Probabilistic Program Analysis for Software Component Reliability

by

David Victor Mason

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of

Doctor of Philosophy

in

Computer Science

Waterloo, Ontario, Canada, 2002

Abstract

Components are widely seen by software engineers as an important technology to address the “software crisis”. An important aspect of components in other areas of engineering is that system reliability can be estimated from the reliability of the components. We show how commonly proposed methods of reliability estimation and composition for software are inadequate because of differences between the models and the actual software systems, and we show where the assumptions from system reliability theory cause difficulty when applied to software.

This thesis provides an approach to reliability that makes it possible, if not currently plausible, to compose component reliabilities so as to accurately and safely determine system reliability.

Firstly, we extend previous work on input sub-domains, or partitions, such that our sub-domains can be sampled in a statistically sound way. We provide an algorithm to generate the most important partitions first, which is particularly important when there are an infinite number of input sub-domains. We combine analysis and testing to provide useful reliabilities for the various input sub-domains of a system, or component. This provides a methodology for calculating true reliability for a software system for any accurate statistical distribution of input values.

Secondly, we present a calculus for probability density functions that permits accurately modeling the input distribution seen by each component in the system - a critically important issue in dealing with reliability of software components.

Finally, we provide the system structuring calculus that allows a system designer to take components from component suppliers that have been built according to our rules and to determine the resulting system reliability. This can be done without access to the actual components.

This work raises many issues, particularly about scalability of the proposed techniques and about the ability of the system designer to know the input profile to the level and kind of accuracy required. There are also large classes of components where the techniques are currently intractable, but we see this work as an important first step.

Acknowledgments

Thanks to John Knight and Rudi Seviora for their kind words, and for being helpful and insightful external examiners.

Thanks to Dick Hamlet for the numerous insightful discussions, and for taking his Fulbright in Ireland so that Denise and I were *forced* to go there in 1998 to work on this research.

Thanks to Dan Berry for his enthusiastic demand to be on my committee and his insights into the academic life.

Thanks to Gord Cormack for support and friendship through the long, “Yes, I’m going to finish this sometime”, years.

A huge ‘Thank you’, to Jo Atlee for helping me turn a collection of papers and ideas into a thesis, and for always being cheerful when I pointed out that something might well be impossible.

Special thanks to my long-time collaborator, Denise Woit, for dragging me into the world of software reliability in the first place, challenging my blithe assertions, and helping me explore smoked-salmon around the world.

Thanks to Hugh Chipman and Chris Grandison for helping me understand statistics and calculus I never thought I’d have to know.

Thanks to all my family and friends (especially the faculty, staff and grad students at the University of Waterloo) for their support over the years, and especially to my parents for waiting all these years, with only *occasional* hints, for there to be a Doctor in the family.

And last, but not least, thank you Beth, Eliza, and Karin for being there for me and providing the love and support to help me get through it all!

Contents

$\boxed{1}$ 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.

$\boxed{1.1}$ Is he *mad?*

You, the reader, will probably greet my thesis with either a puzzled, "Of course it is!", or an incredulous, "Of course it isn't!" The truth, as usual, lies between these extremes. It is possible but, without considerable development, it appears to be very limited in application.

Building systems out of components is a natural part of engineering systems. Software Engineering emphasizes the related concepts of modularity and encapsulation. Increasingly, component based software engineering is seen as a solution to the software productivity problem. However, software component reliability has only been addressed by generic attempts to generate high-quality components and by arguments about components being used in a variety of contexts and hence being well tested.

In particular, there has been little progress in analysing components so that a system's reliability could be computed from the reliability of the components of which it is composed. This is for very sound reasons: the reliability of most components is highly dependent on the inputs to the component, and there is a great deal more feedback in software systems than in traditional engineered hardware systems, such as vehicles or power-generation plants. Thus, to determine system reliability, the designer must fall back on system reliability measures. System reliability measures have their own problems, in addition to being calculated *after* the system is completed. For many systems, reliability is of critical importance, and for such systems a reliability composition calculus would be a significant advance.

The approach that we lay out in this dissertation is a first attempt to describe software reliability carefully enough to be applicable to component composition. Unfortunately in its current state of development, it has a limited application domain and potentially exponential complexity, so is only practical for the most trivial of problems. It *may* not be too long before it is applicable for small to moderate-size embedded systems where reliability is paramount, but will likely be a very long time before it is practical for something the size of a word processor or an operating system.

The structure of the rest of this chapter parallels the dissertation; each section corresponds to a chapter.

1.2 The Nature of Reliability

Software reliability has been analysed using techniques and assumptions appropriate to the analysis of hardware (mechanical and structural) systems. 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 system reliability, and shows how the issues affect the approaches. The last section briefly describes previous work, including ours, on component reliability. The rest of the chapter may be skipped if the reader has a very good understanding of the issues involved in software reliability versus hardware reliability.

1.3 Domain Analysis and Software Reliability

Reliability is the probability that a system will operate according to specification, over a particular profile of inputs, for a specific period of time. For software, we can encode time of arrival as another input parameter, but there is no other temporal aspect (such as wear or chemical deterioration) to consider. In general reliability texts (Leemis 1995), removing time from the definition gives a definition for 'Quality'. Unfortunately both Software Reliability and Software Quality are used somewhat ambiguously. In the interests of clarity and accuracy, we'll use the term 'Probabilistic Correctness'.

The probabilistic correctness of a program (system or component) is the probability that each input produces the specified output, weighted over all possible inputs to the program. This value usually is not directly calculable because most programs have essentially infinite possible discrete inputs. In Chapter 3, we look at an abstraction of probabilistic correctness called path-domains, which groups the possible inputs into sets of inputs that are *specified* in the same way, and sets of inputs that are *calculated* in the same way. We categorize these domains into six classes that have different ways to determine their probabilistic correctness. By weighting these domains by the probability of inputs falling into each domain, we can calculate program probabilistic correctness. This is still impractical for most *systems*, but is viable for many *components* and is a necessary part of a complete reliability composition theory.

1.4 Probability Density Functions

To accurately calculate probabilistic correctness of a component as described in Chapter 3, we need to have an accurate weighting of the probability of the various sub-domains of the input space. The obvious way to do this is with a histogram of the input subdomains, but most common operations (like addition and multiplication) applied to histogram distributions do not produce histogram distributions! Because the output of one component is the input to another component, we need a more general representation of distributions, which probability density functions provide.

Chapter 4 discusses probability density functions and the statistical theory of their transformation where it is applicable to program analysis.

1.5 Composing Systems from Components

Chapter 5 uses the theory developed in chapters 3-4 to calculate an accurate system reliability based on probabilistic correctness of components. We describe how to determine the static probabilistic correctness properties of a component independent of a particular operational profile. We show how to calculate the probabilistic correctness of a component for a particular operational profile based on the static properties. We describe how a component economy would work — what component providers must state about their components, and how system integrators would use that information. We also address the problem that some components will have an essentially infinite set of sub-domains.

1.6 Limitations and Assumptions

The goal of this thesis is to produce component probabilistic correctness estimates accurate enough to allow component composition and calculation of system reliability. Unfortunately that places certain limitations on the problem domains and programs that can be analysed. Some of these limitations appear to be fundamental, others may yield to further research and analysis, and some may be relaxed if accuracy requirements are less stringent.

1. Accurate operational profiles are essential. We discuss the plausibility of sufficient accuracy being achievable in §5.2.1.
2. Analysis is limited to range-based parameters (ordered sets, such as floating point, integer, character).
3. Analysis is limited to scalar parameters, i.e. no arrays, strings, or lists.
4. The analysis is value-based, and hence does not address graphs that are not expressible as trees.
5. Because of limitations in standard mathematical analysis techniques, some limits may apply to the complexity of comparisons in the program if they depend on several input parameters, as detailed in §4.4.1.
6. The worst-case cost of some of the calculations is high.
7. Although the analysis is not restricted to functional programming, the analysis is simpler for programs in that style. We will postpone consideration of other styles to §6.1.4.
8. There are significant limitations in estimating reliability for long-running systems.
9. Analysis becomes intractable for languages such as `C` that have un-tamed pointers.

As well as explaining some of these limitations in more detail, in Chapter 6 we draw conclusions about the thesis and we suggest future work.

2 The Nature of Reliability

There is long-standing 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.

This chapter examines the differences between hardware and software and also describes related work in software reliability. §2.5 re-characterizes previous work by the author. The reader with a good understanding of the differences between hardware and software reliability, and familiarity with the literature, may choose to skip most of the chapter.

When we refer to hardware reliability, in this chapter and elsewhere, 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 Civil 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 over a particular period of time.

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

Definition 2.3
A **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 dissertation.

Definition 2.4
A **Detectable Failure** is a failure that could be detected at run-time, but which may not be 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
A **Continuing Failure** is a failure that does not prevent the program from continuing (usually not a detectable failure). The problem with continuing failures is that erroneous results from one component will propagate to other components without any indication and may cause failure, either faults or incorrect results, a considerable distance away.

Definition 2.6
An **Operational Profile** is a statistical description of the environment in which a system is used.

Definition 2.7
Failure Rate is the observed failures per unit time.

2.2 Markov Models in Hardware Reliability

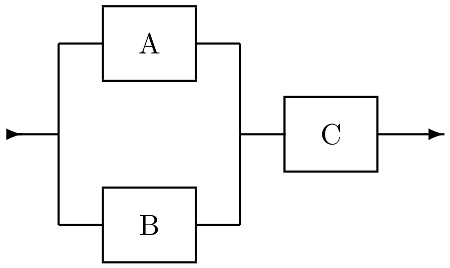
Markov models are commonly used in hardware reliability modeling. They have also been proposed to model software reliability - in particular software component reliability - despite not being well matched by various properties of software. To understand why they are not a good match for software reliability, we will first explore their use in hardware reliability.

A Markov model is useful when a system has a set of states with defined flow rates among the states. The model can be used to determine the probability that, in the steady state, the system will be in each of the states.

Definition 2.8
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, disk controllers and hard disks.

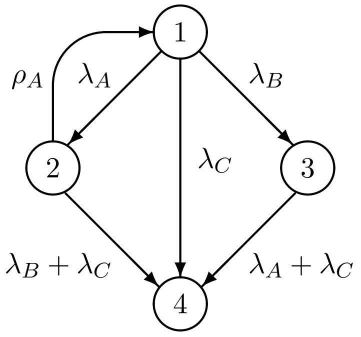
Figure 2.1 A simple three component system



In the 3-component system in Figure 2.1 component C and one of A or B must be working for the system to be working (say C is a computer and A and B are redundant power supplies). 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 three 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 indicates a working component and $\overline{A}$ indicates a failing component). Because the system works when component C and one of A or B are working, there are three working states, (A, B, C), $(A, \overline{B}, C)$, and $(\overline{A}, B, C)$, and five 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:

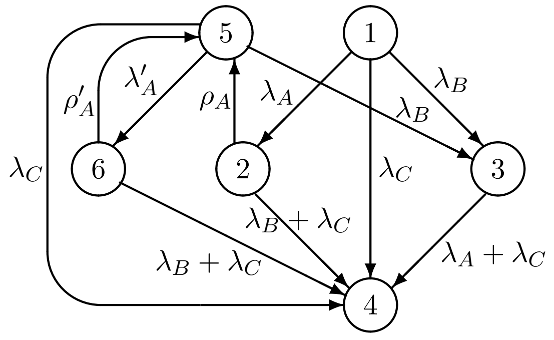
$$\begin{aligned} R(t) &= P[\text{in a success state at time } t] &&= P_1(t) + P_2(t) + P_3(t) \\ &= 1 - P[\text{in failure state at time } t] &&= 1 - P_4(t). \end{aligned} \tag{2.1}$$

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

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

If, for example, component A was less reliable after having been repaired than it was originally, then to maintain Markov properties the state diagram would be unrolled, as in Figure 2.3. Here, the λ'_A and ρ'_A represent the revised failure and repair rates for

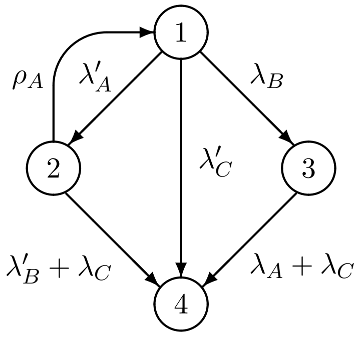
Figure 2.3 Revision of Figure 2.2, where A is not perfectly repairable



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.

Alternatively, the model might be inaccurate because the failure rates were not independent. If a failure of component A could 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 from state 1 to state 4 (reflected in Figure 2.4 as λ'_C, with λ'_A capturing the failures of A that do 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 λ'_B. In this example, for simplicity, we have assumed that component A is not affected by the failure of component B, and that once A is repaired B's reliability is the same as it originally was.

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



2.3 Hardware versus Software Reliability

Current software reliability estimation methods follow the hardware methodology. They use code testing to obtain failure data. This data drives underlying mathematical models by which statistics such as system reliability are estimated. The meaningfulness of the statistics depends on the degree to which the software in question, and the software testing and data, matches the *mathematical model* underlying the estimations (Lyu 1998).

In §2.3.1, we explore how hardware and software reliability differ. In §2.3.2, 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 modeling from that of hardware. We describe why software testing does not provide data analogous to hardware testing in terms of reliability estimation and composition. In §2.4, we explain how typical software tends to violate the underlying assumptions of the models used to estimate reliability.

Because the models for reliability are not a good match for software and software testing, the meaningfulness of the estimates comes 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 §2.4. Chapter 3 presents a definition of reliability, and an approach to its determination, that avoids these problems.

2.3.1 Sources of Failure in Hardware and Software Systems

Engineering is the art of the possible. Therefore, of necessity, every design is a compromise of several elements: costs, utility, and reliability (among others). Reliability for hardware systems is a function of four factors:

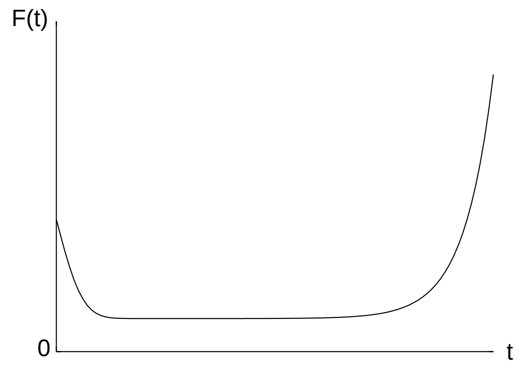
- errors in design,
- errors in manufacture,
- physical defects, and
- chemical and physical wear.

Different designs can trade off the contribution of the various factors against the other elements — particularly cost. Possible trade-offs would include specifying different tolerances to reduce manufacturing errors, or specifying higher quality material to minimize defects and wear. These factors lead to three typical aspects of failure:

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

The characteristic failure curves for these aspects 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.

Figure 2.5 "Bathtub" Failure Curve



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 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 failure 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 failure histories form a family of bathtub curves, which will have the same basic shape, but different trade-offs may lead to changes such as improved (or worse) overall reliability or extended useful life. By using different designs, any of the failure aspects could be reduced, but this will usually be with increased cost (higher quality materials), by increasing other failure aspects, 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 aspects 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, implementation is a part of design) as a source of failure in software systems (Littlewood 1979), and one aspect of failure: load variance failure.

Unfortunately, software systems (and to a lesser extent, discrete hardware systems such as digital electronics) do not exhibit the same continuity properties (which we will explore below) as analog systems (Hamlet 2002). Therefore, considerable care must be taken when attempting to transplant hardware reliability models to software systems, as hinted at by (Laprie and Kanoun 1996).

Operational Profiles

Any reliability figure quoted for a system — hardware or software — must be stated in terms of the usage to which the system will be put, the operational profile. Because of the simple structure of the input domain for many hardware systems, their operational profiles tend to be very coarse-grained, made up of a few large sub-domains. Sub-domains for software systems tend to be much more fine-grained because of the added complexity of the input domain (which may include discrete sets and program structures). Developing operational profiles for software is more complicated even when the software is considered stateless (Donnelly, Everett, Musa, and Wilson 1996, Musa, Fuoco, Irving, and Kropfl 1996). When software retains memory, the complexity of its operational profile is further increased because state must be incorporated into the input domain as extra parameters and results (Woit and Mason 1998b). For example, a program that accumulates values

1 Some people talk about bit-rot, citing problems such as 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. Changing specifications and new versions of the software are also not examples of bit-rot. They are new artifacts and not directly comparable.

might work fine if invoked a few times, but suffer an overflow if invoked a few dozen times. Similar issues can exist in hardware systems with feedback, but are orders of magnitude simpler.

2.3.2 Assumptions Underlying Hardware Reliability

Because of the degree to which hardware systems exhibit simple load continuity, functional continuity, and failure monotonicity, it is usually possible to provide estimates of overall reliability by sampling at relatively few points in the input space, such as the minimum load and the maximum load. Operational profiles for such systems tend to be very robust, in that moderate changes in the operational profile have little effect on the reliability figure, because of the coarse granularity of the input sub-domains. That is, a modification is less likely to cross sub-domain boundaries.

Operational profiles for software systems are potentially much more brittle. Because there may be arbitrarily many input sub-domains, it is generally considered infeasible to identify them. Thus, it is generally not possible to estimate overall reliability through sampling as is done in hardware. Further, a small change in the operational profile could correspond to dramatically different reliability, because it could affect many input sub-domains. See Chapter 3 for our approach to identifying the sub-domains.

