The Optimisation of Stochastic Grammars to Enable Cost-Effective Probabilistic Structural Testing

Simon Poulding
Dept. of Computer Science
University of York
York YO10 5GH, UK
smp@cs.york.ac.uk

Robert Alexander
Dept. of Computer Science
University of York
York YO10 5GH, UK
rda@cs.york.ac.uk

John A. Clark
Dept. of Computer Science
University of York
York YO10 5GH, UK
jac@cs.york.ac.uk

Mark J. Hadley
Dept. of Computer Science
University of York
York YO10 5GH, UK
mjh130@york.ac.uk

ABSTRACT

The effectiveness of probabilistic structural testing depends on the characteristics of the probability distribution from which test inputs are sampled at random. Metaheuristic search has been shown to be a practical method of optimising the characteristics of such distributions. However, the applicability of the existing search-based algorithm is limited by the requirement that the software's inputs must be a fixed number of numeric values.

In this paper we relax this limitation by means of a new representation for the probability distribution. The representation is based on stochastic context-free grammars but incorporates two novel extensions: conditional production weights and the aggregation of terminal symbols representing numeric values.

We demonstrate that an algorithm which combines the new representation with hill-climbing search is able to efficiently derive probability distributions suitable for testing software with structurally-complex input domains.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems; G.1.6 [**Numerical Analysis**]: Optimization

Keywords

Software Verification; Search-Based Software Engineering; Structural Testing; Statistical Testing; Stochastic Context-Free Grammars

(c) 2013 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the national government of United Kingdom. As such, the government of United Kingdom retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.
Copyright 2013 ACM 978-1-4503-1963-8/13/07 ...$15.00.

1. INTRODUCTION

The term 'statistical testing' has a number of meanings in the context of software verification, but refers here to a form of probabilistic structural testing described by Thévenod-Fosse and Waeselynck [12, 13, 14]. The strategy is to randomly sample test inputs from an input profile (a probability distribution over the input domain) having a particular property: that every structural element in the software-under-test is exercised as frequently as possible by the sampled inputs.

Statistical testing is, potentially, a very effective testing strategy. As for deterministic forms of structural testing, the codes's structure provides guidance as how the testing effort should be applied in order to detect faults throughout software. But statistical testing additionally shares the benefits of other probabilistic methods of generating test inputs. In particular, it is easy to generate inputs for tests sets of any chosen size by repeatedly sampling from the input profile. This flexibility in the test set size enables the most effective use to be made of the available testing time and resources. Thévenod-Fosse and Waeselynck demonstrated empirically that the combination of structural guidance and flexible test size enables statistical testing to be more effective at detecting faults than both uniform random testing and deterministic structural testing [12].

However, a significant barrier to the use of statistical testing is the cost of finding a suitable input profile: it is difficult to derive a probability distribution over the software's input domain that induces an effective distribution over the software's structural elements.

In our previous work, we demonstrated a cost-effective search-based algorithm for deriving suitable input profiles [9, 10]. The algorithm represented candidate input profiles as a Bayesian network and therefore had a significant limitation: it could be applied only to software for which inputs consisted of a fixed number of numeric values. In this paper, we propose a grammar-based representation that overcomes this limitation.

The contributions we make are:

1. A new representation for input profiles that is capable

of generating structurally-complex test inputs. The complexity may arise from constraints on the validity of the input, a mixture of categorical and numeric data, and the use of variable-length sequences. The new representation is based on stochastic context-free grammars, but incorporates two novel extensions. Firstly, conditional production weights introduce a limited form of context-sensitivity and enable the representation of more subtle probability distributions than would normally be possible. Secondly, terminal symbols representing numeric values are aggregated so as to reduce the size of the representation and thereby facilitate its optimisation using search. (Section 3.1)

2. A search method that may be applied to the grammar-based representation in order to derive input profiles suitable for statistical testing. (Section 3.3)

3. A demonstration of the cost-effectiveness of a search algorithm using three diverse software examples, and empirical evidence that the two novel extensions to the grammar facilitate the derivation of input profiles. (Section 4)

2. BACKGROUND

2.1 Statistical Testing

Statistical testing requires that the input profile—the probability distribution over the input domain of the software-under-test (SUT) from which inputs are sampled—satisfies an adequacy criterion. The criterion is expressed in terms of a set, $\mathcal{C}$, of coverage elements such as structural elements of the SUT's source code or functional elements of the SUT's specification. For each coverage element, $c_i \in \mathcal{C}$, let p_i be the probability that the coverage element is exercised by a single input chosen at random from the given input profile, and p be the minimum of the p_i. The adequacy criterion requires that p (which we refer to as the *minimum coverage probability*) be as large as possible.

