

Probability Density Functions in Program Analysis

Dave Mason
Ryerson Polytechnic University
Toronto, Ontario, Canada
dmason@sarg.ryerson.ca

Abstract

The component reliability model described in [4] requires that component execution frequency, reliability, and transformations be analysed statistically based on parameters to the component. This paper describes work in progress toward that goal.

1 Introduction

In [4] we describe an approach to component reliability that produces an overall system reliability given the reliabilities of the components that make up the system, and the structure of those components in the system. That paper makes some simplifying assumptions about probability distributions of program variables. For each component in such a system, two functions are required and they are both parameterized on the arguments to the component. One, a reliability function, is partially based on the execution frequency of various parts of the component, as well as the (statistical) values of the variables. The other, a transformation function, describes the probability distributions of the outputs of the component.

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

The simplest determination of the values that a variable could have is the type of a variable. This is extended in languages like Ada[1] to include subranges. This can, in some circumstances rule out certain failures, but for others, says nothing about their likelihood.

The next level of determination of the values uses a symbolic or abstract interpretation of the program to determine the set or range of values that a variable can contain. This provides a finer grain of checking, but is essentially the same as type-based determination.

To provide the most accurate determination, we will need to

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

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

Definitions

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

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

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

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

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

PDFs can be discrete or continuous. An example of a discrete PDF is the probability that a flip of a fair coin will be heads or tails:

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

An example of a continuous PDF is the probability that a person will be a particular height, or have a particular IQ. 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:

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

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

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

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

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

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

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

2 PDFs of Program Variables and Operations

In order to accurately model program execution (for software reliability or code optimization) we need to have a good representation for the values that various program variables can hold. We will use a PDF for each variable in the program, or program fragment, under examination. For a variable `abc` the PDF will be P_{abc} , where $P_{\text{abc}}(x)$ is the probability that the value of `abc` is x , and the domain of P_{abc} will be the set of all values that `abc` could take on.

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

If variables are represented as PDFs, then the result of program expressions must also be PDFs. In the balance of this section we look at expressions of random variables, expressed as PDFs. Some of these are derived from [3]; the remainder are developed in the same style.

Literals

The PDF for any program literal (or compile-time constant)

value \mathbf{c} is the function:

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

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

Monadic Functions

The PDF for negation is:

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

This can be extended to any invertible monadic function f wherever f^{-1} is defined.

Dyadic Functions

The PDF for addition is:

$$P_{x+y}(z) = \int_{\text{dom}(P_y)} P_x(z-y)P_y(y) dy,$$

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

$$P_{f(x,y)}(z) = \int_{\text{dom}(P_y)} P_x(f^{-1}(z, y))P_y(y) dy.$$

This works fine for subtraction and division, but there is a potential problem with multiplication since $x \times y$ is not left-invertible if $y = 0$. However, this is just as applicable to any dyadic right-invertible function (such that: $f^{-1}(x, f(x, y)) = y$), so as long as a function is either left or right invertible, there is a solution. For multiplication this means that if, at least one of $0 \notin \text{dom}(P_x)$ or $0 \notin \text{dom}(P_y)$, we can produce a PDF.

Relations

The PDF for less-than is:

$$P_{x < y}(\text{true}) = \int \int \left\{ \begin{array}{ll} P_x(x)P_y(y), & \text{when } x < y \\ 0, & \text{otherwise} \end{array} \right\} dx dy,$$

$$P_{x < y}(\text{false}) = 1 - P_{x < y}(\text{true}).$$

This can be extended to any relation R as:

$$P_{xRy}(\text{true}) = \int \int \left\{ \begin{array}{ll} P_x(x)P_y(y), & \text{when } xRy \\ 0, & \text{otherwise} \end{array} \right\} dx dy,$$

$$P_{xRy}(\text{false}) = 1 - P_{xRy}(\text{true}).$$

3 Program Flow and PDFs

Program flow affects PDFs because a variable can be defined on multiple pathways. The analysis is essentially that of Static Single Assignment (SSA)[2].

Sequence

In a program sequence, a variable may be assigned multiple times. For our purposes we will treat each such assignment as creating a separate variable, and will number them sequentially. So a program fragment like:

```

1  samp1: function (y)
2      x := 1
3      y := y+1
4      x := x-y
5      return x

```

is transformed by SSA into:

```

1  samp1: function (y0)
2      x0 ← 1
3      y1 ← y0 + 1
4      x1 ← x0 - y1
5      return x1

```

and the resulting set of PDFs is:

$F_{s_{amp1}}=1$
 $P_{y_0}(v)$ is parameter 0 of *samp1*
 $P_{x_0}(v) = P_1(v)$
 $P_{y_1}(v) = P_{y_0+1}(v)$
 $P_{x_1}(v) = P_{x_0-y_1}(v)$
 $P_{x_1}(v)$ is result of *samp1*

Here the $F_{s_{amp1}}$ is the frequency of execution for that basic block. In the generated PDFs, the frequency of the main block is always counted as 1. Later this will be weighted when we consider functions calling other functions.

