G-RankTest: Regression Testing of Controller Applications

Leonardo Mariani, Oliviero Riganelli, Mauro Santoro
University of Milano Bicocca
Viale Sarca, 336 - Milano, Italy

Muhammad Ali
VTT Technical Research Centre of Finland
Tekniikankatu 1, FI-33101, Tampere, Finland

***Abstract*—Since controller applications must typically satisfy real-time constraints while manipulating real-world variables, their implementation often results in programs that run extremely fast and manipulate numerical inputs and outputs. These characteristics make them particularly suitable for test case generation. In fact a number of test cases can be easily created, due to the simplicity of numerical inputs, and executed, due to the speed of computations.**

In this paper we present G-RankTest, a technique for test case generation and prioritization. The key idea is that test case generation can run for long sessions (e.g., days) to accurately sample the behavior of a controller application and then the generated test cases can be prioritized according to different strategies, and used for regression testing every time the application is modified. In this work we investigate the feasibility of using the gradient of the output as a criterion for selecting the test cases that activate the most tricky behaviors, which we expect easier to break when a change occurs, and thus deserve priority in regression testing.

***Keywords*-regression testing; test case prioritization; test case generation; test automation;**

I. 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 few milliseconds). Since controller applications have to deal with the physical world, the inputs, the outputs and the values manipulated by controller applications for the largest part consist of numeric values.

Controller applications are usually validated both outside and inside the target device. 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. Successively they are 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 testing, and regression testing in particular. The extensive use of values in *numeric domains* dramatically simplifies automatic generation of test inputs, for instance there is no need to create complex objects [1]. The *short computations* implemented by components support the execution of a huge number of test cases in a reasonable amount of time. Finally, the *well established practice of testing components in isolation* before testing the components in the target 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 [2], [3]. However many (visual) languages dedicated to the development of embedded software are not yet adequately supported in terms of techniques for change analysis [4], 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 [5], [6]. These techniques exploit the dynamic information collected by executing the software to select the test cases according to their runtime effect, rather than the covered statements.

In this paper we present G-RankTest, a technique for the automatic generation and prioritization of regression test cases for controller applications. G-RankTest can automatically produce a prioritized regression test suite by generating numerical test inputs for a component under test, and heuristically rank the generated tests according to the behaviors exhibited by the application. G-RankTest exploits the characteristics of controller applications to generate a huge number of inputs (e.g., 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 paper is that sequences of close inputs that cause big changes on the output (i.e., the inputs that cause rapid changes on the outputs) correspond to critical behaviors that can be easily broken by a change and that are worth testing before the others. Thus G-RankTest produces huge test suites ranked according this criterion.

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

written in different languages.

This paper investigates the feasibility of the approach with a case study. The case study consists of a component developed at the VTT Technical Research Centre of Finland (VTT), and is part of a robot control system. The robot is designed to carry out the divertor maintenance operations at the ITER nuclear fusion power plant.

This paper is organized as follow. Section II provides background concepts about LabVIEW as a programming language for embedded software. Section III describes G-RankTest. Section IV presents test case generation. Section V describes test case prioritization. Section VI presents early results with the VTT case study. Section VII discusses related work. Finally, Section VIII provides final remarks and outlines future work.

II. LabVIEW in a Nutshell

LabVIEW^{TM} is a graphical programming environment provided by National Instruments^{TM} [4]. It is used world wide 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 its graphical programming language, which is known as G language. The environment is also provided with an integrated compiler, a linker, and debugging tools. LabVIEW programs resemble flowcharts, providing visible information on the data flow as well.

For illustration a simple example of LabVIEW code is shown in Figure 1, where two numbers are added together and a true/false condition is checked. The resemblance of the code with control system block diagrams is obvious making it intuitive for control engineers in many fields.

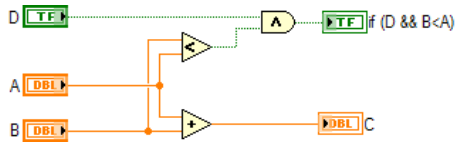

Figure 1. Example of LabVIEW code

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 is shown in Figure 2. The direct relationship of the fields on user interface with the inputs and outputs of the code can be observed without difficulty.

