INFORMATICA, 2016, Vol. 27, No. 4, 819–842

DOI: http://dx.doi.org/10.15388/Informatica.2016.113

Root Cause Analysis of Large Scale Application Testing Results

Rūdolfs OPMANIS, Paulis ĶIKUSTS, Mārtiņš OPMANIS*

Institute of Mathematics and Computer Science, University of Latvia
Rainis blvd. 29, Riga, LV1459, Latvia
e-mail: rudolfs.opmanis@gmail.com, paulis.kikusts@lumii.lv, martins.opmanis@lumii.lv

Received: August 2015; accepted: August 2016

Abstract. We present a new root cause analysis algorithm for discovering the most likely causes of differences found in testing results of two versions of the same software. Problematic points in test and environment attribute hierarchies are presented to a user in a compact way which in turn allows saving time on test result processing. We have proven that for clearly separated problem causes our algorithm gives an exact solution. Practical application of described method is discussed.

Key words: root cause analysis, software regression testing, hierarchy graphs.

1. Introduction

Testing results are an important indicator of overall application state regardless of whether it is in active development or maintenance phase. During the active development phase, testing results can help developers to understand if the current status is consistent with the plan and manage required resources accordingly. However, during the maintenance phase, testing results are necessary to verify if application after internal or external changes still performs as expected. Software maintenance has been identified as the most costly and difficult phase of software life cycle, so ability to measure a quality of the application is critical for managing cost and time (Briski *et al*., 2008).

If an application has a simple structure and is designed for a single environment, then testing can be performed, and testing results can be processed easily. We address large applications, i.e. systems of a complex structure implementing a lot of different features designed for various environments, e.g. multiple operating systems, browsers, architectures. Testing of such applications on several environments demands coverage by huge data sets and may produce a massive volume of testing results. This volume grows enormously in *software regression testing* (SRT) (Dustin, 2002) when the same tests are run on new builds of software frequently. Besides, to fulfill principle "any testing process should include a thorough inspection of results of each test" (Myers *et al*., 2012) processing of testing results inevitably has to be automated. Such automation is the subject of our paper.

*Corresponding author.

It should be pointed out that we deal only with *results of regression testing*, not with design and content of tests. Therefore, such approach is suitable for any software development project regardless of programming language or software development methodology as long as testing is performed in a reasonably frequent manner. Testing personnel can use our approach on top of almost any testing process without essential additional effort.

Executing of individual tests in a particular environment results in execution status where "successful" or "failed" are the most usual (see Section 3). Not all results of failed tests are equally important. For example, failure of a particular test in all tested environments after most recent changes in application points to a more serious problem than the expected failure known for past month in one environment. To point out significant failures we use technique known as *root-cause analysis* (RCA) (Wilson, 2014; Rooney and Vanden Heuvel, 2004) including hierarchical structuring and higher abstraction level from orthogonal defect classification (IBM, 2013; Chillarege, 2013). We present a new RCA algorithm of processing SRT results of two builds to suggest the most likely causes of differences between test execution statuses.

RCA is developed and used in various contexts and fields (Wilson, 2014; Rundle, 2003) and comprises several phases. Our algorithm corresponds to root cause identification phase (Rooney and Vanden Heuvel, 2004) and works with test hierarchies, attributed hierarchies of testing environments, and test results acquired by executing test cases in testing environments. Although test organization in hierarchies is not mandatory, such organization will allow the proposed algorithm to generalize problems by features. Similarly, having multiple attributes for testing environments are not necessary, but providing them to the algorithm will let it produce results that are more compact. So we offer a possibility to automatically group outcomes of the testing process in the larger blocks representing essential aspects of the developed software.

We also involve *graph-based technique* (GBT) to perform RCA since a graph is a well-defined, well-studied structure appropriate to represent both hierarchies. We construct directed acyclic graphs with attributed vertices representing objects on which failure may take place. Arcs represent structural dependencies between objects and have direction from more general objects to specific.

During RCA, we obtain a subgraph of a hierarchy of objects where the most important failures appear. Source vertices of this subgraph give a clue to developers about the most general features implementation of which cause failure. If the reported object seems too general for a developer, we offer a possibility to drill down till the most specific objects (Opmanis *et al*., 2016).

Altogether our algorithm allows speeding up a process of finding a cause of failure pointing immediately to the most significant deteriorations, therefore, avoiding time-consuming routine work like examination of huge test logs. Such pointing helps the user not to overlook serious deteriorations. This is especially important if the code is maintained by several developers when it may take weeks to find a proper cause of a failure in unstructured testing outcomes. Therefore, overall time and cost of software development and maintenance are reduced.

There are essentially different approaches which help to reduce time and cost of application maintenance like still relevant software impact analysis (Bohner and Arnold, 1996)

that investigates software to predict affected items to estimate required effort, cost, and time to implement a new feature or make some adjustment in the existing software product. However, this approach requires very thorough planning and continuous dependency management that might not be available for large projects or projects that are handed over between maintainers.

This paper is organized as follows: in Section 2 we look at related work, in Section 3 we define terms used in the paper, in Section 4 we give data model and a very detailed description of algorithm along with proofs of two of its properties. In Section 5 results of a practical application of our algorithm are discussed. Conclusions and directions of possible future work are described in Section 6.

2. Related Work

The fields SRT, RCA, and GBT, are three cornerstones of our approach and are well described in the literature. We use a combination of methods and concepts of all three fields and to the best of our knowledge, there are no relevant publications for the entire triple RCA-SRT-GBT. Therefore, in this Section, we focus on some publications where two of the fields are represented.

In the context of our paper, an essential combination is the pair SRT-RCA. In general, testing comprises of finding causes of failures that in its turn is an essence of RCA. In the last decades RCA is successfully involved in software development and testing process (Ruberto, 2013; Leszak *et al*., 2002; Kataoka *et al*., 2011). Usage of RCA methodology in software development process is thoroughly described in Linders (2014).

This pair SRT-RCA is also represented in other approaches that are out of the scope of our paper. Say, unlike (Zeller, 2002), we are not analysing source code but use only testing results.

As we use graphs to reveal causes of unwilling software effects using structural relations between involved hierarchy objects, we have special interest about another pair RCA-GBT, which appears in various contexts.

Graph-like *cause-and-effect* diagrams (also known as Ishikawa diagrams) (Tague, 2005, pp. 247–249) are already used in RCA, though without direct referencing to graphs. Also, general tools for RCA based on graph processing are developed by software industry (Tom Sawyer Software, 2005).

RCA is used together with GBT exploiting graph drawing as a very powerful tool to comprehend object structural dependencies. In a field of Internet security, this approach is used to discover root causes of disruptions in Internet traffic (Ohrimenko *et al*., 2013). To create attributed diagrams of object hierarchies in the next Sections, we also applied graph drawing algorithms.

In Steinder and Sethi (2004), a graphical model, called a fault propagation model for communication networks, is investigated. In Marvasti *et al*. (2013), graphs of anomaly events with probability connections of complex IT infrastructures are used to reliably predict root causes of problems. In these models, as in ours, RCA is performed on the underlying graph of objects and relations among them.

Interesting connection between graphs and analysis of failure causes in manufacturing using digraph and matrix methods is presented in Rao (2010), Dev *et al*. (2014).

The third pair SRT-GBT in a context of software testing is well known when software structure itself is represented by a graph (Biswas *et al*., 2011). However, this approach completely differs from ours. As a consequence, there are no relevant publications also for the entire triple RCA-SRT-GBT.

3. Preliminaries

Test is the smallest entity for a software product or module testing. Execution of software artifact in some *environment* using data from a particular test ends with *test execution status*: a value from a fixed non-empty set of available *outcomes*, e.g. "successful", "not completed", "failed", "predictably incorrect", "inconclusive", "unclear", "runtime error NNN", "division by 0", "crash".

Tests are grouped in *testgroups* to test some feature or software component thoughtfully. Each testgroup may be either *simple* or *composite*. A simple testgroup consists of one or more tests with similar characteristics for testing of a particular software component or feature.

A composite testgroup consists of one or more testgroups and during testing is considered as a single object. From graph perspective, testgroups as nodes are organized in tree-like hierarchical structure and altogether constitute a directed forest. Each simple testgroup is a leaf in this forest, and each composite testgroup is in a parent-child relation with each of its immediate constituents. Further we use 'simple testgroup' and 'leaf-testgroup' as synonyms.

