# Architecture-based Reliability Prediction with the Palladio Component Model

Franz Brosch, Heiko Koziolek, *Member, IEEE Computer Society*, Barbora Buhnova, Ralf Reussner, *Member, IEEE, IEEE Computer Society*

**Abstract**— With the increasing importance of reliability in business and industrial software systems, new techniques of architecture-based reliability engineering are becoming an integral part of the development process. These techniques can assist system architects in evaluating the reliability impact of their design decisions. Architecture-based reliability engineering is only effective if the involved reliability models reflect the interaction and usage of software components and their deployment to potentially unreliable hardware. However, existing approaches either neglect individual impact factors on reliability or hard-code them into formal models, which limits their applicability in component-based development processes.

This paper introduces a reliability modelling and prediction technique that considers the relevant architectural factors of software systems and explicitly models the component usage profile and execution environment. The technique offers a UML-like modelling notation, whose models are automatically transformed into a formal analytical model. Our work builds upon the Palladio Component Model, employing novel techniques of information propagation and reliability assessment. We validate our technique with sensitivity analyses and simulation in two case studies. The case studies demonstrate effective support of usage profile analysis and architectural configuration ranking, together with the employment of reliability-improving architecture tactics.

**Index Terms**—D.2.11 Software architectures; D.2.10.h Quality analysis and evaluation; D.2.5.h Reliability; D.2.2 Design tools and techniques.

## 1 Introduction

Software-intensive systems are increasingly used to support critical business and industrial processes, such as in business information systems, e-business applications, or industrial control systems. The reliability of a software system is defined as the probability of failure-free operation of a software system for a specified period of time in a specified environment [1]. To manage reliability, reliability engineering gains its importance in the development process. Reliability is compromised by faults in the system and its execution environment, which can lead to different kinds of failures during service execution: *software failures* occur due to faults in the implementation of software components, *hardware failures* result from unreliable hardware resources, and *network failures* are caused by message loss or problems during inter-component communication.

To support fundamental design decisions early in the development process, architecture-based reliability prediction [2], [3], [4], [5] can be employed to evaluate the quality of system design, and to identify reliability-critical elements of the architecture. Existing approaches suffer from the following drawbacks that limit their applicability and accuracy.

First, many approaches do not explicitly model the influence of the system usage profile (i.e., sequences of system calls and values of parameters given as an input to these calls) on the control and data flow throughout the architecture, which in turn influences reliability. For example, if faulty code is never executed under a certain usage profile, no failures occur, and the system is perceived as reliable by its users. Existing models encode a system usage profile implicitly into formal models, typically in terms of transition probabilities in the Markov models characterizing the execution flow among components (e.g., [6], [7], [8], [9]). Since the models are tightly bound to the selected usage profile, evaluating reliability for a different usage profile requires repeating much of the modelling effort.

Second, many approaches do not consider the reliability impact of a system's execution environment. Even if the software is totally free of faults, failures can occur due to unavailability of underlying hardware resources and communication failures across network links. Neglecting these factors tends to result in less accurate and over-optimistic reliability prediction. On the other hand, approaches that do consider the execution environment (e.g., [10], [11]), typically offer no means to model application-level software failures, which also results in a limited view of software system reliability.

Third, many approaches use Markov models as their modelling notation (e.g., [8], [9], [10], [12]), which

- *F. Brosch is with FZI Forschungszentrum Informatik, Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany. E-mail: brosch@fzi.de*
- *H. Koziolek is with ABB Corporate Research (Industrial Software Systems), Wallstadter Str. 59, 68526 Ladenburg, Germany. E-mail: heiko.koziolek@de.abb.com*
- *B. Buhnova is with Masaryk University, Botanicka 68a, 60200 Brno, Czech Republic. E-mail: buhnova@fi.muni.cz*
- *R. Reussner is with Karlsruhe Institute of Technology (KIT), Kaiserstr. 12, 76131 Karlsruhe, Germany. E-mail: reussner@kit.edu*

is not aligned with concepts and notations typically used in software engineering (e.g., UML or SysML). Although the approaches provide an implicit mapping of Markov states to software components (or their internal behavioural states), they do not explicitly deal with other concepts of the software engineering domain (such as interface descriptions incl. input parameters, connectors, provided/required services, modularity, hierarchical structure of composite components, etc.). Instead, they represent the system through a low-level set of states and transition probabilities between them, which obscures the original software-engineering semantics. Direct creation and interpretation of Markov models without any intermediate notation may be uncomfortable and hard to accomplish for software developers, especially when it is to be done repeatedly during the development process.

The contribution of this paper is a novel technique for architecture-based software reliability modelling and prediction that explicitly considers and integrates the discussed reliability-relevant factors. The technique offers usage-profile separation and propagation through the concept of *parameter dependencies* [13] and accounts for hardware unavailability through reliability evaluation of service execution under different hardware availability states. We realize the approach as an extension of the Palladio Component Model (PCM) [14], which offers a UML-like modelling notation. We provide tool support for an automated transformation of PCM models into Markov chains and space-effective evaluation of these chains. We discuss how software engineers can use architecture tactics to systematically improve the reliability of the software architecture. Furthermore, we validate the approach in two case studies.

The rest of the paper is organized as follows: Section 2 discusses existing approaches for architecture-based software system reliability prediction. Section 3 explains the most important concepts of the PCM and introduces the PCM meta-model extension for reliability. Section 4 describes the transformation of PCM models to Markov chains, and the evaluation of the Markov model. Afterwards, Section 5 discusses architectural improvements for reliability, and their representation in the PCM. The case studies in Section 6 serve as a validation of our approach, where the results of the analysis are compared with values obtained from a reliability simulation, and the capabilities of the approach are demonstrated with sensitivity analyses. Finally, we consider assumptions and limitations of our work in Section 7, and summarize our work in Section 8.

## 2 Related Work

Seminal work in the area of software reliability engineering [1], [15] focuses on system tests and reliability growth models treating systems as black boxes. Recently, several architecture-based reliability analysis approaches have been proposed [2], [3], [4], [5], [16] treating systems as a composition of software components. In the following, we examine the most related architecture-based approaches with respect to our goals (i.e., the three gaps identified above). After the summary of the findings, we discuss our preliminary work and its relation to this paper.

### 2.1 Usage Profile Modelling

System usage can be described in terms of the expected sequences of system calls (including their likelihood) and the values of input parameters used for the calls, which may influence the control flow throughout the system.

In many existing approaches, the system usage profile is not modelled explicitly, but encoded implicitly into transition probabilities between the states or scenarios of the system model [17], [18]. Goseva et al. [2] state that most approaches rely on estimations of transition probabilities. Cheung [6] mentions that the transition probabilities could be obtained by assembling and deploying the components and executing the expected usage profile against them. However, this requires software engineers to set up the whole system during architecture design, which is often neither desired nor possible.

Recent approaches by Wang et al. [9] and Sharma et al. [8] extend Cheung's work to support different architectural styles and combined performance and reliability analysis. They still rely on testing data or the software architect's intuition to determine the transition probabilities. The work of Reussner et al. [16] can be seen as a precursor of the here presented approach as it models explicitly the influence of external components. However, the approach assumes fixed transition probabilities between components, therefore its models cannot be reused if the system-level usage profile changes. Cheung et al. [19] focus on the reliability of individual components and do not include calls to other components.

Several approaches suggest that accurate transition probabilities may be hard to get, and they design methods to include uncertainties in the models [5], [20], [21], [22]. Roshandel et al. [5] suggest to treat uncertainties by designing appropriate constraints in the system, implemented as pre/post conditions, guards, and other types of assertions. Fiondella et al. [20] use confidence intervals to incorporate uncertainties about the architectural and component parameters (including uncertain operational profile) into the analysis.

Scenario-based approaches, employing message sequence charts (MSC) or sequence diagrams (SD) as their modelling notation, provide an implicit capacity to model the details of system usage profiles (since their notation is well suited for describing usage scenarios). However, the existing architecture-based reliability approaches (e.g., [17], [18], [23]) focus purely on the call sequences, not the input value propagation, which is an important factor of system usage. Similarly, Hamlet's approach [24] allows component developers to specify

the call propagation of individual components. However, the dependency of the call propagations to input parameter values is not made explicit.

### 2.2 Execution Environment Modelling

Many approaches do not consider the influence of the execution environment on the reliability of a software system (e.g., [6], [16]). However, some approaches have proposed to include the properties of the execution environment (such as failures of application containers, virtual machines, hardware devices, and communication links) into software reliability models.

Sharma et al. [11] provide a software performability model incorporating hardware availability and different states of hardware resources. The approach calculates the throughput of successful requests in presence of hardware failures, but not the system reliability. Other approaches provide complex availability models of the execution environment (e.g., [25], [26]), but do not link it to the software level to quantify the overall system reliability.

Popic et al. [27], Yacoub et al. [18], and Lipton et al. [28] take failure probabilities of network connections into account, but neglect the availability of other hardware devices, such as processors. Sato and Trivedi [10] combine a system model (of interacting system services) with a resource availability model. However, they do not consider application-level software failures. Malek et al. [29] focus on mobile systems and consider aspects of the execution environment that are especially relevant for these systems (such as network fluctuations, available battery charge, and changes in location).

Some authors use fault injection techniques to simulate hardware defects when determining system reliability [30]. Huang et al. [31] proposed a simulation-based approach to assess the reliability of a system modeled in a hardware description language and included two software segments. Das et al. [32] extended the layered queuing network model (LQN) to include failure and repair rates for hardware nodes. Based on this combined software and hardware model, they conducted availability predictions. However, none of these approaches targets component-based software architectures.

### 2.3 Modeling Notations and Tool Support

Most of the existing approaches use some kind of Markov model (DTMC or CTMC) to conduct reliability predictions. Often, the models are created directly in the Markov-model notation [8], [9], [10], [12], which may discourage software architects not familiar with formal notations.

Some approaches (e.g., [17], [18], [23], [27], [33]) use a high-level notation based on UML sequence and deployment diagrams annotated with reliability properties, such as failure probabilities. Tools can transform such models into Markov models, which then can be evaluated by existing Markov chain solvers. The aim of these approaches is that software developers can quickly enhance existing design specifications in UML to conduct reliability predictions. Furthermore, the complexity of the underlying analysis techniques stays hidden from developers.

However, many of these approaches focus only on selected architectural aspects (such as the component-internal behaviour or component interaction in case of scenario-based approaches [17], [18]). The additional architectural constructs (e.g., composite-component structure or deployment for the mentioned) are included only implicitly, if at all, which hinders the evaluation of their effect on system reliability during the architecture design process.

Moreover, the tool support for existing approaches is still sparse. Cortellessa et al. [23] sketch a possible transformation from UML diagrams into Markov models, but provide no tool support. Goseva et al. [33] use UML sequence diagrams and Markov models in their approach and mention that the implementation of a tool would be straightforward. We argue that an implementation based on UML is not straightforward due to semantic gaps and ambiguities in the language, and means non-trivial effort for the development team.

Yacoub et al. [18] construct component dependency graphs (CDG) from sequence diagrams (SD) manually. Rodrigues et al. [17] discuss a possible transformation from message sequence charts (MSC) to Markov models, and outline its implementation in [34], but do not make the implementation publicly available . Popic et al. [27] extend the ECRA tool for reliability analysis, so that it accepts UML use case, sequence, and deployment diagrams. However, the tool is not aligned with a component-based development process, and does not allow component developers to create reliability specifications of their components independently from each other.

| Paper | Year | Software Failures | Hardware Failures | Parameter Dependencies | Language/Notation | Tool Support |
|---|---|---|---|---|---|---|
| Cheung et al. [6] | 1980 | ✓ | ✗ | ✗ | MM | ✗ |
| Cortellessa et al. [23] | 2002 | ✓ | (✓) | ✗ | UML | ✗ |
| Gokhale et al. [12] | 2002 | ✓ | ✗ | ✗ | MM | ✗ |
| Reussner et al. [16] | 2003 | ✓ | ✗ | ✗ | MM | ✗ |
| Goseva et al. [33] | 2003 | ✓ | ✗ | ✗ | UML | (✓) |
| Grassi [35] | 2004 | ✓ | ✗ | ✗ | MM | ✗ |
| Yacoub et al. [18] | 2004 | ✓ | (✓) | ✗ | SBM | ✗ |
| Rodrigues et al. [17] | 2005 | ✓ | ✗ | ✗ | SBM | (✓) |
| Popic et al. [27] | 2005 | ✓ | (✓) | ✗ | UML | (✓) |
| Wang et al. [9] | 2006 | ✓ | ✗ | ✗ | MM | ✗ |
| Sharma et al. [11] | 2006 | ✓ | ✓ | ✗ | MM | ✗ |
| Sato et al. [10] | 2007 | (✓) | (✓) | ✗ | MM | ✗ |
| Sharma et al. [8] | 2007 | ✓ | ✗ | ✗ | MM | ✗ |
| Cheung et al. [19] | 2008 | ✓ | ✗ | ✗ | MM | ✗ |
| Lipton et al. [28] | 2008 | ✓ | (✓) | ✗ | MM | ✗ |
| Hamlet [24] | 2009 | ✓ | ✗ | (✓) | (MM) | (✓) |

TABLE 1
Modeling Reliability in Architecture-based Approaches

### 2.4 Summary of Findings

Table 1 summarizes our findings regarding existing approaches for architecture-based software reliability analysis. A check mark in parenthesis means that an approach partially supports the feature. Software failures (mostly specified through failure probabilities) are supported by almost all approaches, while hardware failures (e.g., due to unavailability of hardware devices) are supported only by a few approaches. Parameter dependencies reflecting the influence of a system usage profile on the control and data flow throughout the architecture via input propagation are hardly recognized by other approaches. Most approaches rely on Markov models (MM) as a primary modelling notation. Some approaches employ scenario-based models (SBM), such as message sequence charts or sequence diagrams, and some take advantage of a combination of more UML diagrams. Tool support for automated transformation from a high-level notation (if existent) to Markov chains, as well as the evaluation of the Markov chains, is very limited.

