A Genetic Algorithm for Automated Test Generation for Satellite On-board Image Processing Applications

Ulrike Witteck[1], Denis Grießbach[1] and Paula Herber[2]
[1]*Institute of Optical Sensor Systems, German Aerospace Center (DLR), Berlin-Adlershof, Germany*
[2]*Embedded Systems Group, University of Münster, Germany*

Keywords: Image Processing, Software Testing, Genetic Algorithms.

Abstract: Satellite on-board image processing technologies are subject to extremely strict requirements with respect to reliability and accuracy in hard real-time. In this paper, we address the problem of automatically selecting test cases that are specifically tailored to provoke mission-critical behavior of satellite on-board image processing applications. Because such applications possess large input domains, it is infeasible to exhaustively execute all possible test cases. In particular, because of their complex computations, it is difficult to find specific test cases that provoke mission-critical behavior. To overcome this problem, we define a test approach that is based on a genetic algorithm. The goal is to automatically generate test cases that provoke worst case execution times and inaccurate results of the satellite on-board image processing application. For this purpose, we define a two-criteria fitness function that is novel in the satellite domain. We show the efficiency of our test approach on experimental results from the Fine Guidance System of the ESA medium-class mission PLATO.

1 INTRODUCTION

In the satellite domain, on-board image processing applications are subject to extremely strict requirements especially with regard to reliability and accuracy in hard real-time. It is important to test such applications extensively. But their huge input domain makes manual testing error-prone and time-consuming. Further, executing all possible test cases is impossible.

Therefore, we are interested in a test approach that automatically and systematically generates test cases for testing satellite on-board image processing applications. However, the automated test generation for on-board image processing applications poses a major challenge: due to complex algorithmic computations it is difficult to select test cases with a high probability to provoke mission-critical behavior. Mission-critical behavior means scenarios where, for example, the real-time behavior of the system or the delivered mathematical accuracy does not meet specified requirements. Such scenarios may cause system failures, damages, or unexpected behavior during mission lifetime.

In (Sthamer et al., 2001; Wegener and Mueller, 2001; Varshney and Mehrotra, 2014; Hänsel et al., 2011), various automated test approaches for several real-time embedded systems in different domains are presented. The authors investigate systems with huge input domains and complex functional-behavior. However, the presented approaches are not designed to search for test cases provoking real-time critical behavior and scenarios where the mathematical accuracy of the application gets critically low.

In this paper, we present a genetic algorithm based approach to automatically generate test cases that provoke mission-critical behavior of the system. It is based on the master thesis of the first author (Witteck, 2018). With this approach, we aim for an improvement of a given test suite to support robustness testing. In general, a genetic algorithm solves search or optimization problems by applying evolutionary mechanisms. It evaluates solutions with respect to given criteria using a fitness function and improves the best solutions to satisfy these criteria (Moheb R. Girgis, 2005).

For our proposed test approach, we define a novel two-criteria fitness function based on real-time behavior and mathematical accuracy provided by a satellite on-board image processing application. With this fitness function, our genetic algorithm automatically steers the search towards test cases that provoke long execution times and mathematically inaccurate results.

We use the Fine Guidance System (FGS) algorithm of the European Space Agency (ESA) mission PLAnetary Transits and Oscillation of stars (PLATO) as a case study to investigate the efficiency of our test approach.

Witteck, U., Grießbach, D. and Herber, P.
A Genetic Algorithm for Automated Test Generation for Satellite On-board Image Processing Applications.
DOI: 10.5220/0009821101280135
In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT 2020)*, pages 128-135
ISBN: 978-989-758-443-5

The FGS algorithm is a satellite on-board image processing algorithm that calculates high-precision attitude data of the spacecraft by comparing tracked star positions in image frames taken on board with known star positions from a star catalog. The experimental results show the efficiency of the genetic approach in terms of the automated search of specific test cases tailored for robustness testing.

This paper is structured as follows: Section 2 describes the concept of genetic algorithms in general. Furthermore, it gives an introduction of the PLATO mission and the PLATO FGS algorithm. Section 3 outlines related work on the use of genetic algorithms for test case generation. Section 4 presents our genetic algorithm. First, it provides our description of the algorithm components. Then, it gives an overview of our automated test case generation approach. Our implementation and experimental results are presented in Section 5. Finally, Section 6 provides a summary of the main results and gives an outlook on future work.