2.3.2.1 Load Continuity

Load Continuity is the assumption that the system's input can be modeled by a continuous function, and thus random testing over the input space provides a representative failure rate. For some hardware systems, there is not a single continuous function that accurately models the input. In these cases, the input domain can be divided into a small number of sub-domains and modeling functions can be developed for each. Moreover, hardware systems are often simple enough that a linear function will provide a sufficiently-good approximation to the input domain. There *are* software systems that have a load that can be accurately modeled by a small number of continuous functions. However many do not, especially because typical software domains tend to include discrete sets such as integers or enumerated values.

2.3.2.2 Functional Continuity

Functional Continuity is the assumption that a system's transformation of input to output can be modeled by a continuous function, and thus random testing over an appropriate sub-domain of the input space provides a representative failure rate. It is well-known that a lack of functional continuity is a major problem in determining software reliability. The discrete nature of software is such that a system could succeed on input K but fail on $K + \epsilon$ and $K - \epsilon$, for arbitrarily small ϵ. For software, a simple test of input K provides *no information* about the behaviour of the system at points near K — in contrast to hardware. We will explore in Chapter 3 how to overcome this problem.

2.3.2.3 Failure Monotonicity

Failure Monotonicity is the assumption that if a failure rate is observed with a particular load, a higher load will have at least as high a failure rate and, conversely, a lighter load will have as good, or better, failure rate. This is predicated on there being a partial order on loads so that it makes sense to talk about "higher" loads. Designers of hardware systems can use failure monotonicity to good effect by using margins of error in the over-design of the systems and by using accelerated testing of systems. These practices are generally not applicable in the software world because of issues relating to load and functional continuity.

Margins of Error

Designers of hardware systems can use failure monotonicity to good effect by using margins of error in the over-design of the systems. For example, if a rope is used to lift a 100 kg. load, a designer might choose to assume a load of 200 kg. to handle occasional overloads, and add a 50% overload factor to account for jerky pulling on the rope. In order that the rope can be pre-tested at this worst-case 300 kg. load, the designer might double that (so that the pre-test would not induce failure or weaken the rope) leading to a specification of 600 kg. for the rope. Such margins of error are not always applicable because of weight, cost, or similar considerations,[2] but they generally simplify design in a way that is unavailable in the software world (Hamlet 1999). The analogy in software would be to identify a set, S, of the software input domain so that if the software is built to succeed on S it is guaranteed to succeed on some other sets (Weyuker and Ostrand 1980). The problem of identifying such an S can be simple for hardware and intractable for software. Furthermore, because of a lack of load and functional continuity, the failure-relation can not be established with software testing alone, but requires the incorporation of code analysis.

2 For example, the available ropes meeting the 600 kg. specification might be too heavy, too large for the pulley, or too expensive.

For certain categories of software, margins of error are applicable in a limited sense. For real-time continuous-operation software it can be assumed that if the software is designed so that it can process events at rate N, then it will be able to process events at rate $N - i$. For example, if a switching system is designed to be able to process N simultaneous calls, then fairly straight-forward analysis of the program structure can lead one to reasonably assume that the system can process $N - i$ simultaneous calls. However, even if the software is designed as above, no conclusions are possible about the *failure per call* resulting from design errors. For example, the handling of call forwarding might fail with only two simultaneous attempts on a system that could handle a thousand simultaneous standard calls, so any claim of load continuity in such a system would have to be very carefully specified - unlike the common hardware case.

Accelerated Testing

For hardware, failure monotonicity is used to good effect in accelerated testing, where sequences of increased vibration and temperature have been found to predict failure rates under normal circumstances. Direct translation of this idea to software is not possible because of lack of similar margins of error, as noted above.

2.3.2.4 Sample Continuity

Sample Continuity is the assumption that instances of the system under study come from a continuous sample space - that is they are independent random variables from a particular distribution - so that tests performed on randomly selected instances will allow us to draw statistically valid conclusions about the response of any given instance. This could also be called Testing Continuity. Note that this doesn't always hold in the hardware world. In his design of the Crystal Palace, Joseph Paxton stipulated that *all* cast-iron girders were to be stress-tested, because cast-iron does not have good sample continuity, whereas the wrought-iron trusses needed only sampling (Petroski 1985).

Sample Uniformity

In software and other discrete systems, such as digital electronics, all instances of the system have exactly the same characteristics. With simple systems, sample uniformity improves reliability because a single test of each possible input on *any* system instance will provide complete characterization of the correctness of the output for *all* system instances. Unfortunately, as system complexity increases, the number of potential inputs increases exponentially and exhaustive testing becomes infeasible. Clearly exhaustive testing is intractable for typical software systems.

Redundancy

Sample Continuity supports designing redundancy into a system: by having two or more sub-systems to do a particular job, if one of them fails, the other can take over. This can be extended to voting systems where decisions are made by a majority of redundant systems. Unfortunately, sample uniformity means that there is no benefit in having multiple copies of the same software system, as they will all fail in the same way. It has been proposed to use several different implementations of a system to resolve this problem. Knight and Leveson discuss this issue, but conclude that there is probably little benefit to the approach for software because of the likelihood that implementors will make coincident decisions about the implementation, which means that the implementations don't form a continuous sample space (Knight and Leveson 1990).

Typical hardware systems have properties of load continuity, functional continuity, failure monotonicity, and sample continuity. Methods of reliability calculation were developed under such assumptions. In §2.4 we discuss these methods and identify implications of the lack of these properties in reliability calculation for software systems.

2.4 System and Software Reliability

For a system consisting of hardware components, software components, or some mixture of both, there are three ways that reliability can be established: composition, analogy, and life-testing. In addition, for software, it is possible to *prove* that an implementation meets a specification. Of course, the software must run on some hardware, so a *system* reliability will still require one of the other methods.[3] In the remainder of this section, we explain the application of each of the above general methods of establishing reliability to software systems, and we describe the pitfalls of the imprudent application of hardware techniques. Ramamoorthy and Bastani (1982) discuss various approaches to software reliability from a somewhat different perspective.

2.4.1 Composition

Composition is applicable when a system is composed of parts for which reliabilities are known. By examining the system structure and reliability dependencies, formulas may be derived which can be used to calculate the system reliability. This critically depends on the reliability ascribed to each component being the reliability when the component

3 Although hardware designs can also be proved, they still must be realized in silicon, and hence are subject to traditional hardware failure modes.

is used in the way, and under the conditions, in which it will be used in the system under study.

Composition is a traditional method of estimating hardware reliability. Methods of software reliability composition have been proposed (Littlewood 1979, Cheung 1980, Laprie 1984, Friedman and Voas 1995, Laprie and Kanoun 1996, Woit 1997). However, as explained in §2.5 and §2.6, these methods are inadequate for two reasons: (1) the models used are not reflective of the physical component transition properties in actual software; (2) they assume independence among components — a property violated by most software systems.

2.4.2 Analogy

Analogy can be used when reliability values are available for similar systems and it is clear how to calculate with those reliabilities to determine the reliability of the new system. The applicability of analogy is highly dependent on the experience of the designer to recognize factors that will, or will not, allow a projection of the reliability of past designs onto the current design. For software systems, reliability growth models attempt to predict reliabilities of future versions of the software by analogy to previous versions of the software, although the proponents of this approach generally do not describe it as analogy,. Using these models, measures are calculated regarding when the software is expected to achieve a given reliability target, and what reliability can be expected upon testing termination. The calculations can aid in determining allocation of resources to the project.

During development of the software, it is tested, errors are uncovered, and they are fixed (possibly introducing more errors), making a new version of the software. This new version then undergoes the same process (test, uncover, fix), making a new version, and so forth. As this process continues, failure data is collected. Such data can be in the form of time between failures, or failures per unit of time. When the software is in version i, reliabilities of versions $i+1, i+2, \ldots$ are predicted by extrapolating a curve determined from failure data collected from versions $1, 2, \ldots, i$. Many different models have been proposed to fit curves to failure data (Farr 1996, Stark 1996).

It is commonly assumed that the failure curve of a reliability growth model is the infant-mortality part of a *software* bathtub curve (Lyu 1996a, Lyu 1998). Given the similarity of the shapes of the curves (see Figure 2.5), this assumption is understandable, but not correct. In hardware, the infant-mortality part of the bathtub curve represents the expected failure intensity of *one design* of the system, due to manufacturing errors, defects, and other physical properties that are not present in software. The curve of reliability growth models tracks *several designs* of the system. What growth models *are* doing is attempting to establish a trend-line for a set of similar systems; with an analogy argument, they are projecting a reliability figure for the next one or more iterations of the design. A similar approach may be used in some hardware systems during the prototype period to obtain some prediction of the ultimate reliability of the system, but it is not related to the bathtub curve, *per se*.

An analogy argument must be based on a formulation of continuous load functions and reliabilities of related systems. When software systems exhibit little load and functional continuity, they do not have such functions, as hinted at in (Butler and Finelli 1993, Hamlet 1994b, Hamlet 1994c). The lack of load continuity means that the system reliability is subject to significant variability with minor perturbations in the input, hence there is considerable uncertainty in the true reliability of the existing versions of the system. Because the analogy argument is an extrapolation, such uncertainty can lead to wildly divergent projections. The lack of functional continuity means that even minor changes between the existing versions of the system and future versions can also lead to significant change in system failure rate (Hamlet 1994a, Parnas 1993).

2.4.3 Life-testing

Life-testing is the fall-back method that is used when there are no other means to calculate reliability. For this calculation, sufficient prototypes must be built so that testing will be statistically valid and they must be tested for their lifetime with the expected environment and usage. The observed failures may then be plotted to form the bathtub curve. Parallel testing may reduce the amount of clock time required to do the life testing, and in many hardware systems accelerated testing may reduce the time required for each test.

Developers of safety-critical systems are unwilling to use growth models because in software, the analogy argument is impossible to make with the necessary accuracy. For safety-critical applications, reliability is calculated based on tests of the current version. We refer to these software life-testing models as single-version reliability models.

Estimating reliability with single-version reliability models is time-consuming because a lack of failure monotonicity does not allow any kind of accelerated testing. Thus an extraordinarily large number of tests are required to demonstrate the necessary reliability value (Butler and Finelli 1993, Parnas 1993). This can be somewhat ameliorated, if a system exhibits sample uniformity, by testing in parallel on a very large scale. Imperfect detection of errors can also limit the confidence level achievable with life testing (Ammann et al. 1994). Lack of functional continuity also makes reliability estimation difficult, as explained in §2.3.2.2, because complete characterization of the failure rate requires exhaustive testing (Butler and Finelli 1993). In Chapter 3, we will describe how to achieve functional continuity and thus be able to ameliorate this problem.

$\boxed{2.5}$ State Transition Properties and Continuation Passing Style

In this section, we identify inconsistencies between software systems and the Markov models typically used to represent them. We outline how these problems can be partially overcome by using Continuation Passing Style λ-Calculus (CPS) (Appel 1992). An earlier version of this section and §2.6 were published in (Woit and Mason 1998b).

Parnas differentiates between the mutually exclusive relations "uses" and "invokes" (Parnas 1974) as follows:

- USES(C_i, C_j) iff C_i calls C_j and C_i will be considered incorrect if C_j does not function properly.
- INV(C_i, C_j) iff C_i passes control to C_j but does not use C_j.

Thus if USES(X,Y), then the reliability of component X will *incorporate* the reliability of Y. Their reliabilities (and thus failure rates, because in this instance reliability is 1−failure-rate) will be dependent, and we write RelUSES(X,Y). An assumption of the Markov model is that the future evolution of the system will be independent of its history, which means that transitions between nodes must correspond to INV.

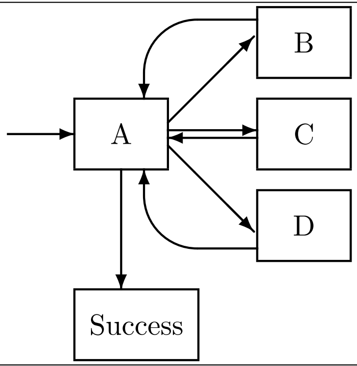
A software component often uses the results of other components to transform its input to output. Consider component A, which is presented in programming languages `C` (Kernighan and Ritchie 1988) and `Scheme` (R4RS 1991) in Figure 2.6. A typical Markov model for A is given in Figure 2.7, in which component interactions are modeled as INV transitions. However, in the actual component interactions of the system given in Figure 2.6, it is apparent that transitions are *not* INV. For example, Figure 2.7 indicates INV(A,B), but the system of Figure 2.6 indicates USES(A,B). Thus, the typical Markov model is not an accurate description of the actual system. Worse, the relations between components A, B, C and D in Figure 2.6 are RelUSES(A,B), RelUSES(A,C), and RelUSES(A,D), which means the reliability of A already incorporates the reliabilities of components B, C, and D. In fact, for any system thus modeled, the reliability of the *start* component, C_1, incorporates the reliabilities of components C_i, where C_i is the transitive closure of USES on C_1. Thus, the reliability for C_1 *is* the overall system reliability, making moot the entire exercise of calculating system reliability from component reliabilities.

The problems described above are alleviated by designing or transforming the system into CPS form. When the system is CPS converted (or designed), all of the components are related by INV, none of the components are related by USES, and each component

Figure 2.6 Component A

```
  void A() {       (define (A)
1   int x=0;          (define x 0)
2   do {              (define (loop)
3      if (x%2==0)      (if (even? x)
4         B(x);             (B x)
5      else
6         C(x);             (C x))
7      x=x+1;           (set! x (+ x 1))
8   } while (x<10);     (if (< x 10)
9                           (loop)))
10                    (loop)
11   y=D(x);          (set! y (D x))
12   printf(          (format #t
13    "y=%d\n",y);}        "y=~a~%" y))
```

Figure 2.7 Simplistic model of A



performs an atomic transformation of its input to output. The Markov model is thus applicable to software components in terms of component transformations.

The component, A, of Figure 2.6, can be transformed into CPS by dividing it into *fragments*, A0, A1, A2, and A3 as follows.

A0: set x=0 and invoke A1 (line 1)

A1: depending on the result of the `if` statement, invoke either B or C (lines 3–6)

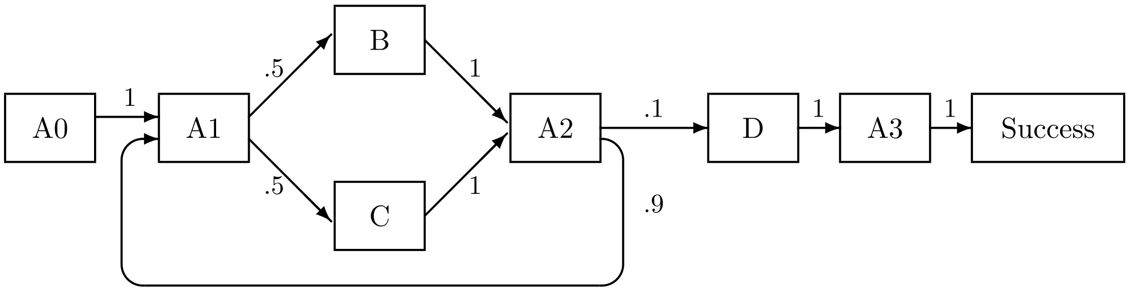
A2: increment x; depending on the result of `x<10`, invoke either A1 or D (lines 2,7–10)

A3: set y and print (lines 11–13)

Components B and C must be modified to invoke their continuation, A2. Component D must be modified to invoke its continuation, A3. The components of the CPS-converted system are thus A0, A1, A2, A3, B, C, D, and the corresponding Markov model is given in Figure 2.8. The nodes in the model correspond to the program fragments from the CPS-converted code, and the edges are labeled with the probability of transfer along that edge to the next fragment.

Note that the two problems above are now solved. Each component performs an atomic transformation of input to output, and uses the results of no other component in its transformation. Thus, INV holds for all component interactions. The reliability of any component is not dependent on that of other components because no USES relation exists among components and because the components use no global variables.[4]

Figure 2.8 Markov model after CPS conversion, with edge probabilities



2.5.1 CPS Transformation

To CPS convert a procedure/function, one partitions the component into *fragments*, which are sequences of instructions that do not involve a call to a component. Each fragment may be considered to be a function. Fragments become components in the CPS converted system. The new set of components *invoke* other components, passing along relevant program state and the next component to be executed. For example, consider component C in Figure 2.9. The section prior to the call to F becomes a fragment, C1. The section following the call to F becomes a fragment (function), C2. C1 executes in the state S1 that is visible in C before the call to F. C2 executes in the state S2 that is visible in C after the call to F. The components of CPS conversion are C1, F and C2. C1 passes to F both C2 and S1. F will perform its function in state S1, and it in turn will pass control and S2 (including the result of F) to C2. C2 will perform its function in state S2.

We have found CPS conversion to be straightforward when using functional programming languages because of the high-level data-structure facilities. CPS conversion of

4 The relation of reliability and program state is investigated in §2.6.

Figure 2.9 Component C

$$\begin{array}{ll} \mathrm{C1} & \begin{cases} \cdots \\ \cdots \\ \cdots \end{cases} \\ & \ \texttt{F}(\cdots) \\ \mathrm{C2} & \begin{cases} \cdots \\ \cdots \\ \cdots \end{cases} \end{array}$$

component A from Figure 2.6 using the programming language `Scheme` is given in Figure 2.10. In a language such as `C`, it is slightly more complicated, but can be facilitated using macros. Fan (2002) constructed tools by which a CPS conversion can be automatically produced for `C`.

