Extending Reliability Block Diagrams to Software Architectures

Ahmed Abd-Allah
Center for Software Engineering
Computer Science Department
University of Southern California
Los Angeles, CA 90089 USA
aabdalla@cs.usc.edu
Technical Report: USC-CSE-97-501

ABSTRACT
Reliability block diagrams focus on components and connectors as do software architectures. However, some architectural styles possess characteristics which make traditional reliability block diagrams unusable as an analysis technique. In order to use the diagrams, they must be extended to reflect common architectural choices such as concurrency, distribution, dynamism, and implicit connectors.

Keywords
Software architecture, software reliability, reliability block diagrams

INTRODUCTION
Software architectures are typically chosen for their nonfunctional properties ([GACB95]). This choice is sometimes done in an ad hoc manner, relying on qualitative observations. However, there is a growing body of work which is attempting to provide a quantitative framework for evaluating an architecture's nonfunctional properties. One of the most important of these properties is the architecture's reliability.

There are many ways to think of the reliability of a software architecture. Software reliability is a large field, and part of the vaster field of reliability engineering. Reliability block diagrams, fault tolerance, fault estimation models, and complexity-related failure prediction are just a few examples of the breadth of software reliability work ([LYU96]). While not as broad, there is a similar group of research efforts within software architectures, including identification and formalization of architectural styles, analysis of architectural composition, and the development of architectural description languages. Drawing a correlation between software reliability and software architecture presents a broad number of choices.

For our purposes, reliability block diagrams appear to be a good starting point to examine the reliability of software architectures because of the similarity to the 'components and connectors' emphasis of software architectures. These diagrams can be used to provide a sensitivity analysis of different competing architectures, and they are also amenable to the attachment of other nonfunctional information such as performance or cost figures.

In this report, we show how reliability block diagrams can be used to estimate the reliability of different software architectures. We show that there is architectural information which does not allow the straightforward use of reliability block diagrams, and so the latter must be extended to allow for important architectural characteristics - otherwise the resulting reliability estimate may be severely flawed.

We begin by presenting a brief background of reliability block diagrams and of software architectures and styles. Next, we show how certain properties of some software architectural styles cannot be modeled using classic reliability block diagrams, and how to extend the diagrams to capture those properties. A small example is given which highlights this analysis. Finally, we summarize the primary issues, and point out future directions.

RELIABILITY BLOCK DIAGRAMS
A reliability block diagram (RBD) is a graphical depiction of the system's components and connectors which can be used to determine the overall system reliability given the reliability of its components (figure 1). An RBD has one or

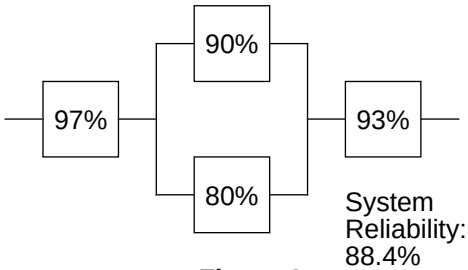


Figure 1

more paths through it which represent successful system operation. Every path is constructed out of blocks (components) and lines (connectors). Blocks represent computational elements, and the lines are used to describe which paths through the blocks are essential to success. If any path is executed successfully, then the overall system is said to succeed, otherwise if all paths fail, then the overall system also fails.

There are several important assumptions that accompany RBD's ([BREA95]):

- the reliability of each individual block is known or estimated

- the lines have a reliability of one
- all lines share the same semantics (type-less)
- failures of blocks are statistically independent
- blocks are bimodal: they either operate or fail completely (degradation of service is not allowed for)
- all success paths are shown in the diagram

Given these assumptions, it is possible to calculate the reliability of a system expressed in an RBD.

Using Failure Rates to Express Reliability

The reliability of each component or block in an RBD can be given in a variety of ways: mean time to failure, reliability percentage over a period of time, failure rate, etc. Later in this report, we assume that failure rates are associated with each component. From a component's failure rate λ, we can derive the reliability of that component over a period of time T using the following equation (assuming a homogeneous failure rate):

$$R = e^{-\lambda T}$$

Thus, if a component is estimated to have a failure rate of 10 failures per 1000 hours, then its reliability over a 24 hour period is calculated to be approximately 79%.

