Smalltalk Metaprogramming supports Probabilistic Program Analysis

Dave Mason
Ryerson University
350 Victoria Street
Toronto, ON, Canada M5B 2K7
dmason@ryerson.ca

ABSTRACT

Program paths (equivalently program input subdomains) are interesting for a variety of purposes in software engineering, including testing, reliability, and estimating of resource requirements. The full potential of paths is often not realized because it is difficult to generate the most important paths from the – potentially infinite – set of possible paths for a program.

This paper introduces two probabilistic algorithms to generate paths based on an operational profile describing the probability distribution of the possible inputs. The first is a theoretically clean, but implementationally difficult algorithm based on program continuations. The second is a more practical, but less accurate algorithm based on Monte Carlo techniques. Prototype implementations of both algorithms are briefly examined.

Smalltalk provides the ideal testbed for these ideas because of its metaprogramming capabilities. These implementations utilize: dynamic code generation, full object orientation, and first-class continuations – a combination of features found in very few other languages.

Categories and Subject Descriptors

F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*probabilistic computation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools, tracing*; G.3 [**Probability and Statistics**]: Reliability and life testing

General Terms

Measurement, Reliability, Theory, Metaprogramming

1. INTRODUCTION

A program path or program trace is one of the many (possibly infinite) legal sequences of basic blocks through a program. That is, a program path represents a distinct sequence of instructions, executed for a particular input value, determined by a sequence of predicates representing the program conditionals and loops. In fact, the same sequence of instructions will be executed for the set of values – a program input sub-domain – selected by the conjunction of that sequence of predicates.

This paper introduces two different probabilistic algorithms to generate paths based on an operational profile describing the probability distribution of the possible inputs. The paths and corresponding domains produced by these algorithms could be provided as input to other programs to evaluate various software metrics on the program under examination. Because the generation of paths from these algorithms is driven by the operational profile, the most relevant ones will be produced first and significant problems will be detected earlier.

Discovered paths can be useful for several problems in software engineering.

1. When looking at test coverage, path coverage is the gold standard, but rarely seriously considered because the number of potential paths is too large, and it is difficult to prioritize among the available paths.[2, 4, 7, 8, 19, 21, 22] Given that these algorithms deliver paths in the order of most significance, they would make path coverage highly usable. Generated paths/domains can be fed to a program that checks each domain against all the existing tests to verify that each domain is covered, flagging those that are not.
2. When considering reliability of a program, path domains are good candidates for appropriate units to which to assign a pass/failure since all points in the domain will execute exactly the same instructions and thus should have comparable correctness.[6, 10, 15, 16, 17, 18, 20] Generated paths/domains can be fed to a program that tests the program under examination against a gold standard in each domain and verifies that they produce the same result. From this a reliability calculation can be made for the program operating with the particular operational profile.[1]

[1]It's a bit more complicated than this, but the complications

3. When considering resource consumption, such as execution time or stack utilization, since the same instructions are executed, all the points in the same domain should have comparable resource comsumption.[5] The definition of "instruction" can be important to this because things like variable-sized heap allocation may be treated as a single "instruction". With the frequency information provided by the operational profile, paths could also be used for program performance tuning and profile-directed compilation,[9]

1.1 Identifying Paths

Paths can be identified via structural induction across the program control flow graph (CFG), using different walks of the control flow graph. The challenge for any such walk is generating the most important paths from the potentially infinite set of possible paths.

Depth First

Depth-first search is probably the simplest approach, but because it takes no account of possible input values, it runs almost immediately into potentially infinite loops – and hence infinite path lengths. While such loops can be detected, the question of when to stop moves into the realm of the heuristic option.

Breadth First

Breadth-first search is more complicated, but potentially not much better because many of the first paths identified may be error-handling and other low-frequency paths.

Heuristic

Heuristic approaches use knowledge about program structures, such as typical numbers of loop iterations and error handling, to control the depth of the search for paths and to focus on the more core paths through the program.[13]

1.2 Operational Profiles

An operational profile is a statistical description of the context in which the program will be used.[11, 12] Ideally an operational profile will be based on actual field usage. While it is convenient to think of an operational profile as a multivariate (one for each parameter to the program) probability density function (PDF), which could be a continuous function, an operational profile is often developed as a histogram. The important property is that the function is non-negative at all points and that the integral over the entire input space is exactly 1.

