Optimizing the Parameters of an Evolutionary Algorithm for Fuzzing and Test Data Generation

Ralf Gerlich
Dr. Rainer Gerlich System and Software Engineering BSSE
Immenstaad, Germany
https://orcid.org/0000-0001-6309-927X

Christian R. Prause
DLR Space Administration
Bonn, Germany
https://orcid.org/0000-0003-4856-529X

***Abstract*—Communication interfaces are particularly challenging to test using automatically generated test data. The test data sent through the interface must be "valid enough" to overcome initial sanity checks of the interface and reach functions deep inside the integrated software. Machine-readable information about what data forms "valid enough" messages is rarely available to test data generation tools. So instead, we evolve the messages with an evolutionary algorithm. This enables efficient fuzz testing for the communication interface between a satellite and its ground station. In this paper, using an algorithm implementation in our fuzzing tool DCRTT, we investigate the impact of algorithm parameter selection on the performance and the possibility of efficient general default parameter values. The preliminary results promise significant improvements to automated testing with respect to software security testing and quality assurance.**

I. Introduction

As illustrated by well-known examples such as the first flight of the Ariane 5 launcher in 1996 [1], the more recent losses of the Hitomi space telescope [2] or the Schiaparelli lander [3], single failures or malfunctions in the onboard or flight software can cause total loss of the spacecraft. A substantial increase in losses due to software problems has been observed within the last 20 years, with software problems being responsible for between 13% and 30% of failures ([4], [5]).

Specifically after the failure of Ariane 5 on its maiden flight, strictness of verification and validation processes in European space programmes has increased considerably. Among the standards published by the European Cooperation for Space Standardisation (ECSS) those for software engineering (ECSS-E-ST-40C) and software product assurance (ECSS-Q-ST-80C) are the most extensive.

Automation of test data generation for software is a crucial component in the strategy for keeping up with these rising demands. However, communication interfaces pose particular challenges for automatic test data generation: Messages are typically passed in the form of byte streams, while at the same time the contents of these streams must follow strict rules (see Fig. 1).

```
tc_error_t process_telecommand(
        unsigned char* data,
        size_t size);
```

Fig. 1. Example of a generic byte-stream interface

Often, information about the structure and semantics of these byte streams are not readily available in a form that can be processed automatically. The information may be spread out over multiple documents written in natural language, making manual extraction of formal information costly and automatic extraction too difficult or error-prone. In such cases, structural coverage may be the only formal criterion remaining, although coverage-directed test case selection may be considered problematic [6].

Several approaches for coverage-driven test data generation have been suggested and evaluated in the past, ranging from random test data selection to constraint-based concepts. While the former are simple to implement, they often are unable to achieve sufficient coverage or require a large number of test inputs to do so [7], [8]. In contrast, the latter are more effective in achieving coverage, but come with the additional burdens of implementation and computational complexity [9], [10].

Optimisation-based test data generation may be a third alternative. Here, the coverage goal is expressed in the form of a cost function to be minimised, and a solution is found using classical optimisation algorithms. For this, the actual target hardware can be used and only simple instrumentation is required.

In this paper, we define an optimisation-based algorithm for coverage-driven generation of unstructured test data using evolutionary algorithms and present results from systematic performance measurements. For the algorithm, specifically adapted cost functions and mutation operators were developed. Preliminary investigation [11] provided initial evidence that the concept is feasible and indicated potential for further optimisation. Besides its simplicity, the algorithm remains based on randomisation, addressing at least in part some of the concerns related to coverage-driven test case selection.

Theory of operation of the algorithm suggests that there are optima for at least some of the configuration

©2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

parameters. Further, some of the evolutionary operators may be redundant.

Thus, the following research questions are addressed in this paper:

- Can optima for parameters be observed in practice?
- Is it possible to simplify the algorithm by removing elements?
- How can the performance be improved further?

The rest of the paper is structured as follows: In Section II we present related work. This is followed in Section III by the introduction of general terms and theoretical concepts which form the basis of the approach. In Section IV we describe our specific algorithm used for test data generation, followed by a discussion of possible parameter optima in Section V. The measurement setup as well as the software specimens used as test subjects for the measurements are introduced in Section VI. Results from the measurements are presented in Section VII and discussed in Section VIII. Conclusions on the feasibility of the algorithm as well as possible future enhancements are given in Section IX.