Even if a component's failure rate is measured meticulously using extended benchmarks and simulations, it may still need to be modified when it is actually placed in a real system. In fact, the estimated failure rate λ of a component can be expressed as a function of at least four parameters ([MIO87]):

- *f*, the original measured failure rate of the component
- *t*, the fraction of time spent in the component (with respect to the overall system execution time)
- *u*, the utilization or the fraction of CPU cycles consumed by the component
- *s*, the relative speed of the underlying platform with respect to the platform on which the failure rate was originally measured

The product of these four parameters yields a more accurate estimate of the component's failure rate than the original measured failure rate:

$$\lambda = ftus$$

Common Patterns in RBD's

One of the simplest examples of an RBD is a system which is composed purely out of a chain of components (blocks). The overall reliability of such a pipeline can be calculated as the probability of all components executing successfully, or the product of the individual reliabilities (figure 2).

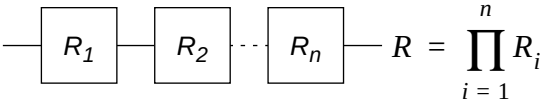


Figure 2

Another simple example is a system of components running in parallel. In this case, the overall reliability is 1 minus the probability of all the components failing (figure 3).

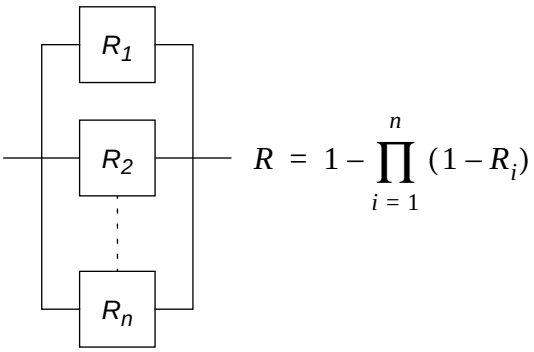


Figure 3

The serial and parallel configurations are useful, and they turn up in many different systems, however there are many systems which cannot be reduced to alternating sequences of serial and/or parallel subsystems. Those types of systems can still be analyzed by using a Karnaugh map which reflects all the paths through the system, and which identifies which paths are successful and which are not. Each path can be expressed in terms of the components which it does or does not traverse, and this information is used to generate a reliability estimate for that path in terms of a simple product of terms. For each component which is traversed, we factor in that component's reliability, and for each component which is not traversed, we factor in that component's probability of failing (or one minus its reliability). The overall reliability of the system is given by the sum of the reliabilities of the successful paths through the RBD (see [MIO87] for an example).

SOFTWARE ARCHITECTURES AND STYLES

A software architecture is composed of many views which revolve around the components and connectors in the system ([GACB95]). Traditional views include structure, topology, and behavior, and these have been modeled in many different ways (see [ABDA96] for a bibliography). Many systems are known to possess similar architectures, and this has led to the identification of architectural styles ([GASH93]). Each style is a collection of attributes and constraints which are shared by all instantiations or architectures of that style.

There are many examples of known styles (table 1), and there are likely many others to be identified. It is not immediately clear how the different styles are related to one another, especially since many of them have attributes and constraints which are apparently orthogonal to one another. For example, a layered architecture describes a general topological constraint with respect to some type of connector, whereas an event-based system places constraints on the type of communication which is allowed (messages or events) as well as specifying the presence of implicit invocations.

The orthogonality of the styles is an obstacle if we are to determine a reliability estimate for different architectures.

Table 1: Examples of Architectural Styles

Main/Subroutine	Pipe & Filter
Layered	Software Bus
Multi-threaded	Feedback System
Rule-based	Blackboard
Distributed Processes	Event-based
Object-oriented	Logic Computing
Database	Real-time System

For that reason, we utilize a previously established common foundation of architectural styles to surmount this problem ([ABDA96]). This foundation is composed of two different parts: - a base set of architectural entities - a recurring set of conceptual features

The first part observes that there are essentially eight base entities which underly many different styles. These entities are: port, data component, control component, object, data connector, control connector, trigger, and system. Style-specific entities can be derived from these eight by adding additional constraints. For example, a pipe in the pipe and filter style is a refinement of the base entity data connector. Similarly, an event queue in the event-based style is a refinement of the base entity port.

