Computer Science Journal of Moldova, vol.**24**, no.2(71), 2016

The general prioritization framework

Alexey Malishevsky

Abstract

This paper proposes the general prioritization framework for test case prioritization during regression testing. Regression testing (RT) is done to ensure that modifications have not created new faults or that modifications fulfilled their intended purpose by correctly altering software functionality. Being performed multiple times, RT can have a profound effect on the software budget. The test case prioritization orders test cases for execution to reach a certain objective. Usually, such an objective is to detect faults as early as possible during the testing process. Many prioritization techniques have been developed that successfully reach this objective. However, most of these techniques were developed and studied independently from each other despite the fact that they have many similarities. This article presents the framework that allows to represent known prioritization techniques. Thus, it helps to improve existing and devise new techniques. Also, it allows to implement a single tool that emulates any prioritization technique by just setting the correct parameters. The proposed framework includes the combination/condensation (CC) structure and the structure functions including *element combination functions*, *condensation functions*, and a *super-group combination function*. By defining two such structures together with the corresponding structure functions, one for computing award values and one for their update, any known prioritization technique can be expressed. A general prioritization algorithm is presented that can express any known prioritization technique.

Keywords: Prioritization, regression testing, software testing, prioritization framework, test case prioritization.

1 Introduction

Each time a software system is modified and is to be released, it is regression tested. Regression testing (RT) is similar to testing in general: it involves executing tests and checking the results for correctness. RT, however, is done to ensure that modifications have not created new faults or that modifications fulfilled their intended purpose by correctly altering software functionality.

Being performed multiple times, RT can have a profound effect on the software budget. Because RT itself accounts for a large percentage of software cost [1, 9], even small reductions in RT cost can have a profound effect on the software cost.

If engineers must execute all test cases, which order of test cases should be used? One test order can be better than another under some metric. Test case prioritization orders a test suite to maximize some objective function defined on test orderings. The test case prioritization problem is defined as follows: given a test suite T, the set of permutations PT of T, and a function f from PT to the real numbers; the problem is to find $T' \in PT$ such that $(\forall T'')\ (T'' \in PT)\ (T'' \neq T')\ [f(T') \geq f(T'')]$, where PT is the set of possible prioritizations (orders) of T, and f is an objective function that, applied to any such order, yields an *ordering quality* value for that order.

There are many possible goals for prioritization. For example, testers may wish to increase the coverage of code in the system under test at a faster rate, increase their confidence in the reliability of the system at a faster rate, or increase the rate at which test suites detect faults in that system during regression testing. In the definition of the test case prioritization problem, f represents a quantification of such a goal.

In the literature, many prioritization techniques have been proposed and their effectiveness studied. We will mention just a few of them. Elbaum et al.[6, 7, 8] and Rothermel et al. [15, 16] proposed a set of modification-, coverage-, and fault-exposing-potential-based prioritization techniques. Elbaum et al. [5] incorporated tests costs and fault severities in prioritization. Malishevsky et al. [11] proposed the cost-benefits model for prioritization. Do et al. [4] introduced time constraints into prioritization. Mei et al. [12] applied prioritization to service-oriented business applications. Do et al. [3] explored the usage of mutation faults in prioritization. Bryce et al. [2] utilized prioritization for event-driven software. Raju et al. [14] based prioritization of test cases on four factors such as the rate of fault detection, requirements volatility, fault impact, and implementation complexity. Zhang et al. [18] used integer linear programming for time-aware prioritization. Also, Walcott et al. studied time-aware prioritization in [17]. Mirarab et al. [13] employed Bayesian Networks for the test case prioritization. Hla et al. [10] used particle swarm optimization methods for prioritization.

