Chapter 9
G-RankTest: Dynamic Analysis and Testing of Upgrades in LabVIEW Software

Leonardo Mariani, Oliviero Riganelli, Mauro Santoro and Ali Muhammad

Abstract In this chapter we present G-RankTest, a technique for the automatic generation, ranking, and execution of regression test cases for controller applications.

9.1 Introduction

Controller applications are real-time embedded software applications designed for interacting and controlling the environment through sensors and actuators. Controller applications must typically execute cyclic tasks within critical time constraints and are often designed as an integration of multiple components that implement functions that are computed quickly (e.g., in a few milliseconds). Since controller applications have to deal with the physical world, the inputs, the outputs, and the values manipulated by controller applications typically consist of numeric values.

Controller applications are usually validated both outside and inside the target devices. In particular, they are first executed and tested outside the target, where the execution can be easily controlled and monitored. A simulator of the embedding device might be needed if the tested component interacts with the hardware. They are successively executed and tested within the target device with the aim of validating the interaction between the software and the real hardware.

Some of the characteristics of controller applications provide unique opportunities of automation for test case generation and regression testing. In particular:

- The extensive use of values in *numeric domains* dramatically simplifies the automatic generation of test inputs, for instance there is no need to create complex objects [JKXC10].

L. Mariani (✉) · O. Riganelli · M. Santoro
Department of Informatics, Systems and Communication,
University of Milano Bicocca, Milano, Italy
e-mail: mariani@disco.unimib.it

A. Muhammad
VTT Technical Research Centre of Finland Ltd., Tampere, Finland

H. Chockler et al. (eds.), *Validation of Evolving Software*,
DOI 10.1007/978-3-319-10623-6_9

- The *short computations* implemented by components support the execution of a huge number of test cases in a reasonable amount of time.
- The *well-established practice of testing components outside the target device* before testing the components in the target device guarantees the existence of an environment with adequate resources for testing and monitoring.

The regression testing of controller applications could be addressed with classic regression-testing techniques that identify the test cases that must be re-executed on a new program version according to the changes in the code [RUCH01, EMR02]. However, many (visual) languages dedicated to the development of embedded software are not yet adequately supported in terms of techniques for change analysis [Ins12], and the design of a regression-testing solution that focuses on code changes might be hard. An interesting and complementary approach is given by the recent idea of focusing test case selection on the behavioral differences rather than the code differences [JOX10, MPP07]. These techniques exploit the dynamic information collected by executing the software to select test cases according to the behavior observable during run-time rather than the covered statements.

In this chapter we present G-RankTest, a technique for the automatic **G**eneration, **Rank**ing, and execution of regression **Test** cases for controller applications. Given two versions of a component, G-RankTest can automatically produce a prioritized regression test suite by generating numerical test inputs from the base version of a component under test, heuristically ranking the generated tests according to the behavior exhibited by the application, and executing the prioritized test suite on the upgraded version of the component under test. G-RankTest exploits the characteristics of controller applications to generate a huge number of inputs (typically billions) covering a large portion of the input domain and then heuristically identifies the behaviors that can be most easily broken by a change. The key idea implemented in the heuristic illustrated in this chapter is that sequences of close inputs that cause rapid changes in the outputs (i.e., small changes in the input values that result in big changes in the output values) correspond to critical behaviors that can be easily broken by an upgrade and that are worth testing before the others. Thus G-RankTest produces huge test suites ranked according this criterion.

In the rest of the chapter we will specifically refer to applications implemented in LabVIEW, one of the most popular graphical languages for the development of real-time control software. However the concepts introduced in this chapter can also be implemented for controller applications written in different languages.

This chapter demonstrates the effectiveness of the technique with a case study. The case study consists of two components developed at the VTT Technical Research Centre of Finland (VTT), which are part of a robot control system. The robot is designed to carry out divertor maintenance operations at the ITER nuclear fusion power plant.

This chapter is organized as follow. Section 9.2 provides background concepts about LabVIEW as a programming language for embedded software. Section 9.3 describes G-RankTest. Section 9.4 presents test case generation. Section 9.5 describes test case ranking. Section 9.6 presents our prototype implementation of G-RankTest,

and discusses the empirical results obtained with two components of the VTT case study. Section 9.7 discusses related work. Finally, Sect. 9.8 provides final remarks.