Figure 2.10 CPS conversion of component A from Figure 2.6

```
(define (A)
  (define (a1)
    (if (even? x)
        (B a2 x)
        (C a2 x)))
  (define (a2 result)
    (set! x (+ x 1))
    (if (< x 10)
        (a1)
        (D a3 x)))
  (define (a3 result)
    (set! y result)
    (format #t "y=~a~%" y))
  (define x 0)
  (a1))
```

2.5.2 CPS and Markov Properties

While CPS conversion of programs is a *necessary* condition to give them the Markov property, it unfortunately is not *sufficient*. It correctly exposes the internal node structure of individual functions, as demonstrated in the previous section. It does not make inter-component transitions memory-less, as required by Markov models, because components still may use the continuation to jump to different nodes. Thus to correctly build a Markov model of components (which have been CPS converted), each component must be replicated for each call point. Furthermore, this must be done transitively, so that any components invoked from a replicated component must themselves be replicated. Obviously this cannot be done completely for recursive procedures, and can cause an exponential explosion of nodes and edges. We observed this problem, and reported on it in (Woit and Mason 1998a). The related observation was that by unrolling the recursive procedure we could achieve increasingly accurate Markov models. This unrolling could be done automatically by an analysis program. By unrolling to the maximum depth actually used by a program executing with a particular operational profile, full accuracy could be attained.

2.6 Static-Independence

§2.5.1 showed how to break modules into fragments so that the INV relation holds on all component interactions in a system. In order to use the resulting components in a Markov model, the Markov property must hold: the probability distribution for the transitions from the current node must be independent of the path taken to arrive at the current node.

Like most other papers that discuss composing component reliabilities, Woit (1997) assumes that modules are "independent", without stating exactly what independence means or why that assumption is important. We define *static-independence* to mean that two modules do not share state. This is important because if there is state shared between modules, the complete history of that state must be captured in the Markov model, with a resulting exponential explosion in nodes and edges. For example, if two modules A and B shared a global variable `x`, with A adding various values to `x` and B looping based on the value of `x`, then the number of times that A was called would affect the edge probabilities for B. This is similar to the situation as pointed out by Woit: if a module is called from several different modules, the calling structure of the model of the system may require splitting nodes to maintain the Markov property. It appears that this is what is generally meant by the independence assumption.

State in a system consists of:

1. Global variables. This is the simplest and most explicit state commonly used by programmers.
2. Mutable data-structures. Non-global variables are sometimes passed as parameters to other modules which mutate the values. A common example of this is pass-by-reference. This is another form of explicit state.

3. Memory. In some sense this can be considered a generalization of mutable data-structures, but in an unsafe language such as `C`, *any* write to the target of a pointer must be assumed to change every memory location in the program (because it is very difficult to analytically determine which locations actually were changed, in the absence of a proof to the contrary), although Ghiya and Hendren have made some progress on this problem(Ghiya and Hendren 1998).
4. I/O state. By this we refer to all interactions between the system and the environment in which it runs. This includes: the position of file read/write pointers, values in device registers, segments of files that can be read and written, state maintained by the operating system or other systems with which the current system interacts, and physical state such as the velocity of a vehicle.

While some system parameters, such as the amount of free memory or the time of day, are certainly state in the larger sense, they are rarely relevant to a running program, so we omit them from the definition of static-independence. (When relevant, they must be incorporated into the model, or they impose a separate proof obligation on a system designer.)

There are three design rules that help to establish static-independence:

- Write in a safe programming language such as `Scheme`, `Java` (Joy et al. 2000), `ML` or `Ada` (Ada Joint Program Office 1983) where pointers are constrained and memory management is automated.
- Write using the functional-programming paradigm. Individual components need not be purely functional, but there must be no mutable global variables or data-structures *between* components (fragments) or retained between calls to a component. Calls that pass parameters by reference should be changed to pass by value-result. The desirability of writing in this paradigm has been recognized by Lucent and Ericsson in their support for the development of the mostly-functional programming languages Standard ML (Milner et al. 1990) and `Erlang`.
- Maintain atomicity. Where a component needs to update I/O state, it must assume nothing about the existing state when performing its update and must perform the update as a transaction entirely within the component. For example, upon entry, a component must not assume that the writing point on the screen is in a particular position, but must move to the desired position before writing.

Building systems where the components use CPS form and are static-independent would mean that the components meet the requirements of the Markov model, and we would be able to calculate system reliability from component reliabilities. As reported in (Woit and Mason 1998a), we have built a version of the Unix `grep` utility that met the requirements for static independence and we were able to use it in a Markov model to calculate reliabilities near those of the actual program. Because we were unrolling the recursion by hand we only unrolled a few times, which limited the accuracy.

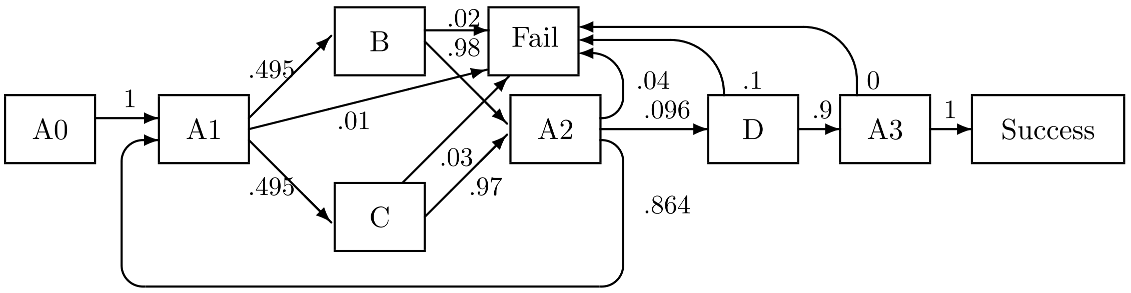
Unfortunately, not all systems can be made static-independent, and even for those that can, replacing or changing a component may require a great deal of work to find the reliability of the other components in the system in the new circumstances.

2.7 Why a new approach is warranted

Although the CPS analysis in §2.5 and the static-independence requirement in §2.6 are necessary, they may still not be sufficient to make a set of components and their transitions valid for Markov analysis. In order for Markov analysis to be valid, for each node the probabilities on all exit edges must be unchanged, regardless of which input edge passed control to the node. Considering Figure 2.8 and the program in Figure 2.6, it can be seen that the first nine times that control reaches node A2, the edge from A2 to A1 will have probability 1 and the edge from A2 to D will have probability 0. The tenth time that control reaches node A2, the edge from A2 to A1 will have probability 0 and the edge from A2 to D will have probability 1. Therefore, the model does not have the Markov property.

For a Markov-based reliability analysis, the model in Figure 2.8 would be augmented with a Failure node and every existing node except the Success node would have an edge added from the node to the Failure node. This edge would be labeled with the failure rate of the source node and the other edges from the source node would be reduced proportionally. An example of this is shown in Figure 2.11 where the following nodes have hypothetical non-zero failure rates: A1 - 1%, B - 2%, C - 3%, A2 - 4%, and D - 10%.

Figure 2.11 Markov model from Figure 2.8, with failure rates



For this static labeling to be correct, every node would have to have exactly the same failure rate *every* time it executes, regardless of the path that led to it. For very low reliability requirements and very stable components this approach *may* give reasonably estimates, but it is unlikely that anyone would be willing to go to the required work unless high reliability was a requirement. All of the previous work in the area, including our own, eventually falls flat when high reliability is a requirement. They fail because the discontinuities extant in the software domain potentially lead to different failure rates with only minor perturbations to the input to any node and because they have no accurate way to model the inputs that the components will see.

The balance of this dissertation describes a useful form of continuity in software and exploits this to produce estimates of “software reliability” that are *correct*; enough so to make composing of component reliabilities worth considering.

$\boxed{3}$ Subdomain Analysis

In §2.1, we gave a general-purpose definition of reliability that said that it was the probability that a system will operate according to specification on a particular operational profile over a period of time. Because software doesn't have the same temporal properties as hardware, such as wear and aging, every execution of a particular program *in a given environment* will have identical outcome. Therefore the term reliability is a bit of a misnomer and we will use the term Probabilistic Correctness.

Definition 3.1
The **Probabilistic Correctness** (ρ) of a program (system or component) is the sum over all inputs of the likelihood of each input occurring in the input times the correctness function, where the correctness function is 1 or 0 as the program matches, or does not match, its specification at each point. Formally,

$$\rho = \sum_{x \in X} \begin{cases} p_{\mathtt{OP}}(x), & \text{if } c(x) = s(x); \\ 0, & \text{otherwise;} \end{cases} \tag{3.1}$$

where X is the input domain, $p_{\mathtt{OP}}(x)$ is the probability, based on the Operational Profile, that a random selection from input domain X will be the value x, $c(x)$ is the output of the *program code* on input x, and $s(x)$ is the output *specified* for input x.[5] This is the same definition used by Thayer et al.(Thayer et al. 1978).

Unfortunately this value is not directly calculable as most programs have essentially infinite possible discrete inputs. We must find a useful way to group these inputs together into coherent and useful sub-domains. In Chapter 4 we will add necessary computational tools and in Chapter 5 explain the calculation of probabilistic correctness.

Precursors of some of the ideas in this chapter were published as (Mason and Woit 2000). Related work by others will be discussed in §3.8.

5 This does not address continuous-running systems. They will be discussed in §5.4.

3.1 Continuous Domains

As was explained in Chapter 2, the major difference between hardware and software systems is continuity. While the total input space of a system is almost certain to be discontinuous, it should be possible to partition it into a series of domains that are each continuous. If we can identify sub-domains such that every point in the sub-domain experiences the *exact same* sequence of program instructions (ignoring branch instructions), that have the identical specification, and such that no machine exceptions would occur for any point in the sub-domain, we would have a useful form of local functional continuity. In the next two sections, we will develop just such a notion of sub-domains. The domains are essentially the same domains described by Richardson and Clarke, but there are significant differences in how they are used(Richardson and Clarke 1985).

3.2 Code Domains

First of all, we will look at the sub-domains that are provided by the program code. The most obvious domains for a notion of continuity would be the set of points that have the same stream of instructions executed upon them. The idea of identical instruction streams is captured in the definition of a path.

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

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.

The important point here is that every point that arrives at the beginning of a basic block will execute exactly the same sequence of instructions within the block, and hence every point that arrives at the beginning of a path will execute exactly the same sequence of instructions through the path (and program).

Figure 3.1 shows a simple program with the basic blocks labeled.

Conventionally, the paths for this code are: abg, abcdfbg, abcdfbcdfbg, abcdfbcdf-bcdfbg, abcdfbcdfbcdfbcefbg. However, we are concerned with reliability, so we should explicitly capture the detectable 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. This explicit recognition of detectable failures leads to the definitions of a fault block.

Figure 3.1 A simple program with basic blocks

```
    [   input z;
a   |   x=1;
    [   y=0;
b   [   while (x<5 && x<z) do
c   [     if (x<4) then
d   [       y=y+x;
          else
e   [       w=z*z-25;
    [       y=y/w;
          fi
f   [     x=x+1;
        od
g   [   output y;
```

Definition 3.4
A **Fault Block** is either the single statement $\langle fault \rangle$ or a basic block. In the latter case, every instruction in the block will be executed and no instruction will have anomalous behaviour such as subscript out-of-range, overflow, underflow, or divide by zero. Any such anomalous behaviour will be identified with a $\langle fault \rangle$. Every $\langle fault \rangle$ is a direct consequence of some instruction; e.g. a $\langle fault \rangle$ would exist from a subscripting operation, and it would be conditionalized in the analysis by the index value being out of range of the array. If we assume a safe language (essentially pointer-free), then all potential $\langle fault \rangle$s can be identified statically.[6]

With the conditional checks to identify the potential $\langle fault \rangle$s, we get the revised blocks in Figure 3.2.[7]

The paths for this code are shown in Table 3.1. Note that the two paths that don't reach "g" are paths ending in $\langle fault \rangle$. These paths are known to fail; the rest of the paths will complete, but the results must be compared against the specification.

Set of Code Domains

Each of these paths defines a **Path Domain**, which is a subset of the system's potential input space that exercises the path and terminates. The set of all of the path do-

6 All potential $\langle fault \rangle$s can be identified in unsafe languages too, but essentially every operation can potentially fail, so the analysis quickly becomes intractable.

7 To simplify the exposition, we have removed some checks for faults where the faults are impossible due to previous checks or faults.

Figure 3.2 A simple program with fault blocks and fault checking

```
   [ input z;
a  [ x=1;
   [ y=0;
b  [ while x<5 && x<z do
c  [   if x<4 then
d  [     y=y+x;
       else
h  [     if z>$\sqrt{MAX}$ then
i  [       $\langle fault \rangle$
         else
e  [       w=z*z-25;
           if y/w is an error then
j  [         $\langle fault \rangle$
           else
e' [         y=y/w;
           fi
         fi
       fi
f  [   x=x+1;
     od
g  [ output y;
```

Table 3.1 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	abcdfbcdfbcdfbchi	$z > \sqrt{MAX}$	$\langle fault \rangle$
6	abcdfbcdfbcdfbchej	$z = 5$	$\langle fault \rangle$
7	abcdfbcdfbcdfbchee'fbg	$4 < z < 5 \vee 5 < z \leq \sqrt{MAX}$	$\frac{6}{z^2-25}$

mains, augmented by the set of all input points $\hat{C}$ not included in any path domains (and hence with a null implementation), defines the set of **Code Domains**, C. There are four important properties for each path.

1. Each of the paths will calculate a result: a $\langle fault \rangle$ or a (possibly constant) function of the inputs.
2. Exactly the same sequence of machine instructions will be executed for every possible value in a single path-domain.
3. 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).
4. The domain is a conjunctive predicate of all of the conditionals that must be true to form the particular path from program start to end.

Thus each Path Domain exhibits Functional Continuity and is a candidate for the continuous domains we are looking for.

3.3 Specification Domains

It is clear that our definition of probabilistic correctness 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 sub-domains that would signal faults[8], but that certainly doesn't imply that all other input values are correct.

Specifications must be formal to be of any use for calculating probabilistic correctness. If a specification is executable, then an analysis similar to that in the previous section can be applied to it and the appropriate domains and results may be extracted. If a specification is not executable, corresponding analysis must be applied to acquire the domains extant in the specification. For example if the specification were provided in Parnas tables(Janicki 1995) or other decision tables, then each sub-domain could be read off the table. For a specification written in a language like Z (Potter et al. 1996, Jacky 1997), the specification domains would be derived from variable declarations and predicates.

The domains recovered from the specification, augmented by the set of all input points $\hat{S}$ not included in the specification (and hence with a null specification), defines the set of specification domains S.

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

It is worth noting that errors in the specification or oracle (which may be used in some of the characterizations in §3.4.1) will go undetected. We will here assume that there are no such errors.

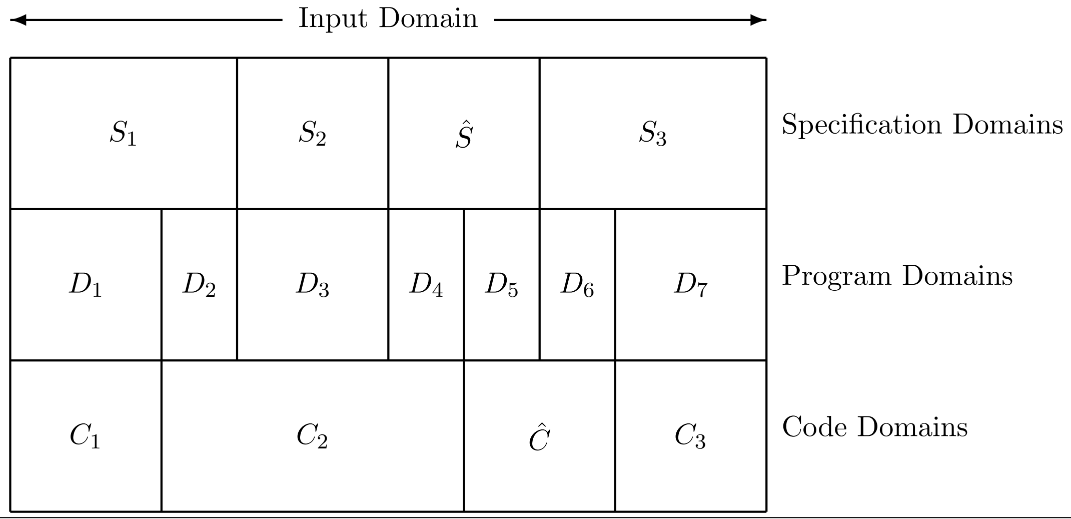
3.4 Program Domains and Failure Rates

Given 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 both a code domain and a specification domain.

$$D = \{\, c \cap s \mid c \in C, s \in S \,\} - \{\emptyset\} \tag{3.2}$$

By construction, each of these program domains is a subset of some specification domain and therefore has a single specification. Similarly, each of these program domains is also a subset of some code domain and therefore has a single computed result. An example of this is shown in Figure 3.3.

Figure 3.3 Input Domain Partitioning



3.4.1 Determination of Failure Rate

Once the program domains have been determined, the failure rate must be determined for each domain. That is, for every program domain D_i, a failure rate F_i must be assigned. The failure rate is independent of any operational profile, whether used to partition the sub-domains, or that encountered in production.

If the specification domain for a particular program domain is the set of unspecified points $\hat{S}$, such as D_4 and D_5 in Figure 3.3, then, according to logic, any implementation will meet this (null) specification so $F_i = 0$.[9] If the code domain for a particular program domain is the set of unimplemented points $\hat{C}$, such as D_6 in Figure 3.3, then $F_i = 1$. If the code domain represents a $\langle fault \rangle$ path, then $F_i = 1$. Otherwise, there is some chance that the code produces the specified result and therefore it will have some lesser failure rate, and a comparison must be made between the specification and implementation for the particular program domain.

