1 My Thesis

It is possible to calculate the reliability of a software system from the reliabilities of the components of which the system is composed.

I would like to also claim that it is practical, but as most people would claim it wasn't possible, I will settle for the lesser claim.

1.1 Why components?

Building systems out of components is a natural part of engineering systems. With the rise of Object Oriented Programming, component technology is widely seen as the panacea to the software productivity problem. For many systems, reliability is of critical importance and for such systems a reliability composition calculus would be a significant advance.

1.2 The Nature of Reliability

Therefore there is at least as much *need* for reliability and reliability composition in software as there is in hardware. Unfortunately, most intuition about reliability in the hardware world is worse than useless in the software world, because the properties of the problem domains are so different. This is primarily because of several kinds of continuity and monotonicity that exist in the physical world that don't exist in the virtual world. Chapter 2 examines these issues, describes the major approaches to software reliability, and shows how the issues affect the approaches.

1.3 Program Input Domains

The reliability of a program (system or a component) is the probability that a given input is presented to the program as input, multiplied by one or zero, depending if the program produces

the same output as the specification says it should or not, summed over all possible inputs to the program. However, this value is not directly calculable as most programs have essentially infinite possible discrete inputs. In chapter 3 we look at an abstraction of this called path-domains, which groups the possible inputs into sets of inputs that are specified to be calculated in the same way, and sets of inputs that *are* calculated in the same way. We categorize these domains into four classes that have different ways to determine their reliabilities. By weighting these domains by the probability of inputs falling into each domain, we can calculate program reliability. This is still impractical for most *systems*, but is viable for many *components*.

1.4 Component Independence

There has been a fair amount of previous work in modeling software component reliability. The most glaring problem with that work is that they generally talk about independence without formally defining what is necessary. Independence is a fundamental requirement for calculating system reliability from component reliabilities, whether in hardware or software systems. Markov analysis is often used in such calculation. However, procedures as conventionally used do not qualify as nodes in a Markov system. In chapter 4 we outline the requirements for several classes of component independence and use the CPS transformation to convert conventional procedures into fragments appropriate to Markov analysis.

1.5 Probability Density Functions

To calculate reliability of a component, we need to have an accurate weighting of the probability of the various subdomains of the input space. The obvious way to do this is with a histogram of the input subdomains, but there is no good way to determine the histogram of the output of the component. Chapter 5 discusses probability density functions and the statistical theory of their transformation through operations such as addition and multiplication, and explains how this is applicable to programs.

1.6 Composing Systems from Components

Chapter 6 describes how a component economy would work - what component providers must state about their components, and how system integrators would use that information. We discuss a range of composition techniques with different performance and degrees of conservativism/accuracy. We also address the problem that some components will have an essentially infinite set of subdomains and provide two practical approaches to this problem.

1.7 Extension to Other Paradigms

In the rest of the thesis we have assumed that components are functional (i.e. that there is no state that is retained between invocations that is ever modified). Given that much of the impetus for components is the current popularity of Object Oriented Programming, chapter 7 outlines a sound treatment of object-oriented and imperative programs within this model.

1.8 Conclusions

Chapter 8 draws conclusions about the thesis and suggests future work.

2 The Nature of Reliability

There is longstanding practice for determining reliability for hardware systems. However, there are many differences between hardware and software systems, and in this chapter we show where the assumptions from system reliability theory cause difficulty when applied to software.

In this chapter and elsewhere when we refer to hardware reliability we are not talking about the reliability of the computers on which software runs, but rather of the reliability determinations used in traditional engineering domains such as Electrical or Mechanical Engineering. The background for this is derived from (Bentley 1999; Leemis 1995; Ramakumar 1993).

2.1 Definitions

DEFINITION 2.1
Reliability is the probability that a system will not suffer a Failure while executing with a particular operational profile.

DEFINITION 2.2
Failure means that the system is not operating according to the specification. Failures may be terminating or continuing.

DEFINITION 2.3
Terminating Failure is a failure that causes the system to halt. Although it is somewhat optimistic, we generally will use this definition of failure in this thesis.

DEFINITION 2.4
Detectable Failure is a failure that can be detected at run-time, but is not mandated by the semantics of the programming language to halt the system. For example, in `C`, referencing an invalid array index is not a terminating failure, whereas in safe languages such as `Ada`, `Scheme`,

and Java an exception or error is thrown upon such a reference. We assume a safe programming model and will therefore assume that detectable failures will be treated as terminating.

DEFINITION 2.5
Continuing Failure is a failure that does not prevent the program from continuing.

DEFINITION 2.6
Operational Profile is the description of the environment in which a system will be used.

DEFINITION 2.7
Multi-Version Programming (MVP) means having several different teams implement a program, and then having a voting system that chooses the majority answer or the unanimous answer. The idea is that the diversity of the implementations will not have coincident failures and that the system will thus have higher reliability.

2.2 Markov Models in Hardware Reliability

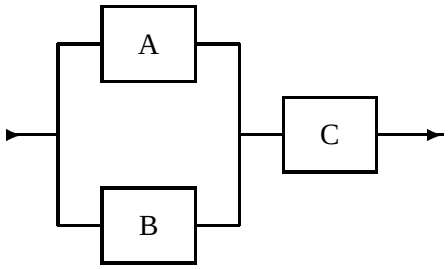
The Markov property states: given the current state of the system, the future evolution of the system is independent of its history.

The Markov property is assured if the transition probabilities are given by exponential distributions with constant failure or repair rates. In this case, we have a stationary, or time homogeneous, Markov process. This model is useful for describing electronic/mechanical systems with repairable components, which either function or fail. As an example, a Markov model could describe a computer system with components consisting of CPUs, RAM, network card and hard disk controllers and hard disks.