For example, in one leaf-testgroup there may be tests checking data import from XML source, in another there may be tests checking data import from a database and these testgroups may be included in a higher level testgroup checking data import in general. Such approach allows to directly point to functional parts of tested software when some erroneous testgroup is found out.

Environments are parameterized objects characterized by various *attributes*, e.g. operating system, and their *values*, e.g. 'Windows', 'Linux', 'Mac OS'. For web applications, one of the attributes may be a browser, for mobile devices – application. We assume that several attributes do not share the same attribute value. Therefore, any attribute value is enough to identify an attribute. Each environment is identified by an *attribute value set* where each attribute is represented by at most one value. For different environments, these attribute value sets may be of different size. Any non-empty subset of environments attribute value set is called *attribute bundle* (*bundle* or *subbundle* for convenience).

Attribute bundles are organized hierarchically using relation *be a subset of*. In contrary to testgroups establishing forest, sets of environment attributes as nodes constitute more general structure – directed acyclic graph (DAG). Each attribute bundle is in a parent-child relation with each superset containing one more attribute, and each child may have several parents.

We reference hierarchy of testgroups and hierarchy of attribute bundles as *hierarchies* and their elements as *hierarchy objects*.

Build is one particular version of the same software product to be tested.

Testrun is an execution of a build using simple testgroup in a specified testing environment to obtain *testrun result*: a tuple of integers, where each integer is the number of tests having particular status for each of available test outcomes.

We are interested in a comparison of testrun results of two chosen builds called *reference build* and *active build*. The comparison makes sense just if the testgroup and the environment are the same for the both testruns. This requirement is satisfied in regression testing. Each comparison may report *deterioration* – an observation that the active build testrun result is worse than the reference build testrun result. A simple way is to report a deterioration if the number of failed tests increases.

Our purpose is to introduce a measure of deterioration significance for hierarchy objects separately for each hierarchy and using this measure to recognize *deterioration objects*. Such testgroups and/or attribute bundles must attract developer's attention at first. Therefore, we think unreasonable to report a large number of less significant deterioration objects. Instead, we consolidate information about them into a smaller number of *resulting deterioration objects* of higher hierarchy levels. As the result of the proposed analysis two collections of resulting deterioration objects – *resulting deterioration testgroups* and *resulting deterioration attribute bundles* are provided.

4. Root Cause Analysis

We describe testing process by the following elements:

- sequence $\mathcal{B}$ of builds,
- set $\mathcal{G}$ of testgroups organized hierarchically,
- set $\mathcal{E}$ of environments,
- set $\mathcal{R}$ of testruns.

In addition, we denote by $\mathcal{L}$ the set of all leaf-testgroups of $\mathcal{G}$; by $avs(e)$ the attribute value set of environment $e \in \mathcal{E}$; by $\mathcal{A} = \bigcup_{e \in \mathcal{E}} 2^{avs(e)} - \emptyset$ the set of all $\mathcal{E}$ subbundles organized hierarchically.

For a chosen reference build b_0 and active build b_1 we consider two testrun sets $\mathcal{R}_0$ and $\mathcal{R}_1$, where $\mathcal{R}_0 \subseteq \mathcal{R}$ is a set of testruns using b_0 and a subset of $\mathcal{L} \times \mathcal{E}$, and $\mathcal{R}_1 \subseteq \mathcal{R}$ is a set of testruns using b_1 and a subset of $\mathcal{L} \times \mathcal{E}$. Each testrun is characterized by tuple (bld, grp, env, res), where bld is a build, grp is a leaf-testgroup, env is an environment, res is a testrun result. Clearly, $r.bld = b_i$ for each $r \in \mathcal{R}_i$ $(i = 0, 1)$.

Results of every two *coupled* testruns $r_0 \in \mathcal{R}_0$ and $r_1 \in \mathcal{R}_1$ where $r_0.grp = r_1.grp$ and $r_0.env = r_1.env$ are subjects of comparison which is performed by a special boolean function $isDeterioration(r_0.res, r_1.res)$ returning *true* when r_1 result is worse than result of r_0. The basis of our deterioration analysis is a *deterioration set* D consisting of testruns $r_1 \in \mathcal{R}_1$, that with respect to corresponding element $r_0 \in \mathcal{R}_0$ have $isDeterioration(r_0.res, r_1.res) = true$.

4.1. *Algorithmic Principles of Deterioration Analysis*

As said before, the hierarchies of testgroups and environment attribute bundles are represented by DAGs and aims of analysis are similar. Therefore, analysis of both hierarchies will be carried out in a similar manner:

- calculating basic significance characteristics (Algorithm 1),
- thresholding significance values (Algorithm 2),
- calculating coverage of hierarchy objects within a deterioration set,
- DAG-based two-stage filtering (*sink refining* and *source refining*).

Each step is described below in details.

4.1.1. *Calculating Basic Significance Characteristics*

For all testgroups and attribute bundles, we introduce two functions *det* and *com* calculated by Algorithm 1 which iterates through leaf-testgroups and environments.

Algorithm 1: Calculating basic data: deterioration set D, values of *det* and *com*.

Input: testgroup set $\mathcal{G}$, environment set $\mathcal{E}$,
testrun set $\mathcal{R}_0$ corresponding to reference build b_0,
testrun set $\mathcal{R}_1$ corresponding to active build b_1
Output: deterioration set D, values of *det* and *com*
begin
  $D := \emptyset$
  for all $\mathcal{G}$ and $\mathcal{A}$ elements initialize *det* and *com* values to 0
  foreach $l \in \mathcal{L}$ **do**
    foreach e *in* $\mathcal{E}$ **do**
      $r_0 :=$ a testrun of $\mathcal{R}_0$, built on pair (l, e)
      $r_1 :=$ a testrun of $\mathcal{R}_1$, built on pair (l, e)
      if r_0 *exists and* r_1 *exists* **then**
        foreach $g \in predecessors(l)$ **do** increase $com(g)$ by 1
        foreach $a \in predecessors(avs(e))$ **do** increase $com(a)$ by 1
        if $isDeterioration(r_0.res, r_1.res)$ **then**
          add r_1 to D
          foreach $g \in predecessors(l)$ **do** increase $det(g)$ by 1
          foreach $a \in predecessors(avs(e))$ **do** increase $det(a)$ by 1
        end if
      end if
    end foreach
  end foreach
end

For each leaf-testgroup $l \in \mathcal{L}$ the value $det(l)$ indicates how many times the leaf-testgroup l occurs as constituent of testruns of D and $com(l)$ indicates the number of coupled testruns $(r_0 \in \mathcal{R}_0, r_1 \in \mathcal{R}_1)$ where $r_1.grp = r_0.grp = l$.

If a testgroup $g \in G$ is not a leaf-testgroup, then the value $det(g)$ is a recursive sum of *det* values over all g children $g_1, g_2, \ldots$: $det(g) = det(g_1) + det(g_2) + \ldots$, and, analogously, $com(g) = com(g_1) + com(g_2) + \ldots$.

In their turn, for each bundle $a \in \mathcal{A}$ the value $det(a)$ indicates how many times a occurs as a subbundle of the attribute value set $avs(e)$ of an environment e, where e is a constituent of a testrun of D. For each bundle $a \in \mathcal{A}$ the value $com(a)$ indicates the number of coupled testruns $(r_0 \in \mathcal{R}_0, r_1 \in \mathcal{R}_1)$ where a occurs as a subbundle of $avs(e)$ and $r_1.env = r_0.env = e$.

Based on parent-child relation let's by $predecessors(n)$ denote the set of DAG nodes being predecessors of node n together with n itself, and by $successors(n)$ denote the set of DAG nodes being successors of node n together with n itself. Let's say that node n_2 is *reachable* from node n_1 if $n_2 \in successors(n_1)$, and node n is reachable from a node set N if n is reachable from some element of N.

We rely on the consideration that sources of deterioration should manifest themselves via hierarchy objects that appear most frequently in the testrun set D. This consideration is the basis for the proposed deterioration analysis.

For a hierarchy object x its significance $sig(x)$ is characterized by its amount of deteriorations $det(x)$, i.e. the number of testruns it is involved with and deterioration that occurs, relate to the entire number of comparisons $com(x)$ it is involved with: $sig(x) = \frac{det(x)}{com(x)}$ (0, if $com(x) = 0$).

Of course, besides this consideration of a statistical nature, object importance may be taken into account. Say, the importance of tests for a customer or in comparison with allied program products. As well more complicated expressions of already introduced functions like $\frac{det(x)}{com(x)} \times \frac{det(x)}{|D|}$ may be useful. However, investigation of such alternatives is quite complicated and out of the scope of this paper.

