Root cause analysis of large scale application testing results *

Rūdolfs Opmanis, Paulis Ķikusts, and Mārtiņš Opmanis

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

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

Keywords: Root cause analysis, regression testing, hierarchy graphs.

1 Introduction

Testing results are important indicator of the overall application state regardless if it is in active development or maintenance. During the active development phase testing results can help to understand if the current status is consistent with the plan and manage required resources accordingly. However during maintenance phase testing results are necessary to verify if the application after internal or external changes still performs as expected. Software maintenance has been identified as the most costly and difficult phase in the software life cycle [1,2,3] so ability to measure quality of the application is critical for managing cost and time.

One of basic principles of testing is "Any testing process should include a thorough inspection of the results of each test" [4]. If application is small then all testing results can be processed easily, but for large applications even test result processing has to be automated, because large applications usually have huge data sets of automatic and manual tests that needs to be run with different frequencies and on various environments. Also not all failed test results are equally important to the user, because tests that have been failing for the past month and are known are not as important as a set of failed tests that started failing after the most recent changes in application. For large data sets finding

* Supported by ERDF project 2014/0013/2DP/2.1.1.1.0/14/APIA/VIAA/034

and filtering the most interesting failed tests manually is very time consuming so an automatic algorithm is required to perform necessary test result analyis.

Root cause analysis [5] is used to compare two testing results and to suggest the most likely causes of the reported differences and may be used as part of regression testing. Results produced by root cause analysis of the testing results helps to present them to the user in compact way which in turn allows to save time on manual processing of test results.

In this paper we present a new root cause analysis algorithm for test result processing. The algorithm is used in root cause identification phase [6] and works with test hiearchies, attributed testing environments and test results that are acquired by executing test cases on testing environments. Although test organization in hierarchies is not mandatory, such organization will allow the proposed algorithm to generalize problems by features. Similary, having multiple attributes for testing environments are not necessary, but providing them to the algorithm will let it to produce more compact results. Unlike [7] we are not analyzing source code, but use only testing results which gives us the ability to apply this knowleadge to any software development project independently of programming language or software development mehodology as long as testing is done reasonably frequently.

There are other approaches which help to reduce time and cost of application maintenance like software impact analysis [8] that analyzes software to predict affected items in order to estimate required effort, cost and time to implement new feature or do some adjustment in existing software product, but this approach requires very thorough planning and continuous dependency management which might not be available for big projects or projects that are handed over between maintainers.

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

2 Preliminaries

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

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

A composite testgroup consists of one or more lower level testgroups and during testing is considered as a single object. From the graph perspective testgroups as nodes are organized in tree-like hierarchical structure and all together constitutes a directed forest. Each simple testgroup is leaf in this forest and each composite testgroup is in 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 the XML source, in another there may be tests checking data import from database and these testgroups may be included in higher level testgroup checking data import in general. Such approach allows directly point to functional parts of tested software when 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 attributes may be browser, for mobile devices – application, etc. We assume that the same attribute value is not used for more than one attribute, therefore any attribute value is enough to identify the attribute. Each environment is identified by *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 nonempty subset of environments attribute set is called *attribute bundle* or *bundle* or *subbundle* for convenience.

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

Hierarchy of testgroups and hierarchy of attribute bundles in this paper is referenced as *hierarchies* and their elements as *hierarchy objects*.

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

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

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

Our purpose is to introduce measure of significance of deterioration 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 the first and therefore we think unreasonable to report large number of less significant deterioration objects. Instead we consolidate information about them into smaller number of *resulting deterioration objects* of higher hierarchy levels. As result of proposed analysis two collections of resulting deterioration objects – *resulting deterioration testgroups* and *resulting deterioration attribute bundles* are provided.

3 Root cause analysis

Testing process is described 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 $\mathcal{L}$ set of all leaf-testgroups; by $avs(e)$ 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 subset of $\mathcal{L} \times \mathcal{E}$, and $\mathcal{R}_1 \subseteq \mathcal{R}$ is a set of testruns using b_1 and 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 the 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 specialized boolean function $isDeterioration(r_0.res, r_1.res)$ returning *true* when r_1 result is worse than the result of r_0. The basis of deterioration analysis is a *deterioration set* D consisting of the testruns $r_1 \in \mathcal{R}_1$, that with respect to the corresponding element $r_0 \in \mathcal{R}_0$ have $isDeterioration(r_0.res, r_1.res) = true$.