In the 3-component system in figure 2.1, for the system to be working, component C and one of A or B must be working. We will assume that component A is repairable.

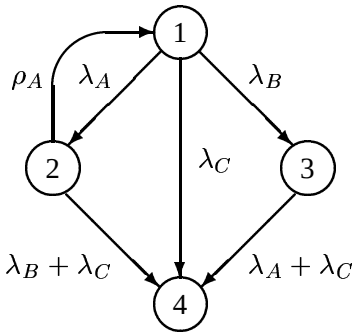
To produce a Markov model for a system, the system is analyzed to determine the possible states that the system can occupy. In figure 2.1, any of the 3 components can be working or failing, so there are a total of $2^3 = 8$ possible states: (A, B, C), $(A, B, \overline{C})$, $(A, \overline{B}, C)$, $(A, \overline{B}, \overline{C})$, $(\overline{A}, B, C)$, $(\overline{A}, B, \overline{C})$, $(\overline{A}, \overline{B}, C)$, $(\overline{A}, \overline{B}, \overline{C})$ (where A is working and $\overline{A}$ is failing). Since for the

Figure 2.1 A simple three component system



system to be working, component C and one of A or B must be working, there are 3 working states: (A, B, C), $(A, \overline{B}, C)$, and $(\overline{A}, B, C)$, with the other 5 states being failure states.

Figure 2.2 Markov model for the simple system in figure 2.1



The reliability (Markov) model for this is shown in figure 2.2, where all of the failure states are lumped together into state 4. In such models the labels on the arcs describe the rates at which the described system makes transitions from one state to another. The λ_i arcs are failure rates for component i, and the ρ_i arcs are repair rates for component i. Once the model has been derived, standard Markov analysis will allow us to calculate $P_i(t)$ — the steady-state probability of being in state i. Then the reliability will be:

$$\begin{aligned} R(t) &= P[\text{in a success state at time } t] \\ &= 1 - P[\text{in failure state at time } t]. \end{aligned}$$

For this model there are 4 states:

- State 1: A,B,C working;
- State 2: B,C working and A failing;
- State 3: A,C working and B failing;
- State 4: A,B failing or C failing.

Since states 1,2,3 are all success states and state 4 is a failure state,

$$\begin{aligned} R(t) &= P_1(t) + P_2(t) + P_3(t) \\ &= 1 - P_4(t). \end{aligned}$$

This derivation is dependent on the system meeting the Markov assumptions:

- state transitions independent: λ_A constant, regardless of how system arrived in state 1
- failure rates independent: P[A fail | B fail]= P[A fail]

If, for example, component A was less reliable after having been repaired than originally, then in order to maintain Markov properties the state diagram would probably be unrolled, as in figure 2.3. Here the λ'_A and ρ'_A represent the revised failure and repair rates for component A after it has been repaired once. If the reliability deteriorated materially in subsequent repairs, this model would continue to be unrolled until the component was unrepairable or the post-repair reliability stabilized.

Figure 2.3 Revision of figure 2.2, where A is not perfectly repairable

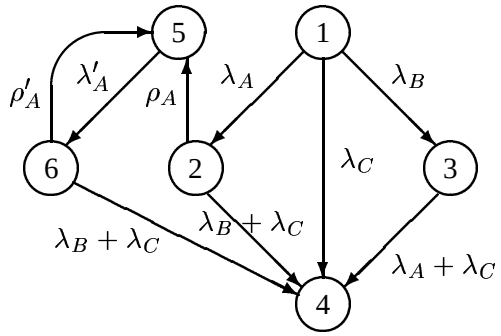
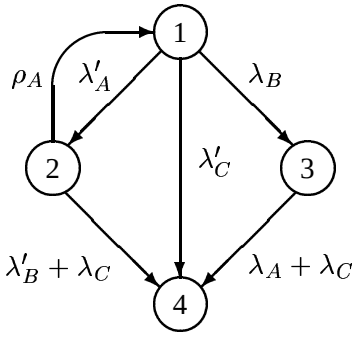


Figure 2.4 Revision of figure 2.2, where failure of B is not independent



Alternatively, the model might be inaccurate because the failure rates were not independent. If a failure of component A would immediately cause a failure of component B with some probability (because of increased load, for example), then it would be more accurate to show this as a direct transition to state 4 (reflected in figure 2.4 as λ'_C, with λ'_A capturing the failures of A that did not cause failure of B). In such a system, it is very likely that even if component B didn't fail immediately, it would have a lower reliability. This is reflected in figure 2.4 by $\lambda_{B'}$. In this example, for simplicity, we have assumed that component A is not affected by the failure of component B.

2.3 Software Reliability

Current software reliability estimation methods use code testing to obtain failure data which drives underlying mathematical models by which statistics such as system reliability are estimated. The meaningfulness of the statistics depend on the degree to which the software in question, and the software testing and data, match the *mathematical model* underlying the estimations (Lyu 1998). In §2.5 we explain how typical software tends to violate the underlying assumptions of the models used to estimate reliability. In §2.4, we expand upon previous observations that in many cases the models are not good matches for the software and testing processes. We explain how differing input continuity properties for hardware and software make it impossible to borrow certain aspects of software reliability modelling from that of hardware. We describe why

software testing does not provide data analogous to hardware testing in terms of reliability estimation and composition.

Because the models for reliability are not a good match for software and software testing, the meaningfulness of the estimates come into question, as previously indicated by many others, including (Butler and Finelli 1993; Hamlet 1994a; Lyu 1998; Parnas 1993). We expand upon these observations in §2.5.