4.1.2. *Thresholding Significance Values*

We use found deterioration significance values of objects of the both hierarchies to locate the most problematic points of software tested.

Reasonably, problematic points are indicated by hierarchy objects with the maximum *sig* value. However, usually there are just one or very few hierarchy objects having the largest *sig* value and restricting our interest just to them, so we neglect objects with values close to the maximum. To maintain also such cases as a base of analysis result we will use object collection having *sig* values not lower than a particular threshold value found by *half-sum thresholding* procedure based on Algorithm 2 that finds *dominating greatest values* in a non-decreasing ordered sequence of *sig* values.

Figure 1 illustrates a bar diagram for 16 *sig* values: 0.077, 0.25, 0.294, 0.333, 0.357, 0.4, 0.467, 0.552, 0.563, 0.571, 0.573, 0.587, 0.613, 0.643, 0.647, 0.867. This is a typical case with exactly one maximum *sig* value. On these data Algorithm 2 returns index 11, so the corresponding threshold value is $L[11] = 0.573$ and the result of the half-sum

Algorithm 2: Calculating dominating greatest values of an ordered nonnegative real number list.

Input: A non-empty list L of non-decreasing nonnegative real numbers
Result: index of the first dominating value
begin
  $k_{\max} := length(L)$ **if** $k_{\max} = 1$ **or** ($k_{\max} = 2$ **and** $L[1] = L[2]$) **then**
    return 1
  else
    return greatest index k satisfying
    $L[1] + L[2] + \cdots + L[k1] < L[k] + L[k+1] + \cdots + L[k_{\max}]$
  end if
end

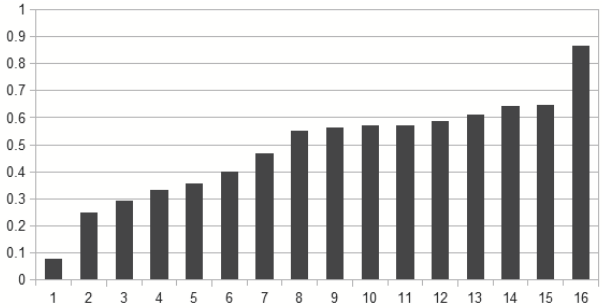

Fig. 1. Bar diagram of *sig* values for the example from Fig. 2.

thresholding procedure are six objects ensuring broader view of the analysed hierarchy than using just a maximum.

There can also be thresholding approaches different from the described. One of them is to look for the biggest increase between two consecutive values. In the described example the corresponding threshold value is $L[16]$ which is the maximum in L. However, deeper analysis shows instability of such criterion – even in this example close to being a threshold is the value $L[2]$ splitting the list only formally.

Another approach is histogram-inspired analysis – find the longest subsequence of consecutive close enough values and split right after it (or right before if subsequence includes the greatest value). In the given example such value is $L[12]$; however, this approach also seemed not stable enough and to be relatively complicated.

Executing the selected thresholding procedure on a hierarchy, we get a *thresholded set* of all hierarchy objects having *sig* value not less than the corresponding threshold value: $\mathcal{G}'$ from the testgroup hierarchy $\mathcal{G}$ and $\mathcal{A}'$ from the attribute bundle hierarchy $\mathcal{A}$.

It turns out that the thresholded sets still may retain comparatively many hierarchy objects sometimes providing redundant information. Therefore, the final stage of the algorithm is a specific refining procedure reducing redundancy and amount of reported deterioration objects.

4.1.3. *Calculating Coverage of Hierarchy Objects Within Deterioration Set*
The refining procedure consists of two consecutive graph filters. The first one takes into account an additional important parameter *covering number*: for a hierarchy object x covering number $cov(x)$ denotes the number of deterioration objects reachable from x and having no successors. For each testgroup $g \in \mathcal{G}'$ the function value $cov(g)$ is the number of g successors that belongs to the set $\mathcal{G}'$ and are leaf-testgroups. For each attribute bundle $a \in \mathcal{A}'$ the function value $cov(a)$ denotes the number of $\mathcal{A}'$ bundles that are attribute value sets of environments from the deterioration set D containing a as a subbundle.

4.1.4. *DAG-Based Two-Stage Filtering*
The first, *sink refining filter* is defined for the both thresholded sets by the following rules:

- **if** $g_1 \in \mathcal{G}'$ is a parent of $g_2 \in \mathcal{G}'$ and $cov(g_1) = cov(g_2)$, **then** g_1 should be excluded from $\mathcal{G}'$.
- **if** $a_1 \in \mathcal{A}'$ is a parent of $a_2 \in \mathcal{A}'$ and $cov(a_1) = cov(a_2)$, **then** a_1 should be excluded from $\mathcal{A}'$.

The meaning of these rules: if an object x_1 of a hierarchy is a predecessor of an object x_2 of the same hierarchy and from x_1 are reachable exactly the same objects without successors as from x_2, this relation alone is not a reason to report x_1 as a deterioration object of the tested software since unnecessary generalization of a deterioration object takes place. In other words, the procedure keeps the object x_2 as more precise specialization if compared with x_1.

The second, *source refining filter* is defined for the both thresholded sets by the following rules:

- **if** testgroups g_1 and g_2 after applying the first filter still belong to $\mathcal{G}'$ and g_1 is a predecessor of g_2 in $\mathcal{G}$, **then** g_2 should be excluded from $\mathcal{G}'$.
- **if** bundles a_1 and a_2 after applying the first filter still belong to $\mathcal{A}'$ and a_1 is a predecessor of a_2 in $\mathcal{A}$, **then** a_2 should be excluded from $\mathcal{A}'$.

The meaning of these rules: if an object x_2 from the hierarchy is a successor of an object x_1 from the same hierarchy, this relation alone is not a reason to report x_2 as a deterioration object of the tested software since unnecessary specialization of a deterioration object takes place. In other words, the procedure keeps x_1 because it consolidates deterioration information about successor objects from the same hierarchy. For attribute bundles it may also be explained: if an intersection of bundles is non-empty and is present in the same attribute bundle set, then intersecting bundles are excluded, and only the intersection is retained.

From graph perspective the first filter keeps sink objects of subgraphs of a particular hierarchy determined by equality classes of function *cov*. The second filter in its turn keeps source objects of the hierarchy determined by hierarchy objects remaining after the first filter. So the corresponding implementations are straightforward.

4.1.5. *Justification of the Proposed Approach*
In the next sections processing of both hierarchies is discussed and the proposed approach is justified by a formal proof for cases when deterioration sources are *clearly located*.

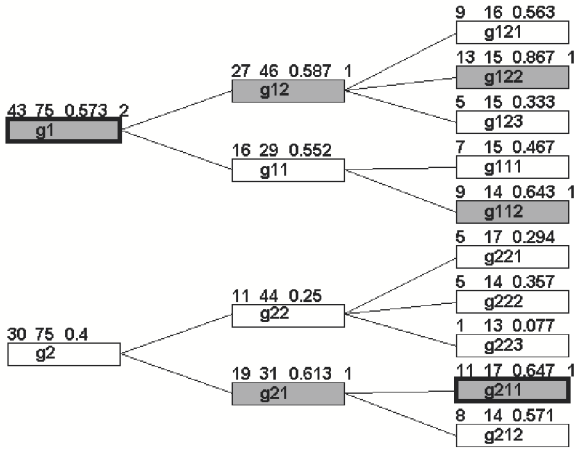

Fig. 2. Example hierarchy of testgroups.

Clearly located sources are objects from some deterioration object set S having the following properties:

- for any coupled reference build testrun r_0 and active build testrun r_1 all cases when the value of $isDeterioration(r_0.res, r_1.res)$ is *true* are caused by **exactly one** deterioration object;
- each element of S belongs to a distinct connected component of the same hierarchy.

Under additional specified conditions depending on the hierarchy type, the set of clearly located deterioration objects can be found precisely, and this is formally proved in Sections 4.2.2 and 4.3.2.

When these conditions are not satisfied, e.g. failure of a particular testrun is caused by more than one object from the same hierarchy and unique reason for the failure can not be discovered, our analysis algorithms work as heuristics.

4.2. *Root Cause Analysis of Testgroups*

4.2.1. *Processing of Testgroup Hierarchy*

As stated above, a testgroup is an intrinsic element within the entire hierarchical structure of the testgroup set $\mathcal{G}$ and is characterized by its relationship with other testgroups via parent-child relations. Each testgroup is identified by a unique name. Every testgroup has exactly one parent (if any), and some amount of children (if any).