II. Related Work

Random test data generation injects randomly selected data into the system under test based on the declared interfaces such as functions available in the code and parameter types specified. The goal is to achieve structural code coverage or consistent coverage of the input domain ([12], [13], [7]). One of the benefits of this approach is its simplicity. However, random testing also does not explicitly distinguish valid from invalid inputs, and is often unable to provide test data for complete coverage. This is specifically an issue when generating test data for interfaces with very generic parameter types.

Considering a continuum of generic methods for test data generation according to their complexity, with random testing on the one end, at the other end we might find constraint-based approaches ([14], [15], [16], [17]). These apply constraint-solving techniques to determine inputs that lead to coverage of specific elements of the code, and can be distinguished into path- and goal-oriented methods. Most of these methods use symbolic execution techniques to reason about the programs to be tested, and thus also share the quite considerable challenges of symbolic program analysis such as computational complexity [10], [9], accuracy of models with respect to hardware, the pointer aliasing problem [18] as well as infeasible paths.

A path in a control flow graph (CFG) is considered *infeasible* if there is no input that leads to its execution. The proportion of infeasible paths among the possible paths in a CFG may be considerable [17]. As a consequence, repeated random selection of new paths until a feasible path is found may require an unacceptable amount of retries.

The latter has been quite successfully addressed in the area of fuzz-testing by concolic strategies, as evidenced by the performance of tools such as CUTE [19] and DART [20]. Concolic testing uses the execution paths observed during testing with concrete values to determine feasible paths, and determines test inputs for slightly modified execution paths using symbolic execution. This way, the issue of infeasible paths can be diminished quite effectively. However, the other issues of modelling and complexity remain.

Search-based approaches [14], [21] and evolutionary algorithms [22] may be a possible middle ground between random and constraint-based methods. Specifically fuzzers based on evolutionary principles such as AFL[1] or VUzzer have shown good results [9].

III. Theoretical Foundations of our Approach

Our approach is based on some already known and pre-established theoretical concepts, which will be introduced in this section, and may be skipped by readers already familiar with these principles.

As we aim to select test data based on *structural coverage criteria*, we describe those used in the algorithm. Based on these concepts, we consider the *cost functions* defined by Korel [14] for test goals based on structural coverage. We also describe the *dominator tree*, which is used to determine a measure of distance in our cost function definition.

Finally, we detail the basic concepts of evolutionary algorithms used in our approach. This includes the more general concepts of populations, cross-over and mutation, but also more specific ideas such as elitism, immigration and mutation reversal.

A. Structural Coverage Criteria

Our algorithm aims to select test inputs with the goal of achieving structural code coverage. Specifically, the approach targets *node* and *edge coverage* based on the *control flow graph* (CFG), of which Fig. 2a shows an example. A number of widely-used structural test criteria such as *statement-*, *decision-* and *condition-coverage*, as well as related criteria such as *Modified Condition-/Decision-Coverage* (MC/DC) can be generalised to these concepts.

A node in the CFG of a *function under test* (FUT) is considered to be *covered* by a specific input if and only if during execution of the FUT with the given input the respective node is traversed at least once. A node is considered *unreachable* if there is no input under which the node can be covered.

The appropriate definitions regarding edges follow in an analogous manner. It should be noted that coverage of all edges also implies coverage of all nodes.

B. Korel's Cost Functions

In order to express coverage goals as optimisation problems, the conditions encountered along a path to the node or edge to be covered need to be transformed into a *cost*

[1] http://lcamtuf.coredump.cx/afl/

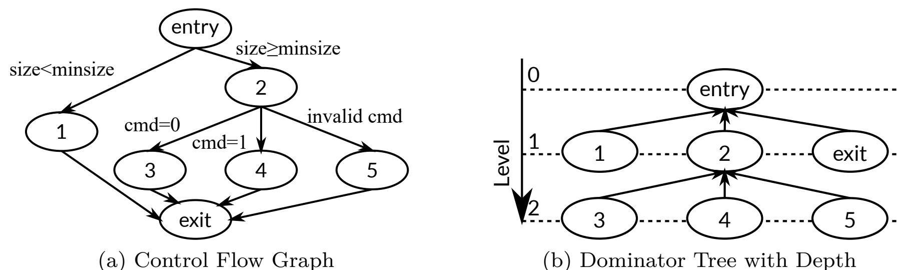

Fig. 2. Control Flow Graph and associated Dominator Tree

function to be minimised. In Fig. 2a, to execute the path from the entry via nodes 2 and 3 to the exit, the input must fulfil the path constraint (size ≥ minsize) ∧ (cmd = 0).

