Proceedings of the Federated Conference on Computer Science and Information Systems pp. 99–108
DOI: 10.15439/2017F215
ISSN 2300-5963 ACSIS, Vol. 11

Using Classification for Cost Reduction of Applying Mutation Testing

Joanna Strug
Department of Electrical
and Computer Engineering,
Cracow University of Technology
ul. Warszawska 24, 30-059 Krakow, Poland
Email: pestrug@cyf-kr.edu.pl

Barbara Strug
Department of Physics, Astronomy
and Applied Computer Science,
Jagiellonian University,
ul. Łojasiewicza 11, 30-348 Krakow,Poland
Email:barbara.strug@uj.edu.pl

***Abstract*—The paper uses machine learning methods to deal with the problem of reducing the cost of applying mutation testing. A method of classifying mutants of a program using structural similarity is proposed. To calculate such a similarity each mutant is firstly converted into a hierarchical graph, which represents the mutant's control flow, variables and conditions. Then using such a graph form graph kernels are introduced to calculate similarity among mutants.The classification algorithm is then applied for prediction. This approach helps to lower the number of mutants which have to be executed. An experimental validation of this approach is also presented.**

I. Introduction

ARTIFICIAL intelligence has a long history and has been successfully applied in different domains. While the use of artificial intelligence methods in medical applications, image processing or even art has been widely accepted, the domain of software engineering has taken longer to start using such methods. As they are faced with a complex task of designing, building and testing systems at large scales, software engineers have started to adopt and use many of the practical algorithms and techniques that have been proposed by the AI community. AI algorithms are well suited to such complex software engineering problems, as they are designed to deal with one of the most demanding challenges of all; the replication of intelligent behavior. In particular in software engineering community three areas of AI are mostly used. The first one can be described as based on computational search and optimization techniques (the field known as Search Based Software Engineering (SBSE). In SBSE based approach, the aim is to re-formulate software engineering problems as optimization problems that can then be dealt with by using search algorithms. This approach has been widely used with applications from requirements and design to to maintenance and testing [1], [2].

The other two are related to fuzzy and probabilistic methods for reasoning in the presence of uncertainty and methods using classification, learning and prediction algorithms. In classification and prediction research there has been great interest in modeling and predicting software production costs as part of project planning. A wide variety of traditional machine learning techniques such as artificial neural networks, cased based reasoning and rule induction have been used for example in software project prediction, and defect prediction [3].

The paper presents a continuation of an ongoing research on using methods of machine learning to reduce the costs of applying mutation testing [4], [5]. Mutation testing [6] is a recognized software testing technique used to support selection of tests [7], [8]. It provides means and an adequacy measure to assess the quality of the tests and thus it helps to obtain a suite of tests being adequate to provide dependable testing results. Testing results are one of the main sources of information used to establish the degree to which a developed system meets certain requirements and to decide whether the system is ready to be deployed or should undergo further improvements.

The concept behind mutation testing is fairly simple, yet it yields useful results. The assessment of the tests quality is carried out by checking their ability to detect faulty versions of a tested system [6]. The faulty versions (called mutants) are generated from the original version of the system (usually its source code or model) by inserting into a copy of the original system small syntactic changes, one per mutant, and then executed with the tests under assessment. When results of executing a mutant differs from results of executing the original system for at least one test from the assessed suite the mutant is considered to be killed by the test otherwise it stays alive. The ratio of mutants detected (killed) by the tests to the total number of the generated mutants (called a mutation score) is considered to be the most reliable measurement of the tests adequacy [6]. Presence of alive mutants, if they are not equivalent mutants [8], indicate inadequacy of the assessed suite that should be dealt with to increase its quality.

Mutation testing is the most reliable test assessment technique [7], but unfortunately its application can be very time consuming [8]. The main reason of the problem is a large number of mutants that are typically generated and executed. The approaches presented in [5] and in this paper focus on providing some solution to the problem.

In this paper a classification based approach is proposed as a tool to lower the cost of software testing with the use of mutation testing by limiting the number of times a test suite has to be executed. The approach proposed is based on the structural similarity of mutants. To calculate such a

similarity each mutant is firstly converted into a hierarchical graph, which represents the mutant's control flow, variables and conditions. Then using such a graph form graph kernels are introduced to calculate similarity among mutants. The kernels are then used in predicting whether test suite, provided for the program, would detect (kill) a given mutant or not. The classification algorithm is then applied for prediction This approach helps to lower the number of mutants which have to be executed. Moreover, as the similarity calculations have to be done only once for a given set of mutants, they can be used for any new test suite developed for the same system. An experimental validation of this approach is also presented in this paper. An example of a program used in experiments is described and the results obtained, especially classification errors, are presented.

II. Related work

Mutation testing was originally introduced to assess tests ability to detect faults in programs [6], but with the time its application area has expanded. It is currently used at both, implementation and model level, to assess the quality of existing test suites, to improve such suites or to generate new ones [9], [10], [11], [12], [14], [15], [16]. Mutation testing provides a very reliable measurement of a test quality [7]. The effectiveness of the technique can be partially attributed to the systematic and unbiased way of generating the mutants by applying so called mutation operators [6]. The mutation operators controlling the mutants generation process are rules that specify syntactic changes that can be introduces into the mutated artifact and the elements that can be changed, so that the mutants are executable. However, the number of mutants generated by applying such mutation operators can be very large and thus their generation and execution can take a huge amount of time. Several costs reductions techniques have been proposed so far. They can be roughly divided into two groups: approaches targeting the mutants generation phase (selective mutations [17]) and approaches targeting the mutants execution phase (e.g. mutant sampling [21], mutant clustering [20], [19] or parallel processing [18]). A short surveys of costs reduction approaches can be found in [8].

The approach presented in this paper belong to the second group and shares some similarities with mutant sampling and mutant clustering. The mutant sampling, as proposed by Acree [21] and Budd [22], consists in generating all mutants and executing only their randomly selected subset. However, random selection may decrease the reliability of test assessment results. More sophisticated approaches using clustering algorithms were proposed by Hussain [19] and Ji at. all [20]. Ji at. all [20] proposed to weight mutants using a domain specific analysis, divide them into groups basing on the weighting results and then execute only some mutants being representative for the groups. Hussain [19] applied clustering algorithms to group mutants accordingly to their detectablity and used the results to minimize a suite of tests.

