configurations

Our model is better suited for our purposes than UML extended with the MARTE-DAM profile [3] because it allows model reuse through the core asset base and is reduced to concepts needed for the prediction. The following sections explain the core asset base (Section 3.1), the fault-tolerance mechanisms (Section 3.2), the resource environment (Section 3.3) and the product (Section 3.4) in detail.

3.1 Core Asset Base

The modelling element `CoreAssetBase` of our meta model (cf. Fig. 2) represents a repository for elements assembled into products and contains `ComponentTypes`, `Interfaces`, and `FailureTypes`. `ComponentTypes` can either be atomic `PrimitiveComponents` or hierarchically structured `CompositeComponents` with nested inner components.

Composite components allow the core asset base to contain whole architecture fragments (e.g., SPL core assemblies) that can be reused in different products. Such a core assembly can have optionally required interfaces as variation points. Component types are associated with interfaces through `ProvidedRoles` or `RequiredRoles`, and can export `ComponentParameters` that allow for implementation-level variation points. Fig. 3 shows an excerpt of an instance of a core asset base including a composite component (*4*) and a component parameter (*value*).

For reliability analyses, the model requires constructs to express the behaviour of component services in terms of using hardware resources and calling other components. Therefore, a component type can contain a number of `ServiceBehaviours` that specify the actions executed upon calling a specific `Signature` of one of the component's provided interfaces. The behaviour may consist of `InternalActions`, `ExternalCallActions`, and control flow constructs, such as probabilistic branches and loops.

An internal action represents a component-internal computation and can contain multiple `FailureOccurrences` and `ResourceDemands`. `FailureOccurrences` model software failures during the execution of a service using a probability. These failure probabilities can be determined with different techniques [11, 5, 17], such as reliability growth modelling, defect prediction based on code metrics, statistical testing, or fault injection.

An external call action represents a call to other components and thus references a `Signature` contained in one of the required interfaces of the current component. Fig. 4 shows a small example of a service behaviour containing internal actions and external call actions. Notice that the transitions of the `BranchAction` are labelled with input parameter dependencies (e.g., $P(X \leq 1)$), because the concrete values for the input parameters (e.g., X) are not known to the component developer providing the specification. The

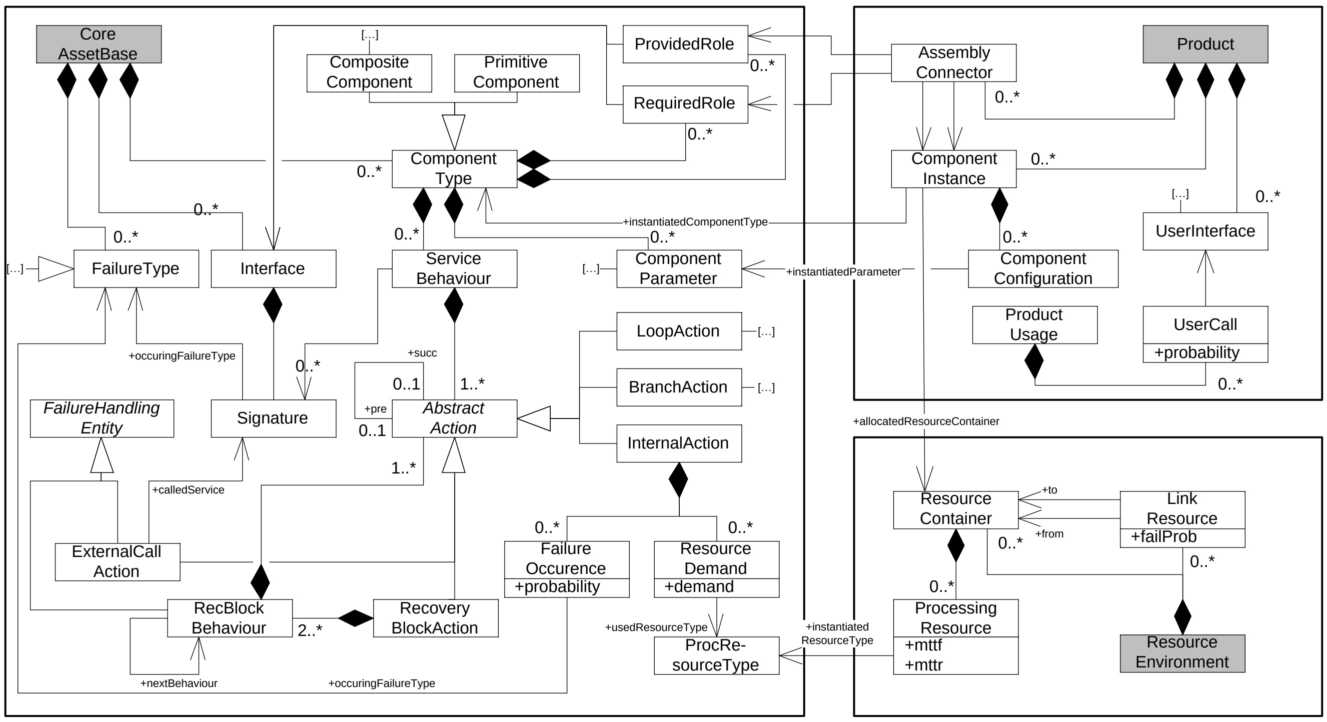
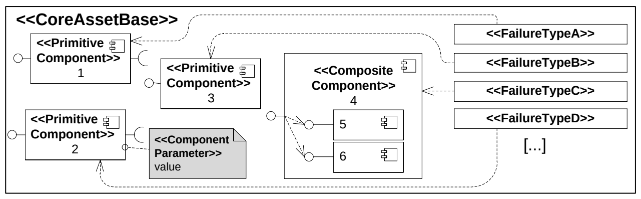


Figure 2: Excerpt of our meta model for specifying reliability characteristics in a software product line

component developer should keep the component as reusable as possible, i.e. make no assumptions about the usage profile. The developer specifies parameter dependencies as an influencing factor on the component reliability. The dependencies are automatically resolved during system analysis, when the usage profile is known. Hence, the influence of the given system-level usage profile on the system reliability is explicitly considered by our approach by propagating the system-level input parameters through the component-based architecture [4].



