A Large-Scale Industrial Case Study on Architecture-based Software Reliability Analysis

Heiko Koziolek, Bastian Schlich, Carlos Bilich
Industrial Software Systems
ABB Corporate Research
Ladenburg, Germany
heiko.koziolek@de.abb.com

***Abstract*—Architecture-based software reliability analysis methods shall help software architects to identify critical software components and to quantify their influence on the system reliability. Although researchers have proposed more than 20 methods in this area, empirical case studies applying these methods on large-scale industrial systems are rare. The costs and benefits of these methods remain unknown. On this behalf, we have applied the Cheung method on the software architecture of an industrial control system from ABB consisting of more than 100 components organized in nine subsystems with more than three million lines of code. We used the Littlewood/Verrall model to estimate subsystems failure rates and logging data to derive subsystem transition probabilities. We constructed a discrete time Markov chain as an architectural model and conducted a sensitivity analysis. This paper summarizes our experiences and lessons learned. We found that architecture-based software reliability analysis is still difficult to apply and that more effective data collection techniques are required.**

***Keywords*-Software reliability growth, software architecture, Markov processes**

I. Introduction

Software reliability is defined as the ability of a software system to operate without failures. Analyzing software reliability during early design stages bears the potential for significant cost savings of future testing activities. Researchers have proposed several architecture-based software reliability analysis (ABSRA) methods [1]. They allow to predict system reliability based on formal, stochastic software models (e. g., Markov chains). Using these models, developers can identify critical software components and quantify their influence on the overall system reliability to optimize future testing activities.

ABB is continuously developing new release of industrial control systems. Engineering the software architectures of these systems for high reliability is a major priority. Thus, applying an ABSRA method based on failure data taken from former systems can potentially support design decisions for new systems. However, empirical research on ABSRA methods is sparse and the true benefits are widely unknown. A major issue of these methods is the complicated data collection for the models (i.e., determining component failure rates and transition probabilities), which is time-consuming and error-prone.

Multiple surveys [1]–[3] describe more than 20 methods for ABSRA. Only a handful of case studies have been reported for these methods, most of them under laboratory conditions and performed by the authors of the methods themselves [4]–[9]. The systems under study are typically small (<40 KLOC) and therefore avoid many practical problems for large-scale systems. Most of these studies use debatable techniques for determining component failure behavior (e. g., manual fault injection), and none of them provides a cost-benefit analysis. The validation is inherently difficult, because it is hard to measure system reliability. Thus, it is hard for third party users to assess whether the effort for such an analysis is justified.

This paper presents an industrial case study on ABSRA. We construct a coarse-grained discrete-time Markov chain (DTMC) model [10], [11] for a large scale industrial control system (>3 MLOC) comprising more than 100 components, which are structured into 9 subsystems. The system has been in use by customers for several years. We estimate the current subsystem failure rates from customer failure reports of the system using the Littlewood/Verrall software reliability growth model (SRGM) [12], which provided the best fit to the available data. To determine transition probabilities, we have instrumented the system and filtered large amounts of logging data. With the model, we perform a sensitivity analysis identifying the components critical for system reliability. Finally, we evaluate the model for its suitability to support design decisions for future releases of ABB control systems.

The contribution of this paper is an independent, empirical evaluation of ABSRA methods. To the best of our knowledge, the system under study is the largest system that any of these methods has ever been applied to. In contrast to former studies and due to the size of the system, we use different data collection techniques, provide insight on the applicability of the ABSRA, and perform an initial cost/benefit analysis. We discuss the advantages and drawbacks of different data collection techniques proposed in literature. Our study can help other practitioners in assessing

the benefits of these methods and direct future research to build better data collection techniques.

The paper is structured as follows. Section 2 surveys related work, before Section 3 provides a rough overview of our case study and characterizes the system under study. Section 4 describes the implementation of the case study in detail and discusses the advantages and drawbacks of different data collection techniques. Section 5 presents the results of the case study and a sensitivity analysis. Section 6 discusses the validity of the model, the overall applicability of the method, an initial cost-benefit analysis, and lessons learned. Section 7 concludes and sketches future work.

II. Related Work

Seminal research on software reliability engineering focused on system testing and system-level reliability growth models [13], but did not take software architecture into account. However, multiple recent surveys ([1]–[3]) review more than 20 methods for ABSRA (e. g., [11]). Gokhale [2] pointed out in 2007 that "very little effort has been devoted to the validation of ABSRA techniques".

To assess the situation with more detail, Tab. I compares case studies with goals similar to ours. The following paragraphs describe the techniques used in these case studies.

Table I
Overview of existing case studies

Name	Year	Language	Lines of Code	Components / Subsystems
SHARPE [4]	1998	C	35,000	30 / -
ESA [5]	2001	C	10,000	3 / -
GCC [7]	2005	C	350,000	13 / -
SMS [8]	2006	C/C++	13,000	15 / -
IDN [9]	2006	C	11,000	6 / -
ABB CS	2010	C++	$> 3,000,000$	$>$ 100 / 9

Gokhale et al. [4] analyzed the SHARPE tool (35 KLOC, C-code) for stochastic modeling by constructing a DTMC. For estimating component failure rates, they used the enhanced non-homogeneoues Poisson process model that incorporates the failure intensity of a component (i.e., 4 errors per 1 KLOC in this study) and the expected time spent in each component. The latter was determined by profiling the system with the ATAC tool while executing 735 test cases from a regression test suite. The study found that the system reliability could be increased from 0.9903 to 0.9950 if the fault density per component was reduced from 4 to 1 error per 1 KLOC.

