Evaluating Test Data Generation for Untyped Data Structures Using Genetic Algorithms

Ralf Gerlich
Dr. Rainer Gerlich System and Software Engineering
Auf dem Ruhbuehl 181
D-88090 Immenstaad
Germany
Email: ralf.gerlich@bsse.biz

Christian R. Prause

Sankt Augustin
Germany
Email: prause@acm.org

***Abstract*—Untyped data such as the byte streams used in communications between spacecraft and ground stations present a specifically challenging field for automatic test data generation. We investigate variations of genetic algorithms to improve test data generation, and present measurements and preliminary results obtained using our prototype. The future goal is to extend our white-box random testing tool DCRTT with these methods and thus apply the approach to industry-grade software.**

I. Introduction

Onboard or flight software that controls spacecraft is thoroughly verified and validated because a single failure or malfunction can have serious adverse effects; in the extreme case, it can lead to the loss of a system, rocket or satellite. Well-known examples may be the first Ariane 5 flight in 1996, or more recently the losses of the Hitomi space telescope [1] or Schiaparelli lander [2]. While Newman [3] reports almost no losses of spacecraft due to software problems before 1995, until 2000 this number jumps up to 13% out of all losses for software design and 30% for software quality assurance problems. Among the 150 failures investigated by Gorbenko et al. [4] in the time from 2000 until 2009, software problems are responsible for 13% (launcher rockets) and 20% (other spacecraft) of failures.

Hence, it is no surprise that there is an entire ecosystem of validation and verification processes to prevent such failures. The European Cooperation for Space Standardisation (ECSS) has two standards for the software discipline: ECSS-E-ST-40C (software engineering) and ECSS-Q-ST-80C (software product assurance). In these two standards, prescriptions for planning and management of software tests alone are more extensive than for any other process area including requirements management, architecture and design, or integration. Automatically generated test data fills a niche in this ecosystem.

Automated testing drills test probes down into the code to find robustness weaknesses which could derail program execution in the field. This is done automatically, i.e., not requiring much human effort. The test generator identifies functions to call, determines their parameters, and generates (random) test data to achieve a high coverage of execution paths and input values. However, telecommands – packets used for commanding a spacecraft from a remote station – are received as an un-typed byte stream. These packets are validated and decoded in a central function and dispatched from there into the subsystems of the spacecraft. A test data generator that is unaware of the telecommands' data structure immediately faces an impractically large exponential search space. Exercising the whole chain of validation and decoding operations using only random data thus becomes an intractable problem.

Random test data generation [5] and its variants [6], [7] are simple to implement, yet ineffective for treating this problem. Constraint-based test data generation [10], [11] at the other end of the spectrum is deemed effective for these cases, but comes at the price of computational and implementation complexity [13]. A middle ground between the two extremes may be heuristic methods such as search-based [8] approaches and genetic algorithms [9].

In this paper, we describe preliminary results evaluating an approach based on genetic algorithms for generating test data for the telecommand validation and decoding stage of spacecraft software. Section II lays out our technical approach and implementation. Section III presents measurements and preliminary results obtained through practical evaluations of our prototype. This guides strategic decisions between different possible variants of the algorithm. Finally, we provide conclusions and an outlook on future work (Section IV).

II. Technical Approach

In a genetic algorithm, principles of evolution are used to evolve a population of n candidate solutions – the *individuals* – towards an optimization goal. Each iteration of the optimization procedure produces a new *generation* of the population.

In our case, the individuals are candidates for test inputs. The *representation* of each individual is given by a byte-string, defined by its contents and its size. In order to

form the initial population, the individuals are initialized with random contents.

We optimise the population so that invoking the function under test with them as input covers a given statement or branch in the code – the coverage goal – for at least a given number of them. For simplicity, we will assume that the respective statement or branch is reachable from the function entry point.

In our approach, the search stage is executed on a host system, allowing us to run the function under test with a large number of test inputs. The actual number of useful test inputs identified during the search stage can be expected to be much lower, so that the actual test can be executed on the target.

A. The Algorithm

For each generation, we first invoke the code under test using the inputs defined by the individuals to identify the cost associated with the individual. The next generation then consists of

- the elite of the previous generation [14],
- a portion of immigrants [15], and
- a portion of offspring generated from the individuals in the current generation.