Besides the graphical programming, the LabVIEW environment also supports a number of options to support the use of syntax based programming. For example, the programs written in C or C++ can either be directly copied inside LabVIEW blocks or can be embedded as DLLs. This makes the development of simulations and control algorithms possible using tools such as Matlab®, Maple^{TM}, and Mathematica®, and use them directly in LabVIEW. Additionally, it 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 as Virtual Instruments.

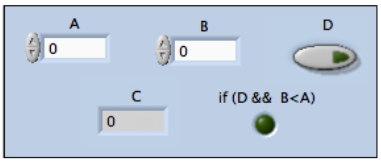

Figure 2. LabVIEW Graphical User Interface

III. G-RankTest

G-RankTest is a test case generation and prioritization technique for components with numerical inputs and outputs, which represent a large proportion 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 value of 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 body of components used in embedded software that belongs to this category. In the future, we aim at extending the ideas and the preliminary results presented in this paper to the case of stateful components by taking into consideration sequences of inputs rather than single inputs.

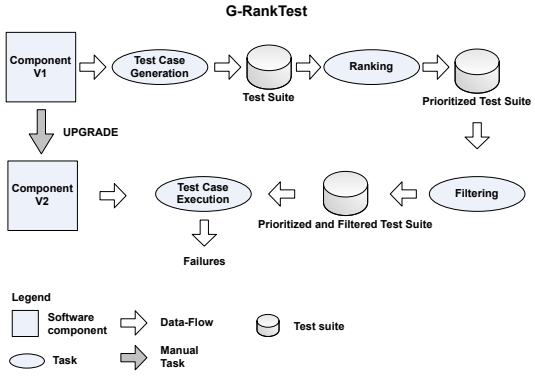

Figure 3. G-RankTest

For the purpose of this paper 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 distribution of the test inputs that it is intended to obtain. Section IV presents a strategy for the generation of a regularly distributed set of inputs.

In the second step, G-RankTest prioritizes 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 at identifying the test cases that cover the behaviors that can be more 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 contrary, the most difficult to control situations produce big changes on the outputs for small changes on 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 exactly aims at assigning high priority to the test cases that produce big changes on the outputs for small changes on the inputs (the difficult cases according to our heuristic). The prioritization procedure is described in Section V.

Figure 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 soon. Thus even if it is feasible to produce a huge prioritized test suite with billions of test cases that sample the component behavior and 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 if 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 classified as outdated and are discarded from the test suite (this activity is represented by the Filtering task in Figure 3).

IV. 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 uniform distribution of probabilities if no additional information is available) [7]. 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 [8] (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 it 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 paper we consider regular sampling that is widely used in practice [9]. For each input variable x_i, we consider a number of samples n_i that are regularly distributed in I_i, that is the distance between two consecutive samples in I_i is constant. 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 $S = S_{I_1} \times \ldots S_{I_n}$. 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.

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

The estimation of the total number of samples that can be executed can be done according the following simple process: execute a bunch of random inputs (e.g., $1,000$), compute the average execution time per sample, and finally 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 exploit at best the available, that is $n_0 * \dots n_i = |S| \simeq T$.

The characteristics of many controller applications make this simple approach about 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 execution of so many test cases allows to sample well the input domain. 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 prioritize the generated test cases and increase the effectiveness of regression testing. The implementation of other strategies, such as adaptive random testing, would allow taking advantage of the information about the function under test not only for test case prioritization, but also to dynamically drive the generation of the test inputs. The investigation of this strategy is part of our future work.