2.3.1 Background: Hardware vs. Software Reliability

Reliability for hardware systems is a function of 4 factors:

- errors in design,
- errors in manufacture,
- physical defects, and
- chemical and physical wear.

Engineering is the art of the possible. Therefore, of necessity, every design is a compromise of several elements: costs, utility, and reliability (among others). Different designs can increase or decrease the contribution of the remaining factors (such as specifying different tolerances to reduce manufacturing errors, or specifying higher quality material to minimize defects and wear).

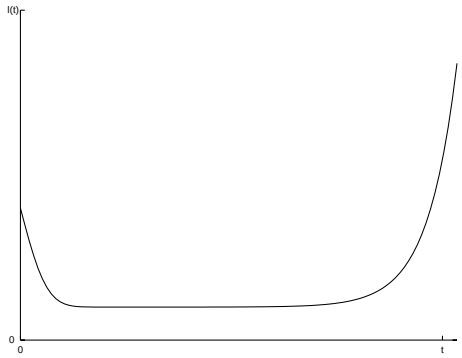
These factors lead to three typical components of failure:

- infant mortality,
- load variance failure, and
- aging failure.

The characteristic failure curves for these components combine to form what is referred to as a "bathtub curve". (see figure 2.5). This reflects a high early failure rate (infant mortality) primarily based on errors in design and manufacturing as well as physical defects.

Instances of the system that get beyond this initial infant mortality period tend to have very low failure rates for an extended period of time. During this period, failure is due to variations in the load imposed on the system and to other random environmental factors such as dust and shock. Finally the failure rate begins to rise as aging sets in. In this period components start to wear out, metals become embrittled, and atoms migrate in semiconductors, steel and cement. While the rate

Figure 2.5 "Bathtub" reliability curve



of onset of these factors can be affected by environmental factors such as heat, dust, and shock, and can often be postponed with preventive maintenance, the eventual failure is inevitable.

It should be noted that a bathtub curve reflects the reliability history (real or predicted) of a *single* design over its lifetime. Thus, if a design for a system is refined or modified over time, the resulting sequence of reliabilities form a family of bathtub curves. There are many possible designs for any given system. Each will have different engineering trade-offs in terms of costs and benefits. By using different designs, any of the failure components could be reduced, but this will usually be with increased cost (by increasing other failure components) or with reduced utility (heavier, or less capable). This can affect both the infant mortality and aging (e.g. tolerances too tight or too loose), but neither is ultimately preventable through design and none of the failure components can be completely eliminated.

Software is fundamentally different in that it does not wear out.[1] Nor, in most senses, are there manufacturing errors. This leaves us with only design (for software, this includes implementation) as a source of failure in software systems (Littlewood 1979), and one component of failure: load variance failure.

1 Some people talk about bit-rot, citing problems like the Y2K problem. However it is not that the program is deteriorating - simply that the conditions under which the program is run (the operational profile) have changed.

3 Subdomain Analysis

In §2.1 we gave a general-purpose definition of reliability that essentially said that it was the probability that a system will operate according to specification on a particular operational profile. Let us restate that in a form that is useful to evaluate software reliability:

DEFINITION 3.1
The Reliability of a program (system or a component) is the probability that a given input is presented to the program as input, multiplied by one or zero, depending if the program produces the same output as the specification says it should or not, summed over all possible inputs to the program. Formally,

$$\sum_{x \in X} \begin{cases} p(x), & \text{if } c(x) = s(x); \\ 0, & \text{otherwise;} \end{cases}$$

where X is the input domain, $p(x)$ is the probability that a random selection from X will be x, $c(x)$ is the output of the **program code** on input x, and $s(x)$ is the output **specified** for input x.

Unfortunately, this value is not directly calculable as most programs have essentially infinite possible discrete inputs. So let's see if we can find a useful way to group these inputs together into coherent subdomains.

3.1 Path Domains

First of all, we will look at the domains that are provided by the program code. For that, we need a definition of a path.

DEFINITION 3.2
A Path is a sequence of basic blocks leading through a program, from entry to successful termination or failure.

For example, consider the simple program in figure 3.1.

Figure 3.1 A simple program

```
1  input z;
2  x=1;
3  y=0;
4  while (x<5 && x<z) do
5    if (x<4) then
6      y=y+x;
7    else
8      w=z*z-25;
9      y=y/w;
10   fi
11   x=x+1;
12 od
13 output y;
```

DEFINITION 3.3
A Basic Block is a sequence of statements that has a single entry at the top and a single exit (conditional or unconditional jump) at the bottom.

Figure 3.2 shows the same program with the basic blocks labeled.

Conventionally, the paths for this code are: abg, abcdfbg, abcdfbcdfbg, abcdfbcdfbcdfbg, abcdfbcdfbcdfbcefbg. However, we are concerned with reliability, so we should explicitly capture the potential failures in the code. These would include things like divide-by-zero, arithmetic overflow, and subscript-out-of-range, and we will treat them like jumps, which leads to the definitions of a fault block.

DEFINITION 3.4
A Fault Block is the single statement $\langle fault \rangle$, or is a sequence of statements that has a single entry at the top and a single exit (conditional or unconditional jump) at the bottom. In the latter case, every instruction in the block will be executed, and no instructions will have anomalous behaviour such as overflow, underflow, or divide by zero.

Figure 3.2 A simple program with basic blocks

```
1    ⎡  input z;
2  a ⎢  x=1;
3    ⎣  y=0;
4  b ⎡  while (x<5 && x<z) do
5  c ⎢    if (x<4) then
6  d ⎢      y=y+x;
7    ⎢    else
8  e ⎢      w=z*z-25;
9    ⎣      y=y/w;
10   ⎢    fi
11 f ⎣    x=x+1;
12   ⎢  od
13 g ⎣  output y;
```

Adding conditional checks for these we get the revised blocks in figure 3.3. To simplify the exposition, we have removed some checks for faults where they are impossible due to previous checks or faults.