Goseva et al. [5] performed a case study on a system (10 KLOC, C-code) of the European Space Agency (ESA). They divided the system into three subsystems and constructed a DTMC according to Cheung [11]. For estimating failure probabilities, faults discovered during integration testing and runtime were re-inserted into the software. The authors then executed random tests and estimated the reliability of each component with the ratio of failed tests versus successful tests. Finally, they used the DTMC model in a sensitivity analysis.

The largest case study reported in literature so far is also from Goseva et al. [7]. Here, the authors divided the implementation of the GCC C-compiler into 13 software components. They executed 2126 test cases from GCC 3.3.3 on GCC 3.2.3, so that 111 failures could be detected in the old version of the software, for which test cases had been added in the new version. The authors used the tool gprof to record a number of execution profiles into a database and filtered this data to determine transition probabilities between components. Finally, a DTMC was constructed and solved. The authors compared the system reliability prediction (0.9997) to the actual system reliability (0.9724). The same authors applied a similar approach on a smaller system (11 KLOC) [9].

Wang et al. [8] analyzed the so-called stock market system (SMS), which is widely used in industry (13 KLOC, C/C++ code, 15 components). They executed 13,596 test cases against the system and observed 121 failures, which they mapped to component failure probabilities. They recorded transitions between components via manual instrumentation of the the code and derived transition probabilities from this data. The authors constructed a DTMC and predicted the system reliability. None of the existing studies analyzed how the system under study could be improved and what the benefits of the model were.

Compared to existing case studies, our study analyzes a significantly larger system and thus offers more insight into the industrial applicability of ABSRA methods. Due to the size of the system, we use different methods for determining component failure probabilities and transition probabilities as in former studies. We put more emphasis on exploiting the analysis results and perform a sensitivity analyses. Our focus is on determining the cost-effectiveness of these methods.

Several other studies analyzed *failure behavior* in large software systems, but had other goals than ABSRA. Kanoun et al. [14] estimated the failure rates of four components in a telephone switching system but did not incorporate software execution profiles. Podgurski et al. [15] partitioned the execution profiles of different systems (1-17 KLOC) and combined them with non-architectural reliability prediction models. Ostrand et al. [16] predicted fault-prone files in an inventory control system (500 KLOC) without involving the software architecture.

Some recent approaches have advanced the *model expressiveness* in ABSRA. Reussner et al. [17] computed component reliabilities as a function of reliabilities of required services. Cheung et al. [18] proposed a new method for determining component reliabilities by constructing state models with explicit failure behavior. Brosch et al. [19] introduced parameter dependencies into component reliability models. However, these approaches provide only limited empirical validation.

III. Method for Architecture-based Reliability Analysis

This section describes the objectives of our case study, sketches the system under study, and provides a quick overview of the activities performed. Details about the data collection steps follow in Section IV, while details about the sensitivity analyses with the constructed model follow in Section V.

The objective of our study was to assess the applicability (i. e., in terms of learnability, modeling effort, tool support, etc.) of architecture-based software reliability engineering methods to improve architectures and reliability of software systems continuously being developed at ABB. We decided to construct an architectural reliability model based on the current version of one of these systems to evaluate different architectural alternatives for future releases of the system.

The current version of the selected ABB control system (CS) manages production processes for several industries (e. g., power generation, pulp and paper). This ABB CS is in customer use for several years. Customer failure reports can be used for modeling. The core of the system consists of more than 3,000,000 lines of C++ code and is structured into 9 subsystems, which are decomposed into more than 100 components. Because we want to start with a cost-effective model and because subsystem failure reports are available, we decided to construct a coarse-grained model on the subsystem-level, which can be refined later.

We perform our investigation in the context of the EU-project 'Quality Impact Prediction for Evolving Service-oriented Systems' (Q-ImPrESS). For reliability analysis, the project's method requires constructing a discrete time Markov chain (DTMC) according to the Cheung model [11]. Such a model reflects the components of the CS and their call relationships. As input it requires component failure rates and component transition probabilities.

Figure 1 depicts our method for collecting the necessary input data. First, we obtained access to the documentation to incorporate existing architectural documentation (step 0). To derive component failure rates (steps 1-5), we exploited the system's bug tracker and fitted a SRGM against failure report data for each subsystem (Sec. IV-A). To derive component transition probabilities (steps 6-11), we executed the system using two representative workloads and exploited internal logging facilities of the system to detect control flow among the subsystems (Sec. IV-B). The resulting DTMC is analyzed using the 'Probabilistic Symbolic Model Checker' (PRISM) [20] (step 12) (Sec. IV-C).

IV. Implementation of the Case Study

This section details on the estimation of failure probabilities (Sec. IV-A), the derivation of transition probabilities (Sec. IV-B), and the creation of the model (Sec. IV-C). For each activity, we discuss different data collection methods known from literature.