V. Test Case Prioritization

When generating the set of samples S, G-RankTest also executes them and records the outputs produced by the component under test. In particular, for each $s \in S$, G-RankTest records the value of $f(s) = (f_1(s), f_2(s), \dots f_m(s))$ where $f_i(s) \in O_i \ \forall i = 1, \dots m$. The set consisting of every pair $\{(s, f(s)) | s \in S\}$ 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 the generated test cases is extremely large because 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 the cheaper is to fix them. To anticipate the discovery of faults when test cases are executed, we prioritize the test suite using heuristics. In the following we present the heuristic we are currently using to test components.

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, if the robot detects an unexpected obstacle the arm should suddenly stop moving, while if the obstacle is not detected, the arm may even hurt the obstacle and instantaneously stop. The rationale underlying the heuristic for test case prioritization 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 contrary, special cases are hard to program and may be easily broken because of their complexity. Thus, programmers may easily introduce regression faults in that behaviors. G-RankTest prioritizes test cases assigning higher priority to the test cases that correspond to behaviors that introduce sharp variations on the outputs, and lower priority to test cases that cover the most 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 $f = (f_1, \dots f_m)$ defined in the domain $D = I_1 \times I_2 \times \dots \times I_n$, the gradient of each output f_i is a vector $\nabla f_i = (\frac{\partial f_i}{\partial x_1}, \dots \frac{\partial f_i}{\partial x_n})$. The value of the numerical gradient for a point $s = (s_1, \dots s_n) \in D$ is $\nabla f_i(s) = (\frac{\partial f_i}{\partial x_1}(s), \dots \frac{\partial f_i}{\partial x_n}(s))$, where $\frac{\partial f_i}{\partial x_j}(s) = \frac{f(s_1,\dots,s_j+h_j,\dots s_n) - f(s_1,\dots,s_j-h_j,\dots s_n)}{2h_j}$. The value of the gap h_j can be different according to the considered dimension. In our case the value of h_j in each dimension j is chosen to match the value of the gap C_j used for regular sampling.

In order to evaluate how sharp the variation of the outputs of the function f at the point s is, 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 prioritize 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 of analyzing other aspects that might be relevant for testing such as the second derivative.

VI. Case Study

In this section we describe the subject of the study, we shortly describe our toolset, and we present the early results about the feasibility of the approach.

Subject of the Study

The component selected for the study is 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 a part of a series of experimental reactors which are meant to investigate and demonstrate the feasibility of using nuclear fusion as a practical source of energy [10].

Due to a set of very specialized requirements the maintenance operations of ITER reactor demand the development and testing of several new technologies related to software, mechanics, electric and control engineering. Many of these

technologies are under investigation at VTT Technical Research Centre of Finland [11]. 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.

Figure 4. CMM robot at the DTP2 facility at VTT Tampere

Among the many components in the control system, the CMM, shown in Figure 4, plays a key role in the ITER divertor maintenance activities. The CMM will be required to transport ITER's 54 divertor cassettes, each 3.5m-long 2.5m-high weighing about 9t, through three access ports at the bottom of the reactor following a complex trajectory 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.

The software component selected for this study is part of the CMM simulation models. The component 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 the water leakages by constantly comparing the measured volume with the output of the component. This ensure the integrity and safety of the system and facilitate 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.

More in detail the component selected for the study has two numerical inputs and two numerical outputs. The two numerical inputs represent the cylinder radius (we represent this input with the symbol r) the cylinder position (we represent this input with the symbol x). The radius ranges from 0.05 to 0.5 meters, while the position ranges from 0 to 1,000 millimeter. The two numerical outputs, represented with symbols Va and Vb, indicate the volume of water inside chamber A and B, respectively.

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 prioritizing the test cases, according to the value of the gradient, and visualizing the behavior observed for the component and its gradient directly in MatLab.

Feasibility Study

Our objective is to show that G-RankTest can be applied to real-world software. In particular, we want to show that:

- G-RankTest can be used to effectively sample the input domain of a real-world component and collect information about the function computed by the component;
- the heuristic used to prioritize test case can effectively discriminate the behaviors of the component under test.

Given the characteristics of the component under analysis we estimated in 12 hours a time slot 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 has been determined together with domain experts from VTT, based on the number of significant digits for input variables. The total number of input samples that have been generated is $451,000,451$.