2 PRELIMINARIES

In this section, we introduce the concept of genetic algorithms in general and give an overview of the PLATO mission as well as its mission-critical FGS algorithm.

2.1 Genetic Algorithms

Manual test case generation for embedded software tests is often error-prone and inefficient. Especially, the large the number of input parameter combinations makes manual testing expensive. A solution to this problem is an automated test approach, designed to search for test cases specifically tailored to provoke erroneous behavior, i.e. the violation of given system requirements. A promising approach is based on genetic algorithms. The test case design is thus an optimization problem: the genetic algorithm searches for parameter combinations that satisfy given test criteria.

In general, a genetic algorithm is a search-based method that solves complex optimization problems. The approach evaluates automatically generated optimization parameters with respect to predefined test criteria using a cost function. It is an efficient method which rapidly delivers high-quality solutions to a problem (Alander and Mantere, 1999; Sharma et al., 2016).

Genetic algorithms are inspired by the concept of biological evolution. The solutions to a problem experience evolutionary mechanisms like selection, mutation, and recombination. In terms of genetic algorithms, a solution to a problem is considered as an individual. It consists of a specified number of genes. The algorithm uses the cost function to assign a fitness value to each individual as a measure of their quality with respect to specified criteria. In each generation, the genetic algorithm creates a population of individuals from previously created individuals. This is done until a population satisfies a certain criterion. The fitness value is decisive for the survival probability of an individual and therefore for the selection into the next generation. The selection strategy affects the convergence of the genetic algorithm. A too high convergence is a common problem. In that case, the algorithm delivers a locally optimal solution. However, solutions do not evolve if the convergence is too low. To generate new individuals the genetic algorithm applies crossover and mutation operators. The goal of the crossover operator is to generate a better population by exchanging genes from fitter individuals. The mutation operator preserves the diversity of genes by inserting new genes into the population (Sharma et al., 2016; Moheb R. Girgis, 2005; Gerdes et al., 2004).

An advantage of genetic algorithms is the possibility to run on parallel processors. They solve different complex, computation intensive problems, with many possible solutions in a wide search-space. It is possible to automatically search in a huge input domain for optimal test data that provoke a specified behavior of the software application (Alander and Mantere, 1999; Moheb R. Girgis, 2005; Gerdes et al., 2004).

2.2 Case Study: PLATO Mission

PLATO is an ESA mission in the long-term space scientific program “cosmic vision”. The main mission goal is to find and characterize Earth-like exoplanets orbiting in the habitable zone of solar-type stars.

Its scientific objective is achieved by long uninterrupted ultra-high precision photometric monitoring of large samples of bright stars. This requires a very large Field of View (FoV) as well as a low noise level. To achieve a high pupil size and the required FoV the instrument contains 26 telescopes for star observation. 24 normal cameras monitor stars fainter than magnitude 8 at a cycle time of 25 s. Two fast cameras observe stars brighter than magnitude 8 at a cycle time of 2.5 s. The cameras are equipped with four Charge Coupled Devices (CCDs) in the focal plane, each with 4510×4510 pixels. Each fast camera comes with a data processing unit that runs the FGS algorithm. The algorithm calculates attitude data with an accuracy of milliarcseconds from the CCD image data.

In each cycle, the FGS reads a 6×6 pixel window for each guide star from a full CCD-image. Guide stars are predefined stars in a star catalog that satisfy given

criteria. A linear center of mass calculation estimates the initial centroid position in each window. To get a more precise solution, the FGS algorithm refines each centroid using a Gaussian Point Spread Function (PSF) observation model. The PSF describes the distribution of star light over CCD pixels. Based on measured pixel intensities, the algorithm determines the PSF model including: centroid position, intensity, image background and PSF-width. A non-linear least square fitting method iteratively refines the parameters.

The input star signal affects the quality of the centroid calculation. If the star signal in a pixel is little interfered by noise and the signal-to-noise ratio is high, the star information is usable. The distribution of the star signal over pixels depends on the star position on the Focal Plane Assembly (FPA), the sub-pixel position, the magnitude and the PSF shape. If the star signal in the window pixels is not sufficiently good, then the centroid estimation is less accurate or the algorithm does not converge or converges late.