The approach introduced by Korel [14] in his concept of optimisation-based test data generation uses cost functions for elementary boolean expressions. These are also the basis for the cost functions in our approach.

The elementary cost function for a condition of the form E_1 *op* E_2 has the form $F_{op}(E_1, E_2)$ *rel* 0, where *op* represents comparison operators ($\leq$, $<$, $\geq$, $>$, $=$, $\neq$) and *rel* is either $\leq$, $<$ or $=$.

For example, a comparison $E_1 < E_2$ can be transformed to $E_1 - E_2 < 0$. The equality relation $E_1 = E_2$ can be represented using $|E_1 - E_2| = 0$ and the inequality $E_1 \neq E_2$ by using $-|E_1 - E_2| < 0$ [21].

Thus, an input $\mathbf{x}$ that ensures that a specific path traversing edges $p = e_1, \ldots, e_n$ is taken, can be be found by minimizing the value of

$$F_p(\mathbf{x}) = \max\{F_{e_i}(\mathbf{x}) : i = 1, \ldots, n\} \quad (1)$$

where F_{e_i} is the cost function associated with the condition at edge e_i. If the minimum value of F_p is greater than 0, there is no input that would lead to execution of p and the path is infeasible. To represent non-atomic conditions, short-circuit-code can be applied.

C. Domination and the Dominator Tree

For our cost-function, a measure of distance between nodes in the CFG is required, which we define based on the so-called dominator tree. If in a CFG, all paths from the entry node to a node n traverse a node d, then d is said to *dominate* n [23]. For example, in the CFG shown in Fig. 2a, any path from the entry to Node 3 must traverse Node 2, and thus, the latter dominates the former.

The domination relation can be represented as a tree with the entry node as root, which allows the definition of node distance by the differences in node level. The dominance tree for the CFG in Fig. 2a is given in Fig. 2b.

D. Evolutionary Algorithms

Evolutionary or *Genetic algorithms* (GA) are a class of optimisation methods which are based on the application of evolutionary principles, an area of artificial intelligence. A *population* consisting of a set of candidate solutions – the *individuals* – is gradually evolved towards an optimisation goal using concepts analogous to survival of the fittest, procreation and mutation.

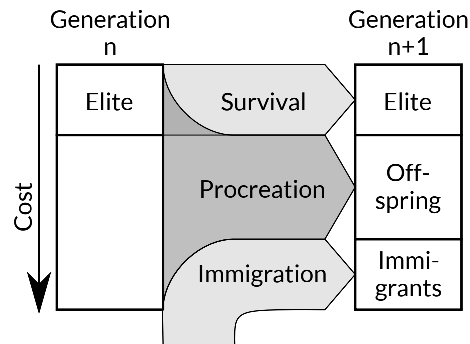

Fig. 3. Evolutionary Principle

Different from other optimisation methods, evolutionary algorithms consider a possibly large number of solution candidates at the same time. This way, typical issues of non-linear optimisation problems such as convergence to local minima can be avoided.

The optimisation objective is represented by a fitness or a cost function, depending on whether a maximization or a minimization is to be performed. The value of that function is defined on each individual, and determines how well the individual solves the respective problem.

Optimisation is performed iteratively, and the composition of the population at each iteration is called a *generation*. The process ends when either an acceptable solution has been found or a given number of generations has been evaluated.

In Fig. 3, the general principle of iteration for the genetic algorithms used in this paper is shown:

- A portion of the population with the lowest cost values – the *elite* – is transferred to the new generation without any modification [24].
- Another specified portion of the population is filled with new, randomly generated individuals. This operation is referred to as *immigration* [13] or *hypermutation* [25].
- The remainder of the new generation is filled up with offspring generated from individuals in the previous generation by *recombination* in pairs and *mutation*.

In the algorithm,

- preservation of the *elite* serves to prevent the loss of well-adapted individuals,
- *immigration* and *mutation* are intended to introduce variability into the population and prevent convergence towards local minima, and
- *recombination* is aimed at the production of new solution candidates by combining features from previous solution candidates.

In addition to the definition of the elite, the cost function also plays a role in recombination: Individuals with lower cost value shall have a higher chance of taking part in recombination.

IV. Our Algorithm

Our algorithm aims to find a test input for covering a specific target node or edge in the FUT. This can be used to complement coverage achieved by other means such as random testing. As indicated in Section III-B, this can be transformed into a minimization problem.