<<CoreAssetBase>>
<<Primitive Component>> 1
<<Primitive Component>> 3
<<Primitive Component>> 2
<<Composite Component>> 4
5
6
<<Component Parameter>> value
<<FailureTypeA>>
<<FailureTypeB>>
<<FailureTypeC>>
<<FailureTypeD>>
[...]

Figure 3: Example instance of a core asset base

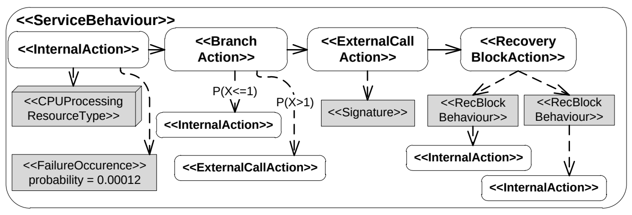


Figure 4: Service behaviour example (partial view)

3.2 Fault-tolerance Meta-Model

To model fault tolerance within component services, we use the concept of recovery blocks, which is analogical to the exception handling in object oriented programming. It is represented with a `RecoveryBlockAction` that contains an acyclic sequence of at least two `RecBlockBehaviours`. The first behaviour models the normal execution within a service, while the following behaviours handle failures of certain types and initiate alternative actions (analogically to try and catch blocks in exception handling). Each behaviour contains an inner sequence of `AbstractActions` that again can contain any type of action and even nested recovery blocks.

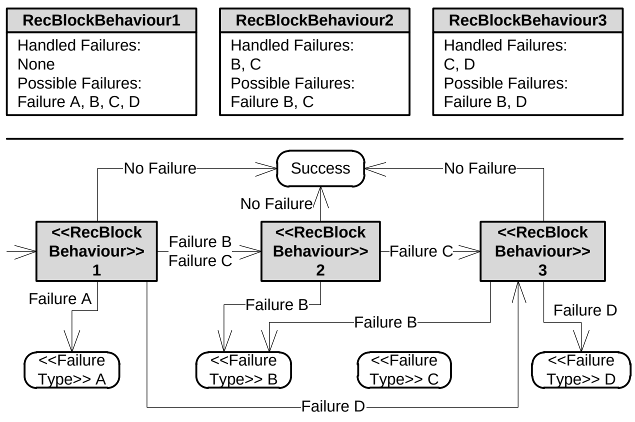


Figure 5: Recovery-block example and its semantics

Consider the example in Fig. 5 of a recovery block action with three `RecBlockBehaviours` and four different failure types A, B, C, D. During the first behaviour all failure types can occur. Failures of type A cannot be recovered and lead to a failure of the whole block. The second behaviour handles failures of type B and C and the third behaviour handles failures of type C and D. In the second behaviour, failures B and C are again possible, whereas failures of type

B lead to a failure of the block, while failures of type C and D are handled by the third behaviour. Notice that failures of type C cannot occur from this recovery block as they are handled by behaviour 3 and cannot again occur during the execution of behaviour 3.

As a `RecBlockBehaviour` can contain arbitrary behavioural constructs, such as internal actions, calls, branches, loops, and nested recovery block actions, they are a flexible means to model FT mechanisms. They allow modelling exception handling, checkpoint and restarting, process pairs, recovery blocks, or consensus recovery blocks and are therefore capable of making reliability predictions for a large class of existing FT mechanisms. If an external call action is embedded into a `RecBlockBehaviour`, errors from the called component (and any other component down the caller stack) can be handled. The case studies in Section 5 show different possible usages of recovery block actions.

3.3 Resource Environment

A `ResourceEnvironment` contains `ResourceContainers` modelling server nodes and `LinkResources` modelling network connections. Each resource container can contain a number of `ProcessingResources` like CPUs or hard disks. To include hardware availability into the calculation of the system reliability, each processing resource contains a mean time to failure (MTTF) and a mean time to repair (MTTR) attribute. These values can be determined from vendor specification and/or former experience [23]. Link resources contain failure probabilities for network problems, which can be determined using simple test series.

A resource environment model can be reused across different `Products` using allocation references (i.e. mapping of components to resources). Furthermore, it is possible to have multiple resource environments (e.g., different server sizes or high availability servers), which then constitute an additional variation point or feature of the product line. Components in the core asset base are decoupled from concrete resource environments, because they only refer to abstract `ProcResourceTypes`, but not to concrete `Processing-Resources`. Thus, it is possible to connect a product to a specific resource environment through an allocation reference without the need to alter the core asset base.

3.4 Product

A `Product` contains a number of `ComponentInstances` wired through `AssemblyConnectors` and accessed through a `UserInterface`. Only components with matching interfaces may be composed in a product. It is possible to connect different component instances complying to the same interfaces at different points in the architecture, thus implementing architecture-level variation points. Through the compositions, the overall system behaviour is defined as a connection of all service behaviours (i.e., after composition, external call actions in service behaviours can be replaced by the service behaviours of the called services).

Component instances can contain `Component-Configurations` that realize implementation-level variation points. We introduced these parameters in our former work [4]. They are data values that can model different configurations or features of a component implementation and change component behaviour.

Component instances within a `Product` must be allocated to `ResourceContainers` through an allocation reference that models a deployment on a certain server. Thus, the component reliability depends on the availability of the server's hardware resources, employed by the component instance. The `ProductUsage` contains a number of `UserCalls` with different probabilities, which model the system usage profile of a product.

4. RELIABILITY PREDICTION

This section first describes how the system reliability (under a specified usage profile) is calculated on the software layer (Section 4.1) and then integrated with calculations on the hardware and network layers (Sections 4.2, 4.3).

4.1 Software Layer

For the software layer, the approach derives the reliability from a Markov model that reflects all possible execution paths through a `Product` architecture, and their corresponding probabilities.