The second part of the common foundation is also an observation that certain attributes and constraints occur over and over again across many different styles. These recurring attributes and constraints can be used to identify a set of seven conceptual features which act as the dimensions of an architectural style space (table 2). Each feature has a small set of choices associated with it, and it is mainly the choices in these features which determine the style which is being applied in a particular system ([ABDA96]).

Table 2: Recurring Architectural Conceptual Features

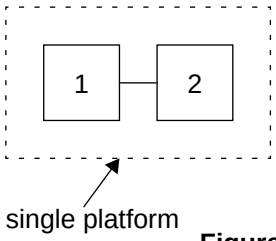
Feature	Choices
Dynamism of Computations	static, dynamic
Supported Data Transfers	implicit global data distributor, explicit data channels, shared variables
Triggering Capability	yes/no
Concurrency of Computations	single-threaded, multi-threaded
Distribution	single node, multiple nodes
Layering	yes (connector-specific), no
Encapsulation	yes/no

IMPACT OF ARCHITECTURE ON RBD's

Certain software architectural styles possess characteristics which make them difficult to model using reliability block diagrams. Though both software architectures and RBD's seem to share a common focus on 'boxes and lines', there are enough semantic differences between the two that one cannot assume a simple mapping.

The common foundation described above can be used to highlight the issues which make classic RBD's incompatible with certain styles. The incompatibility invariably stems from a particular style's noncompliance with one or more of the assumptions for RBD's. In the discussion below, we examine the impact of the base entities and the conceptual features on the failure rates of components in an RBD.

Concurrency is a recurring conceptual feature in many architectures. From a reliability standpoint, the immediate effect of running two components concurrently on a single platform is to reduce the CPU utilization of each component (figure 4). This means that the effective failure rate of each

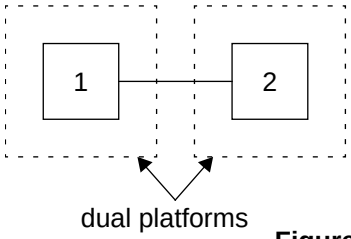


$$\lambda_1 = f_1 t_1 u_1 s$$
$$\lambda_2 = f_2 t_2 u_2 s$$
$$u_1 + u_2 \leq 1$$
$$\lambda_{system} = \lambda_1 + \lambda_2$$

Figure 4

component will be reduced, and consequently the reliability will be increased (with a performance penalty as the tradeoff).

Another conceptual feature, *distribution*, has the opposite effect of concurrency. A simple distributed system built out of two components where each component runs on its own separate platform will allow each component to reach maximum utilization of their respective CPU's (figure 5).



$$\lambda_1 = f_1 t_1 u_1 s_1$$
$$\lambda_2 = f_2 t_2 u_2 s_2$$
$$u_1 + u_2 \leq 2$$
$$\lambda_{system} = \lambda_1 + \lambda_2$$

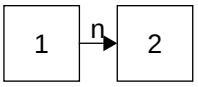
Figure 5

The additional processing power allows more instructions to be executed, increasing the effective failure rate and decreasing the reliability (with a performance gain as the tradeoff in this case).

Dynamism is a third conceptual feature which affects

RBD's, and the first one to really break one of the assumptions of RBD's. It is a conceptual feature which is also reflected in the set of base entities because the latter includes as one of its members control connectors which can be non-blocking spawns. These spawns give an architecture the ability to be dynamic. However, the presence of different connectors violates the type-less connector assumption of RBD's, and the presence of spawns in particular poses certain difficulties.

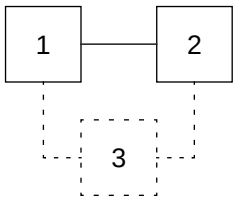
A spawn connector between two components can be traversed more than once during runtime, producing many copies of the same component running concurrently (with concurrency's utilization-limiting effect noted above). As more copies are instantiated, the effective failure rate will increase if we assume for the moment that these components are running on a lightly loaded machine or on different machines entirely. Thus, whereas an RBD is usually used to give a point estimate of a system's reliability, a dynamic architecture with spawns will have its reliability given as a function of how many copies of a component are expected to be instantiated (figure 6).