When comparing the result produced by the code with the specified result, there are several ways we could determine whether the code meets the specification.

3.4.2 Exact Equivalence

If the results of the implementation and the specification are calculations that can be proven equivalent in the specification language and the implementation language, $F_i = 0$. If they can be proven to be different, $F_i = 1$.

3.4.3 Enumerable

If the program domain has a very small number of members, then it may be practical to run the code on each point in the domain and verify that the result is the same as the one specified for that point. If they all pass, $F_i = 0$. If not, the likely approach would be to fix the exposed bug and rerun the process to determine the new code domains. Alternatively, the failing points can be put in a domain of their own with $F_i = 1$, and the sub-domain containing the correct points can be assigned $F_i = 0$. Alternatively, a very conservative approach could be taken and assign the whole sub-domain $F_i = 1$. (Knight, Cass, Fernandez, and Wika 1994) has suggested that this kind of characterization is possible more often than one might think(Knight, Cass, Fernandez, and Wika 1994).

3.4.4 Linear

If the specification domain is of a simple ordered set, such as integers or the real numbers, and all the transformations, both specification and implementation, on that input are affine (multiplication by a constant, addition, subtraction), then the result can be

9 Although it would probably be desirable to point out the unspecified but implemented domain.

tested at the end points of the domain and, if correct at both ends, a failure rate of 0 can be assigned. This is a very common situation, as confirmed by computer instruction set studies where almost all operations turn out to be data movement, addition, or multiplication-by-a-constant (Patterson and Hennessy 1994, Geller 1978).

If the specification domain has two ordered sets then it describes a plane, and if more than two, a hyper-plane. If there are any parameters that do not have a total order, then there cannot be a hyper-plane. If the input domain is a hyper-plane and the specified results and the code results are also hyper-planes (because they are affine transformations of the input(s)), the results can be shown to be equal by testing at the 2^n vertices resulting from the various extrema for each of the n variables. If they all pass, then $F_i = 0$. If not the likely approach would be to fix the exposed bug and rerun the process to determine the new code domains. Alternatively, a conservative failure rate of $F_i = 1$ can be assigned.

3.4.5 Polynomial

If the specification and implementation functions over the path domain are both polynomials of the same degree n of the inputs then a sampling of $n+1$ points will establish the equivalence of the functions and allow an assignment of $F_i = 0$, or alternately a conservative assignment $F_i = 1$. The sampling of $n+1$ points can be random, although a more sensitive test may be achieved by testing at key points of the polynomial (such as local maxima, minima and roots). It is important that the sampling not be influenced by any operational profile, because that might cluster the samples and make the resulting test less sensitive to errors. This is essentially Algebraic Program Testing (Howden 1978).

3.4.6 Sampling

If none of the other approaches works, the fall-back is to perform a uniform random sampling of the domain. As above, the sampling should not be influenced by any operational profile, because that might cluster the samples and make the resulting test less statistically valid. Unlike the usual case in software sampling, sampling of a program domain is sound because the domain has been constructed so that no discontinuities can exist. Testing n samples with k errors will allow the test analyst to derive an expected failure rate using standard statistical techniques such as a binomial distribution(Littlewood and Stringini 1993). If errors are found, the likely approach would be to fix the exposed bug and rerun the process to determine the new code domains. Alternatively, if the expected failure rate lies within an acceptable confidence level, then the assessed reliability can be used as the failure rate, F_i. It should be noted that if the program is to be composed with others, or the output will be the input to another program, *very* high confidence levels will be required, as discussed in §5.4. If confidence intervals are introduced into the failure rate, they must be carried throughout all the subsequent calculations, all the way through any composition.

3.4.7 Summary of Probabilistic Correctness

Because every point in the input space belongs to one of the program domains, we can re-write equation 3.1 as,

$$\rho = 1 - \sum_{i=1}^{|D|} F_i \sum_{x \in D_i} p_{\mathtt{OP}}(x) \tag{3.3}$$

where D_i are the program domains, D is the set of program domains, $\bigcup D_i$ is the complete input domain, $p_{\mathtt{OP}}(x)$ is the probability, based on the Operational Profile, that a random selection from the input domain will be x, and F_i is the failure rate for program domain D_i, as determined by §3.4.1-3.4.6.

3.5 Dealing with an infinity of path-domains

There are potentially a lot of paths in a typical program. Even a simple function that has three separate `if` statements will have eight 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 two such loops, you square that number ... but you get the idea. In fact, it's worse than that - either or both of C and S may not be finite sets and therefore D may not be finite. In most applications, this will not matter. We can often calculate a (conservative) close approximation to a program's failure rate by analysing the most interesting and influential program domains in D and assuming the worst about the unanalysed elements of D.

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

At this point, the reader may be protesting that in high reliability applications all of those exceptional cases are very important. The response is that it is *critical* that the operational profile be accurate. If it *is* accurate, and the particular path has extremely low probability of being executed, then by definition it cannot contribute significantly to the system failure rate. It has been previously pointed out that testing for debugging and reliability-determination purposes are often different (Frankl et al. 1998).

The rest of this section describes a method to generate the paths that *will* contribute to the program's probabilistic correctness. Even though some very simple components can have an immense number of potentially interesting domains, the actual use of the component can reduce the number of relevant domains. For example, a common integer exponentiation algorithm generates separate paths for every possible integer exponent, but if the application calculates powers of integers on a 32-bit computer, no exponent larger than 31 will have a meaningful result (because the smallest interesting base, two, raised to the power 32 is already larger than the maximum positive integer).

3.5.1 Traditional Path Generation

Assume that the program has been broken up into basic blocks (a standard operation for a compiler or most program analysers) or, in our case, fault blocks, and that `startNode` is the first fault block in the program. Then Algorithm 1 shows the standard algorithm for generating all paths. This algorithm generates the paths in shortest-first order. Nodes are assumed to end with one or two jump targets, or to be exit or $\langle fault \rangle$ nodes.

In Algorithm 1 there are three classes referenced.

- `PriorityQueue` is a Priority Queue of nodes waiting to be examined. It has three methods/fields:
  - `add(priority,element)` puts the `element` into the queue, ordered by the priority;
  - `notEmpty` signifies that there are more elements in the queue waiting to be examined;
  - `next` returns the highest priority element in the queue - in this case a pair of node and path.
- `Node` represents an individual Fault Block. It has five methods/fields:
  - `exit` is true if this is a node that will return a result, and the function will exit;
  - `fault` is true if this is a faulting node;
  - `jumps` is the number of paths from this node to others - apart from exit and fault nodes, each node will have one or two jumps to other nodes;
  - `target` is an array with `jumps` references from this node to other nodes;

  - `minlength` is the minimum possible path to an exit or fault node - produced from the analysis that broke the program into Fault Blocks.

- `Path` represents the sequence of Fault Blocks that make up a path. There are two methods:

  - `add(node)` adds a node to a path;
  - `clone()` makes a copy of the current path;
  - `emit(type)` outputs a path, along with the type of path.

Algorithm 1 Generate All Paths from Block List

```
procedure GeneratePath(startNode)
  workList := new PriorityQueue()
  workList.add(0,(startNode,new Path())
  while workList.notEmpty do
    node,path := workList.next
    path.add(node)
    if node.exit then
      path.emit('exit')
    else if node.fault then
          path.emit('fault')
        else if node.jumps=1 then
              workList.add(node.target(1).minlength,
                  (node.target(1),path))
            else
              workList.add(node.target(1).minlength,
                  (node.target(1),path.clone()))
              workList.add(node.target(2).minlength,
                  (node.target(2),path.clone()))
            fi
        fi
    fi
  od
```

This algorithm is purely structural; it generates all the paths through the program, caring only whether a fault block has one, two or no next-nodes.

After initialization, the algorithm repeatedly gets the next node and path, adds the node to the path, and then either emits the path (if it's an exit or fault) or puts the next node(s) into the queue. By maintaining the queue in shortest-path order, it emits the paths in increasing order of length.

3.5.2 Probabilistic Path Generation

For our purposes, we want an algorithm that generates the most important paths first. That is, the paths with the highest likelihood of being executed. Algorithm 2 shows the algorithm for generating the paths in most frequent order. Whereas Algorithm 1 is purely structural and static, Algorithm 2 is partially dynamic. It doesn't compute with the actual values of the input parameters, but with their statistical properties as described by `inputValues`. The operation of this algorithm is a form of abstract interpretation (Cousot and Cousot 1992, Shivers 1991) where calculations with the input parameters are symbolic and predicates are used to split off path domains. Thus it can run with an arbitrary distribution for the `inputValues`.

Algorithm 2 Generate Paths in Frequency Order

```
procedure GeneratePathInFreqOrder(startNode,inputDomain,inputValues)
  workList := new PriorityQueue()
  workList.add((0,0),(startNode,new Path(),inputDomain,inputValues))
  while workList.notEmpty do
    node,path,domain,result := workList.next
    path.add(node)
    if node.exit then
      path.emit('exit',domain,result)
    else if node.fault then
        path.emit('fault',domain)
      else
        newResult := node.calculate(result,domain)
        if node.jumps=1 then
          workList.add((domain.weight,node.target(1).minlength),
              (node.target(1),path,domain,newResult))
        else
          newDomain1 := node.filter(1,domain)
          workList.add((newDomain1.weight,node.target(1).minlength),
              (node.target(1),path.clone(),newDomain1,newResult))
          newDomain2 := node.filter(2,domain)
          workList.add((newDomain2.weight,node.target(2).minlength),
              (node.target(2),path.clone(),newDomain2,newResult))
        fi
      fi
    fi
  od
```

The same assumptions apply as before, except for the following differences.

1. The priority queue has values which are the 4-tuple of target node and path, as before, augmented by the domain expressed as this path and the set of variable bindings and values calculated to this point in the execution of this path.
2. The priority queue orders on a pair, maximizing the first of the pair (the estimated coverage of the input space), and minimizing the second of the pair (the minimum path length).
3. `inputDomain` is the distribution representing the initial, and complete, input domain.

The following changes/additions have been made to the methods.

- `path.emit` has additional parameters for the domain predicate (which is the conjunction of the comparisons that specify which values are included in the domain) and the result calculated by the path.
- `node.calculate` returns the new value (bound variables) calculated by all of the statements and expressions in that node based on the results to date and the domain predicate.
- `node.filter` returns the new domain predicate based on the incoming domain predicate and the restrictions from this node associated with the decisions leading to the target - 1 or 2 as specified by the parameter.
- `domain.weight` approximates the proportion of the whole input space that is covered by the domain, weighted by the operational profile, used to produce the most important domains first.

Complexity

Because each step of the process performs an abstract interpretation on a set of code, it can take an arbitrary amount of time to do each step. It is even conceivable that a given step may not terminate.

More likely is that the Operational Profile does not sufficiently distinguish the domains into high-frequency and low-frequency domains. In this case, an arbitrary number of interpretation steps could be required before the next domain will be output. Although this is a possible outcome, we believe there are a significant set of programs that do not suffer from this problem.

3.6 Experimental Implementation

Scheme is a general purpose programming language of the LISP family. A few years ago, the author wrote an interpreter for Scheme, called mscheme, to pursue some experiments in operating systems and programming languages. As part of this thesis work, the mscheme interpreter was extended to support the derivation of sub-domains and use of probability distributions. This is a very preliminary implementation, that has served more as a platform for experimentation than anything resembling a robust tool.

The implementation is briefly described to give a sense of how an interface to the algorithms might be designed and to show how little change was required in the interpreter to add the algorithms.

3.6.1 Implementation Details

Because the original interpreter was written by the author a couple of years previous to this experimentation, some familiarity with the code can be assumed, but there was originally no provision to allow this extension. Despite that, only three small changes were required to the execution engine of the interpreter to accommodate this analysis.

- The match for `case` had to be changed to go in all directions with various predicates, i.e. add all the alternatives to the priority queue, rather than just one direction as before.
- Results from primitives, such as divide, can now return multiple domains, and this had to be handled to add the domains to the priority queue of pending paths.
- When there are no paths to evaluate, a new path is pulled from the priority queue.

There are other changes elsewhere in the system, including the interface described in §3.6.2, and extending the primitive operators, but the total code to support random parameter distributions in mscheme is about 600 lines. The current implementation of input distributions in mscheme is restricted to discrete random variables.

All of the standard operators on numbers and the numeric comparison operators have been extended so that they will accept random variables and symbolic expressions as parameters and return symbolic expressions as results.

3.6.2 Program Analysis Interface

These are the functions that would be the basis to build a production analysis tool, but are presented here simply to help explain the examples in §3.6.3.

`(pdf-eval callback function p1 p2 ...)` is the operation that takes a function and a set of parameters, possibly random, and performs an abstract interpretation of the user function. For each path that is discovered in the function, the callback function is called with:

1. an estimate of the probability of the path being executed, based on the input distributions;
2. the predicate that defines the input domain that corresponds to the path; and
3. the produced result (whether $\langle fault \rangle$ or valid result).

These partial results will be produced in approximately most-probable to least-probable order. This may not be strictly descending probable order because of simplifications used to make the analysis run faster.

`(pdf-discrete 1 5)` will create an integer random variable with 5 possible values, each of the numbers 1, 2, 3, 4, and 5, all with a probability of 0.2.

3.6.3 Examples of Extracted Domains

To give some sense of how the interface works, here are a couple of examples of path generation from ordinary `Scheme` functions.

In Figure 3.4, all five possible paths through the component are output. This demonstrates a key result of this analysis, because rather than producing the two paths obvious in the code, the divide operator has produced three possible paths because of the discontinuity at zero. The failing path is immediately output, and the other two paths are bifurcated by the '`<`' comparison. Even though the particular parameter passed into the evaluation didn't have instances that triggered two of the cases they were still output, albeit last because they had zero probability with this distribution. Note that the 4th case is actually impossible, and could be removed by a sophisticated mathematical analyser.

Figure 3.5 shows the output from the factorial function.[10] It initially generated the two domains triggered by the provided random variable, and then proceeds to generate the remaining, infinite, set of alternative domains. This demonstrates clearly that a production version of the tool would have to be augmented with some knowledge of algebra and basic mathematics so that it could prevent generation of impossible paths.

10 The output was edited slightly utilizing mathematical identities to shorten the lines for expository purposes.

Figure 3.4 Generated Paths from Simple Function

```
> (define (callback freq predicate result)
>    (display (list freq predicate result))
>    (newline))

> (define (example x)
>    (if (< (/ 20 x) 3)
>       x
>       3))

> (pdf-eval callback
>          example
>          (pdf-discrete -2 6))

(0.200000 (and (= x 0)) <fault:divide by zero>)
(0.600000 (and (> x 0) (>= (/ 20 x) 3)) 3)
(0.200000 (and (< x 0) (< (/ 20 x) 3)) x)
(0.000000 (and (< x 0) (>= (/ 20 x) 3)) 3)
(0.000000 (and (> x 0) (< (/ 20 x) 3)) x)
```

Figure 3.5 Generated Paths from `fact` Function

```
> (define (fact n)
>    (if (< n 2)
>        1
>        (* n (fact (- n 1)))))

> (pdf-eval callback fact (pdf-discrete 0 2))
(0.666667 (and (< n 2)) 1)
(0.333333 (and (>= n 2) (< (- n 1) 2)) (* n 1))
(0.000000 (and (>= n 2) (>= (- n 1) 2) (< (- n 2) 2)) (* n (* (- n 1) 1)))
(0.000000 (and (>= n 2) (>= (- n 1) 2) (>= (- n 2) 2) (< (- n 3) 2)) (* n
...
```

3.7 Where does the Operational Profile enter the picture?

By analysing the implementation and the specification we have identified the program domains and we have an accurate characterization of the reliability of each program domain. By weighting these domains by the probability of inputs from each domain, we can calculate the program reliability. If we have an accurate operational profile for a program as it is actually to be used, we can predict its reliability accurately. This is likely to be intractable for complete systems because normally they will have too many domains to be usable.

However, finding accurate operational profiles for components is likely to be more feasible. We will discuss how to do this in Chapter 5, once we've laid some ground work in Chapter 4.

3.8 Related Work

Analysing a program's domains based on the paths though the code was described in the context of testing in (Howden 1976). Around the same time, the idea of calculating the reliability from the product of failure rate and path execution rate was explored in very broad terms (Shooman 1976).

Analyzing a program's domains based on the paths found in both the code and the specification was called Partition Analysis in (Richardson and Clarke 1985). What we call program domains are very similar to their *procedure domains*, except they did not treat $\langle fault \rangle$ the same way. It appears that they used techniques somewhat similar to those in §3.4.2-§3.4.5 to analyse their domains. Because they did not treat $\langle fault \rangle$ as we do, their sub-domains did not have the property of functional continuity, so sampling would not have been statistically valid, but they did observe very high levels of detection of errors in their tests. The other difference is our introduction of a method to deal with the infinite number of paths/domains.

Weyuker and Ostrand 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, with no indication that it could scale to real systems (Weyuker and Ostrand 1980).

Although we chose to talk about probabilistic correctness using the binary definition of Thayer et al., it may be possible to extend to a definition of closeness, similar to that done by (Weiss and Weyuker 1988). The extension would have to be very carefully done if composing component reliabilities is important because small variations in one component's output might affect the probabilistic correctness of successive components.

There are several techniques in the testing literature that are related to the path analysis described here, starting with (Goodenough and Gerhart 1975). For example, path coverage uses the set of paths in a module as a benchmark against which to measure testing completeness (Beizer 1990). Paths have also been used in program visualization. All of these approaches have had problems dealing with programs with an infinite number of paths. The key difference here is that we don't care about the most error-prone path, or the one that is least likely to be adequately tested; we only care about the ones that execute the most and therefore contribute most to the (un)reliability of the program.