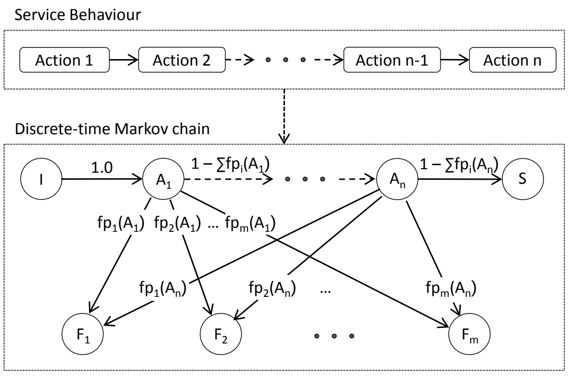


Figure 6: Markov model for service behaviours

Fig. 6 shows how a `ServiceBehaviour` represented through a sequence of n `AbstractActions` is transformed into an absorbing discrete-time Markov chain. The chain contains an initial state I, an absorbing success state S, one state A_i for each action, and one absorbing failure state F_j for each of the m user-defined software `FailureTypes`. A transition from A_i to F_j denotes that action i might exhibit a failure of type j upon execution, with a probability of $fp_j(A_i)$. The probability of failure of the whole behaviour $fp(Beh)$ is the probability to reach any of the failure states F_j (and not the success state S) from the initial state I:

$$fp(Beh) = 1 - \prod_{i=1}^{n}(1 - \sum_{j=1}^{m} fp_j(A_i))$$

For the success probability of the behaviour $sp(Beh)$, we have:

$$sp(Beh) = 1 - fp(Beh)$$

The calculation of $fp_j(A_i)$ depends on the type of the action A_i. For `InternalActions`, the failure probabilities $fp_j(A_{internal})$ are given as a direct input to the model (cf. Fig 2). `LoopActions`, `BranchActions`, `ExternalCall-Actions`, and `RecoveryBlockActions` have nested `Service-Behaviours` Beh (again sequences of `AbstractActions`), which need to be evaluated in a recursive step first. For loops, we have:

$$fp_j(A_{loop}) = \sum_{i=1}^{k}(P(c_i) \cdot \sum_{l=0}^{c_i-1}(sp(Beh)^l \cdot fp_j(Beh)))$$

with a finite set of loop iteration counts $\{c_1, \ldots, c_k\} \subseteq \mathbb{N}$, each with its probability of occurrence $P(c_i)$. For branches, we have:

$$fp_j(A_{branch}) = \sum_{i=1}^{k} (P(Beh^i) \cdot fp_j(Beh^i))$$

with a finite set of nested behaviours $\{Beh^1, \ldots, Beh^k\}$ and their probabilities of occurrence $P(Beh^i)$. An `External-CallAction` fails if the called behaviour fails:

$$fp_j(A_{call}) = fp_j(Beh)$$

Recovery blocks are characterized through a sequence of nested behaviours $[Beh^1, \ldots, Beh^k]$. Having $fp_j(Beh^i)$ for every behaviour and failure type, each Beh^i can be represented with a trivial Markov model $T(Beh^i)$, as illustrated in Fig. 7, and the recovery-block model constructed as their combination (following the semantics illustrated in Fig. 5).

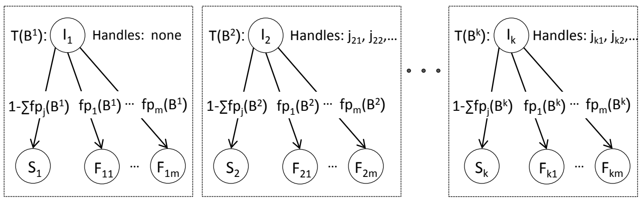

T(B¹): I₁ Handles: none
1-∑fpⱼ(B¹) fp₁(B¹) ··· fpₘ(B¹)
S₁ F₁₁ ··· F₁ₘ
T(B²): I₂ Handles: j₂₁, j₂₂,...
1-∑fpⱼ(B²) fp₁(B²) ··· fpₘ(B²)
S₂ F₂₁ ··· F₂ₘ
T(Bᵏ): Iₖ Handles: jₖ₁, jₖ₂,...
1-∑fpⱼ(Bᵏ) fp₁(Bᵏ) ··· fpₘ(Bᵏ)
Sₖ Fₖ₁ ··· Fₖₘ

Figure 7: Markov models for recovery behaviours

Let for each model $T(Beh^i)$ be I_i its initial state, S_i its success state and F_{ij} its failure state for each failure type j. To connect the isolated trees into a single Markov chain, we add $j+2$ states, namely I, S and F_j for each j, and the following transitions:

- $I \xrightarrow{1.0} I_1$,
- $S_i \xrightarrow{1.0} S$ for each $i \in \{1, \ldots, k\}$,
- $F_{ij} \xrightarrow{1.0} I_x$ where $x \in \{i+1, \ldots, k\}$ is the index of the closest tree handling failure type j if such a tree exists,
- and if not, $F_{ij} \xrightarrow{1.0} F_j$ for each F_{ij} with no outgoing transitions.

Finally, the failure probabilities $fp_j(A_{recovery})$ are computed as the probability of reaching F_j from I (via summation of the multiplied probabilities over available paths).

Based on the described calculations, the success probability of the topmost behaviour $sp(Beh_T)$ (sequence of system services invoked by the user) can be calculated in a recursive way, yielding the system-level reliability with respect to the software layer.