3.1 Algorithmic principles of deterioration analysis

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

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

Each step is described below in details.

3.1.1 Calculating the 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.

For each leaf-testgroup $l \in \mathcal{L}$ value of $det(l)$ indicates how many times the leaf-testgroup l occurs as constituent of the testruns of D and $com(l)$ indicates 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 testgroup $g \in G$ is not a leaf-testgroup, than the value $det(g)$ is a recursive sum of the values det of all g children $g_1, g_2, ... : det(g) = det(g_1) + det(g_2) + ...$, and, analogously, $com(g) = com(g_1) + com(g_2) + ...$

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

Based on parent-child relation let's by $predecessors(n)$ denote set of DAG nodes being predecessors of node n together with n itself and by $successors(n)$ denote 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 node set N if n is reachable from some element of N.

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

```
Input: the testgroup set $\mathcal{G}$, the environment set $\mathcal{E}$, the reference build $b_0$
       corresponding testrun set $\mathcal{R}_0$, the active build $b_1$ corresponding testrun
       set $R_1$
Output: the 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 the pair $(l, e)$
            $r_1$ := a testrun of $\mathcal{R}_1$ built on the 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
```

We rely on the consideration that the sources of deterioration should manifesting via hierarchy objects that appear most frequently in the testrun set D. This consideration is fundamental for 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 the testruns it is involved with and the deterioration 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, beside this consideration of a statistical nature, other considerations of object significance may be taken in account. Say, 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 scope of this paper.

3.1.2 Thresholding the significance values.

Now when deterioration significance values of objects of both hierarchies are found, we use them to locate the most problematic points of the software tested.

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

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

```
Input: A non-empty list $L$ of non-decreasing nonnegative real numbers
Result: The 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 the greatest index $k$ satisfying
        $L[1] + L[2] + ... + L[k–1] < L[k] + L[k+1] + ... + L[k_{max}]$
    end if
end
```

Fig. 1 illustrates 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 the index 11, so the corresponding threshold value is $L[11] = 0.573$ and the result of half-sum thresholding procedure are six objects ensuring broader view of the analyzed hierarchy than using just maximum.

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

Another approach was histogram-inspired analysis – find longest subsequence of consecutive close enough values and split right after it (or right before if

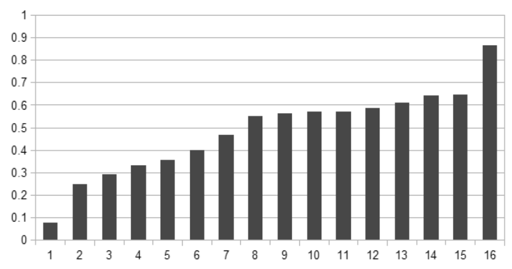

Fig. 1. Bar diagram of *sig* values for example from Fig. 2.

subsequence includes the greatest value). In the given example such value is $L[12]$, however, also this approach seemed not stable enough and to be relatively complicated.

Executing selected thresholding procedure on a hierarchy, we get a *thresholded set* of all hierarchy objects having *sig* value at least threshold value: $\mathcal{G}'$ from testgroup hierarchy $\mathcal{G}$ and $\mathcal{A}'$ from attribute bundle hierarchy $\mathcal{A}$.

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

3.1.3 Calculating the presentence of hierarchy objects within the deterioration set. Refining procedure consists of two consecutive graph filters. The first one takes into account an important additional parameter *covering number*: for hierarchy object x covering number $cov(x)$ denotes number of deterioration objects without successors reachable from x. For testgroups $g \in \mathcal{G}'$ function value $cov(g)$ is number of g successors that belongs to the set $\mathcal{G}'$ and are leaf-testgroups. For attribute bundles $a \in \mathcal{A}'$ function value $cov(a)$ denotes number of $\mathcal{A}'$ bundles that are attribute sets of environments from the deterioration set D containing a as subbundle.

3.1.4 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 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 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 object x_1 of a hierarchy is a predecessor of 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 deterioration object of the tested software because in this case unnecessary generalization of deterioration object takes place. In other words, object x_2 is kept 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 belongs 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 belongs 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 object x_2 from the hierarchy is successor of object x_1 from the same hierarchy, this relation alone is not a reason to report x_2 as deterioration point of the tested software because in this case unnecessary specialization of deterioration point takes place. In other words, x_1 is kept because it consolidates deterioration information about successor objects from the same hierarchy. For attribute bundles it may be also explained as if intersection of bundles is non-empty and is present in the same attribute bundle set, then intersecting bundles are excluded and just intersection is retained.

