Parameterized Reliability Prediction for Component-Based Software Architectures

Franz Brosch[1], Heiko Koziolek[2], Barbora Buhnova[3], and Ralf Reussner[1]

[1] FZI Karlsruhe, Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
[2] ABB Corporate Research, Wallstadter Str. 59, 68526 Ladenburg, Germany
[3] Masaryk University, Botanicka 68a, 60200 Brno, Czech Republic
{brosch,reussner}@fzi.de, heiko.koziolek@de.abb.com, buhnova@fi.muni.cz

Abstract. Critical properties of software systems, such as reliability, should be considered early in the development, when they can govern crucial architectural design decisions. A number of design-time reliability-analysis methods has been developed to support this task. However, the methods are often based on very low-level formalisms, and the connection to different architectural aspects (e.g., the system usage profile) is either hidden in the constructs of a formal model (e.g., transition probabilities of a Markov chain), or even neglected (e.g., resource availability). This strongly limits the applicability of the methods to effectively support architectural design. Our approach, based on the Palladio Component Model (PCM), integrates the reliability-relevant architectural aspects in a highly parameterized UML-like model, which allows for transparent evaluation of architectural design options. It covers the propagation of the system usage profile throughout the architecture, and the impact of the execution environment, which are neglected in most of the existing approaches. Before analysis, the model is automatically transformed into a formal Markov model in order to support effective analytical techniques to be employed. The approach has been validated against a reliability simulation of a distributed Business Reporting System.

1 Introduction

Software reliability is defined as the probability of failure-free operation of a software system for a specified period of time in a specified environment [1]. In practice, developers often ensure high software reliability only through software testing during late development stages. Opposed to this, architecture-based software reliability analysis ([2,3,4]) aims at improving reliability of component-based software architectures already during early development stages. This helps software architects to determine the software components mostly affecting system reliability, to study the sensitivity of the system reliability to component reliabilities, and to support decisions between different design alternatives.

To enable architecture-based software reliability analyses, reliability specifications of individual software components are required. Ideally, they are created by the component vendors. However, it is hard for a component vendor to specify a software component's reliability, because it depends not only on the

G.T. Heinemann, J. Kofron, and F. Plasil (Eds.): QoSA 2010, LNCS 6093, pp. 36–51, 2010.

component implementation, but also on factors outside the vendor's control. Besides its implementation, a software component's reliability depends on (i) its usage profile [5] (e.g., how often the component is called, which parameters are used), (ii) the reliability of external services [6] (e.g., how reliable the component's required services are), and (iii) the reliability of the execution environment [1] (e.g., how reliable the underlying middleware/hardware is). Existing reliability prediction methods, typically Markov-chain based, either do not cover all these aspects (mainly neglecting the execution environment reliability), or hard-code their influence into the model (transition probabilities), which reduces the reusability of the model in assessing architectural design alternatives.

We introduce a novel approach that takes all the above mentioned factors into account. We extend the work presented in [6] with the propagation of the usage profile throughout a component-based software architecture, as well as the availability of the underlying hardware resources. We use the Palladio Component Model (PCM) [7] as a design-oriented modelling language for component-based software architectures, and extend the PCM with capabilities for reliability prediction. Besides the inclusion of multiple influence factors to component reliability, our approach bears the advantage of providing a modelling language closely aligned with software architecture concepts (instead of Markov chains, which are then generated automatically).

Using the PCM, multiple developer roles (e.g., component developer, domain expert, etc.) can independently contribute their parts to the architectural model thus reducing the complexity of the overall task. Through parameterisation, software component reliability specifications are reusable with respect to varying system usage profiles, external services, and hardware resource allocations. Software architects can conduct reliability predictions using automatic methods.

The contributions of this paper are (i) a highly parameterized reliability model including all architectural aspects explicitly, (ii) a novel method of propagating hardware-level availability to the system-level reliability based on the real usage of the hardware, and (iii) a developer-friendly support of model creation in a UML-like notation with automatic transformation to Markov chains. The approach is validated on a case study of a distributed business reporting system. The whole approach is implemented as an Eclipse-based tool [8], supporting not only the modelling process and reliability analysis, but also the reliability simulation and sensitivity analysis, which aim to further facilitate the architecture design.

This paper is organised as follows: Section 2 surveys related work. Section 3 describes the models used in our approach and focuses on the PCM reliability extensions. Section 4 explains how to predict the reliability of a PCM instance, which includes solving parameter dependencies, generating a Markov model, and solving the Markov model. Section 5 documents the case study before Section 6 concludes the paper.

2 Related Work

Seminal work in the area of software reliability engineering [1] focussed 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] treating systems as a composition of software components. In the following, we examine these approaches regarding their modelling of the influence factors on component reliability, namely usage profile, and execution environment.