In our approach the mutants are also first generated, and then their fraction is selected to be a training group. Only mutants belonging to the training group are executed. The detectability of the remaining mutants is predicted basing on their similarity to mutants from a training group.

As in our approach the mutants are represented by graphs to calculate the similarity among them some form of graph based measure must be used. The problem of graph similarity has already been the subject of research in various contexts. In the current literature three major approaches can be observed.

One of the approaches uses mainly standard graph algorithms, like finding a maximal subgraph or, in more recent years, mining for frequently occurring subgraphs. The approach based on frequent pattern mining in graph analysis has been researched mainly within the context of bioinformatics and chemistry [25], [26], [27], [28]. The main problem with this approach results from a huge number of frequent substructures often found what leads to high computational costs.

Other approach is based on transforming graphs into vectors. This is done by finding some descriptive features in graphs and enumerating them. A lot of research using this method, called vector space embedding of graphs, have been done by Bunke and Riesen [29], [30]. The main benefit of such transformation of a graph into a vector is the possibility to use standard statistical learning algorithms. This approach suffers from the difficulty in finding appropriate features in graphs and then in enumerating them. It often causes problems similar to those in frequent pattern mining. Nevertheless, this approach has successfully been applied in many domain [29].

Finally the use of the theory of positive defined kernels and kernel methods [29] was proposed, among others, by Kashima and Gartner [32], [33]. A lot of research is available on the defining and use of kernels for structured data, including tree and graph kernels [32], [33], [34], [35], for example the tree kernels proposed by Collins and Duffy [34] that were applied in natural language processing.

Considering graphs we have several choices of different kernels proposed so far. One of them is the so called all subgraph kernel which requires enumerating all subgraphs of given graphs and the calculating the number of isomorphic ones. This kernel is known to be NP-hard [32]. An interesting group of graph kernels consists of different variants of kernels based on computing random walks on compared graphs. This group includes the product graph kernel [30] and the marginalized kernels [33]. These kernels are computable in polynomial time ($O(n^6)$ [30]).

Many of the graphs kernels are based on the so called convolution kernels proposed by Haussler [37]. The main idea of these kernels is to use the number of substructures of any structured object. This approach was extended by Shin et al [38], who proposed a mapping kernel for tree data.

Currently the main research focus in kernels is on improving the performance of the kernel algorithms for simple graphs, mainly in bioinformatics. This paper on the other hand is focused on the application of kernel methods in software testing domain.

III. KERNEL METHODS FOR STRUCTURED DATA

Many of machine learning algorithms require the input data to be in a vector or matrix format. It is usually assumed that there is a number of features that can describe a given problem.In case of classification problems the data for would contain vectors of values for all the features and a correct output value. The input can then be divided into training and test sets used to find the model describing the problem. Unfortunately it is not always easy to represent a given problem in vector format without loosing some internal relationships within the data. There are situations when some form of structured representation fits better then a numerical, vector based data, yet we still would like to use machine learning algorithms for this problems.

One of the structure that is becoming more and more important is a graph. There is a wide acceptance of the fact that it is important to represent some internal relationships in data in a form of graphs, yet using machine learning approach in problems where data has a graph representation is rather limited.

There has been research on transforming graphs into vectors by finding some descriptive features in graphs and enumerating them. This approach has been carried out for example by Bunke and Riesen [29], [30]. The obvious benefit of transforming a graph into a vector is the possibility to use standard statistical learning algorithms.

Another approach, which allows for the direct use of structured data is based on the application of kernel methods. In order to apply kernel methods to structured data objects, firstly a kernel function between two structured objects must be defined. However, defining a kernel function is not an easy task, because it must be designed to be positive semi-definite, and some ad hoc similarity functions are not always positive semi-definite.

A. Kernel methods for graph data

In order to use kernel methods for non-vector data a so called kernel trick is often used. It consists in mapping elements from a set A into an inner product space S (with a natural norm), without having to compute the mapping, i.e. in case of graphs they do not have to be mapped into some objects in target space S, but only the way of calculating the inner product in that space is needed. The results of the linear classification algorithm in target space are then equivalent with classifications in source space A. To be able to use this approach and avoid actual mapping the learning algorithms which need only inner products between the elements (vectors) in target space are used. Moreover, the mapping has to be defined in such a way that these inner products can be computed on the objects in the source t space by means of a kernel function. To use classification algorithms a kernel matrix K must be positive semi-definite (PSD) [32], although there are empirical results showing that some non PSD kernels may still do reasonably well, if only they well approximate the intuitive perception of similarity among given objects.

In case of graph data the first kernel was based on comparing all subgraphs of two graphs and then the value of such a kernel was calculated as the number of identical subgraphs. This is a very good similarity measure but enumerating all subgraphs has a high cost. Another approach proposed by Kashima et all. [33] uses kernel on sequences of labels of nodes and edges along each path. It is defined as a product of subsequent edge and node kernels. Computing this kernel requires summing over an infinite number of paths but it can be done efficiently by using the product graph and matrix inversion [32].

A more general approach was proposed by Haussler in the form of convolution kernels [37], which are a generic method for different types of structured data, not specific to graph data. This approach is based on the fact that any structured object can be decomposed into components, then kernels can be defined for those components and the final kernel is calculated over all possible decompositions.

Let X be the space of all possible structures in a given problem. Formally, for any two points x, y, from the space X, a convolution kernel is defined by equation 1 :

$$K(x,y) = \sum_{(x',x)\in R} \sum_{(y',y)\in R} k(x',y'), \tag{1}$$

where $R \subseteq X' \times X$ is a decomposition relation and k is a base kernel. Haussler [37] proved that if k is positive semi-definite then also K is positive semi-definite.