Conditional

A conditional creates multiple paths, conditionalized on some test. On each such path, variables may be created similarly to those created in the sequence. Where the paths come together again, SSA describes this joining of multiple values for a variable as a ϕ node that creates a new value as a combination of the previous values.

```

1  samp2: function (x)
2      if x<10
3          y := 2
4      else
5          y := 3
6      return y

```

becomes the SSA:

```

1  samp2: function (x0)
2      if x0 < 10
3          goto samp21
4      else
5          goto samp22
6
6  samp21:
7      y0 ← 2
8      goto samp23
9
9  samp22:
10     y1 ← 3
11     goto samp23
12
12  samp23:
13     y2 ←  $\phi(y_0, y_1)$ 
14     return y2

```

In SSA, the ϕ simply shows that the value comes from multiple paths. For our analysis, this is complicated by the need to weight the ϕ of the PDFs (p_1, \dots, p_n) with the frequencies associated with the paths (f_1, \dots, f_n) giving:

$$\phi_{f_1, f_2, \dots, f_n}(p_1, p_2, \dots, p_n) = \frac{\sum f_i p_i}{\sum f_i}$$

For this trivial case the resulting set of PDFs is:

$F_{s_{amp2}}=1$
 $P_{x_0}(v)$ is parameter 0 of *samp2*
 $F_{s_{amp2_1}}=F_{s_{amp2}} \times P_{x_0 < 10}(true)$
 $P_{y_0}(v) = P_2(v)$
 $F_{s_{amp2_2}}=F_{s_{amp2}} \times P_{x_0 \geq 10}(true)$
 $P_{y_1}(v) = P_3(v)$
 $F_{s_{amp2_3}}=F_{s_{amp2_1}} + F_{s_{amp2_2}}$
 $P_{y_2}(v) = \phi_{F_{s_{amp2_1}}, F_{s_{amp2_2}}}(P_{y_0}(v), P_{y_1}(v))$
 $P_{y_2}(v)$ is result of *samp2*

More Complex Conditional

As a slightly more complicated example:

```

1  samp3: function (x, z, v)
2      if x<1
3          y := x+z
4          y := x+v
5      else
6          y := x-3
7      w := y+z
8      return w

```

becomes the SSA:

```

1  samp3: function (x0, z0, v0)
2      if x0 < 1
3          goto samp31
4      else
5          goto samp32
6
6  samp31:
7      v1 ← v0
8      x1 ← x0
9      z1 ← z0
10     y0 ← x1 + z1
11     y1 ← x1 + v1
12     goto samp33
13
13  samp32:
14     x2 ← x0
15     z2 ← z0
16     y2 ← x2 - 3
17     goto samp33
18
18  samp33:
19     y3 ←  $\phi(y_1, y_2)$ 
20     z3 ←  $\phi(z_1, z_2)$ 
21     w0 ← y3 + z3
22     return w0

```

Now the analysis is further complicated by the need to weight the values with the conditionals that got us to the current block.

Conditional control flow in programs restricts the resulting PDFs for variables that are dependent on the conditionals. The PDF for conditional restriction is:

$$P_{x|wRv}(y) = \frac{g(y)}{\int g(z)dz},$$

$$\text{where } g(z) = \int \int \left\{ \begin{array}{ll} P_x(x), & \text{when } wRv \\ 0, & \text{otherwise} \end{array} \right\} dw dv.$$

For this case the resulting set of PDFs is:

$$\begin{aligned} F_{samp3} &= 1 \\ P_{x_0}(v) & \text{ is parameter 0 of } samp3 \\ P_{z_0}(v) & \text{ is parameter 1 of } samp3 \\ P_{v_0}(v) & \text{ is parameter 2 of } samp3 \\ F_{samp3_1} &= F_{samp3} \times P_{x_0 < 1}(true) \\ P_{v_1}(v) &= P_{v_0 | x_0 < 1}(v) \\ P_{x_1}(v) &= P_{x_0 | x_0 < 1}(v) \\ P_{z_1}(v) &= P_{z_0 | x_0 < 1}(v) \\ P_{y_0}(v) &= P_{x_1 + z_1}(v) \\ P_{y_1}(v) &= P_{x_1 + v_1}(v) \\ F_{samp3_2} &= F_{samp3} \times P_{x_0 \geq 1}(true) \\ P_{x_2}(v) &= P_{x_0 | x_0 \geq 1}(v) \\ P_{z_2}(v) &= P_{z_0 | x_0 \geq 1}(v) \\ P_{y_2}(v) &= P_{x_2 - 3}(v) \\ F_{samp3_3} &= F_{samp3_1} + F_{samp3_2} \\ P_{y_3}(v) &= \phi_{F_{samp3_1}, F_{samp3_2}}(P_{y_1}(v), P_{y_2}(v)) \\ P_{z_3}(v) &= \phi_{F_{samp3_1}, F_{samp3_2}}(P_{z_1}(v), P_{z_2}(v)) \\ P_{w_0}(v) &= P_{y_3 + z_3}(v) \\ P_{w_0}(v) & \text{ is result of } samp3 \end{aligned}$$