To model the influence of the usage profile on system reliability, the propagation of inputs from the user to the components and from components to other components (i.e., external calls) have to be modelled. Goseva et al. [2] state that most approaches rely on estimations of transition probabilities between software components. Cheung [5] states 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 architects 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. [10] extend Cheung's work to support different architectural styles and combined performance and reliability analysis. However, they rely on testing data or the software architecture's intuition to determine the transition probabilities. Reussner et al. [6] assume fixed transition probabilities between components, therefore their models cannot be reused if the system-level usage profile changes. Cheung et al. [11] focus on the reliability of individual components and do not include calls to other components.

Several approaches have been proposed including properties of the execution environment into software reliability models. Sharma et al. [12] provide a software performability model incorporating hardware availability and different states of hardware resources, but disregard the usage profile propagation and component dependencies. Furthermore, the approach calculates the throughput of successful requests in presence of hardware failures, but not the system reliability. The same holds for the approaches of Trivedi et al. [13] and Vilkomir et al. [14], who design complex availability models of the execution environment, but do not link it to the software level to quantify the overall system reliability.

Popic et al. [15] take failure probabilities of network connections into account, but not the failure probabilities of other hardware resources. Sato and Trivedi [16] combine a system model (of interacting system services) with a resource availability model. However, they do not include pure software failures (not triggered by execution environment), assume fixed transition probabilities among services, and do not model usage profile dependencies of services. Yacoub et al. [17] include communication link reliabilities in their approach but neglect hardware availability.

We described a preliminary work to the approach in this paper, which was not related to the PCM and did not consider hardware availability, in [18].

3 Modelling Reliability with the PCM

To provide the reader with a quick introduction to the modelling capabilities of the PCM we first discuss a simple example (Section 3.1), then describe the modelling capabilities more in detail structured according to the involved developer

roles (Section 3.2), and finally introduce our extension to the PCM to allow for reliability analysis (Section 3.3).

3.1 Example

Figure 1 shows a condensed example of a PCM instance. It is composed out of four kinds of models delivered independently by four different developer roles.

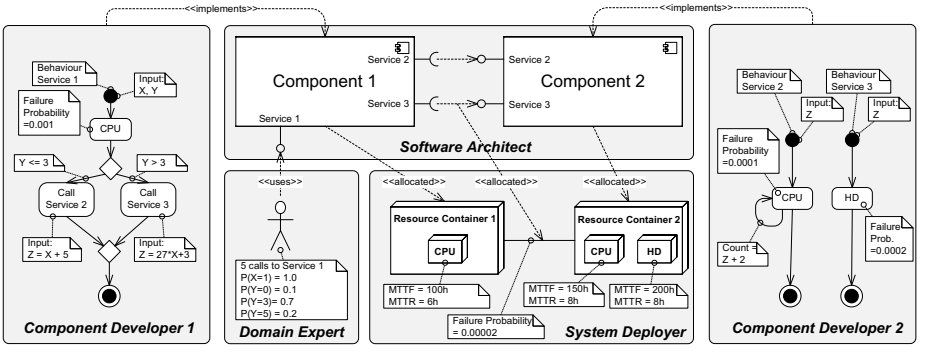

Fig. 1. PCM Example

Component developers provide abstract behavioural specifications of component services. They can annotate internal computations of a service with failure probabilities. Additionally, they can annotate external calls as well as control flow constructs with parameter dependencies. The latter allow the model to be adjusted for different system-level usage profiles. *Software architects* compose the component specifications into an architectural model. *System deployers* model the resource environment (e.g., CPUs, network links) annotated with failure properties and allocate the components in the architectural model to the resources. Finally, *domain experts* specify the system-level usage model in terms of stochastic call frequencies and input parameter values, which then can be automatically propagated through the whole model. Once the whole model is specified, it can be transformed into a Markov model to conduct reliability predictions (cf. Section 4).

3.2 Modelling Software and Hardware with the PCM

In this section, we informally describe the features of the PCM meta-model and then focus on our extensions for reliability prediction. The division of work targeted by component-based software engineering (CBSE) is enforced by the PCM, which structures the modelling task to different languages reflecting the responsibilities of the discussed developer roles.

Using the PCM, *component developers* are responsible for the specification of components, interfaces, and data types. Components can be assembled into

composite components making the PCM a hierarchical component model. For each provided service of a component, component developers can supply a so-called 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 flow of the component service. To specify parameter dependencies on control flow constructs, we have developed a so-called stochastic expression language [19], which enables modelling arithmetic or boolean operations on input parameter values. At design time developers model SEFFs manually. After implementation developers can apply static code analysis [20] or execute the component against different test cases to derive SEFFs.