By defining X'_x as a set $\{x' \in X' | (x',x) \in R\}$ the equation 1 can be reformulated into equation 2 :

$$K(x,y) = \sum_{(x',y')\in X'_x \times Y'_y} k(x',y'). \tag{2}$$

B. Mapping kernels

Looking at the equation above it can be observed that the main problem with this approach is that the kernel has to be computed over the whole cross product of $X'_x \times Y'_y$. To deal with this problem a so called mapping kernel was introduced by Shin et al.[38]. It has been successfully used for trees and it allows to limit the calculations to the subset of the cross product. This subset is defined as $M_{x,y} \subset X'_x \times Y'_y$. Then, the mapping kernel is defined by equation 3:

$$K(x,y) = \sum_{(x',y')\in M_{x,y}} k(x',y'). \tag{3}$$

It has to bo noticed, however, that while for the convolution kernel if k is PSD then K is always PSD, in case of the mapping kernel for K to be PSD k has to be PSD and M has to be transitive [36]. Thus the deciding factor here is the choice of the mapping system M.

IV. STRUCTURAL REPRESENTATION OF PROGRAMS

Programs are usually graphically represented in a form of a so called control flow diagram (CFD). An example of a CFD is depicted in Fig. 1 (It was obtained from a Java source

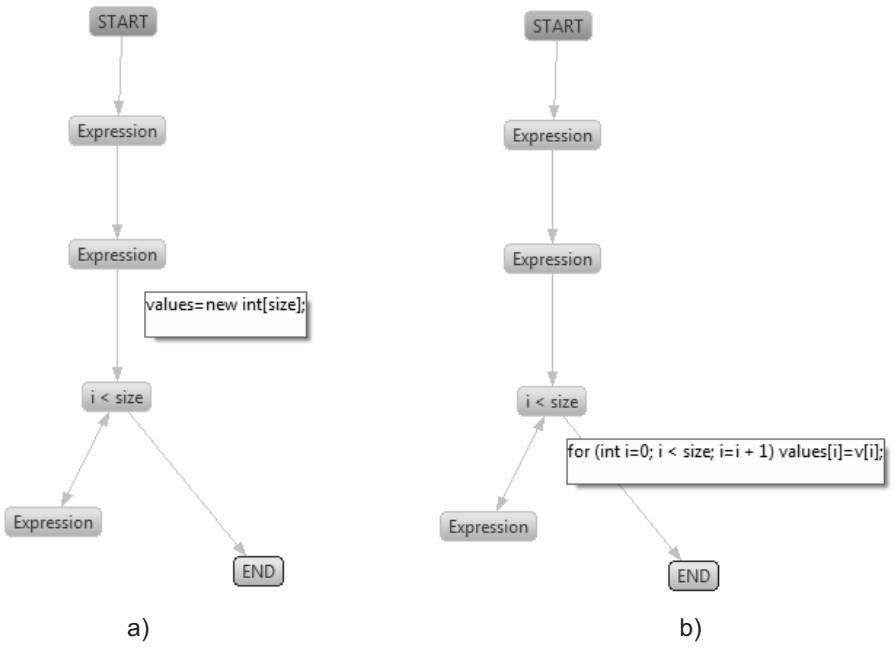

Fig. 1. Examples of control flow diagrams for program from Fig. 2a showing a) internal elements of an expression and b) of a condition

code by an Eclipse plug-in). In the figure it can be observed that it labels the nodes representing expressions with the term "expression" and the actual expression is only available as an attribute (Fig. 1a). Also loops are labelled only by conditions without information on the loop type; this data is also only available as attributes (Fig. 1b). Moreover this attribute contains the whole expressions, and thus it cannot be directly used to compare programs, as for that reason we need to compare each element of any expression or condition separately. However, this information can be parsed to obtain a structural representation better suited for comparing programs.

A. Hierarchical graphs

In this paper we propose to use a so called hierarchical control flow graph (HCFG), which is a combination of CFD and hierarchical graphs [40], [4]. Such a graph adds a level of hierarchy to the traditional control flow diagram. As a result it facilitates the representation of each element of a method in a single node. Such a structural representation is much more adequate for comparing.

In Fig. 2b an example of such a hierarchical control flow graph (HCFG) is depicted. This graph represents a method $search(...)$ presented in Fig. 2a. It can be noticed that each non-hierarchical node (i.e. a node that does not contain child nodes) represents the most basic elements of a program, such as variables, constants, operators. Hierarchical nodes, on the other hand, represent expressions or composed statements such as conditions or loops. The hierarchical nodes not only simplify the representationę but also reflect the context in which the basic elements are placed within the program. Edges of the graph represent flow of control between nodes, both hierarchical and non-hierarchical as well as internal structure of expressions.

A hierarchical graph consists of nodes and edges, that can be labeled and attributed. As opposed to simple graphs, nodes in hierarchical graph can contain other nodes. More formally, let R_V and R_E denote the sets of node and edge labels, respectively and ϵ be a special symbol used for unlabelled edges. The set R_V consists of the set of all possible keywords, names of variables, operators, numbers and some additional grouping labels (like for example $declare$ or $array$ shown in Fig. 2b. The set of edge labels R_E contains Y and N. Then a hierarchical control flow graph is defined formally in the following way:

Definition 1: A labelled hierarchical control flow graph $HCFG$ is a 5-tuple (V, E, ξ_V, ξ_E, ch) where:

1) V is a set of nodes,
2) E is a set of edges, $E \subset V \times V$,
3) $\xi_V : V \to R_V$ is a function assigning labels to nodes,
4) $\xi_E : E \to R_E \cup \{\epsilon\}$ is a function assigning labels to edges,
5) $ch : V \to P(V)$ is a function, which assigns to each node a set of its children, i.e. nodes directly nested in v.

As in further consideration an access to a node of which a given node is a child may be needed we have to formally define such a node- called a parent. Let $ch(v)$ denotes the set of children of v, and $|ch(v)|$ the size of this set. Let par be a function assigning to each node its parent (i.e. a direct ancestor) and let λ be a special empty symbol (different from ϵ), $par : V \to V \cup \{\lambda\}$, such that $par(v) = w$ if $v \in ch(w)$ and λ otherwise.