### 2.5 Preliminary Work

The PCM as a meta-model and tool suite for performance prediction has been published in [14]. The published version of PCM is not related to reliability. It neither includes meta-model constructs for reliability, nor a transformation to a Markov model and evaluation of the latter. However, we reuse the existing meta-model for component-based software architectures, including parameter dependencies for usage profile propagation (see Section 3).

In [16], we presented an approach to calculate the reliability of a component-based software architecture based on the *Rich Architecture Definition Language* (RADL). The approach neither considers parameter dependencies nor failures of the execution environment (hardware and network). Tool support for automated transformation from RADL to Markov chains is not given.

In [13], we focused on parameter dependencies for usage profile propagation throughout a component-based software architecture. We used *Stochastic Regular Expressions* (SRE) as a modelling notation, which have substantial limitations compared to the PCM. We did not consider failures of the execution environment. Our validation was limited to sensitivity analysis, while in this paper, we also perform a reliability simulation.

Furthermore, this paper goes beyond our work published in [36] through extended support for concurrent and stateful systems, a more extensive validation, a definition of a process of its application, and a far more elaborate detailed description and discussion of the approach.

## 3 Modelling Reliability with the PCM

To make the reader familiar with the foundations of our modelling approach, this section introduces the PCM modelling capabilities, and the extensions we provide for reliability. Our PCM overview starts with a simple example in Section 3.1, followed with a more detailed description of the modelling capabilities in Section 3.2. Finally, Section 3.3 introduces our extensions to the PCM regarding reliability modelling.

### 3.1 Example

Fig. 1 shows an example of a PCM instance, modeling a simple library system. It allows for a book search that browses either the library repository (in case of new books), or a number of archives sorted by their relevance (in case of older books).

The PCM model is divided into four individual parts, each corresponding to a certain *developer role*. The roles may contribute their parts independently from other roles, supporting a distributed component-based development process through equally distributed modelling contributions. The roles envisioned by PCM are the *component developer*, *software architect*, *system deployer*, and *domain expert*.

In the example, *component developers* specify *service behaviours* of two *components* (resulting in three *component service behaviour models*). Each model specifies a single *service* provided by the modelled component, to capture the high-level control and data flow of the service implementation. *Software architects* compose the components into an *architectural model* by specification of component wiring (connectors). *System deployers* define a resource environment (e.g., CPUs, network links) and allocate the components in the architectural model to the resources, building the *deployment model* of the system. Finally, *domain experts* specify the system-level *usage model* in terms of stochastic call sequences and input parameter values.

### 3.2 PCM Architectural Modelling Capabilities

This section gives a more detailed description of the individual PCM models (without reliability extensions). For a complete description, including the meta-model, refer to [14].

#### 3.2.1 Component Service Behaviour Models

This part of the PCM model includes behavioural specifications of component *provided services*, together with their *input parameters* and associated *component parameters*.

Service behaviour is given in terms of a *service effect specification (SEFF)*, which abstractly models the usage of required services by the provided service (i.e., external calls), and the consumption of resources during component-internal processing (i.e., internal actions). SEFFs may include probabilistic or value-guarded branches, loops, and forks, to model the control and data flow within the component service. To guard the control-flow constructs, input and component parameters can be used to form the guard expressions.

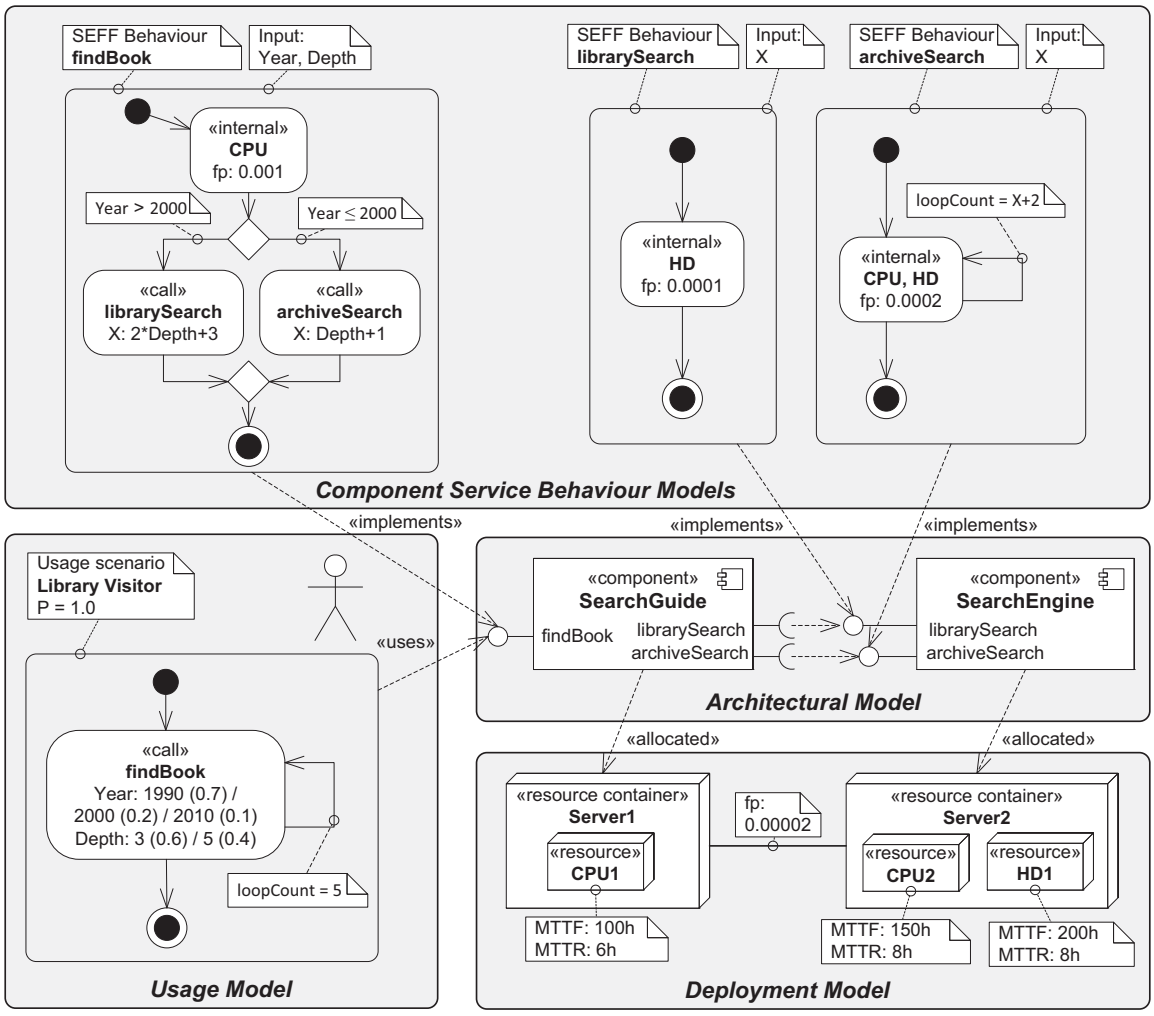



Fig. 1. PCM Example (Overview)

Input parameters are an important means to propagate user-input information and customize system behaviour accordingly. Component parameters are used to set component's internal state, and to use its value to configure component's behaviour. In the current PCM version, the values of component parameters cannot be changed at run time. Although component parameters can be defined as random variables to reflect possible state values, this limitation makes PCM support for stateful modelling only partial. The full stateful modelling capacity is however going to be part of the next PCM version, as already designed in [37].

The example in Fig. 1 consists of three SEFF models (delimited with rounded rectangles), each modelling a single provided service of a component used in the architectural model. The leftmost service behaviour model (SEFF) in Fig. 1 specifies the only provided service of the `SearchGuide` component, i.e. the `findBook` service. The `findBook` SEFF specifies the high-level control and data flow as follows. After execution start, the internal action (stereotype ≪internal≫) represents preprocessing of the search query, using a CPU resource type. Subsequently, a branch either leads to a synchronous call (stereotype ≪call≫) to `librarySearch`, if the book has been published after the year 2000, or to `archiveSearch` otherwise. The publication year is given in the `Year` parameter of the `findBook` service (propagated from the usage model). Besides `Year`, `findBook` has a parameter `Depth`, which is used to compute parameter `X` propagated further to the called services. After returning from the external call, `findBook` finishes with the stop action. The remaining two SEFFs in Fig. 1 specify the two provided services of the `SearchEngine` component. The `librarySearch` service consists of only one internal action representing a simple search algorithm (using a HD resource type). The `archiveSearch` service executes a more complex search (using CPU and HD resource types) iteratively over a list of archives, with the number of iterations influenced by the propagated `X` (resp. `Depth`) parameter.

Notice that in all cases of our example, an internal action represents a substantial block of functionality. The only rule is that a computation cannot be abstracted into a single internal action if it contains service calls reaching outside the current component. Moreover, some computations can be completely abstracted away from

the model, if they are not interesting from the reliability point of view. Similarly, component developers are free to decide on the abstraction level of the interface description (input parameters, data types). However, the more information is provided, the higher accuracy of the analysis can be expected.

At design time, developers model SEFFs manually. After implementation, developers may apply static code analysis [38] or execute the component against different test cases [39] to derive SEFFs. For this paper, we assume that the composition of SEFFs always results in valid component communication as constraints on valid call sequences (i.e., interface protocols) can be checked before our reliability analysis with our former work on interoperability checking [40].

#### 3.2.2 Architectural Model

The components—as specified through their respective component service behaviour models—are connected by a software architect into an *architectural model* of the system. Furthermore, software architects define the system boundaries and determine the provided interfaces that shall be exposed to system users or other systems.

In Fig. 1, the architectural model connects the `SearchGuide` and `SearchEngine` components through their required and provided `librarySearch` and `archiveSearch` services, with the `findBook` service being exposed to external system users.

#### 3.2.3 Deployment Model

The *deployment model* defines the system's resource environment, consisting of a set of resource containers (i.e., physical computing nodes) connected via network links. Each resource container may include a number of hardware resources (e.g., CPU, hard disk, memory, etc.). System deployers specify concrete resources, while component SEFFs only refer to abstract resource types. When specifying the allocation of components to resource containers, the resource demands can be directed to concrete resources. This method allows to easily exchange the resource environment in the model without the need to adapt component specifications.

The resource environment of Fig. 1 consists of two resource containers `Server1` and `Server2` that are connected through a network link. The resource containers provide CPU and HD (hard disk) resources that may be used by the `SearchGuide` and `SearchEngine` components, corresponding to the specified component allocation.

#### 3.2.4 Usage Model

The *usage model*, provided by domain experts, captures the system's usage profile and consists of a set of *usage scenarios* representing different user classes or use cases of the system. Each scenario specifies sequences of calls to system services, including probabilistic control flow constructs (e.g., branches or loops) to express existing variabilities within each user class or use case. If a service signature contains input parameters, the domain experts may characterize their values and other properties (for example, the number of elements in a collection). They can use the *stochastic expression language* to model parameter properties with arbitrary probability distributions [41].

Fig. 1 contains a simple usage model, consisting of a single usage scenario. Each system user (human or other system) arrives at the system, enters a series of 5 invocations of the `findBook` service, and leaves the system. Each call to `findBook` has the same probability distribution for input parameters `Year` (1990, 2000, or 2010) and `Depth` (3 or 5).

Our approach supports both analysing single usage scenarios in isolation as well as multiple parallel usage scenarios. In the latter case, a probability for the invocation of each usage scenario must be specified. The reliability analysis then determines the overall system reliability weighted by the usage scenario probabilities. A formal definition of the PCM usage model is available in [41, pp.116].

### 3.3 Model Extensions for Reliability

In this section, we provide rationale for using the PCM as a basis for our approach, describe the new concepts introduced in the PCM, and summarize the usage of this information for reliability prediction.

#### 3.3.1 The PCM for Reliability Prediction

In this paper, we reuse the PCM meta-model originally designed for performance modelling and prediction, and incorporate the notion of *software failures*, *communication link failures*, and *unavailable hardware*.

It would have been possible to build our approach upon the widely accepted UML2 meta-model instead of the PCM. However, by choosing the PCM, we avoid the complexity and the semantic ambiguities of UML2, which make it hard to provide automated transformations from UML2 to analysis models. Still, the PCM tooling (see Section 4.4) allows for graphical, UML-like modelling. For a more detailed discussion of PCM versus UML2, see [14].

#### 3.3.2 Software Failures

Software failures occur during service execution due to faults in the implementation. In a PCM behavioural specification (i.e., SEFF), all component-internal processing is abstracted into *internal actions*. Hence, each internal action can be annotated with a *failure probability*, describing the probability that the action fails while being executed. Techniques for determining these values have been discussed extensively in the literature (see Section 7 for more details) and are beyond the scope of this paper.

Notice that software components do not behave randomly and, in contrast to hardware, fail systematically.

However, because non-trivial software components usually cannot be tested exhaustively, the failure probabilities express the uncertainty about the component correctness due to the practically limited number of test cases. Furthermore, they model the uncertainty of correct component operation in the context of complex activation patterns (so-called *Mandelbugs* [42]) during run time, which can hardly be accounted for during testing.

Fig. 1 shows the specification of an internal action of the `findBook` service with a failure probability of 0.001, as well as internal actions of `librarySearch` and `archiveSearch` with failure probabilities 0.0001 and 0.0002. Section 7.1 lists some methods from literature on how to determine these failure probabilities. Furthermore, one of our former papers [43] showed how these values were defined for a large industrial software system.