The paths for this code are: abg, abcdfbg, abcdfbcdfbg, abcdfbcdfbcdfbg, abcdfbcdfbcdfbceh, abcdfbcdfbcdfbceij, and abcdfbcdfbcdfbceikfbg. Note that the 2 paths that don't get to "g" are already known to be failure paths.

Each of these paths will be followed unconditionally for a particular subset (or path-domain) of the possible inputs (or domain). And the union of all of the path-domains will be the program domain.

Further, each of the paths will calculate a result. It may be a single value, or it may be a function of the inputs, but it will be exactly the same sequence of machine instructions that will be executed for every possible value in the path-domain. Furthermore, by the method of construction of the paths, none of those instructions can cause a fault, an overflow, or any other source of discontinuity (except on paths that terminate in $\langle fault \rangle$ and are identified as such).

Figure 3.3 A simple program with basic blocks and fault checking

```
1    ⎡  input z;
2  a ⎢  x=1;
3    ⎣  y=0;
4  b ⎡  while x<5 && x<z do
5  c ⎢    if x<4 then
6  d ⎢      y=y+x;
7    ⎢    else
8  e ⎢      if z>√MAX then
9  h ⎢        ⟨fault⟩
10   ⎢      else
11 i ⎢        w=z*z-25;
12   ⎣        if y/w is an error then
13 j ⎢          ⟨fault⟩
14   ⎢        else
15 k ⎢          y=y/w;
16   ⎢        fi
17   ⎢      fi
18   ⎢    fi
19 f ⎢    x=x+1;
20   ⎢  od
21 g ⎣  output y;
```

Figure 3.4 The paths for the simple program

Number	Path	Domain	Result
1	abg	$z \leq 1$	0
2	abcdfbg	$1 < z \leq 2$	1
3	abcdfbcdfbg	$2 < z \leq 3$	3
4	abcdfbcdfbcdfbg	$3 < z \leq 4$	6
5	abcdfbcdfbcdfbceh	$z > \sqrt{MAX}$	$\langle fault \rangle$
6	abcdfbcdfbcdfbceij	$z = 5$	$\langle fault \rangle$
7	abcdfbcdfbcdfbceikfbg	$4 < z < 5 \vee 5 < z \leq \sqrt{MAX}$	$\frac{6}{z^2-25}$

3.2 Specification Domains

Referring back to our definition of reliability, it is clear that it is meaningless without a specification for the program. In the absence of a specification we could use the analysis in the previous section to identify input subdomains that would signal faults[2], but that certainly doesn't imply that all other input values are correct.

Specifications can come in many forms. They may be executable or not, but they must be formal to be of any benefit to us. If they are executable, then an analysis similar to that in the previous section can be applied to them and the appropriate domains and results may be extracted. If they are in some other form, corresponding analysis must be applied to acquire the domains extant in the specification.

Note that we are *not* attempting to assist in debugging, or characterize errors, or any of the other things for which people have used path and domain testing, but simply trying to determine the reliability of classes of code.(**?**; **?**)

3.3 Program Domains and Failure Rates

Once we have the set C of code domains, and the set S of specification domains, we can determine the set D of program domains, each of which is covered by some code domain and some specification domain.

$$D = \left(\bigcup_i \bigcup_j C_i \cap S_j \right) - \emptyset$$

By construction, each of these program domains is a subset of some specification domain and therefore has a single specified result. Similarly, each of these program domains is also a subset of some code domain and therefore has a single computed result.

---

2 An analysis that simply identified potentially faulting paths may be useful in itself, but we are here interested in the larger problem.

If the code domain represents a $\langle fault \rangle$ path, then the domain has a failure rate of 1. Otherwise there is some chance that the code produces the specified result and therefore it will have some lesser failure rate. There are several ways we could determine whether the code produces the specified result.

3.3.1 Exact Equivalence

The simplest case is equivalence. Either the result is a constant, or it is a calculation that can be proven equivalent in the specification language and the implementation language. If so, the failure rate is 0.

3.3.2 Enumerable

If the domain has a very small number of members, then it may be practical to run the code on each value in the domain and verify that the result is the same as the one specified. If they all pass, the failure rate is 0. If not, the likely approach would be to fix the exposed bug and rerun the process ≪*Denise: when I got to that point in my thesis talk at BNR, the developers rolled around on the floor laughing...* ≫ to determine the code domains. Alternatively, the failing points can be put in a domain of their own with a failure rate of 1, and the correct points can be assigned a failure rate of 0.

ToDo⇒

3.3.3 Linear

If the specification is of a hyperplane calculated from the input, and the code result is also a hyperplane, the results can be shown to be equal by testing at all the extrema points of the hyperplane. This can be done in 2^n tests for a n-dimensional hyperplane. If they all pass, the failure rate is 0. If not, the likely approach would be to fix the exposed bug and rerun the process to determine the code domains. Alternatively, the domain can be assigned a failure rate of 1. ≪*conservative* ≫.

ToDo⇒

3.3.4 Sampling

If none of the other approaches works, the fallback is to perform a uniform random sampling of the domain. Unlike the usual case in software sampling, here sampling is a sound approach. Because the domain has been constructed so that no discontinuities can exist, sampling will provide meaningful results. Standard statistical techniques will allow you to derive an expected failure rate from the number of tests that you perform without uncovering any errors. If errors are found, the likely approach would be to fix the exposed bug and rerun the process to determine the code domains. Alternatively you could use the assessed reliability if it is only small numbers of errors.