Most of the proposed prioritization methods were developed and studied independently from each other despite the fact that they had a lot of similarities. We exploited these similarities among prioritization techniques to develop a unifying prioritization framework. This framework can express every prioritization technique developed so far. Its main benefits include the ability to facilitate creation of new prioritization techniques and their analysis, while providing a standard way of looking at techniques. This framework allows us to implement a general prioritization algorithm whose parameters can instantiate various prioritization techniques. It allows rapid prototyping of and research on a variety of new prioritization techniques with minimal coding, while shortening the time to study them, encouraging experimentation with development of new techniques, and reducing the number of errors that might occur if every technique is to be implemented from scratch.

2 Combination/Condensation Structure

We now formally define the combination/condensation (CC) structure on which our framework is based. We first define an *element* e. An element represents a single piece of data used by a prioritization technique. For example, an element $e \in \mathbb{E}$ can represent coverage information for a given statement s, modification information (number of lines changed) for function f, or the fault-exposing-potential for location l. A single element, however, represents this information for the *whole test suite*; thus, it contains such data for every test from the test suite. We also define the set $\mathbb{E}$ as the set of all elements used in the combination/condensation structure.

We define *sub-element* e^t to be a constituent part of element e corresponding to a given test t. We represent an *element* $e \in \mathbb{E}$ as a tuple $< e^1, e^2, ..., e^{|T|} >$ of size $|T|$, where T is a test suite. For example, if element e represents coverage information for statement s, each sub-element e^t represents coverage information for statement s with respect to test case t.

A *vector* v in our structure is a one dimensional array of elements. More formally, $v =< c_1, c_2, ..., c_{|v|} >$, where $\forall c_l \; 1 \leq l \leq |v| \quad \exists e \in \mathbb{E} \quad c_l \equiv e$.

We define a set of elements that comprise a vector v to be E_v.

We define a *group* G to be a tuple of vectors, $G =< v_1, v_2, ..., v_{|G|} >$.

We define V to be a set of all vectors across all groups. $\bigcup_{v \in V} E_v \subseteq \mathbb{E}$, but this subset may be proper: some elements may not belong to any vector. We define such elements as *free* (e is free iff $\forall v \in V \; e \notin E_v$). All vectors that belong to the same group must be *compatible*, defined as having the same number of elements. Vectors across different groups need not be compatible.

For each group G, we define $M_G = (m_{i,j})$ to be the matrix whose columns are the vectors in G, so $m_{i,j}$ represents the i-th component of the j-th vector in G.

Finally, on the top level, a *super-group* SG is defined as a set of groups.

Informally, each element represents a single item of information used by a prioritization technique. It may include function coverage, test costs [5], module criticalities [5], change information [7], or fault-exposing-potential information [6]. Vectors usually represent sets of elements that are treated in the same way, such as coverage, change information, fault-exposing-potential, etc. Groups usually represent sets of vectors used to compute a single component of the final value (produced by the structure).

Next, we define several functions that operate on this CC structure. First, for each group G containing $|G|$ vectors, we define an *element combination function*

$$f^G_{\text{element\_combine}}(x_1, x_2, ..., x_{|G|}, \alpha) \tag{1}$$

This function takes a slice $M^G_{i,1...|G|}$ (an array $< x_1, x_2, ..., x_{|G|} >$ where x_l is the i-th component of the l-th vector in G). This function also has argument α which will be explained later. Each group has its own function $f^G_{\text{element\_combine}}$.

Next, for each group G, we define a *condensation function*

$$f^G_{\text{condensation}}(y_1, y_2, ..., y_k, \alpha), \tag{2}$$

where k is the number of elements in each vector in group G. This function takes the array $< y_1, y_2, ..., y_k >$, where each y_i is the result of applying an *element combine* function to the i-th slice of matrix M_G (defined earlier) ($y_i = f^G_{\text{element\_combine}}(m_{i,1}, m_{i,2}, ..., m_{i,|G|}, \alpha) \quad \forall i, \quad 1 \leq i \leq k$), and an argument α (explained later). Each group has its own *condensation function*.