#### 3.3.3 Communication Link Failures

Communication link failures, specified by system deployers, include loss or damage of messages during transport, which results in a service failure. Though transport protocols like TCP include mechanisms for fault tolerance (e.g., acknowledgement of message transport and repeated message sending), failures can still occur due to overload, physical damage of the transmission link, or other reasons. As such failures are generally unpredictable from the point of view of the system deployer, we treat them like software failures and annotate communication links with a failure probability in the PCM model. System deployers can define these failure probabilities either from experience with similar systems or by running tests on the target network.

Fig. 1 contains one network link as part of the deployment model. It is annotated with a failure probability of 0.00002.

#### 3.3.4 Unavailable Hardware

Unavailable hardware causes a service execution to fail. Hardware resource breakdowns mainly result from wear-out effects. Typically, a broken resource (e.g., a CPU, memory, or storage device) is eventually repaired or replaced with a functionally equivalent new resource. In the PCM, system deployers annotate hardware resources with their *Mean Time To Failure* (MTTF) and *Mean Time To Repair* (MTTR) values. The MTTF values can often be found in specification documents of hardware vendors, and can be refined by system deployers on experience [44]. This way, the MTTF values can be used to implicitly reflect the expected hardware usage intensity (faster hardware aging), its resistance to failure (hardware fault tolerance mechanisms, like replication) or other factors.

The resource environment in Fig. 1 contains three resources situated within `Server1` and `Server2`, annotated with the MTTF and MTTR values. The `CPU1` of `Server1` is used by the internal action of `findBook` (because the `SearchGuide` component is allocated to this resource container). Likewise, `CPU2` and `HD1` of `Server2` are used by internal actions of `librarySearch` and `archiveSearch` services (implemented by the `SearchEngine` component). Notice that internal actions can fail due to multiple reasons: either because of a software failure, or because of a required hardware resource being unavailable.

#### 3.3.5 System Reliability

Taking into account the three types of failures described above, we assume that the execution of a system usage scenario fails if (i) a software failure happens during component-internal processing, (ii) a communication link fails in propagating a call between two components, or (iii) a hardware resource is unavailable when required by service execution. Our goal is to predict the *probability of successful execution* (PSE) of the usage scenarios given by the PCM instance. The aimed probability of successful execution (PSE) under a given usage scenario is the direct counterpart of the *Probability of Failure on Demand* (POFOD): $PSE = 1 - POFOD$.

The combined consideration of the software, hardware and network dimensions enables the reflection of their interplay in the context of the overall architecture and system usage profile:

- Unavailable hardware resources and failing communication links only impact the system's reliability if they are actually required by the service execution.
- Software faults only lead to failures if the respective parts of the implementation are actually executed under a given usage profile.
- The failure potential of all three dimensions depends on the control and data flow through the architecture, which is captured by the component behaviour specifications (Section 3.2.1) and the architectural model (Section 3.2.2).

Only an integrated analysis can provide an accurate view on the relations between all dimensions (also see the sensitivity analyses conducted in Section 6.4).

## 4 Predicting Reliability with the PCM

Once a full PCM instance is specified by combining the models contributed by all developer roles (as described in Section 3), we can predict its reliability in terms of the *probability of successful execution* $PSE = 1 - POFOD$ of the given usage scenarios. The prediction process is depicted in Fig. 2, together with the preceding modelling and subsequent design assessment steps. The prediction part starts with the *e. PCM instance* input and finishes with the *h. System reliability* output. In between, the process requires solving parameter dependencies, i.e. turning all parameters in the model into their system-usage implied probability distributions (step 5, Section 4.1), and interconnecting possible sources of failure into an analytical approach quantifying system-level reliability (steps 6–9).

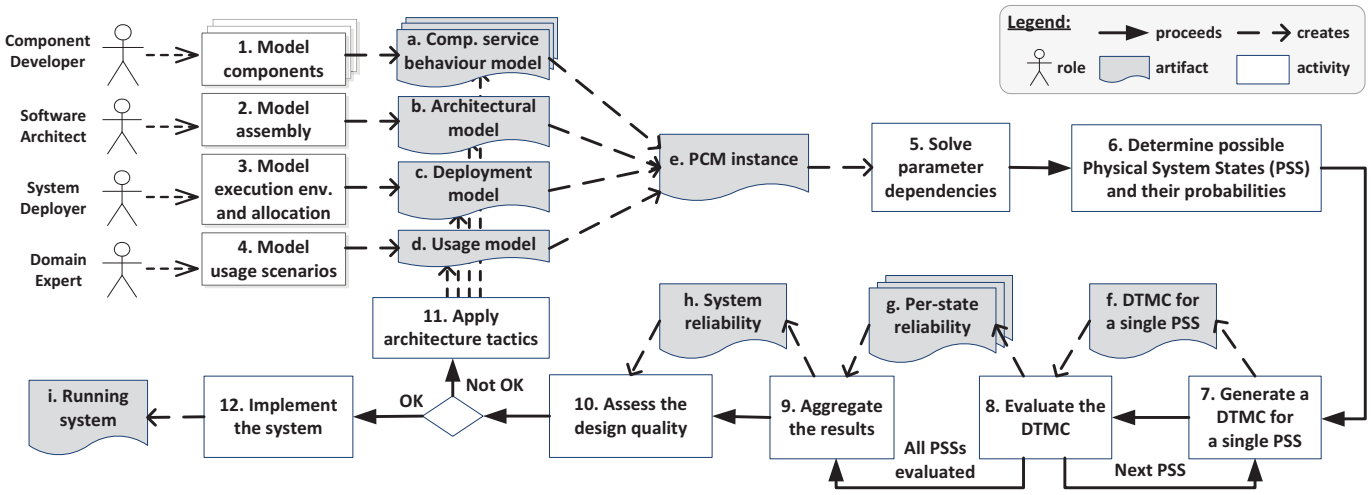



Fig. 2. Reliability Engineering Process

When modelling the propagation of different failure sources, the approach stresses the condition that an unavailable hardware resource causes a (system-level) failure only if it is accessed during its unavailability state. To integrate this condition without building a timed model, we characterize each resource with the probability of being available, based on its MTTF and MTTR, and employ the probability in determining resource state during execution. To make the computation more realistic while keeping the model untimed, the resource state is assumed to be fixed along the execution of a single usage scenario (see the discussion of this assumption in Section 4.3.3 and its validation in Section 6.6). This is reflected with separate consideration of system behaviour under each possible state of all resources (i.e. physical system state). Hence the technique first determines possible physical system states together with their probabilities (step 6, Section 4.2) and later generates (step 7) and evaluates (step 8) Markov chains individually for each physical system state, with results aggregation (step 9) at the end (Section 4.3).

### 4.1 Solving Parameter Dependencies

As described in Section 3.2, component developers specify the high-level behaviour of their components through service effect specifications (SEFFs). These SEFFs may contain *parameter dependencies* to express the influence of input parameter values on the control and data flow. For example, consider the branch within the `findBook` service in Fig. 1 – either the `librarySearch` or `archiveSearch` service is called, depending on the value of the input parameter `Year`. The concrete branch probabilities can only be derived through the probability distribution of `Year`, which is given as part of the usage scenario. The influence of input parameter values on the control and data flow propagates across SEFF boundaries. In the example, the input values `X` of the `librarySearch` and `archiveSearch` services depend on the input value `Depth` of `findBook` service.

As the example shows, the probabilities to invoke `librarySearch` and `archiveSearch` services from the `findBook` service depend (i) on the input parameter values of the `findBook` service, and (ii) on the control and data flow within `findBook`. However, these two aspects belong to different developer roles (domain expert and component developer), where each role knows only about one aspect. Manual estimation of transition probabilities between services and components, as done in related work (see Section 2.1), mixes both aspects and thus violates the independence of developer roles, leading to weak support for a distributed component-based development process.

To resolve all parameter dependencies throughout a PCM instance (step 5 in Fig. 2), we reuse the existing PCM *Dependency Solver*. We only sketch the solution, for a detailed description of the algorithm see [41]. Starting from the given usage scenario, the Dependency Solver traverses the specified SEFFs recursively and resolves all parameter dependencies on its way. Each resolving step includes parsing and resolving a stochastic expression with arbitrary probability distributions, references to input parameters with different data types, and different kinds of operators (compare, boolean, and arithmetic operators).

In Fig. 1, the algorithm starts with the usage scenario containing the probability distributions for `Year` and `Depth` parameters (which are the same for each of the five calls to the `findBook` service). The results, namely all resolved expressions, are shown in Table 2.

### 4.2 Determining Probabilities of Physical System States

After solving the parameter dependencies, the next step involves determining all possible *physical system states* and their probability of occurrence (step 6 in Fig. 2). Thereby, a physical system state is composed of all individual states of the system's hardware resources,

| Expression | Original | Resolved |
|---|---|---|
| `usage scenario`: loop iteration count | $count = 5$ | — |
| `findBook`: input parameter `Year` | $P(Year{=}1990){=}0.7$ $P(Year{=}2000){=}0.2$ $P(Year{=}2010){=}0.1$ | — |
| `findBook`: input parameter `Depth` | $P(Depth{=}3) = 0.6$ $P(Depth{=}5) = 0.4$ | — |
| `findBook`: left branch probability | $prob = P(Year > 2000)$ | $prob = 0.1$ |
| `findBook`: right branch probability | $prob = P(Year \leq 2000)$ | $prob = 0.9$ |
| `librarySearch`: input parameter `X` | $X = 2 * Depth + 3$ | $P(X{=}9) = 0.6$ $P(X{=}13) = 0.4$ |
| `archiveSearch`: input parameter `X` | $X = Depth + 1$ | $P(X{=}4) = 0.6$ $P(X{=}6) = 0.4$ |
| `archiveSearch`: loop iteration count | $count = X + 2$ | $P(count{=}6) = 0.6$ $P(count{=}8) = 0.4$ |

TABLE 2
Solved Dependencies

which are defined in the PCM resource environment and allocated to resource containers.

Let $R = \{r_1, r_2, .., r_n\}$ be the set of resources in the system. Each resource $r_i$ is characterized by its $MTTF_i$ and $MTTR_i$ and has two possible states $OK$ and $NA$ (not available). Within our approach, we do not use the specified $MTTF_i$ and $MTTR_i$ values directly for reliability prediction. Instead, we calculate the *steady-state availability* $Av$ of resource $r_i$:

$$Av(r_i) = MTTF_i/(MTTF_i + MTTR_i)$$

We interpret $Av(r_i)$ as the probability that the resource is available when required by an internal action during service execution. The value of $Av(r_i)$ only depends on the ratio between $MTTF_i$ and $MTTR_i$. Multiplying both values with the same factor $x \cdot MTTF_i$ and $x \cdot MTTR_i$ yields the same value for $Av(r_i)$.

Let $t$ be an arbitrary point in time (during system run time ), and let $s(r_i, t)$ be the state of resource $r_i$ at time $t$. Then, we have:

$$P(s(r_i, t) = OK) = Av(r_i)$$

$$P(s(r_i, t) = NA) = 1 - Av(r_i)$$

This calculation effectively ignores the concrete point in time $t$ and just assumes the system to be in its steady state. As a future work, the calculation may be refined using continuous-time Markov chains (CTMCs) and transient analysis (see [11]).

Let $S$ be the set of possible physical system states, that is, $S = \{s_1, s_2, .., s_m\}$, where each $s_j \in S$ is a unique combination of possible states of all $n$ resources at time $t$:

$$s_j = (s_j(r_1, t), s_j(r_2, t), .., s_j(r_n, t)) \in \{OK, NA\}^n$$

As each resource has two possible states, there are $2^n$ physical system states, that is, $m = 2^n$. Let $P(s_j, t)$ be the probability that the system is in state $s_j$ at time $t$. Assuming independent resource failures, the probability of each physical system state is the product of the individual resource-state probabilities:

$$\forall j \in \{1, .., m\} : P(s_j, t) = \prod_{i=1}^{n} P(s(r_i, t) = s_j(r_i, t))$$

Considering the example in Fig. 1, there are $n = 3$ resources included in the model (two CPUs and one HD). We can calculate the steady-state availability of each resource, for example:

$$Av(CPU1) = 100/(100 + 6) \approx 0.943396$$

There are $m = 2^3 = 8$ physical system states, whose probability of occurrence is the product of the individual resource state probabilities. For example, let $s$ be the physical system state with CPU1 and CPU2 being $OK$ and HD1 being $NA$ at time $t$:

$$s := (OK, OK, NA)$$

Then, we have:

$$P(s, t) = Av(CPU1) \cdot Av(CPU2) \cdot (1 - Av(HD1))$$

Our evaluation method involves determining all physical system states and their probabilities of occurrence (step 6 in Fig. 2), which are used for reliability evaluation (steps 7–9 in Fig. 2, Section 4.3).

### 4.3 Generating and Evaluating the Markov Model

Based upon a PCM instance with solved parameter dependencies and known physical system states with probabilities of occurrence, our approach generates and evaluates absorbing discrete-time Markov Chains (DTMCs) in a recursive manner, in order to predict system reliability. The algorithm has two main parts: First, an individual DTMS generation (step 7 in Fig. 2) and evaluation (step 8 in Fig. 2) takes places per physical system state (Section 4.3.1). Second, all individual results are aggregated to gain the final result (step 9 in Fig. 2, Section 4.3.2). In the following, we describe both parts and conclude with a discussion of the complexity of the algorithm.

#### 4.3.1 Per-State DTMC Generation and Evaluation

The probability of successful execution of the given usage scenario is calculated under the assumption of a given physical system state $s_j$. To this end, behavioural specifications of the PCM instance (usage scenario behaviour and SEFFs) are transformed into Markov chains according to the scheme illustrated in Fig. 3.

