

Toward Reliability Analysis for Software Product Families

Roshanak Roshandel¹, Loizos Markides¹, Lindsay Stetson¹, Zhen Yang¹, Chris A. Mattmann², Kyle Zielinski¹

¹ Department of Computer Science and Software Engineering
Seattle University, Seattle, WA 98122
{roshanak, markides, robideau, yangz2, zielins1}@seattleu.edu

² Jet Propulsion Laboratory, California Institute of Technology,
Pasadena, CA 91109, USA
{mattmann@jpl.nasa.gov}

Abstract. The product-line approach to software development offers a promising cost-effective methodology to development and evolution of large and complex software systems. Ensuring dependability of these products requires systematic methods to evaluate various dependability properties throughout the development life cycle. In this paper, we focus on reliability – an important dependability attribute. Specifically, we are interested in developing a compositional and reusable methodology to analyze the reliability of software product line architectures (PLAs), with the goal of leveraging products’ commonalities and differences to maximize potential for reuse. We present a first step in this path, by describing an approach to abstract portions of a Markov-based reliability model corresponding to the commonalities among products, without significantly impacting the accuracy of the analysis. We evaluate our approach to assess its accuracy and generalizability on a real world PLA from the National Aeronautics and Space Administration (NASA).

Keywords: Software Product Lines, Reliability

1. Introduction and Motivation

Over the last two decades the complexity of software systems has crossed a critical threshold that necessitates a more rigorous approach to engineering software systems. *Software product lines* or *product families*¹ has emerged as a new paradigm that enables companies to exploit the commonality between software products while preserving the ability to expose different functionality in these products. The result is a systematic approach to reuse and cost effective development [25]. The reasoning is simple: software companies quite naturally develop *variants* of a product (e.g., for

¹ In the subsequent discussion, we use the terms *software product lines* and *software product families* interchangeably.

different markets). Moreover, successive development of these variants results in a proliferation of multiple *versions* of these variants. The extent of commonality among versions and variants facilitates *reuse* among a variety of artifacts.

Important *architectural design* decisions impact system dependability. Dependability properties such as *reliability, performance, availability* and *security* thus must be built into a system's design. This in turn requires approaches that bring dependability to the forefront of the development, and enable prediction and assessment of dependability properties even before implementation starts. Early dependability analysis is particularly challenging due to uncertainties associated with the system's operational profile and runtime environment.

In the context of software product families, analysis of product dependability is further challenged by the intricate relationships between various products within the family, both in terms of functionality and non-functional properties. A systematic approach to reuse of dependability analyses, results in a more efficient and cost effective approach to design and development of dependable product families. As presented later in Section 5, the state of the art in product family modeling and analysis is primarily focused on assessing the functional relationship among products. Our long term research goal aims at developing a systematic framework for tradeoff analysis of dependability properties in product families.

Our current research is focused on *architecture-based reliability analysis of software product families*. Ultimately, our goal is to enable architects to compare and contrast the impact of various design choices on the reliability of different products in a family. This work extends upon our past results in architecture-based reliability analysis of standalone software systems using a compositional approach both at the level of software components [3] and software systems [20]. We also have developed an approach for continuous runtime reliability analysis of mobile and pervasive software systems in which the software architecture changes dynamically at runtime [16]. A suitable methodology for product-line reliability analysis must be parameterized to incorporate the notion of *commonality* and *variability* in order to maximize reuse among products.

In this paper, we present our preliminary results in addressing the complexity of reliability analysis for software product line architectures (PLAs) using Markov-based models (a prominent technique used in the area). Our approach uses structural and behavioral models of a product within a family and leverages a compositional technique to reduce the complexity of reliability analysis, by focusing on commonalities and differences between products. Consequently, our approach enables partial reuse of reliability analysis results for commonalities among products.

The rest of this paper is organized as follows. In Section 2, we present necessary background information and a running example. Section 3 provides an overview of our approach. Section 4 presents the result of our evaluation. Section 5 addresses the related works. We conclude in Section 6 and discuss our future work.

2. Background

Our approach to reducing the complexity of analyzing the reliability of software product line architectures relies on existing approaches to behavioral modeling of

software systems, as well as our past research in architecture-centric reliability analysis of standalone software systems. In this section, we first provide a short description of a motivating example which will be used throughout the paper to elaborate the approach. We then provide a brief overview of our Markov-based reliability analysis methodology.

2.1. Motivating Example

The NASA's Object Oriented Data Technology (OODT) project provides a methodology, middleware, and reference architecture for the development of distributed data-intensive systems [17]. The families of software systems based on OODT offer access to geographically distributed and heterogeneous data sources by concealing the details of mediation at each data source, and offering extensible and flexible data sharing and transporting. OODT's reference architecture is comprised of several components responsible for identifying and locating data within the system, retrieving data from heterogeneous data stores, and ingesting and processing data into data stores. OODT connectors are responsible for integrating heterogeneous and third-party data sources, providing reliable messaging mechanisms, marshalling data and meta-data between components, and any security related issues.