The *elite* of a population is the set of the n_e best-fitting individuals, where the fraction $\frac{n_e}{n}$ is the *elite proportion*. This way, the best-fitting individuals are preserved.

Immigrants are individuals which are newly initialized with random data. Given the number of immigrants n_i, the ratio $\frac{n_i}{n}$ is called the *immigrant proportion*.

The remaining $n - n_e - n_i$ elements of the next generation are built by cross-over and mutation. For cross-over, two individuals from the current generation are selected as parents. Selection is performed such that for each individual the chance of being selected increases with decreasing cost value.

For cross-over, we cut both of the parents at the same, randomly selected byte index and create a new individual by joining the first part of one parent to the second part of the other parent.

The offspring is then randomly mutated by one or more of the *mutation operators*:

- Cut off the last byte (reduction),
- add a random byte at the end (extension), or
- flip a random bit within the byte-stream

Reduction and extension of size can happen at most once per mutation step. Bit flips are performed in a loop, with the *Bit Flip Probability* giving the probability that the loop continues execution at each iteration. Note that theoretically the same bit may be flipped multiple times during a mutation step.

All three forms of mutations may be reversed with a fixed probability if they increase the cost associated with the individual.

The algorithm is repeated until a pre-defined number of individuals in the population fulfill the coverage goal.

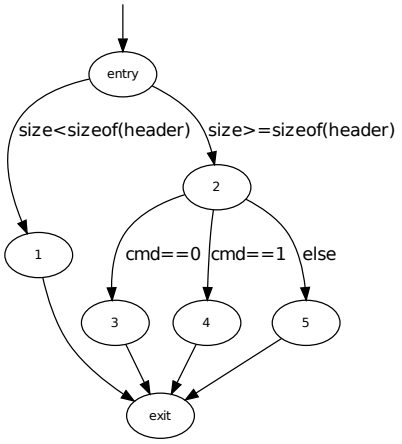

Fig. 1. Control-Flow Graph

TABLE I
Cost Function Definition

Condition	Cost Function
`E op F` ($\text{op} \in \{=, <, >, \leq, \geq\}$)	$\begin{cases} 0 & \text{if } E \text{ op } F \\ \lvert F - E \rvert & \text{otherwise} \end{cases}$
`E != F`	$\begin{cases} 1 & \text{if } E = F \\ 0 & \text{otherwise} \end{cases}$

B. The Cost Function

To determine the value of the cost function for each individual, we instrument the code under test and call it with the byte stream defined by the individual. If the coverage goal is reached, the cost of the individual is zero. Otherwise, there must have been a decision that led execution away from the target code.

Consider the example given in Fig. 1. Here, Node 3 can only be reached if `size` is greater than or equal to `sizeof(header)`. If this is not the case, execution will traverse the edge from the entry node to Node 1, from which Node 3 cannot be reached any more.

To reach Node 3 we would have to change the value of `size` by at least `sizeof(header)-size`. Thus we can use this distance as the cost of the individual.

The cost functions for each of the comparison operators are shown in Tab. I. Similar definitions were already given by Korel [11]. In the example, the condition `size>=sizeof(header)` would have to be fulfilled. The cost function then has value 0. Otherwise, the value of the cost function would be |sizeof(header) − size|.

Conditions of the form `E != F` are a special case. Whenever these are not fulfilled, this means that $E = F$ holds, and thus the value of either E or F has to be changed by at most 1 to fulfill the condition.

C. Use of Intermediate Targets

Consider once again the example in Fig. 1. If we wanted to reach Node 4, a cost value of 1 could result either from the size being off by 1 in the entry node, or from

TABLE II
Parameters and Variation Ranges

Parameter	Base Value	Min	Max
Population Size	600	200	1000
Elite Proportion	0.25	0.2	0.8
Immigrant Proportion	0.10	0.0	0.25
Mutation Reversal Probability	0.8	0.3	1
Byte Extension Probability	0.2	0.0	1.0
Byte Reduction Probability	0.2	0.0	1.0
Bit Flip Probability	0.4	0.0	0.7

a correct size with the value of `cmd` in Node 2 being off by one. Clearly, the second case would be closer to our goal, but this would not be reflected by the cost value. Considering both, the original cost and the distance between the decision and the goal, would require a relative weighting factor between the two.