Our solution uses a modified evolutionary algorithm to find such test inputs. The individuals in our algorithm take the form of a byte stream with a finite length, where contents and length may vary between individuals within a configured range. For the algorithm, we define

- a specific cost function,
- appropriate cross-over and mutation operations, and
- a method of probabilistic mutation reversal.

A. Cost Function

The cost value of an individual is determined by executing the FUT with the associated byte stream as input and observing the execution path using instrumentation. If the target node or edge is reached, the value of the cost function is zero. Otherwise, the cost value is determined by looking at the first branch that led away from the target. Such a branch must have at least one successor node from which the target can be reached and one, from which this is not possible.

For example, consider the CFG in Fig. 2a. The way to reach Node 4 is via the entry node and Node 2. Thus, the decisions at these two nodes are relevant for reaching Node 4. If execution proceeds from Node 2 to Node 3, then that is the first branch leading away from the target, as Node 4 would have been reachable from Node 2, but is not reachable any more from Node 3.

Formally, the cost value $f\left(c, d; m\right)$ is given as

$$f\left(c, d; m\right) := \min\left\{c; m-1\right\} + md \tag{2}$$

where d is the *distance* of the branch from the target (1 in the example above), c is the *condition cost* associated with the specific condition that led to the unwanted branch (see Section IV-C for details), and m is the *cut-off value* for the condition cost, ensuring that cost values associated with different distances from the target have disjoint ranges.

A candidate for the distance would be the length of the shortest path from the source node of the current condition to the target node or the destination node of the target edge. However, that would be too costly to calculate for each pair of nodes. In our implementation, d is the distance of the branching node from the target node in the dominator tree (cf. Section III-C).

TABLE I
Cost Function Definition for E_1 *op* E_2

op	F_{op}
$\leq$	$\max\left\{E_1 - E_2; 0\right\}$
$<$	$\max\left\{E_1 - E_2 + \delta; 0\right\}$
$\geq$	$\max\left\{E_2 - E_1; 0\right\}$
$>$	$\max\left\{E_2 - E_1 + \delta; 0\right\}$
$=$	$\mathrm{bits}\left(E_1 \; xor \; E_2\right)$
$\neq$	$\begin{cases} \delta & \text{if } E_1 = E_2 \\ 0 & \text{otherwise} \end{cases}$

B. Handling Loops

If the target node or edge is contained in a *loop*, there may be a new chance of reaching the target for each iteration of the loop. Due to the loop being a strongly connected component, none of the decisions made inside the loop – except for those exiting the loop – would lead to a cost being applied.

Therefore a CFG without back-edges is used as the basis for reachability decisions. For this, the minimum distance of each node from the entry node is defined. A *back-edge* is an edge leading from a node with higher minimum distance to lower minimum distance.

As a consequence, in a program containing loops, a branch away from the target may occur multiple times if the target lies inside the loop. In these cases, the minimum cost occurred in any of the iterations is used as the cost for the individual.

C. Condition Cost

The functions used in this algorithm to determine condition cost for different conditional operators are given in Table I. The relation *op* is the one that has to be fulfilled in order to lead execution towards the desired branch.

They are similar to those introduced in Section III-B, but with some modifications:

- Their values are ensured to be positive. This way, the actual minimum is at zero.
- The small value $\delta > 0$ ensures that the cost function only becomes zero if the desired condition is fulfilled.
- The definition of the equality operator is based on the bit distance. In context of the bit flip mutation operator (cf. Section IV-E) this cost function has shown better monotonicity.
- For two equal operands only a small change is necessary to make them not equal. Thus, the cost function for $\neq$ only has two possible outcomes.

D. Generation of Offspring

In Fig. 4 the principal operation of recombination and mutation is shown. The process consists of multiple steps:

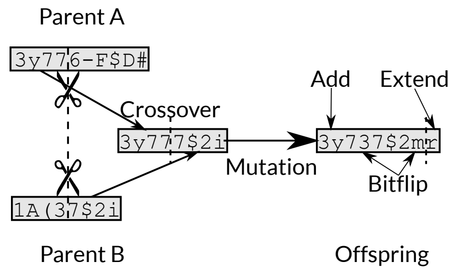

Fig. 4. Creation of a new individual

1) First, two parent individuals are selected randomly using *stochastic acceptance* [26].
2) A random cut-point k is selected using a uniform distribution.
3) A new byte stream is formed by concatenating the first k bytes from the first parent to the bytes from the second parent starting at byte $k + 1$.
4) Mutation is applied to the new individual.