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

3.1.5 Justification of the proposed approach.

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

Clearly located sources are objects from some deterioration object set S that is characterized by the following:

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

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

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 individual reason of failure can't be discovered, our analysis algorithms work as heuristics.

3.2 Root cause analysis of testgroups

3.2.1 Processing of testgroup hierarchy.

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

In Fig. 2 example hierarchy of testgroups is shown. Testgroup names are labels placed inside nodes. The values *det*, *com*, and *sig* are added from the left above the corresponding nodes in the named order. Nodes of $\mathcal{G}'$ testgroups are supplemented with the forth parameter the covering number *cov* and are colored gray.

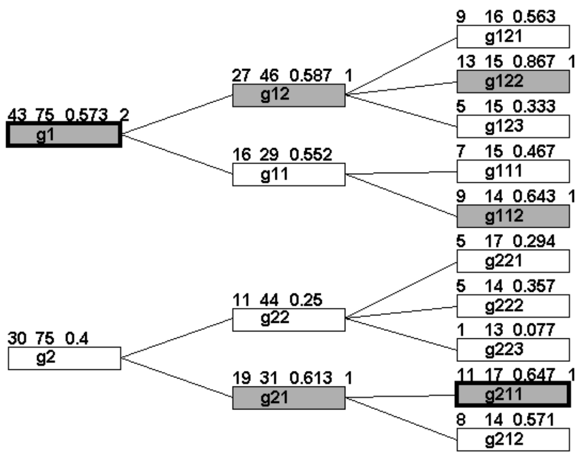

Fig. 2. Example hierarchy of testgroups.

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

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

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

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

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

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

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

(1) all active build testruns r_1 together with coupled testruns r_0 have the property $isDeterioration(r_0.res, r_1.res) \leftrightarrow r_1.grp$ is reachable from G. Note that equivalent form of the right side is *leafs*$(G) =$ *groups*(D);

(2) each element of G belongs to 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 deterioration set D exists a testgroup set G satisfying conditions (1), (2) and (3), then result of testgroup analysis procedure is exactly G.

Proof.