First, each behavioural specification $B$ (usage scenario or SEFF) is represented as a linear sequence of actions $\{A_1, A_2, \ldots, A_n\}$, including internal actions, call actions, branches, loops, and forks, with a nested semantics. In this semantics, the whole block of behaviour belonging to a branch, loop or fork is represented with a single action, having nested behaviours. In the example in

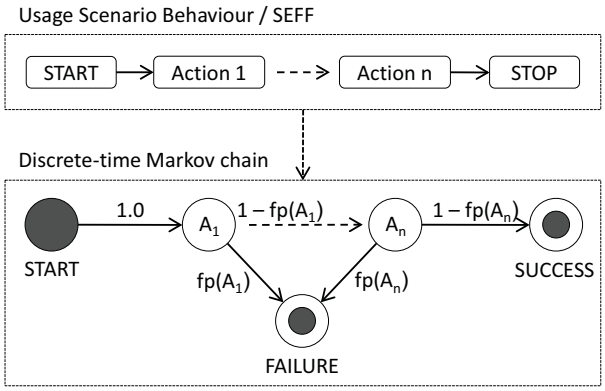



Fig. 3. Markov Chain Generation

Fig. 1, the `findBook` behaviour would be represented as a sequence of two actions – the internal and the branch action – where the branch action would have two nested behaviours, each again modelled with a sequence of actions.

After adding a START and a STOP action to the sequence, this specification is transformed into a DTMC such that each action of the behaviour becomes a state of the DTMC. The START action becomes the initial DTMC state; the STOP action a SUCCESS state. Additionally, a FAILURE state is introduced to express that any action $A_i$ can fail with probability $fp(A_i)$. The resulting DTMC is absorbing and acyclic. The failure probability $fp(B)$ of the overall behaviour is the probability to reach the FAILURE state from the initial state:

$$fp(B) = 1 - \prod_{i=1}^{n}(1 - fp(A_i))$$

To determine $fp(B)$, each $fp(A_i)$ must be calculated, which in turn depends on type of the action $A_i$. In the following, we describe the calculation of $fp(A_i)$ for the different kinds of actions that may occur in the behaviour.

A loop action $A_{loop}$ has exactly one nested behaviour $N$ with failure probability $fp(N)$, which has to be calculated in a recursive step first. The loop contains a specification of loop iteration counts as a random variable over a finite domain of iteration counts $\{c_1, c_2, \ldots, c_k\} \subseteq \mathbb{N}_0$, each assigned a probability $P(c_i)$ of its occurrence. For the loop failure probability $fp(A_{loop})$, we have:

$$fp(A_{loop}) = 1 - \sum_{i=1}^{k}(P(c_i) \cdot (1 - fp(N))^{c_i})$$

A branch action $A_{branch}$ has a finite set of nested behaviours $\{N_1, N_2, \ldots, N_k\}$ with failure probabilities $fp(N_i)$, which have to be calculated first. Each nested behaviour has a given execution probability $P(N_i)$. For the branch failure probability, we have:

$$fp(A_{branch}) = 1 - \sum_{i=1}^{k}(P(N_i) \cdot (1 - fp(N_i)))$$

A fork action $A_{fork}$ has a finite set of nested forked behaviours $\{N_1, N_2, \ldots, N_k\}$ with failure probabilities $fp(N_i)$. Assuming the reliability independence of the individual forked behaviours (i.e. that the order and timing of thread execution does not affect the reliabilities of $N_i$)[1], the failure probabilities $fp(N_i)$ can be calculated independently, and $fp(A_{fork})$ expressed as:

$$fp(A_{fork}) = 1 - \prod_{i=1}^{k}(1 - fp(N_i))$$

A call action $A_{call}$ may occur within a usage scenario, as well as a SEFF. It always references another behaviour $N$, whose failure probability $fp(N)$ has to be calculated first. If the referenced behaviour is executed within the same resource container as the original one, the failure probability of the call is equal to $fp(N)$. Otherwise, the call procedure includes two message transports over a network link (request and return), and the failure probability of the link $fp(L)$ has to be considered, resulting in:

$$fp(A_{call}) = 1 - (1 - fp(N)) \cdot (1 - fp(L))^2$$

Finally, internal actions $A_{int}$ are the base case of the generation algorithm. They have a given set of required resources $R(A_{int})$ and a software failure probability $fp_s(A_{int})$. If the internal action requires a resource that is unavailable under the assumed physical system state $s_j$, the internal action fails with probability 1.0. Otherwise, the internal action may still fail with the given software failure probability. Thus, we have:

$$fp(A_{int}) = \begin{cases} fp_s(A_{int}) & if \ \ \forall r_i \in R(A_{int}) : \\ & s_j(r_i, t) = OK \\ \\ 1.0 & if \ \ \exists r_i \in R(A_{int}) : \\ & s_j(r_i, t) = NA \end{cases}$$

Concluding, the failure probabilities of all actions $fp(A_i)$ can be calculated to determine the failure probability of the behaviour $fp(B)$ as a whole. The Markov generation algorithm starts by constructing the DTMC for the given usage scenario, evaluates all nested and referenced behaviours recursively, and calculates the usage scenario failure probability $fp(B_{usage})$, which leads to the probability of successful execution under the assumption of the physical system state $s_j$:

$$PSE(s_j) = 1 - fp(B_{usage})$$

This calculation is done repeatedly for the whole set $S$ of physical system states (see Fig. 2), so that for all states $s_j$ we know their success probabilities $PSE(s_j)$.

1. Note that this method does not explicitly consider concurrency errors resulting from thread interaction or shared resource access, which can be removed with existing techniques before the analysis [40], or implicitly included into the individual failure probabilities $fp(N_i)$.

#### 4.3.2 Results Aggregation

The results of system reliability under each physical system state $s_j$ are aggregated as follows. Having the success probability of scenario execution under each $s_j$, i.e. $PSE(s_j)$, and for an arbitrary time $t$ the probability of system being in the state, i.e. $P(s_j, t)$, the overall probability of successful execution $PSE$ can be determined as a weighted sum over all physical system states (step 9 in Fig. 2):

$$PSE = \sum_{j=1}^{m} (PSE(s_j) \cdot P(s_j, t))$$

#### 4.3.3 Complexity

Thanks to the recursive nature of our algorithm, the reliability evaluation is space-effective. The number of Markov chains present at any moment is limited by the maximal depth of the stack of called and nested behaviours throughout the PCM instance. The size of each Markov chain is equal to the size of the corresponding behaviour plus 1 (for the additionally added FAILURE state, see Fig. 3). The aggregation of results over all physical system states can be calculated successively, without the need to store each result separately.

Regarding time-effectiveness, the calculations involved in determining the success probability of a single physical system state $PSE(s_j)$ are generally very fast, apart from the fact that loops involve calculating powers with exponents being equal to the loop iteration count (see above), which may be high depending on the given loop specification. The fact that each physical system state has to be evaluated to determine the overall result leads to a complexity of $O(2^n)$ of our algorithm relating to the number $n$ of resources modelled as a part of the resource environment. This exponential complexity presents an issue regarding the scalability of our approach. We therefore include scalability considerations into our case study (see Section 6.7).

The exponential complexity could be avoided if the approach refrained from the individual evaluation of all physical system states. Then, the back step in Fig. 2 (from activity 8 to 7) could be omitted. The calculation of internal action failure probabilities $fp(A_{int})$ (see Section 4.3.1) would take into account resource availabilities instead of fixed states for the required resources $r_i \in R(A_{int})$:

$$fp(A_{int}) = 1 - (\prod_i Av(r_i)) \cdot (1 - fp_s(A_{int}))$$

A single evaluation as described in Section 4.3.1 would already yield the result, no aggregation (Section 4.3.2) would be required.

We decided against this strategy because it introduces high inaccuracies if the same resource is accessed many times during a single service execution. The reason is that it allows each resource in the model to give a different answer about its actual state during each access. This is very unlikely to reflect the reality, since resource failure and repair times are very long, orders of magnitude longer than the duration of a single usage scenario execution [10]. Hence it is much more likely that the resource state does not change along the whole usage scenario execution, which is reflected by our approach. Section 6.6 validates this argument by comparing the results of both strategies when applied to our case study system.

### 4.4 Implementation

We have implemented the algorithms described above as Eclipse plug-ins integrated into the *PCM Bench* [45]. The PCM itself is implemented in Ecore using the Eclipse Modelling Framework (EMF). Users can edit PCM instances in a UML-like fashion with graphical editors generated by GMF (Fig. 4). The PCM Bench can check predefined semantic constraints on the models and generate different kinds of analysis models (e.g., Markov chains, queuing networks, and simulation models) via model transformations.

For the reliability analysis, a reliability solver takes a fully specified PCM instance as an input, and calculates system reliability as an output. Optionally, a more detailed output differentiates the determined failure potential according to user-defined failure types. Furthermore, stop conditions can be defined for fast evaluation of systems with many hardware resources, on the cost of prediction accuracy. Such stop conditions define a maximal number of evaluated physical system states, a maximal solving time, or a minimal reached numerical accuracy. Moreover, the analysis tools allow for sensitivity analyses over varying user-defined parameters of the models.

All tools are freely available and open source [45].

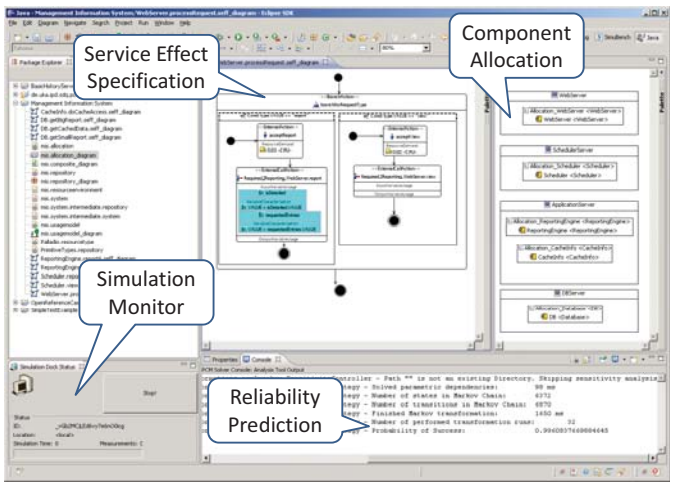



Fig. 4. Graphical Modelling and Prediction Tool

## 5 Improving Reliability with the PCM

In the preceding sections, we have discussed how to model a component-based software architecture with the

PCM (see Section 3), and how to trigger the transformation into the corresponding analysis model and its evaluation (see Section 4). This section gives an overview over possible architectural changes for improved reliability, and explains how these changes are reflected in the PCM model (see Fig. 2).

If the predicted reliability of a system is too low, it can be improved by applying certain architectural changes or *architecture tactics* [46]. Generally, these tactics improve reliability, but come with additional costs, and potentially degrade the architecture with respect to other quality attributes. It is the task of the software architect to evaluate various solutions and determine a good trade-off between all existing quality and cost goals.

The PCM supports two categories of architecture tactics for improved reliability: *scalar* and *topological improvements*. Scalar improvements change one or more values of reliability annotations, leaving the rest of the model unchanged. They can be further classified into improvements of software reliability, hardware availability, and network reliability. Topological improvements cover all changes to the structure of the architecture, namely component deployment and component assembly, that lead to improved reliability. Each change falls exactly into one category and may be applied independently from other changes. Table 3 lists examples of architecture tactics for each category, including their potential impacts on costs and other quality attributes.

Each tactic implies a certain type of change in the PCM model. However, the exact model change depends on the concrete architectural improvement. As an example, consider the replacement of a hardware resource $r_i$ by an array of $n$ redundant resources. Assuming that the resources in the array fail independently from each other, and that the array as a whole only fails when none of its resources is available, we can state for the availability $Av(r_i, n)$ of the array:

$$Av(r_i, n) = 1 - (1 - Av(r_i, 1))^n$$

where $Av(r_i, 1)$ is the given availability of the single resource $r_i$:

$$Av(r_i, 1) = MTTF_i/(MTTF_i + MTTR_i)$$

In the PCM model, we can reflect the change by adjusting the $MTTF_i$ and $MTTR_i$ values of the resource $r_i$ to $MTTF_n$ and $MTTR_n$ such that:

$$Av(r, n) = MTTF_n/(MTTF_n + MTTR_n)$$

A solution to this problem is given by:

$$MTTR_n = (MTTR_i)^n/(MTTF_i + MTTR_i)^{n-1}$$

$$MTTF_n = MTTF_i + MTTR_i - MTTR_n$$

As Table 3 shows, scalar improvements comprise decreases of software failure probabilities, increases of MTTF values, and decreases of MTTR values. Topological improvements change the assembly or allocation of software components. However, not each change of topology is an improvement. For example, a re-allocation of a software component to another resource container can have both a positive and negative influence on system reliability, depending on the architecture.

Our approach enables systematic evaluation of the influence of changes on system reliability without the need for re-configuration or re-implementation and execution of the real system. Starting from an initial architecture model $A_{init}$, the software architect evaluates several options for change and chooses the most beneficial one. This process may be repeated to stepwise improve the architecture, until a sequence of changes has converted $A_{init}$ into another architecture $A_{final}$ that satisfies existing quality and cost goals. Thereby, the order in which changes are applied may influence the reliability impact of a single change (e.g., it is more beneficial to reallocate component $C$ to server $S$ after increasing its $MTTF$ then before doing so). If the number of available options for change is too big to evaluate all of them, the architect may apply meta-heuristics to guide the selection of changes [50], [51].

## 6 Case Study Evaluation

The goal of the case study evaluation described in this section is to demonstrate the capabilities of our reliability prediction approach (Sections 6.3, 6.4), to provide evidence for the correctness of the involved calculations (Section 6.5), to validate specific assumptions of the Markov transformation (Sections 6.6), and to assess the scalability of our approach (Section 6.7). To this end, we introduce the PCM instances of two distributed, component-based systems (Sections 6.2, 6.8) and use these instances as our case studies. We also discuss general challenges of software reliability validation and provide a reasoning for our validation approach (Section 6.1).

