At the start of my academic career, a manager of a company that provides independent software verification and validation services to NASA approached me with a deceivingly simple question: Is it possible to predict software reliability before the system is completely implemented and integrated? The manager’s question likely came from his knowledge of independent verification and validation’s specific practices, which include rigorous analysis and quality-improvement activities applied early and throughout the development life cycle. Given these practices, the ability to find out early on whether a project will reach its target reliability is a logical aspiration.

Software reliability is one of the few software quality attributes with a sound mathematical definition: the probability of a software failure’s occurrence within a given period and under specific use conditions. By this definition, reliability is a strictly operational quality attribute. Traditional software-reliability prediction methods such as reliability growth models base estimates on observing failures (and fixing faults) in validation testing, during which operational patterns represent the product’s actual field use. However, most traditional methods ignore all the quality indicators collected before system integration.

A reliability prediction method that integrates quality information from such sources as architectural system descriptions, use scenarios, system deployment diagrams, and module testing lets managers identify problem areas early and make any necessary organizational adjustments.

Architecture-based techniques

Few software projects these days start development from scratch. We create systems by evolving existing ones, such as extending product lines, adding COTS modules, or just including library functions. Some components come with significant quality-related information; others come with almost nothing. An architectural drawing board can provide quality engineers with valuable information—if only they could piece together the amazing puzzle of seemingly unrelated quality indicators. Yet, when it comes to software reliability modeling techniques, most engineers don’t consider the specifics of component-based and reuse-oriented software-development paradigms.

Several techniques analyze component-based applications. These methods all aim to calculate application reliability using the reliability of individual components and their interconnections. We can broadly classify component-based estimation techniques as either state- or path-based models.1

State-based models use control graphs to depict application architecture. Software analysis tools extract these graphs from the application code. Analysts can transform a program control graph into a Markov model—a state-based rep-
representation that describes control transfer among the components and takes into account presumed component and interface failure rates.

Path-based models consider as the basis of a reliability model all possible program execution paths with frequencies, as well as those paths’ reliabilities. The models extract paths from component execution traces—meaning the executable program must be available—then multiply component reliabilities along the same path. We obtain system-level reliability from a weighted sum (based on use frequencies) of path reliabilities. Component-based models usually assume that software components fail independently of each other—a simplifying assumption that assists in modeling.

Generally, these approaches provide a good foundation for understanding component-based systems’ reliability. But few of them can be applied readily in system development’s early stages, before an executable version of the entire system is available. Therefore, my students and I decided to develop a probabilistic reliability prediction technique applicable at the design level, before the actual coding and system integration phases.

UML and software reliability

Our first step was to choose software design notation that supports reliability analysis. UML notation is the de facto toolset for software design and analysis. Because UML use is widespread, any support for reliability prediction incorporated with UML would reach numerous practitioners. Furthermore, UML notation doesn’t require developers to use a specific standard software process; designers can choose the diagrams that, in each lifecycle phase, allow the most appropriate approach to application modeling. Finally, UML diagrams’ graphical representation and the fact that the UML project is open to notational extensions let developers introduce reliability-related annotations. Annotations enrich software representation with information that supports tasks such as validating nonfunctional requirements, timing properties, or, in our case, reliability prediction. (I’ll discuss reliability annotations further in the next section.)

Software reliability analysis can exploit UML’s characteristics. Traditional software reliability prediction techniques concentrate on statistical testing performed at the end of the development life cycle. The richness of design artifacts in early development stages, supported by the UML diagrams and their annotations, can help reliability prediction.

Objectives

We set several objectives for developing our early life cycle reliability assessment model. First, the model must integrate information available in UML annotations into a software-reliability prediction technique. Typical annotations will represent known, perceived, or predicted failure rates of individual components, subsystems, and connectors. Second, the model must be able to study the sensitivity of the application reliability to component and connector. This feature lets system architects select components with suitable reliability characteristics in situations when alternative reusable assets are available. Lastly, the model must integrate reliability prediction in the system-design phase with reliability assessment based on failure occurrences observed in the system validation testing phase. This feature creates a reliability-assessment technique that spans the life cycle, from requirements modeling to deployment, corrective maintenance, and evolution.

Assumptions

While defining the reliability model for component-based applications, we made several assumptions. One assumption addresses knowledge about failure rates of software components and connections between them. Traditional component libraries don’t provide this kind of information. However, we speculate that as libraries of reusable assets mature, information about failure histories will become available, especially if the existing quality models demand it. Alternatively, prior to component selection, testing could assess candidate components’ failure rates. However, early in design stages, we can only guess component reliability. In an ideal scenario, software components would be sold with specification sheets that indicate their failure rates, much like electronic components.