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

```
$r_0$ = a testrun of $\mathcal{R}_0$, built on the pair $(l, e)$
$r_1$ = a testrun of $\mathcal{R}_1$ built on the 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 $0 \le det(l) \le com(l)$. Moreover, for l not reachable from G $det(l) = 0$ because increasing of $det(l)$ is skipped. If l is reachable from G then values *com* and *det* during calculations grow simultaneously, hence $det(l) = com(l)$.

Thus for the significance values *sig* of the $l \in$ *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 values of the function *sig* of composite testgroups.

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

Hence

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

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 the case (a), when the testgroup g is reachable from some element of G, the 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, ..., and from (2) immediately follows $sig(g) = 1$. Recursively backtracking in direction of g parents till the g ancestor from G, we see that all testgroups on this predecessor path also have sig values equal to 1.

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

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

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

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 the 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 $G_{12} = \{g \in C \mid leafs(g) = leafs(C \cap \mathcal{G}')\}$ every element of which satisfies conditions (1) and (2). Set G_{12} is non-empty because it contains G element having sig value 1 and so belonging to $\mathcal{G}'$. Note that cov value of all G_{12} testgroups is the same and maximum possible in C.

Let's examine particular element $g \in G_{12}$.

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 $\bar{g} \in G_{12}$.

For number of g children three mutually exclusive cases must be distinguished:

- If g has exactly one child $\bar{g}$ then $cov(\bar{g}) = cov(g)$ and by definition $\bar{g} \in G_{12}$. During processing the first refining filter g will be excluded from $\mathcal{G}'$.
- If g has at least two children, then for each child $\bar{g}$ is a set $leafs(\bar{g})$ is proper subset of $leafs(g)$ hence $cov(\bar{g}) < cov(g)$ and g is sink of G_{12}.
- If g has no child, i.e. it is leaf-testgroup, then $cov(g) = 1$ and g is sink of G_{12}.

Thus G_{12} testgroups in C constitutes a path having unique sink g_0 which is also result of applying sink refining filter.

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

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

Note that condition (3) was not used in reasoning as 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 Proposition 1 assertion would be false.

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

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

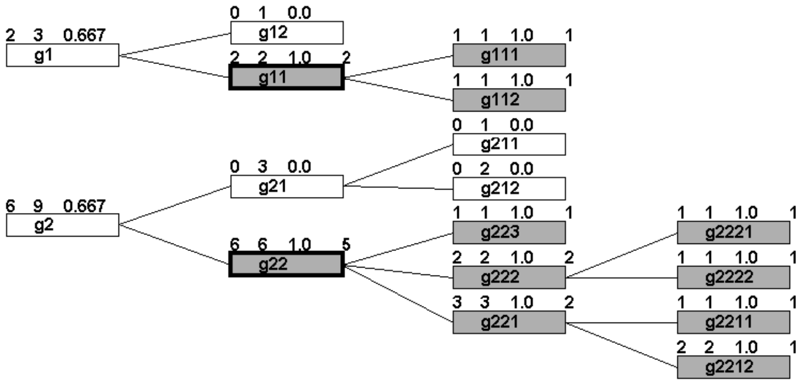

Fig. 3. Testgroup hierarchy with all conditions satisfied, $G = \{$g11, g22$\}$.

Example at Fig. 4 illustrates the case when also all conditions of Proposition 1 are satisfied and $G = \{$g11$\}$. In this case the g11 parent g1 though has the *sig* value less than 1, nevertheless has got 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 result of proposed analysis is clearly located testgroup set. Fig. 5 and Fig. 6 illustrates situations where clearly located testgroup set cannot be found, as analysis result does not satisfy one of considered conditions.

In example at Fig. 5 result of analysis is a set {g1} and since *leafs*({g1}) = {g121, g122, g111, g112} differs from *groups*(D) = {g121, g111}, this is a violation of condition (1). Despite to fact that found result formally is not clearly

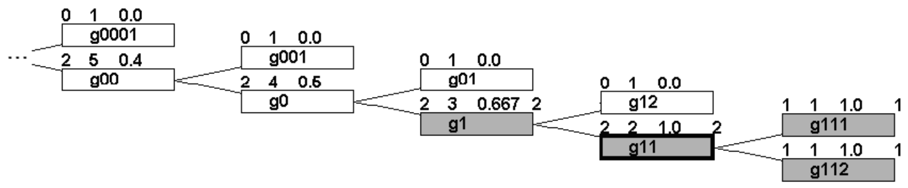

Fig. 4. Testgroup hierarchy with all conditions satisfied, threshold 0.667, G = {g11}.