To date, a group of large scale software products within the OODT family have been developed, including the *Planetary Data System (PDS)* and *Orbiting Carbon Observatory (OCO)* instantiations within NASA [17] as well as the National Cancer Institute's *Early Detection Research Network (ERDN)* [9]. The architecture-centric focus of OODT, along with its intentional distinction between data and meta-data enables development of flexible products across different domains, and addresses the challenges of complexity, scalability, maintainability and distribution inherent to large, complex, and distributed software systems.

The flexibility of developing new products using the OODT infrastructure makes it an appropriate testbed for evaluating our approach in reliability analysis of software product families. To clarify the concepts in this paper, we focus on two products within this family: PDS and EDNR. Figure 1 shows the high-level architecture of the two products. The shaded boxes are the reference components (commonly reused among products). Among other components, an OODT-based system consists of a set of *Clients*, one or more *Profile Handlers*, and a set of *Profile Servers*. *Clients* request a set of services that may be provided by different *Profile Servers*. The clients are oblivious to the number, type, and location of these servers. *Profile Handlers* act as mediators, and route requests and responses between Clients and Servers. A state machine representing the behavior of the PDS is depicted in Figure 2. The figure on the right zooms into a portion of the original state machine and depicts behavior of the part of the system that roughly corresponds to the functionality of the *Query* and *Profile Clients* in PDS. The functionality of the two *Profile Client* components is similar in both PDS and EDNR products. Their behavior in terms of interaction patterns and operational profile, however, may vary depending on their interactions with the rest of the system. Ultimately, the goal of our reliability analysis approach is to provide a reliability model based on commonalities and differences among products, parameterized for their interactions with the rest of the system. The first step

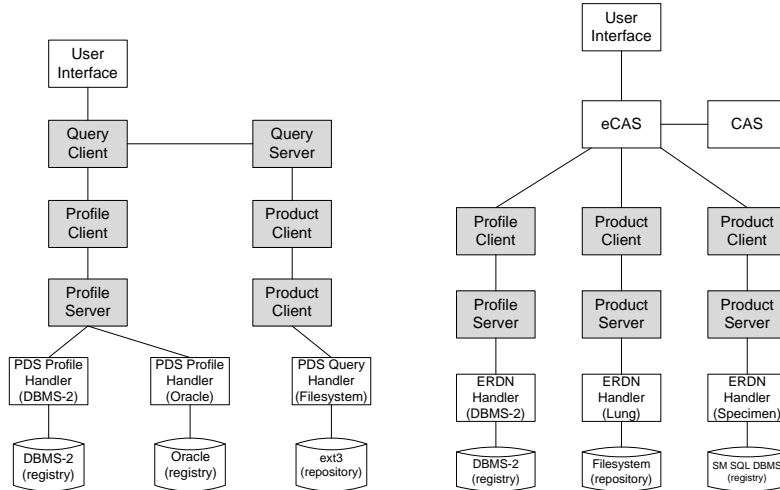


Figure 1. High-level Architecture of PDS (left) and EDRN (right) products within the OODT Product Family

in doing so is to develop a methodology that enables composing the elements of the reliability model corresponding to the *commonalities* among products. The goal is to ensure that the composition does not adversely affect the accuracy of the reliability analysis. In this paper, we discuss an approach in merging the states within the Markov-based reliability model in a way that the corresponding reliability analysis is not affected by the changes in model granularity. The details of the behavioral models may be found at [26]. In the next subsection we provide a brief overview on our approach to architecture-based reliability analysis. We then describe the details of our approach in Section 3.

2.2. Architecture-based Reliability Analysis

Structural and behavioral knowledge embedded in software architectural models provides an appropriate level of abstraction from which reasoning about a system's quality attributes is feasible. Software architectural models are typically compositional in nature: structure and behavior of complex systems are described in terms of the structure of constituent components and their interaction.

Traditionally, software systems are modeled both structurally and behaviorally. Components' configurations, along with their interfaces and corresponding pre- and post-conditions represent the system's *structure* and its *static behavior*. Dynamic models such as state-based approaches (state machines, labeled transition systems, etc.) are used to describe *dynamic behavior* of the system both in terms of its components' internal operations and overall system behavior.

Granularity of these models varies from fairly high-level (mainly at the level of individual components) to lower-level models that focus on specific behavioral states of the system at runtime.