9.2 LabVIEW in a Nutshell

LabVIEW is a graphical programming environment provided by National Instruments [Ins12]. It is used worldwide by engineers and scientists to conduct experiments, collect and analyze measurements, and develop control systems for a variety of environments. The distinctive feature of LabVIEW is the graphical programming language: LabVIEW programs resemble flowcharts, providing visible information on the data flow. The LabVIEW environment is provided with standard development tools, such as an integrated compiler, a linker, and a debugger.

A simple LabVIEW program that checks a condition and sums two numbers is shown in Fig. 9.1. The resemblance of the code to control system block diagrams is quite obvious and intuitive for control engineers in many fields.

Another advantage of programming in LabVIEW is the automatic generation of GUIs for controlling programs. In fact the software is ready to run and be used as soon as the coding is finished without putting any effort into developing the GUI separately. The user interface for the example code shown in Fig. 9.1 is shown in Fig. 9.2. The direct relationship of the fields in the user interface with the inputs and outputs in the code can be recognized without difficulty.

In addition to the visual environment, the LabVIEW environment also supports a number of options to enable syntax-based programming. For example, programs written in C or C++ can be either directly copied inside LabVIEW blocks or embedded as DLLs. These capabilities enable developers to use simulations and control

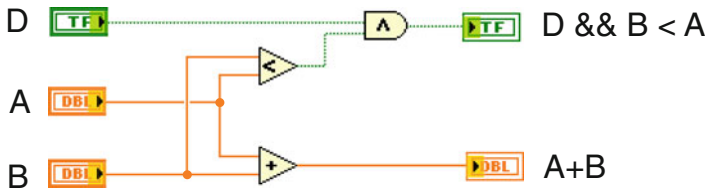

Fig. 9.1 Example of LabVIEW code

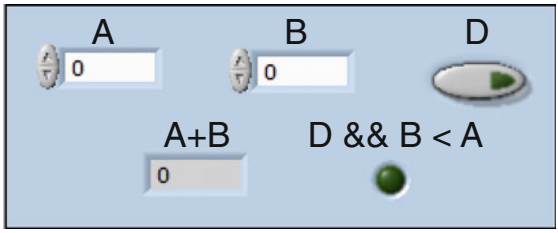

Fig. 9.2 LabVIEW graphical user interface

algorithms, written for instance in MATLAB, Maple, and Mathematica, directly in LabVIEW. Additionally, the LabVIEW environment supports the integration with a variety of hardware devices and provides built-in libraries for data analysis and visualization. For this reason software components developed in LabVIEW are termed Virtual Instruments.

9.3 G-RankTest

G-RankTest is a test case generation, ranking, and execution technique for components with numerical inputs and outputs, which represent a large portion of the components used in embedded software. When the component under analysis includes non-numerical inputs, G-RankTest can still be applied to the numerical part of the input space by assigning constant values to the non-numeric inputs. The process can be repeated multiple times with different values of the constants to study the behavior of the component for different configurations. Non-numeric outputs can be simply ignored for the purpose of the analysis. We also assume that the component under analysis implements a stateless computation, that is the outputs uniquely depend on the values of the inputs, and do not depend on the previous inputs. Even if this assumption restricts the applicability of the technique, there still exist a large number of components used in embedded software that belong to this category. In the future, we aim to extend the ideas and the preliminary results presented in this chapter to the case of stateful components by taking into consideration sequences of inputs rather than single inputs.

For the purpose of this chapter a component under analysis can be modeled as a software unit that implements a function $f : D \rightarrow C$, where $D = I_1 \times I_2 \times \ldots \times I_n$, with $I_i \subset \mathbb{R}$ numerical input, and $C = O_1 \times O_2 \times \ldots \times O_m$, with $O_i \subset \mathbb{R}$ numerical output. Note that inputs and outputs are strict subsets of $\mathbb{R}$ because in a computer system every numerical representation is finite.

G-RankTest produces a prioritized test suite for a target component (i.e., a function $f : D \rightarrow C$) in two steps. In the first step, it generates a (large) test suite $TS = \{tc_1, \ldots, tc_k\}$, where each test case tc_i is a pair (i_i, o_i), with $i_i \in D$ and $o_i = f(i_i) \in C$. Test cases can be generated according to different strategies, depending on the desired distribution of the test inputs. Section 9.4 presents a strategy for the generation of a regularly distributed set of inputs, that is the distance between consecutive inputs is constant.