3.4 Dealing with an infinity of path-domains

As you may have guessed by now, there are potentially a lot of paths in a typical program. Even a simple function that has 3 separate `if` statements will have 8 paths though it. If you wrap that in a loop that repeats 100 times, you are up to 8^{100} possible paths and if you have 2 such functions you square that number ... but you get the idea.

The useful insight here is that there are huge numbers of paths that we don't actually care about. For example, many of those paths will end in $\langle fault \rangle$, and in a useful program those won't actually get run. And then there are all the special situations that almost never get executed. These add huge numbers of paths, but they have almost no impact on the reliability of the system, because they essentially never run.

Unfortunately, even if we found a way to weed out all of those, this kind of path analysis is still impractical for most *systems*. The good news is that it could be viable for many *components*.

The other bad news is that even for some very simple components the number of interesting domains is immense. See chapter 8 for some prospects for future work. ≪*put in the reference to the section in system-level on feeding OPs into the path generation queue.* ≫ ⇐**ToDo**

We haven't talked about how to actually generate paths from the program. Assume that the program has been broken up into basic blocks, or in our case, fault blocks, and that BB_0 is the first one in the program. Then algorithm 1 shows the standard algorithm for generating the paths. This algorithm generates the paths in shortest-first order. Nodes are assumed to have 1 or 2 jump targets, or to be exit or fault nodes.

Algorithm 1 Generate Paths from Block List

```
1  procedure GeneratePath(startNode)
2    workList := new PriorityQueue()
3    workList.add(0,(startNode,new Path())
4    while workList.notEmpty do
5      node,path := workList.next
6      path.add(node)
7      if node.exit then
8        path.emit('exit')
9      else if node.fault then
10       path.emit('fault')
11     else
12       workList.add(node.target(1).minlength,(node.target(1),path))
13       if node.jumps=2 then
14         workList.add(node.target(2).minlength,(node.target(2),path))
15       fi
16     fi
17   od
```

For our purposes we want the most important paths first. That is, the ones with the highest likelihood of being executed. Algorithm 2 shows the algorithm for generating the paths in most frequent order. The same assumptions as before, except that `input` is the initial (complete) input domain, and the priority queue orders on a pair, maximizing the first of the pair, and minimizing the second of the pair.

3.5 Previous Work

Analyzing the domains based on the paths found in both the code and the specification was called Partition Analysis in (Richardson 1981; Richardson and Clarke 1985), but was used for determining the correctness of a program through a combination of formal verification and testing.

Weyuker and Ostrand (1980) were exploring similar ground, trying to find domains where either all tests passed or all tests failed, although in the extreme their domains came down to each being a single point.

Algorithm 2 Generate Paths in Frequency Order

```
1  procedure GeneratePathInFreqOrder(startNode,input)
2    workList := new PriorityQueue()
3    workList.add((0,0),(startNode,new Path(),input,null))
4    while workList.notEmpty do
5      node,path,domain,result := workList.next
6      path.add(node)
7      if node.exit then
8        path.emit('exit',domain,result)
9      else if node.fault then
10       path.emit('fault',domain)
11     else
12       newResult := node.calculate(result,domain)
13       newDomain := node.filter(1,domain)
14       workList.add((newDomain.size,node.target(1).minlength),
15          (node.target(1),path,newDomain,newResult))
16       if node.jumps=2 then
17         newDomain := node.filter(2,domain)
18         workList.add((newDomain.size,node.target(2).minlength),
19            (node.target(2),path,newDomain,newResult))
20       fi
21     fi
22   od
```

The contribution of this chapter is the use of fault blocks as the constituent element of paths and the consequent homogeneity of the associated domains, the classification of domains, and the technique to generate an ongoing sequence of the most useful domains.

Some of the ideas in this chapter were published as (Mason and Woit 2000).

3.6 What about that Operational Profile?

At this point, we have an accurate characterization of the reliability of each program domain. By weighting these domains by the probability of inputs falling into each domain, we can calculate

the program reliability. So if we had an accurate operational profile for a program as it was actu-
Ref⇒ ally used, we could predict its reliability with fair success. As mentioned in ≪*intractable* ≫, this is intractable for complete systems because normally they will have too many domains to be useable.

However, for components, finding accurate operational profiles is quite possible. We will discuss how to do that in chapter 6, once we've laid some ground work.

5 Probability Density Functions in Program Analysis

To accurately, or even conservatively, determine the reliability of programs, we will need to know the values that the variables and expressions in the program can take on. This is partly because the values can determine the reliability directly (for example if a variable could be zero and is used in a division, there is a chance of failure), and partly because the values of variables usually determine the frequency of execution of various parts of the program.

The simplest characterization of the values that a variable could have is the type of a variable. This is extended in languages like Ada(Ada Joint Program Office 1983) to include subranges. This can, in some circumstances rule out certain failures, but for others, says nothing about their likelihood.

To increase the accuracy of characterization of the values we could use a symbolic or abstract interpretation of the program to determine the set or range of values that a variable can contain. This provides a finer grain of checking, but is essentially the same as type-based determination.

To provide the most accurate characterization, we will need to have a profile of the set of values that each variable can take on. For some program variables (synthetic variables such as loop controls) this can be determined via an abstract interpretation of the program. For others (input variables), statistical information can be provided about the environment in which the program will run.

In this chapter, we examine the concept of Probability Density Functions and how they relate to variables, expressions, and statements in a programming language.

An early version of this was presented as (Mason 2001).