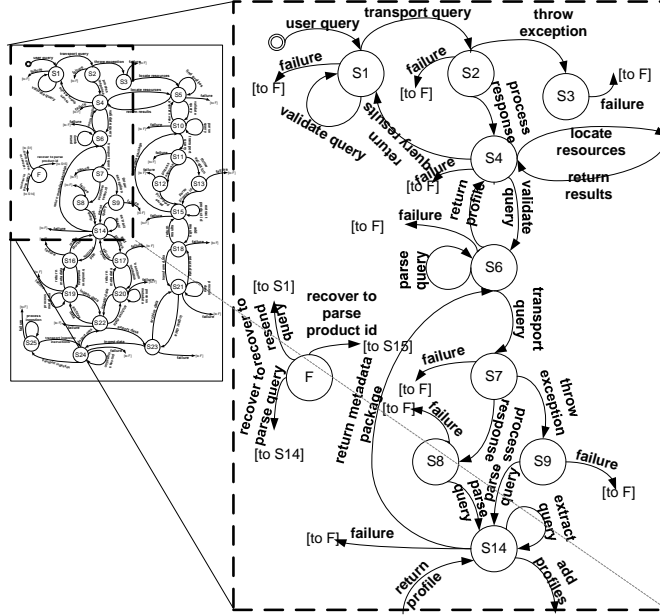


Figure 2. Behavioral Model of PDS

In our past work in compositional reliability analysis of standalone software systems, we developed an approach for early reliability prediction of software components [3], a compositional approach to reliability analysis of software systems [20], and a specialized technique for continuous reliability assessment of mobile and distributed software systems with dynamic runtime architecture [16]. The core of all these approaches is a Markov-based reliability model that leverages a behavioral model of a software component (or a system) described in a state-based modeling approach such as a state machine. In this section, we briefly describe the core of this reliability analysis approach.

Using a Markov-based approach, reliability is assessed stochastically, by estimating the fraction of time that a system operates correctly. For example, using discrete time Markov chain approach, a stochastic process with a set of states $S = \{s_1, s_2, \dots, s_N, f_1, f_2, \dots, f_F\}$ where s_i corresponds to component's/system's behavioral states and f_j corresponds to one or more failure states. The transition matrix P denotes the probability $P(y/x)$ of going from state x to state y , where $x \in S$ and $y \in S$. Reliability is then estimated by computing the steady state behavior of the system. Let $\pi(z)(t)$ be the probability that a component/system is in state $z \in S$ at time t . As t goes to infinity (i.e., as the component/system operates for a long time), these probabilities converge to a stationary distribution $\bar{\pi} = [\pi(s_1), \dots, \pi(s_N), \pi(f_1), \dots, \pi(f_F)]$ which is uniquely determined by the following equation:

$$\begin{cases} \sum_{i \in S} \pi(i) = 1 \\ \vec{\pi} = \vec{\pi}P \end{cases} .$$

This system of linear equations can be solved using standard numerical techniques [21]. The component's or system's reliability can then be defined as the probability of

not being in a failure state: $R = 1 - \sum_{i=1}^F \pi(f_i)$

Details of our approach, examples, and evaluations may be found at [3,16]. In our past work [3] we also addressed how availability of different *information sources* could impact the accuracy of analysis. For example, during early design phases given the wide range of uncertainties with system's operational profile, the analysis may be less accurate as compared to those obtained at deployment time. As demonstrated via our evaluation results, our model is flexible to incorporate various sources of information that may be available, and combine them to maximize the accuracy of analysis. For example, (1) system engineers' intuitions can be combined with (2) simulations of a component's behavior constructed from the architectural model, and (3) execution logs of functionally similar components (e.g., from a previous version of the system under construction). By combining these different information sources, we can produce candidate operational profiles for reliability prediction. Obtaining these parameters can be particularly challenging early in the design stages when implementation or simulation of the system's behavior may not be available. For those cases particularly, we rely on a variation of our approach that uses a Hidden Markov Model (HMM) methodology [19] to help estimate the transition probability parameters based on other available sources of information [3].

3. Approach

As motivated earlier, *reuse* is a fundamental motivation for adopting a product-line approach to software development. In the context of modeling and analysis of a system's functionality as well as its dependability, this translates to the ability to leverage the commonalities among different products to reduce the complexity of the analysis, without significantly impacting the accuracy of the results. The approach described in this section, takes a first step in developing a PLA-based reliability analysis framework. Specifically, we have developed a methodology where in a Markov-based reliability model a group of states and transitions may be combined, resulting in a simpler and more scalable model without significantly impacting the accuracy of the reliability analysis.

As described in Section 2.2, the states in the Markov model are directly related to the states in behavioral models of a system's software architecture. An architect may decide to combine some of these states because they correspond to the *commonalities* among products in a product line. Our approach enables merging the corresponding states in the Markov reliability model, which in turn offers the ability to reuse parts of the reliability analysis results. Our contribution includes an approach that enables

improving the scalability and reducing the complexity of Markov-based reliability analysis without significantly affecting the accuracy of results. More specifically, we present an algorithm for adjusting transition probabilities on a Markov model upon combining a set of states.

Briefly, let us assume that in a Discrete Time Markov Chain the set of states $S: \{s_1, \dots, s_n, f\}$, where f corresponds to the failure state. For simplicity and without affecting the generality of our approach, we assume that there exists a single failure state. Let us assume that M is a subset of S which represents a set of states that are intended to be merged into a single state; in other words $M \subseteq S - f$.