Finally, we define a *super-group combination function*

$$f_{\text{group\_combine}}(z_1, z_2, ..., z_{|SG|}, \alpha) \tag{3}$$

This function takes an array $< z_1, z_2, ..., z_{|SG|} >$ (z_j corresponds to group $G_j \in SG$) and an argument α (explained later). Each z_j is produced by a *condensation function*: $z_j = f^{G_j}_{\text{condensation}}(y_1, y_2, ..., y_k, \alpha)$.

We call the *element combination function*, *condensation function*, and *super-group combination function*; *structure functions* (SF).

We call this sub-element/element/vector/group/supergroup structure, together with the structure functions, a *combination/condensation structure*.

Let CCF be a particular combination/condensation structure. We define a function $F_{CC}(CCF, \mathbb{E}, \alpha)$ which takes CCF, set $\mathbb{E}$ of elements, and an argument α (explained later), and produces the result of applying the structure functions to the elements of $\mathbb{E}$.

3 Framework and algorithm

In the framework that we now present, we use an iterative approach in which we apply combination/condensation structures to compute award values[1] and to alter elements each time a new test is selected. The main idea is to modify elements $e \in E$, given a newly selected test. This framework uses two CC structures: one, CCF_{award}, for award value computation and another, CCF_{update}, for updating elements from $\mathbb{E}$, where $\mathbb{E}$ is the set of all elements used in the framework.

To compute award values, for each test $t \in T$, we obtain $a_t = F_{CC}(CCF_{\text{award}}, E, t)$. To select the test with the best award value, we compute $t_s = f_{\text{best}}(\vec{a})$. Function f_{best} takes a vector of award values $\vec{a}$ and finds the test (id) with the best award value.

After a new test is selected, the framework algorithm updates all elements $e \in \mathbb{E}$. To compute a new value for an element e, several steps are taken. First, for each test $t_u \in T$ and for each element $e \in \mathbb{E}$, we compute $u_e^{t_u} = F_{CC}(CCF_{\text{update}}, E, < t_s, t_u, e >)$, using CC structure CCF_{update}, where t_s is the test selected in the previous step. Second, we update every element

[1]An award value is the measure of test case's "worth".

Algorithm 1 The prioritization framework algorithm.

```
 1: Initialize elements in $\mathbb{E}$
 2: $List = \epsilon$
 3: for all $t_u \in T$ do
 4:     for all $e \in \mathbb{E}$ do
 5:         $x_e^{t_u} \leftarrow f_{\text{update}}(e^{t_u}, F_{CC}(CCF_{\text{update}}, E, < nil, t_u, e >))$
 6:     end for
 7: end for
 8: for all $t_u \in T$ do
 9:     for all $e \in \mathbb{E}$ do
10:         $val(e^{t_u}) \leftarrow x_e^{t_u}$
11:     end for
12: end for
13: loop
14:     for all $t \in T$ do
15:         $a_t \leftarrow F_{CC}(CCF_{\text{award}}, E, t)$
16:     end for
17:     $t_s \leftarrow f_{\text{best}}(\vec{a})$
18:     if $a_{t_s} = nil$ then
19:         HALT
20:     end if
21:     Add $t_s$ into $List$
22:     for all $t_u \in T$ do
23:         for all $e \in \mathbb{E}$ do
24:             $x_e^{t_u} \leftarrow f_{\text{update}}(e^{t_u}, F_{CC}(CCF_{\text{update}}, E, < t_s, t_u, e >))$
25:         end for
26:     end for
27:     for all $t_u \in T$ do
28:         for all $e \in \mathbb{E}$ do
29:             $val(e^{t_u}) \leftarrow x_e^{t_u}$
30:         end for
31:     end for
32: end loop
```

$e \in \mathbb{E}$ for every test $t_u \in T$ $val(e^{t_u}) = f_{\text{update}}(e^{t_u}, u_e^{t_u})$. We define $val(x)$ to be the value of x.