We are not concerned about how well path-testing works (Duran and Ntafos 1984, Zeil and White 1981, Jasper et al. 1994, Hamlet and Taylor 1990). Nor are we attempting to assist in debugging, or characterize errors (Chen and Yu 1996a, Chen and Yu 1996b), or find domain errors (Clarke et al. 1982), or find linear errors (Zeil et al. 1992), or any of the other things for which people have used path and domain testing (Woodward et al. 1980, Goldberg et al. 1994). We are simply trying to determine the probabilistic correctness of classes of code.

3.9 Summary

The three contributions of this chapter are:

1. the use of fault blocks as the constituent element of paths and the consequent continuity of the associated program domains;
2. the addition of sampling to the available approaches to classifying the probabilistic correctness of domains; and
3. the algorithm to generate the unbounded sequence of the program domains in decreasing relevance to the program's probabilistic correctness.

$\boxed{4}$ Probability Density Functions (PDFs) in Program Analysis

In Chapter 3, Algorithm 2 showed how to generate paths and domains in most-significant-first order. This required an approximate measure of the proportion of the input space covered by each domain, based on statistical information about the input parameters. We also alluded to Chapter 5, where we will need to have a fairly accurate measure of the same proportion in order to calculate component reliability.

Therefore, to accurately determine program reliability, we need to have a fair measure of the coverage of each domain. To determine the coverage of domains 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, 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` to include sub-ranges. In some circumstances, knowledge about types of variables can rule out certain failures, but for other kinds of failures it says nothing about their likelihood. Eggert carried this to its logical conclusion and extended the intervals in the types of his language which allowed him to catch almost all of `Pascal`'s run-time errors at compile time (Eggert 1980).

To increase the accuracy of characterization of variable values we could use a symbolic (King 1976) or abstract (Cousot and Cousot 1992, Shivers 1991) interpretation of the program to determine the set or range of values that a variable can contain. A set of values 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, e.g. synthetic variables such as loop controls, the profile can be determined via an abstract interpretation of the program. For 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 we use them to model variables and expressions in a programming language.

4.1 PDFs versus Histograms

Virtually all previous work with operational profiles (Lyu 1996a, Mason and Woit 2000, Musa 1998, Hamlet et al. 2001) has used histograms to represent the input density for the operational profile. There are two fundamental problems with the use of histograms. The first is that the boundaries reflected in the histogram may have little or no relationship to the way the program works because they are generated by monitoring the environment or by an expert's calculation, in either case, without any attention to the program and hence independent of code domains. Therefore testing based on it may seriously misrepresent the actual reliability of the program. For example, assume a program with an integer input in the range $[1 \cdots 100]$. Further assume that 98% of the time the input is in the range $[1 \cdots 5]$, 1% of the time it is in $[6 \cdots 10]$, and 1% of the time it is in $[11 \cdots 100]$. Of course, if we know that much, we should say so in the operational profile, but the person doing the sampling might create histogram A with $[1 \cdots 10]$ for 99% and $[11 \cdots 100]$ for 1%, or histogram B with $[1 \cdots 5]$ for 98% and $[6 \cdots 100]$ for 2%, both of which are correct. If we have a failure when the input is 7, we would get different results for histograms A and B. For histogram A, 1/10 of the first range would fail, giving an overall reliability of 90.1%. For histogram B, 1/95 of the second range would fail, giving an overall reliability of 98.02%. The true reliability, with the real operational profile, is 99.2%.

The second fundamental problem is that operations (like addition and multiplication) on histograms do not produce histograms as a result. Therefore this kind of simple calculation cannot work if frequency information or operational profiles are going to flow from one component to the next, as we will want to do later and as is done in (Hamlet et al. 2001).[11]

As an example of this problem, assume we have two input parameters. To make this as simple as possible, we will assume that they have uniform distribution, $X = [1 \cdots 2]$ and $Y = [3 \cdots 6]$, both shown in Figure 4.1.

Figure 4.2 shows the sum of the two variables, and Figure 4.3 shows the product. As can be clearly seen, the nice uniform histogram shape has changed significantly, even after a single operation.

11 That paper works around the problem with the correct, but computationally intractable approach of enumerating the input values to determine which output bucket they would fall into.

Figure 4.1 X (darker) and Y with Continuous Uniform Distributions

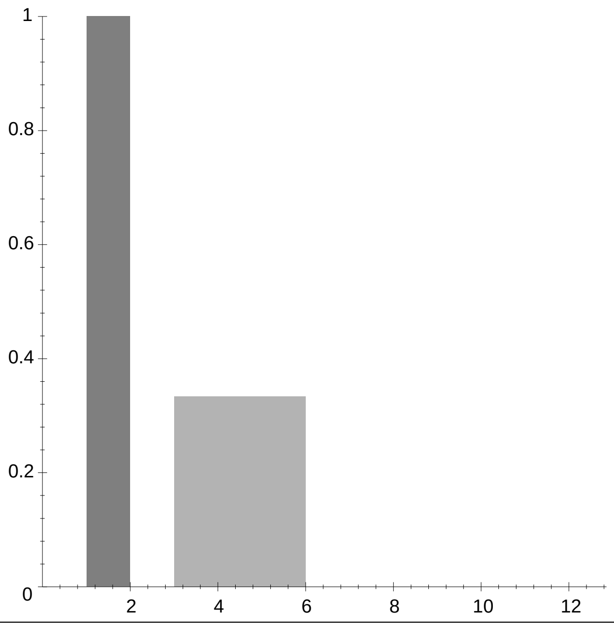


Figure 4.2 X plus Y with Continuous Distributions

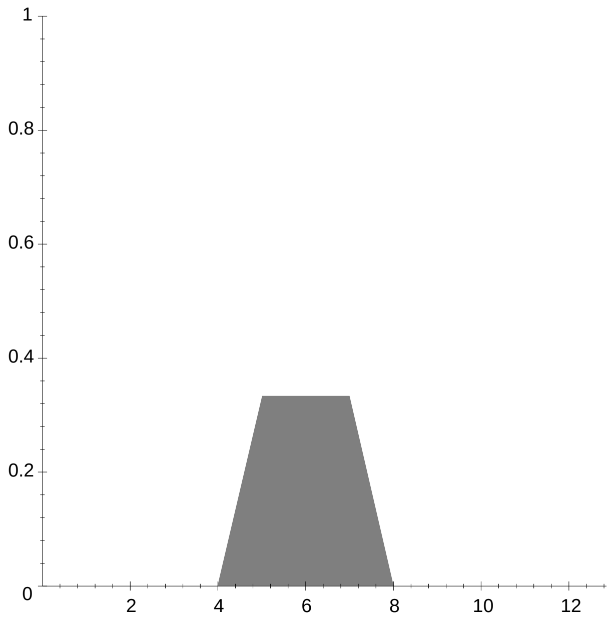
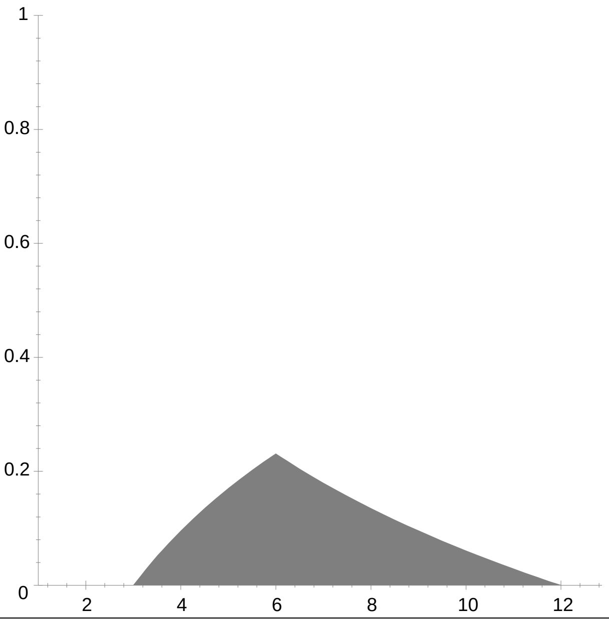


Figure 4.3 X times Y with Continuous Distributions



4.2 Introduction to Probability and PDFs

If we want to track the distribution of values through a program, we need something that is more accurate than histogram approximations, and more efficient than enumerating all possible inputs, where enumeration is even possible. The most natural mathematical construct to use for this is a Probability Density Function (PDF).

This introduction to probabilities will be fairly brief and restricted to that necessary for an understanding of the balance of the chapter. The interested reader is referred to (DeGroot 1989, Leemis 1995, Lyu 1996b).

In general, probabilities are expressed as the likelihood of a set of events A, a subset of the total event space S, occurring:

$$0 \leq \Pr(A) \leq 1, \tag{4.1}$$

where

$$\Pr(S) = 1, \tag{4.2}$$

and

$$\Pr(\overline{A}) = \Pr(S \setminus A) = 1 - \Pr(A). \tag{4.3}$$

If we have a random variable X, expressed as a real number,[12] we can talk about the probability that the random variable will have a value less than some value z as:

$$\Pr(X < z), \tag{4.4}$$

which we could express as a function of z, identified with the random variable, X:

$$P_{\mathtt{X}}(z). \tag{4.5}$$

This function is called the distribution function (DF). Because the DF is the sum of all the probabilities less than z, where it is continuous and differentiable, the derivative of the DF with respect to z is the Probability Density Function (PDF),

$$p_{\mathtt{X}}(z), \tag{4.6}$$

which is the point-wise probability that the random variable X has the value z.

Every **Probability Density Function (PDF)** is a non-negative function with an integral of 1. That is, f is a PDF iff:

$$f(x) \geq 0, \forall x \in dom(f), \text{ and} \tag{4.7}$$

$$\int_{dom(f)} f(x)\,dx = 1. \tag{4.8}$$

The value of the function $f(x)$ is the probability that if a random value in the $dom(f)$ is chosen, 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:

$$\int_a^b f(x)\,dx. \tag{4.9}$$

Probability Density Functions (PDFs) can be discrete or continuous. We will postpone consideration of discrete PDFs until §4.6.2.

Throughout this chapter PDFs will be designated as a function $p_{\mathtt{x}}(y)$ where the subscript will take two main forms:

- $p_{\mathtt{a,b,\cdots}}(x, y, \cdots)$ describes the joint probability for the variables a, b, $\cdots$ evaluated at the points x, y,$\cdots$. The definition of independence is the equivalence,

$$p_{\mathtt{a,b,\cdots}}(x, y, \cdots) = p_{\mathtt{a}}(x) p_{\mathtt{b}}(y) \cdots \tag{4.10}$$

- $p_{\mathtt{f(a,b)}}(z)$ is the function that maps a z value into the probability of that value occurring, given the function "`f(a,b)`" and the density function for $p_{\mathtt{a,b}}(x, y)$, as described in §4.5.1.

12 This could be applied to any totally-ordered set, but for our purposes, the real numbers provide a suitable example.

4.3 PDFs of Program Variables and Operations

In the abstract evaluation behind Algorithm 2 the program values that are stored in variables or that are the results of expressions fall into three categories: (1) conventional program values, (2) parameters, or (3) symbolic expressions of parameters.

Standard semantics are used in the evaluation of all operations and conditionals that contain only conventional values. All operations on conventional values return conventional values as their result.

Abstract semantics are used in the evaluation of all program operations and conditionals involving parameters or symbolic expressions of parameters. The abstract semantics for operations produce symbolic expressions as their result. The abstract semantics for conditionals will split off new paths.

PDFs and Operational Profiles

A program under analysis will be evaluated in the context of an operational profile defined by a joint PDF $p_{\mathtt{r_1},\mathtt{r_2},\cdots,\mathtt{r_n}}(x_1,\cdots,x_n)$ of a set of n random variables corresponding to the input parameters of the program. Typically these will be independent random variables, perhaps one variable describing the length of widgets and another describing temperatures. Sometimes the joint PDF will be more complex because the variables are inter-related in some way, such as the height and weight of people. All of the operations described in the next two sections can be thought of as operating on an n-dimensional space (typically $\mathcal{R}^n$), where n is the number of random input variables. We will usually refer to the complete PDF for the operational profile even if some of the random variables are unused. This will allow the number of integrals to be constant throughout the discussion.

The only requirement for the PDF of the operational profile is that the PDF be integrable in practice. In the common case where the random variables are independent, the PDF will be the product of the individual PDFs. See §4.6.1 for more discussion.

4.4 PDFs and Predicates

The primary need for probabilities in Chapter 3 and Chapter 5 is to determine the weighting of an operational profile that is covered by a domain. For Chapter 3 the accuracy requirements are modest because all that the weighting determines is the exact order in which domains are generated, and a slight permutation of that order will have little impact on the algorithm. For Chapter 5 the accuracy requirement is more stringent because it directly affects the accuracy of the probabilistic correctness calculated for a program.

A domain is defined by a predicate which is the conjugation of all of the conditional tests performed at some point in the evaluation of a path. For example, consider the program fragment in Figure 4.4 and the path resulting from going through the loop three times and getting to the point X in the program. The complete predicate would be

Figure 4.4 Program fragment generating a path.

```
procedure ex(float x,float y)
   float z=1
   float w=x+y
   while (z<10 and y*y>=z)
      w:=w-0.5
      z:=z+1
      if (z-1>x*x-y)
         ...
         if (w<0)
            ...
            X
```

$$y^2 \geq 1 \wedge y^2 \geq 2 \wedge y^2 \geq 3 \wedge y^2 < 4 \wedge 3 > x^2 - y \wedge x + y - 1.5 < 0. \tag{4.11}$$

The first two terms are included in the third term, so this is equivalent to $y^2 \geq 3 \wedge y^2 < 4 \wedge 3 > x^2 - y \wedge x + y - 1.5 < 0$.

To calculate the weighting of an operational profile over a sub-domain, we need to know the volume under the PDF that describes the operational profile of the region of that sub-domain. That volume is the integral of the operational profile over the sub-domain, for which the general form is,

$$\int \underset{D_i}{\cdots} \int p_{\mathtt{OP}}(x_1, \cdots, x_n) dx_1 \cdots dx_n \tag{4.12}$$

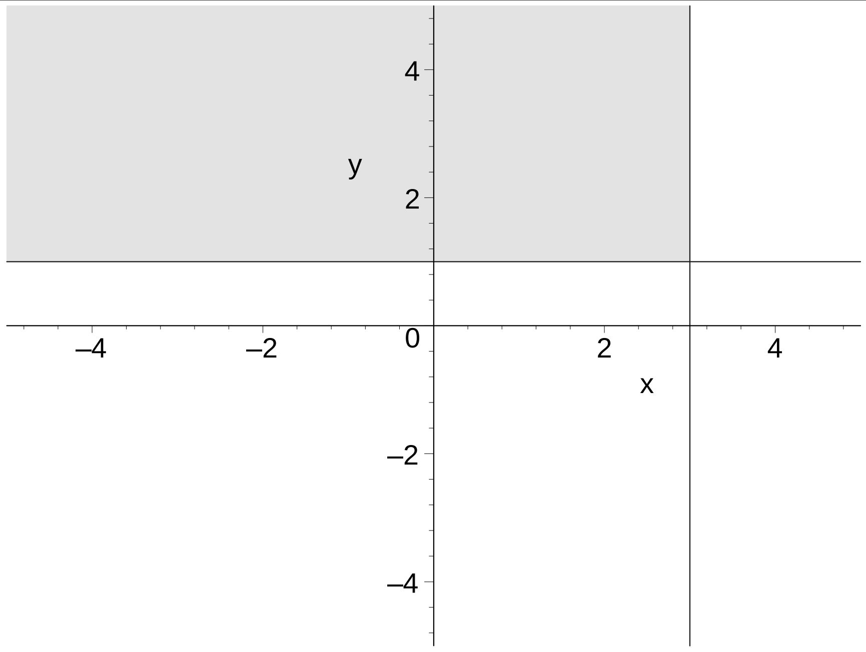
where D_i is the domain of interest and $p_{\mathtt{OP}}(\cdots)$ is the PDF that describes the operational profile. A path domain is specified as a conjunctive predicate; a series of relations anded together, for example $x < 3 \wedge y > 1$. The coverage for the domain is calculated as the integral of the operational profile over the domain specified by the predicate; in this case,

$$\int_{-\infty}^{3} \int_{1}^{\infty} p_{\mathtt{r}_1,\mathtt{r}_2}(x, y) dy\, dx. \tag{4.13}$$

The best way to think of the domain is as a geometric region in n-space. This example is shown in Figure 4.5, with the visible part of the domain shaded.

A predicate can have many terms and they will rarely be so convenient as to be parallel to one of the axes. In fact, a domain may not even be connected. In almost all cases the integral must be calculated in pieces, with the integrals of the pieces being subsequently summed.

Figure 4.5 The region defined by the domain $x < 3 \wedge y > 1$.



Consider a predicate $y^2 < 4 \wedge 3 > x^2 - y \wedge x + y - 1.5 < 0$, simplified from equation 4.11, composed of three terms:

1. $y^2 < 4$ or $-\sqrt{4} < y < \sqrt{4}$ means the region between $y = -2$ and $y = 2$;
2. $3 > x^2 - y$ or $y > x^2 - 3$ means the region above the parabola $y = x^2 - 3$;
3. $x + y - 1.5 < 0$ or $y < 1.5 - x$ means the region below the line $y = 1.5 - x$.