In the second step, G-RankTest ranks the test cases in the test suite *TS*, finally obtaining a prioritized test suite available for regression testing. The ranking of the test cases aims to identify the test cases that cover the behaviors that can be easily broken by an upgrade. Our intuition is that since inputs and outputs represent values derived from real-world variables, in the majority of the cases the outputs will change smoothly for small changes on the inputs. For instance the temperature of an engine typically changes smoothly while it is operating in normal condition. On the other hand, the most difficult-to-control situations produce big changes on the outputs

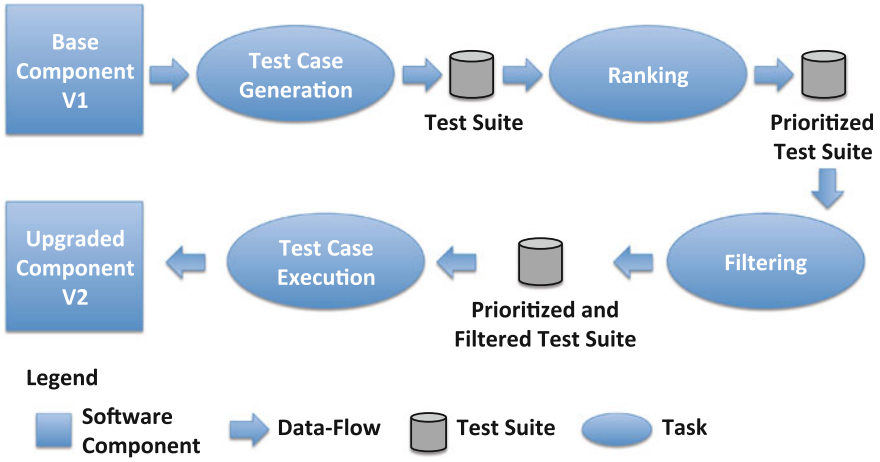

Fig. 9.3 G-RankTest

for small changes to the inputs. For instance, the temperature of an engine changes quickly as soon as the engine is turned on or if the cooling system stops working. Similar examples apply to variables like speed, pressure, and position. Our ranking strategy assigns high priority to the test cases that produce big changes to the outputs for small changes to the inputs (the difficult cases according to our heuristic). The ranking procedure is described in Sect. 9.5.

Figure 9.3 summarizes how G-RankTest works when a component is upgraded and the prioritized test suite is used to reveal regression problems. Note that while the prioritized test suite can be automatically generated without human intervention and in parallel with other development activities, the validation of an upgrade should produce useful results quickly. Thus even if it is feasible to produce a huge prioritized test suite with billions of test cases that sample the component behavior and that requires multiple days to be executed, it is important to prioritize the test cases to reveal failures soon when an upgrade is checked.

Finally, since we focus on regression testing, we assume that the prioritized test suite will be used to check whether the behaviors that should not be affected by the upgrade are really preserved in the new version of the component under test. Thus, before executing the prioritized test suite, the test cases that sample behaviors that are intentionally modified by the upgrade are manually classified as outdated and are discarded from the test suite (this activity is represented by the Filtering task in Fig. 9.3).

9.4 Test Case Generation

The definition of a strategy for the generation of a regression test suite requires the definition of a strategy for sampling the input domain of the function implemented

by the component under test. We assume we do not have any information about the function f that must be tested, with the exception of the range of values that can be assigned to the inputs. Thus, if $D = I_1 \times I_2 \times \ldots \times I_n$ with $I_i \subset \mathbb{R}$ is the input domain, the only available information about the function $f(x_1, \ldots, x_n)$ is that the values accepted by each input variable x_i are defined in $I_i = [b_i, e_i]$, where b_i and e_i denote the minimum and maximum values that can be assigned to x_i, respectively. For simplicity we refer to the case of a closed interval. Any other case can be represented as an union of multiple closed intervals,[1] and the following definitions can be trivially extended to that case.

Many strategies can be potentially defined for sampling D. Three relevant options are: regular sampling, random sampling, and adaptive sampling. Regular sampling implies the generation of a set of inputs that are regularly distributed in the input space. Random sampling implies the generation of random inputs in the input space (according to a uniform distribution of probabilities if no additional information is available) [DN84]. Adaptive sampling implies incrementally generating random inputs, and adapting the generation process according to the characteristics of the function that are captured by the execution of the inputs [CLM05] (e.g., to better sample the most irregular behaviors and sample less the most regular behaviors). In every case the stopping criterion is determined by the time that can be devoted to the testing process. Since the sampling process can be executed without human intervention and without affecting any other development activity, but only requires adequate hardware support, it can be potentially used to generate a huge number of inputs (i.e., test cases), which are successively ranked.