We instead apply the algorithm to a sequence of intermediate goals before the final goal. To provide a sufficient basis for the next intermediate goal, the search on each goal continues until a given proportion of the population meets that goal. This is the *Sequential Approach*, as opposed to the (original) *Single-Step Approach*.

In order to reach Node 4, we first have to reach Node 2. Node 2 is a *decision node* for Node 4 in that the decision taken in Node 2 influences whether we can reach Node 4: Taking any of the edges to Node 3 or Node 5 means that Node 4 cannot be reached any more. Node 2 also *dominates* Node 4 in that every path from the entry point to Node 4 must traverse Node 2 [16].

In general, some node d is a decision node for a target node t if the latter can be reached from the former, but there is at least one successor of d, from which t cannot be reached. Further, some node d dominates the target node t if and only if every path from the entry point to t traverses d.

For the intermediate goals, we will use the decision nodes for target t which are at the same time its dominators. The dominator property ensures that our choice of intermediate goals does not restrict the set of paths the target t.

Note that the dominator relation in a control flow graph forms a tree with the entry point at the root. This tree also gives us a suitable order in which to consider the intermediate targets, starting at the root of the tree and traversing it towards our final goal.

III. Evaluation and Results

We evaluated the impact of each of the parameters of the algorithm on its performance by measuring the runtime of the algorithm in a series of experiments. All measurements were executed on the same hardware (a typical off-the-shelf PC), so that the time values can be considered in relation to each other.

In a first step, we compared the execution times of the Sequential Approach to the Single-Step Approach. Both

```
tc_error_t process_tc(void* data, size_t size) {
  tc_header_t* header = data;
  if (size<sizeof(tc_header_t))
    return tc_error_invalid_size;
  switch (header->type) {
  case tc_set_log_parms: {
    tc_set_log_parms_t* tc = data;
    if (size!=sizeof(tc_set_log_parms_t))
      return tc_error_invalid_size;
    if (tc->res!=0 || tc->frq<1 || tc->frq>100)
      return tc_error_invalid_param;
    return process_set_log_parms(tc);
  } /* ... */
  default: return tc_error_invalid_type;
  }
}
```

Fig. 2. Example Code: Telecommand Handling (excerpt)

TABLE III
Execution Time Statistics

Variant	Min (s)	Mean (s)	Max (s)
Sequential	0.161	2.595	15.931
Single-Step	0.268	15.553	146.180

approaches were configured with the same set of configuration parameters and run 400 times each. In a second step, we applied the Sequential Approach repeatedly, randomly varying a single parameter within a given range. For each parameter, 2000 different parameter values were applied. Both, the base value and the variation range for each of the parameters, are shown in Tab. II. For some of these parameters not their full range was evaluated, as exceeding the range given here has lead to excessive execution times.

For our preliminary measurements, we used a simple example modelling the structure of code typically employed in flight software to validate incoming telecommands (Fig. 2[1]). The function verifies that the packet is large enough to contain the header, which includes the command type field. Depending on the command, different parameters in the packet are validated, depending on the command type. If any of these parameters is found to be out of range, the function is left prematurely.

For all of our measurements the goal was to pass all telecommand validation checks. Variants failing specific validation checks were not considered.

The results of the comparison between the Single-Step and the Sequential Approach are shown in Tab. III. The Single-Step Approach has an approximately 6-fold mean execution time compared to the Sequential Approach, with the maximum execution times differing by a factor of 9.