The graphical interpretation of this is shown in Figure 4.6. Here the area of interest is shaded, and each of the separately integrated pieces demarcated by a dotted line. The coverage in this case would be:

$$\int_{-\sqrt{5}}^{-1} \int_{x^2-3}^{2} p_{\mathtt{r_1},\mathtt{r_2}}(x,y)dy\,dx \tag{4.14}$$

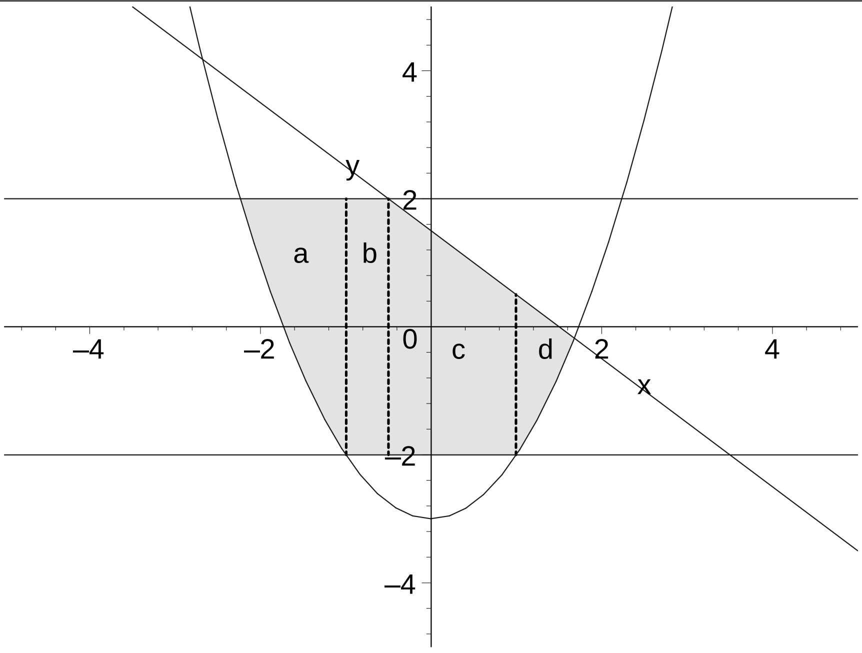
$$+ \int_{-1}^{-0.5} \int_{-2}^{2} p_{\mathtt{r_1},\mathtt{r_2}}(x,y)dy\,dx \tag{4.15}$$

$$+ \int_{-0.5}^{1} \int_{-2}^{1.5-x} p_{\mathtt{r_1},\mathtt{r_2}}(x,y)dy\,dx \tag{4.16}$$

$$+ \int_{1}^{1.67945} \int_{x^2-3}^{1.5-x} p_{\mathtt{r_1},\mathtt{r_2}}(x,y)dy\,dx. \tag{4.17}$$

where each limit on the integrals for x come from the intersection of two of the predicate terms, and the limits on the integrals for y come from the equations corresponding to

Figure 4.6 The region defined by the domain $y^2 < 4 \wedge 3 > x^2 - y \wedge x + y < 1.5$.



the predicates. The labeled integration regions in Figure 4.6 correspond to the parts of the integral as follows: a is equation 4.14; b is equation 4.15; c is equation 4.16; and d is equation 4.17.

This example is very convenient because the domain is connected and convex, so we could partition it into strips along the x axis[13] but, in theory, similar techniques are applicable regardless of the shape of the domain. In practice we are not so lucky. §4.4.1 will explore this issue.

As Algorithm 2 runs, every time it comes to a conditional node it must evaluate the `node.filter` method which must calculate the `weight` for the new domains. The `weight` is the integral shown in equation 4.12. The time taken to perform these calculations will obviously dominate the execution time of the algorithm. The accuracy of the weighting is not critical because the only result of an error is that the resulting paths may be processed somewhat earlier or later than they otherwise would. There is no danger of the path getting lost. We will discuss possible exploitation of this in §4.5.1.

13 If we hadn't removed the first $y^2 \geq 3$ term, the picture would have had a swath cut out of the middle, from $x = -\sqrt{3}$ to $x = \sqrt{3}$, and there would have been six regions instead of four.

Similarly, all of the time spent calculating equation 5.1 is in performing the same integral over all the domains. However, in this case accuracy is essential.

4.4.1 Analytical Integral of PDFs

To find the integral of the region of the PDF defined by a predicate, we must know the points of intersection of the curves, surfaces, or hyper-surfaces, represented by the terms of the predicate, and we must know the equations of the boundary functions. For example, in equation 4.17, the lower bound for the integration over x was determined by the intersection of the curves $y = -2$ and $y = x^2 - 3$, and the upper bound was determined by the intersection of the curves $y = x^2 - 3$ and $y = 1.5 - x$. In this case the bounds for integration over y were simply the two functions of x. In higher dimensions they could be functions of several variables.

Generation of the points of intersection happens in three steps:

1. Initialize the set of equations with the terms from the predicate.
2. Solve every pair of equations as a system of two equations. The result is one of:
    (a) an inconsistency - ignore and proceed
    (b) one or more points - add to the set of intersection points
    (c) one or more equations - add to the set of equations

    Repeat until a fix-point is reached (no new points or equations).
3. For each point, for each term in the predicate, discard the point if it is on the wrong side of the curve or surface defined by the term.

Before going any further we must look at that approach to generating intersection points. Unfortunately the approach requires "solving" a system of equations for which there is no general solution. The limits of standard mathematical analytic methods force the following restrictions:

1. There is no limit on terms containing a single variable raised to any power, e.g. $x^7 > 3$.
2. Polynomials of one variable are limited to degree four, except for special cases.
3. Polynomials of two variables are generally limited to degree two.
4. Polynomials of three or more variables are limited to degree one.

Fortunately these restrictions are not as serious as they might first appear. The limits are on expressions of program parameters, so internal program variables do not affect the analysis. These restrictions have the side benefit that the boundary functions for the integrals will not be difficult to determine.

The number of integrable regions is not easily calculated but, given the restrictions above, the number of terms in the predicate after removing duplicates ($x < 3 \wedge x < 4$ becoming $x < 3$) is a rough lower estimate.

4.4.2 Numerical Integral of PDFs

In principal, numerical integration can remove some of the restrictions of §4.4.1 because we are no longer limited to analytic techniques to find roots or boundary curves. Unfortunately, in practice, the operations we want to perform on higher-degree polynomials in several dimensions tend to have stability problems, so there are limits on how much might be gained from this approach. Furthermore the accuracy available from numeric techniques may not be sufficient.

4.4.3 Monte Carlo Integral of PDFs

Monte Carlo integration is done by choosing a finite bounding box, the sample space, that encloses the target region for which the integration is required. The edges of the sample space are usually chosen to be parallel to the axes, so the volume of the sample space is easy to calculate. Then points are randomly generated within that sample space. Each point that is generated is checked to see if it is in the target region. The proportion of the random points that are within the target region, multiplied by the area of the sample space, gives the area of the target region. Any degree of accuracy can be provided by generating sufficient random points.

Monte Carlo integration can be extended naturally to higher-dimensional spaces, with significant growth in the number of required points for a given degree of accuracy. A simple experiment to take the integral of the n-dimensional unit-sphere using forty million randomly generated points resulted in as many as 12 digits of precision in 6-space, but only 6 digits in 13-space and 1 digit in 15-space. The precision has more to do with the tightness of the bound of the sample space than with the number of dimensions, in 15-space only one point in 100,000 points generated in the sample space falls within the target region, even though the sample space is the smallest rectangular region that encloses the unit sphere.

When generating the random points to check against a conjunct, the points can be generated in a subspace of only the relevant random variables. Monte Carlo integration is a very general-purpose technique that is always applicable as long as a bound is available in every dimension. There are two issues determining the usefulness of Monte Carlo integration, for our purposes.

Bounds

The conjuncts for the domains do not always provide bounds, e.g. our earlier example if the parabola were removed. However if the random variables involved in the conjunct have bounded range where the PDF is non-zero, those bounds further limit the sample space. Therefore if every variable in the conjunct has upper and lower bounds, either because of a bounded range or from the conjunct, then we have a sample space and Monte Carlo integration is applicable.

Accuracy

The accuracy of Monte Carlo integration is tunable by adjusting the number of samples. Therefore where accuracy is not paramount (e.g. Algorithm 2), Monte Carlo integration may provide a rough estimate for the domain weight. There are additional tricks that could be applied to Algorithm 2, such as lazy sampling to refine priorities in the priority queue only as required. Alternatively, where we require very high accuracy (e.g. equation 5.1), a much larger number of samples can be used to achieve the desired accuracy, but performance may be very poor.

4.4.4 Combined Techniques for Integration of PDFs

A combination of approaches could also be used, with analytical techniques applied where appropriate, and Monte Carlo techniques used as a fall-back. This would give highly accurate answers in bounded time for many predicates, but a guaranteed answer for any PDF with finite ranged random variables.

It might also be appropriate to use Monte Carlo techniques for Algorithm 2, where accuracy is less critical, and the combined approach of the previous paragraph for equation 5.1, where accuracy is essential.

4.5 Alternative Predicate Weighting

There is another approach which does not give the correct result when different terms in a predicate use the same variable, or dependent variables, but can be calculated more quickly and might give weights that are "good enough" for Algorithm 2. This approach involves calculating the weight for each term of the predicate separately, and multiplying the separate weights together. To calculate the weight we want a function that captures the PDF of the operational profile and allows us to easily calculate the integral. For example, if we had a predicate term like $3 < b - c$, we would like to be able to calculate the integral,

$$\int_{3}^{\infty} p_{\mathtt{b-c}}(z)dz. \tag{4.18}$$

4.5.1 PDFs for Function Application

In many cases it is possible to determine a new PDF corresponding to a function over the variables of an existing PDF.

Assume we have a function of n random variables, $f(x_1, \cdots, x_n)$. Using the same logic as in §4.2, what we want is:

$$\Pr(f(x_1, \cdots, x_n) \leq z), \tag{4.19}$$

which is:

$$P_{\mathtt{f(x_1,\cdots,x_n)}}(z). \tag{4.20}$$

Unfortunately there is no general expression for this, so we must cast about for a way to write it in terms of things we know. Let us assume there is an inverse for the function, f. Without loss of generality, assume it is invertible over the n^{th} variable:

$$g(x_1, \cdots, f(x_1, \cdots, x)) = x \tag{4.21}$$

Because the information we have about probabilities of the individual variables, x_1, x_2, $\cdots$, x_n is in the form of probability density functions, the DF expands to:

$$P_{\mathtt{f(x_1,\cdots,x_n)}}(z) = \int_{-\infty}^{\infty} \cdots \int_{x_n=-\infty}^{g(\cdots,z)} p_{\mathtt{r_1,\cdots,r_n}}(x_1, \cdots, x_n)dx_n dx_1 dx_2 \cdots dx_{n-1} \tag{4.22}$$

Because we want this expressed as a PDF, we take the derivative:

$$\begin{aligned} p_{\mathtt{f(x_1,\cdots,x_n)}}(z) &= \frac{d(P_{\mathtt{f(x_1,\cdots,x_n)}}(z))}{dz} & (4.23) \\ &= \int_{-\infty}^{\infty} \cdots p_{\mathtt{r_1,\cdots,r_n}}(x_1, \cdots, g(\cdots, z)) \frac{d(g(\cdots, z))}{dz} dx_1 \cdots dx_{n-1} & (4.24) \end{aligned}$$

4.5.2 Incremental Calculation of Weights

We can use the function derivation of §4.5.1 to calculate an approximation to the correct weight for a predicate. Consider the predicate $a < 4 \wedge 3 < b - c \wedge b + a < 1.5$, and assume we have the operational profile $p_{\mathsf{a,b,c}}$. The correct weights for the three terms are:

$$w_1 = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{4} p_{\mathsf{a,b,c}}(x, y, z) dx dy dz \qquad (4.25)$$

$$w_2 = \int_{3}^{\infty} p_{\mathsf{b-c}}(z) dz \qquad (4.26)$$

$$w_3 = \int_{-\infty}^{1.5} p_{\mathsf{b+a}}(w) dw, \qquad (4.27)$$

but $w_1 \times w_2 \times w_3$ is *not* correct for the complete predicate. Note that $w_1 \times w_2$ *is* the correct weight for $a < 4 \wedge 3 < b - c$, because there are no dependencies between the terms (assuming a, b and c are independent in the operational profile).

There are two speed advantages for this approach, compared to the correct approach.

1. Because of the derivative and integral properties of uniform and histogram PDFs described in §4.6.1, closed form expressions of the integral of the PDF for each predicate term can be derived, for uniform random variables. This will allow calculation of the weight directly.

2. Because the predicates are cumulative, i.e. each new conditional encountered narrows the domain, the coverage of each term can be derived as it is encountered, and simply multiplied by the coverage of the path to this point. Thus each predicate term need only be calculated once.[14]

Unfortunately there are three problems with this approach:

1. Predicates calculated this way will likely be wrong if there are two or more terms, because the terms are calculated independently, so inter-dependencies will not be recognized. The calculation can over-estimate or under-estimate the correct coverage. However the domain splitting of Algorithm 2 does not require perfect accuracy, so this calculation may produce answers that are accurate enough to produce domains in close to proper order. Any remaining error can be ameliorated by periodically running through all the pending domains and re-calculating their coverage using the correct algorithm.

2. The formula for a predicate term may not be analytically invertible, in which case this approach is not applicable at all.

14 All current predicates may have be be re-calculated periodically if a revised operational profile becomes available, as described in §5.3.1.

3. The PDFs for the random variables required may not be analytically integrable (i.e. closed form), in which case this approach is not applicable. As pointed out in §4.6.1, uniform or histogram based random variables are integrable, but not all standard distributions, in particular the normal or Gaussian distribution, have analytical integrals.

4.6 Cost of Calculating with PDFs

This chapter has described how to find the coverage of a PDF over a domain described by a predicate. This section will describe some of the cost implications.

4.6.1 Histogram PDFs

The uniform and histogram PDFs that are the base values for our software reliability determination have some properties that are very convenient. Firstly, they have anti-derivatives to arbitrary level, and all the anti-derivatives are continuous and piece-wise differentiable for all values. Secondly, the PDFs are piece-wise differentiable over their non-zero range, and the derivative is zero. The effect of these properties is that closed form expressions can be derived for the integrals above and in equation 5.1.

The only problem with the uniform PDFs, and most others, is that they are not continuous over all values. However the uniform PDFs, and most others, are at least piece-wise-continuous and differentiable. This means that we can apply the analytical techniques discussed in this chapter, as long as we only apply them over continuous and differentiable pieces of the domain of *every* relevant PDF.

What this means in practice is that if the n random input variables each have k_i continuous regions, then we will have to evaluate all of our integrals k_{total} times and sum the results.

$$k_{total} = \prod_{i=1}^{n} k_i \tag{4.28}$$

This means that the complexity of every integral we evaluate analytically is increased by a factor of $\hat{k}^n$, where $\hat{k}$ is the geometric mean of k for the n variables.

The value of k depends on the number of continuous regions. For a simple uniform distribution $1 < x \leq 2$, there are 3 continuous regions: $[-\infty, 1]$, $(1, 2]$, and $(2, \infty]$. The leftmost region can be ignored, because its integral is 0. Thus the distribution $1 < x \leq 2$ has $k = 2$. If a histogram has five non-zero continuous regions then $k = 6$.

There are some continuous PDFs for which k will be one, but they are likely to be rare in practice.

$\boxed{4.6.2}$ Performance of Discrete Random Variables

To this point we have exclusively used continuous random variables. The main reason is that they are amenable to analytical techniques and therefore can be calculated in fairly reasonable time.

However, the world, and particularly the computer world, is also full of examples of discrete probabilities. An example of a discrete PDF is the probability that a flip of a fair coin will be heads or tails:

$$\begin{aligned} flip(\texttt{heads}) &= 0.5 \\ flip(\texttt{tails}) &= 0.5. \end{aligned} \tag{4.29}$$

In standard statistical usage, discrete PDFs are called probability functions (PFs). The important difference is that in a PF a single value can be significant:

$$\sum_a^a f(x) = f(a) \tag{4.30}$$

and may be non-zero, whereas for PDF:

$$\int_a^a g(x)\,dx = 0, \tag{4.31}$$

regardless of the value of $g(a)$. Note that g need not be a fully continuous function, but it must be at least left-continuous or right-continuous at every point in its domain. To simplify the presentation, we have also used the term PDF for PFs. There are also mixed discrete/continuous functions such as:

$$f(x) = \begin{cases} 0.25, & \text{if } -1 < x < 0; \\ 0.5, & \text{if } x = 0; \\ 0.25, & \text{if } 1 > x > 0; \\ 0 & \text{otherwise,} \end{cases} \tag{4.32}$$

which can be handled without much additional difficulty.

To integrate a PDF containing discrete random variables requires evaluating the integral for each value at which the variable is non-zero, for each discrete random variable. In principle this is no different from dealing with multiple discontinuous regions, as discussed in §4.6.1. But in practice there will typically be *many* more values for discrete random variables, e.g. integers, than continuous regions for histograms.

While discrete distributions can be approximated by appropriate continuous distributions, the results will not be exact, in general. This inaccuracy means that continuous approximations may be useful for Algorithm 2, but will not likely be usable for equation 5.1.

One advantage of discrete distributions it that there is no analysis required or possible! Therefore there are no limitations on the degree of discrete random variables in polynomials.

4.7 Previous Work

A very early version of this chapter was presented as (Mason 2001).