4.2 Hardware Layer

For `ProcessingResources`, the approach employs a hardware availability model, and integrates this model with the software layer for a combined consideration of software and hardware failures. Having the $MTTF_i$ and $MTTR_i$ values for all resources $\{r_1, \ldots, r_p\}$, we first calculate the *steady-state availability* A_i of r_i:

$$A_i = MTTF_i / (MTTF_i + MTTR_i)$$

and interpret it as the probability of r_i being available when requested at an arbitrary point in time. Furthermore, we define the set of *physical system states* as the set $\{s_1, \ldots, s_q\}$ where each s_j is a combination of the individual resource states. We distinguish two possible states for each resource, being either available (OK) or unavailable ($FAIL$). Assuming independent hardware failures, the probability $P(s_j)$ for the system to be in state s_j is the product of the individual state probabilities. For example, for a system with 2 resources, the probability for r_1 being OK and r_2 being $FAIL$ is:

$$P((r_1 = OK) \wedge (r_2 = FAIL)) = A_1 \cdot (1 - A_2)$$

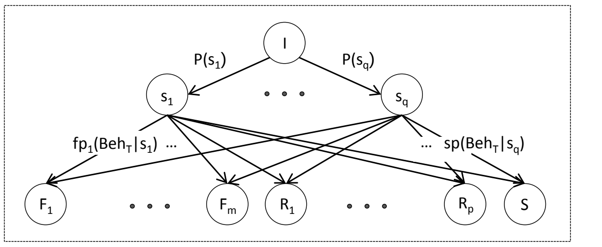


Figure 8: Combined HW/SW consideration

Fig. 8 illustrates the combined consideration of the software and the hardware layer. Upon a user call, the system might be in any of the physical system states s_j. This is reflected by a transition from the initial state I to s_j with the corresponding state probability $P(s_j)$. Being in s_j, the execution might either fail due to an unavailable hardware resource accessed by system control flow (s_j to R_i), or with a software failure (represented through transitions from s_j to F_k), or it might succeed (s_j to S).

To calculate the failure probabilities $fp_k(Beh_T|s_j)$, $k \in \{1, \ldots, m+p\}$, we need to incorporate the failures due to hardware unavailability into the software layer model (as shown in Fig. 6). To this end, we set the hardware-caused failure probability $fp_k(Beh_T|s_j)$, $k > m$, of `Internal-Actions` to 1.0 if they require a `ProcessingResource` that is not available under the given physical system state s_j. If an internal action requires two or more unavailable resources, the failure probability is distributed evenly among these. In this manner, the software layer is evaluated independently for each physical system state, and the overall success probability of service execution (represented by its topmost behaviour Beh_T) is computed as the weighted sum over the success probabilities of all physical system states:

$$sp(Beh_T) = \sum_{j=1}^{q} (P(s_j) \cdot sp(Beh_T|s_j))$$

where for each physical system state we have:

$$sp(Beh_T|s_j) = 1 - \sum_{k=1}^{m+p} fp_k(Beh_T|s_j)$$

Thanks to the separation of service behaviour according to the individual physical system states, the approach can better reflect the correlation of subsequent resource demands (i.e. a resource accessed twice within a single service execution is likely to be either OK in both cases or $FAIL$ in both cases), caused by significantly longer resource failure and repair times compared to the execution time of a single user-triggered service [22].

4.3 Network Layer

To incorporate the network layer into the model, we assume that each call routed over a `LinkResource` involves two message transports (request and return), and that each transport might fail with the given failure probability of the link $fp(L)$. We adapt the software layer model (see Section 4.1) by a differentiation of `ExternalCallActions` into local calls and remote calls. We keep the failure probabilities for local calls:

$$fp_j(A_{localCall}) = fp_j(Beh)$$

but incorporate the link failure probability into the calculation for remote calls:

$$fp_j(A_{remoteCall}) = fp_j(Beh) \cdot fp(L)^2$$

Thus, we enable coarse-grained consideration of network reliability, without going into the details of more sophisticated network simulations, which are out of scope of this paper.

5. EVALUATION

5.1 Goals and Setting

This section serves to validate our reliability and fault tolerance modelling approach. The goals of the validation are (i) to demonstrate how the new FT modelling techniques can support design decisions, (ii) to provide a rationale for the validity of our models and their resilience to imprecise input parameters, and (iii) to show the effectiveness of our models for SPLs.

Regarding goal (ii), validating reliability prediction against measured values is inherently difficult, as failures are rare events, and the necessary time to observe a statistically relevant number of failures is infeasibly long for high-reliability systems. Several existing approaches therefore limit their validation to demonstrating examples and sensitivity analyses (e.g., [8, 10, 12, 22, 25]), showing how the approaches can be used to learn about system failure behaviour, and proving the robustness of prediction results against imprecise input data at design time. A number of authors involve real-world industrial software systems in their validation (e.g., [5, 16, 26]). We follow the same path, and additionally compare our numerically calculated predictions against a more realistic, but also more time-consuming queueing network simulation [4], in order to at least partially validate prediction accuracy. The simulation has fewer assumptions than the analytical solution. It takes system execution times (encoded into `ResourceDemands`) into account and lets resources fail and be repaired according to their MTTF and MTTR, not based on the simplified steady-state availability.

We have validated our approach on a number of different systems: a distributed business reporting system, the common component modelling example (CoCoME), a web-based media store product line [2], an industrial control system product line [17], and the SLA@SOI Open Reference Case. The models for these systems can be retrieved from our website [1]. In the following, we describe the predictions for the web-based media store (Section 5.2) and the industrial control system (Section 5.3) in detail. The media store allows us to present all reliability predictions, while the predictions for the industrial control system have been obfuscated for confidentiality reasons.

5.2 Case Study I: Web-based Media Store

The media store model is inspired by common web-service-based data storage solutions and has similar functionality to the ITunes Music Store [2]. The system provides a centralized storage for media files, such as audio or video files, and a corresponding up- and download functionality. The media store product line contains three standard product configurations: *standard*, *comfort*, and *power*. More configurations are possible by instantiating the feature model in Fig. 9.