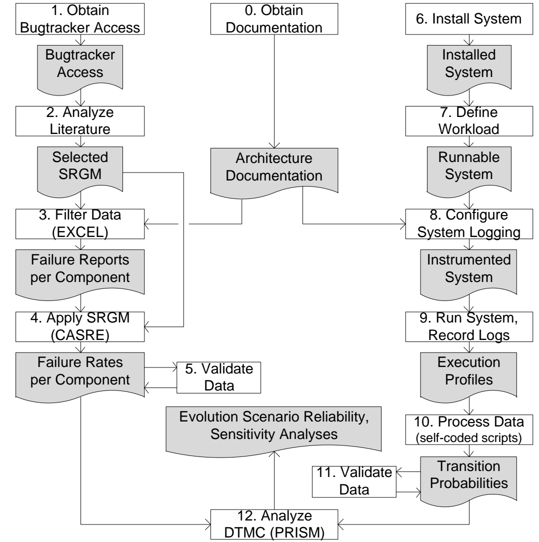
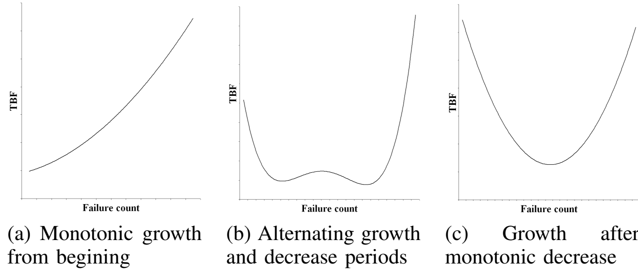


Figure 1. Process Model of the Case Study

A. Estimating Component Failure Rates

The DTMC model to be used for predictions requires component failure rates. Literature provides several methods to obtain component failure rates (see also [1]–[3], [18]):

- **Defect prediction based on code metrics [21], [22]:** compute code metrics, such as lines of code, inheritance depth, or cyclomatic complexity to estimate the number of component defects (e. g., four defects per 1 KLOC [4]). Given source code this method is easy to execute, but its validity is not proven [22] and even debated in literature [21].
- **Reliability growth modeling [13], [23]:** assume that software reliability grows over time due to bug fixes and extrapolate curves of field failure report dates to predict future failures using statistical regression. Many SRGM are available from literature [13]. However, this method is reasonably applicable only on already completed or almost completed software. Applying the method on individual components is often not possible because of limited failure reports.
- **Random/statistical testing [24]**: generate random test data for individual components and incorporate the number of successfully executed tests into a statistical failure rate estimation model. This method is applicable to any type of software and does not require source code or historical data. However, the effort for generating and executing a sufficient number of test cases is high and the method might not scale. Because of inter-component relationships, it is difficult to test components in isolation.
- **Fault injection [4], [5]:** manually insert faults into the source code or use test cases from fixed bugs on former versions of the software. The failure rates can then be estimated as the number of failed vs. the number of



(a) Monotonic growth from begining (b) Alternating growth and decrease periods (c) Growth after monotonic decrease

Figure 2. Three exemplary cases summarizing the reliability characteristics encountered in most of the subsystem of our study

successful test case executions. This method is accurate for former versions of the software and the effort can be low if suitable test cases are available. However, this method does not determine the *current* component failure rate. Additionally, it is often difficult to attribute test case failures to component faults [7].

- **Explicit failure modeling [18]:** construct a state-based behavior model per component explicitly including manually specified transition probabilities for failure states. To create such a model, user requirements, domain knowledge, and/or experience with similar software can be used. While this method is useful for newly developed components, it requires manual estimation of failure rates and its accuracy is not proven.

ABB systematically records all problems that a software product experiences during its entire life cycle in a bug tracker database. Thus, from our point of view, a failure occurs when the developer and/or the end user reports a failure. Given the availability of such data, we decided to use a SRGM to estimate the failure rates of single subsystems. In the following, we detail step 1-5 from Fig. 1.

Step 1: The ABB CS bug tracker database includes issue reports with different types of severity. For each report, it includes title, description, status, action performed (e. g., change applied, duplicate, forwarded), affected subsystem, and further information.

We only accounted failures that were fixed and assumed that the corresponding defects causing the failures were located in the same subsystem because we lacked further data about which parts of the code have been updated to fix a bug. Note that this simplifying assumption might have introduced a deviation into our model [7].

Step 2: We decided to use the IEEE Std. 1633-2008 "Recommended Practice on Software Reliability" [23] for selecting a suitable SRGM. We did a preliminary analysis of the failure behavior of a number of components of the ABB CS and found that the evolution of the time between failures (TBF) vs. failure count can be summarized in three exemplary cases (Fig. 2).

Assuming a stable usage profile, the behaviors depicted in Fig. 2b and Fig. 2c suggest that new errors are sometimes

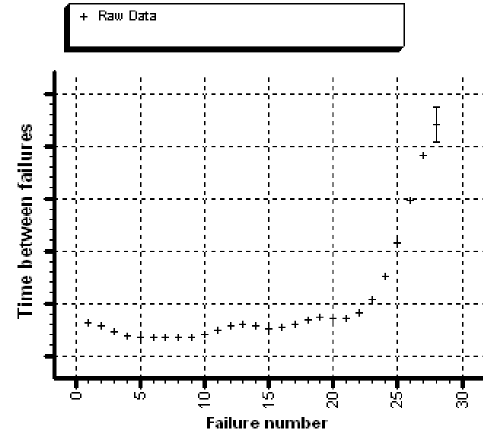


Figure 3. Time between failures of subsystem 8 (smoothed)

introduced in the component during a repair action, thus reducing its reliability. This shortens the time before next failure and consequently increases its failure rate. Therefore, an SRGM needs to be selected that takes the characteristics of these failure rates into account.