E. Mutation

There are several *mutation operators* available, which are optionally applied in sequence:

- *Extension* of the byte stream by one random byte at the end,
- *reduction* of the byte stream by one byte, cutting off the last byte,
- random *bit flips*, and
- *addition* of a random value to a randomly selected byte.

Each of these mutation operators is applied with a configurable probability. Extension and reduction of the byte stream length are performed at most once per recombination operation, and only if the length of the resulting stream would still be within the configured range.

Bit flips and addition mutations may be executed multiple times. The probability for n such operations to be performed on a single offspring is $p^n\,(1 - p)$, where p is the bit flip or addition probability parameter, respectively.

F. Mutation Reversal

Mutation reversal *randomly reverts* individual mutations if they increase the cost value. For this, the FUT has to be executed after each mutation to re-evaluate the cost function. This strategy allows employing probabilistic gradient descent in addition to other optimisation operators.

V. Expectations for Optima

For some of the parameters, the existence of optimal values is to be expected:

- Increasing *population size* will increase the number of executions of the FUT, but also the chance of finding a solution.
- An *elite proportion* of 0 would mean that well-adapted individuals may be lost from one generation to the next. In contrast, for an elite proportion of 1 would degenerate the algorithm to an initial random sampling without evolution.
- An *immigrant proportion* of 1 would degenerate the algorithm to constant random re-sampling of the population without evolution.
- A too high level of mutation may lead to too much variation of already good solution candidates and thus stagnating progress in later phases of iteration.
- Reversal with a probability of 1 would lead to simple gradient descent, with the possibility of finding only local minima. However, this may be counteracted by the fact that the genetic algorithm does consider multiple solution candidates.

VI. Methods

To evaluate the impact of the algorithm parameter values on performance, we ran measurements on several code examples. We considered search success and the number of executions of the FUT as *primary end-points*.

Execution time was not considered as primary end-point, as it is influenced also by the execution time of the FUT. Optimizing configuration parameters for execution time might thus implicitly optimise for the selection of input values that minimize execution time. However, execution time was recorded to validate this assumption.

The algorithm was implemented as part of the commercial random test tool DCRTT [7]. DCRTT first stimulates the function under test using random data. Then the evolutionary algorithm is applied by successively considering targets remaining uncovered. This also means that targets accidentally covered by earlier applications of the evolutionary algorithm are not considered again later. In this way, we only look at targets that are difficult to cover.

For each target, measurements were repeated multiple times ($n = 2000$) with values for the algorithm parameters selected randomly according to a uniform distribution. Each measurement ran until either the search for a solution succeeded or the maximum number of generations (2000) had been reached.

Two measurement campaigns were executed:

- The goal of the first campaign was to identify major influences on algorithm performance.
- The goal of the second campaign was to support finer analysis of specific aspects of the algorithm. For this, strongly disadvantageous elements identified in the first campaign were deactivated.

For each measurement, the following data was recorded:

- The values of all algorithm parameters,
- a flag indicating whether a solution was found (search success),
- the total execution time in seconds,
- the total number of generations evaluated, and
- the total number of times the FUT was executed.

TABLE II
PARAMETER RANGES

Parameter	Minimum	Maximum
Maximum Individual Size (Bytes)	1	32
Population Size	2	500
Elite Proportion	0	0.90
Immigrant Proportion	0	0.90
Extension Probability	0	0.95
Reduction Probability	0	0.95
Bit Flip Probability	0	0.95
Addition Probability	0	0.95
Mutation Reversal Probability	0	1.00

A. Parameter Ranges

The ranges of the parameters considered are given in Table II. Minimum individual size was fixed to 0 bytes. The size of valid messages for both examples does not exceed 32 bytes, so this was chosen as a maximum. Note that some probability parameters where not evaluated to their full range for reasons discussed in Section V. Immigrant proportion was reduced if elite and immigrant proportions together would otherwise exceed 1.

B. Analysis Models

We used a logistic regression model to analyse the impact of the parameters given in Table II on search success probability. For the execution counts, a linear model was used, with the execution count normalised by its standard deviation used as endogenous variable. A significance level of $p \leq 0.05$ was used for all analyses.