located, deterioration object g1 still is useful as least common testgroup covering all leaf-testgroups pointing to problems.

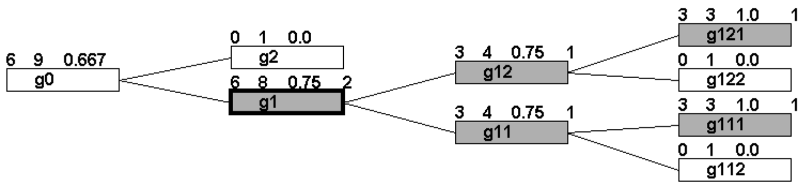

Fig. 5. Testgroup hierarchy where result does not satisfy condition (1).

In example at Fig. 6 result of analysis is a set {g221, g222} and because both elements are from the same connected component, this is a violation of condition (2). Also in this case result of testgroup analysis formally is not clearly located. However, g221 and g222 together covers all leaf-testgroups pointing to problems and at the same time having no leaf-testgroup without deterioration as constituent.

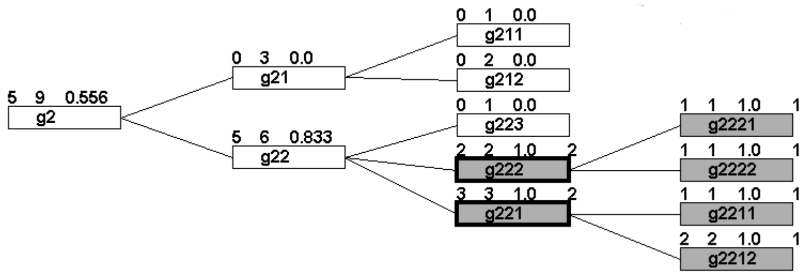

Fig. 6. Testgroup hierarchy where result does not satisfy condition (2).

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

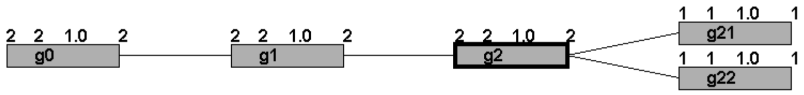

Fig. 7. Testgroup hierarchy illustrating necessity of condition (3).

3.3 Root cause analysis of environments

3.3.1 Processing of attribute bundle hierarchy. As stated above, 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 sets constituting 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 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 in any 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} 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 root cause analysis of environments is attribute bundle hierarchy $\mathcal{A}$. The bundles of our example are brought in Table 1 additionally grouped by attribute count and are represented by ordered tuples where absent attributes are marked by asterisks.

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)

The example hierarchy is illustrated in Fig. 8. Presence of asterisks in nodes allows to easy follow the attribute bundle relations. Namely, substituting an asterisk by the value of the corresponding attribute we directly get the respective child of this bundle in the hierarchy. The nodes corresponding to the 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.

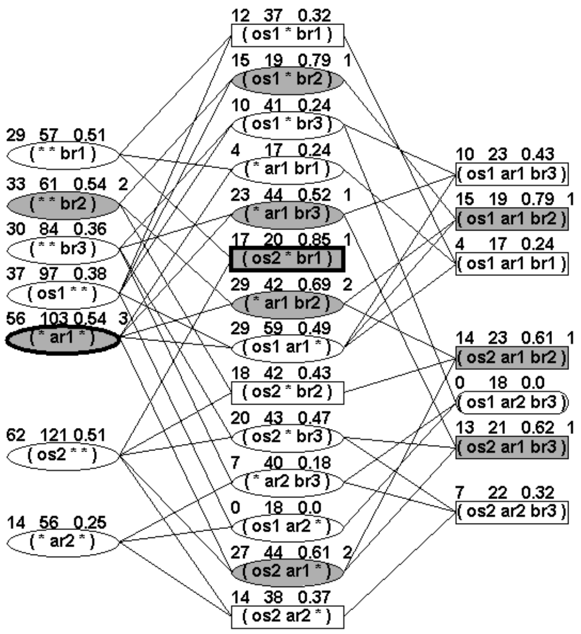

Fig. 8. Example attribute bundle hierarchy.

The values *det*, *com*, and *sig* are added from the left above the corresponding nodes in the named order. Nodes of $\mathcal{A}'$ bundles are supplemented with the forth parameter the covering value cov and are colored gray. The bar diagram at the Fig. 9 illustrates all 28 values of *sig* of example in Fig. 8. Threshold value found by using Algorithm 2 is $L[19] = 0.52$.

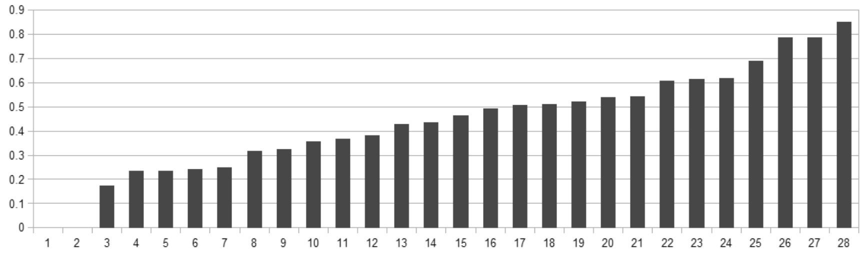

Fig. 9. Bar diagram of *sig* values for example from Fig.8.

The final stage of processing procedure is two-stage filtering refining the set $\mathcal{A}'$. After applying the first refining filter bundle {br2} is excluded from $\mathcal{A}'$ because bundle {br2} and its successor {ar1, br2} belongs 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 a final result of attribute bundle analysis and in the Fig.8 corresponding nodes are highlighted by bold frames.

3.3.2 Justification of proposed procedure.

The further needs some additional designations:

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

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

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

The following conditions are based on ones stated at 4.1.5.

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

(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 equivalent form of the right side is $\exists b \subseteq avs(r_1.env)(b \in B)$.

(2) each element of B belongs to distinct connected component of $\mathcal{A}$.

(3) for all environment attribute sets having common subbundle $b \in B$, b is intersection of these attribute sets.

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

Proof.

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