In Fig. 2 an example hierarchy of testgroups is shown. Testgroup names are labels placed on nodes. Values of *det*, *com*, and *sig* are added from the left above the corresponding nodes in named order. Nodes of $\mathcal{G}'$ testgroups are supplemented with the fourth parameter: the covering number *cov*. These nodes are coloured gray.

The bar diagram in Fig. 1 illustrates all 16 *sig* values of the example depicted in Fig. 2. The thresholding procedure based on Algorithm 2 returns index 11, so the found threshold value is $L[11] = 0.573$.

The final stage of processing procedure is two-stage filtering refining the of set $\mathcal{G}'$.

In the example (Fig. 2), after applying the sink refining filter, testgroups that are excluded from $\mathcal{G}'$ are g12 and g21. The testgroup g21 and its successor g211 belong to $\mathcal{G}'$ and $cov(\text{g21}) = cov(\text{g211}) = 1$. There is no reason to blame the testgroup g21, even more because it contains also the testgroup g212, within which tests that are performed give relatively better results. Likewise the testgroup g12 and its successor g122 belong to the thresholded set $\mathcal{G}'$ and $com(g12) = com(\text{g122}) = 1$.

Further, after applying the source refining filter, also g112 and g122 as successors of g1 are excluded from $\mathcal{G}'$. Therefore, in refined $\mathcal{G}'$ there remain only testgroups g1 and g211. These testgroups constitute the final result of our analysis and in Fig. 2 are emphasized by bold frames.

We would like to add that in the first component of the example, before applying both refining filters, the testgroup g1 together with its successors g12, g122, g112 were in the thresholded set $\mathcal{G}'$, but the analysis concludes that focus of attention must be paid to the software aspect tested by g1.

4.2.2. *Justification of the Proposed Procedure*

Let $leafs(g) = \mathcal{L} \cap successors(g)$ denotes the set of leaf-testgroups that are reachable from g, and for an arbitrary testgroup set $G \subseteq \mathcal{G}$ denote by $leafs(G)$ the set of leaf-testgroups reachable from some element of G. By $groups(D)$ denote the set of leaf-testgroups that correspond to the testruns of a deterioration set D.

The following conditions are based on ones stated at Section 4.1.5. A set $G \subseteq \mathcal{G}$ is a set of clearly located testgroups if

(1) all active build testruns r_1 together with coupled testruns r_0 have property $isDeterioration(r_0.res, r_1.res) \leftrightarrow r_1.grp$ is reachable from G. Note that an equivalent form of the right side is $leafs(G) = groups(D)$;
(2) each element of G belongs to a distinct connected component of $\mathcal{G}$. Note that for each two distinct elements $g_1, g_2 \in G$: $predecessors(g_1) \cap predecessors(g_2) = \emptyset$;
(3) all G elements are either leaf-testgroups or have at least two children.

Proposition 1. *If for a deterioration set D there exists a testgroup set G satisfying conditions* (1)*,* (2) *and* (3)*, then the result of the testgroup analysis procedure is exactly G.*

Proof. Meaning of the fragment of Algorithm 1 calculating *det* and *com* values for leaf-testgroups may be expressed as:

r_0 = a testrun of $\mathcal{R}_0$, built on pair (l, e)
r_1 = a testrun of $\mathcal{R}_1$, built on pair (l, e)
if r_0 *exists and* r_1 *exists* **then**
    increase $com(l)$ by 1
    if l *is reachable from* G **then** increase $det(l)$ by 1
end if

Hence for all $l \in leafs(G)$ is $0 \leqslant det(l) \leqslant com(l)$. Moreover, for l which is not reachable from G, $det(l) = 0$ because increasing of $det(l)$ is skipped. If l is reachable from G, then *com* and *det* values during calculations grow simultaneously, hence $det(l) = com(l)$.

Thus for significance values *sig* of $l \in leafs(G)$ we have

$$sig(l) = \begin{cases} 1, & \text{if } l \text{ is reachable from } G, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Now determine the *sig* values of composite testgroups.

If a testgroup g is not a leaf-testgroup, then it has children $g_1, g_2, \ldots$, and from Section 4.1.1 $det(g) = det(g_1) + det(g_2) + \ldots$, and $com(g) = com(g_1) + com(g_2) + \ldots$.

Hence

$$sig(g) = \frac{det(g)}{com(g)} = \frac{det(g_1) + det(g_2) + \ldots}{com(g_1) + com(g_2) + \ldots}. \tag{2}$$

For each testgroup g the following cases are possible:

(a) g is reachable from some element of G,
(b) g is a predecessor of some element of G,
(c) g satisfies neither (a) nor (b).

In case (a), when the testgroup g is reachable from some element of G, leaf-testgroups reachable from g are also reachable from this element of G, hence all such leafs belong to $leafs(G)$. If all children of g are leaf-testgroups, then $det(g_i) = com(g_i)$, $i = 1, 2, \ldots$, and from (2) immediately follows $sig(g) = 1$. Recursively backtracking in direction of g parents till g ancestor from G, we see that all testgroups on this predecessor path also have *sig* values equal to 1.

In case (b), at least one leaf-testgroup is reachable from g and belongs to $leafs(G)$, so $det(g) > 0$, and hence $sig(g) > 0$.

And finally, in case (c), when the testgroup g is not a successor nor a predecessor of elements of G, no leaf-testgroup from $leafs(g)$ belongs to $leafs(G)$, and hence $sig(g) = 0$. So, summarizing all three cases we have:

$$sig(g) = \begin{cases} 1, & \text{if } g \text{ is reachable from some element of } G, \\ 0 < \cdots \leqslant 1, & \text{if } g \text{ is a predecessor of some element of } G, \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Now, by the thresholding procedure based on Algorithm 2, all testgroups with *sig* value 1 are included into the set $\mathcal{G}'$ and there are no testgroups with *sig* value 0 in this set.

Due to condition (2) in order to complete the proof, it is enough to examine some separate connected component of $\mathcal{G}$ that contains at least one testgroup with *sig* value greater than 0. In any such component there exists exactly one testgroup from G. Denote by C this component and consider the set $G_{(1,2)} = \{g \in C \mid leafs(g) = leafs(C \cap \mathcal{G}')\}$ every element of which satisfies conditions (1) and (2). Set $G_{(1,2)}$ is non-empty because

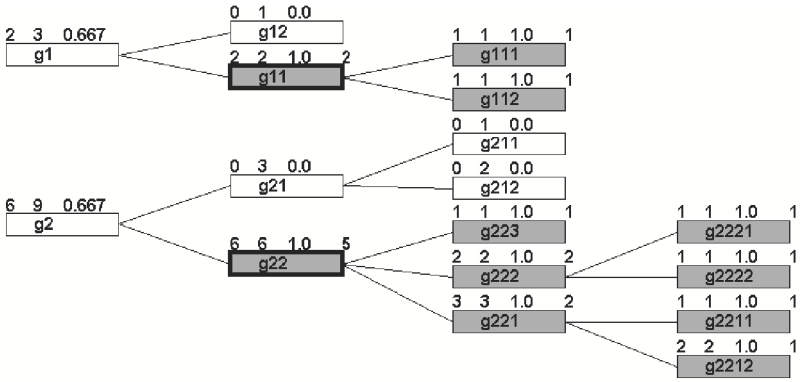

Fig. 3. Testgroup hierarchy with all conditions satisfied, $G = \{g11, g22\}$.

it contains G element having *sig* value 1 and so belonging to $\mathcal{G}'$. Note that *cov* value of all $G_{(1,2)}$ testgroups is the same and the largest possible in C.

Let's examine a particular element $g \in G_{(1,2)}$.

As in $C \cap \mathcal{G}'$ there are only testgroups with *sig* value greater than 0, every of them belongs either to $predecessors(g)/\{g\}$ or to $successors(g)$.

For each testgroup $\bar{g} \in C \cap \mathcal{G}'$ if $\bar{g} \in predecessors(g)/\{g\}$ then by definition $\bar{g} \in G_{(1,2)}$.