These models included a linear and a quadratic term for each varied parameter in order to determine possible optima. To assess the impact of parameters, the difference between their respective worst- and the best-case contribution was assessed.

A linear model of execution time based on execution counts and maximum individual size was used to determine whether the latter sufficiently explained the former.

C. Test Corpus

Six functions – two groups of three each – were used to evaluate the algorithm. They are based on entry-point functions found in real flight software.

The two groups of functions mainly differ in that those in the first group only consider constant-length messages while those in the second group also allow for variable-length messages. Each of the functions checks each of the validity conditions for fulfilment, and returns a success code if the command is valid, or a failure code if it is not. Thus, reaching the check for a validity condition depends on the other conditions to be fulfilled.

The main function of the first group is a dispatch function, shown in Fig. 5. The second group only contains decoding functions for commands of similar structure, but differing in the way validation is performed.

Note that the `type`-field used for command dispatch in Fig. 5 is 32 bits wide. In a single run of the evolutionary

```
tc_error_t process_telecommand(
                              unsigned char* data,
                              size_t size) {
  tc_header_t* header = (tc_header_t*)data;
  if (size<sizeof(tc_header_t))
    return tc_error_invalid_size;
  switch (header->type) {
  case tc_set_log_parms:
    return set_log_parms(tc);
  case tc_download_log: /* Block 9 */
    return download_log_tc(data,size);
  default:
    return tc_error_invalid_type;
  }
}
```

Fig. 5. Example Code from First Group (excerpt)

algorithm without mutation reversal, the FUT is executed up to 10^6 times – once per individual and generation. The probability of hitting at least one in 2^{32} values randomly with that many tries is less than $3\,10^{-4}$. Thus, if the evolutionary algorithm can repeatedly find matching inputs within this number of FUT executions or less, it is clearly superior to random sampling.

VII. RESULTS

For the examples, a total of five nodes were not covered by random test data generation, and were thus submitted to the evolutionary algorithm. For these nodes, measurement data has been captured. The individual targets are identified by the function and block numbers assigned by the DCRTT tool for the nodes to be covered[2].

A. Required Number of Generations

In Figure 6 the cumulative proportion of successful runs over the number of generations for the second measurement campaign is shown. The steep curve for Function 1, Block 5 indicates that random search is actually sufficient given the restricted message size as specified by the algorithm parameters. As a consequence, only results from the other four targets are used in the following.

B. Explanatory Power of the Parameter Models

A large part of variance in execution time can be explained using a linear model of execution counts and maximum input size ($R^2 > 0.47$ for the first and $R^2 > 0.86$ for the second measurement campaign).

The variance of execution count explained by the configuration parameters was lower, with R^2 ranging from 0.18 to 0.27 for the first and from 0.26 to 0.40 for the second measurement campaign. For the logistic regression model of search success, pseudo-R^2 ranged from 0.37 to 0.50 for the first and from 0.62 to 0.83 for the second campaign.

[2] A data package containing inputs and results is available at https://www.bsse.biz/tedaga_eval2020.zip. A free, limited-time evaluation license of DCRTT solely for purposes of replication is available on request. The function and block numbers are kept as reference to the data package.

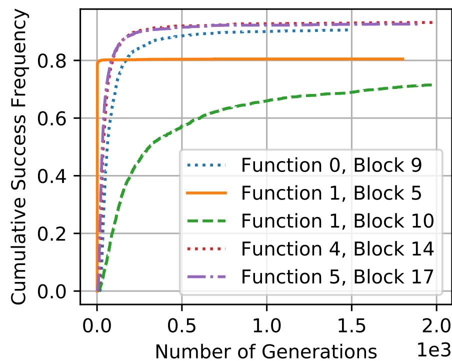

Fig. 6. Success by Generation Count – Second Measurement

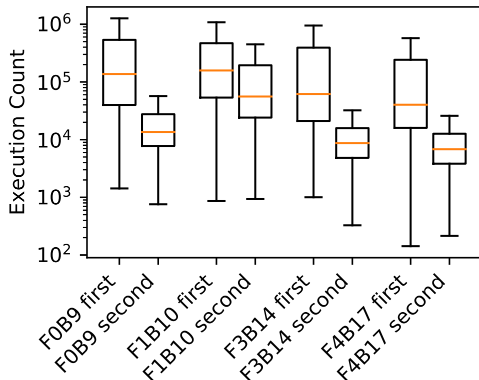

Fig. 7. Comparison of Execution Counts

C. Impact of Hypermutation and Mutation Reversal

Data from the first measurement campaign indicated that mutation reversal has no defined impact on either search success or execution counts. Immigration increases execution count.