Another assumption addresses the independence of failures among different components. This assumption isn’t essential, but it simplifies reliability modeling. To some extent, proposals to build COTS-based applications that include component wrappers—ensuring that the failure is caught in time and close to its source—make this assumption realistic.2

We also assume that component failures follow the principle of regularity; that is, a component is expected to exhibit the same failure rate whenever it’s invoked. The tasks that a component executes in various invocations might be different. In an object-oriented development environment, for example, we can identify invocations with the methods (entry points) of the component. So, the unit of reliability assessment could be a method rather than a component. In such cases, method-level testing could provide a failure-rate-per-demand estimate. The reliability assessment of individual methods seems impractical. At issue is the granularity of the analysis and perceived gains in the reliability prediction’s precision. If a component’s failure probability is assessed with some understanding of the expected usage profile, it should represent a “good enough” estimate for use in the system reliability model.
Reliability annotations in UML diagrams

We annotate use case diagrams, sequence diagrams, and deployment diagrams, and use these annotations for reliability modeling. A use case diagram provides a functional description of a system, its major scenarios or use cases, and its external users, called actors. Annotations of use case diagrams represent probabilities of actors invoking specific use cases. We can further subdivide the probability of choosing a use case from the scenarios that describe it.

Sequence diagrams depict how groups of components interact to accomplish a given scenario. One or more sequence diagrams describe each use case. Component interactions fall along a time axis, thus defining a partial order of execution. When an interaction enters the component—that is, when the component receives a request for service—the component becomes busy. We developed a program that automatically analyzes sequence diagrams and counts the number of busy periods for each participating component. Reliability annotations of sequence diagrams represent known or assumed failure probabilities of each participating component.

A deployment diagram describes mapping components to sites. Nodes represent platform sites (for example, workstations and servers), and links represent logical connectors (for example, LANs and WANs). Communication failures in distributed systems are detrimental to reliability. We annotate the deployment diagrams with the failure probabilities of connectors between sites and, if necessary, connectors linking components residing on the same site. Again, these probabilities are either known from current similar applications or guessed or assumed in early estimates and novel applications.

System reliability prediction

With the annotations I’ve described, we can combine component and connector failure rates and weight them with usage probabilities. We derive the probability of component use from the use case annotations. Our reliability prediction algorithm is a bit more complicated, because it lets users enter an interval for each component’s failure rate. For reused components, this interval is expected to be narrow, indicating the level of prior experience with their deployment. For unknown components (newly developed or acquired), the failure rate interval can be wider. The reliability algorithm generates beta-distribution representations of component failure rates. Figure 1 shows failure probability distributions of components in a distributed information-retrieval system that we recently analyzed.3

Analytically, we can derive system reliability from component and connector reliabilities and from operational profile information available in use case annotations. We developed a toolset.
Another use for ECRA emerges from its statistical underpinnings. After all the system components are developed and assembled, following the integration tests, validation testing can benefit from the reliability predictions obtained earlier. Bayesian statistics let the predicted system reliability distribution function serve as a prior estimate. As system failures emerge from validation testing, their frequency of occurrence can be combined with the prior estimate to form a posterior statistical distribution of system reliability. The main benefit of this use of modeling becomes obvious if the prior and posterior distributions are similar—that is, the predicted reliability was reasonably accurate. In such a case, a relatively small number of test cases will be sufficient to establish a good estimate of predeployment software reliability.

Harshinder Singh and his colleagues previously described this approach to system reliability assessment.² An architectural reliability-assessment technique already affect component-based software engineering. But, as with any modeling techniques, reliability prediction’s precision will depend strongly on the quality of modeling parameters. The ability to accurately estimate modeling parameters improves throughout the life cycle. In mature development environments, where quality control is an integral part of the software engineering process, these models can be easily integrated and used in project quality management.

**References**


Bojan Cukic is an associate professor in the Lane Department of Computer Science and Electrical Engineering at West Virginia University. Contact him at cukic@csee.wvu.edu.

---

**Figure 3. Significant component-reliability changes result in system reliability fluctuations of about 2 percent: (a) distributed information-retrieval system, (b) system reliability fluctuations.**

called Extended Component-Based Reliability Assessment, which predicts system reliability on the basis of this model. To simplify ECRA’s implementation, we used extensive Monte Carlo simulations to obtain the entire system’s reliability distribution. Figure 2 shows ECRA’s output, containing the system-level, reliability-probability density function, its mean value, and the 95-percent confidence interval.

A development organization can benefit from ECRA in several ways. One of the most common applications of component-based reliability models is analyzing the impact of reliability variations and uncertainties of individual components on the system’s overall reliability estimate. This analysis is particularly useful when the system is built partially or fully from off-the-shelf components.

For example, we used a reliability model of a distributed information-retrieval system (see Figure 3a) to discover a Web server’s impact on system reliability. We wanted to find out what we can expect if we use a Web server with a significantly higher reliability than the original server versus a remote server with a significantly lower reliability. The ECRA tool assessed the resulting system-reliability models and indicated that system reliability would in both cases vary only about 2 percent when compared with the original system configuration’s reliability (see Figure 3b). These reliabilities’ advantages (or disadvantages) need to be considered in the context of costs related to different configurations.

We can also use ECRA to identify critical components and connectors, with respect to their reliability impact, and to investigate the application reliability’s sensitivity to these elements. Early detection of critical architectural elements—those that most affect the system’s overall reliability—is useful in delegating verification and validation resources in later development phases.