For the number of g children we distinguish three mutually exclusive cases:

- If g has exactly one child $\bar{g}$, then $cov(\bar{g}) = cov(g)$ and by definition $\bar{g} \in G_{(1,2)}$. During processing of the first refining filter, g will be excluded from $\mathcal{G}'$.
- If g has at least two children, then for each child $\bar{g}$ the set $leafs(\bar{g})$ is a proper subset of $leafs(g)$, hence $cov(\bar{g}) < cov(g)$ and g is a sink of $G_{(1,2)}$.
- If g has no child, i.e. it is a leaf-testgroup, then $cov(g) = 1$ and g is a sink of $G_{(1,2)}$.

Thus, $G_{(1,2)}$ testgroups in C constitute a path having unique sink g_0 which is also the result of applying the sink refining filter.

Since all other testgroups from $C \cap \mathcal{G}'$ are g_0 successors, the sinks of subgraphs of C determined by *cov* values different from $cov(g_0)$ are reachable from g_0. As the testgroup g_0 is predecessor of all other sinks in C, the second refining filter keeps g_0 as the unique result in C.

By construction, only g_0 also satisfies condition (3) and is the only element in $C \cap G$. □

Note that condition (3) was not used in the reasoning as a requirement. However, this condition cannot be excluded because under just two first conditions there could be more than one valid candidates for G and, therefore, the assertion of Proposition 1 would be false.

We end the chapter with some examples demonstrating meaningfulness of Proposition 1.

The example in Fig. 3 with the set $G = \{g11, g22\}$ illustrates the case when all conditions of Proposition 1 are satisfied. In this example the threshold value for *sig* is 1. Therefore, for each g from the thresholded set all $leafs(g)$ in $\mathcal{G}$ belong to $leafs(G)$ and the union of all $leafs(g)$ is exactly $leafs(G)$.

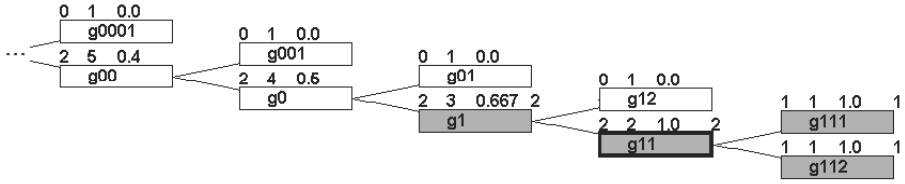

Fig. 4. Testgroup hierarchy with all conditions satisfied, threshold 0.667, $G = \{g11\}$.

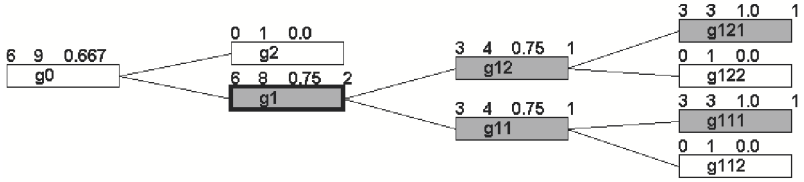

Fig. 5. Testgroup hierarchy where result does not satisfy condition (1).

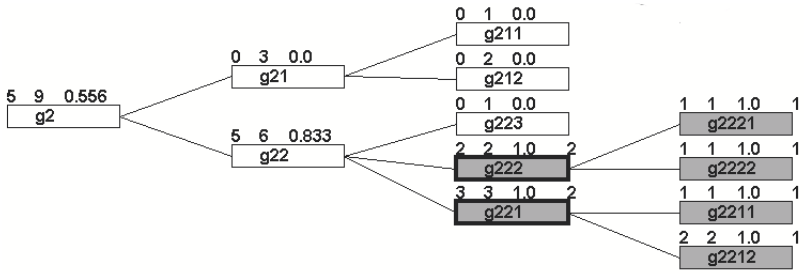

Fig. 6. Testgroup hierarchy where result does not satisfy condition (2).

The example in Fig. 4 illustrates the case when also all conditions of Proposition 1 are satisfied and $G = \{g11\}$. In this case, although $g11$ parent g1 has *sig* value less than 1, it has gotten into the thresholded set $\mathcal{G}'$, and, therefore, $cov(g1) = cov(g11)$. Only a part of the testgroup hierarchy is depicted: in the entire graph, the testgroup $g11$ has nine analogously attached predecessors.

Not always the result of the proposed analysis is a clearly located testgroup set. Figures 5 and 6 illustrate situations where a clearly located testgroup set cannot be found, as the result of the analysis does not satisfy one of the considered conditions.

In the example in Fig. 5 the result of the analysis is the set $\{g1\}$ and since $leafs(\{g1\}) = \{g121, g122, g111, g112\}$ differs from $groups(D) = \{g121, g111\}$, this is a violation of condition (1). Despite the fact that the result formally is not clearly located, the deterioration object g1 is still useful as the least common testgroup covering all leaf-testgroups pointing to problems.

In the example in Fig. 6 the result of the analysis is the set $\{g221, g222\}$ and since the both elements are from the same connected component, this is a violation of condition (2). Also, in this case, the result of testgroup analysis formally is not clearly located. However,

Fig. 7. Testgroup hierarchy illustrating necessity of condition (3).

g221 and g222 together covers all leaf-testgroups pointing to problems and at the same time having no leaf-testgroup without deterioration as a constituent.

The example in Fig. 7 demonstrates that without condition (3) besides the result of the analysis {g2}, also sets {g0} and {g1} satisfy the first two conditions, and the assertion of Proposition 1 is false.

4.3. *Root Cause Analysis of Environments*

4.3.1. *Processing of Attribute Bundle Hierarchy*

As stated above, an environment is characterized by attributes and attribute values where the same value is not used for more than one attribute. Objects of our analysis are attribute bundles, i.e. subsets of environment attribute value sets constituting an attribute bundle hierarchy.

To illustrate such hierarchy we use the following abstract environment attributes and their values: operating systems os1, os2; computer architectures ar1, ar2; browsers br1, br2, br3. Based on these values, we consider the example environment set $\mathcal{E}$ with corresponding attribute value sets: {os1, br1}, {os2, ar2}, {os2, br2}, {os2, br1}, {os1, ar1, br1}, {os1, ar1, br2}, {os1, ar1, br3}, {os1, ar2, br3}, {os2, ar1, br2}, {os2, ar1, br3}, {os2, ar2, br3}.

The purpose of analysis of environments is to point out those attribute bundles that are common to the most significant deterioration. The pointed attribute bundle can be a set of an existing environment attribute values, or it can be a subset of attribute values of some defined environments meaning that we are referring to a generalized environment which is not directly accessible for testing. For example, for attribute value sets {os1, ar1, br2}, {os2, ar1, br2}, {os2, ar1, br3} the attribute bundle {ar1} generalizes three attribute bundles into a single, more general attribute bundle. So as an analysis result either attribute value sets or some of their generalizations, i.e. subbundles may be reported.

The main structure of our root cause analysis of environments is attribute bundle hierarchy $\mathcal{A}$. The example introduced above are brought in Table 1. The bundles are grouped by attribute count and are represented by ordered tuples where asterisks denote absent attributes.

A hierarchy corresponding to Table 1 is shown in Fig. 8. Asterisks in nodes allow to follow relations between the attribute bundles easily. Namely, substituting an asterisk by a value of the corresponding attribute we directly get the respective child of this bundle in the hierarchy. Nodes corresponding to given environments are depicted as rectangles: ordinary if deterioration is observed and rounded if there is no deterioration. All other nodes are depicted as ovals. Nodes are supplemented with some bundle parameters, and some of them are graphically highlighted, that is discussed further.

Table 1
Example attribute bundles.

(os1 * *)	(os1 ar1 *) (os2 ar1 *)	(os1 ar1 br1)
(os2 * *)	(os1 ar2 *) (os2 ar2 *)	(os1 ar1 br2)
(* ar1 *)	(os1 * br1) (os1 * br2)	(os1 ar1 br3)
(* ar2 *)	(os2 * br1) (* ar1 br1)	(os1 ar2 br3)
(* * br1)	(os2 * br2) (* ar1 br2)	(os2 ar1 br2)
(* * br2)	(os1 * br3) (os2 * br3)	(os2 ar1 br3)
(* * br3)	(* ar1 br3) (* ar2 br3)	(os2 ar2 br3)

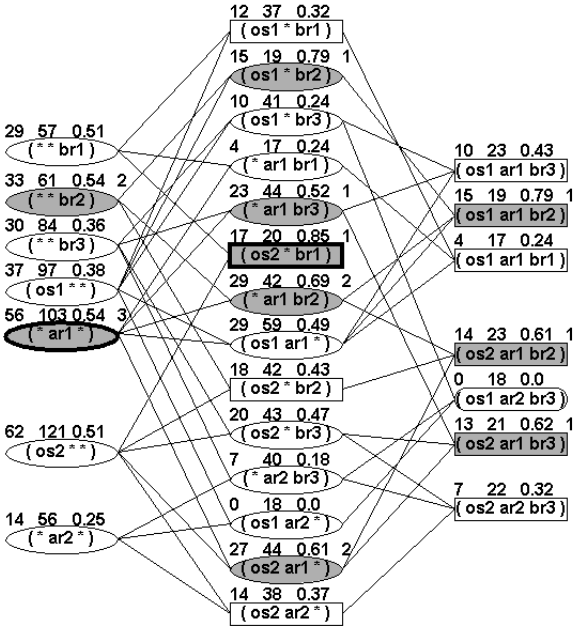

Fig. 8. Example attribute bundle hierarchy.