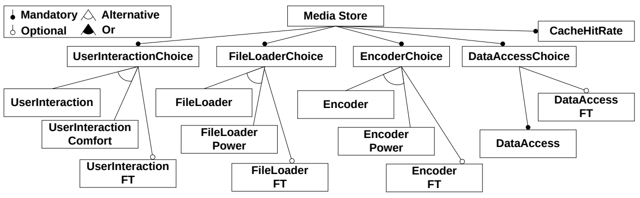
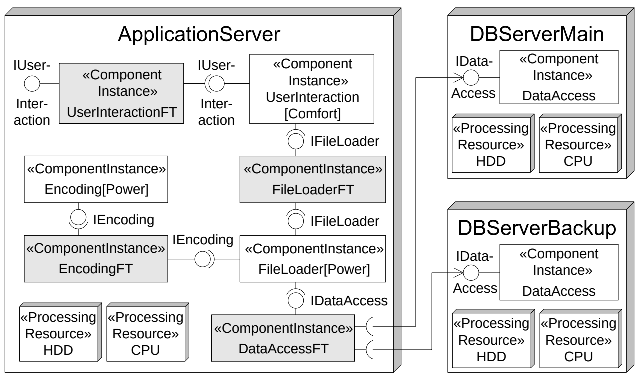


Figure 9: Feature model of the media store SPL

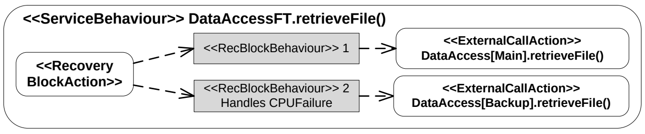
Fig. 10 summarizes the different products and design alternatives of the media store product line. The core functionality is provided through four component types: `User-Interaction`, `FileLoader`, `Encoding`, and `DataAccess`. For some of these components alternative comfort or power variants are present in the core asset base for the different product variants (cf. Fig. 9). The database and the `DataAccess` component are deployed on one (or optionally two) separated database server(s).



ApplicationServer
IUser-Inter-action
«Component Instance» UserInteractionFT
IUser-Inter-action
«Component Instance» UserInteraction [Comfort]
IFileLoader
«ComponentInstance» Encoding[Power]
«ComponentInstance» FileLoaderFT
IEncoding
IFileLoader
«ComponentInstance» EncodingFT
IEncoding
«ComponentInstance» FileLoader[Power]
IDataAccess
«Processing Resource» HDD
«Processing Resource» CPU
«ComponentInstance» DataAccessFT
DBServerMain
IData-Access
«Component Instance» DataAccess
«Processing Resource» HDD
«Processing Resource» CPU
DBServerBackup
IData-Access
«Component Instance» DataAccess
«Processing Resource» HDD
«Processing Resource» CPU

Figure 10: Media store products and design alternatives

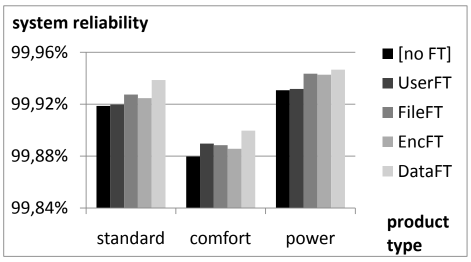
During up- and download of media files, different types of failures may occur in the involved component instances: A *BusinessLogicFailure* may occur during the processing of user requests in the `UserInteraction[Comfort]` component. A *CacheAccessFailure* may occur in the `FileLoader[Power]` component induced by malfunctioning cache memory. Bugs in the compression algorithm of the `Encoder[Power]` component may lead to an *EncodingFailure*. A *DataAccessFailure* may occur in the `DataAccess` component due to internal database errors or faults in the database server's file system. Additionally, as hardware failures, a *CommunicationFailure*, *CPUFailure*, and/or *HDDFailure* can occur.



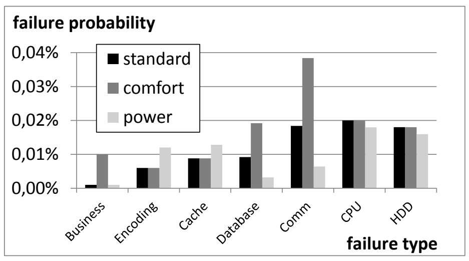
<<ServiceBehaviour>> DataAccessFT.retrieveFile()
<<Recovery BlockAction>>
<<RecBlockBehaviour>> 1
<<RecBlockBehaviour>> 2 Handles CPUFailure
<<ExternalCallAction>> DataAccess[Main].retrieveFile()
<<ExternalCallAction>> DataAccess[Backup].retrieveFile()

Figure 11: Service behaviour of DataAccessFT

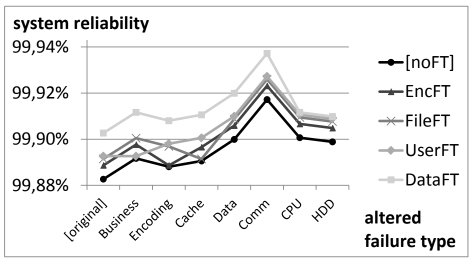
For illustrative purposes, we set the software-level failure probabilities to 10^{-5} for each individual failure occurrence in the model, with the following exceptions distinguishing the



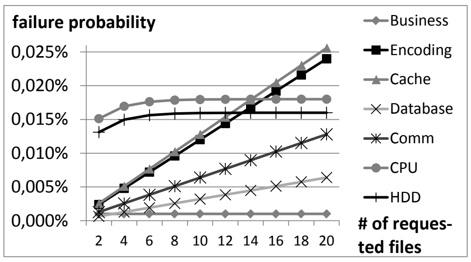
(a) System reliability for all design alternatives



(b) Failure probabilities per failure type



(c) System reliability changes for altered input data



(d) Failure probabilities depending on usage profile

Figure 12: Media store prediction results

products: in the `UserInteractionComfort` component, the probability of *BusinessLogicFailures* rises to 10^{-4} because of the more complex business logic compared to the standard variant. Compression algorithms are generally complex and may fail with a probability of 10^{-4} in the `Encoder` component, and with 2×10^{-4} in the `EncoderPower` component. For all hardware resources, we assume the MTTF being one year and the MTTR 50 minutes, implying an availability of 99.99%. In other settings, these values could have been extracted from log files of existing similar systems.

Fault tolerance mechanisms may be optionally introduced into each media store product, in terms of additional components which are shown in grey in Fig. 10. For example, a `UserInteractionFT` component may be put in front of the `UserInteraction[Comfort]` component. It has the ability to buffer incoming requests, to re-initialise the business logic in case of a *BusinessLogicFailure*, and to retry the failed request. As another example, the `DataAccessFT` component may be used to handle *CPUFailures* on the main DB server by redirecting calls to the backup server. Fig. 11 illustrates the service behaviour of the file retrieval call, which consists of a single `RecoveryBlockAction` with two `RecBlock-Behaviours`.