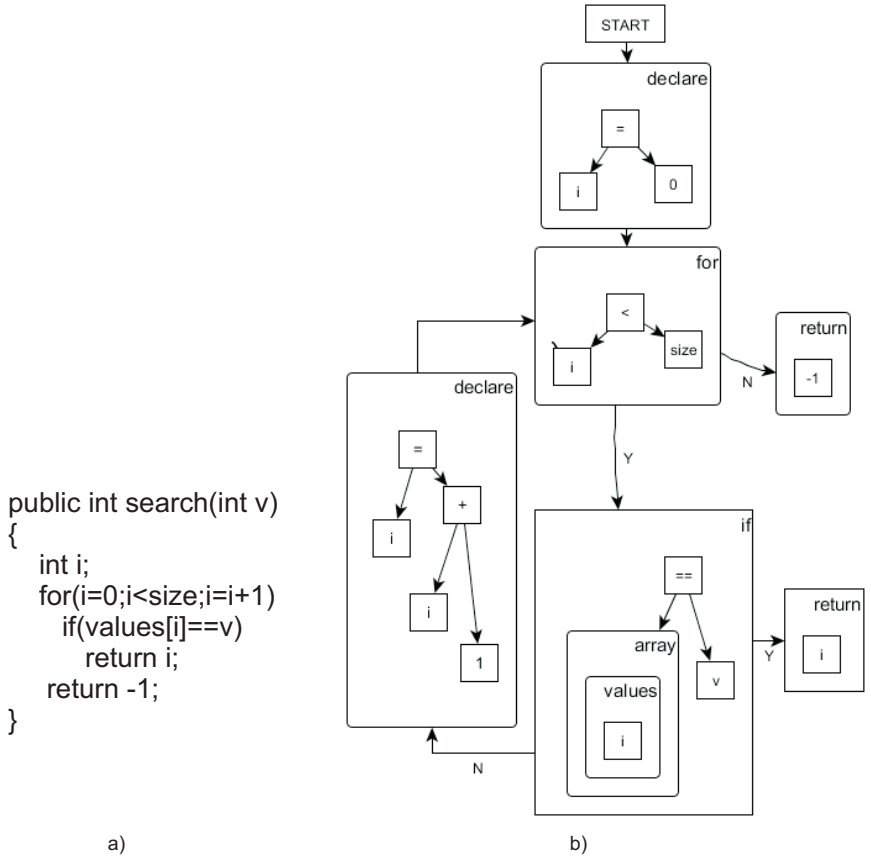

Fig. 2. a) Source code of a part of the example program b) a hierarchical flow graph for this source code

B. Kernel methods for programs

In this paper new kernel functions, based on the convolution kernel [37] and on mapping kernel template proposed by Shin et al ([38]) are introduced for hierarchical control flow graphs.

These kernels are based on the decomposition of a HCFG into substructure composed of a node, its parent, the set of its children and all its first level neighbors. For the kernel calculations the label of a given node, number and labels of its children (and thus its internal complexity), the label of its parent (and thus its position within the structure of the program), the number and labels of edges connecting this node with its neighborhood nodes (both incoming and outgoing edges are taken into account) and the labels of the neighboring nodes are taken into account. This kernel uses two node kernels, an edge kernel and a tree kernel as base kernels. The way the node and edge kernels are defined is presented below. The tree kernel, used within the node kernel to compare expression trees, is a standard one [36].

The first node kernel is a binary kernel and is used for a simple comparison of two nodes on the basis of the identity of their labels.

Definition 2: A binary node kernel, denoted $k_{node}(v_i, v_j)$, where v_i, and v_j are nodes of a hierarchical control flow graph, is defined in the following way:

$$k_{node}(v_i, v_j) = \begin{cases} 1 & : \xi_V(v_i) = \xi_V(e_j) \\ 0 & : \xi_V(v_i) \neq \xi_V(e_j). \end{cases}$$

The second node kernel uses the hierarchical structure of some nodes and is denoted $k_V(v, w)$, where v, and w are nodes of a hierarchical control flow graph.

Definition 3: A node kernel, denoted $k_V(v, w)$, where v, and w are nodes of a hierarchical control flow graph, is defined in the following way:

$$k_V(v, w) = \begin{cases} K_T(ch(v), ch(w)) & : \xi_V(v) = \xi_V(w) \\ 0 & : \xi_V(v) \neq \xi_V(w). \end{cases}$$

It can be observed that if the nodes have children, thus containing an expression trees, a tree kernel K_T is used to compute the actual similarity. For nodes having different labels the kernel returns 0.

Definition 4: An edge kernel, denoted $k_E(e_i, e_j)$, where e_i, and e_j are edges of a hierarchical flow graph, is defined in the following way:

$$k_E(e_i, e_j) = \begin{cases} 1 & : \xi_E(e_i) = \xi_E(e_j) \\ 0 & : \xi_E(e_i) \neq \xi_E(e_j). \end{cases}$$

C. Decomposition kernel for HCFGs

On the basis of the above kernels a similarity for HCFG is computed by a decomposition kernel denoted $K_{Ker2HCFG}$.

Definition 5:

$$K_{Ker2HCFG}(G_i, G_j) = \sum_{i=1}^{m} \sum_{j=1}^{n} K_S(S_i, S_j), \qquad (4)$$

where m and n are the numbers of nodes in each graph and

$$K_S(S_i, S_j) = k_V(v_i, v_j) + k_{node}(par(v_i), par(v_j)) + \sum_{w_i \in Nb(v_i)} \sum_{w_j \in Nb(v_j)} k_{edge}((v_i, w_i), (v_j, w_j)) k_{node}(w_i, w_j), \quad (5)$$

where each S_i is a substructure of G_i centered around node v_i and consisting of this node, its parent $par(v_i)$, and its neighbourhood $Nb(v_i)$ (containing nodes and edges linking them with v_i).

The kernel defined by equation (4) is a modification of the one proposed in [4], [41], as it uses a binary node kernel given by definition 2 to compare the neighbouring nodes rather than the more complex k_V kernel which was previously used. It lowers the number of times a tree kernel is computed and does not affect the accuracy of the classification.

D. Mapping kernels for HCFGs

The above kernel has to compute the substructure kernel over all possible combinations of substructures but a mapping kernel can limit the number of this calculation by taking into account only those pairs of substructures which belong to the mapping M. In this paper two mappings are proposed.

The first one is relatively straightforward; two substructures S_i of G_1 and S_j of G_2 belong to mapping only if the labels of the nodes around which they are centered (i.e. $v_i \in V_{G_1}$ and $v_j \in V_{G_2}$, respectively), have identical labels. Thus $M_1 = \{(S_i, S_j) | \xi_V(v_i) = \xi_V(v_j)\}$. Such a mapping is transitive, as it is based on the equality of labels.