In this chapter we consider regular sampling, which is widely used in practice [Uns00]. For each input variable x_i, we consider a number of samples n_i that are regularly distributed in I_i. More formally, the set of samples for the interval $I_i = [b_i, e_i]$ is given by $S_{I_i} = \{vi_0, \ldots, vi_{n_i}\}$, with $vi_0 = b_i$, $vi_{n_i} = e_i$, $vi_j - vi_{j-1} = C_i > 0 \; \forall j = 1, \ldots, n_i$. The set of samples for the entire input domain D is $S = S_{I_1} \times \ldots \times S_{I_n}$. The value of the gap between two consecutive samples (C_i in the formula) can be different for each dimension of the input space (i.e., each interval I_i), and it is defined by the tester according to the characteristics of the input variables and the time available for testing.

The estimate of the total number of samples that can be executed is done according to the following simple process: we execute a large number of random inputs (e.g., 1,000), we compute the average execution time per sample, and finally we compute the number of samples that saturate the time available for testing. If T is the total time available for testing and avg is the average time for the execution of a single sample, the maximum number of test cases that can be executed is $\frac{T}{avg}$. The values of the gaps C_i are chosen to best exploit the available time, that is $n_0 * \ldots * n_i = |S| \simeq \frac{T}{avg}$.

The characteristics of many controller applications make this simple approach to test case generation extremely useful. In fact, the ratio between the time that is available and the cost of execution of a single input is usually a huge number. The

[1] This is true because numbers have a finite representation in computer systems; it is obviously false in the domain of real numbers.

execution of so many test cases allows the input domain to be well sampled. Moreover, the sampling process leads to the extraction of interesting information about the function computed by the component under analysis. The discovered information can be exploited to rank the generated test cases and increase the effectiveness of regression testing.

9.5 Test Case Ranking

While executing the set of samples S, G-RankTest records the outputs produced by the component under test. In particular, for each $s \in S$, G-RankTest records the value of $f(s) = (f_1(s), f_2(s), \ldots, f_m(s))$ where $f_i(s) \in O_i \ \forall i = 1, \ldots, m$. The set of all the pairs $\{(s, f(s)) | s \in S\}$ is the test suite generated by G-RankTest, where s is the input, and $f(s)$ is the expected output.

We already clarified that the set of generated test cases is extremely large and it is generated through an automatic process that does not affect the development loop. However, the cost of test case execution is important when testing a new version of a component, because the sooner the faults are revealed the easier and cheaper it is to fix them. To anticipate the discovery of faults when test cases are executed, we rank the test suite using heuristics. In the following we present our heuristic to test controller applications.

Controller applications mostly deal with real-world variables, which typically evolve smoothly; for instance, the speed of a robotic arm typically increases or decreases smoothly. However, in some specific cases these variables can even have sharp variations or discontinuities. For instance, a sharp variation in the speed should occur if the robot detects an unexpected obstacle and the arm suddenly stops moving, while a discontinuity should appear if the robot hurts an obstacle. The rationale underlying the heuristic for test case ranking is that regular behaviors are easier to control and design, and there is a small probability that a regression fault is introduced in a regular behavior. On the other hand, special cases are hard to program and may be easily broken because of their complexity. Thus, programmers may easily introduce regression faults in the behaviors corresponding to rare situations. G-RankTest ranks test cases, assigning higher priority to test cases that correspond to behaviors that introduce sharp variations in the outputs, and lower priority to test cases that cover the more regular behaviors.

More formally, every time an input s is executed, in addition to recording the outputs, G-RankTest records the value of the numerical gradient at the same point. The numerical gradient is the numerical approximation of the gradient of a function and indicates how sharp the variation of the output is. Given the function $f = (f_1, \ldots, f_m)$ defined in the domain $D = I_1 \times I_2 \times \ldots \times I_n$, the gradient of each output f_i is a vector $\nabla f_i = (\frac{\partial f_i}{\partial x_1}, \ldots, \frac{\partial f_i}{\partial x_n})$. The value of the numerical gradient for a point $s = (s_1, \ldots, s_n) \in D$ is $\nabla f_i(s) = (\frac{\partial f_i}{\partial x_1}(s), \ldots, \frac{\partial f_i}{\partial x_n}(s))$, where $\frac{\partial f_i}{\partial x_j}(s) = \frac{f(s_1,\ldots,s_j+h_j,\ldots,s_n) - f(s_1,\ldots,s_j-h_j,\ldots,s_n)}{2h_j}$. The value of the gap h_j can be different

according to the considered dimension. In our case a default value of h_j in each dimension j is provided to match the value of the gap C_j used for regular sampling.

