Generating Random Telecommand Test Data Using Genetic Algorithms

Ralf Gerlich, Rainer Gerlich
Dr. Rainer Gerlich BSSE System and Software Engineering
Immenstaad, Germany
e-mail: Ralf.Gerlich@bsse.biz, Rainer.Gerlich@bsse.biz

***Abstract*—Generating useful test data is one of the big challenges in automatic software testing. While random test data generation is the easiest method, the test inputs generated by it may fail to exercise the software under test properly if the internal structure of the data is unknown to the generator and at the same time relevant for the decisions taken in the code. Handling of telecommands in space onboard software is one example where this is the case. We investigate a method of generating test data for these cases using genetic algorithms.**

Keywords: automated software test, test data generation, telecommand interface, software defects, defect identification, software verification

I. INTRODUCTION

Software testing can be used to assess the software in a representative execution environment. Even though it is impossible to prove the absence of defects, it can reveal existing defects simply by executing the software with pre-determined inputs and checking for the desired results. Given a sufficient number of such test cases, not detecting further faults can lead to a sufficient level of confidence in the correctness of the software itself.

This sufficient number of test cases, however, is very difficult to achieve if test cases are selected manually. The associated effort is typically prohibitive. Thus, automatic methods for test data generation will help to overcome this constraint.

Random test data generation lends itself to quick generation of large, generally unbiased sets of test inputs or even test cases[1][2] and can be easily used for detecting basic defects (fuzzing)[3][4].

However, large portions of randomly generated data typically represent invalid inputs for the software under test, even more so if structural information about the input data is not available. This is specifically the case when generating test data for telecommand handling code received as pure byte stream.

At this point, telecommands are passed between hardware and software in the form of sequences of bytes, and there is no structured type information associated with the data within the code at the point of reception. Random testing here would heavily exercise the validation code in terms of robustness testing, generating mostly invalid command packets, while leaving the actual functionality mostly untested.

Constraint-based[5][6] approaches may lend themselves to generation of appropriate data for these cases, but they come at the price of computational, but also general complexity[7][8].

A middle ground between these extremes may be occupied by heuristic methods such as genetic algorithms[9]. We present our approach to practical evaluation of such methods in the specific context of generating test data for telecommand processing code.

The paper is structured as follows: After this introduction, we will lay out the approach considered so far. This will be followed by a brief presentation of measurement data guiding strategic decisions between different variants. Finally we will provide intermediate conclusions and an outlook on our future work.

II. APPROACH

Our approach uses classical genetic algorithms, extended by elitism[11], immigration[12] and directed mutation. The genome of each individual is represented by a byte-string, initially filled with random data. The length of each byte-string is chosen randomly from a configurable range.

We aim at fulfilling structural coverage, either in the form of statement or of condition/decision coverage. While structural code coverage by itself is not a sufficient measure of usefulness for test data selection[10], at minimum achieving full coverage according to these criteria is a necessary condition for detecting faults. After all, a fault contained in code that is never executed during test will not be detected by the test.

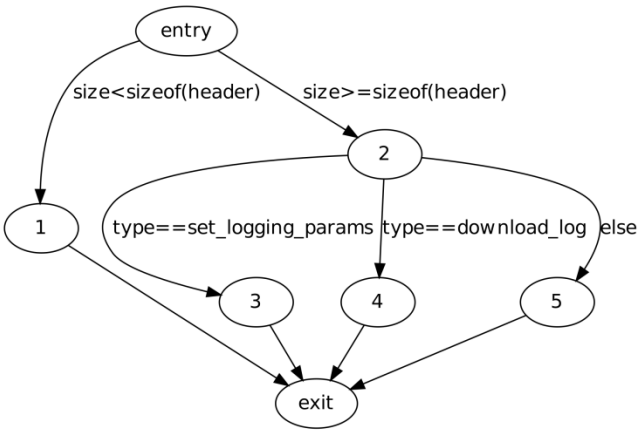

Fig. I-1 Control-Flow Graph

Also, for the application case we are targeting – telecommand processing – the structure of the validation and processing code usually represents different forms and variants of telecommand contents. Decisions taken during validation will separate the possible set of inputs into those packets seemed valid by the implementation and those that are not. This can be seen in the example given by Fig. I-1, where decisions are taken based on the size of the packet and

the contents of specific fields within the packet header (here: the packet type).