Definition 6:

$$K_{Map1HCFG}(G_i, G_j) = \sum_{(S_i, S_j) \in M} K_S(S_i, S_j), \quad (6)$$

where $K_S(S_i, S_j)$ is defined as above.

For the second mapping we take into account the position of a given node within the structure of the hierarchical control flow graph. This can be done by taking into account the label of a given node and the label of its parent. Then the mapping $M_2 = \{(S_i, S_j) | \xi_V(v_i) = \xi_V(v_j) \wedge \xi_V(par(v_i)) = \xi_V(par(v_j))\}$. As this mapping is also defined on the basis of the equality relation it is also transitive. The second mapping kernel is defined in the following way:

Definition 7:

$$K_{Map2HCFG}(G_i, G_j) = \sum_{(S_i, S_j) \in M_2} K_S(S_i, S_j), \quad (7)$$

where $K_S(S_i, S_j)$ is defined as above.

V. Experiments and Results

A. Data Preparation

The experiment was carried out on two simple, but representative example programs. Both examples are modified versions of benchmarks commonly used in mutation testing related research [23], [24], [42], [43]. A part of one of the example programs is shown in Fig. 2a. The following two steps were performed for each example:

- generation of mutants
- generation of hierarchical control flow graphs for the mutants

All mutants were generated by muJava tool [39]. For the first example the tool generated 38 mutants, and for the second one - 67 mutants. The tool uses a set of traditional and object-oriented mutation operators [39], [44]. The traditional mutation operators usually modify expressions by replacing, inserting or deleting arithmetical, logical or relational operators or some parts of expressions. Mutation operators related to object-oriented features of Java implemented into the tool refer to the object-oriented features such as encapsulation, inheritance, polymorphism and overloading.

Examples of mutants generated for the method $search(...)$, depicted in 2a are shown in Figs. 3a and b. The one depicted in Fig. 3a is an example of applying a Relational Operator Replacement (ROR), that in this case replaced the condition $values[i] == v$ within the if instruction by $false$. The mutant in Fig. 3b is an example of using an Arithmetic Operator Insertion (AOI). Here the short-cut operator $--$ was inserted into variable i in a $return$ statement. In Figs. 4a and b the hierarchical control flow graphs for the above mutants are depicted. It can be observed in Fig. 4a that the subtree representing the condition within the if instruction was replaced by a single node labeled with value $false$. The HCFG representation of the second mutant (in Fig. 4b) differs from the graph representation of the original by the replacement of a single node labelled i, inside the node representing one of the $return$ statements, by the subtree representing the expression $--i$.

The experimental data includes also test suites. For the first example three test suites were provided and for the second example (more complex one) - five test suites were used.

B. Classification results

The k$-NN$ classification algorithm was used on mutants generated for the two examples. Previously graph edit distance and a distance computed from simple HCFG kernel were used in this algorithm [4]. In this paper distances are computed from the three new kernels described in sections IV-C and IV-D. (Denoted as Ker2HCFG - defined by equation 4 and Map1HCFG and Map2HCFG - by equations 6 and 7, respectively).

In the experiments for the first example all three test suites provided were used. The set of mutants was divided into three parts of similar size, the first was used as a training set and the remaining two as instances to classify. The division of mutants was guided by the type of mutation operators used to obtain particular mutants. That is each set was generated in such a way that it consisted of mutants generated by all operators. Moreover the proportion of mutants generated by a given operator in each set was related to the proportion of such mutants in the full set. The process of dividing the set of mutants was repeated five times resulting in obtaining different

```
public int search(int v)
{
   int i;
   for(i=0;i<size;i=i+1)
     if(false)
        return i;
    return -1;
}
```

a)

```
public int search(int v)
{
   int i;
   for(i=0;i<size;i=i+1)
     if(values[i]==v)
        return --i;
    return -1;
}
```

b)

Fig. 3. An example of a) the ROR mutant and b) one of the AOI mutants of the method from Fig. 2a