5.1 Probability Density Functions versus Histograms

Virtually all previous publication on work with operational profiles (Musa 1998; Lyu 1996a; Woit and Mason 1998b; Hamlet, Mason, and Woit 2001; Mason and Woit 2000) has used histograms to represent the input density for the operational profile. There are two fundamental problems with this. The first is that the boundaries reflected in the histogram may have little or no relationship to the way the program works, so testing based on it may seriously distort the actual reliability of the program. The second is a more serious issue if histogram values are going to be flowed from one component to the next, as is done in (Hamlet, Mason, and Woit 2001).[6]

As an example of the problem, assume we have two input parameters. To make this as simple as possible, we will assume that they have uniform integer values, $X = \{1, 2\}$ and $Y = \{4, 5, 6, 7\}$, both shown in figure 5.1 (X is the darker colour).

Figure 5.1 X and Y with discrete distributions

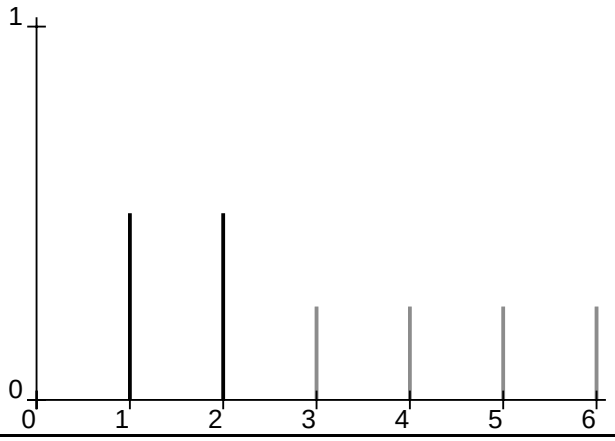


Figure 5.2 shows the sum of the two variables, and figure 5.3 shows the product. As can be clearly seen, the nice uniform histogram shape has changed significantly, even after a single operation. With any more operations, it becomes completely unrecognizable.

6 That paper works around the problem with a correct, but computationally intractable approach.

Figure 5.2 X plus Y with discrete distributions

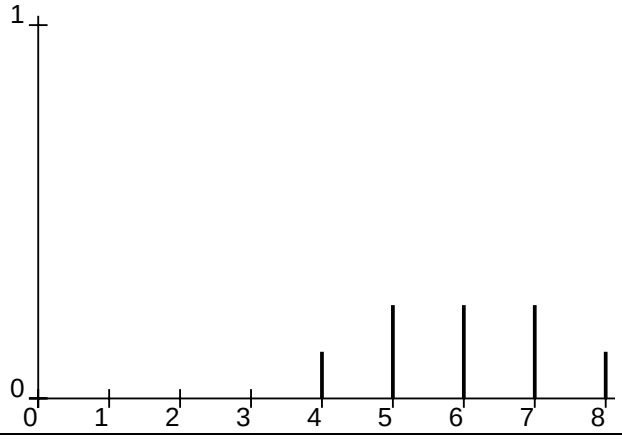
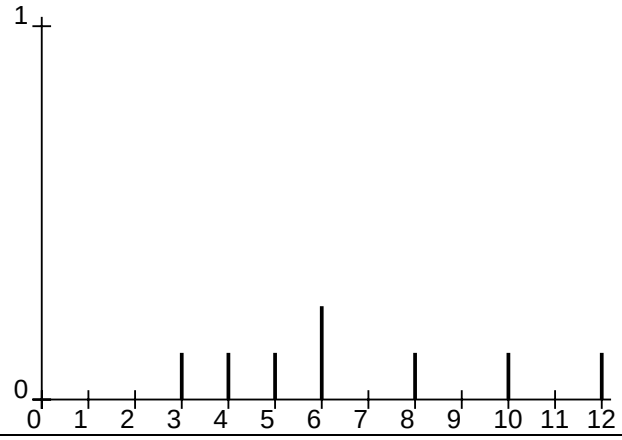


Figure 5.3 X times Y with discrete distributions



5.2 Introduction

A Probability Density Function is a non-negative function with an integral of 1. That is, f is a Probability Density Function iff:

$$f(x) \geq 0, \forall x \in dom(f), \text{ and}$$

$$\int_{dom(f)} f(x)\,dx = 1.$$

The value of the function $f(x)$ is the probability that if a random value in the $dom(f)$ is chosen, that the value will be x. For continuous functions it is usually more relevant to consider the probability that a value will lie within some range, $a < x \leq b$, which is:

$$\int_a^b f(x)\,dx.$$

Probability Density Functions can be discrete or continuous. An example of a discrete Probability Density Function is the probability that a flip of a fair coin will be heads or tails:

$$flip(\texttt{heads}) = 0.5$$
$$flip(\texttt{tails}) = 0.5.$$

An example of a continuous Probability Density Function is the probability that a person will be a particular height, or have a particular IQ. In standard statistical usage, discrete Probability Density Functions are called Probability Functions (PFs). The important difference is that in a PF, a single value can be significant:

$$\sum_a^a f(x)\,dx$$

is equal to $f(a)$ and may be non-zero, whereas for a Probability Density Function:

$$\int_a^a f(x)\,dx$$

is equal to 0, regardless of the value of $f(a)$. Note that f may not be a fully continuous function, but it will be at least left-continuous or right-continuous at every point in its domain. To simplify the presentation, we will use the term Probability Density Function for both.

To further simplify the presentation, we will use integrals ($\int$) throughout - even over discrete sets where sum ($\sum$) would be literally more correct. As long as we are talking about the integral/sum over the complete domain of the Probability Density Function (which we almost always do) they are essentially equivalent. Only when actual values of the functions are required will we replace the integrals with sums.

The other difference between discrete and continuous Probability Density Functions is that continuous Probability Density Functions will require a Jacobian in the convolutions. We will use the Jacobians, but understand that it will be the constant 1 when the Probability Density Function is discrete.