The rationale for this criterion is efficient coverage of the structural elements. Let Q be the probability that every coverage element is exercised by at least one test case in the test set. The relationship between Q, p, and the test size N is given by (adapting a result given in [13]):

$$Q = 1 - (1 - p)^N \quad (1)$$

For effective testing, Q should have a value close as possible to 1 in order to cover as much of the software as possible. Maximising the minimum coverage probability enables either (a) an increase in Q for a given test set size (which will improve the fault-detecting ability); or (b) a reduction in the test size (and thus the testing costs) while maintaining an acceptable value of Q.

To construct input profiles with suitably high minimum coverage probabilities, both static and dynamic techniques are described in the literature.

Gouraud and Denise describe a static technique which assigns weights to edges in the SUT's control-flow graph in order to sample execution paths in a manner that satisfies the statistical testing adequacy criterion [5, 2]. However, computationally expensive constraint-solving (similar to that used in symbolic execution) is required to derive test inputs which exercise the sampled paths, and many of sampled paths are infeasible in practice. Both these factors limit the scalability of the technique.

Thévenod-Fosse and Waeselynck describe a dynamic technique [13, 14]. The minimum coverage probability of a candidate input profile is estimated by executing an instrumented version of the SUT with inputs sampled from the profile. The results are used to guide the manual adjustment of the profile, and the process repeated until a suitable value of the minimum coverage probability is attained. However, Thévenod-Fosse and Waeselynck speculate that this technique is unlikely to be practical for large SUTs in its manual form.

2.2 Deriving Input Profiles Using Search

In our previous work, we demonstrated that metaheuristic search can automate the dynamic technique of Thévenod-Fosse and Waeselynck, and thereby reduce the cost of deriving suitable input profiles for statistical testing [9].

We made the assumption that inputs to the SUT were a fixed number of integer or real-valued arguments. This assumption enabled input profiles to be represented as Bayesian networks: directed acyclic graphs in which nodes correspond to input arguments, and edges describe conditional dependence between arguments. The conditional distributions at each node were represented by partitioning the domain of the input vector component into bins and assigning a probability to each bin.

To derive input profiles using this Bayesian network representation, we use random mutation hill-climbing to optimise the edges between nodes, the number and size of bins at a node, and the probabilities assigned to each bin.

In subsequent work, we demonstrated an enhancement to the search algorithm that improved performance by a factor of five for some SUTs [10]. The enhancement—described as 'direction mutation'—used additional information obtained from executing the instrumented SUT to bias the mutation operators to parts of the candidate input profile that exercised the coverage element(s) having the lowest coverage probability.

Although this work has shown that metaheuristic search is an effective and cost-efficient approach to deriving input profiles, the assumption of inputs being a fixed number of numeric values limits the SUTs to which this approach may be applied. It is this limitation we seek to remove using the new reprentation described in this paper.

3. PROPOSED SEARCH ALGORITHM

3.1 Representation

3.1.1 Stochastic Context-Free Grammars

Formal grammars define a language of *strings*: sequences of symbols drawn from a set of *terminal* symbols. The grammar restricts valid strings to be a subset of all possible sequences of terminals by means of production rules: only strings that can be constructed by the application of the grammar's production rules are valid. The productions of context free-grammars, on which our representation is based, have the form:

```
V → X1 X2 ...Xn
```

where `V` is one of an additional set of symbols called *variables*, and the `Xi` are either variable symbols or terminals.

The generation a valid string from the grammar begins with a string consisting of a single copy of the designated 'starting' variable symbol, usually denoted `S`. Generation proceeds by considering the leftmost variable in the current string. A production is chosen which has the this variable on the left-hand side, and the variable is replaced in the string with the symbols on the right-hand side of the production rule. The process of replacing variables continues until the string contains no variable symbols.

As an example, consider the following context-free grammar for defining simple arithmetic expressions. (For clarity, we surround terminal symbols in this and subsequent grammars by single quotes. We also use the standard notation of concatenating productions with same left-hand side variable, separating the alternative right-hand sides using vertical bars: `X` → `Y | Z` is equivalent to the two productions `X` → `Y` and `X` → `Z`.)

```
   S → Expr
Expr → Num | Expr Op Expr
 Num → '0' | '1' | '2' | '3' | '4' | '5'
  Op → '+' | '-' | '*' | '/'
```

This grammar generates valid strings such as `'4 + 3 * 5'`, but does not generate invalid strings such as `'- 3 1'`.

In a stochastic context-free grammar (SCFG), each production is additionally annotated by a weight. During string generation, if more than one production could be applied, the weights are interpreted as a probability distribution and one of the productions is chosen at random according to distribution.

When used as a representation for input profiles, the grammar productions define the valid inputs to the SUT, and the weights are used to define the probability distribution over the inputs, i.e. an input profile. (We note, however, that some types of input structures cannot be represented using context-free grammars.)

3.1.2 Conditional Weights

Context-free grammars are so-called because the set of productions that may be applied to a variable does not change depending on the symbols occurring before or after the variable in the string during the generation process; this is a consequence of only a single variable being permitted on the left-hand side of production rules.