The FGS algorithm transforms the pixel coordinates of the calculated centroid position into a star direction vector. From at least two star directions and the corresponding reference vectors from a star catalog, the algorithm calculates the attitude by means of the QUaternion ESTimator (QUEST) algorithm. Within the QUEST algorithm, the scalar TASTE test measures the validity of the input data (Shuster, 2008). The TASTE value is high, if an input star is misidentified (Griebach, 2020). We use the value as a qualitative measure of the mathematical accuracy of the FGS algorithm and denote it as quality index.

The input of the FGS algorithm is a combination of stars. Since the star parameters of a single star affect the performance and accuracy of the centroid calculation, the performance and accuracy of the FGS algorithm depends on the combination of input stars. If the FGS result is incorrect or the delivery is too late, then the attitude data is unusable. In this case, all captured science data cannot be further processed and the mission is lost. Hence, the FGS is regarded as mission-critical component, which therefore requires an extensive test procedure (Pertenais, 2019).

3 RELATED WORK

Various papers describe automated software test methods which use genetic algorithms. In (Varshney and Mehrotra, 2014; Sthamer et al., 2001; Hänsel et al., 2011), the authors used genetic algorithms to automatically generate test data for structural-oriented tests, like control flow testing and data flow testing. Function-oriented tests, for example examining the temporal behavior of an application are shown in (Sthamer et al., 2001; Wegener and Mueller, 2001).

Genetic algorithms for structural testing are used in (Varshney and Mehrotra, 2014). Their algorithm uses data flow dependencies of a program to automatically optimize test data. The study shows that genetic algorithms are feasible to generate test data that achieve high coverage of variable definition and reference paths in the program code. Moreover, the study shows that data generated by the genetic algorithm achieves higher coverage of the program flow graph in fewer generations than data generated by random testing. However, we look for a test approach that does not depend on the internal system structure.

In (Sthamer et al., 2001), the authors used an evolutionary approach to investigate the temporal behavior of embedded systems. Their approach automatically searches for input situations where the system under test violates specified timing constraints. Their fitness function is based on the execution time. Sthamer et al. used an engine control system as a case study. The experiments show that the evolutionary approach generates test data that detect errors in the timing behavior of systems with large input domain and strict timing constraints. The study shows that the evolutionary approach is applicable to different test goals as well as for testing systems of various application fields. But, our goal is to consider temporal behavior as well as mathematical accuracy of image processing applications for various input values. We define a fitness function that includes additional metrics to evaluate the individuals.

All of these approaches show that genetic algorithms improve the software test efficiency. The studies confirm that genetic algorithms are suitable to automatically generate test cases that satisfy special test criteria from a wide input domain. However, the fitness function must be adapted to the specific problem.

4 GENETIC TEST APPROACH

Many satellite on-board image processing applications perform complex algorithmic computations. Such computations make it hard to find test cases that are tailored to provoke real-time critical behavior or scenarios where the mathematical accuracy gets critically low. But, such test cases are necessary to verify compliance with strict requirements of satellite on-board image processing applications in reliability and mathematical accuracy in hard real-time.

To overcome this problem, we define a test approach based on a genetic algorithm that automatically searches for test cases to increase the robustness of a given system. Our key idea is a novel two-criteria fit-