### 6.1 Validation of Software Reliability Models

For a software reliability model, there are several aspects to validate. First of all, it should model plausible "behaviour". This means, a variation of the input parameters should result in plausible changes of the prediction result. Beyond this, the accuracy of the predictions should be validated, ideally against measured values. From a software engineering perspective, also the applicability should be validated: what is the training effort and modelling effort to yield reasonable predictions? How large is the influence of personal estimations and experience? And ultimately, one would like to validate the benefits: does the application of reliability predictions really result in better software designs?

However, the challenges in the empirical validation of software reliability models are so strong, that in practice validations are much weaker and mostly are only done on the plausibility level. The main reason lies in the difficulty of measuring high reliability values for software. Ideally, one would like to have a benchmark

| | | Tactics Name | Description | Non-Reliability Impacts | PCM Modelling |
|---|---|---|---|---|---|
| Scalar | Software Reliability | High-reliability software components | Apply a high-quality development process to software components for high reliability | Increased implementation and testing efforts | Decrease internal action failure probabilities |
| | | Design diversity ($n$-version programming) [47], [48] | Let each request be handled simultaneously by $n$ (different) versions of the same algorithm, with voting or other selection strategy at the end to choose the result | Additional costs for the design of $n$ algorithms, performance impact due to redundant computation | Decrease internal action failure probabilities |
| | | Rejuvenation techniques | Automatically restart components, application servers, or op. systems after failures or periodically | Performance impact due to frequent restart actions | Decrease internal action failure probabilities |
| | Hardware Availability | High-availability hardware | Operate the system on hardware with low failure rates and low service times in case of failure | Increased hardware and servicing costs | Increase MTTF/decrease MTTR of the resource |
| | | Redundant hardware (fault-tolerant HW, fail over) [49] | Use redundant hardware resources (e.g., RAID arrays, redundant CPUs, redundant servers) | Increased hardware costs, performance overhead due to fail-over | Increase MTTF/decrease MTTR of the resource |
| | | Heartbeat (ping/echo) [46], [47] | Use a monitoring system that periodically tests the availability of hardware resources, and react with a repair action | Additional monitoring costs, performance overhead | Decrease the MTTR of the monitored resource |
| | Netw. Rel. | High-reliability network | Use network links with high capacity and reliability | Increased network costs | Decrease communication link failure probabilities |
| Topological | | Change component deployment | Relocate "reliability-sensitive" components to servers with high-availability resources | Impacts on other quality attributes, e.g., performance | Change the deployment model |
| | | Change component assembly | Change assembly of components such that services are provided by the least "reliability-sensitive" components | Impacts on other quality attributes, e.g., performance | Change the architectural model |

TABLE 3
Architecture Tactics for Reliability Improvement

for reliability, like we have benchmarks for performance. However, the only approach to measure reliability values before execution is the approach of statistical testing. In statistical testing one tries to simulate the later usage of the system through test cases. It is well known that the number of test cases to be executed to measure high reliability values (e.g., $1 - 10^{-6}$ and higher) is prohibitively large. Research in increasing the expressiveness of statistical testing would be highly relevant but is beyond the scope of this paper.

Hence, we are left with the only possibility to use the data measured during the execution of the software system. While some failure databases exist that record the reliability of specific software systems, these databases have one common problem. They do not record the input values of the measured system execution. The input values are however a crucial reliability-influencing factor, significantly influencing system execution and its consequent reliability. One can hardly assume that the in-out data used implicitly in the often very old databases correspond to the data used in new reliability models.

As a result, if one surveys existing reliability models, validation is done by plausibility (i.e., parameters are varied and its sensitivity on the result is investigated) and predictions are, at the best, compared to simulations, i.e., to other prediction models (see Table 4 for details of the validations of related work). Goseva-Popstojanova et al. state: "Although numerous papers were devoted to architecture-based software reliability modelling, most of them either do not include numerical illustrations, or illustrate the models on simple made-up examples. A few papers that so far applied the theoretical results on real case studies did not include building the software architecture or identification of faults" (taken from [52], p1).

In this paper we validate our new model with plausibility (i.e., sensitivity analysis) and by comparing predictions against simulated data.

| Paper | Year | Illustrating Example | Sensitivity Analysis | Simulation/Measurements | Real-World Application |
|---|---|---|---|---|---|
| Cheung et al. [6] | 1980 | ✓ | ✓ | ✗ | ✗ |
| Cortellessa et al. [23] | 2002 | ✓ | ✓ | ✗ | ✗ |
| Gokhale et al. [12] | 2002 | ✓ | ✓ | ✗ | ✗ |
| Reussner et al. [16] | 2003 | ✓ | ✓ | ✓ | ✗ |
| Goseva et al. [33] | 2003 | ✓ | ✓ | ✗ | ✗ |
| Grassi [35] | 2004 | ✓ | ✓ | ✗ | ✗ |
| Yacoub et al. [18] | 2004 | ✓ | ✓ | ✗ | ✗ |
| Rodrigues et al. [17] | 2005 | ✓ | ✗ | ✗ | ✗ |
| Popic et al. [27] | 2005 | ✓ | ✓ | (✓) | ✗ |
| Wang et al. [9] | 2006 | ✓ | ✗ | ✗ | (✓) |
| Sharma et al. [11] | 2006 | ✓ | ✓ | ✗ | ✗ |
| Sato et al. [10] | 2007 | ✓ | ✗ | ✗ | ✗ |
| Sharma et al. [8] | 2007 | ✓ | ✓ | ✓ | (✓) |
| Cheung et al. [19] | 2008 | ✓ | ✓ | ✗ | (✓) |
| Lipton et al. [28] | 2008 | ✓ | ✓ | ✗ | ✗ |
| Hamlet [24] | 2009 | ✓ | ✓ | ✓ | ✗ |

TABLE 4
Validation of Software Reliability Models

### 6.2 Business Reporting System

Fig. 5 illustrates a high-level view on the Business Reporting System (BRS), which generates management reports from business data collected in a database. The model is based on an industrial system [53]. Users can

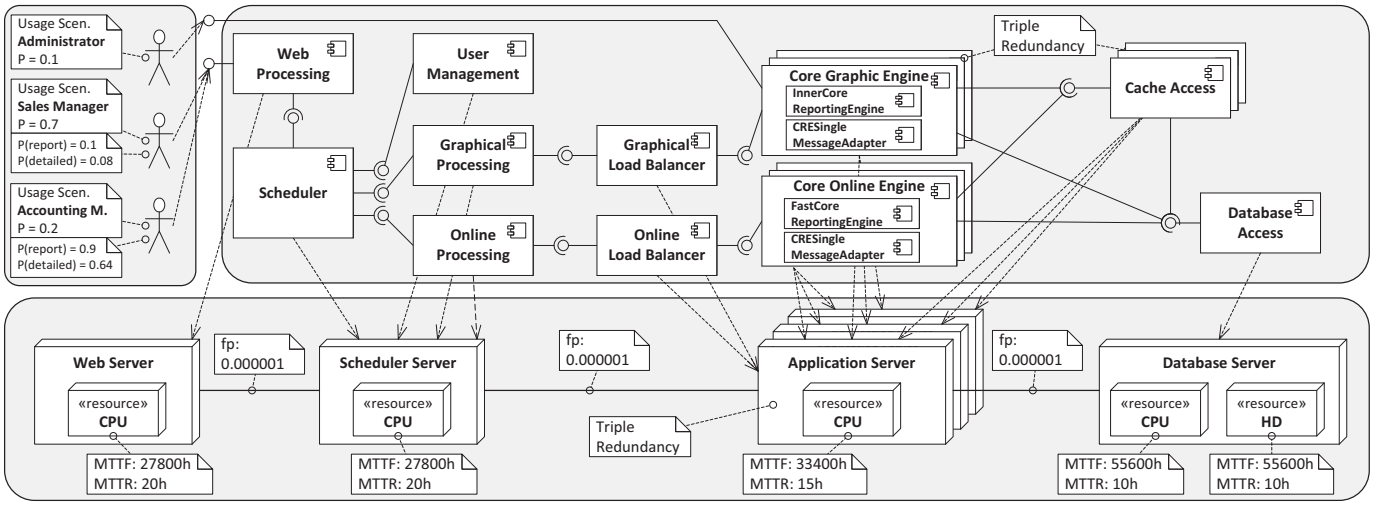



Fig. 5. PCM Instance of the Business Reporting System (Overview)

query the system via web browsers. They can view the currently collected data or generate different kinds of reports (coarse or detailed) for a configurable number of database entries. Furthermore, users can either request to see the current live data (online mode) or aggregated statistics of historical data (graphical mode). The details of the model are described at our website [54]. The model can also be downloaded there.

The BRS system consists of 23 software components deployed on 6 servers. The web server propagates user requests to a scheduler server, which hosts, amongst others, a scheduler and a user management component. From there, requests reach the main application server and are possibly dispatched to 2 further application servers by 2 load balancer components. Request processing and logging of user actions takes place concurrently (modelled through fork actions). A database server hosts the database and a corresponding data access component. The system includes caches to reduce the need of database accesses. The current cache hit rate at the time of a request is a function of the cache size and fill degree. Thus, it depends on the component state and is modelled as a probabilistic component parameter of the cache access components.

The usage model provided by the domain expert (as sketched in Fig. 5) contains 3 usage scenarios that present different use cases of the system: sales managers use the system mainly for online viewing of live data, accounting managers request the production of graphical reports, and administrators perform system maintenance activities.

As concrete failure data was not available for the system, we used generic failure probability and hardware availability estimations as an input to the model. In a real setting, such estimations could be based on a variety of factors, including historical data, statistical testing, expert knowledge, and (in particular for hardware MTTF values) vendor specifications. For internal actions, we set software failure probabilities to $10^{-5}$. This is roughly in line with existing empirical case studies (consider, as an example, the point estimates of component reliabilities in [55]). For hardware MTTF values, we took the work of Schroeder et al. [44] as a basis, who analysed the actual failure rates of hardware components of several large systems over the course of several years. For MTTR values, we assumed that a repair takes place within 10 to 20 hours.

### 6.3 BRS Prediction Results

To illustrate the capabilities of our prediction approach, we executed our analytical Markov chain solver described in Section 4 for different BRS usage scenarios and design alternatives. As the model involves 7 resources, Markov chains for 128 ($= 2^7$) different physical system states were generated and evaluated in each prediction run. Solving the parameter dependencies, generating the different chains, and computing their absorption probabilities took about 20 seconds on an Intel Core 2 Duo with 2.6 GHz and 2 GB of RAM.

In addition to the base version of the BRS, we considered 3 design alternatives that result from the application of architecture tactics for reliability improvement as described in Section 5. Alternative 1 employs the "high-reliability software components" tactic by ensuring rigorous quality-assuring measures during the development process, setting software failure probabilities in the BRS model to 0. Alternative 2 employs the "high-availability hardware" tactic using additional backup servers, implicitly captured through decreased MTTR values in the BRS model. Finally, alternative 3 decreases the network failure probability to 0 through application of the "high-reliability network" tactic. As each of the alternatives improves reliability with respect to either the software, hardware or network dimension, the corresponding prediction results also indicate the influence

of each dimension on the overall failure potential (see Section 3.3.5).

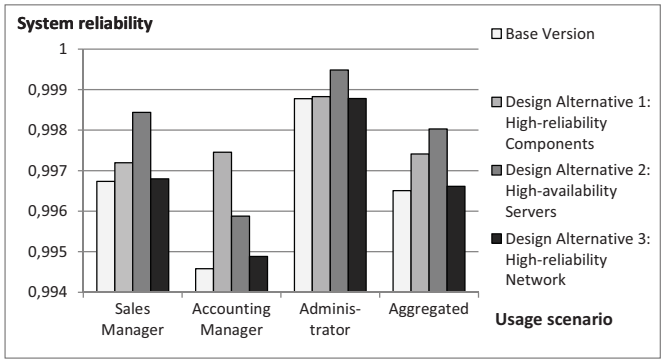



Fig. 6. BRS Reliability Prediction Results

Fig. 6 shows the BRS prediction results. In general, accounting managers experience more failures than sales managers, because the detailed report generation is more computing intensive than the online data view. However, eliminating the software failure potential (alternative 1) leads to very similar success probabilities for both user types, with a particular high improvement for accounting managers (namely, 53% less failure occurrences). The administrator profile is generally very reliable, but cannot be significantly improved by the alternatives, except for improving hardware availability (alternative 2). Improved network reliability (alternative 3) does not create a significant benefit regarding BRS system reliability. Fig. 6 also presents an aggregated prediction result of the overall system usage profile, under the assumption that 70% of all system calls come from sales managers, 20% from accounting managers, and 10% from administrators. Overall, design alternative 2 has the highest positive impact on system reliability. However, the results in Fig. 6 are subject to uncertainty. The following section discusses this issue in detail.

### 6.4 Sensitivity Analyses

Sensitivity analyses, which vary certain parameters of the architectural model and observe the effects on the prediction results, are an important means to gain more insights about the reliability characteristics of the system under study [12], [20]. This section investigates the uncertainty of the decision between the BRS design alternatives, as well as the criticality of individual software components and hardware servers with respect to the system reliability (details in [54]).

Fig. 7 illustrates the uncertainty regarding the ranking of the BRS design alternatives. Assuming an aggregated usage profile as introduced in Section 6.3, the figure varies individual aspects of the architectural model such as the values of all software failure probabilities (Fig. 7(a)), hardware MTTF values (Fig. 7(b)) and network failure probabilities (Fig. 7(c)). Crossing lines indicate changes in the ranking of the alternatives. For example, Fig. 7(a) shows that alternative 2 is the preferred choice for software failure probabilities between $10^{-7}$ and $10^{-5}$, while alternative 1 is preferable above $10^{-5}$. Software architects can take into account the existing level of confidence of the estimated failure probabilities and MTTF values, and use the results of Fig. 7 to make a more well-informed decision between the design alternatives. For example, assuming that software and network failure probabilities are generally not above $10^{-5}$, and hardware MTTF values are not above 8 years, a decision for alternative 2 (high-availability servers) can be made with a high confidence that this is actually the most beneficial choice with respect to the system's reliability.