```
$r_0$ = a testrun of $\mathcal{R}_0$, built on the pair $(l, e)$
$r_1$ = a testrun of $\mathcal{R}_1$ built on the 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 $0 \leq det(a) \leq com(a)$.

If $a \subseteq avs(e) \subseteq V$, i.e. from a is reachable some $b \in V$, 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 subbundle of some element of V and for such bundles the values *com* and *det* during calculations grow simultaneously.

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

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

Now, in accordance with the thresholding procedure based on Algorithm 2, into the set $\mathcal{A}'$ definitely get all bundles with the *sig* value 1, and definitely do not get bundles with the *sig* value 0.

Due to condition (2), each element of B belongs to different connected component of $\mathcal{A}$. Thus, to complete the proof, it is enough to examine separate $\mathcal{A}$ component comprising some bundle from B. Denote this component by C and consider the set $B_{12} = \{b \in C \mid$ all attribute sets from $C \cap V)$ are reachable from b, i.e. b is subset of all attribute sets from $C \cap V\}$ every element of which satisfies conditions (1) and (2).

Set B_{12} is non-empty because it contains B element having *sig* value 1 and so belonging to $C \cap \mathcal{A}'$. Let's denote by $b_0 \in B_{12}$ intersection of all attribute sets from $C \cap V$. All other B_{12} elements are subsets from b_0, hence b_0 is reachable from them.

We express $cov(a)$ for arbitrary bundle $a \in C$ as $|\{s \in C \cap V \mid a \subseteq s\}|$. So for all B_{12} bundles *cov* value is $cov(b_0)$ which is maximum possible *cov* value in C. Since by B_{12} definition, there are no C elements with the same *cov* value outside B_{12}, so B_{12} constitutes equality class with b_0 as unique sink in it. Thus result of the sink refining filter contains b_0 as only representative from B_{12}.

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 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 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 satisfies also condition (3) and is the only element in $C \cap B$. $\square$

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

We end the chapter with some examples demonstrating the meaningfulness of Proposition 2. Both the cases when all conditions of the Proposition 2 are completely satisfied and the cases when individual conditions are violated are presented.

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

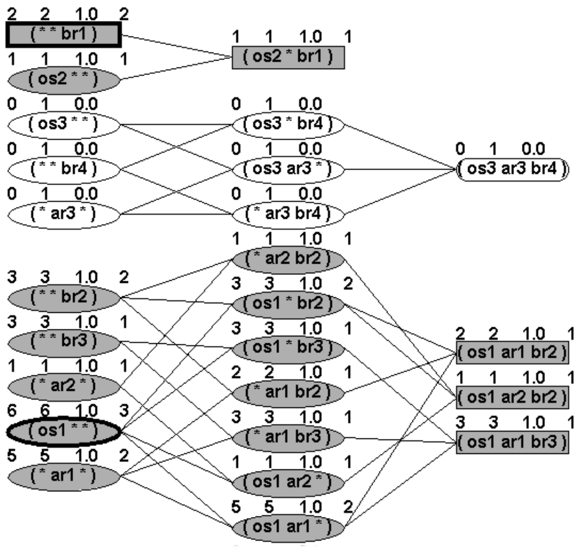

Fig. 10. Attribute bundle hierarchy with all conditions satisfied, $B = \{\{\text{os1}\}, \{\text{br1}\}\}$.

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

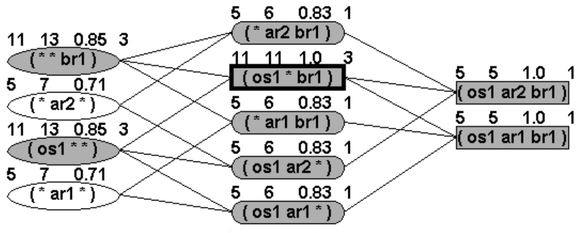

Fig. 11. Attribute bundle hierarchy with all conditions satisfied, threshold 0.83, $B = \{\{\text{os1, br1}\}\}$.