Software architects retrieve the component specifications of the component developers from a repository and connect them to form an architectural model that realises a specific application. They create assembly connectors, which connect required interfaces of components to compatible provided interfaces of other components. They ideally do not deal with component internals, but instead fully rely on the SEFFs supplied by the component developers. Furthermore, software architects define the system boundaries and expose some of the provided interfaces to be accessible by users.

System deployers are responsible for modelling the resource environment, which is a set of resource containers (i.e., computing nodes) connected via network links. Each resource container may include a number of modelled hardware resources (e.g., CPU, hard disk, memory, etc.). Resources have attributes, such as processing rates or scheduling policies. 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 the component specifications.

Domain experts specify the usage model, which involves the number and order of calls to component services at the system boundaries. The model can contain control flow constructs (e.g., branches, loops). For each called service, the domain experts also characterise its input parameter values. They can use the stochastic expression language to model a parameter taking different values with specific probabilities. Once the usage model is connected to the system model by the software architect, tools can propagate the parameter values through the parameterised expressions specified by component developers. Because of the parameterisation, the usage model can easily be changed at the system boundaries and the effect on the component specifications can be recalculated.

3.3 PCM Extensions for Modelling Reliability

In this paper, we incorporate the notion of *software failures*, *communication link failures*, and *unavailable hardware* into the PCM and extend its meta model

accordingly. The following paragraphs briefly describe the rationale behind our approach.

Software failures occur during service execution due to faults in the implementation. A PCM *internal action* from a SEFF abstracts component-internal processing and can be annotated with a *failure probability*, describing the probability that the internal action fails while being executed. We assume that any failure of an internal action leads to a system failure. To estimate the failure probabilities component developers can use software reliability growth models [1], statistical testing [2], or code coverage metrics on their components. Our approach relies on these proven approaches to determine the failure probabilities. We will show in Section 5 on how to deal with uncertain failure probabilities using a sensitivity analysis.

Communication link failures include loss or damage of messages during transport, which results in 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.

Unavailable hardware causes a service execution to fail. Hardware resource breakdowns mainly result from wear out effects. Typically, a broken-down resource (e.g., a CPU, memory, or storage device) is eventually repaired or replaced by a functionally equivalent new resource. In the PCM, we annotate hardware resources with their *Mean Time To Failure* (MTTF) and *Mean Time To Repair* (MTTR). System deployers have to specify these values. Hardware vendors often provide MTTF values in specification documents. System deployers can refine these values on experience [21]. MTTR values can depend on hardware support contracts. For example, IT administration could warrant replacing failed hardware resources within one working day.

While we are aware that there are other reasons for failure (e.g., incompatibilities between components), we focus on the three failure classes described above, which in many cases have significant impact on overall system reliability. We will target further failure classes as future work.

4 Predicting Reliability with the PCM

Once a full PCM instance is specified by combining the different models described in the former section, we can predict its reliability in terms of the *probability of failure on demand* (POFOD) for a given usage model. The prediction process requires solving parameter dependencies (Section 4.1), determining probabilities of physical system states (Section 4.2), and generating and solving Markov chains (Section 4.3).

4.1 Solving Parameter Dependencies

Once the domain expert has specified input parameters in the usage model and the software architect has assembled an architectural model, a tool can propagate the parameter values of the usage model through the architectural model to solve the parameter dependencies on branch probabilities and loop counts.

The algorithm behind the tool [18] requires to separate the input domain of a component service into a finite number of equivalence classes and to provide a probability for each class. The equivalence classes can be derived using techniques from partition testing [22]. The probabilities for input classes of components directly accessed by users (i.e., the system-level usage profile) have to be estimated by domain experts. After running the algorithm, all parameter dependencies are resolved and all SEFFs contain calculated branch probabilities and loop iteration counts, which can later be used for the construction of Markov chains. We have documented the model traversal algorithm for resolving the parameter dependencies formally in [19].

Consider the example in Fig. 1. The domain expert (lower left) has specified that the parameter X for calling Service 1 will always have the value 1, while the parameter Y will take the value 0 with a probability of 10 percent, 3 with a probability of 70 percent, and 5 with a probability of 20 percent. Our tool uses the values for Y to derive the branch probabilities in the SEFFs of Component Developer 1 from the parameter dependencies $Y \leq 3$ and $Y > 30$ to 0.8 and 0.2 respectively. Furthermore, it uses the value for X $(= 1)$ to resolve the value for Z in the SEFF to $6 = 1 + 5$ for the call to Service 2 and to $30 = 27 * 1 + 3$ for the call to Service 3. In the SEFF for Service 2 (Component Developer 2), the current value for Z $(= 6)$ can be used to resolve the parameter dependency on the loop count, which is determined to be 8 according to the calculated input parameter values.