The context-free property simplifies the grammar representation, but limits the types of input profiles that may be represented using the weights of a SCFG. Consider again the example grammar for generating simple arithmetic expressions. If an optimal input profile must avoid too many divide-by-zero arithmetic operations (i.e. the substring `'/ 0'`), the only mechanism available to do this is to assign a low weight to the production `Op` → `'/'`, or to the production `Num` → `'0'`. However such weights would also reduce the probability of otherwise desirable substrings such as `'/ 2'` and `'+ 0'`.

To overcome this limitation, we enhance the standard form of SCFGs by introducing conditional weights for the productions. A 'child' variable may be identified as having one or more 'parent' variables (the set of which may include the child variable itself) on which it is dependent. The distribution of weights over the productions for the child variable are then conditionally dependent on which productions were most recently applied to the parent variables.

For example, we may identify the variable `Num` in our example grammar as being dependent on the variable `Op`. The weights over the 6 productions for `Num` are then conditionally dependent on which production was most recently applied to `Op`. In effect, there are 5 (potentially different) distributions of weights over the productions of `Num`: one for each of the 4 possible productions of `Op`, plus a further distribution that is used if no production has yet been applied to `Op`.

The conditional dependency of weights extends the types of input profiles that may be represented by introducing a limited form of context-sensitivity (but only in the weights; the grammar itself remains context-free). It would, for example, enable divide-by-zero errors to be avoided more accurately by setting the weight of `Num` → `'0'` to a low value *only* in the conditional distribution used when the most recent production applied to `Op` was `Op` → `'/'`, but retaining a normal weight for `Num` → `'0'` for all the other productions of `Op`.

The conditional dependency between variables—which parents are related to which child variables—is discovered by the search algorithm; it does not need to be specified *a priori*.

3.1.3 Aggregation of Numeric Terminals

Some variables in the grammar may be used to represent a numeric range. For example, the variable `Num` in the example grammar above represents an integer in the range $[0, 5]$. When the cardinality of the range is small, it is feasible to use a separate production for each possible value in the range. However, for much larger ranges, such an approach would create an excessive number of productions and associated weights: it is unlikely that it would be practical to optimise such a large representation using search.

To accommodate large numeric ranges, we propose a further extension to the standard form of SCFGs. Variables representing numeric ranges are identified as special type that we term 'binned variables'. Rather than specifying a production for each possible value in the range, the range is instead partitioned into a small number of bins.

For example, if the variable `Num` instead represented the range $[-32768, 32767]$, it might be partitioned into three bins (of different sizes) as follows:

$$\texttt{Num} \rightarrow [-32768, -2480] \mid [-2479, 238] \mid [239, 32767]$$

When one of these three productions is chosen during the generation of a string from the grammar, the output is not the bin, but a value picked at random from a uniform distribution across the bin. For example, if the production `Num` → $[-2479, 238]$ is applied, then the output is an integer between -2479 and 238.

By aggregating a potentially large number of terminals into a much smaller number of bins, the size of the representation is reduced; we speculate that this will improve the practicality of the search process.

The number and lengths of bins need not specified *a priori*: the search process attempts to derive a suitable partitioning.

3.2 Fitness Metric

The fitness metric is an estimate of the minimum coverage probability induced by the candidate input profile. It is evaluated by sampling a number (K) of inputs from the stochastic grammar, and executing an instrumented version of the SUT with the inputs. For each coverage element c_i,

the estimated coverage probability, p_i, is the proportion of the K inputs that exercised the element one (or more) times. The lowest value of p_i across all the coverage elements is an estimate of the minimum coverage probability.

Since the sample of inputs is of finite size, the fitness metric is an estimate and may sometimes be higher than the 'true' value for the candidate input profile. In this situation, the input profile may be retained in error over many iterations of the search. In our previous work, we were able to minimise this effect by continuing to evaluate the current input profile (for up to μ_{eval} hill-climbing iterations), and re-calculating a more accurate estimate of fitness from the accumulated instrumentation data.

3.3 Search Method

There is some correspondence between the new grammar-based representation and the previous Bayesian network representation. For this reason, we propose a hill-climbing search similar to that described in [10].

3.3.1 Random Mutation Hill-Climbing

The search begins from an random input profile constructing by randomising the weights, conditional dependencies between variables, and number and size of bins in the grammar. A limit (μ_{prnt}) may be specified for the number of parent variables any one variable may be conditionally dependent on, and on the number of bins that a binned variable may have (μ_{bins}): both the initial random profile and neighbours created by mutation respect these limits.