In common usage in statistics, the domain of Probability Density Functions is the set of real numbers. For our purposes, it may be any set, and rather than using ranges over the real numbers we will often use subsets of the domain of the Probability Density Function.

When we refer to the domain of Probability Density Functions we will mean values or subranges where the probability is non-zero, i.e. $x \in dom(P)$ iff $P(x) \neq 0$.

5.3 Independence of Probability Density Functions

Two (or more) random variables may be dependent or independent. Independence means that the Probability Density Function for one variable is not affected in any way by the Probability Density Function for another variable. Formally:

$$P_{\mathtt{x},\mathtt{y}}(x,y) = P_{\mathtt{x}}(x)P_{\mathtt{y}}(y)$$

or

$$P_{\mathtt{x}_0,\mathtt{x}_1,\mathtt{x}_2}(x,y,z) = P_{\mathtt{x}_0}(x)P_{\mathtt{x}_1}(y)P_{\mathtt{x}_2}(z)$$

If the variables are dependent, then the joint density function $P_{\mathtt{x}_0,\mathtt{x}_1,\mathtt{x}_2}$ will be a more complex function, for example:

$$P_{\mathtt{x},\mathtt{x}}(x,y) = \left\{ \begin{array}{ll} P_{\mathtt{x}}(x), & \text{when } x = y \\ 0, & \text{otherwise} \end{array} \right\}$$

As an example of this, consider the 2 simple uniform ranges $X = 1 \ldots 2$ and $Y = 3 \ldots 5$. If X and Y are independent, then $P_{\mathtt{X}+\mathtt{Y}}$ is depicted in figure 5.4. Alternatively if, for example, $Y = X \times 2 + 1$, and hence Y is dependent on X, then $P_{\mathtt{X}+\mathtt{Y}}$ is as shown in figure 5.5.

In the rest of this chapter we will use joint Probability Density Functions without talking directly about the independence of the variables.

Figure 5.4 Sum of Independent values

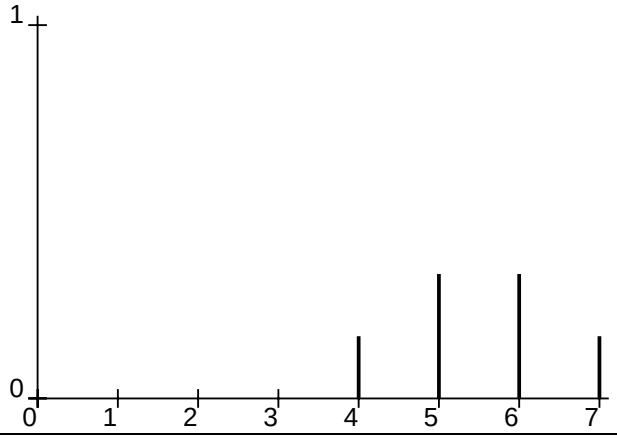
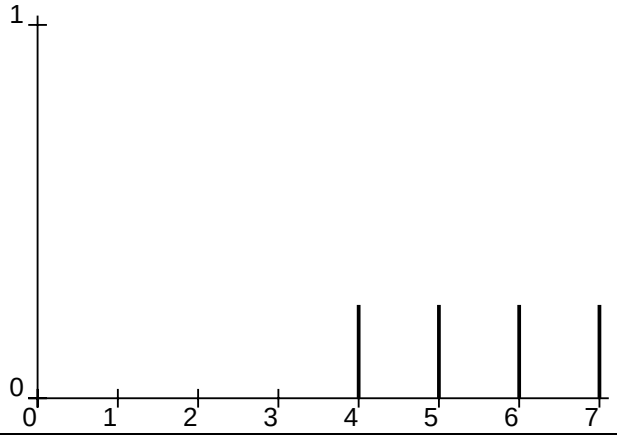


Figure 5.5 Sum of Dependent values



5.4 Probability Density Functions of Program Variables and Operations

In order to accurately model program execution (for software reliability or code optimization) we need to have a good representation for the values that various program variables can hold. We will use a Probability Density Function for each variable in the program, or program fragment, under examination. For a variable abc the Probability Density Function will be $P_{\texttt{abc}}$, where $P_{\texttt{abc}}(x)$ is

the probability that the value of abc is x, and the domain of $P_{\texttt{abc}}$ will be the set of all values that abc could take on.

For all integers and literals, the Probability Density Functions will be discrete distributions. Floating-point, or "real" values will also be represented as discrete distributions if their values are enumerable from examination of the program, or if they are input variables with specified discrete distributions. If they are the result of functions such as sin, log, or exp, or are continuous input values, then they will be treated as continuous distributions. Note that this assumption, while not technically correct since computer floating point numbers are actually finite-precision rational numbers, is a useful fiction that will facilitate deriving closed-form solutions to the systems of equations. By paying careful attention to confidence intervals, accurate probabilities should be attainable.

If variables are represented as Probability Density Functions, then the result of program expressions must also be Probability Density Functions. In the balance of this section we look at expressions of random variables, expressed as Probability Density Functions. Some of the these are derived from (DeGroot 1989); the remainder are developed in the same style.

5.4.1 Literals

The Probability Density Function for any program literal (or compile-time constant) value **c** is the function:

$$P_{\mathbf{c}}(\mathbf{c}) = 1.$$

The Probability Density Function for each literal (whether real or integer) is a discrete Probability Density Function, with a single element in the domain.

5.4.2 Monadic Functions

The Probability Density Function for negation is:

$$P_{-\mathtt{x}}(z) = P_{\mathtt{x}}(-z).$$

This can be extended to any invertible monadic function f as:

$$P_{\mathtt{f(x)}}(z) = P_{\mathtt{x}}(f^{-1}(z))J(z), \quad \text{where } J(z) = \left|\frac{d}{dz}f^{-1}(z)\right|,$$

wherever f^{-1} is defined.