Values of *det*, *com*, and *sig* are added from the left above the corresponding nodes in named order. Nodes of $\mathcal{A}'$ bundles are supplemented with the fourth parameter: the covering number *cov*. These nodes are coloured gray.

The bar diagram in Fig. 9 illustrates all 28 values of *sig* of the example in Fig. 8. The threshold value found by using Algorithm 2 is $L[19] = 0.52$.

The final stage of our processing procedure is two-stage filtering refining of the set $\mathcal{A}'$. After applying the first refining filter the bundle {br2} is excluded from $\mathcal{A}'$ because the bundle {br2} and its successor {ar1, br2} belong to the initial $\mathcal{A}'$ and $cov(\{\text{br2}\}) = cov(\{\text{ar1, br2}\}) = 2$. Also bundles {os1, br2} and {ar1, br3} having child nodes with equal *cov* value are excluded from $\mathcal{A}'$.

Further, after applying the source refining filter also bundles {ar1, br2}, {os2, ar1}, {os1, ar1, br2}, {os2, ar1, br2} and {os2, ar1, br3} as successors of {ar1} are excluded from $\mathcal{A}'$. Therefore, in the refined $\mathcal{A}'$ there remain only attribute bundles {ar1} and {os2, br1}. This is the final result of our attribute bundle analysis and in Fig. 8 the corresponding nodes are highlighted by bold frames.

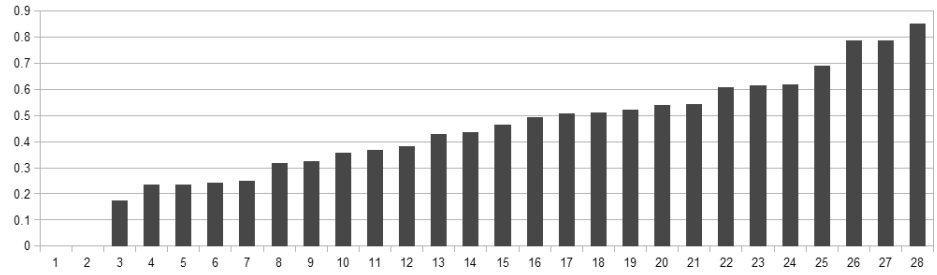

Fig. 9. Bar diagram of *sig* values for the example from Fig. 8.

4.3.2. *Justification of the Proposed Procedure*

The further needs some additional designations:

$env(R)$ – the set of environments of a testrun set R;

$avs(E)$ – the set of attribute value sets of environment set E;

$V = avs(env(D))$ – attribute value sets of environments of a deterioration set D.

The following conditions are based on ones stated at Section 4.1.5.

A set $B \subseteq \mathcal{A}$ is a set of *clearly located attribute bundles* if

(1) all active build testruns r_1 together with coupled testruns r_0 have the property $isDeterioration(r_0.res, r_1.res) \leftrightarrow avs(r_1.env)$ is reachable from B. Note that an equivalent form of the right side is $\exists b \subseteq avs(r_1.env)(b \in B)$.
(2) each element of B belongs to a distinct connected component of $\mathcal{A}$.
(3) for all environment attribute value sets having common subbundle $b \in B$, b is the intersection of these sets.

Proposition 2. *If for a deterioration set D there exists an attribute bundle set B satisfying conditions* (1)*,* (2) *and* (3)*, then the result of the attribute bundle analysis procedure is exactly B.*

Proof. The fragment of Algorithm 1 calculating *det* and *com* values for attribute bundles is:

$r_0 =$ a testrun of $\mathcal{R}_0$, built on pair (l, e)
$r_1 =$ a testrun of $\mathcal{R}_1$, built on pair (l, e)
if r_0 *exists and* r_1 *exists* **then**
  foreach $a \in predecessors(avs(e))$ **do** increase $com(a)$ by 1
  if $avs(e)$ *is reachable from* B **then**
    foreach $a \in predecessors(avs(e))$ **do** increase $det(a)$ by 1
  end if
end if

Clearly, for all $a \in \mathcal{A}$ is $0 \leqslant det(a) \leqslant com(a)$.

If $a \subseteq avs(e) \subseteq V$, i.e. from a some $b \in V$ is reachable, then $det(a) > 0$ and vice versa. Moreover, if additionally the bundle a is reachable from some $b \in B$, i.e. $\exists b \in B(b \subseteq a)$,

then $det(a) = com(a)$, because for $b \in B$ each bundle a with $b \subseteq a$ is a subbundle of some element of V and for such bundles *com* and *det* values during calculations grow simultaneously.

Thus, for significance values *sig* of an attribute bundles of $\mathcal{A}$ we have

$$sig(a) = \begin{cases} 1, & \text{if } a \text{ is reachable from } B, \\ 0 < \cdots \leqslant 1, & \text{if some element of } V \text{ is reachable from } a, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

Now, by the thresholding procedure based on Algorithm 2, all bundles with *sig* value 1 are included into the set $\mathcal{A}'$ and there are no bundles with *sig* value 0 in this set.

Due to condition (2), each element of B belongs to a different connected component of $\mathcal{A}$. Thus, to complete the proof, it is enough to examine a separate $\mathcal{A}$ component comprising some bundle from B. Denote this component by C and consider the set $B_{(1,2)} = \{b \in C \mid \text{all attribute value sets from } C \cap V \text{ are reachable from } b, \text{ i.e. } b \text{ is a subset of all attribute value sets from } C \cap V\}$ every element of which satisfies conditions (1) and (2).

The set $B_{(1,2)}$ contains B element having *sig* value 1 and so belonging to $C \cap \mathcal{A}'$. Therefore, $B_{(1,2)}$ is non-empty. Let's denote by $b_0 \in B_{(1,2)}$ intersection of all attribute value sets from $C \cap V$. All other $B_{(1,2)}$ elements are subsets of b_0, hence b_0 is reachable from them.

We express $cov(a)$ for a bundle $a \in C$ as $|\{s \in C \cap V \mid a \subseteq s\}|$. So for all $B_{(1,2)}$ bundles *cov* value is $cov(b_0)$ which is the largest possible *cov* value in C. Since by $B_{(1,2)}$ definition, there are no C elements with the same *cov* value outside $B_{(1,2)}$, so $B_{(1,2)}$ constitutes an equality class with b_0 as unique sink in it. Thus the result of the sink refining filter contains b_0 as the only representative from $B_{(1,2)}$.

For $a \in C \cap \mathcal{A}'$ and all $\bar{b} \subseteq b_0$ holds $cov(\bar{b} \cup a) = cov(b_0 \cup a)$. Every equality class of the function *cov* together with the bundle a contains also the bundle $b_0 \cup a$ reachable from b_0. So in C every sink of every equality class of the function *cov* is reachable from b_0. Hence all attribute bundles constituting the result of the sink refining filter are reachable from b_0 which is kept as the unique result of the second refining filter in C.

By construction, only b_0 also satisfies condition (3) and is the only element in $C \cap B$. □

Note that condition (3) was not used in the reasoning as a requirement. However, this condition is essential because under just two first conditions also b_0 predecessors would be valid candidates for B, and, therefore, the assertion of Proposition 2 would be false.

We end the chapter with examples demonstrating the meaningfulness of Proposition 2 for the both cases: when all conditions are completely satisfied, and when separate conditions are violated.

Figure 10 demonstrates the example when all conditions of Proposition 2 are satisfied and $B = \{\{\text{os1}\}, \{\text{br1}\}\}$ is a clearly located attribute bundle set.