(Monniaux 2001) and (Kozen 1979) are theoretical treatments of probabilistic programs, by which they mean programs with random variables. (Morgan et al. 1995) extends work on predicate transformers to include probabilities and is oriented to program proof and derivation. None of these papers are very closely related to this work, but there doesn't appear to be anything closer. (White and Cohen 1980) also uses a geometric interpretation of the input space, with more severe limitations on the programs, for purposes of evaluating testing.

5 Component Reliability Composition

Now we have all the pieces for component reliability composition and are ready to put it all together. In this chapter, we will start by using the pieces we have developed to determine an accurate estimate of component reliability. Then, starting in §5.3, we will explore the possible operations of a component economy and look at some cost/performance trade-offs.

Earlier discussion of component economies dates back to (McIlroy 1969). Our version of the component economy idea was first published as (Hamlet et al. 2001).

5.1 Static Probabilistic Properties of Components

Let us assume that we have a specification for a component and an implementation of that specification. We can set up a veritable assembly-line of processes to determine component probabilistic correctness.

The first part of the process, as described in this section, is to find the static properties of the implementation-specification pair. Although an operational profile is used to direct the generation of program domains, the resulting sub-domains are independent of the operational profile that was used to generate them. These static properties need only be determined once, and then can be used repeatedly to determine probabilistic correctness of the component with various operational profiles. The static properties are:

1. the program domains for the component, each of which includes the predicate and the function calculated by the implementation;
2. the failure rate for each program domain; and
3. the partitioning of the sub-domain into integrable regions.

These are all expensive to calculate and therefore worth calculating once and saving. If the implementation or specification changes, then these properties may have to be regenerated, but see the discussion in §5.1.5.

5.1.1 Generating Path Domains

The first step uses Algorithm 2 to generate the paths from the implementation. Because the algorithm is driven by an operational profile, we must choose a likely distribution before starting Algorithm 2 on the component. 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. The interpretation will start producing path domains that are extant in the implementation.

The process of generating path domains may take a considerable period of time, or even not terminate if there are an infinite number of paths. Fortunately it is amenable to parallelization. At any time a domain and path could be removed from the priority queue on one machine and injected into the priority queue of the same algorithm running on another machine.

At any point while this process is proceeding, if more information becomes available about a more important operational profile, it can be injected into the process. All that is necessary is to re-calculate the weight for each domain in the priority queue, based on the new operational profile. Then the process can be resumed, and it will begin processing domains based on the priorities relevant to the new operational profile.

5.1.2 Generating Program Domains

In order to produce program domains, we must take the path domains, generate related specification domains, and then take the intersection.

Depending on the nature of the specification, this may be a simple matter of looking up the potential domains in a table, or it may involve the abstract interpretation of an executable specification. In the latter case, there are two ways to generate relevant domains to be combined with the path domains.

The first, and most obvious, approach is to seed the process with the same operational profile used by the process described in §5.1.1. The drawback of this is that it may not immediately generate domains that have a non-null intersection with the path domains being produced.

A better approach is to repeatedly seed the process with a fake operational profile that is non-zero in the currently available path domain and zero everywhere else. This will force the abstract interpretation of the executable specification to generate all of the specification domains that intersect with the path domain.

As the intersections are performed to generate program domains, any piece of a domain, whether specification or path, that does not intersect with domains available from the other process must be retained until another domain is produced by the other generator that intersects with this domain.

If the program domain generation process is allowed to run to completion then the set of program domains, D, is complete, and requests for probabilistic correctness for the component can be answered through the process in §5.2.

For other components, particularly those with an infinite set of program domains, program domain generation may be suspended at some point. By saving the priority queue from Algorithm 2, and the domain fragments mentioned above, the program domain generation process can be resumed later in response to a request for a probabilistic correctness for an operational profile that emphasizes different program domains.

5.1.3 Characterizing Program Domains

Once we have produced each program domain, D_i, we must determine its failure rate, F_i. The failure rate is determined by comparing the function for the path domain corresponding to D_i, and the function for the specification domain corresponding to D_i. This comparison is done using the failure rate tests described in §3.4.1-§3.4.6.

Characterization 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 program domains can be run in parallel on a farm of machines.

5.1.4 Partitioning Program Domains for Integration

To perform the calculation of probabilistic correctness in §5.2 we must integrate over the program domain.

As seen in §4.4, the n-dimensional region delimited by the domain predicates must be partitioned into sub-regions convenient for integration. This partitioning is moderately expensive to calculate, but it is static information and can be determined for each program domain as soon as the program domain has been generated. That is, the partitioning can proceed in parallel with the program domain characterization.

5.1.5 Dealing with Implementation/Specification Change

Any change to the specification or the implementation will likely move sub-domain boundaries and make two or more of the sub-domains incorrect. There are three possible ways to approach this.

1. Throw away all of the information and regenerate it all.

2. Redo the sub-domain generation, but retain the information in a lookup table. If the domain is unchanged then the integration sub-regions can be retained. If the calculated function is also unchanged then the failure rate can also be retained. For small component changes this should require minimal re-calculation of these values. However the full cost of the abstract interpretation will be incurred.

3. The information about the fault blocks and predicates in the implementation could be maintained in the database and therefore only path domains for which a changed fault block is part of the path need be discarded and re-calculated. By clever manipulation of the operational profile with which the path generation is seeded the abstract interpretation can be directed to the affected sub-domains. If it is the specification domains that change, then the implementation-based path domains need not be regenerated and only information about program domains that involved changed specification domains would need to be discarded and re-calculated. Tracking the changes in fault blocks would be a natural feature to integrate into a development environment. Wholesale deletion or additions of code may cause significant re-calculation to take place, but in the more common instance of bug-fixes or minor feature enhancement, much less work would be required.

5.2 Calculating Probabilistic Correctness of the Component

Once a set of program domains and their static properties have been determined for the component we can calculate the probabilistic correctness for the component with a particular operational profile.

First, let us consider the case where all of the program domains have been determined. Equation 3.3 expresses the discrete probabilistic correctness of a component. With the PDFs from Chapter 4, the equation can be rewritten to be the continuous probabilistic correctness:

$$\rho = 1 - \sum_{i=1}^{|D|} F_i \int \underset{D_i}{\cdots} \int p_{\mathtt{OP}}(x_1, \cdots, x_n) dx_1 \cdots dx_n \tag{5.1}$$

Equation 5.1 takes the integral of the PDF that describes the operational profile over each domain D_i and multiplies that number by the failure rate for that domain F_i, and then sums that product over all program domains D. That gives the weighted failure rate, so the probabilistic correctness ρ will be 1 minus that sum.

Commonly we will want an estimate of a component's probabilistic correctness before all of the sub-domains have been determined and characterized. We can calculate an interim probabilistic correctness value $\rho_{interim}$, which is the same as ρ except using D', the set of all the characterized program domains, rather than using D, the potentially infinite set of all program domains.

$$\rho_{interim} = 1 - \sum_{i=1}^{|D'|} F_i \int \underset{D_i}{\cdots} \int p_{\texttt{OP}}(x_1, \cdots, x_n) dx_1 \cdots dx_n \tag{5.2}$$

A conservative approximation to the probabilistic correctness 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 estimated system reliability will only become better as the residual sub-domains are characterized.

The coverage of the operational profile by the currently available domains D' is:

$$cov = \sum_{i=1}^{|D'|} \int \underset{D_i}{\cdots} \int p_{\texttt{OP}}(x_1, \cdots, x_n) dx_1 \cdots dx_n \tag{5.3}$$

The most conservative estimate will assume that all of the domains that cover the currently uncovered inputs will fail in all cases:

$$\rho_{conservative} = \rho_{interim} - (1 - cov). \tag{5.4}$$

$\rho_{interim}$ could itself be used as a possible estimate. That which would assume that all of the undiscovered domains are perfect, which is almost certainly unrealistic but it provides a useful upper bound for the true probabilistic correctness:

$$\rho_{conservative} \leq \rho \leq \rho_{interim}. \tag{5.5}$$

If the value of $\rho_{interim}$ is below the required reliability then the customer can give up on this component without further calculation.

Another possible estimate would result from assuming that all of the undiscovered domains have the same probabilistic correctness as the ones that have been characterized. This is probably unrealistic because these are, by definition, less common sub-domains and hence will probably not have been tested as well by the programmers before release:

$$\rho_{optimistic} = \rho_{interim}/cov. \tag{5.6}$$

A last possible estimate would result from assuming that all of the undiscovered domains have a probabilistic correctness that is based on previous experience with this supplier. This could suffer from the same problem as the previous estimate if it was a single number, but as a function of the current coverage it may have some value:

$$\rho_{historic} = \rho_{interim}/f(cov). \tag{5.7}$$

5.2.1 Threats to Accuracy

The probabilistic correctness ρ is an accurate estimation of the probabilistic correctness of the component with the specified operational profile. The accuracy is limited only by that of the operational profile and the failure rate characterization of the domains. Because of the continuity properties of the path domains, and consequently the program domains, the failure rate characterization can be made arbitrarily accurate in most cases. Path domains that are characterized by the analytical techniques, §3.4.1-§3.4.5, should be completely accurate, and path domains that are characterized by sampling, §3.4.6, can achieve any desired confidence level by acquiring sufficient samples. It may take arbitrarily long to gain the desired accuracy, especially for components with infinite domains and very flat distributions, because $\rho_{conservative}$ may very slowly approach the desired probabilistic correctness. Because the program domains can be characterized to arbitrary confidence levels, the primary limitation on the accuracy will be the quality of the operational profile.

This requirement for an extremely accurate operational profile may not be reasonable. However, if extremely accurate operational profiles are not available, it is hard to imagine how one could possible hope to calculate system reliability using *any* mechanism. Given an accurate operational profile for the system, the operational profile for individual components can be calculated with high accuracy when the techniques of this dissertation can be applied.

5.2.2 Cost of the Calculation

All of these calculations can be done automatically using the static properties for program domains determined in §5.1 and an operational profile. Automatically does not necessarily mean quickly. The number of domains is unbounded for some applications and calculating the integrals in equation 5.2 and equation 5.3 are *potentially* exponential in the number of random variables (i.e. parameters to the component) and the number of terms in the predicate for each domain.

5.3 Possible Operation of a Component Economy

There are distinct roles for the component supplier and the component consumer, who we'll call the System Designer. Both terms need not be construed to refer to single people; they may be whole companies.

All of the analysis described to this point in this dissertation could be done automatically. The only manual tasks are those described in this section: writing component and glue code, and defining an appropriate and accurate operational profile.

5.3.1 Role of the Component Supplier

Naturally the first job of the component supplier is to receive the specification and determine if they have a component on the shelf that might meet the specification.[15] If not they must write it and start the processes described in §5.1.

Eventually a system designer will want to know the probabilistic correctness in a particular context and will provide the supplier with a specific input distribution and a request for the probabilistic correctness of the component for that distribution. If the input sub-domains which have already been characterized cover the non-zero parts of the specified distribution then a probabilistic correctness number can be calculated and returned to the system designer. Otherwise, the provided input distribution can be substituted into the characterization process which will direct the characterization process toward sub-domains most relevant to the customer needs.[16]

If the characterization process has completed then the supplier can calculate ρ, using equation 5.1, and provide that to the system designer.

If the process has not completed then the supplier can use the other formulas from §5.2, and provide the results to the system designer. The system designer would likely want at least $\rho_{interim}$ and cov, from which they can calculate the other quantities.

5.3.2 Role of the System Designer

The system designer has five 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, and determining the basic control structure.

2. Specify the components. The specifications must be formal so that they can be used to determine program domains as described in Chapter 3.

15 They might instead create a component speculatively, and wait for a system designer to request it.

16 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.

3. Determine the input distribution for the system. The more accurately this can be determined, the more accurately the estimated system reliability can be reported.

4. Implement the skeletal system. This will be the glue code that connects the components. This is required to be able to extract from the system operational profile the specific operational profile that will be presented to each component. This is very similar to the component characterization process in §5.1, in that the program will be executed with PDFs representing the system input, and the parameters to all components will be collected. Ideally, perfect implementations of the components, such as executable specifications, will be available so that the results of each component can be used as the values provided to subsequent components (however, see §5.3.3).

5. Execute a variant of Algorithm 2 where the generated paths are sequences of component invocations. Calculate the probabilistic correctness for each path as the product of the probabilistic correctness of the components on the path. Weight the path probabilistic correctnesses by the weight of each path according to the system operational profile, using equation 4.12. Sum those weighted path probabilistic correctnesses to obtain the overall system probabilistic correctness for that operational profile. Unfortunately, unlike in the component case, there are no static properties, so all of this work must be done for any new operational profile.

   As in the component analysis, there is the potential for an infinite number of paths. It should probably be considered a good design rule that the system designer should avoid loops and nested conditionals wherever possible.

Note that the third and fourth tasks can potentially be performed in parallel with the component suppliers' tasks.

5.3.3 Blurring the Distinction: Components that Use Components

As mentioned in §5.3.2, the system-designer must build the skeletal system, or glue-logic, in order to determine the inputs that will be provided to various components. But of course, the skeletal system is a component, or set of components, itself. It will take inputs and likely make decisions based on the input as to which components to invoke. Further, results from one component will almost certainly be provided as inputs to other components, possibly after modification or selection.

There are two questions that must be resolved.

1. How should the probabilistic correctness of the glue logic be dealt with?

Glue logic is simply a sequence of components, albeit very simple components with very high probabilistic correctness. Thus it should be factored into the general calculation of system probabilistic correctness.

2. How will we get the output of a component to provide as input to another component?

   The first alternative is to use a form of executable specification for the component in place of the component in the system. The executable specification would have to work with PDFs in order to produce the correct operational profile for the following components.

   Alternatively, the calculated results from Algorithm 2 could be used with a dispatch table based on the corresponding sub-domain.

5.4 Long-Running Systems

Long running systems can be thought of as a component with very long paths, where the paths are made up of thousands or millions of component invocations. Therefore the issues from the two previous sections are very applicable. Firstly, because the overall probabilistic correctness of the path will be the product of the probabilistic correctness of all the components, achieving reasonable reliability estimates will be exceedingly difficult. For example, even if *every* component on a path of one million elements had a probabilistic correctness of 0.99999999, the overall probabilistic correctness will only be 0.99 — likely unacceptably low.

Secondly, even the slightest error in the operational profile of the program, or the output of each component feeding into another, will produce disastrous reliability estimates. For example, if only 0.1% of those million elements had the estimate of their probabilistic correctness reduced to 0.999 because of inaccuracies in their operational profiles, the overall probabilistic correctness will drop to 0.364 — i.e. the system will fail almost two times out of three!

5.5 Summary

In this chapter we have shown how to derive a component's static properties related to program domains and how to use a component's static properties and an operational profile to calculate the probabilistic correctness. We have suggested how component suppliers and system designers might operate in a component economy, and how a system designer can use component probabilistic correctness for their system's use of each component to establish system probabilistic correctness.

6 Conclusions and Future Work

This thesis provides a framework for the use of reliability composition for software component-based systems.

There are five significant contributions of this work:

1. the use of fault blocks as the constituent element of paths and the consequent continuity of the associated program domains;
2. the addition of sound sampling to the set of approaches to classifying the probabilistic correctness of domains;
3. the algorithm to generate an ongoing sequence of the domains in decreasing likelihood of relevance to the program's probabilistic correctness for a given operational profile;
4. the application of PDFs to program analysis;
5. a description of how a component economy could be structured to utilize these techniques to make probabilistic correctness a first-class property of components.

I believe that I have provided strong evidence to support my thesis, which was:

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

I consider this a good-news/bad-news thesis.

- The good news is that you *can* determine system reliability through composition.
- The bad news is that you almost certainly don't want to, because it's not worth what it costs.

However we *have* laid out what is required, and few of the problems are theoretical limitations, so hopefully some of the problems will be solved and the promise can be realized.

6.1 Limitations

There are several serious limitations of the current work that offer up future work.

6.1.1 Scalar Range-based Probabilities

Probably the most important limitations are those imposed on the path-generation process by the use of Scalar, Range-based Probabilities. This prevents the use of this probabilistic analysis for arrays, strings, lists, and other aggregate, position-based data structures, as well as bit pattern and other discrete interpretations of scalar values.

This is not a fundamental limitation of the analysis, but a limitation of the PDFs used here. Future work to find ways to express PDFs of those other kinds of values, and ways to probabilistically partition input domains based on those PDFs, would allow the application of the analysis to those extended domains.

It is not difficult to imagine specifying and using PDFs that expressed arrays or lists of probabilistic length where every element was a separate random variable with particular properties. It is somewhat more difficult to see how to use probabilistic information about the position of the value within the array. It appears to be impossible to automatically generate such position information, which makes probabilistic correctness for components such as searches and sorts very difficult to imagine. Moreover, there are an exponential number of paths through a sort algorithm. Fortunately, many algorithms that suffer from these and similar problems are well understood and amenable to proofs of correctness.

6.1.2 Accurate Operational Profile

Likely the second most important limitation is the difficulty of obtaining a very accurate operational profile.

If the process described in this dissertation is performed with an accurate operational profile, we can get an accurate probabilistic correctness. However, if the operational profile is not accurate, we are wasting our time.

It is possible that some sort of perturbation analysis or sensitivity analysis could at least provide a measure of how much error in the probabilistic correctness we could experience due to inaccuracies in the operational profile (Bishop 2002).