2. PROBABILISTIC PATH GENERATION

This paper introduces two probabilistic algorithms that utilize an operational profile describing the probability distribution of the possible inputs. Both algorithms generate paths more-or-less in the order from the the most to least significant, as determined by the operational profile.

Probabilistic execution of a program looks at values differently. For discussion purposes, consider programs with numeric parameters. Where in a normal program execution, variables all contain actual numeric values, in a probabilistic execution, variables contain either numbers or expressions of the (probabilistic) input parameters. The input parameters are never reified, except as defining a region of the input space – a sub-domain.

Predicates then become comparisons of expressions of the input parameters and, rather than truth or falsity, they have only probabilistic meaning. When combined with a set of other probabilistic predicates forming a path, they define a portion of the input space and have a particular probability of input values being in the domain, based on the operational profile.

2.1 Continuation-based path generation

This algorithm is theoretically very clean; the first path produced is strictly the most frequent path and the frequency is available as soon as the path is produced. Then the next path produced is strictly the next most frequent, and so on.

A continuation is the "rest of the program execution".[1] Capturing a continuation allows the program to do some other work and then to return to the continuation to allow it to continue executing as if nothing had happened.

There are two difficulties with this algorithm. Firstly, the algorithm is dependent on being able to capture the program continuation at the point a probabilistic predicate is used to determine program flow. Secondly, the correctness of the frequency promise above is dependent on the accuracy of operational profile and the quality of the integration of the operational profile over the sub-domains.

The algorithm maintains a priority queue of continuation/ path/ frequency ordered by frequency. When a probabilistic predicate must be evaluated to determine which of two paths to follow, the continuation is captured and two objects are put on the priority queue. One object placed on the priority queue contains

- the predicates that led to the decision point, augmented by the new predicate,
- the continuation to return `true`, and
- the frequency of that augmented set of predicates.

The other object placed on the priority queue contains

- the predicates that led to the decision point, augmented by the complement of the new predicate,
- the continuation to return `false`, and
- the frequency of that augmented set of predicates.

Then the highest frequency partial path on the priority queue is popped, the current predicate set is set to that of the popped object, and the continuation is resumed.

This may lead to many partial path executions before one execution gets to the end of the program under test. When

are outside the scope of this program. See [10, 15] for more information.

```
ProbabilisticExec evaluate: [:x |
    t:=0.
    [t*0.1<x] whileTrue: [
            t:= t+t+1.
            r:=t].
    r]
  value: (PDFNormal new mean: 1.5 stdDev: 0.5)
  result: [:result :path|
       "use the result and path"
     ].
```

Figure 1: Using ProbabilisticExec class

one execution gets to the end of the program under test, its path, frequency, and result are reported, and the next object on the priority queue is popped. When there are no more objects on the priority queue, all the paths have been generated and executed to completion.

2.2 Monte Carlo-based path generation

This algorithm is somewhat simpler as no integral is *required*, and no continuations are captured.

The drawbacks of this algorithm are that frequencies for paths are not generated by the algorithm and immediately available, and the paths are not strictly ordered, such that the first-produced path may not be the highest frequency path. However, the first paths produced will tend to be the most important, so the basic properties of the first algorithm will be upheld.

The algorithm generates a datapoint in the input space, randomly but using the operational profile to chose the value from the more-probable sub-domains.[2] It then tests the point against a list of domains identified to date. If that domain has already been identified, the algorithm goes back to generate another point. Otherwise, the program is executed with the chosen point. Probabilistic predicates are generated as in the probabilistic algorithm, but when one must be evaluated to determine which of two paths to follow, the list of current predicates is augmented by the new predicate and then execution is resumed with the correct true or false value so that the branch goes in the proper direction.

3. PROTOTYPE IMPLEMENTATION

The actual code to handle the probabilistic control flow is only a few dozen lines of Smalltalk code, but when added to code for integration, multi-variate polynomials, and tests it currently totals about 2500 lines of code.

Figure 1 shows how easily paths can be generated from a block of code. The frquency and the set of predicates are available from the `path` object. The `MonteCarloExec` class is used in exactly the same way, although as mentioned earlier, doesn't produce as useful a frequency value.

For example, the path $0 < v_1 <= 0.1$ – one of the paths generated from the program in figure 1 – would be represented

[2]Generating datapoints in this order is an open problem.

```
mustBeBoolean
    ^ProbabilisticExecution currentExecution
          atContext: thisContext sender
          addPredicate: self
```

Figure 2: `mustBeBoolean` method from `Maybe`

as a compound predicate equivalent to (and compiled as) `[:v1| 0.1 >= v1 and: [0 < v1]]`.