Each described fault tolerance mechanism can be used for each product, and more than one mechanism may be applied in parallel. We focus on cases where at most one mechanism is used.

The usage profile of the media store consists of 20% upload calls and 80% download calls, an average of 10 requested files per call, and a probability of 0.3 for files to be large, i.e. requiring compression during upload. We calculated the expected system reliability for each product and design alternative. Each calculation took below one second on a standard PC with a 2.2 GHz CPU and 2.00 GB RAM.

To provide evidence about the possible decision support for different design alternatives (goal (i) in Section 5.1), Fig. 12(a) shows the system reliability for each product and fault tolerance alternative. The comfort product has the lowest reliability, because of the included statistics functionality, which involves additional computing and requests to the database for storage and retrieval. The power product has the highest system reliability, as the high hit rate in the `FileLoaderPower` cache decreases the number of necessary database accesses. Employing the `DataAccessFT` component has the highest effect compared to the design alternatives without fault tolerance. Notice that the FT mechanisms have different influences in the different variants. For example, the `UserInteractionFT` is most effective for the comfort variant.

Fig. 12(b) provides more detail and shows the probability of a system failure due to a certain failure type. Summarized over all products, *CommunicationFailures*, *CPUFailures* and *HDDFailures* most probably cause a system failure. The risk of a *CommunicationFailure* is especially high for the comfort product, which requires many database accesses and corresponding network traffic. Thus, a software architect may recognize the need to introduce new fault tolerance mechanisms for these failures.

To demonstrate the robustness of our model to imprecise input data (goal (ii) in Section 5.1), we first examined the robustness of the reliability prediction to alterations in the input failure probabilities. We changed the failure probabilities of the components of the comfort product one at a time by multiplying them with 10^{-1}. We also increased the hardware resource availability 99.99% to 99.999% one at a time for each resource. Fig. 12(c) shows new system reliabilities for the different fault tolerance variants, indicating that the ranking of the design alternatives is almost identical over the different failure type alterations. The `DataAccessFT` is always top-ranked. However, rank changes do occur in case of altered *BusinessLogicFailure* and *CacheAccessFailure* probabilities, indicating that these probabilities should be estimated as careful as possible.

As a second sensitivity analysis, Fig. 12(d) focuses on the power product without fault tolerance. It shows the sensitivity of the failure probabilities per failure type to the number of requested files (i.e., a change to the usage profile). For most failure types, the failure probabilities rise with the number of files, as more database accesses and message transports over network are required, as well as more file compressions and cache accesses. The business logic, however, is independent of the number of files, which keeps the *BusinessLogicFailure* probability constant. *CPUFailures* and *HDDFailures* do not influence the system reliability for more than 8 files. For cases with 2 to 8 files there is a chance that all files are found in the cache, and no database access is necessary, thus lowering the failure probability.

To analyse prediction accuracy, we ran the reliability simulation for each of the three products in the `UserInteractionFT` variant for 10^7 simulated seconds, and got a deviation from the analytical results between 0.0006% and 0.0067%. Fig. 13 shows the results. The ranking of the three considered variants is confirmed by the simulation, which indicates that the analytical results are sufficiently accurate.

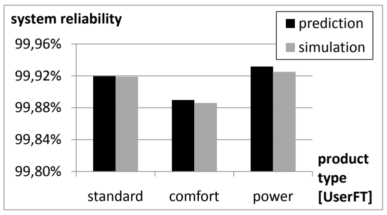


Figure 13: Media store simulation results

The effectiveness of the approach for SPLs (goal (iii) in Section 5.1) is demonstrated by the fact that nearly all model parts can be reused throughout the media store products and design alternatives. Only some `ComponentInstances` are specific to certain alternatives and need to be connected via additional `AssemblyConnectors`: the `UserInteractionComfort`, `EncodingPower`, `FileLoaderPower`, and all FT components.

5.3 Case Study II: Industrial Control System

As a second case study, we analysed the reliability of a large-scale industrial control system from ABB, which is used in many different domains, such as power generation, pulp and paper handling, or oil and gas processing. The system is implemented in several millions lines of C++ code. On a high abstraction level, the core of the system consists of eight software components that can be flexibly deployed on multiple servers depending on the system capacity required by customers. Fig. 14 depicts a possible configuration of the system with four servers. The names of the components and their failure probabilities have been obfuscated for confidentiality reasons.

The upper part of Fig. 14 shows the `ServiceBehaviours` for the components `C7` and `FT2`. The components `FT1` and `FT2` have been introduced into model inspired by existing FT mechanisms. `FT1` is able to restart component `C1` upon failed requests. `FT2` is able to query two redundant instances of component `C4`, which are deployed on different servers, thereby implementing fault tolerance against hardware failures.

The reliability of the core system has been analysed in a former study [17], where no fault tolerance mechanisms or product variants were considered. For this case study, we reused the failure probabilities from the former study, which had been determined using software reliability growth models based on bug tracker data. We also reused the transition probabilities between the components, which had been

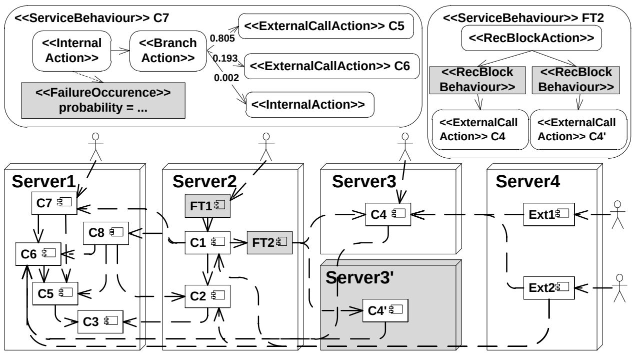


Figure 14: Control system product line model with two exemplary service behaviours

determined using source code instrumentation and system execution. The hardware reliability parameters were based on vendor specifications.

The industrial control system is realized as a product line and sold to customers in different variants depending on their requirements. Fig. 15 shows a small excerpt of variants in terms of a feature model. There are many more possible variants, as third party components can be integrated into the system via standardized interfaces. The components `C1`-`C8` are mandatory. For component `C4`, there are two alternative implementations ($C4_1$ and $C4_2$), which address different customer requirements. There are two external components `Ext1` and `Ext2`, which can be optionally included into the core system. The feature model also includes the different FT mechanisms as variants.