Furthermore, let us assume that R refers to the set of *remaining* states in the model excluding the failure state f such that $R = S - M - f$. Given the transition probability matrix P (obtained from solving the HMM model described earlier), our goal is to calculate the new transition probabilities of the state machine S' where $S' = (S - M) \cup newState$, so that $newState$ is the new combined state obtained from merging the states of M .

As an example, recall Figure 2 depicting the OODT's PDS behavioral model. The 26-state Markov model (left) depicts the PDS system behavior in terms of interactions among its components (details may be found here [26]). A partial view of the corresponding Markov Model annotated with transition probabilities is shown in Figure 3. Now, consider the set of states $M = \{S_6, S_7, S_8, S_9\}$ corresponding to the behavior of the *Profile Client* component. This component belongs to the common reference architecture of OODT and is shared among various products and thus the set M is a suitable candidate for the merge operation. The following steps describe the detail of the *StateMerge algorithm* using this example.

StateMerge Algorithm

1. The first step requires combining all of the transitions between the states within M , and then normalizing them to create a self-transition on the state $newState$.
The self-transition probability for the $newState$ is given by:

$$P(newState | newState) = \frac{\sum_{m_i \in M} \sum_{m_j \in M} P(m_i | m_j)}{|M|} \quad (1)$$

Therefore, in our example, the self-transition probability of $newState$ is equal to:

$$P(newState | newState) = \frac{\sum_{i=6}^9 \sum_{j=6}^9 P(S_i | S_j)}{4} = \frac{1.58}{4} = 0.395$$

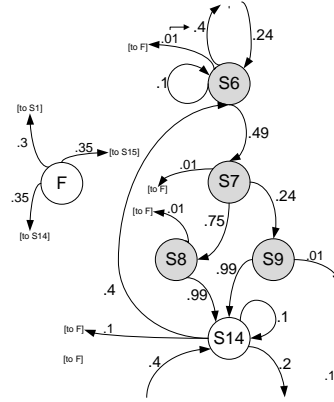


Figure 3. Partial View of the PDS Markov Model

2. Next, all of the outgoing transitions from any state $m_i \in M$ to a single state $r_j \in R$ will be combined into a single outgoing transition from the state $newState$, with assigned normalized probability equal to:

$$P(r_j | newState) = \frac{\sum_{m_i \in M} P(r_j | m_i)}{|M|} \quad (2)$$

The corresponding outgoing transition probabilities that have to be calculated for our example are the following:

$$P(S_4 | newState) = \frac{\sum_{i=6}^9 P(S_4 | S_i)}{4} = 0.1 \quad \text{and} \quad P(S_{14} | newState) = \frac{\sum_{i=6}^9 P(S_{14} | S_i)}{4} = 0.495$$

$$P(S_j | newState) = 0 \quad \text{because} \quad \sum_{i=6}^9 P(S_j | S_i) = 0 \quad \forall j = 1, 2, 3, 5, 10, \dots, 13, 15, \dots, 25$$

3. All incoming transitions from any single state $r_j \in R$ to any state $m_i \in M$ must be combined into a single incoming transition to the $newState$ with assigned probability given by:

$$P(newState | r_j) = \sum_{m_i \in M} P(m_i | r_j) \quad (3)$$

The corresponding incoming transition probabilities that have to be calculated for our example are the following:

$$P(newState | S_4) = \sum_{i=6}^9 P(S_i | S_4) = 0.24 \quad P(newState | S_{14}) = \sum_{i=6}^9 P(S_i | S_{14}) = 0.4$$

$$P(newState | S_j) = \sum_{i=6}^9 P(S_i | S_j) = 0 \quad \forall j = 1, 2, 3, 5, 10, \dots, 13, 15, \dots, 25$$

4. The last step requires adjustment of the transition probabilities between the failure state f and $newState$. Although for clarity of exposition we separated the failure state f from the set of remaining states R ; in order to adjust the probabilities between the failure state and the $newState$, the failure state is treated as any state $r_j \in R$. Thus the following two equations directly follow equations (2) and (3).

$$P(f | newState) = \frac{\sum_{m_i \in M} P(f | m_i)}{|M|} \quad (4)$$

$$P(newState | f) = \sum_{m_i \in M} P(m_i | f) \quad (5)$$

Therefore, the failure and recovery transition probabilities for the state $newState$ are equal to:

$$P(f | newState) = \frac{\sum_{i=6}^9 P(f | S_i)}{4} = 0.01 \quad P(newState | f) = \sum_{i=6}^9 P(S_i | f) = 0$$

Using this algorithm, we can generate the transition probability matrix P' corresponding to state machine S' , by calculating transition probabilities involving state $newState$. Other transition probabilities among states in R remain unchanged. In

the case of the example outlined above, solving the Markov model for the original state machines S and new state machine S' , resulted in reliability values of 95.8% and 95.69% respectively.