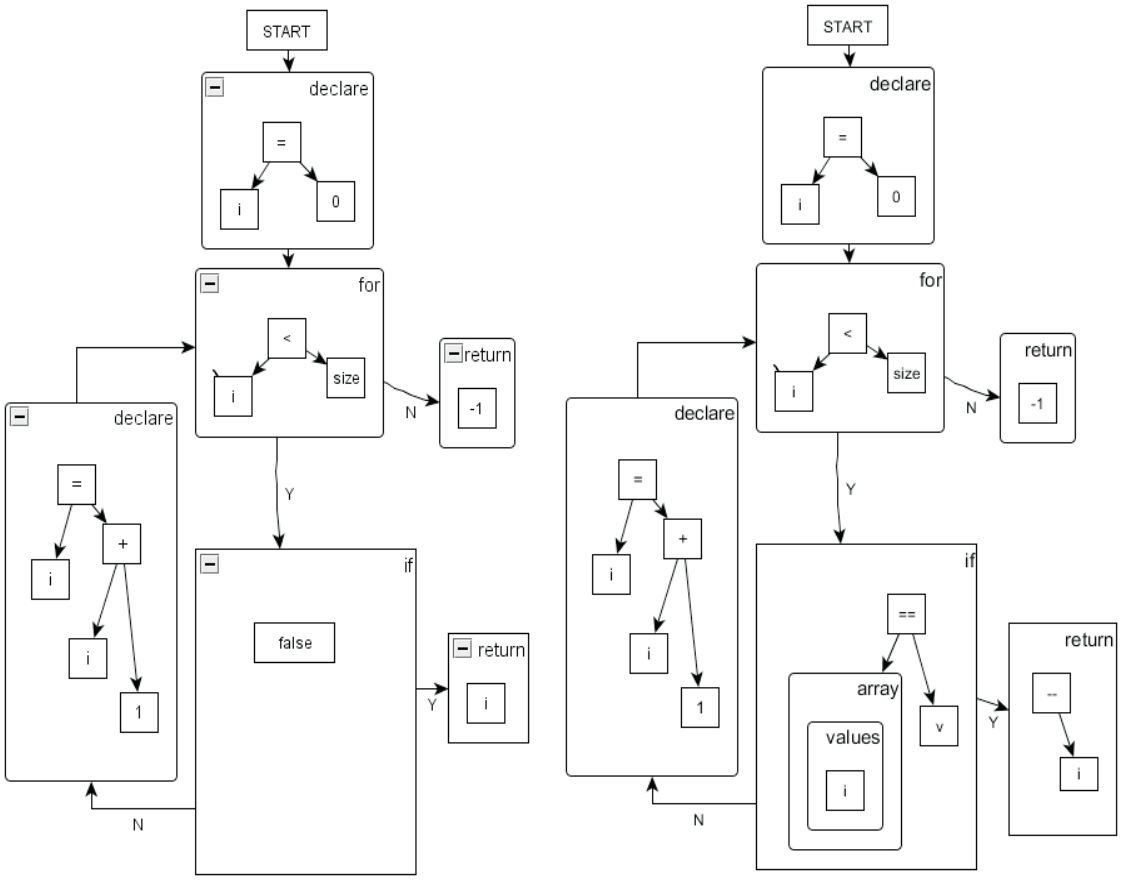

Fig. 4. Examples of flow graphs a) a graph for ROR mutant from Fig. 3a, b) a flow graph for AOI mutant from Fig. 3b

partitions of this set and the classification results obtained were averaged.

Table I presents the results obtained for this example using all three kernels introduced in this paper, and compares them to results obtained with edit distance (from [4], [41]). Parameter k for $k-NN$ was, after some experimental tuning, set to 5 for all experiments. In the first column of the table the percentage of instances classified correctly, i.e. classification accuracy, is shown. The percentage of mutants classified incorrectly is given separately for those classified as detectable, while actually they are not (column labelled incorrect killed) and for those classified as not detected, while they actually are detected by a given test suite (column labelled incorrect not killed). Separating these two values was done because of the meaning of these misclassifications. Misclassifying a mutant not to be detected may lead to overtesting, while the misclassification of the second type is potentially more dangerous as it can result in missing some faulty code. Especially, taking into account that the results are to be used to evaluate the quality of test suites provided for the application. Thus incorrectly classifying a mutant as not detected leads to giving a test suite lower score than actual one, why the second misclassification leads to overvaluation of a given test suite.

As it can be observed in table I the classification performed

TABLE I
THE CLASSIFICATION OF MUTANTS OF EXAMPLE 1

	method	correct	incorrect killed	incorrect not killed
TS 1	GED	65.2%	13.06%	21.74%
	Ker2HCFG	76.1%	5.3%	18.6%
	Map1HCFG	79.2%	4.2%	16.6%
	Map2HCFG	84.5%	3.1%	12.4%
TS 2	GED	78.25%	8.5%	13.25%
	Ker2HCFG	84.9%	5.8%	9.3%
	Map1HCFG	87.6%	5.1%	7.3%
	Map2HCFG	91.3%	5.2%	3.5%
TS 3	GED	82.6%	8.7%	8.7%
	Ker2HCFG	83.1%	4.5%	12.4%
	Map1HCFG	86.4%	4.1%	9.5%
	Map2HCFG	89.8%	3.8%	6.4%

TABLE II
THE CLASSIFICATION OF MUTANTS OF EXAMPLE 2

	method	correct	incorrect killed	incorrect not killed
TS 1	GED	75.7%	12.1%	12.2%
	Ker2HCFG	79.3%	7.1%	13.6%
	Map1HCFG	82.4%	5.5%	12.1%
	Map2HCFG	86.1%	4.5%	9.4%
TS 2	GED	73.4%	6.5%	20.1%
	Ker2HCFG	79.1%	5.3%	15.6%
	Map1HCFG	79.2%	4.8%	16.0%
	Map2HCFG	82.5%	4.5%	13.0%
TS 3	GED	60.5%	26.2%	16.3%
	Ker2HCFG	61.2%	23.7%	15.1%
	Map1HCFG	70.2%	18.4%	11.4%
	Map2HCFG	73.8%	16.1%	10.1%
TS 4	GED	78.2%	10.3%	11.5%
	Ker2HCFG	84.5%	4.25%	11.25%
	Map1HCFG	88.6%	3.5%	7.9%
	Map2HCFG	92.1%	3.1%	4.8%
TS 5	GED	76.4%	11.3%	12.3%
	Ker2HCFG	79.6%	8.2%	12.2%
	Map1HCFG	83.3%	7.6%	9.1%
	Map2HCFG	88.2%	5.0%	6.8%

reasonably well for all test suites. All three new kernels proposed in this paper gave better classification results than results obtained by using graph edit distance. It can be explained by the fact that graph edit distance captures only the structural difference in a graph i.e. it takes into account only a change but not the location of this change. For example the mutant shown in Fig. 4b, where variable i was replaced by $--i$, what resulted in replacing a node be a subtree, and another one, in which variable i in different location undergone the same change would result in identical graph edit distance from other mutants. In the domain of mutation testing the location of the change is as important as the kind of change itself.

The results obtained by the use of the decomposition kernel ($Ker2HCFG$) and the first mapping kernel ($Map1HCFG$) are similar in case of classification accuracy, but the mapping kernel results in a bit smaller number of mutants incorrectly predicted to be detected. But the most important difference between these two kernels is related to the number of kernel calculations which have to be performed. In case of the decomposition kernel it is calculated for each node of each control flow graph, while in case of mapping kernel it is limited to a small subset of the nodes. While the actual number of the calculations can not be estimated, as it depends on the program structure, in typical program no given element or construction should constitute more than several percent of all the elements of the program.

The second mapping kernel ($Map2HCFG$) shows improvement over both previous kernels. It can be explained by the observation concerning the nature of the mutants. Mutant obtained by the introduction of a particular change in a particular location is not necessarily detectable by the same test as the mutant obtained by the introduction of the identical change in different location. The second mapping system contains substructures centered around nodes not only representing identical constructs, but also located within nodes representing identical elements, thus the results obtained for this kernel are more accurate. Moreover, as it takes into account less pairs of the substructures the number of calculations decreases further.