4.2 Determining Probabilities of Physical System States

After solving the parameter dependencies, our approach generates Markov chains for all possible cases of hardware resource availability. We call each of these cases a *physical system state* and calculate their occurrence probabilities from the MTTF/MTTR values specified in a PCM instance. 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). Let $s(r_i)$ be the current state of resource r_i. Then, we have:

$$P(s(r_i) = OK) = \frac{MTTF_i}{MTTF_i + MTTR_i}$$

$$P(s(r_i) = NA) = \frac{MTTR_i}{MTTF_i + MTTR_i}$$

This calculation of the resource availabilities can be refined using continuous time Markov chains (CTMC), also see [12]. 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 combination of states of all n resources:

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

As each resource has 2 possible states, there are 2^n possible physical system states, that is, $m = 2^n$. At an arbitrary point in time during system execution, let $P(s_j)$ be the probability that the system is in state s_j. Assuming independent resource failures, the probability of each state is the product of the individual resource-state probabilities:

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

Considering the example from Figure 1, there are three hardware resources included in the model (two CPUs and one HD), leading to $2^3 = 8$ possible physical system states, whose probabilities are listed in Table 1. The state probabilities are used for calculation of overall system reliability (see Section 4.3).

Table 1. Physical System State Probabilities for the Example PCM Instance

		State 1	State 2	State 3	State 4	State 5	State 6	State 7	State 8
Resource Status	CPU1	NA	NA	NA	NA	OK	OK	OK	OK
	CPU2	NA	NA	OK	OK	NA	NA	OK	OK
	HD	NA	OK	NA	OK	NA	OK	NA	OK
Probability		0,000110	0,002756	0,002067	0,051671	0,001837	0,045930	0,034447	0,861182

4.3 Generating and Solving Markov Chains

For each physical system state determined above, our tool generates a separate absorbing discrete-time Markov chain (DTMC). The chain represents all possible service execution paths, together with their probabilities, under the specific physical system state. Thus, the state (availability of hardware resources) is fixed along a system service execution, which better reflects the fact that resource failure and repair times are orders of magnitude longer than the duration of a single service. Note that this means that resources are not expected to fail or be repaired during service execution. However, this inaccuracy is negligible, which is confirmed also by the validation (see Section 5).

The DTMC is based on a combination of all SEFFs in a PCM instance triggered by the usage model. It contains a state for each action of each SEFF. Three additional states represent execution start, success, and failure. The DTMC transitions denote all possible execution paths and their probabilities.

Figure 2 illustrates the DTMC generated for the example from Figure 1, assuming that the system is in a state where both CPUs are ok, but the HD is unavailable. The execution starts with a call to Service 1, followed by an internal action requiring the first CPU. Afterwards, either Service 2 or 3 are called over the network, which then use the second CPU and the HD respectively.

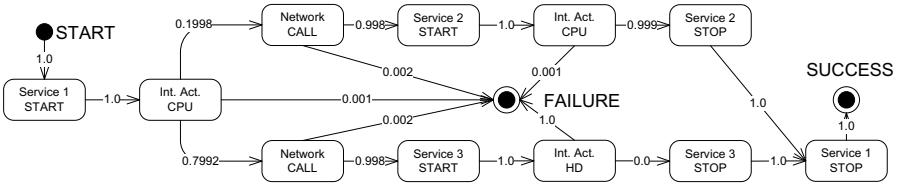

Fig. 2. Discrete-time Markov Chain

Markov states originating from internal actions and network calls have a transition to the failure state. For internal actions the transition can be triggered by either software failure (with given failure probability) or by unavailability of required hardware. Depending on the physical system state represented by the Markov chain, the failure probability of the internal action is either equal to the specified failure probability or equal to 1.0 if the required resource is unavailable. In the example, the internal action of Service 3 requires the unavailable HD and thus fails with probability 1.0. For network calls, the transition probability to the failure state is the failure probability of the communication link. In the example, each of the calls to Services 2 and 3 involves the communication link between Resource Container 1 and 2.

For each physical system state s_j, we denote $P(SUCCESS|s_j)$ as the probability of success on condition that the system is in state s_j. We calculate $P(SUCCESS|s_j)$ as the probability to reach the success state (from the start state) in the corresponding DTMC. In the example, we have $P(SUCCESS|s_j) = 0.1992$. Having determined the state-specific success probabilities, the overall probability of success can be calculated as a weighted sum over all individual results:

$$P(SUCCESS) = \sum_{j=1}^{m} (P(SUCCESS|s_j) \times P(s_j))$$

In our example, we have $P(SUCCESS) = 0.8881$.