Thus, coverage of the code should also imply close coverage of the different kinds and structures of telecommands.

However, genetic algorithms are still heavily based on random data and thus the set of solution candidates can still be expected to exhibit sufficient variance wherever the conditions imposed on the solution allow for that.

The algorithm is thus not bound to provide only the most straight-forward inputs that happen to fulfil the coverage criteria. Instead, the bias usually expected from manual, coverage-driven test data selection can be counteracted by randomly selecting one or even more than one element from the provided set of matching inputs.

A. General Principles of Genetic Algorithms

Genetic algorithms apply the principles of evolution to optimize a set of candidate solutions – the *individuals* forming a *population* – towards a specific optimisation goal. In each iteration of the algorithm, a new generation of individuals is produced from the current generation using several genetic operators, most specifically cross-over and mutation.

For cross-over, two individuals from the current generation are selected and recombined into a new individual, similar to how in nature two individuals create offspring that carries genetic information combining parts of the genetic information of its parents.

Individuals better adapted to their environment have a higher chance of participating in pro-creation than their less-adapted peers. The idea is that the combination of genetic traits of well-adapted parents will lead to well-, and possibly even better-adapted children.

In case of optimisation, the level of adaptation to the environment – or fitness – of an individual is defined by how well the individual solves the optimisation problem.

Mutation operators randomly modify the individuals after recombination, aiming to keep the variance in the population sufficiently high so that new solutions can be found on recombination.

B. Elitism

In classical genetic algorithms, each generation is completely replaced by its succeeding generation. However, in nature, well-adapted individuals also tend to survive longer than their less-adapted counterparts. This may also allow them to produce more offspring, and thus benefit the population as a whole by passing their good genetic traits on to more children.

Elitism[11] is a modification to classical genetic algorithms which aims to model this survival by copying a certain portion of the population of pre-defined size to the next generation, namely the best-adapted individuals in the population – the *elite*.

C. Cross-Over and Mutation Operations

We employ single-point cross-over: The individuals are cut at a common, randomly selected cross-over point and two new individuals are generated by swapping the ends of the byte-strings.

Three different types of mutation are possible:

- Cutting off the last byte,
- add a random byte at the end, or
- flipping a random bit within the byte-stream.

Reduction and extension of size can happen at most once per mutation step, while multiple bit flips are possible. The number of bit flips performed during a single mutation step is chosen randomly, with $(X = N) = p^N\ (1 - p)$ giving the probability of N bit flips occurring in a row. Note that theoretically the same bit may be flipped multiple times during a mutation step.

D. Measuring Fitness

Different from usual genetic algorithms, we apply a cost function instead of a fitness function: Low cost corresponds to high fitness and vice versa.

The cost of an individual shall express how far the individual is away from reaching the selected coverage target. It is determined by executing the telecommand handling procedure on it and monitoring the control flow decisions taken.

For illustration, consider the control-flow graph shown in Fig. 1. To reach Node 4 from the entry point, execution must traverse the edge from the entry point to Node 2, and then the edge from Node 2 to Node 4.

Let us assume that – different from what is desired – execution proceeds from the entry node to Node 1, from where we cannot reach Node 4 anymore.

The branch taken implies that `size<sizeof(header)` is fulfilled. To change that decision, we would have to change the value of size by at least $|size - sizeof(header)|$. Thus, this value can be considered to be the distance between the current input and one that would fulfil one more of our requirements.

We can define cost functions for other relations between expressions E and F evaluated at a node as shown in Tab. II-1[13][14].

In summary, the value of the cost function is determined by executing the function under test on the respective individual. The code is instrumented such that whenever a branch is taken, the cost function is updated. If the target point can still be reached, the cost function remains unchanged. If the target point cannot be reached anymore, the value of the cost function is set to the value of the respective expression given inTab. II-1.