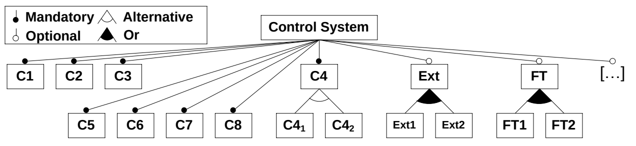
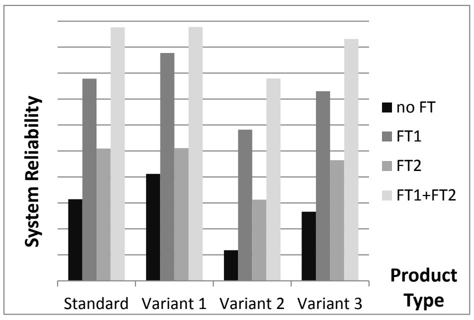


Figure 15: Feature model of the control system product line variants (excerpt)

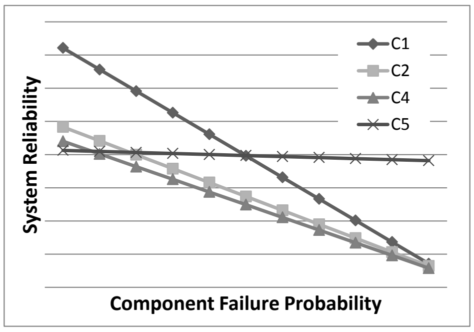
For the scope of this paper we restricted the reliability analysis to the core system (standard) and three different variants. Variant 1 uses component $C4_2$ instead of $C4_1$ but is otherwise identical to the core system. Variant 2 incorporates the external component `Ext1`, which is only connected to component `C4` (cf. Fig. 14). Variant 3 incorporates component `Ext2`, which is connected to component `C1`, `C2`, `C4`, and `C6`. These variants correspond to realistic configurations, which have formerly been sold to customers.

To demonstrate the decision support for different alternatives (goal (i) in Section 5.1) we analysed how the predicted system reliability varies for the different variants and FT mechanisms (Fig. 16(a)). The actual values are obfuscated for confidentiality reasons. Variant 1 is the predicted as being the most reliable. Introducing `FT1` generally bears a higher increase in reliability than introducing `FT2`, which includes adding an additional server for the redundant instance of component `C4`. The impact on system reliability of `FT2` is less pronounced for variant 1 than for the other variants, because it already uses a higher reliable version of component `C4`. Thus, the software architect can decide whether the increased costs for adding an additional server for realising `FT2` in this variant are justified.

To show the robustness of the models against imprecise



(a) Prediction results for different variants



(b) Sensitivity to component failure probabilities

Figure 16: Exemplary prediction results for the industrial control system

input parameters (goal (ii) in Section 5.1), we conducted a sensitivity analysis modifying the failure probabilities of selected components (Fig. 16(b)). The system reliability is most sensitive to the component reliability of `C1` as the curve has the steepest slope. The system reliability is robust to the component reliability of `C5`. Overall, the model behaves linearly and the deviations of the system reliability are comparably small to changes in individual component failure probabilities. In this case, the ranking of the design alternatives remained robust against uncharacteristically high variations of the component failure probabilities.

For a comparison between numerically computed predictions and simulation data, we ran a simulation for each variant for 10^6 simulated seconds. The mean error between the numerically computed and simulated system reliability across all variants was 0.0077 percent. The ranking of the variants remained was the same for the simulation results as for the numerical results. We conclude that the numerical calculations were sufficiently precise in this case.

To show the effectiveness of the approach for SPLs (goal (iii) in Section 5.1), we quantified the amount of changes necessary to model the product variants in our case. For Variant 1, a single `ComponentInstance` and `AssemblyConnector` had to be added to the standard `Product` and deployed to the respective `ResourceContainer`. This did not require the adjustment of transition probabilities. For Variants 2 and 3, also only single `ComponentInstances` had to be added to the standard `Product`.

6. RELATED WORK

Our method for architectural fault tolerance modelling is related to approaches on software architecture reliability modelling [13, 21], fault tolerance modelling on the level of software architecture [18], and reliability engineering for software product lines [7].

Multiple surveys on *software architecture reliability modelling* are available [11, 13, 15]. R.C. Cheung [6] was among the first to propose architectural reliability modelling using Markov chains. Some recent approaches refine such models to support compositionality [21], and different failure modes [10], but do not regard fault tolerance mechanisms. L. Cheung et al. [5] use hidden Markov models to determine component failure probabilities for Markov chain architecture models. Further approaches in this area apply the UML modelling language [8, 12] or are specifically tailored to service-oriented systems [27], but also do not include fault tolerance mechanisms or support for reusing model artefacts in different contexts, such as product configurations.

Some approaches do tackle the problem of incorporating *fault tolerance* mechanisms into the architectural prediction models [18]. Sharma and Trivedi [25] includes additional states and recovery transitions into architecture level Markov model to model component restarts or system reboots. Wang et al. [26] provides constructs for Markov chains to model replicated components. Kanoun et al. [16] model fault tolerance of hardware/software systems using generalized stochastic Petri nets. These approaches do not consider component-internal control and data flow, and how it is influenced by error handling constructs. Thus, they may yield inaccurate predictions when fault tolerant software behaviour deviates from the specific cases considered by the authors. Furthermore, none of these approaches supports reusing model artefacts.

Considering reliability during the design of a *software product line* is a major challenge, because different product variants may have different influences on the expected reliability. Immonen [14] proposes the 'reliability and availability prediction' (RAP) method for SPLs. RAP, however, does not support compositional models, hardware reliability, or explicit fault tolerance mechanisms. Olumofin et al. [19] tailor the architecture trade-off analysis method to evaluate SPLs for different quality attributes. They focus on the identification of scenarios but provide no architectural model or predictions. Dehlinger et al. [9] introduce the PLFaultCAT tool to analyse SPL safety using fault tree analysis. Their models do not reflect the software architecture and therefore complicate evaluating different design alternatives. Auerswald et al. [1] model product families of embedded systems using block diagrams, but provide no usage profile model or quantitative reliability prediction.