6.1.3 Computational Complexity

In order to produce the important sub-domains first for an operational profile we need to perform an abstract interpretation over the program, whether system or component. The abstract interpretation needs to integrate the operational profile over the predicate at *every* conditional branch that depends on the random variables (i.e. the input parameters to the program). The exact, analytical, solution to that integration is exponential in the number of random variables in the predicate and exponential in the length of the predicate. Because every sub-domain is differentiated from other sub-domains by a different predicate, every sub-domain will require one integration to be performed. There are some approaches suggest in Chapter 4 that might allow these integrations to be speeded up somewhat. The cost to characterize each domain for its failure rate is highly variable, but several of of the techniques in §3.4 are exponential in the number of random variables to establish high reliability. The number of paths in a typical program is immense, even if limited to interesting paths. So the total cost to generate sub-domains and characterize their reliability will be quite high, but fortunately it only has to be done once. Small changes in the specification or implementation can probably be made to have limited requirement for re-calculating all the static properties of the program.

Producing the probabilistic correctness for a program is straightforward; in principle it's one formula! However that formula will require evaluating an integral of a new operational profile over all of the domains. Unfortunately, this time the integrals must be very accurate, so the earlier tricks can not be applied. Moreover, there is no general way to know which domains are the most important to the probabilistic correctness for this operational profile. So the total cost to determine the probabilistic correctness for an operational profile will be very high.

Now consider the system paths, which are made up of sequences of components being called in succession. Each of those components must have their probabilistic correctness determined separately for all the invocations of the component because even the same component won't likely be called with the same operational profile twice. Finally we multiply together all those probabilistic correctness numbers for a path's components and we get the path probabilistic correctness. One more integral of the system's operational profile to determine the likelihood of that path, and a summation, and we're done. We have calculated the system probabilistic correctness for that operational profile!

The bottom line: we can theoretically calculate the right number, but in practice, for non-trivial systems it just won't be worth it.

The calculation of the composed system reliability may be where the amount of work involved changes the process from barely usable to completely unusable. The careful process for calculating probabilistic correctness for a *component* with a stated operational profile may be useful in some contexts.

6.1.4 Functional Programming

All of the discussion in this dissertation has been about functional programming. The abstract interpretation and reliability calculations do not require this, but if you want to compose sets of components into systems, it is important that the components are independent, and this is assured by the external view of the components being that of pure, call-by-value functions.

If mutation of state is allowed within a component, the analysis in §3.5.2 must be extended to maintain the state associated with each path. This is because evaluation of portions of the paths is arbitrarily interleaved so, for example, if both the `then` and `else` branches of a conditional mutated a variable they might see the other path's mutation if separate copies of the state were not maintained for each. This has memory costs associated with it, but otherwise has very little other impact on the analysis.

The analysis can be extended to imperative programs with global variables by treating those variable as additional input/output parameters to each component. This will make the analysis more clumsy, and much more expensive, but is doable.

The analysis is value-based. That means that arbitrary records of values can be composed, although finding the correct joint PDFs will likely be challenging, but that data structures are limited to trees. This has the effect of forbidding cycles in the program data graph. This is probably a fairly serious limitation for real-world applications. It is difficult to imagine this limitation being lifted.

Theoretically the analysis can also be applied to object-oriented programming if the set of objects forms a tree-structured graph.

6.2 Open Questions

While I believe that this dissertation has provided a useful elaboration of the principles behind component-based reliability composition, it perhaps raises more questions than it answers. Some of these are analytical techniques and some are scaling questions.

1. Can lazy evaluation of Monte Carlo integration tame the time complexity of evaluating the integrals of the PDFs while retaining the desirable properties of this work?
2. Can analytical techniques be developed to make closed-form equations for discrete PDFs? Or at least can bounds be found for the errors caused by using continuous PDFs to approximate discrete PDFs?
3. Is it statistically sound to treat the floating-point numbers in a computer program as if they were real numbers? While there are large ranges of numbers where it is plausible to treat them as continuous, at the ends of their ranges they can behave very discontinuously. How does that affect probabilistic correctness estimates?

4. If no closed forms can be found for many PDFs, is it tractable to do system composition with high-order polynomial, or potentially exponential, calculations at each iteration of a loop?

5. In practice, is the restriction to predicates of low-order polynomials of the random variables a serious limitation? Can numerical methods provide sufficient accuracy and speed?

6. How can software systems employ components so that the reliability of the whole is better than that of its parts? For safety-critical systems, a design technique must be found analogous to parallel redundant subsystems in mechanical engineering. Can the domain analysis ameliorate the problems with Multi-Version Programming (MVP)? Some programs can be made to perform random redundancy checks at run time, which are sufficient to estimate a component reliability independent of profile (Blum and Kannan 1995, Wasserman and Blum 1997) and (Ammann and Knight 1988). Although the reliability of such a component is not perfect, it *is* profile independent.

7. Can the PDFs produced as transformation functions for components be used to help specify the correctness of those components?

8. Is there a way to relax the requirement for terminating failure but retain composability?

9. Quality is not the only run-time property of a component that can be characterized by the combination of program path domains and PDFs. Properties such as run-time and memory usage can be similarly determined for components and similarly composed for systems. Hamlet explored this idea using the earlier histogram approach (Hamlet 2001). This idea should be explored with the more accurate PDF-based approach.

10. Compilers constantly make space-time trade-offs based on heuristics about program frequency. By feeding them with actual probabilities they could optimize better for particular uses of a program or component.

11. Shivers discusses various flow analyses based on knowledge about the call-site of a function (Shivers 1991). The classes of analysis: 0CFA, 1CFA, etc. seem to be related to dependent and independent variables. It may be fruitful to explore the relationship with this work.

A Bibliography

Ada Joint Program Office (1983). *Reference Manual for the Ada Programming Language, ANSI/MIL–STD–1815A*. Washington, D.C.

Ammann, P. E., S. S. Brilliant, and J. C. Knight (1994, February). The effect of imperfect error detection on reliability assessment via life testing. *IEEE Transactions on Software Engineering 20*(2), 142–148.

Ammann, P. E. and J. C. Knight (1988, April). Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers 37*(4), 418–425.

Appel, A. W. (1992). *Compiling with Continuations*. Cambridge University Press.

Beizer, B. (1990). *Software Testing Techniques* (2 ed.). Van Nostrand Reinhold.

Bentley, J. P. (1999). *Reliability and Quality Engineering* (2 ed.). Don Mills, Ontario: Addison-Wesley.

Bishop, P. (2002, July). Rescaling reliability bounds for a new operational profile. See Frankl (2002).

Blum, M. and S. Kannan (1995, January). Designing programs that check their work. *Journal of the ACM 42*(1), 269–291.

Butler, R. W. and G. B. Finelli (1993, January). The infeasibility of experimental quantification of life-critical software reliability. *IEEE Transactions on Software Engineering 19*(1), 3–12.

Chen, T. Y. and Y.-T. Yu (1996a, June). On some characterisation problems of subdomain testing. In A. Strohmeier (Ed.), *Ada Europe*, Volume 1088 of *Lecture Notes in Computer Science*, pp. 147–158. Springer.

Chen, T. Y. and Y.-T. Yu (1996b, February). On the expected number of failures detected by subdomain testing and random testing. *IEEE Transactions on Software Engineering 22*(2), 109–119.

Cheung, R. (1980, March). A user-oriented software reliability model. *IEEE Transactions on Software Engineering 6*(2), 118–125.

Clarke, L. A., J. Hassell, and D. J. Richardson (1982, July). A close look at domain testing. *IEEE Transactions on Software Engineering 8*(4), 380–390.

Cousot, P. and R. Cousot (1992). Inductive definitions, semantics and abstract interpretations. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on*

Principles of programming languages, pp. 83–94. ACM Press.

DeGroot, M. (1989). *Probability and Statistics.* Don Mills, Ontario: Addison-Wesley.

Donnelly, M., B. Everett, J. Musa, and G. Wilson (1996). *Best Current Practice of SRE*, pp. 219–254. In Lyu (1996a).

Duran, J. W. and S. C. Ntafos (1984, July). An evaluation of random testing. *IEEE Transactions on Software Engineering 10*(4), 438–444.

Eggert, P. R. (1980). *Detecting Software Errors Before Execution.* Ph. D. thesis, University of California - Los Angeles.

Fan, M. (2002). Continuation-passing-style for software reliability. Master's thesis, University of Guelph.

Farr, W. (1996). *Software Reliability Modeling Survey*, Chapter 3, pp. 71–115. In Lyu (1996a).

Frankl, P. (Ed.) (2002, July). *Proceedings of the 2002 International Symposium on Software Testing and Analysis.* ACM Software Engineering Notes.

Frankl, P. G., R. G. Hamlet, B. Littlewood, and L. Stringini (1998, August). Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering 24*(8), 586–601.

Friedman, M. and J. Voas (1995). *Software Assessment.* New York: John Wiley & Sons.

Geller, M. (1978, May). Test data as an aid in proving program correctness. *Communications of the ACM 21*(5), 368–375.

Ghiya, R. and L. J. Hendren (1998). Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 121–133. ACM Press.

Goldberg, A., T.-C. Wang, and D. Zimmerman (1994, August). Applications of feasible path analysis to program testing. See Ostrand (1994), pp. 80–94.

Goodenough, J. and S. Gerhart (1975). Toward a theory of test data selection. *IEEE Transactions on Software Engineering 1*, 156–173.

Hamlet, D. (1994a, July). Are we testing for true reliability? *IEEE Software 11*(4), 21–27.

Hamlet, D. (1994b, November). Connecting test coverage to software dependability. In *Proceedings 5th International Symposium on Software Reliability Engineering*, Monterey, California, pp. 158–165. IEEE.

Hamlet, D. (1994c, November). Special testing session, new ideas: Beyond reliability. In *Proceedings 5th International Symposium on Software Reliability Engineering*, Monterey, California, pp. 140–142. IEEE.

Hamlet, D. (1999, January). Reliability of software components. Private communication.

Hamlet, D. (2001, May). Component synthesis theory: The problem of scale. In *4th ICSE Workshop on Component-Based Software Engineering (CBSE'2001)*, Toronto, Canada.

Hamlet, D. (2002, July). Continuity in software systems. See Frankl (2002).

Hamlet, D., D. Mason, and D. Woit (2001, May). Theory of software component reliability. In *Proc. 23rd International Conference on Software Engineering (ICSE'2001)*, Toronto, Canada.

Hamlet, D. and R. Taylor (1990, December). Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering 16*(12), 1402–1411.

Howden, W. E. (1976, September). Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering 2*(3), 208–215.

Howden, W. E. (1978). Algebraic program testing. *Acta Informatica 10*, 53–66.

Jacky, J. (1997). *The Way of Z: Practical Programming with Formal Methods.* New York: Cambridge University Press.

Janicki, R. (1995). Towards a formal semantics of Parnas tables. In *17th ICSE*, pp. 231–240. ACM Press.

Jasper, R., M. Brennan, K. Williamson, B. Currier, and D. Zimmerman (1994, August). Test data generation and feasible path analysis. See Ostrand (1994), pp. 95–107.

Joy, B., G. Steele, J. Gosling, and G. Bracha (2000). *The Java Language Specification, Second Edition.* Don Mills, Ontario: Addison-Wesley.

Kernighan, B. W. and D. M. Ritchie (1988). *The C Programming Language* (Second ed.). Toronto, Ontario: Prentice Hall.

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM 19*(7), 385–394.

Knight, J., A. Cass, A. Fernandez, and K. Wika (1994, August). Testing a safety-critical application. See Ostrand (1994), pp. 199.

Knight, J. and N. Leveson (1990, January). A reply to the criticisms of the Knight & Leveson experiment. *Sigplan Notices 15*(1), 24–35.

Kozen, D. (1979). Semantics of probabilistic programs. In *Proceedings of the* 20th *Symposium on Foundations of Computer Science*, Long Beach, CA, USA, pp. 101–114. IEEE Computer Society Press.

Laprie, J.-C. (1984, November). Dependability evaluation of software systems in operation. *IEEE Transactions on Software Engineering 10*(6), 701–714.

Laprie, J.-C. and K. Kanoun (1996). *Software Reliability and System Reliability*, Chapter 2, pp. 27–70. In Lyu (1996a).

Leemis, L. M. (1995). *Reliability: Probabilistic Models and Statistical Methods.* Toronto, Ontario: Prentice-Hall.

Littlewood, B. (1979, August). Software reliability model for modular program structure. *IEEE Transactions on Reliability 28*(3), 241–246.

Littlewood, B. and L. Stringini (1993, November). Validation of ultrahigh dependability for software-based systems. *Communications of the ACM 36*(11), 69–80.

Lyu, M. (Ed.) (1996a). *Handbook of Software Reliability Engineering.* New York: McGraw-Hill.

Lyu, M. (1996b). *Reliability Theory, Analytical Techniques and Basic Statistics*, pp. 747–779. In Lyu (1996a).

Lyu, M. (1998, May). Current techniques and tools for software reliability engineering. In *Proceedings QW98 (11th International Software Quality Week)*, San Francisco, CA, USA.

Mason, D. (2001, May). Probability density functions in program analysis. In *4th ICSE Workshop on Component-Based Software Engineering (CBSE'2001)*, Toronto, Canada.

Mason, D. and D. Woit (2000, February). Input domain analysis for software reliability measurement. In *The Fifth International Conference on Computer Science and Informatics*, Atlantic City, USA.

McIlroy, D. (1969). Mass produced software components. In J. M. Buxton, P. Naur, and B. Randell (Eds.), *Software Engineering Concepts and Techniques: Proceedings of the NATO Conferences*, pp. 88–98. Petrocelli/Charter.

Milner, R., M. Tofte, and R. Harper (1990). *The Definition of Standard ML.* Cambridge, Mass.: MIT Press.

Monniaux, D. (2001, January). An abstract monte-carlo method for the analysis of probabilistic programs. In *Proceedings of the* 22$^{\text{th}}$ *Annual ACM Symposium on Principles of Programming Languages*, London, UK, pp. 93–101. Association for Computing Machinery.

Morgan, C., A. McIver, K. Seidel, and J. W. Sanders (1995, February). Probabilistic predicate transformers. Technical Report TR-4-95, Oxford University, U.K.

Musa, J. (1998). *Software Reliability Engineering.* New York: McGraw-Hill.

Musa, J., G. Fuoco, N. Irving, and D. Kropfl (1996). *The Operational Profile*, pp. 167–216. In Lyu (1996a).

Ostrand, T. J. (Ed.) (1994, August). *Proceedings of the 1994 International Symposium on Software Testing and Analysis.* ACM Software Engineering Notes.

Parnas, D. (1974). On a "Buzzword": Hierarchical structure. In *Proceedings of IFIP*

Congress, pp. 336–339. North-Holland Publishing Co.

Parnas, D. (1993, March). Evaluation standards for safety-critical software. *Communications ACM 33*(6), 836–48.

Patterson, D. and J. Hennessy (1994). *Computer Organization and Design: The Hardware/Software Interface*. San Francisco: Morgan Kaufman.

Petroski, H. (1985). *To Engineer is Human: The Role of Failure in Successful Design.* New York: St. Martin's Press.

Potter, B. F., J. E. Sinclair, and D. Till (1996). *An Introduction to Formal Specification and Z* (2 ed.). New York: Prentice Hall.

R4RS (1991, November). Revised^4 report on the algorithmic language Scheme.

Ramakumar, R. (1993). *Engineering Reliability: Fundamentals and Applications.* Toronto, Ontario: Prentice-Hall.

Ramamoorthy, C. V. and F. B. Bastani (1982, July). Software reliability - status and perspectives. *IEEE Transactions on Software Engineering 8*(4), 354–371.

Richardson, D. (1981, September). *A Partition Analysis Method to Demonstrate Program Reliability.* Ph. D. thesis, University of Massachusetts.

Richardson, D. J. and L. A. Clarke (1985, December). Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering 11*(12), 1477–1490.

Shivers, O. (1991, May). *Control-Flow Analysis of Higher-Order Languages.* Ph. D. thesis, Carnegie Mellon University.

Shooman, M. L. (1976). Structural models for software reliability prediction. In *2nd ICSE*, pp. 268–280. ACM Press.

Stark, G. (1996). *Software Reliability Tools*, Chapter A, pp. 729–745. In Lyu (1996a).

Thayer, T. A., M. Lipow, and E. C. Nelson (1978). *Software Reliability.* New York: North Holland.

Wasserman, H. and M. Blum (1997). Software reliability via run-time result-checking. *Journal of the ACM 44*(6), 826–849.

Weiss, S. and E. Weyuker (1988, October). An extended domain-based model of software reliability. *IEEE Transactions on Software Engineering 14*(10), 1512–1524.

Weyuker, E. and T. Ostrand (1980, May). Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering 6*(3), 236–46.

White, L. J. and E. I. Cohen (1980, May). A domain strategy for computer program testing. *IEEE Transactions on Reliability 6*(3), 247–257.

Woit, D. (1997). Assessing reliability of modular software. Technical Report TR-1997-

004, Faculty of Engineering, Ryerson University, Toronto, Ontario, Canada.

Woit, D. and D. Mason (1998a, November). Component independence for software system reliability. In *Quality Week, Europe (QWE'98)*, Brussels, Belgium.

Woit, D. and D. Mason (1998b, November). Software component independence. In *Proc. 3rd IEEE High-Assurance Systems Engineering Symposium (HASE'98)*, Washington, DC.

Woodward, M. R., D. Hedley, and M. A. Hennell (1980, May). Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering 6*(3), 278–286.

Zeil, S. J., F. H. Afifi, and L. J. White (1992, October). Detection of linear errors via domain testing. *ACM Transactions on Software Engineering Methodology 1*(4), 422–451.

Zeil, S. J. and L. J. White (1981, March). Sufficient test sets for path analysis testing strategies. In *5th ICSE*, pp. 184–194. IEEE Computer Society, Catalog No. 81CH1627-9.

B Index