$$\lambda_1 = f_1 t_1 u_1 s_1$$
$$\lambda_2 = f_2 t_2 u_2 s_2$$
$$\lambda_{system} = \lambda_1 + n\lambda_2 \quad (n \geq 0)$$

Figure 6

One of the main choices present in software architectural styles is the type of *supported data transfers*, another conceptual feature. If a style utilizes an implicit global data distributor (e.g. the event manager in the event-based style), this may violate the 'all components shown' assumption of RBD's. The implicit distributor is a useful architectural abstraction, however the reality is that there is some hidden component which is managing the transfer of data between all the 'explicit' components (figure 7). For some types of



$$\lambda_1 = f_1 t_1 u_1 s_1$$
$$\lambda_2 = f_2 t_2 u_2 s_2$$
$$\lambda_3 = f_3 t_3 u_3 s_3$$
$$\lambda_{system} = \lambda_1 + \lambda_2 + \lambda_3$$

Figure 7

systems (e.g. specialized embedded applications), ignoring the reliability of the implicit distributor is unrealistic. Hence if there is a global data distributor, it needs to be explicitly shown in the RBD, as it may potentially increase the effective failure rate of the overall system.

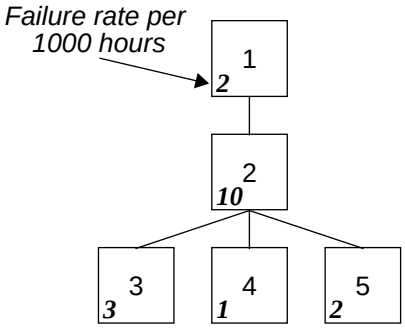
Evaluating the Suitability of RBD's for Specific Styles

Different architectural styles make different choices on the conceptual features. The choices made will affect the suitability of 'simple' reliability block diagrams for architectures of that style. Some styles make choices on the conceptual features which make it necessary to draw upon the extensions to RBD's described above if they are used to estimate the reliability of architectures in those styles. On the following page, table 3 illustrates this for a few styles with gray cells indicating choices which require an extended RBD to model their effects.

For example, a distributed processes architecture may have concurrent, dynamic, and distributed components. Together, these features can produce a complex reliability estimate. However, a main/subroutine architecture is ideally suited for 'simple' RBD analysis because of its lack of concurrency, distribution, dynamism, and an implicit global data distributor. This style was more or less assumed in early RBD analyses of software systems ([LLLI62]).

EXAMPLE

The following example illustrates the impact of architectural considerations on the RBD analysis of a simple system. Consider a system of five components arranged as shown in (figure 8). All five components are



$$\lambda_{\text{system}} = \sum_{i=1}^{5} f_i = 18/1000\text{hrs}$$

$$R_{\text{system}}(10) = e^{-\lambda_{\text{system}} 10} = 83.5\%$$

Figure 8

assumed to be vital to the success of the system, thus there is only one success path through the diagram and we can treat it as a system of five components arranged in a serial arrangement. The resulting reliability of this system over a 10 hour period is calculated by the product of all the individual component reliabilities. If we only use the original measured failure rates, then this very simplified analysis yields a point estimate of 83.5% reliability over a 10 hour period.

The analysis is considerably different if we introduce additional architectural information into the diagram (figure 9). Though the original topology of connectors is the same, the new diagram identifies different types of connectors, and a distributed, concurrent system. The system can now be identified as a simple abstraction of a popular

Table 3: Suitability of Reliability Block Diagrams to Certain Architectural Styles

	Pipe & Filter	Main/Subroutine	Distributed Processes	Event-Based
Dynamism of Computations	static	static	dynamic	static
Supported Data Transfers	explicit data connectors	shared data variables	explicit data connectors	implicit network, shared data variables
Triggering Capability	no	N/A	no	yes
Concurrency of Computations	multi-threaded	single-threaded	multi-threaded	single-threaded
Distribution	unconstrained	single node	multiple nodes	unconstrained
Layering	unconstrained	unconstrained	unconstrained	unconstrained
Encapsulation	no	no	no	yes