Smalltalk was chosen for the protype implementation for several reasons:

- Dynamic types
- First-class booleans
- Dynamic code generation
- Continuation capture

3.1 Dynamic types

Dynamic types allow existing code to be analysed without any change to the code. This is desirable because the very nature of the analysis is to examine the code for errors, so any required manipulation would allow for the introduction (or elimination!) of errors from the original code.

When the block from the example in figure 1 is executed, the variable `x` is substituted with a probabilistic variable representing the normal distribution described by the `PDFNormal` probability density function. While these probabilistic variables can act like numbers (arithmetic operations building up polynomials of the probabilistic variables), they are also the hooks that trigger the analysis. Comparisons among the probabilistic values or between probabilistic values and numbers produce probabilistic booleans of the subclass `Maybe`.

3.2 First-class booleans

In Smalltalk all control flow is described using polymorphism; that is, all subclasses of the `Boolean` class implement the method `ifTrue:`, but the `True` class evaluates the block passed as a parameter while the `False` class ignores it – and *vice versa* for `ifFalse:`. Thus, to implement the probabilistic booleans of the subclass `Maybe` should simply entail defining the required methods from `Boolean`.

However, to get reasonable performance from Smalltalk, the compiler recognizes that these message sends are actually conditionals and compiles them into conditional jumps that only recognize the known `true` and `false` values. The compiler performs similar optimizations for loops.

But all is not lost! If a value tested is not one of the expected values (`true` or `false`), the interpreter sends the message `mustBeBoolean`, which allows the object to do whatever it needs to do and then to return a known boolean value so the branch test can proceed.

Figure 2 shows the `mustBeBoolean` method from `Maybe`. When

```
atContext: context addPredicate: maybe
    | result |
    result := maybe evaluateWith: currentValues.
    currentPath addPredicate: (
        result ifTrue: [maybe]
              ifFalse: [maybe not]).
    context skipBackBeforeJump.
    ^ result
```

Figure 3: `atContext:addPredicate:` **method from** `MonteCarloExec`

```
atContext: context addPredicate: maybe
    | newPath frequency |
    context skipBackBeforeJump.
    newPath := currentPath copy addPredicate: maybe.
    frequency := newPath frequencyWith: currentPDF.
    frequency >0 ifTrue: [
        priorityQueue add:
            (ProbContinuation fromContext: context
                                         path: newPath
                                       result: true)].
    newPath := currentPath addPredicate: maybe not.
    frequency := newPath frequencyWith: currentPDF.
    frequency >0 ifTrue: [
        priorityQueue add:
            (ProbContinuation fromContext: context
                                         path: newPath
                                       result: false)].
    self queueNext
```

Figure 4: `atContext:addPredicate:` **method from** `ProbabilisticExec`

a `Maybe` boolean is tested, it dispatches to the current probabilistic execution engine to capture the predicates and to determine whether the branch should be taken or not.

Figure 3 shows the `atContext:addPredicate:` method from `MonteCarloExec`. Here we simply evaluate the `Maybe` predicate with the current injected point, then based on that evaluation add the predicate or its complement to the current path, and finally return the result to the branch point so that it can branch in the appropriate direction.

Figure 4 shows the `atContext:addPredicate:` method from `ProbabilisticExec`. In this case, no evaluation of the predicate is required because we are going to capture both paths:

1. create the `true` continuation
    - a copy of the current path is augmented with the predicate
    - the frequency of that path is determined by taking an integral of the current PDF over the resulting path
    - a continuation is created with that path that will eventually return `true`
    - that continuation is added to the priority queue.
2. create the `false` continuation
    - the current path is augmented with the complement of the predicate
    - the frequency of that path is determined by taking an integral of the current PDF over the resulting path
    - a continuation is created with that path that will eventually return `false`
    - that continuation is added to the priority queue.
3. Pull the highest priority continuation off the queue and resume it.

3.3 Dynamic code generation

Both algorithms require fast testing of whether a point is contained in a domain determined by a path.

In the `ProbabilisticExec` engine, two integrals need to be calculated over a domain at each conditional point in the program execution. Since Monte Carlo integration is the only feasible approach, this can require potentially hundreds of membership tests per integral.

In the `MonteCarloExec` engine, each generated point has to be checked to see if the domain containing it has already been covered, and the tester's analysis code may also want to calculate the integral over the operational profile.