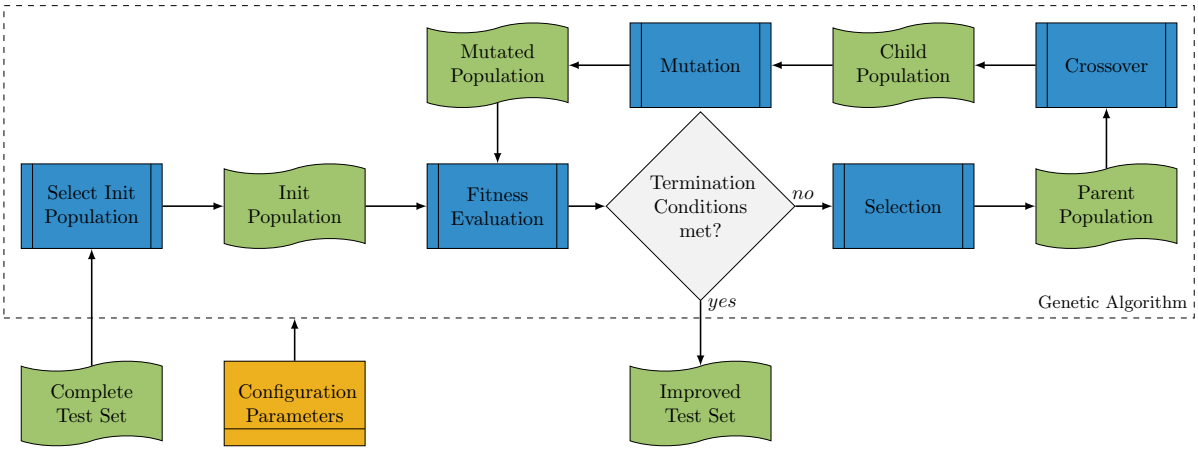

Figure 1: Overview of the automated test case generation approach.

ness function that is specifically tailored for the domain of satellite on-board image processing application.

Figure 1 gives an overview of our proposed approach. As the figure depicts, input of our genetic algorithm (see Section 4.2) is a test set with complete coverage on the input domain and a parameter specification to configure the genetic approach. To calculate a complete test set, we have used an approach for equivalence class testing of on-board satellite image processing applications presented in (Witteck et al., 2019). Their presented approach partitions all input parameters of the FGS algorithm into equivalence classes and systematically selects representative inputs from each partition. Our genetic algorithm selects test cases from the complete test set and evaluates them according to their fitness values. It iteratively evolves promising test cases using evolutionary mechanisms, namely selection, crossover, and mutation. As a result, it delivers test cases that satisfy given test criteria.

4.1 Assumptions and Limitations

We consider systems whose inputs are objects in an image. In our case study, the observed objects are stars uniformly distributed in the image (Griebach, 2020). Performance and mathematical accuracy of the FGS algorithm depend on the number and distribution of preselected guide stars. We specify a test case as a combination of 30 stars, since previous experiments have shown that 30 input stars provide sufficiently good results. In our test approach, we take four star parameters into account that affect run time and mathematical accuracy of the FGS algorithm: position in the image, magnitude, sub-pixel position and PSF shape.

A test set consists of several stars. We denote a test set as complete if it reaches full coverage on the input domain with respect to the coverage criteria defined in (Witteck et al., 2019). Thus the set includes one star for each equivalence class combination.

In our test approach, we use the TASTE-value as a qualitative measure of the mathematical accuracy of the FGS algorithm. Hence, a low quality index corresponds to a high accuracy of the FGS algorithm.

4.2 Genetic Algorithm

We use a genetic algorithm to automatically search for test cases in a given test set that provoke mission-critical behavior with respect to run time and mathematical accuracy. In the following, we describe the components and strategies of our genetic approach.

Individual Representation. In terms of our genetic algorithm, a test case represents an individual with 30 genes, analogous a test case with 30 stars. Our individual representation is based on the equivalence class definitions described in (Witteck et al., 2019). We define a gene as a tuple of equivalence class identifiers (i_P, i_M, i_E, i_G) where P defines the position of the star in the image, M the magnitude of the star, E the sub-pixel position, and G the PSF shape.

Initial Population. Our genetic algorithm uses a complete test set as search space. For each individual, the algorithm randomly selects 30 stars from this space. Each selected star covers a different combination of equivalence classes. The tester specifies the population size and the genetic algorithm generates individuals until the required population size is reached.

Fitness Function. To evaluate the suitability of an individual to survive, the genetic algorithm calculates a fitness value by means of a fitness function.