Condition	Cost Function
E==F	$\lvert E - F \rvert$
E!=F	$\begin{cases} 1 & if\ E = F \\ 0 & otherwise \end{cases}$
E<F, E>F	$\lvert E - F \rvert + 1$
E<=F, E>=F	$\lvert E - F \rvert$

Tab. II-1 Cost Function Definition

E. Mutation Reversal

All three forms of mutations are directed in that the cost function is re-evaluated after an individual mutation, and if the cost after mutation is higher than before the mutation, the mutation is reversed at random with a fixed probability.

This is similar to a probabilistic variant of gradient descent optimisation. However, while gradient descent tries to determine the direction of the local gradient and optimises deterministically, e.g. using Newton's method for finding zeroes, here a random change is applied and taken back probabilistically if the change does not enhance the solution.

Thus, with a given, finite probability, a solution may also be modified in a direction that – at least from a local point of view – diminishes the value of the solution. Thereby the deterministic descent into local, non-global optima can be avoided.

F. Algorithm Variants

Consider once again the example in Fig. I-1. If we wanted to reach Node 4, a cost value of 1 would not indicate whether execution deviated at the entry node or in Node 2. We would favour the second case, as it is closer to our goal, but we would not be able to determine that from the cost value. Adding that information to the cost function would require us to introduce a relative weighting factor for the distance from the target, but we have no basis for choosing a useful value for that weighting factor.

Instead, we select a sequence of intermediate goals before the final goal, and apply the algorithm to all goals in sequence. Before considering the next goal in the sequence, a sufficiently large portion of the population must fulfil our current goal. We call this modification the Sequential Approach, while the original approach shall be referred to as the 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[15].

Thus we select our intermediate nodes from the nodes dominating our target node while at the same time containing decisions that influence whether we can actually reach our final target node.

G. Immigration

In the Sequential Approach, the switch from one intermediate goal to the next can be compared to an abrupt change in environment in terms of evolution. This change may require additional variance within the population, which can be introduced by filling a portion of the next generation of pre-defined size with new random individuals[12]. This process is similar to the influx of new individuals from outside the current domain of the population, i.e. immigration.

```
tc_error_t proc_tc(void* data,
                   size_t sz)
{
  if (sz<sizeof(tc_header_t))
    return tc_error_invalid_size;
  else {
    const     tc_header_t*     header     =
        (const tc_header_t*)data;
    switch (header->type) {
    case tc_set_logging_params:
      return proc_set_log_para(data,sz);
    case tc_download_log:
      return proc_dl_log_tc(data,sz);
    default:
      return tc_error_invalid_type;
  }
}

tc_error_t proc_set_log_para (void * data,
                              size_t sz)
{
  if (sz!= sizeof(tc_set_logging_params_t))
    return tc_error_invalid_size;
  else {
    const tc_set_logging_params_t* tc =
      (const tc_set_logging_params_t*)data;
    if (tc->reserved!=0)
      return tc_error_invalid_param;
    else if (tc->frequency>100)
      return tc_error_invalid_param;
    else if (tc->frequency<1)
      return tc_error_invalid_param;
    else
      return tc_ok;
  }
}

tc_error_t proc_dl_log_tc (char* data,
                           size_t size)
{
  if (size!= sizeof(tc_download_log_t))
    return tc_error_invalid_size;
  else
    return tc_ok;
}
```

Lst. III-1 Example Code

III. MEASUREMENTS

The algorithm described in Section II has seven important parameters which can be modified and which can impact performance:

- Population size,
- Proportion of population kept as elite,
- Proportion of population filled by immigration,
- Probability for byte extension mutation,
- Probability for byte reduction mutation,
- Probability for bit flip, and
- Probability for mutaton reversal.

In addition, we have the option of using one of two variants, the single-step or the sequential method.

Although it is possible to derive predictions for the impact of these parameters from theory for *corner cases*, their impact in intermediate ranges is not that straight forward. For example, we can determine that using 100% of the population as elite will lead to stagnation, but a prediction for a elite proportion of, e.g., 50% is more difficult to make.

Thus we need to measure the impact of these parameters on observables such as total runtime or the number of generations needed until a solution is found.

We performed measurements on an Laptop PC with an Intel®Core™i7-2630M CPU at 2.0GHz with 6GB RAM. The operating system was Debian GNU/Linux 7.11.