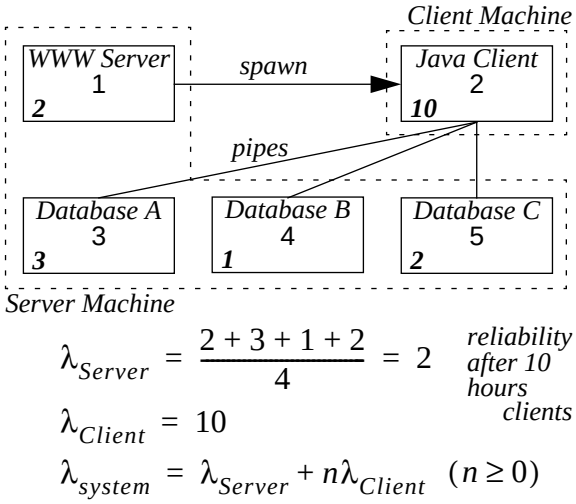


Figure 9

architecture found on the World Wide Web, that of a Java-based client/server architecture.

In this particular example, there is a server machine executing the Web server and three local repositories, and there is also a client machine executing the Java client. For simplicity, we assume that the server and repositories are in constant and concurrent usage, and that they share the CPU equally. The effective failure rate on the server then is given by the sum of its component's failure rates divided by the number of components (four). The effective failure rate on the client is simply that of the client itself (assuming no other applications are running on that machine). The overall effective failure rate of the system is a function of the number of clients which are accessing the server, and it increases as the number of clients (which are distributed) increases (figure 10). The reliability estimate now becomes a decreasing function, starting at 88.7% for one client, dropping to 80.3% for two clients, and so on.

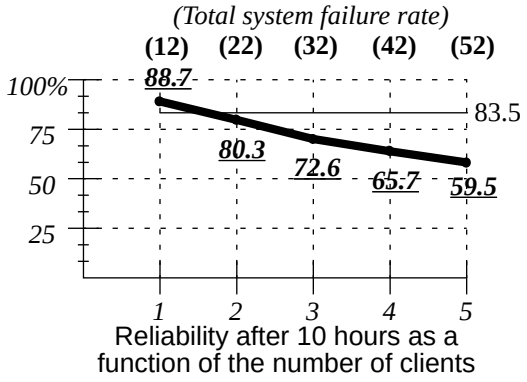


Figure 10

CONCLUSION

We have seen how reliability block diagrams can be extended to software architectures. Different conceptual features that span many architectural styles can have positive or negative effects on the effective failure rates of architectural components modeled in an RBD. The assumptions of RBD's may also be violated by the choices made on the conceptual features, as well as by the choice of different architectural connectors. As software systems increase in complexity via the use of distribution, concurrency, dynamism, and packaged middleware solutions, estimating the reliability of these systems using RBD's will also become more complex.

Our approach is a step towards quantitative evaluation of software architectures. It is more useful to use an RBD analysis of an architecture if we can also do a similar performance and/or cost analysis. The presence of these types of analyses will give architects an opportunity to explore design tradeoffs early in the software lifecycle.

Searching for analytical methods for additional nonfunctional attributes is just one of the future research steps to take. Another step is to integrate the extensions to

RBD's into an architectural analysis tool. Currently, work is proceeding on such a tool at USC. Finally, another promising step is to look for additional correlations between software architecture and software reliability (e.g. using the architecture to drive a reliability estimate based on software complexity metrics).

REFERENCES

[ABDA96] A. Abd-Allah. "Composing Heterogeneous Software Architectures", Doctoral dissertation, University of Southern California, August 1996

[BREA95] B. Bream, curator. "Reliability Block Diagrams and Reliability Modeling", Office of Safety and Mission Assurance, NASA Lewis Research Center, May 1995, http://www-osma.lerc.nasa.gov/rbd/

[GACB95] C. Gacek, A. Abd-Allah, B. Clark, B. Boehm. "On the Definition of Software Architecture", ICSE 17 Software Architecture Workshop, April 1995

[GASH93] D. Garlan and M. Shaw. "An Introduction to Software Architecture" Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Co., 1993

[MIO87] J. Musa, A. Iannino, K. Okumuto. "Software Reliability: Measurement, Prediction, Application", McGraw-Hill, 1987

[LLLI62] D. Lloyd, M. Lipow. "Reliability: management, methods, and mathematics", Prentice-Hall, 1962

[LYU96] M. Lyu, editor. "Handbook of Software Reliability Engineering", McGraw-Hill, 1996