In order to evaluate how sharply the output of the function f varies at the point s, we compute the sum of the norm of each vector in the gradient, that is $variation_f(s) = \sum_{i=1}^{n} \|\nabla f_i(s)\|$. The higher the variation is, the more rapidly the outputs change. The value of the *variation* is the value used by G-RankTest to rank the test cases, that is the test cases $(s, f(s))$ with a high value of $variation_f(s)$ are executed before the others.

The gradient is one of the interesting aspects that can be taken into consideration when analyzing the behavior of a function. We look forward to analyzing other aspects that might be relevant for testing, such as the second derivative.

9.6 Experiments

In this section we describe the subject of the study, we present our toolset, and we discuss the empirical results.

9.6.1 Subject of the Study

The components selected for the study are part of the system that controls the Cassette Multifunctional Mover (CMM), which is part of an ITER nuclear fusion power plant. The ITER nuclear fusion power plant is one of a series of experimental reactors which are meant to investigate and demonstrate the feasibility of nuclear fusion as a practical source of energy [Shi04].

Due to a set of very specialized requirements, the maintenance operations of the ITER reactor demand the development and testing of several new technologies related to software, mechanics, and electrical and control engineering. Many of these technologies are under investigation at VTT Technical Research Centre of Finland [MET+07]. In particular, VTT develops the real-time and safety critical control system for remotely operating devices. The control system is implemented using C, LabVIEW and IEC 61131 programming languages and is distributed across the network.

Among the many components in the control system, the CMM, shown in Fig. 9.4, plays a key role in the ITER divertor maintenance activities. The CMM will be required to transport ITER's 54 divertor cassettes, each 3.5 m long and 2.5 m high, and weighting about 9t, through three access ports at the bottom of the reactor. A complex trajectory is followed in order to negotiate the path along the divertor access duct from the transfer cask to the plasma chamber. This process must be executed with high accuracy since the access route is such that the cassettes have to pass within a few centimeters of the vacuum vessel surfaces that house the fusion reaction and act as a first safety containment barrier.

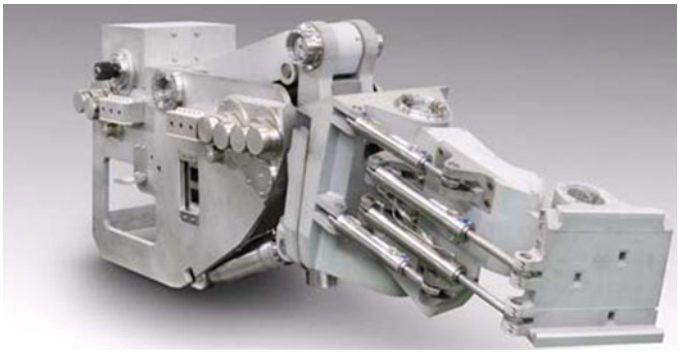

Fig. 9.4 CMM robot at the DTP2 facility at VTT Tampere

9.6.2 Toolset

Our implementation of G-RankTest consists of three components. The first component is implemented in MATLAB and is devoted to test case generation. The output of this component is a grid with every sample that must be executed. The second component is implemented in LabVIEW and is devoted to the execution of the test inputs in the grid and the recording of both the outputs and the gradient. The third component is implemented in MATLAB and is responsible for ranking the test cases, according to the value of the gradient, and visualizing the behavior observed for the component and its gradient directly in MATLAB.

9.6.3 Empirical Evaluation

We empirically evaluated G-RankTest by investigating both the feasibility of the technique to analyze behaviors of real-world software, and the effectiveness of the prioritized test suite to reveal regression faults.

9.6.3.1 Behavior Analysis

We investigated whether G-RankTest can be used to effectively sample the input domain of a real-world component and whether the heuristic can be used to prioritize test cases to effectively discriminate the behaviors of the component under test. The software component selected for this study is part of the CMM simulation models.

We selected the component that calculates the volume of water inside the two chambers of a water hydraulic cylinder. Since hydraulic cylinders with different diameters can be used in the system, the volume of water is calculated as a function of cylinder radius and position. The control system uses the component to detect

water leakages by constantly comparing the measured volume with the output of the component. This ensures the integrity and safety of the system and facilitates the preventive maintenance of the manipulator. The correctness of such components plays a key role in the reliability of the control system of the ITER maintenance equipment.