As reference for the first experiments the code in Lst. III-1 was used, which contains basic handling code for simple telecommands.

For comparing single-step vs. the sequential approach, we executed each form 400 times for a single example each, capturing the runtime of each execution. Otherwise the parameters for both variants were the same.

Minimum, mean and maximum execution times for each of the variants are given in Tab. 2. The measurements show that the single-step variant has an approximately 6-fold mean execution time compared to the sequential variant, with the maximum execution times differing by about a factor of 9. Therefore the sequential approach seems to be at a clear advantage.

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

Tab. III-1 Execution Time Statistics

IV. Conclusions and Future Work

Our work so far has shown that genetic algorithms are a feasible approach to the generation of test data for code processing untyped byte-streams, specifically telecommand handling code, and has provided us with some insights into suitable variations of the pure approach. The experiments also indicate that the calculation of the fitness function can be done by basic instrumentation of the code under test.

In the meantime we have proceeded to integrate the approach with our random testing framework DCRTT[4], with the goal of performing experiments on industry-grade from actual space software. The integration is not yet completed, although we were able to perform some first simple tests.

Further investigations will also consider multi-factorial analysis of the impact of parameter values on the performance.

One important aspect of research will be the dependency of optimum parameter values on the respective software to be tested. In theory, it is possible that different variants of implementations of telecommand handling may require different parameter sets for optimal runtime of the genetic algorithm. The question to be answered is whether these optimum parameter sets differ significantly from each other, or whether there is a basic parameter set that is good enough for practical use.

Acknowledgment

The research presented here is supported by grant DLR-50PS1601 of the Space Administration of the German Aerospace Center (DLR) on behalf of the German Ministry of Economics and Energy (BMWi).

References

[1] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, J. Marciniak (Ed.). Wiley, 970–978, 1994.

[2] T. Y. Chen, De Hao Huang, and F.-C. Kuo. Adaptive random testing by balancing. In *RT '07: Proceedings of the 2nd international workshop on Random testing*. ACM, 2–9, 2007.

[3] Miller, B. P.; Fredriksen, L. & So, B. An Empirical Study of the Reliability of UNIX Utilities, *Communications of the ACM*, Vol. 33, No. 12, pp. 32-44, 1990.

[4] Ralf Gerlich, Rainer Gerlich, Marek Prochazka, Kenneth Kvinnesland, Bengt Solheimdal Johansen. A Case Study on Automated Source-Code-Based Testing Methods. In *Proceedings of the DAta Systems In Aerospace Conference 2013* (DASIA 2013), 2013.

[5] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ACM, 213–223, 2005.

[6] Ralf Gerlich. Verallgemeinertes Rahmenwerk zur constraintbasierten Testdatenerzeugung aus Programmflussgraphen. Dissertation, University of Ulm, 2009.

[7] Clark Barret, Leonardo de Moura, and Aaron Stump. Design and Results of the First Satisfiability Modulo Theories Competition (SMT-COMP 2005). *Journal of Automated Reasoning 35, 4 (November 2005)*, 373–390, 2005.

[8] David R. Cok, David D´eharbe, and Tjark Weber. The 2014 SMT Competition. Journal on Satisfiability, Boolean Modeling and Computation 9 (2014), 207–242, 2014.

[9] James Miller, Marek Reformat, and Howard Zhang. Automatic test data generation using genetic algorithm and program dependence graphs. Information and Software Technology 48 (2006), 586–605, 2006.

[10] Richard Hamlet and Ross Taylor. Partition Testing does not inspire confidence. *IEEE Transactions on Software Engineering* 16, 12 (Dezember 1990), 206–215, 1990.

[11] Shumett Baluja and Rich Caruana. Removing the genetic from the standard genetic algorithm. In *Proceedings of the 12th International Conference on Machine Learning*. Morgan Kaufmann, 38–46, 1995.

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

[13] B. Korel. 1990. Automated Software Test Data Generation. *IEEE Trans. Softw. Eng.* 16, 8 (1990), 870–879, 1990.

[14] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.* 5, 1 (1996), 63–86, 1996.

[15] 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.