In Equation (1), we define a two-criterion fitness function that depends on execution time and quality index of the FGS execution. To capture a trade-off between both parameters and to define the impact of the parameters on the new generation, we apply the weighted sum with weighting factors w_{time} and w_{taste}.

$$\begin{aligned} &fitness(c) = f_{time}(c) \cdot w_{time} + f_{taste}(c) \cdot w_{taste}, \\ &\text{with } f_{time}(c) = \frac{time}{a_{time}},\ f_{taste}(c) = \frac{taste}{a_{taste}}, \\ &0 \leq w_{time}, w_{taste} \leq 1 \text{ and } w_{time} + w_{taste} = 1 \end{aligned} \tag{1}$$

$fitness(c)$ provides the fitness of an individual c. Individuals that cause long execution times and a high quality index, i.e. a low accuracy, have a high fitness value. They are fitter than individuals with lower fitness values. $f_{time}(c)$ calculates the fitness value of an individual c with respect to the FGS execution time. $f_{taste}(c)$ calculates the fitness value of an individual c with respect to the quality index. Since both metrics have different magnitudes, we normalize the values using reference values before combining them in the fitness function. The tester defines both reference value a_{time} and a_{taste} for example as average of execution times or quality values measured by random testing.

```
   Input: population, w_time, w_taste, a_time, a_taste
   Output: fitTime, fitTaste, populationFit
1  maximalFit = populationFit = 0;
2  foreach individual ∈ population do
3      time, taste = FGS(individual);
4      fitValue = time/a_time · w_time + taste/a_taste · w_taste;
5      if fitValue > maximalFit then
6          maximalFit = fitValue;
7          fitTime = time;
8          fitTaste = taste;
9      end
10     individual.fit = fitValue;
11     populationFit += fitValue;
12 end
```

Algorithm 1: Fitness evaluation.

In the evaluation process (Line 12), our genetic algorithm sends each individual in the population as input to the FGS algorithm and calculates its fitness value by means of our fitness function. Line 12 also provides the longest execution time *fitTime*, the worst quality index *fitTaste* and the sum of fitness values *populationFit* of the whole population.

Selection. Our genetic algorithm applies the *stochastic universal sampling* method to select the fittest individuals to generate a new population. Each individual gets a section on a imaginary roulette-wheel proportional to its fitness value. Equally spaced pointers are arranged around the wheel. The number of pointers corresponds to the population size. After turning the wheel once, the algorithm inserts each individual on whose field a pointer points to into the new population. The probability of each individual to be selected is proportional to its fitness value. The selection method reduces the evolutionary pressure but also preserves the variability in the population by selecting test cases with low fitness values (Gerdes et al., 2004, pp. 79-83).

Crossover. Our genetic algorithm performs the *parameterized uniform crossover* strategy to create new individuals (Gerdes et al., 2004, p. 89). The crossover mechanism randomly chooses two not yet selected individuals as parents from the population. For every single gene of the parents, the genetic algorithm decides according to the crossover probability p_c whether the genes are exchanged or not. The genes do not cross if one of them is already contained in its target individual. The genetic algorithm applies the crossover operator to each pair in the population. As a result, the crossover mechanism returns a child population containing new individuals. We define that the tester specifies the crossover probability p_c.

Mutation. The mutation process decides according to a mutation probability p_m for each gene of each individual in the population whether the gene mutates or not. Depending on the mutation probability p_m, the mutation function preserves the diversity in the population or inserts minimal changes to find test cases that locally provoke critical behavior (Gerdes et al., 2004). The tester specifies the mutation probability p_m. If the gene mutates, the genetic algorithm randomly selects a new star from its search space, which is not contained in the individual, as a gene. As a result, the mutation process returns a new mutated population.

Termination Condition. The genetic algorithm terminates if it reaches a given number of generations, if the best solution has not improved in the last n generations (Bhandari et al., 2012), or if the FGS algorithm execution time exceeds a predefined value. The tester defines these criteria.

4.3 Automated Test Generation

The objective of our test approach is to find star combinations that provoke long execution times and inaccurate results of the satellite on-board image processing application. To define a comparatively concise search

space for our genetic algorithm, we utilize the partitioning of the input parameters of the FGS algorithm presented in (Witteck et al., 2019). A given test set contains one star per equivalence class combination of the parameters. This results in approximately 2.8×10^{55} possible combinations of 30 stars as FGS input. Testing all possible combinations is infeasible. Our key idea is a genetic algorithm that is specifically tailored to find particular test cases in a large input domain.