At each iteration of the hill-climb, a small number (λ) of neighbours to the current input profile are created by applying one of the following mutation operators:

$M_{\mathbf{prb}}$ increases or decreases a single production weight by a factor ρ_{prb};

$M_{\mathbf{add}}$ adds a conditional dependency between two variables;

$M_{\mathbf{rem}}$ removes a conditional dependency;

$M_{\mathbf{len}}$ increases or decreases the length of one bin by a factor ρ_{len};

$M_{\mathbf{spl}}$ splits a bin into two new bins;

$M_{\mathbf{joi}}$ joins two adjacent bins.

Each mutation operator M_{x} has an associated weight, w_{x}: operators with higher weights are more likely to be applied when creating a neighbour.

The neighbours are evaluated, and if the fittest neighbour is fitter than the current input profile, the neighbour becomes the current profile in the next iteration.

3.3.2 Directed Mutation

We incorporate an enhancement to the search method that we describe as 'directed mutation'. (This enhancement was discussed in section 2.2 above, and is explained in detail in our previous work [10].) When the fitness of the current input profile is evaluated, data is maintained as to which productions gave rise to inputs that exercised the coverage element(s) with the minimum coverage probability. The weights (and bins, if a binned variable) associated with these productions are then mutated with a higher likelihood when creating neighbours of the current input profile.

Directed mutation is implemented by grouping the mutation operators into three groups:

$G_{\mathbf{edge}}$ operators that modify conditional dependencies: M_{add} and M_{rem};

$G_{\mathbf{bins}}$ operators that modify production weights and bin terminals directly: M_{len}, M_{spl}, M_{joi}, and M_{prb};

$G_{\mathbf{drct}}$ consists of the same weight- and bin-modifying operators as G_{bins}, but applies them *only* to production weights and bins that contributed strings that exercised the element(s) with the lowest coverage probability.

Each group G_{x} has associated weight W_{x}. When choosing a mutation operator to apply, a group is first chosen at random according to the group weights, and then a mutation operator is chosen at random from that group according to the mutation operator weights. The group weights control the 'strength' of directed mutation effect.

If the minimum coverage probability is zero, one or more coverage elements must have been exercised by none of the sampled inputs and so there is no data available with which to apply directed mutation. In this situation, the algorithm foregoes the local mutation operators described above and constructs entirely random neighbours (in the same way as initial random profile). The motivation is to rapidly explore the search space in order to find a region where the minimum coverage probability is non-zero and directed mutation will be effective.

4. EMPIRICAL DEMONSTRATION

4.1 Objectives

The objectives of the empirical work described in this section are:

1. To demonstrate that proposed search algorithm is able to derive input profiles suitable for testing software taking structurally-complex inputs.
2. To assess whether the two extensions to the standard form of stochastic context-free grammars—conditional production weights and the aggregation of numeric terminals—facilitate the search algorithm.

4.2 Software Under Test

4.2.1 circBuff

`circBuff` is the implementation of a circular buffer container in the BOOST C++ library, version 1.50. The testing objective is coverage of branches in the public methods of the public interface class of the container. (We omit the copy constructor and assignment operator in order to avoid a step-change in the complexity of the grammar since these methods take a further container object as a parameter.) The number of branches to be covered is 78. The public methods of the class constitute approximately 700 lines of code, but not all methods include branched code.

Since the container object maintains state between method calls, it is not sufficient for the test case to be a single method call. Instead, we consider a test input to be a sequence of method calls and we use the grammar shown in Figure 1 to create such inputs. The grammar defines a valid structure for the sequence of method calls: a constructor, method calls to the constructed object that fill the buffer, method calls

```
               S → Constructor FillMethods Methods 'destructor'
     Constructor → 'circular_buffer[0]' BufferCapacity
                 | 'circular_buffer[1]' BufferCapacity BufferSize
     FillMethods → FillMethod FillMethods | FillMethod
      FillMethod → 'push_front' | 'push_back'
         Methods → Method Methods | Method
          Method → 'push_front' | 'push_back'
                 | 'linearize' | rotate' IteratorPos
                 | 'set_capacity' BufferCapacity
                 | 'resize' BufferSize
                 | 'rset_capacity' BufferCapacity
                 | 'rresize' BufferSize
                 | 'insert[0]' IteratorPos
                 | 'insert[1]' IteratorPos Number
                 | 'rinsert' IteratorPos | 'erase[0]' IteratorPos
                 | 'erase[1]' IteratorPos IteratorPos
                 | 'rerase[0]' IteratorPos
                 | 'rerase[1]' IteratorPos IteratorPos
                 | 'pop_front' | 'pop_back'
                 | 'front' | 'back' | 'operator[]' Number
  BufferCapacity → [0,9]
      BufferSize → [0,9]
     IteratorPos → [0,9]
          Number → [0,9]
```