7. CONCLUSIONS

We presented an approach to support the design of reliable and fault-tolerant software architectures and software families. Our approach allows modelling different architectural alternatives and product line configurations from a shared core asset base and offers a flexible way to include many different fault tolerance mechanisms. A tool transforms the models into Markov chains and calculates the system reliability involving both software and hardware reliabilities. We evaluated our approach in multiple case studies and demonstrated its value to support architectural design decisions, its robustness against imprecise input data, and its effectiveness for SPLs.

Our approach provides a new perspective for designing software architectures and families. It allows software architects to validate their designs during early development stages and supports their design decisions quantitatively. As the effectiveness of different fault tolerance mechanisms is highly context dependent, our approach enables software architects to quickly analyse many different alternatives and rule out poor design choices. This can potentially lead to more reliable systems, which are built more cost-effectively because late life-cycle changes for better reliability can be avoided.

In future work, we aim to include more sophisticated hardware reliability modelling techniques into our approach to offer more refined predictions. We will extend our tool for automated sensitivity analyses and design optimisation. Our prediction approach can potentially be extended for other quality attributes, such as performance or security.

8. ACKNOWLEDGMENTS

This work was supported by the European Commission as part of the EU-projects SLA@SOI (grant No. FP7-216556) and Q-ImPrESS (grant No. FP7-215013). Furthermore, we thank Igor Lankin for his support in developing the approach.

9. REFERENCES

[1] M. Auerswald, M. Herrmann, S. Kowalewski, and V. Schulte-Coerne. *Software Product-Family Engineering*, volume 2290 of *LNCS*, chapter Reliability-Oriented Product Line Engineering of Embedded Systems, pages 237–280. Springer, 2001.

[2] S. Becker, H. Koziolek, and R. Reussner. The Palladio Component Model for Model-Driven Performance Prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[3] S. Bernardi, J. Merseguer, and D. Petriu. A dependability profile within MARTE. *Software and Systems Modeling*, pages 1–24, 2009.

[4] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner. Parameterized Reliability Prediction for Component-based Software Architectures. In *Proc. of QoSA'10*, volume 6093 of *LNCS*, pages 36–51. Springer, 2010.

[5] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik. Early prediction of software component reliability. In *Proc. of ICSE'08*, pages 111–120. ACM Press, 2008.

[6] R. C. Cheung. A User-Oriented Software Reliability Model. *IEEE Trans. Softw. Eng.*, 6(2):118–125, 1980.

[7] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2001.

[8] V. Cortellessa, H. Singh, and B. Cukic. Early reliability assessment of UML based software models. In *Proc. of WOSP'02*, pages 302–309. ACM, 2002.

[9] J. Dehlinger and R. R. Lutz. PLFaultCAT: A Product-Line Software Fault Tree Analysis Tool. *Automated Software Engineering*, 13(1):169–193, 2006.

[10] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola. Reliability Analysis of Component-Based Systems with Multiple Failure Modes. In *Proc. of CBSE'10*, volume 6092 of *LNCS*, pages 1–20. Springer, 2010.

[11] S. S. Gokhale. Architecture-Based Software Reliability Analysis: Overview and Limitations. *IEEE Trans. on Dependable and Secure Computing*, 4(1):32–40, 2007.

[12] 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 Trans. on Softw. Eng.*, 29(10):946–960, 2003.

[13] K. Goseva-Popstojanova and K. S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2-3):179–204, 2001.

[14] A. Immonen. *Software Product Lines*, chapter A Method for Predicting Reliability and Availability at the Architecture Level, pages 373–422. Springer, 2006.

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

[16] K. Kanoun and M. Ortalo-Borrel. Fault-tolerant system dependability-explicit modeling of hardware and software component-interactions. *IEEE Transactions on Reliability*, 49(4):363–376, 2000.

[17] H. Koziolek, B. Schlich, and C. Bilich. A Large-Scale Industrial Case Study on Architecture-based Software Reliability Analysis. In *Proc. 21st International Symposium on Software Reliability Engineering (ISSRE'10).* IEEE Computer Society, 2010. To appear.

[18] H. Muccini and A. Romanovsky. Architecting Fault Tolerant Systems. Technical Report CS-TR-1051, University of Newcastle upon Tyne, 2007.

[19] F. G. Olumofin and V. B. Misic. Extending the ATAM Architecture Evaluation to Product Line Architectures. In *Proc. of WICSA'05*, pages 45–56. IEEE Computer Society, 2005.

[20] B. Randell. System structure for software fault tolerance. In *Proc. Int. Conf. on Reliable software*, pages 437–449. ACM, 1975.

[21] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 66(3):241–252, 2003.

[22] N. Sato and K. S. Trivedi. Accurate and efficient stochastic reliability analysis of composite services using their compact Markov reward model representations. In *Proc. of SCC'07*, pages 114–121. IEEE Computer Society, 2007.

[23] B. Schroeder and G. A. Gibson. Understanding disk failure rates: What does an MTTF of 1,000,000 hours mean to you? *ACM Trans. Storage*, 3(3):8, 2007.

[24] V. Sharma and K. Trivedi. Quantifying software performance, reliability and security: An architecture-based approach. *Journal of Systems and Software*, 80:493–509, 2007.

[25] V. S. Sharma and K. S. Trivedi. Reliability and Performance of Component Based Software Systems with Restarts, Retries, Reboots and Repairs. In *Proc. of ISSRE'06*, pages 299–310. IEEE, 2006.

[26] W.-L. Wang, D. Pan, and M.-H. Chen. Architecture-based software reliability modeling. *Journal of Systems and Software*, 79(1):132–146, 2006.

[27] Z. Zheng and M. R. Lyu. Collaborative reliability prediction of service-oriented systems. In *Proc. of ICSE'10*, pages 35–44. ACM Press, 2010.