Algorithm 2 gives an overview of the structure of our defined genetic algorithm using the components described in Section 4.2. The test set *TS*, which is complete with respect to the equivalence classes defined in (Witteck et al., 2019) is the search space of our genetic algorithm. The algorithm creates the initial population by randomly selecting stars from its test set until the population size *popSize* is reached. Using our two-criterion fitness function, the algorithm calculates the fitness value for each individual based on the execution time and quality index delivered by the FGS algorithm. By specifying the parameter weights w_{time} and w_{taste}, the tester is flexible to define the test goal.

Input: TS, popSize, w_{time}, w_{taste}, a_{time}, a_{taste}, p_c, p_m, T, maxTime
Output: P

```
1  P ← ∅;
2  t = fitTime = fitTaste = 0;
3  P ← getInitialPopulation();
4  popFit, fitTime, fitTaste ← evaluation();
5  while t < T and fitTime < maxTime do
6      P ← selection();
7      P ← crossover();
8      P ← mutation();
9      popFit, fitTime, fitTaste ← evaluation();
10     t++;
11 end
```

Algorithm 2: Genetic algorithm.

Based on the fitness values, our genetic algorithm generates a new population with fittest individuals in the selection process. On the newly selected, fitter population, Line 11 performs the *parameterized uniform crossover* strategy in the crossover function. This function generates new individuals by mixing the genes of selected individuals according to the crossover probability p_c. The genetic algorithm applies the mutation operator on the newly generated child population. Our genetic algorithm iteratively evolves individuals until it reaches a predefined maximum number of generations T or the achieved maximum execution time or quality index of a generation exceeds a specified maximum execution time *maxTime*. Line 11 provides a population P of individuals that provoke longest execution times and lowest accuracies.

Using the genetic algorithm, our test approach improves a given test set to efficiently provoke worst-case execution time and inaccurate results of the FGS algorithm. If the test detects violations of the requirements, the FGS algorithm has to be corrected and tested again.

5 EVALUATION

We have implemented our test approach to investigate its efficiency for satellite on-board image processing applications. As a case study, we used the FGS algorithm of the PLATO mission.

Our objective is to evaluate our approach for the development and test of the FGS algorithm implementation. Our goal is to test execution time and mathematical accuracy of the algorithm under realistic hardware conditions. We run the FGS algorithm on a GR-XC6S FPGA development board (PENDER ELECTRONIC DESIGN GmbH, 2011) running at 50 MHz.

In our experiments we have used a complete test set that covers all equivalence class combinations presented in (Witteck et al., 2019) as search space. Since Gaussian-PSF stars are unrealistic, we eliminate them from the test set. This reduces the amount of possible star combinations to 1.6×10^{46}. From this test set, our test application selects star combinations and sends picture sequences of 1000 times steps for each star to the development board, where the FGS algorithm calculates the attitude data. As a result, the test application receives execution time and quality index for each time step and averages them over all time steps. Based on these values, our genetic algorithm calculates the fitness value of the executed star combination.

Table 1: Genetic algorithm configuration.

Population size	20
Number of genes	30
Max execution time [ms]	300
a_{time} [ms]	230
a_{taste}	1.5×10^{-9}
p_c	0.5
p_m	0.06
Maximum generation number	50

In our experiments we have used the configuration specified in Table 1. We have taken the reference values a_{time} and a_{taste} from previous experiments.

We have set the population size to 20 and the maximum generation number to 50 due to time reasons. According to PLATO requirements the FGS execution time shall not exceed 300 ms. Thus, we have specified that the genetic algorithm terminates if the execution