5 Case Study Evaluation

The goal of the case study evaluation described in this section is (i) to assess the validity of our prediction results, (ii) to demonstrate the new prediction capabilities with sensitivity analyses, and (iii) to assess the scalability of our approach.

We have applied our modelling and prediction approach on the PCM instance of a distributed, component-based system (Section 5.1). To reach (i), we have predicted its reliability and compared the results to data monitored during a reliability simulation (Section 5.2). To reach (ii), we ran several prediction series, where we analysed the impact of usage profile and hardware changes on system reliability (Section 5.3). To reach (iii), we investigated the execution time for predictions based on different model sizes (Section 5.4).

5.1 Model of a Business Reporting System

Fig. 3 illustrates some parts of the so-called Business Reporting System (BRS), which is the basis for our case study evaluation (the PCM instance for the BRS can be downloaded at [8]). The model is based on an industrial system [23], which generates management reports from business data collected in a database.

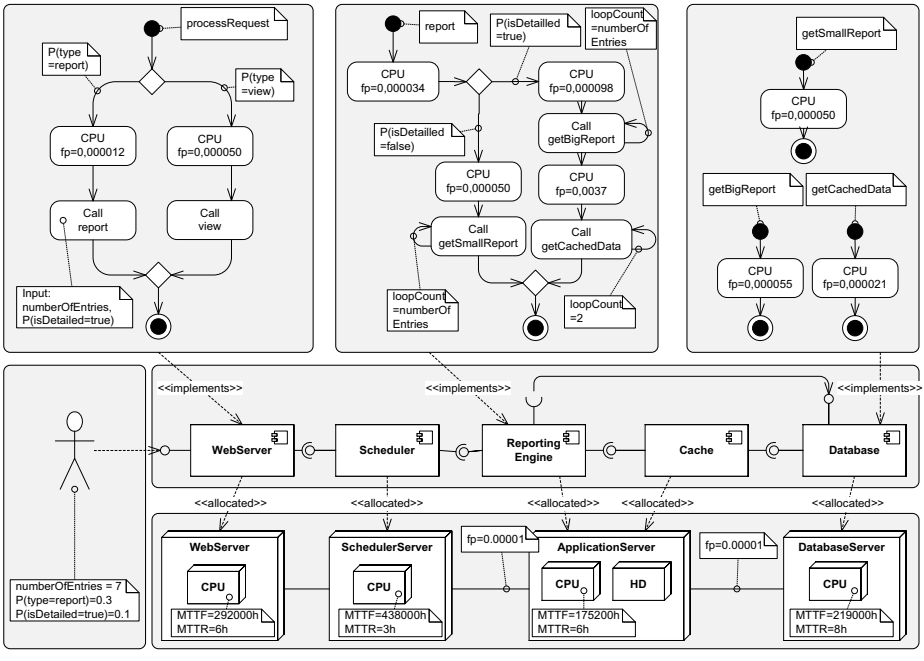

Fig. 3. PCM Instance of the Business Reporting System (Overview)

Users can query the system via web browsers. They can simply view the currently collected data or generate different kinds of reports (coarse or detailed) for a configurable number of database entries. The usage model provided by the domain expert (left hand side of Fig. 3) shows that a user requests a report in 30 percent of the cases, from which 10 percent are detailed reports. An average user requests reports for 7 database entries.

On a high abstraction level, the system consists of five independent software components running on four servers. The web server propagates user requests to a scheduler component, which dispatches them to possibly multiple application servers. The application servers host a reporting engine component, which either directly accesses the database or queries a cache component.

As failure data was not available for the system, we estimated the failure rates of the software components and hardware devices. Determining failure probabilities for software components is beyond the scope of this paper (cf. [1]). However, to make our model as realistic as possible, we used empirical data

as a basis for failure rates estimation. Goseva et al. [24] reported on failure probabilities for software components in a large-scale software system, which were derived from a bug tracking system. We aligned the failure probabilities of the internal actions in the BRS with these failure probabilities. Schroeder et al. [21] analysed the actual failure rates of hardware components of several large systems over the course of several years. Their data provided a basis for estimating the MTTF and MTTR numbers of our case study model, which are considerably lower than the ones provided by hardware vendors in specification documents.

Having the model of the system, we can use our analysis tool to predict system reliability under the given settings (see Section 5.2), or start evaluating alternative design decisions. These may include changing the usage profile, topology of software components, their deployment to hardware resources, or even replacing the components (changing their implementation) or hardware nodes (changing their parameters). Any of these local and transparent changes may significantly influence the generated Markov-chain model and hence also the predicted reliability, which is then reported to the architect to evaluate the alternatives.