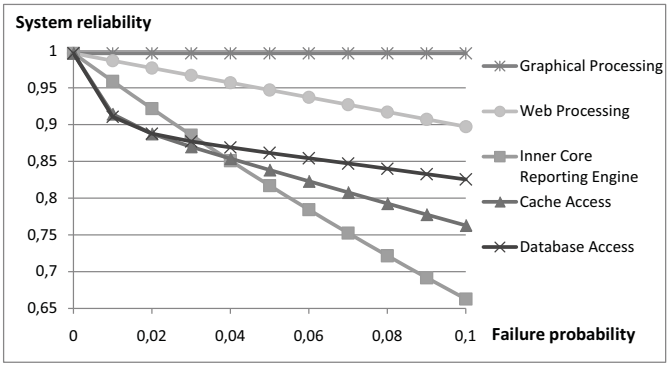
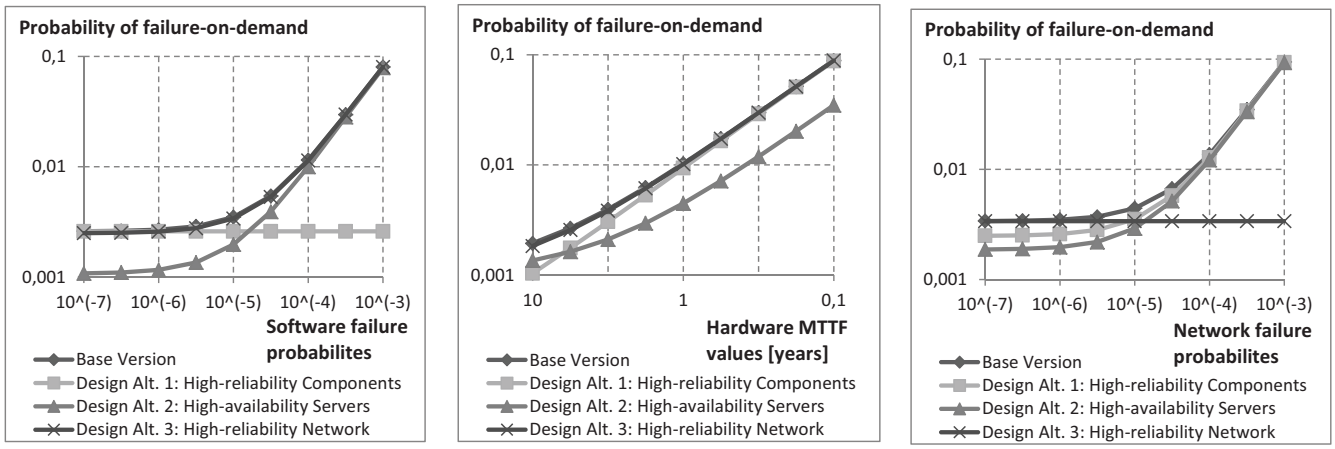



Fig. 8. Sensitivity to Software Failure Probabilities

Having decided for a certain alternative, sensitivity analyses can be used to identify the most critical parts of the system, which should receive special attention during the development. To this end, Fig. 8 illustrates the impact of varying individual software failure probabilities to system reliability (assuming the base version). In each sensitivity run, the failure probabilities of a certain software component were varied between 0 and 0.1, while all other components remained unchanged. We executed all runs under the sales manager usage scenario. As the figure shows, increasing failure probabilities generally lowers system reliability, except for the graphical processing component, which is not required by sales managers (as they request online views rather than graphical reports). System reliability is particularly sensitive to the inner core reporting engine, which provides the central computations for user requests. The cache access and database access components have a non-linear impact on system reliability because of the multiple accesses of those components within a single system service execution (one access per requested data item). Hence, it is most beneficial to focus on the improvement of the cache and database access components, as well as the inner core reporting engine.

Another analysis deals with varying MTTF values of hardware resources under the sales manager usage scenario. Fig. 9 shows the results. Each sensitivity run varies the MTTF values of the hardware resources of one server in the BRS model, while leaving the other servers unchanged. The differences between the individual servers are generally very subtle and only be-




(a) Uncertain Software Failure Probabilities (b) Uncertain Hardware MTTF Values (c) Uncertain Network Failure Probabilities

Fig. 7. BRS Design Alternatives under Uncertainty

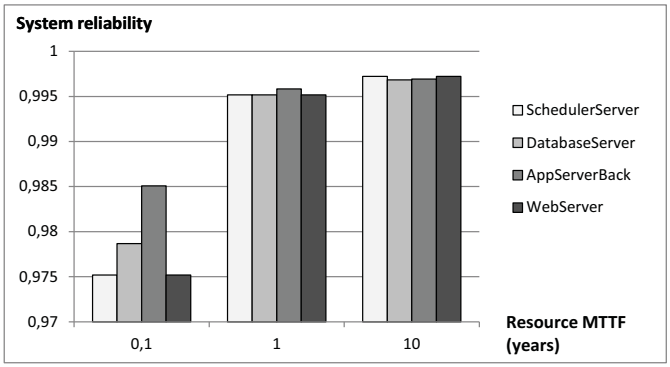



Fig. 9. Sensitivity to Hardware MTTF Values

come more apparent when MTTF values are strongly decreased to 0.1 years. The results show that system reliability is most sensitive to the scheduler server and web server, which are accessed by each user request. The database server is only accessed if the requested data does not already reside in the system caches, and thus has less impact on the success probability of service execution. Furthermore, we included one of the replicated application servers into the analysis. Because of the replication, the system is particularly robust against the unavailability of this server. Thus, the highest priority for further replication should be on the web server and the scheduler server.

### 6.5 Reliability Simulation

As a further indication for the validity of our prediction results, we compared the analytically predicted system reliability to the results of a reliability simulation performed over the PCM instance of the BRS. The goal of this validation was to provide evidence for the correctness of our analysis, i.e. if all inputs (PCM models including reliability annotations) are accurately provided, our method produces an accurate result (system reliability prediction).

For simulation purposes, we extended the existing PCM performance simulation, which is based on the SSJ framework [56]. The tool uses model transformations implemented with Xpand from the Eclipse Modeling Project [57] to generate Java code from the PCM instance under study. During a simulation run, a generated SSJ load driver creates requests to the code according to the usage model specified as a part of the PCM model. For a detailed description of the performance simulation, see [58]. To evaluate system reliability, we included software failures, communication link failures, and the effects of unavailable hardware into the simulation.

To simulate a software failure, an exception may be raised during execution of an internal action. A random number is generated according to the given failure probability, which decides about success or failure of the internal action. Communication link failures are handled in the same way. Furthermore, the simulation includes the notion of hardware resources and their failure behaviour. It uses the given MTTF/MTTR values as mean values of an exponential distribution and draws samples from the distribution to determine actual resource failure and repair times. Whenever an internal action requires a currently unavailable hardware resource, it fails with an exception. Taking all possible sources of failure into account, the simulation determines system reliability as the ratio of successful service executions to the overall execution count.

In contrast to our analytical reliability evaluation, the simulation takes system execution times into account, making it more realistic in this respect. Values of variables in the control flow are preserved within their scope, as opposed to analysis, where each access to a variable requires drawing a sample from its probability distribution (see [41]). Resources may fail and be repaired at any time, not only between usage scenario executions. Resource states are observed over (simulated) time, leading to more realistic failure behaviour of subsequent service executions. The simulation is substantially slower than

the analysis and cannot serve as our main prediction method, but we can apply it for validation purposes. Notice that although the simulated time span may cover several years of system operation, the simulation treats the system as being in a steady state (i.e. the system architecture including reliability annotations, as well as the system usage profile, are constant throughout the simulation). Therefore, the simulation results are comparable to the results obtained by our Markov analysis.

| Usage Scenario | Reliability Prediction | |
|---|---|---|
| | Analysis | Simulation |
| Sales Manager | 0.99673711 | 0.99691003 |
| Accounting Manager | 0.99457976 | 0.99443006 |
| Administrator | 0.99877648 | 0.99878001 |

TABLE 5
Analytical vs. Simulated Reliability Prediction

To validate prediction results, we applied the simulation tool to each usage scenario of the BRS PCM instance and simulated its execution for 10 years (i.e. 315 360 000 seconds of simulation time). Each simulation run took roughly 30 minutes of real time. Depending on the usage scenario, we recorded between 5 and 298 software failures, between 117 and 254 failures due to hardware unavailability, and between 0 and 43 communication link failures during the simulation run. Table 5 compares the predicted and simulated system reliability values. All performed simulation runs deviate from the predicted values by less than 0.0002. Furthermore, if the design alternatives are to be sorted according to their reliability, simulation leads to the same order as prediction. These results give evidence that our approach accurately predicts system reliability and helps software architects to select the best out of multiple design alternatives.

### 6.6 Markov Transformation Assumptions

This section evaluates two underlying assumptions of the Markov transformation algorithm as introduced in Section 4, namely (i) the benefit of the explicit consideration of all physical system states, and (ii) the benefit of separate usage modelling in terms of modelling effort.

In Section 4.3.3, we discuss that the explicit consideration of all physical system states poses an issue with respect to the scalability of the approach. This issue could be avoided if the resource availability was evaluated independently for each access of a resource during service execution, according to the steady-state availability of the resource. We implemented this alternative strategy and evaluated the system reliability for the BRS usage scenarios to 0.979 (sales manager), 0.886 (accounting manager), and 0.997 (administrator). Compared to the original strategy, which was confirmed by simulation results (see Table 5), the alternative strategy potentially leads to high prediction inaccuracy. This is particularly true in the computing-intensive case of the accounting manager, because physical resources are accessed many times during service execution. The higher accuracy of our hardware unavailability consideration justifies the better scalability of the alternative strategy.

The second assumption refers to the separate usage modelling, which is a necessary pre-requisite to the separation of modelling concerns between component developers and domain experts, but also reduces the modelling effort by automatic adaptation of the underlying Markov model to usage profile changes. To measure this effect, we implemented a comparison algorithm that shows the changes between multiple Markov models resulting from the BRS with different usage model variants. We focused on a single system call for the creation of an online report, with different input parameter values, and evaluated the resulting Markov models that represent the corresponding system behaviour. Table 6 shows the results. In the first case, a coarse report for 1 database entry is requested. The resulting Markov model has 367 states and 408 transitions. The size of the chain is due to the recursive nature of the transformation algorithm, which captures all possible execution paths throughout the architecture. In the second case, the number of entries is increased from 1 to 2, leading to 111 new states and 132 new transitions in the resulting Markov model. The model is extended because the BRS repeats a part of its computation for each requested entry. In the third case, the requested report type changes from coarse to detailed, leading to substantially different computations compared to the first case. Existing states and transitions are deleted, and new ones are added. Hence, already a single usage parameter change can have significant impact on the resulting Markov model.

To take the discussion one step further, we compared usage modelling with and without explicit consideration of input parameters. While the former is provided by our approach (as one of its distinguishing characteristics, see Section 2.1), the latter is realized by a number of related works, such as the scenario-based approaches (e.g., [17], [18], [23]). As these approaches do not offer explicit constructs for input value propagation from the usage model to the rest of the system, this information either needs to be abstracted away or encoded into the model. The first alternative is to use probabilistic abstractions replacing each input-dependent control flow decision (e.g. branch or loop) with fixed transition probabilities. However, such an abstraction requires changes of all initially input-dependent transition probabilities whenever the usage profile changes. The second alternative is to specify each system operation or message multiple times, with the parameter values encoded into the name of the message. While investigations on process algebra have shown that such an encoding does not increase the expressive power of the modelling language, it may increase the model size significantly, even exponentially [59]. Hence for each parameter change in the usage model, an exponential number of modifications may be needed in the rest of the model. For a more detailed discussion and illustration of this issue on a

| Case | System Call | Markov States | | Markov Transitions | |
|---|---|---|---|---|---|
| | | Total | Changes wrt. Case #1 | Total | Changes wrt. Case #1 |
| #1 | getOnlineReport(entries = 1, type = coarse) | 367 | – | 408 | – |
| #2 | getOnlineReport(entries = 2, type = coarse) | 478 | added: 111 | 540 | added: 132 |
| #3 | getOnlineReport(entries = 1, type = detailed) | 616 | deleted: 177, added: 426 | 698 | deleted: 207, added: 497 |

TABLE 6
Markov Model Changes through Changing Usage Parameters

simple model, see [54].

In conclusion, the separation of usage modelling with explicit consideration of input parameter values is indeed beneficial compared to other modelling methods.

### 6.7 Scalability

The scalability of our approach requires special attention. The method for reliability evaluation described in Section 4.3 increases the number of physical system states (and thus Markov chains) to be evaluated exponentially in relation to the number of hardware resources in the model. To examine the impact of this relation to the practicability of our approach, we analysed a number of simple PCM instances with a growing number of resources and recorded the execution time for our prediction tool. The results in Fig. 10 indicate that we can analyse models with up to approximately 20 resources within one hour. This involves generating and solving Markov chains for more than 1 000 000 physical system states.

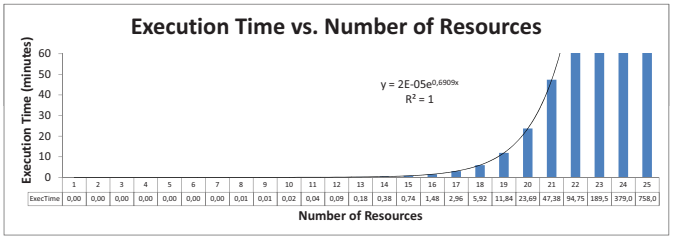



Fig. 10. Execution Time vs. Number of Resources

We deem these capabilities sufficient for typical business information systems (e.g., enterprise resource management or customer relationship management), e-business applications (e.g., online shops), industrial control systems (e.g., the ABB system introduced in Section 6.8), and other types of software-intensive systems. Even for large-scale installations, multiple resources and servers can often be grouped together and aggregated into a single modelled server (also see the example of redundant resources discussed in Section 5). More effective strategies for very large systems with more than 20 non-summable resources remain as a goal for future research.