The example in Fig. 11 illustrates the case when also all conditions of Proposition 2 are satisfied and $B = \{\{\text{os1}, \text{br1}\}\}$. In this case, although $\{\text{os1}, \text{br1}\}$ parents $\{\text{os1}\}$ and $\{\text{br1}\}$ have *sig* value less than 1, they have gotten into the thresholded set $\mathcal{A}'$, and, therefore, $cov(\{\text{os1}\}) = cov(\{\text{br1}\}) = cov(\{\text{os1}, \text{br1}\})$.

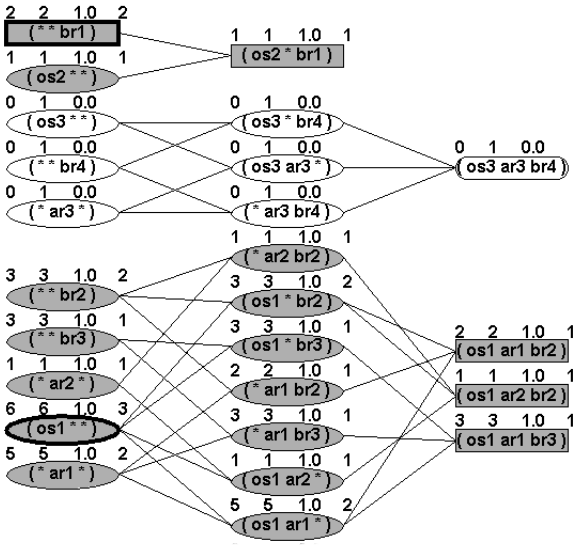

Fig. 10. Attribute bundle hierarchy with all conditions satisfied, $B = \{\{\text{os1}\}, \{\text{br1}\}\}$.

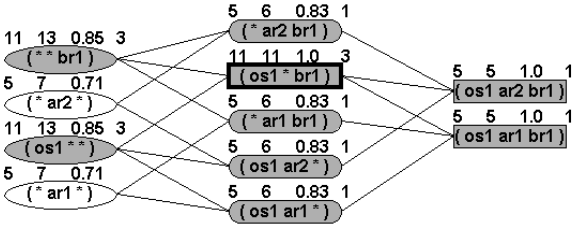

Fig. 11. Attribute bundle hierarchy with all conditions satisfied, threshold 0.83, $B = \{\{\text{os1}, \text{br1}\}\}$.

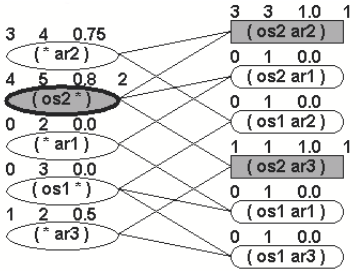

Fig. 12. Attribute bundle hierarchy where result does not satisfy condition (1).

In the example in Fig. 12 the only resulting deterioration attribute bundle is {os2} and, in this case, the result is not clearly located because condition (1) of Proposition 2 is violated since the environment attribute value set {os2, ar1}, which is not deterioration attribute bundle, is reachable from {os2}. However, all attribute value sets from V are covered by {os2} and so pointing to it is reasonable.

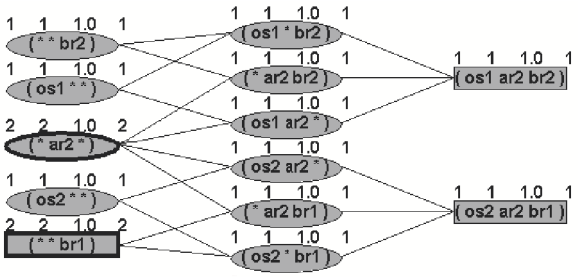

Fig. 13. Attribute bundle hierarchy where result does not satisfy condition (2).

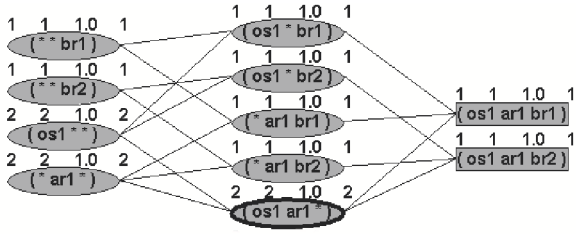

Fig. 14. Attribute bundle hierarchy illustrating necessity of condition (3).

In the example in Fig. 13 the resulting deterioration attribute bundle set contains two bundles {ar2} and {br1} violating condition (2) of Proposition 2. Note that {br1} itself is an environment attribute value set. Despite the fact that the result formally is not clearly located, found bundles are useful because they together cover all deterioration environment attribute value sets.

In the example in Fig. 14, sets {os1} and {ar1} satisfy conditions (1) and (2), whilst the result of the analysis is {os1, ar1}, so the assertion of Proposition 2 is false without condition (3).

5. Practical Results

The proposed algorithm was tested both on generated and real data sets. In the first case, all data elements must be created from scratch, and we are free to choose their structure and size. Some generated simple examples illustrating various aspects of our analysis are shown in the figures above. As small examples are not sufficient to judge about such serious issue as the performance of our algorithm, we estimated performance theoretically and developed series of timing experiments on an ordinary computer (2 core 2.10 GHz Intel® Core™ i3-2310M CPU, 4 GB RAM).

For performance estimation we use hierarchies of homogeneous structure. At $n_{\mathrm{L}} = |\mathcal{L}|$ assuming that there are n_{A} attributes with m_{A} values each, for one pair of builds rough estimation of analysis time is $\mathrm{O}(n_{\mathrm{L}}(m_{\mathrm{A}})^{n_{\mathrm{A}}}(\log(n_{\mathrm{L}}) + 2^{n_{\mathrm{A}}}))$. Impact of the parameters n_{L} and n_{A} for $m_{\mathrm{A}} = 4$ can be observed in Table 2 where cells without data denote that the result of the analysis could not be obtained within 1 minute.

Table 2
Time of analysis in seconds and the number of randomly generated testruns depending on n_L leaf-testgroups and n_A attributes ($m_A = 4$).

n_L	n_A				
	3	4	5	6	7
50	0.2 (6063)	0.7 (24339)	1.3 (97304)	4.7 (389412)	49.6 (1556676)
100	0.3 (12156)	0.7 (48461)	2.0 (194536)	13.4 (778629)	–
150	0.3 (18229)	0.7 (72956)	2.5 (291953)	15.1 (1167687)	–
200	0.4 (24336)	0.8 (97296)	3.1 (389404)	24.3 (1556590)	–
250	0.4 (30403)	1.1 (121626)	3.9 (486705)	–	–
300	0.5 (36483)	1.2 (145950)	4.6 (583939)	–	–

Table 3
Quantitative characteristics of the largest used real dataset.

Description	Minimum	Maximum	Average
Number of testruns containing a build	1	971	111.0
Number of testruns containing a leaf-testgroup	3	83	43.6
Number of testruns containing an environment	4	608	351.5
Number of testruns containing a pair of a leaf-testgroup and an environment	1	35	5.8
Number of coupled testrun pairs for a pair of builds	1	853	26.0

We see that theoretical analysis time grows rapidly with the growth of the number of attributes. Fortunately, the number of attributes in our real data is not too large.

The origin of real data is a testing process of a large scale application maintenance and updating. These data were collected during a longer period and were not adjusted especially for needs of our analysis. So, besides clearly technical work like obtaining data from an original database, adapting of environment attributes should be done.

In the used real data sets environment attributes were obtained from the given environment descriptions and four attributes: *operating system*, *operating system version*, *architecture*, and *browser* with the corresponding number of values 6, 23, 2 and 3 that were chosen. Such choice is not strictly predefined, as some attributes may be merged or split into smaller ones. However, it is unreasonable to define attributes having just one value, because the presence of such attributes does not influence results of our analysis, just increasing a volume of data to be processed. Therefore, the number of attributes for real environments is limited, and our analysis does not suffer from an exponential growth of the number of attribute bundles built on attribute value sets. Moreover, some combinations of attribute values may be incompatible or senseless, i.e. if such combination is not presented in any real environment.

In the largest real dataset used for our algorithm testing, the number of real environment attribute value sets (18) was significantly smaller than the number of possible attribute value sets ($6 \times 23 \times 2 \times 3 = 828$). In total, there were 6327 testruns, 57 builds, 163 testgroups, 145 leaf-testgroups, 18 environments. The depth of the testgroup forest was 3. Additional statistical characteristics of the used dataset are given in Table 3.