5.2 Validity of the Predictions

To validate the accuracy of our prediction approach, we first executed our analytical Markov chain solver described in Section 4 and then compared the predicted system reliability to the results of a reliability simulation performed over the PCM instance of the BRS. Notice that the goal of our validation is not to justify the annotations used for reliability, like software failure probabilities or hardware MTTF / MTTR values, which are commonly used and described in literature [2,12]. Instead, we validate that if all inputs (architectural model including reliability annotations) are accurately provided, our method produces an accurate result (system reliability prediction).

For simulation purposes, we have implemented a tool based on the SSJ framework [25]. The tool uses model transformations implemented with the OAW framework 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. Software failures, communication link failures, and the effects of unavailable hardware are included into the simulation to assess system reliability.

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, and 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.

Compared to our analysis tools, simulation takes longer, but is more realistic and therefore can be used for validation. 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 (cf. [19]). Resources may fail and be repaired anytime, not only between service executions. Resource states are observed over (simulated) time, leading to more realistic failure behaviour of subsequent service executions.

Regarding the case study, our analysis tools predicted the probability of successful service execution as 0.9960837 for the usage model of the BRS sketched in Fig. 3. Because the model involves 5 resources, 32 ($= 2^5$) different Markov chains were generated to include all possible physical system states. Each generated Markov chain consisted out of 6 372 states and 6 870 transitions, because our approach involves unrolling the loops of the service effect specifications according to the specified usage profile and incorporating hardware resources. Solving the parameter dependencies, generating the different chains, and computing their absorption probabilities took less than 1 second on an Intel Core 2 Duo with 2.6 GHz and 2 GB of RAM.

To validate this result, we applied the simulation tool on the BRS PCM instance and simulated its execution for 1 year (i.e., 31 536 000 seconds of simulation time). The usage model described above was executed 168 526 times during the simulation run taking roughly 190 simulated seconds per execution. We recorded 562 internal action failures, 75 communication link resource failures and 26 resource failures during the simulation run. The simulation ran for 657 seconds (real time) and produced more than 800 MB of measured data. The success probability predicted by the simulation tool was 0.9960658, which deviates from the analytical result by approximately 0.00179 percent.

The high number of resource failures during simulation stems from the fact that we divided all given MTTF/MTTR values in the model by a constant factor. This measure allowed us to observe a statistical relevant number of over 20 resource failures during simulation, while leaving probabilities of physical system states (see Section 4.2) and the calculated system reliability unchanged.

Considering validation results, we deem the analytical method and tool implementation sufficiently accurate for the model described in this paper.

5.3 Sensitivity Analyses

To further analyse the system reliability of the BRS, we conducted several sensitivity analyses involving changing failure probabilities and usage probabilities

Fig. 4 shows the impact of different failure probabilities of component internal actions to the system reliability. The failure probabilities of the actions 'acceptView', 'prepareSmallReport', and 'getBigReport' have been varied around $fp = 0.00005$. The slopes of the curves indicate that the system reliability of the BRS under the given usage model is most sensitive to the action 'acceptView' of the web server component. This information is valuable for the software

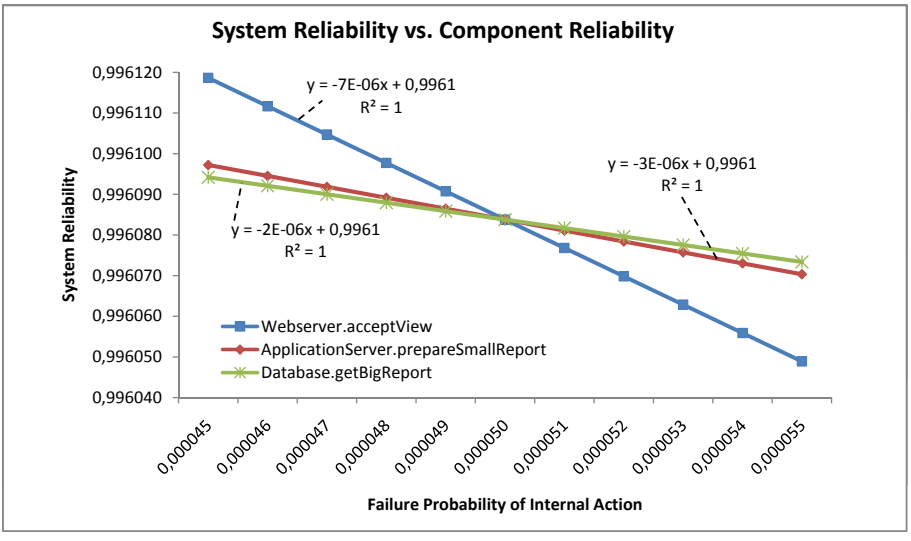

Fig. 4. Sensitivity to Failure Probabilities

architect, who can decide to put more testing effort into the web server component, to exchange the component with another component from a third party vendor, or to run the web server component redundantly.