Apart from the number of hardware resources, other dimensions of complexity (like the number of components or the number of behavioural specifications) are not a limiting factor. Due to the space- and time-effectiveness of the approach (see Section 4.3.3), even very complex models can be handled efficiently.

### 6.8 Industrial Control System

To increase the external validity of our approach, we analysed the reliability of a large-scale industrial control system from ABB. It is used in many different domains, such as power generation, pulp and paper handling, or oil and gas processing. The system implementation consists of several millions lines of C++ code. On a high abstraction level, the core of the system consists of eight subsystems, which we treat as software components in the following. These components can be flexibly deployed on multiple servers depending on the system capacity required by customers.

Fig. 11 depicts a possible configuration of the system with three servers. The names of the components and their failure probabilities have been obfuscated for confidentiality reasons. We modelled five of the most important usage scenarios, which are executed in parallel during system execution. Each usage scenario triggers a different control and data flow through the system. We built the PCM instance for the system based on architectural documentation. The model can be downloaded from our website [54].

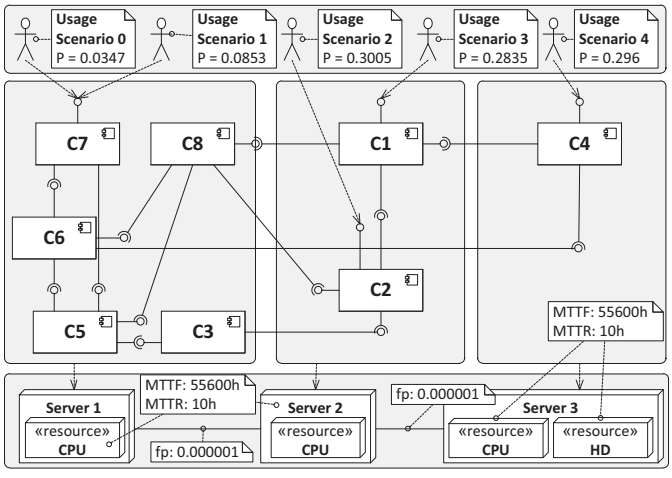



Fig. 11. PCM Instance of the Industrial Control System (Overview)

We collected reliability-related input parameters for the system (e.g., internal action failure probabilities) in a former study [43]. The system had been in customer use for several years, and a bug tracking system with failure reports collected during live operation was available. To determine internal action failure probabilities, we decided to construct a software reliability growth model (SRGM) for each component, as the bug tracking system related each failure report to a specific component.

We selected the Littlewood/Verrall SRGM from IEEE1633-2008 based on initial significance tests and industry affinity. Using the tool CASRE, we performed curve fitting with the failure report dates and the Littlewood/Verrall model to determine a failure probability for each component. We used this data to annotate the internal actions of the PCM instance of the industrial control system.

To determine the branch transition probabilities of PCM SEFFs, we used internal logging facilities of the system, which can be configured to record traces of component transitions. We executed the system in a testbed for two days based on the modelled usage scenarios and recorded trace logs. Using a script, we calculated branch transition probabilities from the number of component transitions in the trace logs. For the hardware reliability parameters, we reused the values from the BRS case study. Details of the data collection for the system are provided in [43].

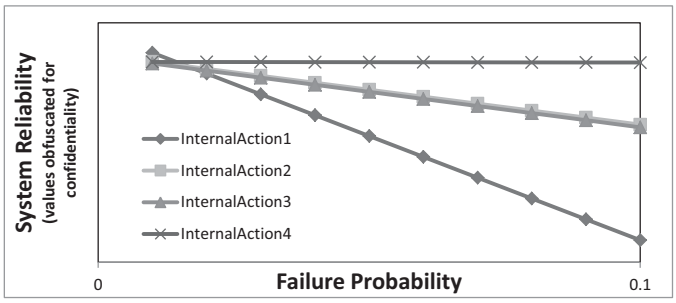



Fig. 12. Sensitivity to Software Failures

With the fully parameterized PCM instance of the industrial control system we performed a sensitivity analysis, investigated the influence of different usage profiles, and also checked the analytical output against results of our reliability simulation.

Fig. 12 shows the sensitivity of the system reliability to varying internal action failure probabilities. We omitted the concrete specification of the system reliability (y axis) for confidentiality reasons. The system reliability is most sensitive to InternalAction1 (from component C1), because the respective component is used with a high probability in usage scenarios 3 and 4. The system reliability is least sensitive to InternalAction4 (from component C3), because this component is used only in rare cases.

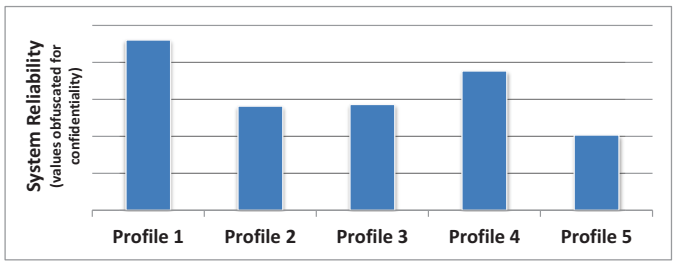



Fig. 13. System Reliabilities for different Usage Profiles

Fig. 13 shows the system reliability for five different variations of the system-level usage profile (i.e., probability distributions for the system-level usage scenarios). These profiles result from adjusting the usage scenario probabilities in the range of typical customer behaviours for applications of the system in different industry branches. The system reliability is again obfuscated. Notice that the origin of the y-axis is not 0.0, as the graph is zoomed in to highlight the differences. The maximum difference in the system reliability is 0.2 percent, which is significant because it results in a number of customer perceived errors.

Due to the inherent difficulties of software reliability validation as discussed in Section 6.1, we could not directly compare our prediction results with system reliability measurements during live operations. However, we executed the simulation tool described in Section 6.5 and emulated system operation for approx. 1.5 years (i.e., 50 000 000 seconds of simulation time), which took approx. 11 hours real time. The error between simulated and analytical result was approx. 0.015 percent. We conclude that the analytical solver produces plausible and sufficiently accurate results. We also discussed the model with developers of the system, who confirmed that the critically ranking gained by the model matched their experiences with the system.

The current model of the system resides on a high abstraction level (8 subsystems for several million lines of code). While this enables determining the most critical subsystem for the system reliability, we would need a lower abstraction level to make detailed recommendations on how to improve the system. A lower abstraction level however requires other data collection methods for internal action failure probabilities. We will investigate this issue in future work.

Nevertheless there are several benefits of the model for the current system. The system is used in different industrial application domains (e.g., power generation, metal production, oil and gas processing etc.) which can be reflected to a certain extent by adjusting the usage scenario probabilities as in Fig. 13. Thus refined reliability predictions can be made for different customers. Furthermore, the model allows for an efficient test effort allocation to the most critical subsystems, which were formerly unknown. This can save significant future testing costs. Additionally, the model allows for discussing the system reliability on the architectural level, involving tradeoffs with other quality attributes, and assessing the impact of different fault tolerance mechanisms. These discussions however have to remain confidential.

In future work, we plan to develop new data collection methods, which will enable more fine-granular system modelling and thus more detailed recommendations for system design.

## 7 Assumptions and Limitations

In this section, we discuss assumptions and limitations of our approach. We focus on three central issues: (i)

the knowledge required to create a PCM model with reliability annotations, (ii) the estimation of proper usage profiles , and (iii) limitations regarding the expressiveness of the model.

### 7.1 Reliability Annotations

In order to predict the reliability of a component-based software architecture with our method, a complete PCM model has to be provided. While it is reasonable to assume that the individual developer roles can contribute their model parts as discussed in Section 3.2, the question arises how to estimate the required reliability annotations, namely

- internal action failure probabilities,
- MTTF/MTTR values of hardware resources, and
- communication link failure probabilities.

Failure frequencies are hard to measure, because failures are rare events. Failure probabilities are hard to estimate, because the exact circumstances of failure occurrences are unknown. Nevertheless, there are approaches that specifically target the problem of determining failure probabilities and related information. Methods using statistical testing [2], software reliability growth models [1], [15] and code coverage metrics [60] have been originally proposed for system-wide analysis, but can in principle also be applied to individual software components. Historical data from functionally similar systems and expert knowledge can serve as further sources of information. For hardware MTTF/MTTR values and network failure probabilities, in many cases vendor specifications are available, which can be refined by own testing and experience.

In a former work [43], we demonstrate how to derive reliability-relevant input information for an industrial control system, which also serves as a case study in the current paper (see Section 6.8). The same techniques could also be applied to other kinds of systems targeted by our approach.

### 7.2 Usage Profile Estimation

Our approach assumes that domain experts are able to create a usage model reflecting typical user behaviour and potentially including several user classes and probability distributions (Section 3.2.4). Such information may not always be available, which potentially limits the benefit of our fine-grained usage modelling capabilities. However, several proposals for fine-grained usage profile estimation exist. John Musa [61] suggests to iteratively refine the assumed operational profile from a customer, user, system mode, and functional profile. Multiple PCM usage model instances can reflect these refinement stages. Whittaker and Poore [62] propose three strategies to determine models of user behaviour and suggest to use prototypes, prior versions, or similar systems to estimate call probabilities. Goseva and Kamavaram [63] discuss further how to deal with uncertainty when defining these values. Remaining uncertainty about usage profile parameters can be tackled using sensitivity analysis.

### 7.3 Model Expressiveness

Regarding the expressiveness of our reliability modelling approach, we face a general trade-off between the model appropriateness for real-world software systems on the one hand, and its complexity on the other hand. A more complex model requires a higher modelling effort, needs more fine-grained reliability annotations as an input, and increases the danger of state-space explosion of the underlying analytical model. Therefore, in analogy to related approaches (see [2], [3], [4]), we have restricted the approach to the most important concepts from our point of view (see Section 3.3). More concretely, we assume that

- software and communication link failures are always transient and have no side-effects on subsequent service calls,
- each failure propagates to the system border and is perceived as a system failure by the user,
- hardware resources have two states $OK$ and $NA$ (no intermediate states are allowed), and
- failure and repair times of hardware resources are exponentially distributed.

Furthermore, we refrain from explicit modelling of

- stochastic dependencies between individual points of failure,
- stochastic dependencies between break-downs of hardware resources and their usage time within a single service call, as well as
- constraints on valid call sequences for component provided interfaces (i.e., interface protocols). These can be expressed and corresponding interoperability checks can be performed with our former work on component protocol checking [40].

The approach generally reflects concurrent system design. However, concurrent threads of control and data flow are assumed to be independent (i.e. not influencing each other's failure probabilities). Regarding system state, the approach allows for probabilistic modelling of component-internal state variables, but abstracts from the concrete modification of state within the control and data flow. We realized a more explicit state consideration for performance prediction based on the PCM (see [37]), and we plan to also enhance reliability prediction accordingly in the future.

## 8 Conclusions

In this paper, we present an approach for reliability modelling and prediction of component-based software architectures. The approach integrates even the rarely included architectural aspects (the usage profile and execution environment) to increase the accuracy of its

predictions, and it describes various reliability-relevant aspects of the architecture through well-known concepts from the software engineering domain. The approach has been realized as an extension of the Palladio Component Model (PCM), including tool support for model transformation into Markov chains, and implementation of a novel space-effective method of reliability evaluation.

By using the PCM as our basis, we take advantage of its features and apply its methodology to the field of software reliability. This includes the support of a distributed development process with multiple developer roles who can independently specify their respective parts of the architecture. Our approach enables software engineers to systematically consider reliability throughout the development process, to assess different architectural alternatives of a system to build, and to identify critical points of failure that require special attention. This kind of analysis helps to meet given reliability goals, and to build software systems that satisfy customer expectations regarding failure-free execution.

We plan to further extend our approach in multiple directions, including modelling, analysis, and validation. Regarding modelling, we aim to integrate more explicit consideration of state and concurrency, as well as new constructs for fault-tolerance mechanisms, to calculate the influence of failure recovery on system reliability. Regarding analysis, we aim to further improve the scalability of the approach for large-scale system installations with a complex execution environment. To this end, we will investigate possible optimization strategies for the involved calculations. Regarding validation, we aim at improved methods to derive reliability-relevant input information for real-world systems. These extensions shall further increase the benefits of our approach and its applicability to industrial software development projects.

## ACKNOWLEDGMENTS

This work was supported by the European Commission as part of the EU-project Q-ImPrESS (grant No. FP7-215013), by the German Federal Ministry of Education and Research (grant No. 01BS0822) and by the Czech Science Foundation.

## REFERENCES

[1] J. D. Musa, A. Iannino, and K. Okumoto, *Software reliability: measurement, prediction, application*. New York, NY, USA: McGraw-Hill, Inc., 1987.

[2] K. Goseva-Popstojanova and K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems," *Performance Evaluation*, vol. 45, no. 2-3, pp. 179–204, May 2001.

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

[4] A. Immonen and E. Niemel, "Survey of reliability and availability prediction methods from the viewpoint of software architecture," *Journal on Softw. Syst. Model.*, vol. 7, no. 1, pp. 49–65, February 2008.

[5] R. Roshandel, N. Medvidovic, and L. Golubchik, "A bayesian model for predicting reliability of software systems at the architectural level," in *QoSA*, ser. LNCS, vol. 4880. Springer, 2007, pp. 108–126.