To give more details, the component selected for the study has two numerical inputs and two numerical outputs. The two numerical inputs represent the cylinder radius r, and the cylinder position x. The radius ranges from 0.05 to 0.5 m, while the position ranges from 0 to 1,000 mm. The two numerical outputs, represented by symbols Va and Vb, indicate the volume of water inside chambers A and B, respectively.

Given the characteristics of the component under analysis we estimated that a time slot of 12 h would be largely sufficient to analyze in detail the behavior of the component. We thus decided to sample each input variable using regular sampling with a step of 0.001. The value of the step was determined together with domain experts from VTT, based on the number of significant digits of the input variables. The total number of input samples that were generated is 451,000,451.

A 12-h testing activity was largely sufficient to precisely analyze the behavior of the function implemented by the component under test. Figures 9.5 and 9.6 show the samples collected for outputs Va and Vb, respectively. Colors are used to indicate the value of the gradient. Note that the points are so dense that the graph appears to show continuous functions, but the plots were instead obtained from a discrete set of values (the ones produced by the test cases). This is early evidence that G-RankTest can be used to analyze the behavior of controller applications.

To evaluate whether our heuristic can be used to discriminate behaviors, for every value of the norm of the gradient, we counted the number of test cases that produce outputs with that norm. Figures 9.7 and 9.8 show the number of test cases that have a

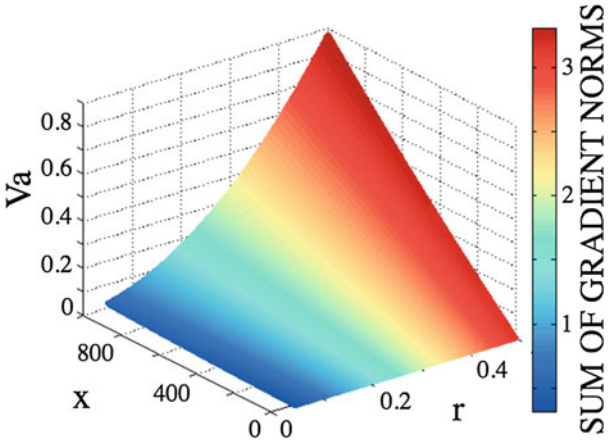

Fig. 9.5 Volume of water in chamber A: samples obtained from test

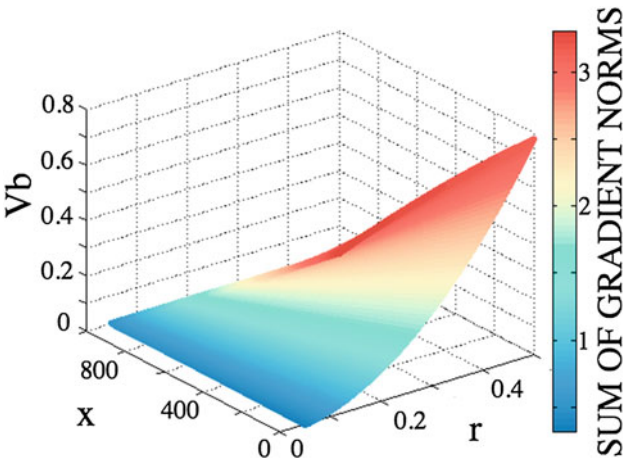

Fig. 9.6 Volume of water in chamber B: samples obtained from test

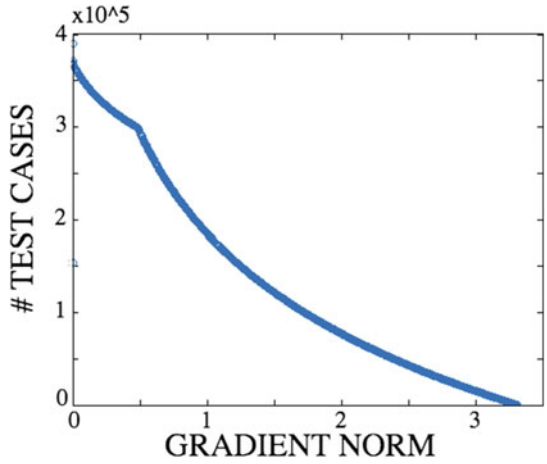

Fig. 9.7 Test case distribution along the gradient norm for the output Va

given value of the norm for the gradient of Va and Vb, respectively. Note that in the majority of cases the outputs change smoothly (small value of the gradient norm), and only a few behaviors produce big changes of the outputs for small changes to the inputs (big value of the gradient norm).