In more general cases when multiple sets must be merged, we have used an approach to *sequentially* apply the *StateMerge* algorithm on each set. Alternatively, a *parallel* application of the *StateMerge* algorithm may also be used in which every step of the algorithm is simultaneously

applied to all sets. Our evaluation results indicate that two approaches produce identical results; however the parallel application of the algorithm is more efficient.

In the case of the PDS example, we repeated the process sequentially for six additional sets, resulting in a reduction in the size of the original state machine S from 26 states to 11 states (less than half). Figure 4 shows the reliability values obtained after each iteration of the merge algorithm given different number of states.

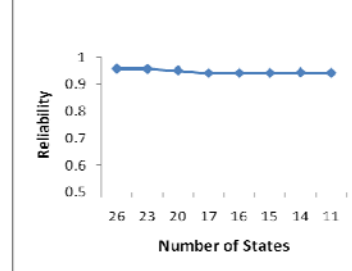


Figure 4. Reliability values for PDS example

3.1. Role of Coupling and Cohesion

In the description of the approach above, we did not discuss how the architect might select the set M to be merged. Considering that in a realistic system, there are many combinations of states that may be a candidate for the merge process, one of our long term goals is to offer the product line architects guidelines to help them identify suitable candidates.

One such approach involves selecting those states in M that correspond to commonalities among different products. Another complementary approach would be to identify candidate states for M based on the frequency of interactions among them. For example, if a group of states seem to have a high frequency of interaction with each other in comparison to their interaction with the rest of the states in the system, it may be appropriate to combine them in order to reduce the complexity of analysis. This directly relates to the notion of *coupling* and *cohesion* based on the interactions among states. Ultimately, such metrics can be used to extend the structural notion of *change sets* and *relationships* proposed by Hendrickson et. Al [10] to characterize *behavioral change sets*.

We define *cohesion* and *coupling* among different sets of states as follows. *Cohesion* of a set of states quantifies the relative strength of the relationship among states based on their frequency of interactions. The cohesion of a set M , denoted by $Coh(M)$ is assigned to be the average of the transition probabilities among the states within M . Upon merging the states into a single state *newState*, its cohesion can be calculated as its self-transition probability:

$$Coh(M) = \frac{\sum_{m_i \in M} \sum_{m_j \in M} P(m_i | m_j)}{|M|} = P(newstate | newstate) \quad (6)$$

Furthermore, we define the *coupling* of a set M to be the degree to which a set relies on interactions with states external to M . The coupling of a set M denoted by $Coup(M)$ is the average of the outgoing transition probabilities from the set M to the rest of the product states and is given by the following equation:

$$Coup(M) = \frac{\sum_{r_j \in R} \sum_{m_i \in M} P(r_j | m_i)}{|M|} = \sum_{r_j \in R} P(r_j | newstate) \quad (7)$$

The failure state probability is always excluded from this process, as we need to study the failure state in isolation. In order to satisfy the Markov principle, the following equation must be always valid for a given set M :

$$Coh(M) + Coup(M) + \frac{\sum_{m_i \in M} P(f | m_i)}{|M|} = 1$$

Using equations (6) and (7) along with the transition probability matrix P' calculated earlier, cohesion and coupling of the set M in the PDS example can be calculated as $Coh(M) = 0.395$ and $Coup(M) = 0.595$

While the selection of sets M for merge is essentially a semantic process, using the metrics discussed here, we can offer automated support to guide the merge process given a threshold for either the cohesion or the coupling metric.

4. Evaluation

Our evaluation to this date has involved preliminary experiments with NASA's OODT product family, as well as large set of simulation based experimentation. Our goal was to ensure that our results are generalizable to (1) different system models with particular interaction behaviors (e.g., highly decoupled vs. highly coupled systems), (2) merge sets of different granularities, and (3) state machines of different granularities. Moreover, we evaluated the approach (4) theoretically in terms of the complexity and scalability of the algorithm.

4.1. Generalization to different kinds of systems

Interaction behaviors of different systems may vary significantly depending on their architecture and design, resulting in Markov reliability models that are fundamentally different in terms of the kind of corresponding transition probability matrices. Using a set of experiments, we have validated that the outcome of our approach is not tied to a specific class of transition probability matrices. Specifically, we applied the approach to random full and random sparse transition probability matrices, corresponding to hypothetical systems with highly coupled and highly decoupled interaction patterns, respectively. We also performed experiments on *controlled* highly decoupled systems in which the interactions resemble a component-based design. Finally, experiments corresponding to highly coupled systems with pre-

defined sets of states to be merged (possibly corresponding to common features or components in the family) were performed.