In general, SGRM are selected via a quantitative or qualitative approach: The quantitative approach uses well-known statistical tests (e. g., chi-square or Kolmogorov-Smirnov) to compute the goodness-of-fit of each model.

To reduce the complexity, we decided to use a single type of SRGM for all subsystems, which we selected according to industry affinity of former applications as suggested in IEEE Std. 1633-2008. The Littlewood/Verrall model [12] was developed from a large SCADA/DCS system (Supervisory Control and Data Acquisition / Distributed Control System) and is the only calendar TBF SRGM that accounts for both operational and imperfect fault removal uncertainty. The model also exhibited good fit to TBF data during the preliminary exploratory analysis and was therefore selected for further modeling.

Step 3: We filtered the failure reports from the bug tracker database according to the following criteria. We selected only one release, and for this release, we selected only "critical" and "high" severity failures, which are defined similar to Severity #1 and #2 in [23]. Failures in these two categories cause downtime that affects the overall availability of the system. In order to comply with the quality of assumption that "faults are immediately removed when failures are observed" [23], we selected only those failures for which a change was applied. Therefore, we only considered eight of the nine subsystems as there were no critical failures reported for one of the subsystems.

Another assumption is that the component being modeled is somehow "stable", i. e., it can run for *some* time before failing. This means that compilation and crude execution errors have been already eliminated during testing. Thus, only subsystem failure records where the failure submit date

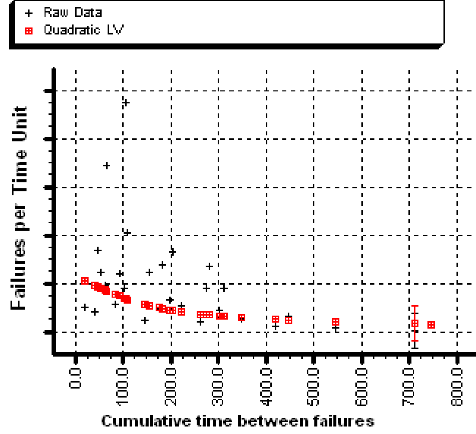


Figure 4. Failure intensity of subsystem 8 showing the original points and fitting curve

is greater than or equal to the system release date were selected.

Step 4: We chose the 'Computer Aided Software Reliability Estimation' tool (CASRE) [25] to perform our failure rate estimations. In the following, we describe exemplary how we applied the methodology for subsystem 8 of the ABB CS.

Figure 3 shows a CASRE plot of its critical and high failure history (smoothed). Notice that the unit of the time between failures has been intentionally obfuscated for confidentiality reasons.

The evolution of TBFs corresponds to the case of Fig. 2b. We were able to fit the whole dataset without filtering data at 5% significance level with the quadratic Littlewood/Verrall model (LV-Q). Fig. 4 shows a plot of the failure intensity.

Finally, the current failure intensity estimated by the model is the value used to annotate the failure rate of subsystem 8. We applied the same procedure to all subsystems of the ABB CS. To assess the influence of the growth model on the overall result, we also created estimates for each subsystem based on a simple average model (number of failures after release per time unit).

Step 5: To get a first hint of the plausibility of the subsystem failure rates predicted by the SRGM, we searched for a correlation between code metrics and failure rates [22], [26]. We compared the component failure rates against the lines of code per subsystem and the arithmetic average cyclomatic complexity per method [27].

Spearman's rank correlation coefficient ρ is low for lines of code vs. failure rate ($\rho = 0.1428$, $p = 0.7825$), but moderately high for average cyclomatic complexity vs. the failure rate ($\rho = 0.6428$, $p = 0.1389$). The slight correlation between complexities and failure rates gives us some confidence that the failure rates predicted by the SRGM are indeed representative for the current failure rates of the system.

B. Obtaining Transition Probabilities

The DTMC model additionally requires transition probabilities between software components. The following methods have been used to determine these values in former case studies [1]–[3], [7]:

- **Exploiting design documents [2]:** derive the number of component transitions from design documents (e. g., sequence or activity diagrams). This method is beneficial, when no code is available for testing. However, architectural design documents often only contain static dependencies between components. Furthermore, deriving transition probabilities from models is usually a manual process and can be time-consuming.
- **Profiling [4], [7]:** run the system with a representative workload and use a profiling tool, such as gprof, JVMPI, VTune, or ATOM to derive call graphs and transition probabilities. As the amount of data produced by a profiler is typically high (on the level of functions), the data needs to be filtered to contain only component transitions as well as control flow start and end points, which can be time-consuming [7]. This method can only be applied to already implemented systems, but the resulting transition probabilities can be very accurate if the workload used was representative.
- **Manual code instrumentation:** insert manual measurement probes into the code to log component transitions and execute the system with a representative workload. This method is similar to profiling. In contrast to profiling, the effort for setting up the execution is much higher because component transitions have to be identified in the code. The effort for filtering the logging data, however, is significantly reduced.