We also investigated the distinguishing capability of our heuristic, which ranks test cases according to the sum of the norms of the gradients of the two outputs. Figure 9.9 shows the number of samples for every value of the sum. Note that there are two uncommon cases: small and high values of the sum of the norms. The presence of a few values with high norms confirms our intuition that our heuristic can be used to select a small subset of complex behaviors that require particular attention every time the application is modified. In the case study, for example, the pressure rises

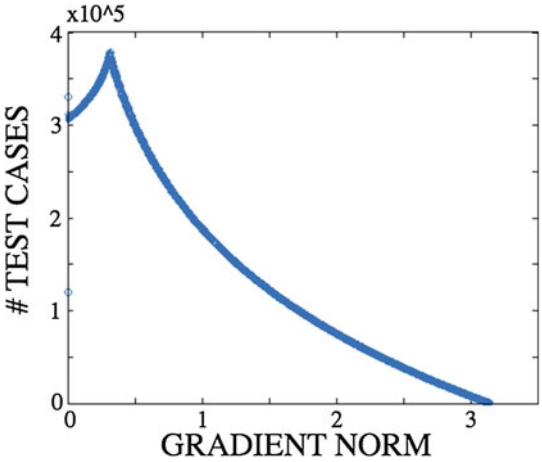

Fig. 9.8 Test case distribution along the gradient norm for the output Vb

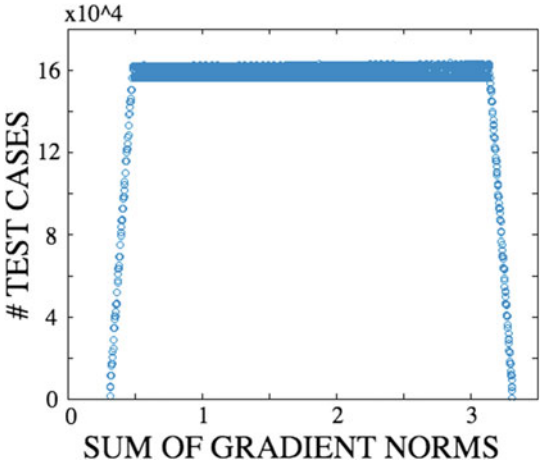

Fig. 9.9 Test case distribution along the sum of gradient norms of the two outputs

inside the cylinders due to external forces. Since the cylinders are not completely stiff, the cylinders under that pressure start flexing and increasing their volume. The points with highest sum of norms are the same points producing the highest flexion. Interestingly there are also a few cases with small values for the sum of the norms. According to our heuristic these values do not correspond to inputs relevant for testing.

9.6.3.2 Regression Testing

We evaluated the effectiveness of G-RankTest to generate test suites that can reveal common regression faults in controller applications on two versions of the same

software component. The component selected for this evaluation is part of the CMM simulation models.

We selected the component that implements a mathematical model that calculates the flow through an orifice (we represent this output with the symbol F). The input variables for the model are the pressure difference across the orifice and the degree of opening of the orifice, represented respectively by symbols pd and o. The parameters such as fluid viscosity and geometry of the orifice are kept constant in the model. The use of such orifices is very common in the fluid power industry. For example a combination of these orifices can be used to create various pneumatic and hydraulic valves. VTT uses this component as part of their systems.

To give more details, the pressure difference pd can range from 0 to 100,000,000, whereas the opening o of the orifice ranges from 0 to 1. The component calculates the flow F of the fluid through the orifice. The component has to consider different computational models (e.g., laminar and turbulent flow) depending on the degree of opening of the orifice. Consideration of multiple types of flows is of critical importance for the accuracy of the models of fluid power systems.

Given the characteristics of the component under analysis, we estimated that a time slot of 24 h was sufficient to analyze in detail the behavior of the component. We thus sampled each input variable using a regular sampling with a step of 1,000 for the pressure difference and 0.1 for the opening of the orifice valve. The total number of input samples that were generated is 1,100,011. A 24-h testing activity was sufficient to precisely analyze the behavior of the function implemented by the component under test. For instance, our tool produced the plot shown in Fig. 9.10. The tester can inspect the plot to check whether the behavior implemented by the component under test satisfies expectations.