Therefore, in the second measurement campaign, mutation reversal and hypermutation by immigration were completely disabled. The box plot in Fig. 7 shows the comparison of execution counts from both campaigns.

D. Parameter Influence on Performance

The statistical execution count model determined the optimum population size to be 0. The impact of population size on execution count ranged from 0.07σ to 0.30σ, where σ is the standard deviation of the execution count for the individual examples. However, as this is not a useful optimum, population size was not considered in optimising execution count.

The impacts, coefficient ranges and optima of parameters for execution count and success probability are shown in Tables III and IV, respectively. The optimal parameter ranges of *elite proportion* and *addition probability* for execution count and *population size* for search success seem large. However, the largest impact size for elite proportion and for population size coincides with their respective

TABLE III
INFLUENCE ON EXECUTION COUNT

Parameter	Impact Range Min	Impact Range Max	Optimum Range Min	Optimum Range Max
Elite Proportion	0.19σ	0.81σ	0.29	0.48
Maximum Individual Size	0.09σ	0.22σ	21	23
Addition Probability	0.03σ	0.29σ	0	0.53
Reduction Probability	0.05σ	0.26σ	0	0
Bit Flip Probability	0.05σ	0.11σ	0.51	0.57
Extension Probability	0.03σ	0.13σ	0.46	0.58

TABLE IV
INFLUENCE ON SUCCESS PROBABILITY

Parameter	Coeff. Range Min	Coeff. Range Max	Optimum Range Min	Optimum Range Max
Maximum Individual Size	12.08	31.41	19	22
Population Size	2.69	5.31	295	500
Elite Proportion	2.35	2.82	0.50	0.59
Addition Probability	1.73	1.73	0.40	0.40

largest value. For addition probability, the smallest value has the highest impact.

VIII. DISCUSSION

The results show that hypermutation by immigration and mutation reversal can be dropped from the algorithm, as the former has an adverse effect and the latter has no effect at all on performance. By deactivating both features, execution counts were reduced by factors between 2 and 10, with the exception of a single target (Fig. 7).

Global optima were identified for some parameters. For example, designating half of the population for an *elite* seems to be a good compromise, with a sizeable impact on both success probability and execution count. Also, there are low-impact optima for extension and bit flip probability.

Maximum individual size is the most important contributor to success probability and ranks second for execution count. In both cases, the optimum is slightly above the largest message size expected by the tested functions. However, this also indicates that execution counts would increase if the size range considered increases. This is consistent with results found by Rawat et al in their analysis of application-aware fuzzing [9].

The optima of *population size* for success probability are less consistent with each other, but increase with larger impact size. This may indicate that there is no general optimum. Clearly, a larger population provides a larger set of samples given a finite limit on the number of generations.

A surprising result is that the optimum for *reduction mutation* seems to be deactivation. As message length can only be changed by reduction and extension mutations and some of the examined targets require exact message lengths, this means that the matches could have only been generated by extending existing smaller individuals.

Similarly, the optimum for *addition mutation* is deactivation as well. Although this mutation can be compen-

sated by the bit flip mutation, the two are not expected to be completely independent of each other.

Block 10 of Function 1 seems to pose a specifically difficult problem for the algorithm. This target can be reached only if the message has a length of exactly 7 bytes, a specific reserved field r of 4 bits widths has the value 0, and another 16-bit unsigned integer field x equals 0. To get to the decision checking x for zero, $x < 100$ must already be satisfied. From then on, any mutation that modifies the 4 bits of r or any of the upper 9 bits of x will set back the respective individual to failing any of the earlier conditions. This seems to be a specifically difficult problem for the algorithm.

After 500 generations none of the examples showed significant progress in terms of success probability. Thus, it seems prudent to use that as a cut-off point to detect unsuccessful runs.

IX. Conclusions and Outlook

We have evaluated the performance impact of several elements of an evolutionary algorithm for coverage-driven test data generation. The analysis shows that several aspects of the algorithm can or should be dropped, improving performance and reducing complexity. Some of these results are surprising and may require further investigation.

One of the examples proved difficult to cover by the algorithm due to a relational operator that works very similar to the equality operator. Thus, the defined cost-functions for relational operators may have to be reconsidered in order to account for such cases.