In our case study, we decided to use the internal logging facilities of the ABB CS, which can be seen as a special case of profiling. The advantage of using the internal logging facilities is that they are capable of only logging subsystem interactions. The disadvantage, however, is that internal logging is not available for all components. We did not use design documents because the existing ones only detail static dependencies. Manual code instrumentation was deemed too expensive because the code base is large and the subsystem transitions cannot be automatically identified. The following describes steps 6-11 shown in Fig. 1.

Step 6: First, we setup the CS, two additional servers for providing data, and three client applications to access the CS. Setup included activities such as installation, configuration, and engineering.

Step 7: To run the system, we had to define a load profile. We decided to use two realistic load profiles for single customers based on two typical use cases. They provide an approximation of different customer load profiles. One of these profiles focuses on the engineering of the industrial process while the other one concentrates on the steady state

of the continuously running process. These load profiles have been identified with the help of domain experts and are aligned with test profiles for the system.

Step 8: Before running the application, we configured an ABB development tool for logging CS system calls to record only interactions between subsystems. This involves identifying the components responsible for subsystem transitions and adjusting the log detail level for these components. Each subsystem transition represents a DCOM call between two processes. Each log entry contains information about which subsystem called which other subsystem.

Step 9: We executed each profile for two days. Running a load profile includes starting all applications (i. e., the CS, the additional servers, the logging application, and all clients), performing the initial setup, and operating the system. In profile 1, operating comprises engineering of the system as well as observing data and interacting with the system. In profile 2, operating of the system only consists of observing data and interacting with the system. Both load profiles resulted in a five-digit number of invocations, and the two log files generated in this step had a size of 2 GB each.

Step 10: The created log files were then passed to a self-coded script, which generates the list of subsystems involved, the transitions between these subsystem, and the probabilities of these transitions. Additionally, it adds an initial and a final state to indicate the beginning and the end of the executions.

The subsystems that are connected to the initial state and to the final state respectively are determined by examining their inner components. Executions start and end in components. Components that start an execution have more outgoing than incoming edges, and components that end an execution have more incoming than outgoing edges. As a subsystem has more than one component, it can start and end executions at the same time.

For each of the transitions involving the initial and the final state, the number of occurrences is stored. Then, for each subsystem, the number of transitions from the initial and to the final state is determined by summing up the corresponding transitions of their components. The distribution between the transitions of the initial state and the transitions of the final state respectively is determined by the proportion of the corresponding transitions. Figure 5 shows the result of script for the load profile 1. The result of load profile 2 is similar, but contains some less transitions and different transitions probabilities.

Step 11: We validated the data by examining the different paths through the model and matching these paths to the operations that we actually executed. During this process, we were supported by two CS experts.

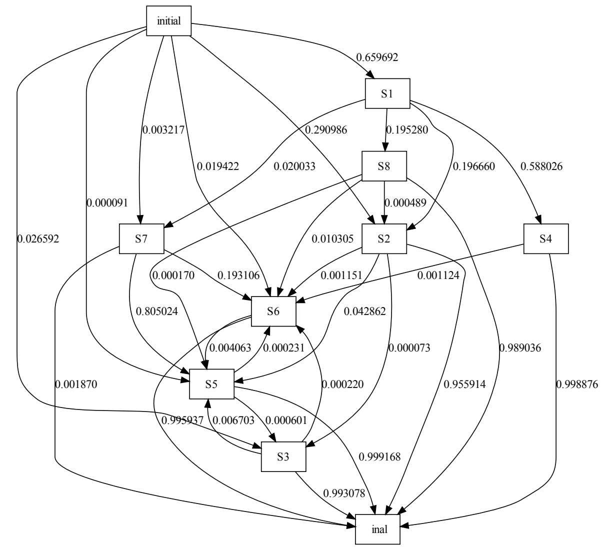


Figure 5. Subsystem transition probabilities for load profile 1

C. Constructing the Model

Step 12: To construct the DTMC, we applied another self-coded script, which takes as input the failure rates of the subsystems and the probabilities of the transitions between the subsystems. The subsystems become states in the DTMC. Each of these states denotes that control currently resides within the corresponding subsystem.

To account for failures, the script additionally adds a failure state and modifies the transition probability matrix P of the DTMC as follows. The former transition probability p_{ij} between subsystems i and j is adjusted to $(1 - f_i) * p_{ij}$, where f_i is the failure rate of subsystem i. For state i representing subsystem i a transition to the failure state with the probability f_i is introduced.

The system reliability can then be calculated by the probability of *not* reaching the failure state [11]. For the sensitivity analysis, we created 8 different PRISM models. For each subsystem, we created a model where the failure rate for this subsystem was provided as a range around the actual failure rate and the failure rates of the other subsystems were the actual failure rates.

V. CASE STUDY RESULTS

Once an architectural reliability model has been created, literature suggests the following possibilities for exploiting the models [1], [2], [11]:

- **Estimate system reliability [1]:** calculate the probability of reaching the failure state to determine the overall system reliability. Due to the assumptions for determining the failure rates and transition probabilities, it is doubtful that this value is observable by customers. However, the

Table II
SLOPES FOR THE SYSTEM FAILURE RATE CHANGES BASED ON SUBSYSTEM FAILURE RATE CHANGES

Subsystem	1	2	3	4	5	6	7	8
Profile 1 (average)	0.632644211	0.416067699	0.026638276	0.374317396	0.030238657	0.024448829	0.015708493	0.124248525
Profile 1 (LV-Q)	0.648495812	0.418800648	0.026640199	0.382279624	0.031138266	0.024706072	0.016092873	0.126920613
Profile 2 (average)	0.633371365	0.416984293	0.026440363	0.374073605	0.029502677	0.022422224	0.014346664	0.125209404
Profile 2 (LV-Q)	0.649247077	0.419700528	0.026440378	0.382053542	0.030408576	0.022545357	0.014697850	0.127869293

value could be used as a goal for testing activities when implementing a system based on the model.