Values of characteristic *det* for the observed dataset were quite low, which is not surprising because in the real software development process at mature phase we cannot expect

a dramatic decrease of quality. As a result, also *sig* values are low (almost all values are 0 with very few less than 0.1) and, although we cannot observe the computational power of our algorithm in full strength, obtained results are cogent.

Also, the performance was acceptable and for the dataset discussed the time of analysis for a pair of builds did not exceed 0.3 seconds, and regarding Table 2, characteristics of our dataset lay in the top left corner giving hope that we will also be able to process other real datasets. Moreover, the real attribute value distribution is far from homogeneous, so that the total number of possible combinations would be lowered.

6. Conclusions

We have presented a new root cause analysis algorithm for discovering the most likely causes of differences found in testing results of two software builds. The proposed algorithm works with hierarchies of testgroups and attributes of testing environments. These hierarchies allow generalizing found problems by tested features. Relevant assigning of attributes produces compact analysis results and allows decreasing the duration of software development cycle.

Obtained analysis results reach the initial goal: direct one's attention to the most problematic points without a necessity to search inside a huge amount of data for any single test case with an unexpected outcome. With the appropriate visualization (Opmanis *et al*., 2016), these results give insight into quality dynamics of a sequence of builds and ensures finding main deterioration places by a few clicks.

We emphasize that in the general case when the failure of a particular testrun is caused by more than one object from the same hierarchy and a unique reason for a failure cannot be discovered, our analysis algorithms work as heuristics. For clearly separated problem causes, we have proven that our algorithm gives an exact solution.

In a case of a large number of uniformly scattered failures our algorithm reports the most general hierarchy objects, therefore giving no focused direction of searching for a cause of failures. Real causes must be investigated separately for smaller failure groups by our visualization tool (Opmanis *et al*., 2016).

Our approach works with a limited number of attributes and their values. However, practical application until now was not even close to these limits. Therefore, a part of future work could be improving our method to allow using a larger number of attributes and values, or ascertaining that the number of attributes of real applications will never be too large. Another way to improve effectiveness is to calculate characteristics of composite testgroups in advance before analysis, especially in cases when a structure of testgroup hierarchy evolves over time. The interesting question is the possibility to obtain an exact result by the actual algorithm also if condition (1) of the Propositions are substituted by weaker ones.

The proposed analysis algorithm may be used during software development phase also even if we have testing results only for the actual build: we can create mock reference testing results in advance and compare our actual testing results to them. The simplest

example of mock reference results is testing results where execution status for all tests is "successful". These results embody highest expected level of software where there are no failed tests. Comparing actual results to such mock reference results is useful at the final stages of the development indicating which parts are not completed.

Mock testing results may also describe real testing milestones, i.e. test results according to a project plan, not asking to be perfect as in the example above. Then comparing real testing results to the mock allows various groups of users (testers, developers, managers) to see if the development is happening according to the plan, and if not, then which areas are falling back the most.

Acknowledgements. This work was supported by European Regional Development Fund project 2014/0013/2DP/2.1.1.1.0/14/APIA/VIAA/034.

The authors thank Juris Strods for presenting problem of a root cause of deterioration in large scale application testing results and Kārlis Podnieks for valuable comments.

References

Biswas, S., Mall, R., Satpathy, M., Sukumaran, S. (2011). Regression test selection techniques: a survey. *Informatica*, 35, 289–321.

Bohner, S., Arnold, R. (1996). *Software Change Impact Analysis*. Wiley, IEEE Computer Society Press. ISBN:978-0-8186-7384-9. 392 pp.

Briski, K.A., Chitale, P., Hamilton, V., Pratt, A., Starr, B., Veroulis, J., Villard, B. (2008). Minimizing code defects to improve software quality and lower development costs. ftp://ftp.software.ibm.com/software/rational/info/do-more/RAW14109USEN.pdf.

Chillarege, R. (2013). 5 differences between classical and ODC root cause analysis in software. http://www.chillarege.com/articles/5-differences-odc-rca.html.

Dev, N., Samsher, Kachhwaha, S.S., Attri R. (2014). Development of reliability index for combined cycle power plant using graph theoretic approach. *Ain Shams Engineering Journal*, 5(1), 193–202.

Dustin, E. (2002). *Effective Software Testing: 50 Specific Ways to Improve Your Testing*, 1 ed. Addison-Wesley Professional.

IBM (2013). Orthogonal defect classification v5.2 for software design and code. http://researcher.watson.ibm.com/researcher/files/us-pasanth/ODC-5-2.pdf.

Kataoka, T., Furuto, K., Matsumoto T. (2011). The analyzing method of root causes for software problems. *SEI Technical Review*, 73, 81–85.

Leszak, M., Perry, D.E., Stoll, D. (2002). Classification and evaluation of defects in a project retrospective. *The Journal of Systems and Software*, 61(3), 173–187.

Linders, B. (2014). Success factors for root cause analysis in software development. http://www.benlinders.com/2014/success-factors-for-root-cause-analysis-in-software-development/.

Marvasti, M.A., Poghosyan, A.V., Harutyunyan, A.N., Grigoryan, N.M. (2013). An anomaly event correlation engine: identifying root causes, bottlenecks, and black swans in IT environments. *VMware Technical Journal*, 2(1), 35–45.

Myers, G.J., Sandler, C., Badgett, T. (2012). *The Art of Software Testing*, 3 ed. John Wiley and Sons, Inc.

Ohrimenko, O., Papamanthou, C., Palazzi, B. (2013). Computer security. In: Tamassia, R. (Ed.), *Handbook of Graph Drawing and Visualization*. CRC Press, 666 pp.

Opmanis, R., Ķikusts, P., Opmanis, M. (2016). Visualization of large-scale application testing results. *Baltic Journal of Modern Computing*, 4(1), 34–50.

Rao, R.V. (2010). *Decision Making in the Manufacturing Environment: Using Graph Theory and Fuzzy Multiple Attribute Decision Making Methods*. In: *Springer Series in Advanced Manufacturing*. Springer, ISBN:978-1849966535.

Rooney, J.J., Vanden Heuvel, L.N. (2004). Root cause analysis for beginners. *Quality Progress*, 37(7), 45–53.

Ruberto, J. (2013). Root cause analysis for software problem. http://blog.ruberto.com/2013/04/root-cause-analysis-for-software-problems/.

Rundle, P.K. (2003). The need for root cause failure analysis. In: *International IEEE Conference Hong Kong & Shenzhen*. http://www.rundlelawcorp.com/files/Abstract2_IEEE_Root_Cause_Analysis.pdf.

Steinder, M., Sethi, A.S. (2004). A survey of fault localization techniques in computer networks, *Science of Computer Programming*, 53(2), 165–194.

Tague, N.R. (2005). *Quality Toolbox*, 2 ed. ASQ Quality Press. ISBN:978-0873896399, 584 pp.

Tom Sawyer Software (2005). *Tom Sawyer Perspectives*. https://www.tomsawyer.com/products/perspectives/.

Wilson, B. (2014). *The Rootisserie – RCA Resources*. http://www.bill-wilson.net/root-cause-analysis/rca-resources.

Zeller, A. (2002). Isolating cause-effect chains from computer programs. In: *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM, pp. 1–10.

R. Opmanis is a researcher at Institute of Mathematics and Computer Science of University of Latvia. His research interests are data vizualization, image processing and testing of large scale applications.

P. Ķikusts, Dr.sc.math., is a senior researcher at Institute of Mathematics and Computer Science of University of Latvia. His main research interests are graph theory and graph drawing, image analysis and machine vision.

M. Opmanis is a researcher at Institute of Mathematics and Computer Science of University of Latvia. His research interests are building and using multilevel data repositories for ontological and meta-modelling, organizing and establishing standards for programming contests.

Didelio masto programų testavimo rezultatų esminių priežasčių analizė

Rūdolfs OPMANIS, Paulis ĶIKUSTS, Mārtiņš OPMANIS

Pristatome naują esminių priežasčių analizės algoritmą atrasti labiausiai tikėtinoms skirtumų priežastims, rastoms dviejų versijų tos pačios programinės įrangos testavimo rezultatuose. Glaustai vartotojui pateikti probleminiai testavimo ir aplinkos atributų hierarchijos taškai savo ruožtu leidžia sutaupyti laiko apdoroti testavimo rezultatams. Įrodyta, kad aiškiai atskirtoms problemų priežastims šis algoritmas pateikia tikslų sprendimą. Aprašyto metodo praktinis taikymas yra aptartas.