Loops

A loop creates multiple paths that flow back on themselves. On each such path, variables may be created similarly to those created in the sequence. Where the paths come together at the top of the loop, SSA describes this joining of multiple values for a variable as a ϕ node that creates a new value as a combination of the previous values. This is similar to the ϕ at the end of a conditional, except that there is a circular dependency.

```

1  samp4: function ( )
2      x := 1
3      y := 1
4      while x < 10
5          y := y * x
6          x := x + 1
7      return y

```

becomes the SSA:

```

1  samp4: function ( )
2      x0 ← 1

```

```

3      y0 ← 1
4      if x0 < 10
5          goto samp41
6      else
7          goto samp42
8      samp41 :
9          x1 ← φ(x0, x2)
10         y1 ← φ(y0, y2)
11         y2 ← y1 × x1
12         x2 ← x1 + 1
13         if x2 < 10
14             goto samp41
15         else
16             goto samp42
17     samp42 :
18         y3 ← φ(y0, y2)
19         return y3

```

and the resulting set of PDFs is:

$$\begin{aligned} F_{samp4} &= 1 \\ P_{x_0}(v) &= P_1(v) \\ P_{y_0}(v) &= P_1(v) \\ F_{samp4_1} &= F_{samp4} \times P_{x_0 < 10}(true) + F_{samp4_1} \times P_{x_2 < 10}(true) \\ P_{x_1}(v) &= \phi_{F_{samp4}, F_{samp4_1}}(P_{x_0 | x_0 < 10}(v), P_{x_2 | x_2 < 10}(v)) \\ P_{y_1}(v) &= \phi_{F_{samp4}, F_{samp4_1}}(P_{y_0 | x_0 < 10}(v), P_{y_2 | x_2 < 10}(v)) \\ P_{y_2}(v) &= P_{y_1 \times x_1}(v) \\ P_{x_2}(v) &= P_{x_1 + 1}(v) \\ F_{samp4_2} &= F_{samp4} \times P_{x_0 \geq 10}(true) + F_{samp4_1} \times P_{x_2 \geq 10}(true) \\ P_{y_3}(v) &= \phi_{F_{samp4}, F_{samp4_1}}(P_{y_0 | x_0 \geq 10}(v), P_{y_2 | x_2 \geq 10}(v)) \\ P_{y_3}(v) & \text{ is result of } samp4 \end{aligned}$$

Note that because of the loop, this is a system of simultaneous equations that must be solved.

4 Independence of PDFs

In the previous sections, the definitions of PDFs assume that the PDFs are independent. This is clearly an unreasonable simplifying assumption, but we are making progress toward removing the assumption.

5 Cumulative Example

For a cumulative example of all of the rules to date, consider the following simple program:

```

1  samp4: function ( )
2      x := 1
3      while x < 10
4          if x < 5
5              y := y * x
6          else
7              y := y + 1
8              x := 1 + x
9      return y

```

is treated as:

```

1  samp4: function (y0)
2      x0 ← 1

```

```

3      if  $x_0 < 10$ 
4          goto  $samp4_1$ 
5      else
6          goto  $samp4_5$ 
7       $samp4_1$  :
8           $x_1 \leftarrow \phi(x_0, x_5)$ 
9           $y_1 \leftarrow \phi(y_0, y_6)$ 
10         if  $x_1 < 5$ 
11             goto  $samp4_2$ 
12         else
13             goto  $samp4_3$ 
14          $samp4_2$  :
15              $x_2 \leftarrow x_1$ 
16              $y_2 \leftarrow y_1$ 
17              $y_3 \leftarrow y_2 \times x_2$ 
18             goto  $samp4_4$ 
19          $samp4_3$  :
20              $x_3 \leftarrow x_1$ 
21              $y_4 \leftarrow y_1$ 
22              $y_5 \leftarrow y_4 + 1$ 
23             goto  $samp4_4$ 
24          $samp4_4$  :
25              $x_4 \leftarrow \phi(x_2, x_3)$ 
26              $y_6 \leftarrow \phi(y_3, y_5)$ 
27              $x_5 \leftarrow 1 + x_4$ 
28             if  $x_5 < 10$ 
29                 goto  $samp4_1$ 
30             else
31                 goto  $samp4_5$ 
32          $samp4_5$  :
33              $y_7 \leftarrow \phi(y_0, y_6)$ 
34             return  $y_7$ 