Figure 1: The grammar defining inputs to `circBuff`

```
                  S → EpuckCluster ObjClusters
       EpuckCluster → EpuckClustAzimuth EpuckClustDist Epucks
             Epucks → Epuck
              Epuck → EpuckAzimuth EpuckDist EpuckAngle
         ObjCluster → ObjClustAzimuth ObjClustDist Objects
        ObjClusters → ObjCluster | ObjCluster ObjClusters
            Objects → Object | Object Objects
             Object → Obstacle | Patch
           Obstacle → ObstAzimuth ObstDist ObstRadius
              Patch → PatchAzimuth PatchDist PatchRadius
  EpuckClustAzimuth → [0,359]
     EpuckClustDist → [0,199]
       EpuckAzimuth → [0,0]
          EpuckDist → [0,0]
         EpuckAngle → [0,359]
    ObjClustAzimuth → [0,359]
       ObjClustDist → [400,599]
        ObstAzimuth → [0,359]
           ObstDist → [0,99]
         ObstRadius → [10,99]
       PatchAzimuth → [0,359]
          PatchDist → [0,99]
        PatchRadius → [10,99]
```

Figure 2: The grammar defining inputs to `epuck`.

that operate on the buffer, and finally a call to the destructor. Terminals, such as `'push_front'`, specify method calls. (The suffices of the form `[n]` distinguish between overloaded methods.) Parameters to method calls are represented by the four binned variables at the end of the grammar. Recursion in the grammar enables the generation of variable-length sequences of method calls.

The strings generated by the grammar are interpreted by a test harness and applied to an instrumented version of the container class. The object returned by a call to a constructor method is stored by the harness and subsequent method calls are made to that object.

4.2.2 *epuck*

'E-pucks' are small, relatively cheap robots used in robotics research. The SUT `epuck` is controller code that forms part of a lightweight simulator for e-puck robots written in C by Paul O'Dowd of the University of Bristol. Features supported by the simulator include infra-red proximity detection, communication between robots, the placement and detection of coded patches on the floor, and the placement of static obstacles in the environment.

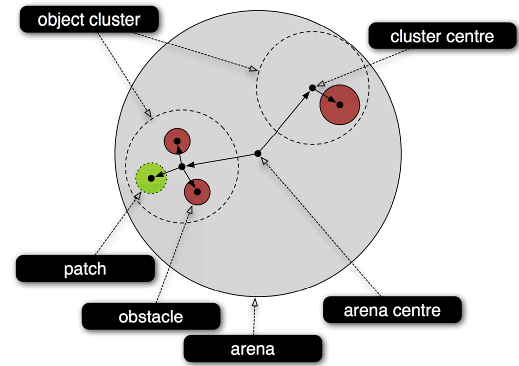


Figure 3: The mission environment specified in terms of object clusters.

The controller code has two significant conditional statements: the first tests whether an obstacle is nearby and if so, directs the robot to take avoidance; the second tests for the presence of coded 'food' patches on the floor and if so, attempts to maintain position over the patch. The testing objective is to exercise the four branches from these two conditional statements during a short 'mission' lasting the equivalent of 10 seconds in the simulation.

A test input is a configuration of the robot's environment in which the mission occurs, consisting of fixed circular arena and a number of obstacles and patches.

The configuration is described by the grammar listed in Figure 2, which defines obstacles and patches in terms of groups we call object clusters. An example of two clusters is shown in Figure 3. The objects in the cluster—obstacles and patches—are defined using distances and angles from centre of the cluster.

The recursion in the grammar permits different numbers of object clusters, each with different numbers of obstacles and patches, to be generated. Binned variables are used to generate the positions of clusters, obstacles, patches, and e-pucks; and the radii of obstacles and patches.

Strings generated by the grammar are interpreted by a test harness and used to configure the simulated environment. To avoid unrealistic environments, only the first three obstacles and first three patches are configured. Similarly, only the first e-puck specified by the generated string is added to the arena.

The controller code itself consists of 53 lines of code, but the testing process requires the code to be executed as part of the much larger simulator. The mission duration of 10 seconds is equivalent to 500 time steps in the simulation. While the real-world elapsed time for each simulated mission is much less than a second, this is nevertheless much longer than the execution of time of the other two SUTs used in this empirical work.

4.2.3 *tcas*

The previous Bayesian network representation could be applied only to SUTs with inputs consisting of a fixed number of numeric arguments. In order to demonstrate that the new grammar-based representation is also practical for

```
  S → A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
 A1 → [0,39999]
 A2 → [0,1]
 A3 → [0,1]
 A4 → [0,39999]
 A5 → [0,4999]
 A6 → [0,39999]
 A7 → [0,3]
 A8 → [0,39999]
 A9 → [0,39999]
A10 → [0,2]
A11 → [1,2]
A12 → [0,1]
```

Figure 4: The grammar defining inputs to `tcas`.

SUTs with this simpler form of input domains, we include a common example from the testing literature: the Traffic Collision Avoidance System (TCAS) used by commercial aircraft. The code was provided by the Software-artifact Infrastructure Repository [3].