Our results also reinforced the dependency of search performance on message size. Here, monitoring memory access similar to VUzzer may be useful to guide mutations.

Finally, many of the telecommand processing functions targeted here are very linear in nature, with each step being executed only if the previous ones succeeded. Thus, the usefulness of evolutionary cross-over for these situations may be doubted and should be investigated.

References

[1] J.-L. Lions, L. Lübeck, J.-L. Fauquemergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran, "Ariane 5 flight 501 failure - report by the inquiry board," Tech. Rep., Jul. 1996.

[2] A. Witze, "Software error doomed japanese hitomi spacecraft," *Nature*, vol. 533, pp. 18,19, May 2016.

[3] T. Tolker-Nielsen, "EXOMARS 2016 - Schiaparelli Anomaly Inquiry," Tech. Rep. DG-I/2017/546/TTN, May 2017.

[4] J. S. Newman, "Failure-space - a systems engineering look at 50 space system failures," *Acta Astronautica*, vol. 48, no. 5-12, pp. 517–527, 2001.

[5] A. Gorbenko, V. Kharchenko, O. Tarasyuk, and S. Zasukha, "A study of orbital carrier rocket and spacecraft failures: 2000-2009," *An Int. J. of Inf. & Sec.*, vol. 28, 2012.

[6] G. Gay, M. Staats, M. Whalen, and M. P. E. Heimdahl, "The risks of coverage-directed test case generation," *IEEE Trans. on Softw. Eng.*, vol. 41, no. 8, pp. 803–819, Aug. 2015.

[7] R. Gerlich, R. Gerlich, K. Kvinnesland, B. S. Johansen, and M. Prochazka, "A case study on automated source-code-based testing methods," in *Proc. of DAta Systems in Aerospace*, 2013.

[8] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong, "Code coverage of adaptive random testing," *IEEE Trans. Rel.*, vol. 62, no. 1, pp. 226–237, Mar. 2013.

[9] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proc. 2017 Netw. and Distr. Syst. Sec. Symp.*, 2017.

[10] C. Barret, L. de Moura, and A. Stump, "Design and results of the 2nd annual satisfiability modulo theories competition (SMT-COMP 2006)," *Formal Methods in System Design*, vol. 31, no. 3, pp. 221–239, Dec. 2007.

[11] R. Gerlich and C. R. Prause, "Evaluating test data generation for untyped data structures using genetic algorithms," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Apr. 2018.

[12] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*, J. Marciniak, Ed. Wiley, 1994, pp. 970–978.

[13] T. Y. Chen, D. H. Huang, and F.-C. Kuo, "Adaptive random testing by balancing," in *RT '07: Proceedings of the 2nd international workshop on Random testing*, 2007, pp. 2–9.

[14] B. Korel, "Automated software test data generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[15] A. Gotlieb, B. Botella, and M. Rueher, "A CLP framework for computing structural test data," *Lecture Notes in Computer Science*, vol. 1861, pp. 399–413, 2000.

[16] A. Denise, M.-C. Gaudel, and S.-D. Gouraud, "A generic method for statistical testing," in *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2004, pp. 25–34.

[17] R. Gerlich, "Verallgemeinertes Rahmenwerk zur constraint-basierten Testdatenerzeugung aus Programmflussgraphen," Ph.D. dissertation, Faculty of Engineering and Computer Science, University of Ulm, Germany, 2009.

[18] B. Elkarablieh, P. Godefroid, and M. Y. Levin, "Precise pointer reasoning for dynamic test generation," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 129–140.

[19] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification*. Springer Berlin Heidelberg, 2006, pp. 419–423.

[20] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2005, pp. 213–223.

[21] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 1, pp. 63–86, 1996.

[22] J. Miller, M. Reformat, and H. Zhang, "Automatic test data generation using genetic algorithm and program dependence graphs," *Inf. Softw. Technol.*, vol. 48, pp. 586–605, 2006.

[23] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, 1979.

[24] S. Baluja and R. Caruana, "Removing the genetic from the standard genetic algorithm," in *Proc. of the 12th Int. Conf. on Machine Learning*. Morgan Kaufmann, 1995, pp. 38–46.

[25] J. J. Grefenstette, "Genetic algorithms for changing environments," in *Proceedings of the 2nd International Conference on Parallel Problem Solving from Nature*, 1992, pp. 137–144.

[26] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 6, pp. 2193–2196, Mar. 2012.