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

[7] K. Goseva-Popstojanova and K. S. Trivedi, "Architecture-based approaches to software reliability prediction," *Computers & Mathematics with Applications*, vol. 46, no. 7, pp. 1023–1036, October 2003.

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

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

[10] N. Sato and K. S. Trivedi, "Accurate and efficient stochastic reliability analysis of composite services using their compact markov reward model representations," in *Proc. IEEE Int. Conf. on Services Computing (SCC'07)*. IEEE Computer Society, 2007, pp. 114–121.

[11] V. S. Sharma and K. S. Trivedi, "Reliability and performance of component based software systems with restarts, retries, reboots and repairs," in *Proc. 17th Int. Symp. on Software Reliability Engineering (ISSRE'06)*. IEEE Computer Society, 2006, pp. 299–310.

[12] S. S. Gokhale and K. S. Trivedi, "Reliability prediction and sensitivity analysis based on software architecture," in *Proc. 13th Int. Symp. on Software Reliability Engineering (ISSRE'02)*. IEEE Computer Society, 2002, pp. 64–78.

[13] H. Koziolek and F. Brosch, "Parameter dependencies for component reliability specifications," in *Proc. 6th Int. Workshop on Formal Engineering Approaches to Software Components and Architecture (FESCA'09)*, ser. ENTCS, vol. 253. Elsevier, 2009, pp. 23–38.

[14] S. Becker, H. Koziolek, and R. Reussner, "The Palladio Component Model for Model-Driven Performance Prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, January 2009.

[15] M. Lyu, Ed., *Handbook of Software Reliability Engineering*. McGraw-Hill, Inc., 1996.

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

[17] G. N. Rodrigues, D. S. Rosenblum, and S. Uchitel, "Using scenarios to predict the reliability of concurrent component-based software systems," in *Proc. Fundamental Approaches to Software Engineering (FASE'05)*, ser. LNCS, vol. 3442. Springer, 2005, pp. 111–126.

[18] S. M. Yacoub, B. Cukic, and H. H. Ammar, "A scenario-based reliability analysis approach for component-based software," *IEEE Transactions on Reliability*, vol. 53, no. 4, pp. 465–480, 2004.

[19] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik, "Early prediction of software component reliability," in *Proc. 30th Int. Conf. on Software Engineering (ICSE'08)*. ACM, 2008, pp. 111–120.

[20] L. Fiondella and S. Gokhale, "Importance measures for modular software with uncertain parameters," *Software Testing, Verification and Reliability*, vol. 20, no. 1, pp. 63–85, March 2010.

[21] S. S. Gokhale, "Quantifying the variance in application reliability," in *Proc. 10th IEEE Int. Symp. on Dependable Computing (PRDC'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 113–121.

[22] I. Meedeniya, I. Moser, A. Aleti, and L. Grunske, "Architecture-based reliability evaluation under uncertainty," in *Proc. 7th Int. Conf. on the Quality of Software Architectures (QoSA'11)*. New York, NY, USA: ACM, 2011, pp. 85–94.

[23] V. Cortellessa, H. Singh, and B. Cukic, "Early reliability assessment of uml based software models," in *Proc. 3rd Int. Workshop on Software and Performance (WOSP '02)*. ACM, 2002, pp. 302–309.

[24] D. Hamlet, "Tools and experiments supporting a testing-based theory of component composition," *ACM Transaction on Software Engineering Methodology*, vol. 18, no. 3, pp. 1–41, May 2009.

[25] K. Trivedi, D. Wang, D. J. Hunt, A. Rindos, W. E. Smith, and B. Vashaw, "Availability modeling of SIP protocol on IBM WebSphere," in *Proc. 14th IEEE Int. Symp. on Dependable Computing (PRDC'08)*. IEEE Computer Society, 2008, pp. 323–330.

[26] S. A. Vilkomir, D. L. Parnas, V. B. Mendiratta, and E. Murphy, "Availability evaluation of hardware/software systems with several recovery procedures," in *Proc. 29th Int. Computer Software and*

*Applications Conf. (COMPSAC'05)*. IEEE Computer Society, 2005, pp. 473–478.
[27] P. Popic, D. Desovski, W. Abdelmoez, and B. Cukic, "Error propagation in the reliability analysis of component based systems," in *Proc. 16th IEEE Int. Symp. on Software Reliability Engineering (ISSRE'05)*. IEEE Computer Society, 2005, pp. 53–62.
[28] M. Lipton and S. Gokhale, *Recent Advances in Reliability and Quality in Design*, ser. Springer Series in Reliability Engineering. Springer, 2008, vol. III, ch. Heuristic Component Placement for Maximizing Software Reliability, pp. 309–330.
[29] S. Malek, R. Roshandel, D. Kilgore, and I. Elhag, "Improving the reliability of mobile software systems through continuous analysis and proactive reconfiguration," in *Proc. 31st Int. Conf. on Software Engineering (ICSE'09)*. IEEE Computer Society, 2009, pp. 275–278.
[30] K. K. Goswami and R. K. Iyer, "Simulation of software behavior under hardware faults," in *Proc. 23rd Int. Symp. on Fault-Tolerant Computing*, 1993, pp. 218–227.
[31] B. Huang, X. Li, M. Li, J. Bernstein, and C. Smidts, "Study of the impact of hardware fault on software reliability," in *Proc. 16th IEEE Int. Symp. on Software Reliability Engineering (ISSRE'05)*. IEEE Computer Society, 2005, pp. 63–72.
[32] O. Das and C. M. Woodside, "Dependability modeling of self-healing client-server applications," in *Proc. Workshop on Software Architectures for Dependable Systems (WADS'03)*, ser. LNCS, vol. 3069. Springer, 2003, pp. 266–285.
[33] 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.*, vol. 29, no. 10, pp. 946–960, October 2003.
[34] G. N. Rodrigues, D. S. Rosenblum, and J. Wolf, "Reliability analysis of concurrent systems using LTSA," in *Proc. 29th Int. Conf. on Software Engineering (ICSE'07)*. IEEE Computer Society, 2007, pp. 63–64.
[35] V. Grassi, "Architecture-based reliability prediction for service-oriented computing," in *Proc. Workshop on Architecting Dependable Systems (WADS'05)*, ser. LNCS, R. de Lemos, C. Gacek, and A. B. Romanovsky, Eds., vol. 3549. Springer, 2005, pp. 279–299.
[36] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Parameterized reliability prediction for component-based software architectures," in *Proc. 6th Int. Conf. on the Quality of Software Architectures (QoSA'10)*, ser. LNCS, vol. 6093. Springer, 2010, pp. 36–51.
[37] L. Kapova, B. Buhnova, A. Martens, J. Happe, and R. Reussner, "State dependence in performance evaluation of component-based software systems," in *Proc. 1st Joint WOSP/SIPEW Int. Conf. on Performance Engineering (ICPE'10)*. ACM, 2010, pp. 37–48.
[38] T. Kappler, H. Koziolek, K. Krogmann, and R. Reussner, "Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering," in *Proc. Software Engineering (SE'08)*, ser. LNI, vol. 121. GI, February 2008, pp. 140–154.
[39] M. Kuperberg, K. Krogmann, and R. Reussner, "Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models," in *Proc. 11th Int. Symp. on Component Based Software Engineering (CBSE'08)*, ser. LNCS, vol. 5282. Springer, Heidelberg, October 2008, pp. 48–63.
[40] R. H. Reussner, "Automatic component protocol adaptation with the CoConut/J tool suite," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 627–639, July 2003.
[41] H. Koziolek, "Parameter dependencies for reusable performance specifications of software components," Ph.D. dissertation, Department of Computing Science, University of Oldenburg, Germany, March 2008.
[42] M. Grottke and K. S. Trivedi, "Software faults, software aging and software rejuvenation," *Journal of the Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005.
[43] H. Koziolek, B. Schlich, and C. Bilich, "A Large-Scale Industrial Case Study on Architecture-based Software Reliability Analysis," in *Proc. 21st IEEE Int. Symp. on Software Reliability Engineering (ISSRE'10)*. IEEE Computer Society, 2010, pp. 279–288.
[44] B. Schroeder and G. A. Gibson, "Disk failures in the real word: What does an MTTF of 1,000,000 hours mean to you?" in *Proc. 5th USENIX Conf. on File and Storage Technologies (FAST'07)*, 2007.
[45] "PCM: Palladio Component Model," www.palladio-approach.net, May 2011, last retrieved 2011-05-04.
[46] S. Kim, D.-K. Kim, L. Lu, and S. Park, "Quality-driven architecture development using architectural tactics," *J. Syst. Softw.*, vol. 82, no. 8, pp. 1211–1231, August 2009.
[47] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practices*. Boston, MA, USA: Addison-Wesley, 2003.
[48] J. Kienzle, "Software fault tolerance: An overview," in *Proc. 8th Ada-Europe Int. Conf. on Reliable Software Technologies*, ser. LNCS, vol. 2655. Springer, 2003, pp. 641–650.
[49] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.
[50] A. Martens, F. Brosch, and R. Reussner, "Optimising multiple quality criteria of service-oriented software architectures," in *Proc. 1st Int. workshop on Quality of service-oriented software systems (QUASOSS'09)*. ACM, 2009, pp. 25–32.
[51] A. Martens, H. Koziolek, S. Becker, and R. Reussner, "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms," in *Proc. 1st Joint WOSP/SIPEW Int. Conf. on Performance Engineering (ICPE'10)*. ACM, 2010, pp. 105–116.
[52] K. Goseva-Popstojanova, M. Hamill, and X. Wang, "Adequacy, accuracy, scalability, and uncertainty of architecture-based software reliability: Lessons learned from large empirical case studies," in *Proc. 17th Int. Symp. on Software Reliability Engineering (ISSRE '06)*. IEEE Computer Society, 2006, pp. 197–203.
[53] X. Wu and M. Woodside, "Performance modeling from software components," in *Proc. 4th Int. Workshop on Software and Performance (WOSP'04)*, vol. 29, 2004, pp. 290–301.
[54] "Reliability Prediction for Component-based Software Architectures," http://sdqweb.ipd.kit.edu/wiki/ReliabilityPrediction, May 2011, last retrieved 2011-05-04.
[55] K. Goseva-Popstojanova, M. Hamill, and R. Perugupalli, "Large empirical case study of architecturebased software reliability," in *Proc. 16th IEEE Int. Symp. on Software Reliability Engineering (ISSRE'05)*. IEEE Computer Society, 2005, pp. 43–52.
[56] "SSJ: Stochastic Simulation in Java," http://www.iro.umontreal.ca/~simardr/ssj/indexe.html, May 2011, last retrieved 2011-05-04.
[57] "Eclipse Modeling Project," http://www.eclipse.org/modeling/, May 2011, last retrieved 2011-05-04.
[58] S. Becker, "Coupled Model Transformations for QoS Enabled Component-Based Software Design," Ph.D. dissertation, University of Oldenburg, Germany, Mar. 2008.
[59] R. Milner, *Communication and Concurrency*. Prentice Hall, 1989.
[60] M. Chen, M. Lyu, and W. Wong, "Effect of Code Coverage on Software Reliability Measurement," *IEEE Trans. on Reliability*, vol. 50, no. 2, pp. 165–170, June 2001.
[61] J. D. Musa, "Operational profiles in software-reliability engineering," *IEEE Software*, vol. 10, pp. 14–32, March 1993.
[62] J. A. Whittaker and J. H. Poore, "Markov analysis of software specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, pp. 93–106, January 1993.
[63] K. Goseva-Popstojanova and S. Kamavaram, "Assessing uncertainty in reliability of component-based software systems," in *Proc. 14th Int. Symp. on Software Reliability Engineering*. IEEE Computer Society, 2003, pp. 307–320.

**Franz Brosch** is a PhD student at the FZI Research Center for Information Technology, Germany. He received his master's degree in computer science from the Karlsruhe Institute of Technology (KIT), and worked as a software developer at SEW-Eurodrive before starting his PhD. His research interests are reliability modelling and prediction of software systems, component-based software architectures, and model-driven software development. Furthermore, he is currently working on software performance and reliability prediction for service-oriented architectures within the European research project SLA@SOI.

**Heiko Koziolek** is a Principal Scientist with the Industrial Software Systems program of ABB Corporate Research in Ladenburg, Germany. He received his diploma degree (2005) and his Ph.D. degree (2008) in computer science from the University of Oldenburg, Germany. From 2005 to 2008 Heiko was a member of the graduate school 'TrustSoft' and held a Ph.D. scholarship from the German Research Foundation (DFG). He has worked as an intern for IBM and sd&m. His research interests include performance engineering, software architecture evaluation, model-driven software development, and empirical software engineering.

**Barbora Buhnova** is an assistant professor of software engineering at Masaryk University, Czech Republic. She received her PhD degree in computer science from the same university, for application of formal methods in component-based software engineering. She continued with related topics as a postdoc researcher at University of Karlsruhe, Germany, and Swinburne University of Technology, Australia. Besides construction of formal models of software systems and their quantitative analysis, she is attracted to modelling of socio-economic systems, for which she recently received a master degree in economics from Mendel University, Czech Republic.

**Ralf Reussner** is full professor of software engineering at the Karlsruhe Institute of Technology (KIT) since 2006 and scientific executive of the IT Research Centre in Karlsruhe (FZI). He graduated in Karlsruhe with a PhD in 2001 and was with the DSTC Pty Ltd, Melbourne, Australia, afterwards. From 2003 till 2006, he held a junior professorship at the University of Oldenburg, Germany, as head of a DFG funded Emmy-Noether Junior Research Group. Ralfs research group works in the area of component based software design, software architecture and predictable software quality. Ralf published over 60 peer-reviewed papers in journals and conferences, but also established and organised various conferences and workshops, including QoSA and WCOP. In addition, he acts as a PC member or reviewer of several conferences and journals, including IEEE TSE and IEEE Computer. At FZI, he consults various industrial partners in the areas of component based software, architectures and software quality.