In all of these experiments we started with an original model of size 101 states (100 behavioral state and 1 failure state). We simulated merge sets of size 20, 40, 60, 80 and 100 states and repeated this process for 50 matrices of the following four classes: (a) random full matrices,

(b) random sparse matrix (80% of the transition probabilities are equal to 0), (c) controlled matrices with pre-defined merge sets with high cohesion (average 0.8) and (d) controlled matrices with pre-defined merge sets with low cohesion (average 0.3).

Figure 5 shows the average standard deviation of the reliability values obtained from these experiments. The use of the standard deviation (SD) metric is necessary to evaluate the results, since comparison of exact reliability values calculated for different systems is meaningless. Note that the standard deviation values (along axis y) belong to the class of 10^{-5} , which is negligible, essentially validating that this approach can be applied to any kind of Markov transition probability matrix.

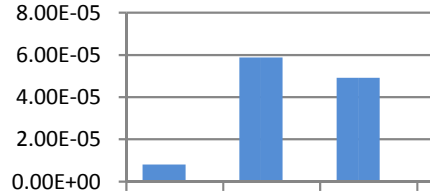


Figure 5. Standard deviation of results for different types of matrices

4.2. Granularity of Merge Sets

In this section, we evaluate our approach to ensure that the accuracy of our analysis is not tied to the size of the merge set M . Specifically, we considered the accuracy of the analysis results when merging a small set of states (e.g., only 2 states) as well as the accuracy of analysis results when merging all of the states in the state machine (excluding the failure states).

Figure 6 depicts the standard deviation (SD) of the reliability values when increasing the size of the merge set $|M|$. The boxplot represents a summary of important indicators (the *smallest* standard deviation, *first quartile*, *median*, *third quartile*, and the *largest* standard deviation) of the results. The experiment involved a behavioral model of a hypothetical system with 101 states, including a single failure state. The analysis was repeated on 50 random sparse matrices. Graph (a) shows the distribution of the SD when applying the *StateMerge* algorithm on a single merge set of increasing size (from 2 to 100). As shown, the second and third quartiles of the standard deviation results are bounded by 1.4×10^{-5} and 1.7×10^{-5} (shown as

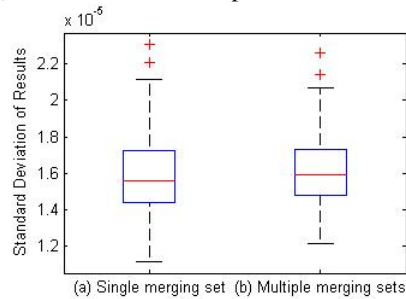


Figure 6. Standard deviation of results for different sizes of merge sets

lower and upper edges of the box). Also, as shown the extreme standard deviation result values are 1.1×10^{-5} and 2.1×10^{-5} (depicted as single lines below and above the box). Graph (b) depicts the distribution of the SD when applying the *StateMerge* algorithm on the same behavioral state machine with 101 states, but with 5 merge sets each of which varied in size from 2 to 20. As shown, the second and third quartiles of the standard deviation results are very close to the previous experiment. The extreme standard deviation result values are 1.2×10^{-5} and 2.05×10^{-5} . Again in both cases, the standard deviation results belong to the class of 10^{-5} , which is negligible. We conclude that the result of our analysis is independent from the size of the merge sets and their granularity levels.

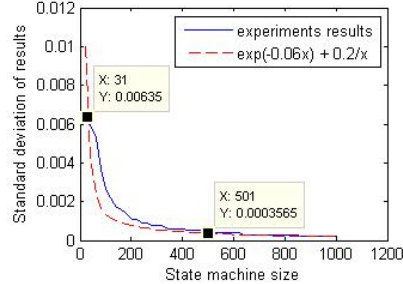


Figure 7. Average standard deviation of reliability values

4.3. Granularity of State Machines

To ensure the validity of our approach as the models grow more complex, we performed experiments on random models with an increasing number of states. The nature of the experiments, conducted for state machines of different sizes, is similar to those described in Section 4.2. Figure 7 shows the average standard deviation of the results as the number of states in the state machine increases. It is evident that as the state machine size increases, the reliability estimation values exhibit less variation. For example, as shown when the state machine consists of only 31 states, the standard deviation for the reliability values is equal to 0.00635, while in the case of a state machine of the size of 501 states, the standard deviation for the reliability values drops to 0.0003565. The dotted line in the figure depicts the graph of the equation

$$y = e^{-0.06x} + \frac{0.2}{x},$$

which seems to approximate the distribution of the resulting standard deviations. Our experiments demonstrate that our approach consistently produces accurate results regardless of the granularity of the state machines.

4.4. Approach Complexity and Scalability

The complexity of the *StateMerge* algorithm presented in Section 3 is a function of the sum of the complexity of equations (1) through (5), multiplied by the number of times they are applied. Assuming that $|S| = n + 1$ (where 1 denotes a single failure state in this case) and $|M| = m$, equation (1) is bounded by $O(m^2)$ and equations (2) through (5) are bounded by $O(m)$. Given that equations (2) and (3) will be executed once for each of the remaining states in R , the corresponding complexity changes

to $O(m \cdot |R|)$ where $|R| = n - m$.