The framework is applied as follows: determine the set of elements E in all structures, determine the CC structure CCF_{award} for award computation containing a subset of E, determine CC structure CCF_{update} for update computation containing a subset of E, determine initial values for all elements in $\mathbb{E}$, determine a function for element updating f_{update}, decide on a sorting function f_{best}, and finally, apply the framework algorithm that initializes all elements and computes award values, selects a test, and updates elements until the halting condition is satisfied.

The framework algorithm that implements prioritization techniques is presented as Algorithm 1. Lines 3-7 compute initial values for every element. Lines 8-12 set element values to the values computed in lines 3-7.

Group1		Group2	
V1	V2	V1	V2
covF1	bfiF1	covF1	fepF1
covF2	bfiF2	covF2	fepF2
covF3	bfiF3	covF3	fepF3
covF4	bfiF4	covF4	fepF4
covF5	bfiF5	covF5	fepF5

Figure 1. The structure for prioritization technique *fn-bfi-fep-nofb*.

Lines 13-32 implement the main loop computing the prioritized test case sequence. Lines 14-16 compute test award values. Line 17 finds the test with the highest award value. Lines 18-20 test the halting condition. Line 19 adds the selected test to the ordered sequence. Lines 22-26 compute the new values of elements. Lines 27-31 update the value of every element using values computed in lines 18-20. We define the nil value to be the lowest award value a test can have. During comparisons, a test case with award value nil can be chosen only if there are no tests in T with award values not equal to nil.

4 An example

Now we will demonstrate how to fit one existing prioritization technique into the framework. The *fn-bfi-fep-nofb technique* prioritizes tests in an order of decreasing values of sum of covered fault-exposing-potential[2] of modified functions [7]. Thus, this technique employs a binary fault index[3] and fault-exposing-potential information. There are two vectors in each of the two groups: coverage and binary fault index vectors in the first group, and coverage and fault-exposing-potential vectors in the second group. In each group, corresponding elements of that group's two vectors are multiplied and summed. Then, for each test case, an award value is created as a tuple consisting of two values, one from each group; this is used to order test cases. Award values, being tuples, are compared element-wise: first elements are used for sorting, and, in a case of a tie, second elements are compared. The

[2]The probability that a given test case reveals a fault in a given location, if one exists [7].

[3]Binary fault index is the metric on a code change [7].

combination/condensation structure for the award value computation for this technique is presented in Figure 1 where the *element combination* function is multiplication, the *condensation* function is summation, and the *group combination* function is tuple creation. $CovFi$ is the binary function coverage information for function Fi, $bfiFi$ is the binary fault index for function Fi, and $fepFi$ is the fault exposing potential for function Fi. Each of $covFi$, $bfiFi$, and $fepFi$ is a vector of size $|T|$ whose components correspond to test cases from test suite T. As we can see, these techniques fit easily into the framework.

5 Conclusions

We exploited similarities among prioritization techniques to develop a unifying prioritization framework. This framework can express prioritization techniques developed so far. This framework helps to create new prioritization techniques and analyse them, while providing a standard way of looking at techniques. This framework allows us to implement a general prioritization algorithm whose parameters can instantiate various prioritization techniques. It allows rapid prototyping of techniques and research on a variety of new techniques with minimal coding, shortening study time, encouraging experimentation with development of new techniques, and reducing the number of errors that might occur if a technique is to be implemented from scratch.

References

[1] B. Beizer, *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.

[2] R. Bryce, S. Sampath, and A. Memon, “Developing a single model and test prioritization strategies for event-driven software.” *IEEE Trans. on Softw. Eng.*, Vol. 37, No. 1, pp. 48–64, Jan.-Feb. 2011. Available: doi: 10.1109/TSE.2010.12.

[3] H. Do and G. Rothermel, “On the use of mutation faults in empirical assessments of test case prioritization techniques.” *IEEE Trans. Softw. Eng.*, Vol. 32, No. 9, pp. 733–752, Sep. 2006. Available: doi: 10.1109/TSE.2006.92.