- **Perform sensitivity analyses [2]:** by varying the failure rates and transition probabilities and subsequently calculating system reliability, the architectural model allows to identify the most critical components. This information can improve test budgeting by allocating more testing efforts to critical components. Also, the most critical components are most effectively used as starting points to introduce special fault tolerance measures (e. g., self-checks, recovery measures, n-version programming).
- **Assess costs of bugs per component [11]:** Cheung suggested to assign penalty costs to each component to quantify the effect of an error in that component. This information can then be used in system-wide cost calculations for errors.
- **Evaluate design alternatives [2]:** by manipulating the architectural model or introducing additional states or transitions based on estimations or experiences, different architectural alternatives (e. g., exchanging components, different allocations to hardware resource, changes in the topology) could be evaluated. However, it is hard for developers to quantify the reliability impact of most changes. Additionally, the Markov model resides on a high abstraction level and makes it hard to incorporate many technical architectural changes.

With our DTMC model of the ABB CS architecture, we analyze the overall system reliability and perform sensitivity analyses. We do not provide the actual system reliability value here for confidentiality reasons. Due to our way of estimating failure rates, the system reliability predicted from the model is lower than the reliability a specific customer of the system would observer because we included failure reports of all customers and not only failure reports of a specific customer.

The validity of the predicted value is inherently difficult to assess, as we cannot compare it with reliability measurements. Comparing the value predicted from the DTMC with the value predicted by the LV-Q model for the overall system yields a deviation of 1.94 percent (profile 1).

The results of the sensitivity analyses are depicted in Fig. 6. It shows the system failure rate over the individual subsystem failure rates. We did not check the sensitivity of system reliability to changing transition probabilities, because these cannot be changed. Each small 'X' at the center of each line represents the failure rate from the SRGM, while the line around the 'X' represents the system reliability from the sensitivity analysis by varying the failure rate by ± 10 points. The numbers located at the end of each line indicate the number of the subsystem.

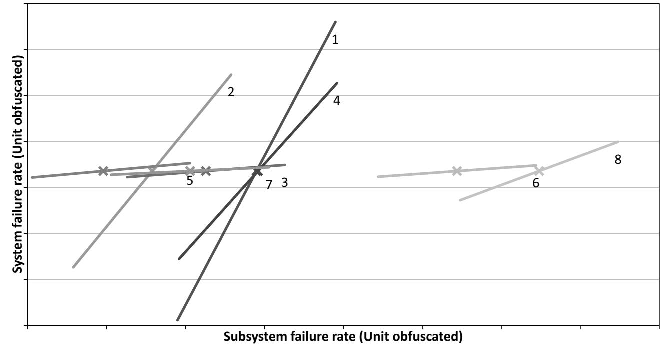


Figure 6. Sensitivity of the system failure rate depending on changes in subsystem failure rates

Subsystem 8 and 6 have the highest failure rates, while subsystem 5 has the lowest failure rate. The slopes of the curves are a measure for the sensitivity of the system failure rate to the subsystem failure rate. Tab. II summarizes these slopes for each subsystem and for the two load profiles. It contains the average prediction model and the LV-Q model.

Subsystem 1 is most sensitive for the system reliability (slope ≈ 0.65), which appears plausible because it is responsible for processing most of the data in the CS and is called most often. Subsystem 6, which is used by many subsystems (cf. Fig. 5), does not contribute much to the overall system reliability. Compared to other subsystems, this subsystem is called only a limited number of times and therefore has a limited impact on system reliability. For subsystem 8, we had estimated the highest failure rate, but it is in fact also only a minor driver for system reliability.

The evaluation of different design alternatives for future releases of the ABB CS is difficult. In the last few versions of the system, there have hardly been any changes on the subsystem level. A more refined model would be needed to evaluate the impact of different component topologies on the system reliability.

VI. DISCUSSION

In the following we discuss the validity of the model, the applicability of architecture-based software reliability anal-

ysis methods at their current maturation level in general, an initial cost/benefit analysis, and additional lessons learned.

A. Validity of the Model

Our DTMC model is valid if it can make predictions that can be verified in reality via measurements. As discussed before, it is hard to use the model to extrapolate into the future, because its abstraction level prohibits modeling many relevant architectural alternatives. For example, the model currently does not consider concurrency explicitly, and therefore we can hardly assess any alternatives for parallel executions. The model does not reflect the underlying technologies (e. g., middleware, persistency frameworks, interprocess communication) explicitly, so that it would be difficult to encode the different reliabilities of the different technologies into the subsystem failure rates.

However, we have tried to validate the individual parts of the model. The subsystem failure rates have been determined with statistical significance. A limited correlation to cyclomatic complexity could be established. The transition probabilities as well as the sensitivities have been discussed with experts of the system. The system reliability predicted by the model has been compared to the estimated of a system-wide SRGM. In conclusion, we deem the current model sufficiently accurate for the current state of the system, but are reluctant to make any future predictions based on it.