Our parameterised behavioural descriptions allow to easily change the system-level usage model and investigate the impact on the system reliability. The parameter values are propagated through the architecture and can influence branch probabilities and loop iteration numbers. Former approaches require to change these inner component annotations manually, which is laborious and may be even hard to determine due to complex control and data flow in a large system. Fig. 5 shows the impact of different usage probabilities on system reliability. The figure suggests that the model is more sensitive to the portion of detailed reports required by the user. The impact of having more users requesting view queries is less pronounced as indicated by the lower slope of the curve.

5.4 Scalability

The scalability of our approach requires special attention. The method for incorporating hardware reliability described in Section 4 increases the number of Markov chains to be solved exponentially in relation to the number of 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.

We found that we can analyse models with up to approximately 20 resources within one hour. This involves generating and solving more than 1 000 000 Markov chains. We believe that the number of 20 different resources is sufficient for a

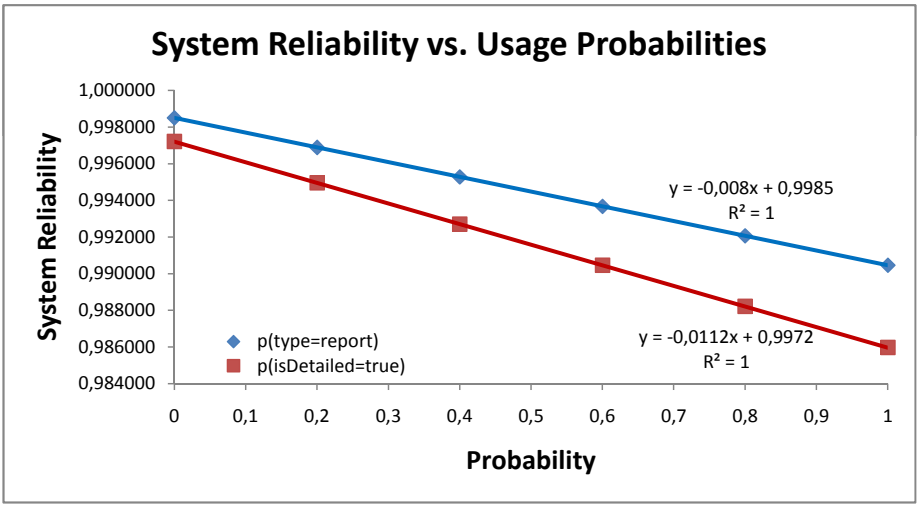

Fig. 5. Usage Profile Change 1: Usage Probabilities

large number of realistic systems. Larger models need to be analysed partially or resources have to be grouped. It is also possible to assume some resources in a large model as always available, which then decreases the effort for the predictions. Other techniques, like the possibilities for distributed analysis and multi-core processors, or employment of more efficient Markov model solution techniques,are meant for future research.

6 Conclusions

We presented an approach for reliability analysis of component-based software architectures. The approach allows for calculation of the probability of successful service execution. Compared to other architecture-based software reliability methods, our approach takes into account more influence factors, such as the hardware and usage profile. The usage profile on the system level is automatically propagated to determine the individual usage profiles of all involved software components. We have used an absorbing discrete-time Markov chain as analysis model. It represents all possible execution paths through the architecture, together with their probabilities.

The extensive parameterization of our model allows for sensitivity analysis in a straightforward way. In our case study, we examined the sensitivity of system reliability to individual failure probabilities, variations in the system-level usage profile, and changing hardware availability due to wear out effects. Furthermore, we implemented a reliability simulation to validate our results. In the case study, simulation results differed less than 0.002 percent from the analytical solution.

We will extend and further validate our approach in future work. We plan to include fault tolerance mechanisms, error propagation, concurrency modelling,

and probabilistic dependencies between individual software and hardware failures. Furthermore, we want to include the reliability of middleware, virtual machines, and operating systems into our approach. With these extensions, we aim to further increase the accurateness of our approach and support analysis for a larger class of systems.

Acknowledgments. This work was supported by the European Commission as part of the EU-projects SLA@SOI (grant No. FP7-216556) and Q-ImPrESS (grant No. FP7-215013), as well as the German Federal Ministry of Education and Research (grant No. 01BS0822).