In the second example five test suites were used and the set was divided into four parts of similar size, because of the higher number of mutants. The process of dividing the set of mutants was carried out according to the same criteria as in the first example. And the classification was done in the same way.

Table II presents the results obtained for this example. It may be observed that the results follow similar pattern as for

the first example. It can be noticed that the results for TS 3 were visibly worse than for other suites. Closer inspection seems to suggest that this results from the fact that TS 3 detects only 22 out of 87 mutants and there may occur an over-representation of detectable mutants in the training set thus leading to incorrectly classifying many mutants as detectable.

C. Using the results

Typical use of classification results is predicting the class membership of new elements, in this case new mutants. While it of course would be possible to generate new mutants for a given program and predict whether they would be killed by a given test suite the typical scenario here is different. After the classification is finished for each mutant not in the training set its k nearest neighbours are stored. Then, when a new test suite for the program is provided its needs to be run only against the mutants belonging to the training set, the detectability of the other mutants is decided on the basis of the stored neighbours.

VI. Conclusions and Future Work

The research presented in this paper deals with the problem of reducing the number of mutants to be executed and thus reducing the cost of applying mutation testing by using the classification algorithm. In contrast to other mutant reduction approaches, which are based on some programming language related knowledge, this approach limits the number of mutants to be executed in a dynamic way i.e. depending on the structure of the program for which the mutants were generated.

The application of the mapping kernel template allowed for comparing control flow graphs with better use of the knowledge of the nature of the mutants and at the same time significantly reduce the number of elements to be compared, thus reducing the time required to compute the similarity of programs.

The results of the classification allow to assess new test suites. During the application building process it is common that tests suites are iteratively updated. Each newly updated test suite has to be assessed to check if it fault detection ability is improved. Thus having to run each new test suite only on a fraction of all mutants significantly reduces the time a developer needs to provide a high quality test suite for an application.

Although the results are satisfactory and allow for using this method in practice there are several possible directions for future research on using classification in mutation testing. One of them is related to better use of the knowledge of the way tests are executed. When a test is executed it follows one of many possible paths within the program flow. Using this fact in the way a training set of mutants is selected could possibly improve the classification. This information could also be incorporating into the way the substructures in the kernel function are defined, for example limiting the neighbourhood of the node taken into account only to the nodes on the same path.

Another possible improvement could be based on replacing an arbitral choice of k by a more flexible approach such as the one proposed in [45]. Future research are also planned to involve trying to define a set of features allowing for the description of programs in a vector form. Such a representation would allow for the use of non-kernel based classifiers.

References

[1] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology,* vol. 51, no. 6, 2009, pp. 957-976.

[2] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Transactions on Software Engineering,* 2010, pp. 742-762

[3] V. U. B. Challagulla, F. B. Bastani, I.-L. Yen, and R. A. Paul, "Empirical assessment of machine learning based software defect prediction techniques," *International Journal on Artificial Intelligence Tools,* vol. 17, no. 2,2008, pp. 389-400.

[4] J. Strug, B. Strug, "Machine learning approach in mutation testing," *LNCS,* vol. 764, 2012, pp. 200-214, http://dx.doi.org/10.1007/978-3-642-34691-0_15

[5] J. Strug, B. Strug, "Classifying Mutants with Decomposition Kernel," *LNCS,* vol. 9692,Springer, 2016, pp.644-654, http://dx.doi.org/10.1007/978-3-319-39378-0_55

[6] R. A. DeMillo, R. J. Lipton, F. G. Sayward, "Hints on test data selection: help for the practicing programmer," *Computer,* vol. 11, no. 4, 1978, pp. 34-41, http://dx.doi.org/ 10.1109/C-M.1978.218136

[7] J. H. Andrews, L. C. Briand, Y. Labiche, "Is mutation an appropriate tool for testing experiments?," *in Proc. ICSE'05,* 2005, pp. 402-411, http://dx.doi.org/10.1145/1062455.1062530

[8] M. Harman, Y. Jia, "An analysis and survey of the development of mutation testing," *IEEE Transactions Software Engineering,* vol. 37, no. 5, 2011, pp. 649-678, http://dx.doi.org/10.1109/TSE.2010.62

[9] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R.J. Martin, A. Mathur, E.Spafford, "Design of mutant operators for the C programming language," Department of Computer Science, Purdue University, Lafayette, Indiana, Technical Report SERC-TR-41-P, April, 2006.

[10] B. Aichernig, J. Auer, E. Jobstl, R. Korosec, W. Krenn, R. Schlick, B. V. Schmidt, "Model-based mutation testing of an industrial measurement device," *LNCS,* vol. 8570, 2014, pp. 1-19, http://dx.doi.org /10.1007/ 978-3-319-09099-3_1

[11] A. Derezinska, "Object-oriented mutation to asses the quality of tests," *in Proc. of the 29th Euromicro Conference,* Belek-Antalya, Turkey, 2003, pp. 417-420.

[12] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for Java," *in Proc of ISSTA'14,* San Jose, Bay Area, CA, 2014, pp. 433-436, http://dx.doi.org/10.1145/2610384.2628053

[13] G. Kaminski, P. Ammann, J. Offutt, "Improving logic-based testing," *Journal of Systems and Software,* vol. 86, no. 8, 2013, pp. 2002-2012, http://dx.doi.org/10.1016/j.jss.2012.08.024

[14] S. Kim, J. A. Clark, J. A. McDermid, "Class mutation: mutation testing for object-oriented programs," *in Proc. Net.ObjectDays Conference on Object-Oriented Software Systems Conference'00,* Erfurt, Germany, 2000, pp. 9-12.

[15] J. Strug, "Applying Mutation Testing for Assessing Test Suites Quality at Model Level," *in Proc. of FedCSIS'16,* ACSIS, vol. 8, 2016, pp. 1593-1596, http://dx.doi.org/10.15439/2016F82

[16] J. Strug, "Mutation Testing Approach to Negative Testing," *Journal of Engineering,* vol. 2016, 13 pages, http://dx.doi.org/10.1155/2016/6589140

[17] A. P. Mathur, "Performance, effectiveness, and reliability issues in software testing," *in Proc. COMPSAC'91,* Tokyo, Japan, 1991, pp. 604-605, http://dx.doi.org/10.1109/CMPSAC.1991.170248

[18] A. P. Mathur, E. W. Krauser, "Mutant Unification for Improved Vectorization," Purdue University, West Lafayette, IN, Technique Repoer SERC-TR-14-P, 1988.