```

and the resulting set of PDFs is:

$$\begin{aligned}
F_{s_{amp4}} &= 1 \\
P_{y_0}(v) & \text{ is parameter 0 of } s_{amp4} \\
P_{x_0}(v) &= P_1(v) \\
F_{s_{amp4_1}} &= F_{s_{amp4}} \times P_{x_0 < 10}(true) + F_{s_{amp4_1}} \times P_{x_5 < 10}(true) \\
P_{x_1}(v) &= \phi_{F_{s_{amp4}}, F_{s_{amp4_1}}}(P_{x_0 | x_0 < 10}(v), P_{x_5 | x_5 < 10}(v)) \\
P_{y_1}(v) &= \phi_{F_{s_{amp4}}, F_{s_{amp4_1}}}(P_{y_0 | x_0 < 10}(v), P_{y_6 | x_5 < 10}(v)) \\
F_{s_{amp4_2}} &= F_{s_{amp4_1}} \times P_{x_1 < 5}(true) \\
P_{x_2}(v) &= P_{x_1 | x_1 < 5}(v) \\
P_{y_2}(v) &= P_{y_1 | x_1 < 5}(v) \\
P_{y_3}(v) &= P_{y_2 \times x_2}(v) \\
F_{s_{amp4_3}} &= F_{s_{amp4_1}} \times P_{x_1 \geq 5}(true) \\
P_{x_3}(v) &= P_{x_1 | x_1 \geq 5}(v) \\
P_{y_4}(v) &= P_{y_1 | x_1 \geq 5}(v) \\
P_{y_5}(v) &= P_{y_4 + 1}(v) \\
F_{s_{amp4_4}} &= F_{s_{amp4_2}} + F_{s_{amp4_3}} \\
P_{x_4}(v) &= \phi_{F_{s_{amp4_2}}, F_{s_{amp4_3}}}(P_{x_2}(v), P_{x_3}(v)) \\
P_{y_6}(v) &= \phi_{F_{s_{amp4_2}}, F_{s_{amp4_3}}}(P_{y_3}(v), P_{y_5}(v)) \\
P_{x_5}(v) &= P_{1 + x_4}(v) \\
F_{s_{amp4_5}} &= F_{s_{amp4}} \times P_{x_0 \geq 10}(true) + F_{s_{amp4_1}} \times P_{x_5 \geq 10}(true) \\
P_{y_7}(v) &= \phi_{F_{s_{amp4}}, F_{s_{amp4_1}}}(P_{y_0 | x_0 \geq 10}(v), P_{y_6 | x_5 \geq 10}(v)) \\
P_{y_7}(v) & \text{ is result of } s_{amp4}
\end{aligned}$$

The PDF of the final result of the function, P_{y_7} , is completely parameterized on the input parameter (actually, free variable in this case), P_{y_0} . In principle this system of equations could be solved to a closed-form equation, so that we could provide a new PDF for y and out would pop the result PDF. Similarly, by providing failure targets, the frequency analysis would give a reliability figure for the routine, also based on the provided PDF for y .

6 Conclusions and Future Work

There are three parts to this work. The first is to describe the transformations from programs into probability distribution functions parameterized by the arguments to the components. The initial effort in this direction is described here. The next stage is to integrate it into the component composition calculus described in [4]. Both of these stages are proceeding well.

The third stage is to develop the analytical tools to allow solutions of the systems of equations that arise from the transformation (particularly for loops, as well as recursive functions). In particular, the differential equations that arise from the transformations are quite different from those that occur in analysis of current and flow in traditional engineering. The other problem is that many (if not most) of the PDFs that come out of this analysis are for discrete sets, for which the traditional mathematical tools are also less well developed.

The ultimate verdict on the usability of this technique must await sufficient time for the development of analytical techniques applicable to this domain. In the interim, we are developing simulation techniques that are applicable to small systems.

REFERENCES

- [1] D. Bjørner and O.Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
- [2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, Oct. 1991.
- [3] M. DeGroot. *Probability and Statistics*. Addison-Wesley, Don Mills, Ontario, 1989.
- [4] D. Hamlet, D. Mason, and D. Voit. Theory of software component reliability. In *Proc. 23rd International Conference on Software Engineering (ICSE'2001)*, Toronto, Canada, May 2001.