The SUT takes 12 integer arguments, and thus the grammar consists only of 12 binned variables—one for each argument across the valid range of the argument—plus the start variable (Figure 4). This simple grammar always processes each variable exactly once and always in the same order. For this (and similar) grammars, the fixed generation order enables the search space to be reduced by restricting the conditional dependencies between variables: the parent variable must be processed earlier than child, otherwise the dependency would have no effect.

The testing objective is coverage of all reachable 62 conditions (atomic Boolean predicates) in both conditional and assignment statements.

4.3 Empirical Method

The empirical demonstration compares four variants of the algorithm implemented in C++:

1. The standard hill-climbing algorithm as described in section 3.
2. A variant in which conditional production weights are prohibited (by setting the parameter μ_{prnt} to 0).
3. A variant in which the range of a numeric variable is not partitioned into bins (by setting the parameter μ_{bins} to 1).
4. An equivalent form of random search: instead of creating neighbours by application of a single mutation operator, an entirely random profile is generated (in the same way that the initial random profile is constructed).

For each variant and for each SUT, 16 trials—each using a different seed to the pseudo-random number generator—were run. Each trial executed on a single CPU core. The clock speed of, and memory accessible to, the cores were typical of a desktop PC. The trials ran for 20,000 iterations for `circBuff` and `tcas`, taking on average of 14 minutes and 1.5 minutes per trial, respectively. Each execution of `epuck` during fitness evaluation takes much longer than the other two SUTs, so the algorithm trials ran for a fewer number of iterations: 400 iterations, taking on average of 22 minutes per trial.

During each trial, the fitness (the estimated minimum coverage probability) of the best input profile found so far was recorded in order to assess the trajectory taken by the search. (The fitness of the *best* profile so far is not necessarily the same as that of the *current* profile owing to the re-evaluation mechanism described in section 3.3.)

Parameter	Description	Setting
K	evaluation sample size	302
λ	neighbourhood sample size	9
ρ_{prb}	production weight mutation factor	54.002
ρ_{len}	bin length mutation factor	2.755
W_{bins}	G_{bins} group weight	1000
W_{edge}	G_{edge} group weight	144
W_{drct}	G_{drct} group weight	10584
w_{rem}	M_{rem} mutation weight	1000
w_{add}	M_{add} mutation weight	689
w_{prb}	M_{prb} mutation weight	1000
w_{joi}	M_{joi} mutation weight	886
w_{spl}	M_{spl} mutation weight	1159
w_{len}	M_{len} mutation weight	1028
μ_{prnt}	max. parent variables per child	1
μ_{bins}	max. terminal bins per variable	$2\sqrt{\lvert\mathcal{C}\rvert}$
μ_{eval}	max. evaluations per profile	10

Table 1: The algorithm parameter settings. (The parameters are described in section 3.3. The expression $|\mathcal{C}|$ denotes the number of coverage elements.)

The remaining algorithm parameter settings are listed in Table 1. The settings were re-purposed from the result of tuning the older Bayesian network algorithm to a different set of SUTs [8]. In order to reduce the threat to validity of using a single set of parameter settings, the parameters were not tuned to new grammar representation nor to the new SUTs under consideration here.

4.4 Results

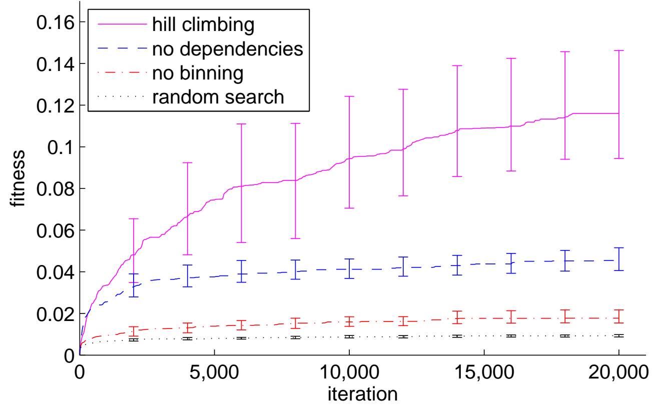
The results are summarised in Figure 5 for each of three SUTs. The graphs illustrate the trajectory of the search by plotting the average fitness, calculated as the mean across all 16 trials of the algorithm variant, at each iteration of the algorithm. The fitness plotted is the estimated minimum coverage probability of the best profile found so far, and so this value is the fitness of the input profile that would have been returned by the search algorithm if it had been stopped at that iteration. The 95% confidence intervals, indicated by error bars at regular intervals along the trajectory, were calculated by bootstrap resampling.

We choose to summarise the results as trajectories to ensure that any conclusions we draw are reliable and not an artefact of a particular point at which we might choose to stop the algorithm.