As mentioned in the example, a path for a domain like $0 < v_1 <= 0.1$ would be represented as a conjunction of comparisons. The fact that this can be converted to the block `[:v1| 0.1 >= v1 and: [0 < v1]]` and be compiled allows this membership test to be much faster than if the predicates needed to be interpreted.

3.4 Continuation capture

As can be seen in figure 4, in the `ProbabilisticExec` engine execution follows a probabilistic path through the program. At every point where an input parameter affects control flow, the engine captures the continuation and adds it twice to a priority queue with appropriate priorities based on the integral of the operational profile over the corresponding domains. Eventually both of the continuations will return – once with `true` and once with `false` – in an order and at a time dictated by the operational profile in use.

Squeak Smalltalk has an implementation of continuation capture that allows the restoring of temporary variables to their state as of the continuation capture. Although not currently implemented, this could conceivably be extended to restore any referenced objects to *their* state, but this could easily get out of hand.

4. CONCLUSIONS

The order of generation of the progam paths can be driven by the operational profile describing the probability distribution of the inputs of a program, so that more important paths can be generated earlier.

To the extent that one can generate accurate operational profiles, the approaches described in this paper can dramatically improve the usefulness of path-based techniques for quality metrics.

If a reasonably accurate operational profile is available for a program, and if a set of paths has been generated, the integrals over the domains corresponding to the paths can provide several pieces of supplementary information.

1. The sum of the integrals gives a lower bound on the domains considered so far for this particular usage.
2. The sum of the integrals of the successful domains (whether refering to coverage or reliability) provides a lower bound on the success rate for this particular usage.
3. The sum of the integrals of the failed domains provides a lower bound on the faliure rate for this particular usage.
4. The combination of the success and failure rates provides an asymptotic bound on the range of possible success rates; as more domains are evaluated those bounds can be moved arbitrarily close.
5. While the integrals over the domains are specific to a particular operational profile, the paths and domains themselves are independent of the operational profile being used, so at any time the success/failure/considered integrals can be recalculated with alternate operational profiles, providing the same information, but for a new usage.

Smalltalk made this experiment possible, and makes it possible to evaluate the efficacy and efficiency of the two algorithms. That evaluation is currently being undertaken.

Other future work is to determine operational profile descriptions that can give useful integrals for non-numeric parameters.

5. REFERENCES

[1] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] R. Cheung and C. Ramamoorthy. Optimal measurement of program path frequencies and its applications. In *Proceedings 1975 Intl. Fed. Automat. Contr. Congr.*, Aug. 1975.

[3] P. Frankl, editor. *Proceedings of the 2002 International Symposium on Software Testing and Analysis.* ACM Software Engineering Notes, July 2002.

[4] A. Goldberg, T.-C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In Ostrand [14], pages 80–94.

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

[6] D. Hamlet. Continuity in software systems. In Frankl [3].

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

[8] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In Ostrand [14], pages 95–107.

[9] S. Jinturkar, J. Thilo, J. Glossner, P. DArcy, and S. Vassiliadis. Profile directed compilation in dsp applications. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT98)*, Sept. 1998.

[10] D. Mason. *Probabilistic Program Analysis for Software Component Reliability.* PhD thesis, University of Waterloo, 2002.

[11] J. Musa, G. Fuoco, N. Irving, and D. Kropfl. *The Operational Profile*, pages 167–216. McGraw-Hill, New York, 1996.

[12] J. D. Musa. Applying operational profiles in testing. In *Proceedings of 10th International Software Quality Week*, pages 1–25, San Francisco, CA, May 1997.

[13] M. N. Ngo and H. B. K. Tana. Heuristics-based infeasible path detection for dynamic test data generation. *Information and Software Technology*, 50(7–8):641–655, jun 2008.

[14] T. J. Ostrand, editor. *Proceedings of the 1994 International Symposium on Software Testing and Analysis.* ACM Software Engineering Notes, Aug. 1994.

[15] D. Richardson. *A Partition Analysis Method to Demonstrate Program Reliability.* PhD thesis, University of Massachusetts, Sept. 1981.

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

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

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

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

[20] D. Woit. *Operational Profile Specification, Test Case Generation, and Reliability Estimation for Modules.* PhD thesis, Queen's University, Kingston, Ontario, February 1994.

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

[22] S. J. Zeil and L. J. White. Sufficient test sets for path analysis testing strategies. In *5th ICSE*, pages 184–194. IEEE Computer Society, Catalog No. 81CH1627-9, Mar. 1981.