[4] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments." *IEEE Trans. on Softw. Eng.*, Vol. 36, No. 5, pp. 593–617, Sep./Oct. 2010. doi: 10.1109/TSE.2010.58.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization." in *Proc. of the Intern. Conf. on Softw. Eng.*, 2001, pp. 329–338 .

[6] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies." *IEEE Trans. of Softw. Eng.*, Vol. 28, No. 2, pp. 159–182, Feb. 2002.

[7] S. Elbaum, P. Kallakuri, A. G. Malishevsky, G. Rothermel, and S. Kanduri, "Understanding the effects of changes on the cost-effectiveness of regression testing techniques." *Journal of Software Testing, Verification, and Reliability*, Vol. 13, No. 2, pp. 65–83, June 2003.

[8] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique." *Softw. Quality J.*, Vol. 12, No. 3, pp. 185–210, Sep. 2004. Available: doi: 10.1023/B:SQJO.0000034708.84524.22.

[9] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineerijng*. 1st ed., Upper Saddle River, NJ: Prentice Hall, 1991.

[10] K. Hla, Y. Choi, and J. Park, "Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting." in *IEEE 8th Intern. Conf. on Computer and Inform. Technology Workshops*, pp. 527–532, 2008. Available: doi: 10.1109/CIT.2008.Workshops.104.

[11] A. Malishevsky, G. Rothermel, and S. Elbaum, "Modeling the cost-benefits tradeoffs for regression testing techniques." in *Proceedings of the International Conference on Software Maintenance*, 2002, pp. 204–213.

[12] L. Mei, Z. Zhang, W. K. Chan, and T. H. Tse, "Test case prioritization for regression testing of service-oriented business applications." in *Proc. of the 18th Intern. Conf. on World Wide Web*, WWW '09, 2009 pp. 901–910, New York, NY, USA: ACM. Available: doi: 10.1145/1526709.1526830.

[13] S. Mirarab and L. Tahvildari, "An empirical study on bayesian network-based approach for test case prioritization." in *2008 1st Intern. Conf. on*

Softw. Testing, Verification, and Validation, 2008, pp. 278–287, Available: doi: 10.1109/ICST.2008.57.

[14] S. Raju and G.V. Uma, “An efficient method to achieve effective test case prioritization in regression testing using prioritization factors.” *Asian J. of Inform. Tech.*, Vol. 11, No. 5, pp. 169–180, 2012. Available: doi: 10.3923/ajit.2012.169.180.

[15] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, “On test suite composition and cost-effective regression testing.” *ACM Trans. Softw. Eng. Methodol.*, Vol. 13, No. 3, pp. 277–331, July 2004. Available: doi: 10.1145/1027092.1027093.

[16] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia, “The impact of test suite granularity on the cost-effectiveness of regression testing.” in *Proc. of the 24th Intern. Conf. on Software Engineering*, 2002, pp. 230–240.

[17] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos, “Time-aware test suite prioritization.” in *Proc. of the 2006 Intern. Symp. on Softw. Testing and Analysis*, ISSTA ’06, 2006, pp. 1–12, New York, NY, USA: ACM. Available: doi: 10.1145/1146238.1146240.

[18] L. Zhang, S. Hou, C. Guo, T. Xie, and H. Mei, “Time-aware test-case prioritization using integer linear programming.” in *Proc. of the 18th Intern. Symp. on Softw. Testing and Analysis*, 2009, pp. 213–224. Available: doi: 10.1145/1572272.1572297.

Alexey Malishevsky

Received March 29, 2016

Institution: ESC “Institute for Applied System Analysis” National Technical University of Ukraine “KPI”
Address: building 35, 37 Prospect Peremohy, 03056, Kyiv, Ukraine
Phone: +380504101177
E–mail: alexeym_s@yahoo.com