A 12 hours testing activity has been largely sufficient to precisely analyze the behavior of the function implemented by the component under test. Figures 5 and 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 a continuous functions, but the plots are instead obtained from a discrete set of values (the ones produced by the test cases). This is an early evidence that G-RankTest can be used to analyze the behavior of controller applications.

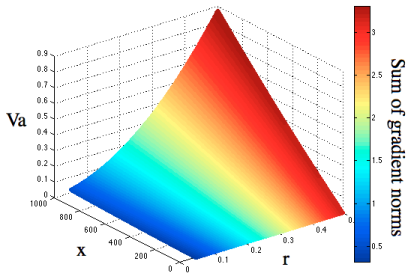

Figure 5. Volume of water in chamber A: Samples obtained from test

We also checked if our heuristic can be used to discriminate behaviors. To this end, for every value of the norm of the gradient, we counted the number of test cases that produce outputs with that norm. Figures 7 and 8 show the number of test cases that have a given value of the norm for the gradient of Va and Vb, respectively. Note that in the

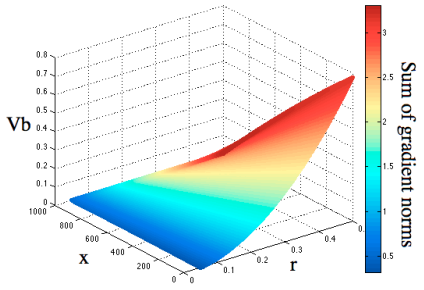

Figure 6. Volume of water in chamber B: Samples obtained from test

majority of the cases the outputs change smoothly (small value of the gradient norm), and only few behaviors produce big changes of the outputs for small changes on the inputs (big value of the gradient norm).

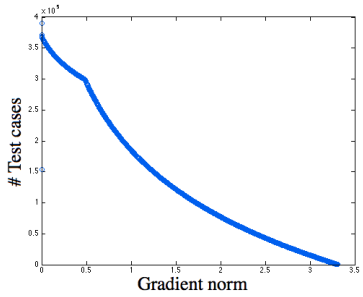

Figure 7. Test case distribution along the gradient norm for the output Va

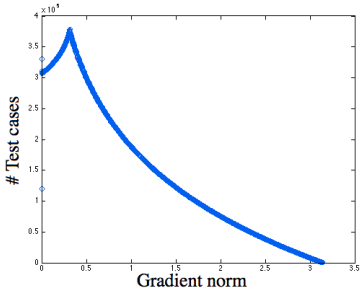

Figure 8. Test case distribution along the gradient norm for the output Va

We also investigated the distinguishing capability of our heuristic, which orders test cases according to the sum of the norm of the gradient of the two outputs. Figure 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. Interestingly there are few cases with small values for the sum of the norms. The presence of few values with high norms confirm 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, a pressure rises 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.

This section reports early evidences about the distinguishing capability of G-RankTest. Further studies are necessary to confirm these evidences and demonstrate that this ranking strategy is also effective in revealing common regression faults in controller applications.

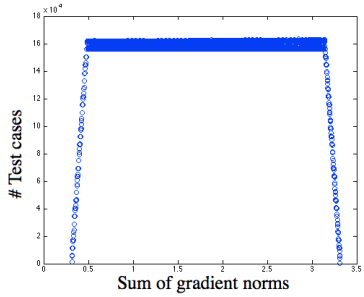

Figure 9. Test case distribution along the sum of gradient norms of the two outputs

VII. Related Work

Test case prioritization is a well-known solution for increasing the effectiveness of regression testing [2], [3]. Most of the 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 the (observed) behaviors have been studied with promising results [5], [6], [12]. Working at the behavioral level rather than the source code level has two major benefits: the strategy can be applied regardless 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 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 [13], and Xest is a regression testing technique for kernel modules [14].