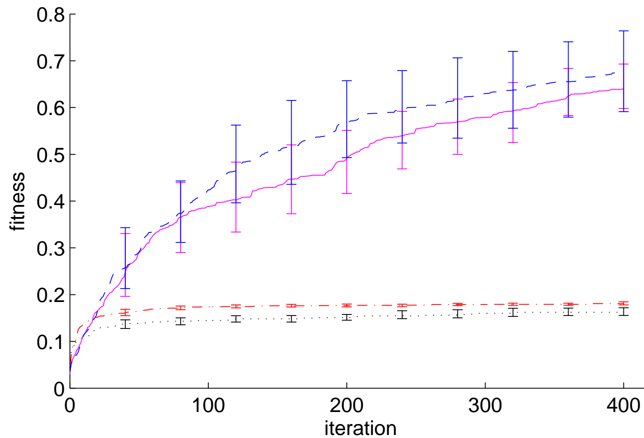
The raw data from the algorithm trials is available at: `http://www.cs.york.ac.uk/~smp/supplemental/`.

4.5 Discussion

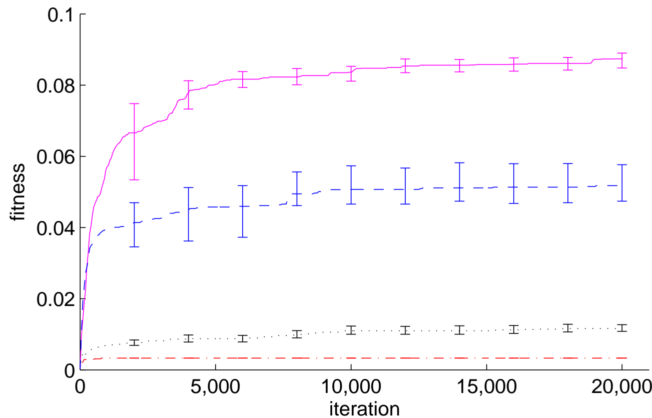
For all three SUTs, the hill-climbing algorithm (solid line trajectories) derives input profiles with the highest minimum coverage probability. (For `epuck`, there is no statistically significant difference between this algorithm and the variant that does not use conditional production weights: the error bars overlap across the entire trajectories.) As discussed in section 2.1, profiles with the highest minimum coverage probability will be the most cost-effective when used for statistical testing. We are not aware of any analysis in the literature of the best possible value of the minimum coverage probability for these SUTs against which we may compare



(a) `circBuff`



(b) `epuck`



(c) `tcas`

Figure 5: Trajectories for the four search algorithms. The solid line plots the mean fitness of the best profile found to date at each iteration for the algorithm using the standard hill-climbing search method; the dashed line is the variant that omits any dependencies between variables; the dash-dot line is the variant that omits binning; and the dotted line is the algorithm using random search. The error bars show the 95% confidence interval; for clarity, they are plotted only at regular intervals.

these results. However, by applying equation (1), we note that *all* the coverage elements will be covered (with a probability, Q, of 0.99) using test set sizes of 38 for `circBuff`, 5 for `epuck`, and 51 for `tcas`. (At such a high value of Q, such test sets will often exercise every coverage element more than once.) These relatively small test sizes demonstrate that the search algorithm is effective at deriving input profiles.

For both `circBuff` and `tcas`, the algorithm variant that does not use conditionally dependent weights (dashed trajectories) derives profiles with a lower fitness than the standard hill-climbing algorithm. This difference is statistically significant across a wide range of iterations (the error bars do not overlap). We conclude that the novel extension of conditional production weights facilitates the search algorithm.

For all three SUTs, the algorithm variant that does not partition numeric variables into bins (dash-dot trajectories) derives profiles with a much lower fitness than the standard hill-climbing algorithm, and the difference is statistically significant. This is only partial evidence in support of aggregating numeric terminals. We do not produce empirical evidence in this paper for the other part of the argument: that if each numeric range were represented by separate terminals for each possible value, then the search space would be impractically large. Instead, we argue merely that for SUTs taking many thousands of different numeric inputs (such as `tcas`), this is a reasonable assumption.

Finally, for all three SUTs, random search (dotted trajectories) derives profiles with a much lower fitness than the hill-climbing algorithm. This result provides a check that the task of deriving input profiles for these SUTs is not a trivial search problem.

5. RELATED WORK

The use of grammars to generate test inputs is an established practice known as *grammar-based testing*. It is motivated by the need to generate test data for SUTs for which the nature of valid inputs is highly-constrained. For example, grammar-based testing can be used to generate test inputs—source code—for compilers and interpreters [7, 4]. Inputs consisting of random sequences of characters would be almost always raise errors in the lexer or parser phases of a compiler; the code of subsequent compilation phases would remain untested. A grammar, however, ensures the construction of semantically correct source code, enabling the compiler functionality to be fully tested.

In many cases, the grammar is deterministic and input data is generated by bounded-exhaustive enumeration: all valid language strings are generated up to a chosen size-related bound. However, a few techniques sampled inputs at random from stochastic grammars. Applications of this probabilistic approach include: testing of hardware arithmetic circuits in simulation [6]; differential testing of compilers [7]; and testing the bytecode verifier in Java Virtual Machine implementations [11]. In all these examples, the grammar's production weights were manipulated manually to achieve desirable properties in the randomly-sampled test inputs.