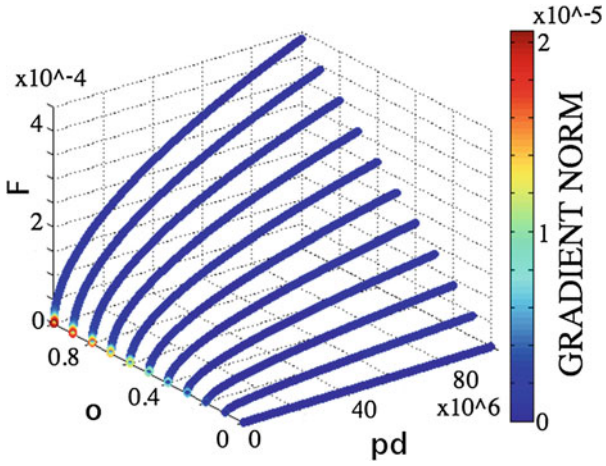

Fig. 9.10 The plot of the function

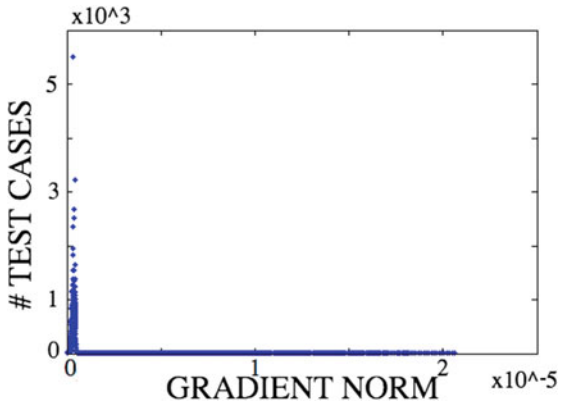

Fig. 9.11 Test case distribution

Our tool ranked the generated test cases. We expected that our heuristic is able to discriminate behaviors. To check the distribution of the test cases, we counted the number of test cases for every value of the norm of the gradient (the numeric gradient was computed with a step of 100 for the pressure difference and 0.01 for the opening of the orifice). The result is shown in Fig. 9.11. Note that the presence of a few values with high norms confirms our intuition that our heuristic can be used to select a small subset of complex behaviors that require particular attention every time the application is modified.

We checked the effectiveness of the prioritized test suite by executing it on the upgraded version of the component. The new version of the component considers the flow to be nearly always turbulent, and thus the calculated flow deviates from the previous version (the upgraded version was also developed at VTT). Even if the variation is small, it occurs in the most critical regions of operation, which correspond to the test cases with high norms according to our heuristic. In particular, the first test that reveals the difference is ranked at position 2.

9.7 Related Work

Test case prioritization is a well-known solution for increasing the effectiveness of regression testing [RUCH01, EMR02]. Most prioritization techniques can rank test cases according to code coverage information and according to the likelihood that a statement includes a fault. Recently regression testing and test case prioritization techniques focusing on (observed) behaviors have been studied with promising results [JOX10, MPP07, MPW04]. Working at the behavioral level rather than the source code level has two major benefits: the strategy can be applied regardless of the accessibility of the source code, and test case selection and prioritization can focus on the actual effect of the test cases (the behavior that is activated), rather than the coverage of statements.

G-RankTest generates and prioritizes test cases working on the observed behaviors only. In the case of controller applications this is particularly interesting because the functions implemented by the components that are part of controller applications mostly deal with numerical values, and their input-output behavior can be approximated and studied using mathematical tools.

A few other approaches have addressed testing of embedded software (controller applications in particular), but none of them defined strategies for test case prioritization. For instance, WISE is a tool that generates test cases for worst-case complexity [BJS09], and Xest is a regression-testing technique for kernel modules [NB10].

Finally, Bongard et al. defined an estimation-exploration algorithm that combines model inference and the generation of training data [BL05]. The resulting technique can sample a state space very efficiently. In the future, we aim to evaluate whether this solution can be used as an adaptive sampling strategy in G-RankTest.

9.8 Conclusions

This chapter presented G-RankTest, a technique for the generation, ranking, and execution of test cases for controller applications. G-RankTest implements regular sampling and test case ranking based on the gradient of the output function. We empirically evaluated G-RankTest with two real-world LabVIEW components that are part of a robot used in a nuclear fusion power plant. The preliminary results suggest that G-RankTest can be effectively used to test controller applications.