B. Applicability of the Method

The application of a method by third party users requires a documented and repeatable process as well as matured tools. In our case, we were not involved in the creation of reliability growth models, profiling tools, or Markov chain analyzers, and are therefore third party users.

We deem ABSRA still difficult to apply for non-experts, because there are many variation points (e. g., for determining failure rates) and limited documented experience. Information about different data collection techniques and prediction methods has to be gathered from various research papers. The method requires statistical skills.

ABSRA is applicable only in restricted cases when designing a new system based on a model from a legacy system. Using the method in green-field development (i. e., creating a new system from scratch) is critical, because failure data and transition probabilities would have to be guessed, which is error-prone. Applying ABSRA for the incremental evolution of legacy systems would be another viable option.

ABSRA needs to be tailored to the available failure and control flow data. There are no standard tools for determining component transition probabilities from large amounts of profiling data. As in other studies [7], we had to code our own data filtering scripts. However, other tools, such as CASRE or PRISM, are freely available and appear mature to us. They are easy to learn as comprehensive documentation is available.

#	Activity Name	Best	Likely	Worst
0	Obtain Documentation	2	4	6
1	Obtain Bugtracker Access	2	6	16
2	Analyze Literature	16	40	56
3	Filter Data	8	16	24
4	Apply SRGM (CASRE)	4	6	8
5	Validate Data	4	6	8
6	Setup System	8	16	40
7	Define Workload	4	8	24
8	Configure System Logging	12	16	20
9	Run System Record Logs	2	8	16
10	Process Data	40	80	100
11	Validate Data	4	8	24
12	Analyze DTMC (PRISM)	16	24	40
	Sum in person hours	122	238	382
	Sum in person months (168h)	0.73	1.42	2.27

Table III
EFFORT ESTIMATIONS (PERSON HOURS) FOR ACTIVITIES IN ABSRA

C. Cost/benefit Analysis

Once the third party applicability of a prediction method has been demonstrated, its claimed benefits (e. g., saving costs in future testing activities) need to be compared to its expected costs. No other study has reported on the *costs* for ABSRA [1]–[3]. In Tab. III, we provide a cost estimation (in person hours) for the steps shown in Fig. 1 under the assumption that we would repeat these steps on a similar system. Further replicated studies have to validate these estimations.

In our case, the highest efforts are required for analyzing the literature and processing the data obtained from executions of the system. Other studies (e. g., [7]) reported unquantified high effort for running test cases on the system and mapping system failures to component defects. In all reported cases, however, data collection required more effort than model creation.

Quantifying the *benefits* of ABSRA is inherently difficult, because the benefit might require years to become visible during future development. We can only qualitatively evaluate the expected benefits of architecture-based reliability prediction:

- **Saving future testing costs:** The sensitivity analyses allows for future test prioritization. More sensitive components could receive more test efforts. A fixed testing budget could be more effectively distributed among the components. This could save testing costs by achieving a desired system reliability earlier or make the system more reliable by spending the given budget more effectively. Comparing cost calculations for test effort distributions would be desirable. However, test effort distribution is usually also driven by other factors (e. g., requirements testing, code coverage).
- **Decreasing maintenance costs:** Future maintenance costs could be lowered because a higher system reliability would potentially result in fewer bug fixes after release.

In the ideal case, the architecture of a system could be optimized for software reliability and architectural design decisions could be supported quantitatively by predictions. The architecture could be engineered to support anticipated future evolution scenarios without lowering reliability. As architectural decisions have a fundamental impact on the system, the potential for cost savings could be very high. However, the currently coarse grained models do hardly allow this kind of analysis.

- **Saving costs with other analyses:** The built models and the data collection methods could be exploited for different kinds of analyses (e. g., performance prediction, maintenance prediction, cost prediction). The models could allow further simulations for future evolution scenarios. This would bring a potential for costs savings with respect to other non-functional properties.

D. Lessons Learned

Additional lessons learned while conducting our case study are summarized in the following:

- **Main obstacle is data collection:** Major efforts for constructing the DTMC have to be spent for input data collection. In comparison creating and solving the model is negligible. The method has to be tailored to the system under study and the available data. Besides the technical challenges when processing large amounts of data there are also political issues in data collection, as developers are reluctant to share sensitive information, such as failure data. Future research should focus more on data collection techniques than on model solution techniques. There could be tool sets for analyzing specific classes of systems (e.g., Java EE, .NET).
- **Influence of changes to the model are different:** The system reliability predicted by our model strongly depends on the subsystem failure rates and the placement of initial states, final states, and self-loops. However, the sensitivity analyses is hardly influenced by the differences of the two analyzed usage profiles or the different failure rate models (e. g., average vs. LV-Q model). The ordering of the slopes stay roughly the same under different failure rates or the two usage profiles. This implies that the sensitivities are mainly driven by the overall usage profile, but not the failure rates.
- **Accurate modeling is expensive:** Usually, there are no perfect data collection facilities for ABSRA for practical systems. Data is distorted, needs to be filtered, and can always be attacked for lack of validity. Thus, the resulting models reflect the real system only to a certain extent, and corresponding prediction results should be viewed critically. More refined models are more costly. It is hard to find a good trade-off between efforts for data collection and possible benefits from a model.
- **Current models are simplistic and hardly support architectural design decisions:** Current Markov models are used because they are mathematically tractable and easy to solve. Their high abstraction level however makes them less useful for decision support. Besides prioritizing future testing activities, hardly any architectural questions can be answered. Future models should be more aligned with architectural description languages (e.g., UML, AADL) and better geared towards answering architectural questions. Exact numeric solutions might not be necessary. The currently available computing power opens up many possibilities for simulation approaches.
- **No feedback for reliability improvement measures:** The models and tools do not provide guidance on how to improve a system for example by introducing fault tolerance measures or other architectural tactics. The use of the models to improve future systems is still unclear.
- **Existing empirical research in this domain is insufficient:** There are hardly any practitioner reports for ABSRA to be found in literature. Most of the existing case studies have been carried out on relatively small systems under laboratory conditions, therefore avoiding many issues for data collection in large-scale systems. Future research should lay more emphasis on third party usage of the methods to make them more robust.