In this paper we have demonstrated the use of automated search—rather than manual manipulation—to adjust the weights of a stochastic grammar for this purpose; the desirable property in our case being a high value of the minimum coverage probability. In this respect, our approach is most similar to the recent work of Beyene and Andrews [1].

Although Beyene and Andrews' approach is not framed explicitly in the context of statistical testing, it achieves similar objectives. Grammars for structured HTML and XML inputs are converted to stochastic data generation programs, and the weights in the generation programs are optimised by metaheuristic search for high coverage of the SUTs.

Our proposed search algorithm differs from Beyene and Andrew's technique in two notable aspects.

Firstly, we use a grammar representation that incorporates two novel extensions: conditional production weights and the aggregation of numeric terminal symbols. Our empirical results show that these extensions enable the search algorithm to derive input profiles with significantly better minimum coverage probability.

Secondly, the adequacy criterion in Beyene and Andrew's experiments was statement coverage: between 50% and 73% of the statements were covered using large test sets of size 1000. Our empirical demonstration uses the much stronger adequacy criteria of branch and condition coverage. Moreover, we calculated above that we are able to (effectively) guarantee that 100% of the coverage elements would be exercised using *much* smaller test sets. (Since the SUTs used in the empirical work of this paper are not the same as those used by Beyene and Andrews, a more rigorous comparison of the two approaches is not currently possible.)

6. CONCLUSION

In this paper we proposed a novel grammar-based representation for input profiles, and demonstrated a search algorithm using this representation that is capable of efficiently deriving input profiles suitable for statistical testing. The grammar-based representation enables the algorithm to be applied to a wide range of software, in particular software with structurally-complex inputs. For three real-world examples, the algorithm took only a few minutes to derive suitable input profiles using computing resources equivalent of a desktop PC. Since the approach is automated and uses readily-available, affordable computing resources, the costs associated with deriving input profiles for statistical testing are reduced.

The new representation extends stochastic context-free grammars through the use conditional production weights and aggregating numeric terminal symbols in bins. The empirical results provide evidence that both of these novel extensions facilitate the search process.

The grammar-based representation and associated search algorithm provide a generic mechanism for describing and optimising probability distributions over the input domain of a SUT: they could be applied to the problems other than statistical testing for which the objective is to induce a particular probability distribution over the executed software. For example, the objective could be to exercise particular parts of the software that have been neglected by earlier testing; to focus on components that have a history of faulty behaviour; or to explore non-functional properties such as execution time.

As future work, we are investigating the use of stochastic grammars that are more flexible than context-free grammars and will facilitate the representation of input profiles for a yet wider range of SUTs.

Acknowledgments

This research was funded by the MOD Centre for Defence Enterprise and EPSRC grant EP/J017515/1, DAASE: Dynamic Adaptive Automated Software Engineering. The authors would like to thank Paul O'Dowd of the University of Bristol for providing the e-puck simulator code.

7. REFERENCES

[1] M. Beyene and J. Andrews. Generating string test data for code coverage. In *Proc. IEEE Int'l Conf.. on Software Testing, Verification and Validation (ICST 2012)*, pages 270–279, 2012.

[2] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. Technical Report 1386, Laboratoire de Recherche en Informatique (LRI), CNRS - Univ. de Paris Sud, 2004.

[3] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[4] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2008)*, pages 206–215, 2008.

[5] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *Proc. IEEE Int'l Conf. on Automated Software Eng.*, 2001.

[6] P. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50 –55, July 1990.

[7] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[8] S. Poulding. *The Use of Automated Search in Deriving Software Testing Strategies*. PhD thesis, Dept. Computer Science, University of York, (in review).

[9] S. Poulding and J. A. Clark. Efficient software verification: Statistical testing using automated search. *IEEE Trans. Software Eng.*, 36(6):763–777, 2010.

[10] S. Poulding, J. A. Clark, and H. Waeselynck. A principled evaluation of the effect of directed mutation on search-based statistical testing. In *Proc. 4th Intl. Workshop on Search-Based Software Testing (SBST 2011)*, pages 184–193, 2011.

[11] E. G. Sirer and B. N. Bershad. Using production grammars in software testing. *SIGPLAN Not.*, 35(1):1–13, Dec. 1999.

[12] P. Thévenod-Fosse and H. Waeselynck. An investigation of statistical software testing. *J. Software Testing, Verification and Reliability*, 1(2):5–26, 1991.

[13] P. Thévenod-Fosse and H. Waeselynck. Statemate applied to statistical testing. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA'93)*, pages 99–109, 1993.

[14] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. Software statistical testing. Technical Report 95178, Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS (LAAS), 1995.