5.4.3 Dyadic Functions

The Probability Density Function for addition is:

$$P_{\mathtt{x+y}}(z) = \int P_{\mathtt{x,y}}(z-y, y)\, dy,$$

or

$$P_{\mathtt{x+y}}(z) = \int P_{\mathtt{x,y}}(x, z-x)\, dx.$$

There will be multiple versions for most operations, but we will only mention them when they are important.

The form for addition can be extended to any dyadic left-invertible function (such that: $f^{-1}(f(x,y),y) = x$) as:

$$P_{\mathtt{f(x,y)}}(z) = \int P_{\mathtt{x,y}}(f^{-1}(z,y),y) \left|\frac{d}{dz}f^{-1}(z,y)\right| dy.$$

This works fine for subtraction and division, but there is a potential problem with multiplication since $x \times y$ is not left-invertible if $y = 0$. However, this is just as applicable to any dyadic right-invertible function (such that: $f^{-1}(x, f(x,y)) = y$), as:

$$P_{\mathtt{f(x,y)}}(z) = \int P_{\mathtt{x,y}}(x, f^{-1}(x,z)) \left|\frac{d}{dz}f^{-1}(z,y)\right| dy.$$

so as long as a function is either left or right invertible, there is a solution. For multiplication this means that if, at least one of $0 \notin dom(P_{\mathtt{x}})$ or $0 \notin dom(P_{\mathtt{y}})$, we can produce a Probability Density Function.

6 Component Reliability Composition

Now we have all the pieces for component reliability composition, and are ready to put it all together into a description of a component economy.

6.1 Role of the Component Supplier

The component supplier designs and programs the component. They then choose a likely input distribution[7] and start characterizing the component, as described in chapter 3. This is a time-consuming process in the common case when there are a lot of domains and where the oracle (which verifies that the implementation matches the specification) is slow. Fortunately the process of characterizing the component can be run in parallel on a farm of machines.

The goal of the characterization is to determine all of the sub-domains of the component, and the quality of each sub-domain.

Likely before this is complete, a request will come from a system designer with a specific input distribution and a request for the quality of the component for that distribution. If the input sub-domains already characterized cover the non-zero parts of the specified distribution, then a quality number can be calculated and returned to the system designer. Otherwise, the provided input distribution can be substituted into the characterization process which will cause the subsequent characterized sub-domains to be the ones most relevant to the customer needs.[8]

---

7 The exact distribution doesn't matter at this stage - even a uniform distribution will work fine if nothing better presents itself - it may just take longer than necessary to characterize the most important paths.

8 Distributions from multiple customers can be combined with a weighting function based on the value of each customer so as to keep them all as happy as possible.

A conservative approximation to the quality number can be calculated by assuming failure in all of the uncharacterized sub-domains. If all of the uncharacterized sub-domains have extremely low probabilities, based on the specified input distribution, this can provide a useful interim value to the system designer, with the sure knowledge that the system will only become more reliable as the residual sub-domains are characterized.

6.2 Role of the System Designer

The system designer has 5 main tasks.

1. Design the structure of the system. This involves all the usual steps of decomposing the problem, deciding if any off-the-shelf components can be used, determining the basic control structure.
2. Specify the components. The specifications must be formal specifications so that they can be used in the process described in chapter 3. It also must be, in some sense, executable so it can be used below.
3. Determine the input distribution. The more accurately this can be determined, the more accurately the actual system reliability will be reported.
4. Implement the skeletal system to determine the distributions that will be presented to the components. This is very similar to the component characterization process, in that the program will be executed with Probability Density Functions representing the input, and the parameters to all components will be collected. Perfect implementations of the components must be available so that the results of each component can be used in the calculations provided to subsequent components.
5. Extract from the system design the model that will use the component quality numbers to calculate overall system quality. This will be an equation in m unknowns (where m is the number of component invocations) or an equation in n unknowns (where n is the number of components used). The choice between these will be explored in §6.2.1. The constants in the equation will reflect the quality of the system structure code.

The term “system designer” need not be construed to refer to a single person - it may be a whole company that is the general contractor for a project. Note that everything after the first two tasks can potentially be performed in parallel with the component suppliers.

6.2.1 Speed versus Accuracy

To calculate the system quality completely accurately would require treating every call to a component separately, which would entail many requests for quality numbers from the component supplier and then incorporating those numbers into the equations. If, on the other hand, we could provide one, properly weighted, quality number for all uses of each component, the calculations would be greatly simplified.

To determine the feasibility of lumping them all together, a Monte Carlo simulation was created comparing the two approaches. The difference between the two depends on three things: the quality of the glue code, the quality of the component, and the number of times the component is called in a path.

If a component is called once per path, then the error will be bounded by the product of the uncertainty of the quality of the glue code and the quality of the component. For example, if the glue has a quality of 0.9 and the component has a quality of 0.99 then the error will be less than 0.001.

If the component is called n times in the longest path then the quality of the component must be n^2x better, where x is the uncertainty in the quality of the glue code. For example, with the glue code the same as in the previous example, but the component called 100 times, the component quality has to improve to 0.9999999 to maintain an error bound of 0.001. Of course, if the component is called that many times it would have to have very high quality anyway, but the level required to bound the error is a much more stringent requirement.

6.3 Blurring the Distinction: Components that Use Components

6.3.1 Long-Running Components

6.4 Previous Work

Very early versions of the ideas that make up this chapter were published as (Hamlet, Mason, and Woit 1999; Hamlet, Mason, and Woit 2001).