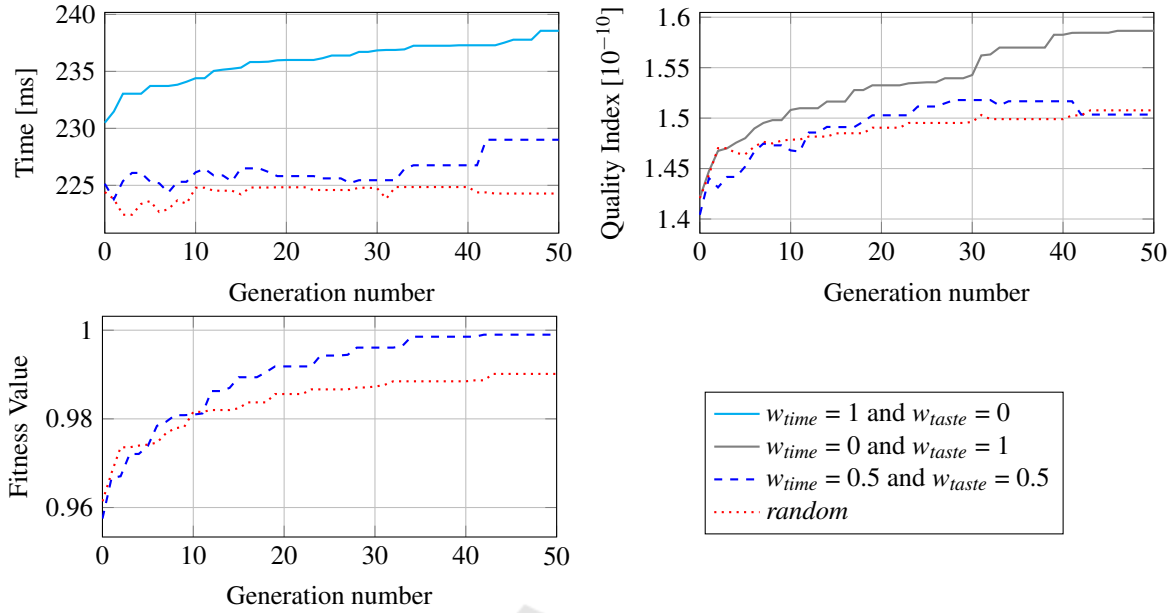

Figure 2: Experimental Results.

time for at least one test case exceeds this value. There are no termination conditions with respect to the quality index as no PLATO requirement exists for this measure. As genetic algorithms involve randomness, due to randomly selecting the initial population or the crossover and mutation process, we have performed 10 independent runs of each experiment and averages the results.

In the first two experiments, our genetic algorithm optimizes solutions for one fitness criteria: either execution time or quality index. For that, we have set the respective weighting factor w_{time} or w_{taste} to 1 and the other to 0. Thus, the calculated fitness value corresponds to the execution time or quality index respectively. The fitness values of both experiments are shown in Figure 2 by the solid lines. The upper left part of Figure 2 presents the average of the highest execution time per generation over 10 runs. The upper right part shows the average of the highest quality index per generation over 10 runs.

To investigate the capability of our genetic algorithm provoking a long execution time and a high quality index at the same time, we have set w_{time} and w_{taste} to 0.5 each. The corresponding execution time and quality index are shown in the upper parts of Figure 2 by the dashed lines. As the figure shows, the execution times do not violate the timing requirement. The execution time and quality index decreases in some generations in favor of a lower accuracy or higher execution time respectively. That is possible because an individual with short execution time may be fitter compared to another individual with longer execution time, because of a much higher quality index.

The resulting evolution of the averaged fitness values per generation is shown in the lower part of Figure 2 by the dashed line. The fitness value increases until the 42nd generation. The curve indicates that fitter individuals would be found if the number of maximum generations was increased. The selected configuration parameters are based on previous tuning experiments and need further research to achieve optimal results.

We have compared our experimental results with random testing. For that, we have randomly selected combinations of 30 stars from our complete test set. Figure 2 shows the measured execution times, quality index and fitness value of the random test by the dotted lines. The results are averaged over 10 runs. We have calculated the fitness values using our fitness function with w_{time} and w_{taste} equals 0.5. Each generation corresponds to 20 random test cases.

Figure 2 shows, the maximum fitness value reached by random testing is lower compared to the genetic algorithm. Thus, our genetic algorithm is more capable to find a higher execution time and higher quality index (i.e. lower accuracy) executing less test cases compared to random testing.

Note that our genetic algorithm automatically provides test sets that have high execution times and quality indexes in a few generations. Hence, it improves the efficiency of the software testing process. However, it will never examine all possible 1.6×10^{46} star combinations. Therefore, we can not rule out if there are other combinations that provoke longer execution times or higher quality indexes. But it increases the

confidence in the robustness of the satellite on-board image processing application.