As the immigrant proportion increases, the algorithm deteriorates towards pure random sampling, which may explain the increase in execution time (Fig. 3). However,

[1] http://www.bsse.biz/processtc.zip

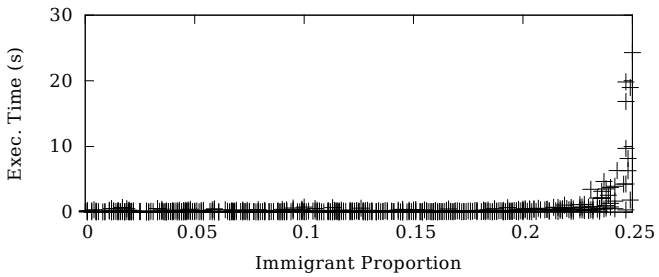

Fig. 3. Impact of Immigrant Proportion

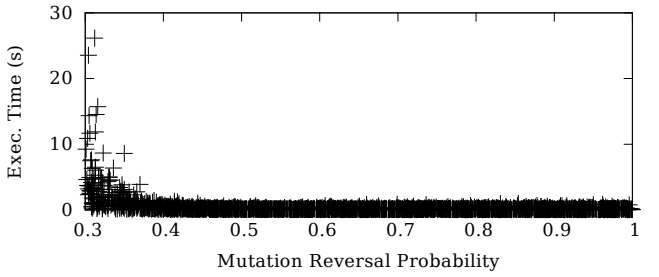

Fig. 4. Impact of Mutation Reversal

TABLE IV
Impact of Byte Extension

Probability	0.00	0.25	0.50	0.75	1.00
Mean Exec. Time (s)	0.46	0.13	0.12	0.12	0.13

lowering the immigrant proportion towards 0 does not have a negative impact on execution time. Contrary to that, Fig. 4 shows that increasing the probability for mutation reversal seems to have a profound positive impact on execution time.

For the size-changing mutation operators, there seems to be an optimum value of $0.5 \lessapprox p \lessapprox 0.75$, as indicated by Tab. IV. We see that the mean execution time decreases by more than a factor of 3 between the values 0 and 0.25, but for $p \to 1$ there is no further noticeable increase.

IV. Conclusions and Future Work

From the results, several conclusions regarding the design of the actual implementation can be drawn:

- The Sequential Approach seems to be superior to the Single-Step Approach regarding execution time.
- Mutation Reversal has a significant positive impact on execution time.
- Immigration seems to have a negative effect on execution time when its proportion becomes $\frac{n_i}{n} \gtrapprox 0.2$. We did not observe further positive effects of lower immigration values.

As a consequence, the Sequential Approach and Mutation Reversal should be implemented, while Immigration can be disregarded.

Our work so far has shown that genetic algorithms are applicable to test data generation for telecommands, and has provided us with some insights into suitable variations of the approach.

We have already started integrating the approach with our random testing framework DCRTT [17], which will allow us to perform experiments on industry-grade code from actual flight software.

V. Acknowledgements

This work is supported by a grant from the German Federal Ministry for Economic Affairs and Energy, based on a decision of the German Bundestag, grant No. 50PS1601.

References

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

[2] T. Tolker-Nielsen, "EXOMARS 2016 - Schiaparelli Anomaly Inquiry," European Space Agency, Tech. Rep. DG-I/2017/546/TTN, May 2017. [Online]. Available: http://exploration.esa.int/mars/59176-exomars-2016-schiaparelli-anomaly-inquiry/

[3] 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.

[4] A. Gorbenko, V. Kharchenko, O. Tarasyuk, and S. Zasukha, "A study of orbital carrier rocket and spacecraft failures: 2000-2009," *An International Journal of Information & Security*, vol. 28, 2012.

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

[6] 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*. ACM, 2007, pp. 2–9.

[7] 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.

[8] 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.

[9] J. Miller, M. Reformat, and H. Zhang, "Automatic test data generation using genetic algorithm and program dependence graphs," *Information and Software Technology*, vol. 48, pp. 586–605, 2006.

[10] 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.

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

[12] R. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 206–215, Dezember 1990.

[13] C. Barret, L. de Moura, and A. Stump, "Design and results of the first satisfiability modulo theories competition (SMT-COMP 2005)," *Journal of Automated Reasoning*, vol. 35, no. 4, pp. 373–390, November 2005.

[14] S. Baluja and R. Caruana, "Removing the genetic from the standard genetic algorithm," in *Proceedings of the 12th International Conference on Machine Learning*. Morgan Kaufmann, 1995, pp. 38–46.

[15] 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.

[16] 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.

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