[19] S. Hussain, "Mutation clustering," Master's thesis, King's College London, Strand, London, 2008.

[20] C. Ji, Z. Chen, B. Xu, Z. Zhao, "A novel method of mutation clustering based on domain analysis," *in Proc. of SEKE Conference'09,* Boston, Massachusetts, 2009, pp. 422-425.

[21] A.T. Acree, "On mutation," Master's thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.
[22] T. A. Budd, "Mutation analysis of program test data," Master's thesis, Yale University, New Haven, Connecticut, 1980.
[23] A. P. Mathur, W. E. Wong, "An empirical comparison of mutation and data flow based test adequacy criteria," *Software Testing, Verification and Reliability,* vol. 4, no. 1, 1994, pp. 9-31, http://dx.doi.org/10.1002/stvr.4370040104
[24] W. E. Wong, "On mutation and data flow," Master's thesis, Purdue University, West Lafayette, Indiana, 1993.
[25] R. Agrawal, T. Imielinski, A. Swami, "Mining association rules between sets of items in large databases," *in Proc. of ACM-SIGMOD'93,* Washinghton, D.C, 1993, pp. 207-216, http://dx.doi.org/10.1145/170035.170072
[26] J. Han, J. Pei, Y. Yin, R. Mao, "Mining frequent patterns without candidate generation: a frequent-pattern tree approach," *Data Mining and Knowledge Discovery,* vol 8, no. 1, 2004, pp. 53-87, http://dx.doi.org/10.1023/B:DAMI.0000005258.31418.83
[27] A. Inokuchi, T. Washio, H. A. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," *in Proc. of PKDD'00,* Lyon, France, 2000, pp. 87-92.
[28] B. Strug, "Using co-occurring graph patterns in computer aided design evaluation," *LNCS*, vol. 9120, 2015, pp. 768-777 , http://dx.doi.org/10.1007/ 978-3-319-19369-4_68
[29] H. Bunke, K. Riesen, "Recent advances in graph-based pattern recognition with applications in document analysis," *Pattern Recognition,* vol. 44, no. 5, 2011, pp. 1057-1067, http://dx.doi.org/10.1016/j.patcog.2010.11.015
[30] K. Riesen, H. Bunke, "Reducing the dimensionality of dissimilarity space embedding graph kernels," *Engineering Applications of Artificial Intelligence,* vol. 22, no. 1, 2009, pp. 48-56, http://dx.doi.org/10.1016/j.engappai.2008.04.006
[31] B. Scholkopf, A. J. Smola, *Learning with kernels,* Cambridge, MT, MIT Press, 2002.
[32] T. Gartner, *Kernels for structured data,* (Series in Machine Perception and Artificial Intelligence), World Scientific, 2009.
[33] H. Kashima, K. Tsuda, A. Inokuchi, "Marginalized kernels between labeled graphs," *in Proc. of ICML'03,* Washington, DC, 2003, pp. 321-328.
[34] K. M. Borgwardt, H. P. Kriegel, "Shortest-path kernels on graphs," *in Proc. of ICDM'05,* Houston, Texas, 2005, pp. 74-81, http://dx.doi.org/10.1109/ICDM.2005.132
[35] B. Strug, "Automatic design quality evaluation using graph similarity measures," *Automation in Construction,* vol. 32, 2013, pp. 187-195, http://dx.doi.org/10.1016/j.autcon.2012.12.015
[36] M. Collins, N. Duffy, "New ranking algorithms for parsing and tagging kernels over discrete structures, and the voted perceptron," *in Proc. of ACL'02,* Philadephia, Pennsylvania, July, 2002, pp. 263-270, http://dx.doi.org/10.3115/1073083.1073128
[37] D. Haussler, "Convolutional kernels on discrete structures," Computer Science Department, UC Santa Cruz, Technical Report UCSC-CRL, 1999.
[38] K. Shin, T. Kuboyama, "A generalization of Haussler's convolution kernel - mapping kernel and its application to tree kernels," *J. Comput. Sci. Technol,* vol. 25, no. 5, 2010, pp. 1040-1054, http://dx.doi.org/10.1007/s11390-010-1082-7
[39] Y. Ma, J. Offutt, Y. R. Kwon, "Mujava: a mutation system for Java," *in Proc. ICSE'06,* Shanghai, China, 2006, pp. 827-830, http://dx.doi.org/10.1145/1134285.1134425
[40] M. Chein, M.-L. Mugnier, G. Simonet, "Nested graphs: a graph-based knowledge representation model with FOL semantics," *in Proc. of KR'98,* Trento, Italy, 1998, pp. 524-535.
[41] J. Strug, B. Strug, "Using structural similarity to classify tests in mutation testing," *Applied Mechanics and Materials,* vol. 378, 2013, pp. 546-551, http://dx.doi.org/10.4028/www.scientific.net/AMM.378.546
[42] P. G. Frankl, S. N. Weiss, C. Hu, "All-uses vs mutation testing: an experimental comparison of effectiveness," *Journal of Systems and Software,* vol. 38, no. 3, 1997, pp. 235-253, http://dx.doi.org/10.1016/S0164-1212(96)00154-9
[43] A. J. Offutt, J. Pan, K. Tewary, T. Zhang, "An experimental evaluation of data flow and mutation testing," *Software, Practice and Experience,* vol. 26, no. 2, 1996, pp. 165-176, http://dx.doi.org/10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K
[44] Y.-S. Ma, Y.-R. Kwon, A. J. Offutt, "Inter-class mutation operators for Java," *in Proc. ISSRE'02,* Annapolis, MD, 2002, pp. 352-366.
[45] H. Wang, I.Düntsch, G. Gediga, and G. Guo, "Nearest Neighbours without k", *In: Barbara Dunin-Keplicz, Andrzej Jankowski, Andrzej Skowron, and Marcin Szczuka, editors, Monitoring, Security, and Rescue Techniques in Multiagent Systems, Advances in Soft Computing*, vol. 28, 2005, pp 179-189, http://dx.doi.org/10.1007/3-540-32370-8_12.