6 CONCLUSION

Due to complex computations performed by satellite on-board image processing applications, it is difficult to find test cases that provoke mission-critical behavior in a potentially huge input domain. In this paper, we have presented a genetic algorithm that is specifically tailored to automatically find test cases that provoke real-time critical behavior or scenarios where the mathematical accuracy gets critically low.

To achieve this, we have defined a novel two-criteria fitness function that is based on execution time and mathematical accuracy of a given satellite on-board image processing application. Using that function our genetic algorithm automatically steers the search to test cases that provoke long execution times or inaccurate results or both. The tester is able to specify which criterion has more impact on the fitness value of a test case. Moreover, the tester specifies the input parameters of the genetic algorithm, for example, population size, termination conditions, etc. This makes our genetic algorithm flexible and adaptable to different test goals and various on-board image processing applications. Further, the search space and individual representation are based on the partitioning of input parameters into equivalence classes. Areas not relevant to solutions are eliminated since redundant test cases are removed. This makes our search faster.

To demonstrate the efficiency of our genetic approach, we have investigated the capability of the algorithm to automatically find test cases that support robustness testing of a given satellite on-board image processing application, namely the FGS algorithm as an application with high criticality for the PLATO mission. In our experiments, our genetic algorithm automatically evolves test cases with higher execution times and lower mathematical accuracy of the FGS algorithm compared to random testing.

In this paper, we have considered the TASTE value as a qualitative measure of mathematical accuracy. To investigate the accuracy of the application more precisely, we plan to additionally consider errors of the results, for example, angle errors for each axis, as criteria for the mathematical accuracy. Furthermore, we have evaluated our approach by means of a single satellite on-board image processing application. Due to the flexibility of our approach the suitability for other application, for example, blob feature extraction in the robotics domain, can be investigated.

REFERENCES

Alander, J. T. and Mantere, T. (1999). Automatic software testing by genetic algorithm optimization, a case study. In *Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering*, pages 1–9.

Bhandari, D., Murthy, C., and Pal, S. K. (2012). Variance as a stopping criterion for genetic algorithms with elitist model. *Fundamenta Informaticae*, 120(2):145–164.

Gerdes, I., Klawonn, F., and Kruse, R. (2004). *Evolutionäre Algorithmen: Genetische Algorithmen - Strategien und Optimierungsverfahren - Beispielanwendungen*. vieweg, 1 edition.

Griebach, D. (2020). Fine Guidance System Performance Report. Technical Report PLATO-DLR-PL-RP-0003, DLR.

Hänsel, J., Rose, D., Herber, P., and Glesner, S. (2011). An evolutionary algorithm for the generation of timed test traces for embedded real-time systems. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 170–179. IEEE Computer Society.

Moheb R. Girgis (2005). Automatic test data generation for data flow testing using a genetic algorithm. *Journal of Universal Computer Science*, 11(6):898–915.

PENDER ELECTRONIC DESIGN GmbH (2011). Gr-xc6s-product_sheet.

Pertenais, M. (2019). Instrument Technical Requirement Document. Technical Report PLATO-DLR-PL-RS-0001, DLR.

Sharma, A., Patani, R., and Aggarwal, A. (2016). Software testing using genetic algorithms. *International Journal of Computer Science & Engineering Survey*, 7(2):21–33.

Shuster, M. D. (2008). The taste test. *Advances in the Astronautical Sciences*, 132.

Sthamer, H., Baresel, A., and Wegener, J. (2001). Evolutionary testing of embedded systems. *Proceedings of the 14th International Internet & Software Quality Week (QW01)*, pages 1–34.

Varshney, S. and Mehrotra, M. (2014). Automated software test data generation for data flow dependencies using genetic algorithm. *International Journal*, 4(2).

Wegener, J. and Mueller, F. (2001). A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-time systems*, 21(3):241–268.

Witteck, U. (2018). Automated Test Generation for Satellite On-Board Image Processing. Master's thesis, TU Berlin.

Witteck, U., Grießbach, D., and Herber, P. (2019). Test Input Partitioning for Automated Testing of Satellite On-board Image Processing Algorithms. In *Proceedings of the 14th International Conference on Software Technologies - Volume 1: ICSOFT*, pages 15–26. INSTICC, SciTePress.