References

1. Musa, J.D., Iannino, A., Okumoto, K.: Software reliability: measurement, prediction, application. McGraw-Hill, Inc., New York (1987)
2. Goseva-Popstojanova, K., Trivedi, K.S.: Architecture-based approach to reliability assessment of software systems. Performance Evaluation 45(2-3), 179–204 (2001)
3. Gokhale, S.S.: Architecture-based software reliability analysis: Overview and limitations. IEEE Trans. on Dependable and Secure Computing 4(1), 32–40 (2007)
4. Immonen, A., Niemelä, E.: Survey of reliability and availability prediction methods from the viewpoint of software architecture. Journal on Softw. Syst. Model. 7(1), 49–65 (2008)
5. Cheung, R.C.: A user-oriented software reliability model. IEEE Trans. Softw. Eng. 6(2), 118–125 (1980)
6. Reussner, R.H., Schmidt, H.W., Poernomo, I.H.: Reliability prediction for component-based software architectures. Journal of Systems and Software 66(3), 241–252 (2003)
7. Becker, S., Koziolek, H., Reussner, R.: The Palladio Component Model for Model-Driven Performance Prediction. Journal of Systems and Software 82(1), 3–22 (2009)
8. PCM: Palladio Component Model (January 2010), `www.palladio-approach.net` (Last retrieved 2010-15-01)
9. Wang, W.L., Pan, D., Chen, M.H.: Architecture-based software reliability modeling. Journal of Systems and Software 79(1), 132–146 (2006)
10. Sharma, V., Trivedi, K.: Quantifying software performance, reliability and security: An architecture-based approach. Journal of Systems and Software 80, 493–509 (2007)
11. Cheung, L., Roshandel, R., Medvidovic, N., Golubchik, L.: Early prediction of software component reliability. In: Proc. 30th Int. Conf. on Software Engineering (ICSE 2008), pp. 111–120. ACM, New York (2008)
12. Sharma, V.S., Trivedi, K.S.: Reliability and performance of component based software systems with restarts, retries, reboots and repairs. In: Proc. 17th Int. Symp. on Software Reliability Engineering (ISSRE 2006), pp. 299–310. IEEE Computer Society Press, Los Alamitos (2006)
13. Trivedi, K., Wang, D., Hunt, D.J., Rindos, A., Smith, W.E., Vashaw, B.: Availability modeling of SIP protocol on IBM WebSphere. In: Proc. 14th IEEE Int. Symp. on Dependable Computing (PRDC 2008), pp. 323–330. IEEE Computer Society Press, Los Alamitos (2008)

14. Vilkomir, S.A., Parnas, D.L., Mendiratta, V.B., Murphy, E.: Availability evaluation of hardware/software systems with several recovery procedures. In: Proc. 29th Int. Computer Software and Applications Conference (COMPSAC 2005), pp. 473–478. IEEE Computer Society Press, Los Alamitos (2005)
15. Popic, P., Desovski, D., Abdelmoez, W., Cukic, B.: Error propagation in the reliability analysis of component based systems. In: Proc. 16th IEEE Int. Symp. on Software Reliability Engineering (ISSRE 2005), pp. 53–62. IEEE Computer Society, Washington (2005)
16. Sato, N., Trivedi, K.S.: 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 2007), pp. 114–121. IEEE Computer Society, Los Alamitos (2007)
17. Yacoub, S.M., Cukic, B., Ammar, H.H.: A scenario-based reliability analysis approach for component-based software. IEEE Transactions on Reliability 53(4), 465–480 (2004)
18. Koziolek, H., Brosch, F.: Parameter dependencies for component reliability specifications. In: Proc. 6th Int. Workshop on Formal Engineering Approaches to Software Components and Architecture (FESCA 2009). ENTCS. Elsevier, Amsterdam (2009) (to appear)
19. Koziolek, H.: Parameter Dependencies for Reusable Performance Specifications of Software Components. PhD thesis, Department of Computing Science, University of Oldenburg, Germany (March 2008)
20. Kappler, T., Koziolek, H., Krogmann, K., Reussner, R.: Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering. In: Proc. Software Engineering 2008 (SE 2008). LNI, vol. 121, February 2008. pp. 140–154. GI (2008)
21. Schroeder, B., Gibson, G.A.: Disk failures in the real word: What does an mttf of 1,000,000 hours mean to you? In: Proc. 5th USENIX Conference on File and Storage Technologies, FAST 2007 (2007)
22. Hamlet, D.: Tools and experiments supporting a testing-based theory of component composition. ACM Transaction on Software Engineering Methodology 18(3), 1–41 (2009)
23. Wu, X., Woodside, M.: Performance modeling from software components. In: Proc. 4th International Workshop on Software and Performance (WOSP 2004), vol. 29, pp. 290–301 (2004)
24. Goseva-Popstojanova, K., Hamill, M., Perugupalli, R.: Large empirical case study of architecture–based software reliability. In: Proc. 16th IEEE Int. Symp. on Software Reliability Engineering, ISSRE 2005 (2005)
25. SSJ: Stochastic Simulation in Java (January 2010), `http://www.iro.umontreal.ca/~simardr/ssj/indexe.html` (Last retrieved 2010-01-15)