Finally, Bongard et al. defined an estimation-exploration algorithm that combines model inference and the generation

of training data [15]. The resulting technique can sample a state space very efficiently. In the future, we aim at evaluating if this solution can be used as adaptive sampling strategy in G-RankTest.

VIII. Conclusion

This paper presents G-RankTest, a technique for the generation and prioritization of test cases for controller applications. G-RankTest uses regular sampling and test case prioritization based on the gradient of the output function. We investigated the feasibility of the approach with a real-world LabVIEW component that is part of a robot used in a nuclear fusion power plant.

Many aspects deserve further investigation. Here we summarize the main research directions of our future work:

- **support of stateful component:** G-RankTest currently reasons on stateless computations. Even if stateless computations are common in embedded software, a number of components are also stateful. Addressing stateful components requires extending our strategies and heuristic from considering single inputs to sequences of inputs.
- **test case generation:** G-RankTest generates test inputs regularly distributed in the input space. This strategy is extremely simple and can be extended in many ways. In particular, we see three relevant directions: the use of strategies for generating test inputs randomly, adaptive randomly or based on a specification.
- **test case prioritization:** G-RankTest ranks test cases according to the value of the gradient of the function computed by the component under test. This strategy is interesting, but many other aspects about the computed function could also be investigated, and the many mathematical and numerical tools available make the investigation of other strategies easily accessible.
- **empirical work:** in this paper we studied the feasibility of G-RankTest when applied to a real-world component. Even if it is reasonable that the most unusual behaviors might be easily broken during upgrades, this fact needs to be confirmed with empirical evidences. We thus look forward of studying the characteristics of regression faults in industrial controller applications, with particular emphasis on the ones developed at VTT, to confirm that the behaviors selected by G-RankTest are also the behaviors that most frequently include faults. Moreover, our prioritization strategy should be compared with standard prioritization strategies, to further confirm that our heuristic is an effective option for the tester. Finally additional case studies and faults must be consider to build strong empirical evidence in favor of this approach.

Acknowledgment

This work has been funded by the European Union FP7 project PINCETTE (grant agreement n. 257647).

References

[1] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "OCAT: object capture-based automated testing," in *proceedings of the International Symposium on Software Testing and Analysis*, 2010.

[2] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[3] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[4] N. Instruments, "LabVIEW," http://www.ni.com/labview, visited in 2012.

[5] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *proceedings of the International Conference on Software Testing, Verification and Validation*, 2010.

[6] L. Mariani, S. Papagiannakis, and M. Pezzè, "Compatibility and regression testing of COTS-component-based software," in *proceedings of the International Conference on Software Engineering*, 2007.

[7] J. Duran and S. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, pp. 438–444, July 1984.

[8] T. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, M. J. Maher, Ed. Springer-Verlag GmbH, 2004.

[9] M. Unser, "Sampling-50 years after shannon," *Proceedings of the IEEE*, vol. 88, no. 4, pp. 569 –587, 2000.

[10] Y. Shimomura, "The present status and future prospects of the ITER project," *Journal of Nuclear Materials*, vol. 329-333, no. 1, pp. 5–11, 2004.

[11] A. Muhammad, S. Esque, M. Tolonen, J. Mattila, P. Nieminen, O. Linna, and M. Vilenius, "Water hydraulic teleoperation system for ITER," in *proceedings of the Scandinavian International Conference on Fluid Power*, 2007.

[12] L. Mariani, M. Pezzè, and D. Willmor, "Generation of selftest components," in *proceedings of the International Workshop on Integration of Testing Methodologies*, 2004.

[13] J. Burnim, S. Juvekar, and K. Sen, "WISE: Automated test generation for worst-case complexity," in *proceedings of the International Conference on Software Engineering*, 2009.

[14] M. H. Netkow and D. Brylow, "Xest: an automated framework for regression testing of embedded software," in *proceedings of the Workshop on Embedded Systems Education*, 2010.

[15] J. Bongard and H. Lipson, "Nonlinear system identification using coevolution of models and tests," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 4, pp. 361–384, 2005.