VII. Conclusions

This paper presents a large-scale case study on ABSRA. We constructed a DTMC model for an industrial control system and performed a sensitivity analysis to identify the most critical subsystems. We found that the main obstacle for applying these methods in practice are missing data collection techniques and missing cost/benefits calculations. Applying this method only took approximately 2.5 person months, but as we could not measure the benefits, we cannot judge whether these efforts are appropriate. We think that applying this method is not cost-effective as long as the actual benefits cannot be measured.

Nevertheless, our study can still help both practitioners and researchers. Practitioners gain a discussion on the advantages and drawbacks of different data collection techniques and can judge whether the effort for trying the methods themselves could be justified. Researchers get new insights on the problems when applying their methods to large-scale systems.

Future research should focus on developing domain-specific data collection tool sets to speed up applying the methods. Models should be made more aligned with standard architecture description languages and geared towards answering architectural design questions. More empirical studies, third party applications, and replicated experiments are needed to bridge the gap between theory and practical application of ABSRA.

Acknowledgment

This work was supported by the European Commision as part of the EU-project Q-ImPrESS (grant No. FP7-215013).

References

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

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

[3] 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.

[4] S. S. Gokhale, W. E. Wong, J. R. Horgan, and K. S. Trivedi, "An analytical approach to architecture-based software performance and reliability prediction," *Perf. Eval.*, vol. 58, no. 4, pp. 391–412, 2004.

[5] K. Goseva-Popstojanova, A. P. Mathur, and K. Trivedi, "Comparison of architecture-based software reliability models," in *Proc. 12th Int. Symp. on Software Reliability Engineering (ISSRE'01)*, 2001, pp. 22–31.

[6] 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. Softw. Eng.*, vol. 29, no. 10, pp. 946–960, 2003.

[7] 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)*, 2005.

[8] 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.

[9] 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, 2006, pp. 197–203.

[10] B. Littlewood, "A reliability model for markov structured software," *SIGPLAN Not.*, vol. 10, no. 6, pp. 204–207, 1975.

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

[12] B. Littlewood and J. L. Verrall, "A Bayesian reliability growth model for computer software," *Applied Statistics*, vol. 22, no. 3, pp. 332–346, 1973.

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

[14] K. Kanoun and T. Sabourin, "Software dependability of a telephone switching system," in *Proc. Int. Symp on Fault-Tolerant Computing (FTCS'87)*. Washington, D.C., USA: IEEE, Jul. 1987, pp. 236–243.

[15] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff, and C. Yang, "Estimation of software reliability by stratified sampling," *ACM Trans. Softw. Eng. Methodol*, vol. 8, no. 3, pp. 263–283, 1999.

[16] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *Proc. Int. Symp. on Software Testing and Analysis (ISSTA '02)*. ACM, 2002, pp. 55–64.

[17] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo, "Reliability prediction for component-based software architectures," *J. Syst. Softw.*, vol. 66, no. 3, pp. 241–252, 2003.

[18] 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.

[19] 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.

[20] M. Z. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic Symbolic Model Checker," in *Proc. 12th Int. Conf. on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS'02)*. Springer, 2002, pp. 200–204.

[21] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 675–689, 1999.

[22] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. 28th Int. Conf. on Softw. Eng. (ICSE'06)*. ACM, 2006, pp. 452–461.

[23] IEEE, "Recommended Practice on Software Reliability, IEEE Std 1633-2008," The Institute of Electrical and Electronics Engineers, Inc., Tech. Rep., 2008.

[24] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill, and J. M. Voas, "Estimating the probability of failure when testing reveals no failures," *IEEE Trans. Softw. Eng.*, vol. 18, no. 1, pp. 33–43, 1992.

[25] M. R. Lyu and A. Nikora, "Applying reliability models more effectively," *IEEE Softw.*, vol. 9, no. 4, pp. 43–52, 1992.

[26] W. Snipes, B. Robinson, and P. Brooks, "Approximating deployment metrics to predict field defects and plan corrective maintenance activities," in *20th Int. Symp. on Software Reliability Engineering (ISSRE'09)*. IEEE, 2009, pp. 90–98.

[27] S. McConnell, *Code Complete: A practical handbook of software construction*. Buffalo, NY: Microsoft Press, 1993.