Consequently, the complexity of our algorithm is bounded by $O(mn)$.

In cases where there is more than a single merge set, let us assume that the size of each set is equal to $m_i; i=1, \dots, k$ where k is the number of sets to be merged. Equation (1) will be repeated k times for different values of m_i , and the complexity is bounded

by $O(\sum_{i=1}^k m_i^2) < O(m_{\max} n)$ where m_{\max} is the size of the largest merge set. In this case,

the complexity of equations (2) and (3) is $O(m_{\max}/|R|) < O(m_{\max})$ with $|R| = n - \sum_{i=1}^k m_i$

resulting in an overall complexity bounded by $O(m_{\max} \cdot n)$.

Finally, we empirically evaluated the efficiency of our approach by benchmarking the elapsed time for reducing the complexity of a state machine to half of its original size. The experiment was repeated on state machines of size 100, 500 and 1000. All experiments were conducted on an Intel Core2 Duo CPU running at 2.2GHz with 3 GB RAM. The elapsed time depicted on the y-axis of Figure 8 demonstrates the efficiency of our approach even on very large Markov models.

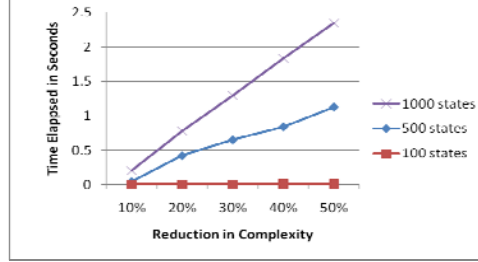


Figure 8. Time efficiency analysis results

5. Related Work

An overview of related work reveals that effective comprehensive methods for dependability analysis of product families are lacking. Specifically, Kang [13] identifies verification of quality attributes such as reliability as a major open research area in software product line engineering.

Several approaches focus on functional analysis of software product line architectures [1,5,6,8,12]. Specifically, Kaestner [12] focuses on syntactic correctness among variants and multiple implementation languages. An extension to model-driven development of product families and other generative techniques [6][5] by Czamecki et. al is aimed at implementing product lines by automatic code generation from specification. Kobra [1] (a UML-based approach) combines principles of Component-Based Engineering, Model-Driven Engineering and Product-Line Engineering. Gannod and Lutz [8] offers a series of structured analyses of an existing product line architecture using formal specification of the high-level architecture, and manual analysis of scenarios. These approaches do not explicitly focus on dependability analysis of PLAs.

There are few approaches that more directly relate to product line dependability analysis. A series of promising works by Lutz et. al [7,15,22] offer an approach and

environment to support safe evolution of safety critical product-line requirements using a model-based approach. HoPLAA – an extension to the popular ATAM (Architecture Tradeoff Analysis Method) approach [18] provides a qualitative analytical treatment of variation points, classification, and prioritization of quality attributes scenarios for PLAs. HoPLAA relies on availability of implementation and runtime operational profile for dependability analyses. Techniques for verifying safe composition properties for all programs in a product line are presented in [23]. RAP [11] is a systematic approach specialized for Product Line reliability and availability analysis, but unlike our approach, does not support compositionality and reusability of analysis results. Liu et. al. [15] focus on analysis of safety properties of software product lines. Auerswald et. al [2] present an evaluation method to predict the reliability of design alternatives during the architecture phase. Prometheus [24] is a goal-oriented methodology that addresses quality control issues, including reliability analysis but is not focused on system software architecture. Finally, an aspect-oriented approach AOPLA focuses on product specific quality attributes in PLAs [4,14]. Challenges related to compositionality and reusability of reliability analysis among products within a PLA have not been directly addressed by any of the existing approaches.

6. Conclusion and Future Work

A product-line approach to software development presents cost advantages by enabling systematic reuse. To date, majority of analysis approaches in this area have focused on analysis of products' functionality. Dependability analysis of software product lines has been identified as an open research area [13].

Our research addresses the problem of reliability analysis for software product families. Our Markov-based approach leverages structural and behavioral models of the PLAs. In this paper, we described a *StateMerge* algorithm that enables us to change the abstraction level of a Markov-based reliability model, without significantly affecting the accuracy of analysis. The long term goal of our research is to develop a reliability analysis framework that relies on architectural knowledge about the products, and the commonalities and variations among these products, in order to provide a scalable and reusable reliability assessment methodology.

Our future work includes development of a methodology to systematically model behavioral commonalities and differences among products in a product family, and apply the methodology presented here to build an environment for architecture-based reliability analysis of PLAs. Such an environment will help architects to compare and contrast the impact of various design decisions on the reliability of the entire set of products within the family. We will continue our collaborations with NASA's Jet Propulsion Laboratory to evaluate our approach using the OODT product family.

Acknowledgement. This work is supported by the National Science Foundation (award #0937472). Effort also supported by the Jet Propulsion Laboratory, managed by the California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

1. Atkinson C. Bayer J., Bunse C. Kamsties E., Laitenberger O., Laqua R., Muthig D., Paech B., Wust J., Zettel J., Component-based Product Line Engineering with UML. Addison-Wesley, London, New York, 2002.
2. Auerswald M., Herrmann M., Kowalewski S., and Schulte-Coerne V., Reliability-oriented product line engineering of embedded systems. In PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering, pages 83–100, London, UK, 2002. Springer-Verlag.
3. Cheung L., Roshandel R., Medvidovic N., Golubchik L., Early Prediction of Software Component Reliability, in Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, 2008.
4. Cortes-Verdin K., and Lemus Olalde C., Assessment of Product Line Architecture and Aspect-Oriented Software Architecture Methods, in Proceedings of the First Workshop on Aspect Oriented Product Line Engineering (AOPLE), Portland Oregon, USA, 2006.
5. Czarnecki K., Software Reuse and Evolution with Generative Techniques, in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE) Atlanta, Georgia, USA, 2007.
6. Czarnecki K., She S., Wasowski A., Sample Spaces and Feature Models: There and Back Again, in Proceedings of the 2008 12th International Software Product Line Conference (SPLC), 2008.
7. Dehlinger J., and Lutz R. PLFaultCat: A Product-Line Software Fault Tree Analysis Tool, Automated Software Engineering, v.13, 2006.
8. Gannod G., Lutz R.R., An Approach to Architectural Analysis of Product Lines, in Proceedings of the 22nd International Conference on Software Engineering (ICSE 22), Limerick, Ireland, 2000.
9. Hart A, Tran J., Crichton D., Anton K., Kincaid H., Kelly S., Hughes J.S. and Mattmann C.. An Extensible Biomarker Curation Approach and Software Infrastructure for the Early Detection of Cancer. In Proceedings of the IEEE Intl. Conference on Health Informatics , pp. 387-392, Porto, Portugal, January 2009.
10. Hendrickson, S.A. and van der Hoek, A. Modeling Product Line Architectures through Change Sets and Relationships. In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, 2007.
11. Immonen, A., A method for predicting reliability and availability at the architectural level, in Research Issues in Software Product-Lines - Engineering and Management, T. Käkölä and J.C. Dueñas, Editors. 2005, Springer.
12. Kaestner C., Apel S., Trujillo S., Kuhlemann M., Batory D., Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach , International Conf. Objects, Models, Components, Patterns, 2009.

13. Kang, K.C., Software product line research topics. In Proc. 10th International Software Product Line Conference, pp. 103–112, 2006.
14. Kiczales, G., Lamping, J., et al. Aspect-Oriented Programming. 11th European Conference on Object- Oriented Programming. p. 220-242, Finland, Jun, 1997.
15. Liu J., Dehlinger J. and Lutz R., Safety Analysis of Software Product Lines Using State-Based Modeling, *Journal of Systems and Software*, v.80, n.11, p. 1879-1892, November 2007.
16. Malek S., Roshandel R., Kilgore D., Elhag I., Improving the Reliability of Mobile Software Systems through Continuous Analysis and Proactive Reconfiguration, in proc. of the 31st International Conference on Software Engineering, New Ideas and Emerging Results, ICSE 09, Vancouver, B.C., 2009.
17. Mattmann C., Crichton D., Medvidovic N. and Hughes S., A Software Architecture-Based Framework for Highly Distributed and Data Intensive Scientific Applications. In Proceedings of the 28th International Conference on Software Engineering (ICSE06), pp. 721-730, Shanghai, China, May 2006.
18. Olumofin F.G., and Misis V. B., Extending the ATAM Architecture Evaluation to Product Line Architectures, in Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), Pittsburgh, PA, USA, 2005.
19. Rabiner L.. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. Proceedings of the IEEE, 77(2), Feb. 1989.
20. Roshandel R., Medvidovic N., Golubchik L., A Bayesian Model for Predicting Reliability of Software Systems at the Architectural Level, in Proc. of 3rd International Conference on Quality of Software Architectures (QoSA), 2007.
21. Stewart W.J., Introduction to the numerical solution of Markov Chains. Princeton University Press, 1994.
22. Sun H., Hauptman M., Lutz R., Integrating Product-Line Fault Tree Analysis into AADL Models, IEEE High Assurance Systems Engineering Symposium, 2007.
23. Thaker S., Batory D., Kitchin D., and Cook W., Safe Composition of Product Lines, Generative Programming and Component Engineering (GPCE), 2007.
24. Trendowicz A., Punter T., Quality Modeling for Software Product Lines, in 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE), 2003.
25. Weiss D.M. and Lai C.T.R, Software Product-Line Engineering, Addison-Wesley, 1999.
26. <http://fac-staff.seattleu.edu/roshanak/web/research/PLA/data/>