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

In example at Fig. 13 the resulting deterioration attribute bundle set contains two bundles {ar2} and {br1} violating the condition (2) of Proposition 2. Note that {br1} itself is environment attribute set. Despite fact that result formally

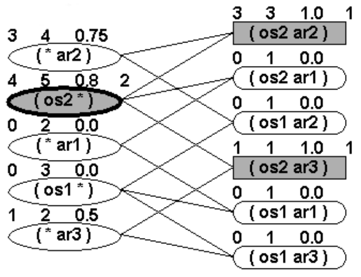

Fig. 12. Attribute bundle hierarchy where result does not satisfy condition (1).

is not clearly located, found bundles are useful because they together covers all deterioration environment attribute sets.

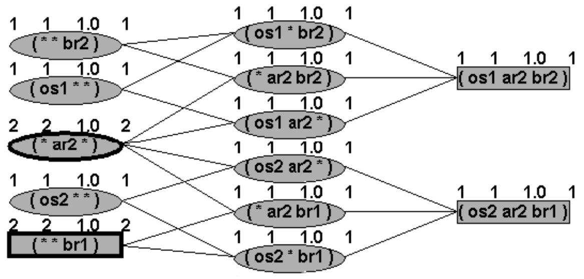

Fig. 13. Attribute bundle hierarchy where result does not satisfy condition (2).

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

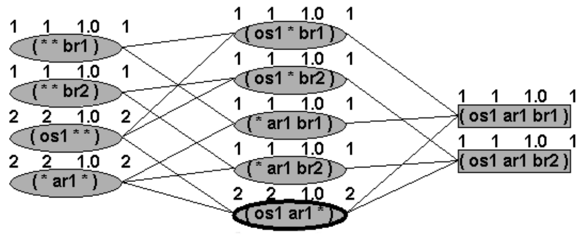

Fig. 14. Attribute bundle hierarchy illustrating necessity of condition (3).

4 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 simple generated 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 performance of our algorithm, we estimated performance theoreticaly and developed series of timing experiments on an ordinary computer (4 core 2.10GHz Intel® Core™ i3-2310M CPU, 4GB RAM).

For performance estimation we use homogenous structure of hierarchies. 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 denotes that result of analysis could not be obtained within 1 minute.

Table 2. Time (in seconds) of analysis and number of randomly generated testruns depending on n_{L} leaf-testgroups and n_{A} attributes ($m_{\mathrm{A}} = 4$).

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)	–	–

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

The origin of real data is testing process of the large scale application maintenance and updating. These data were collected during longer period and were not adjusted especially for our analysis needs. So, besides clearly technical work like obtaining data from the 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 corresponding number of values 6, 23, 2 and 3 were chosen. Such choice is not strictly predefined, as some attributes may be merged together or splitted into smaller ones. However, it is unreasonable to define attributes having just one value, because presence of such attributes does not influence results of our analysis, just increasing volume of data to be processed. Therefore number of attributes for real environments is limited and our analysis does not suffer from exponential 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 data set used for our algorithm testing number of real environment attribute value sets (18) is essentially less than 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. Depth of testgroup forest is 3. Additional statistical characteristics of the used data set are given in Table 3.

Table 3. Quantitative characteristics of the largest used real data set.

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 leaf-testgroup and environment	1	35	5.8
Number of coupled testrun pairs for a pair of builds	1	853	26.0

Values of characteristic *det* for the observed data set were quite low, which is not surprising because in the real software development process at mature phase we cannot expect dramatic decrese of quality. As a result also *sig* values are low (almost all values are 0 with very few less than 0.1) and despite we cannot observe computational power of our algorithm in full strength, obtained results are cogent.

Also performance was acceptable and for the data set discussed time of analysis for pair of builds did not exceeded 0.3 seconds, and in the terms of Table 2 characteristics of our data set lays in the top left corner giving hope that we will be able to process also other real data sets. Moreover the real attribute value distribution is far from homogenous so lowering total number of possible combinations.

5 Conclusions

In this paper 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. The proposed algorithm works with hierarchies of testgroups and attributes of testing environments. These hierarchies allow to generalize found problems by tested features. Relevant assigning of attributes produces compact analysis results and allows to decrease the duration of software development cycle.

Obtained results of analysis reaches initial goal: direct one's attention to most problematic points without necessity to search inside huge amount of data for any single test case with unexpected outcome. With appropriate vizualization these results give insight into quality dynamics of the sequence of builds and ensures solution to find main deterioration places by few clicks.

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

Our approach works with limited number of attributes and their values. However, practical application till now was not even close to these limits. So a part of the future work could be improving the method allowing to use larger number of attributes and values or ascertain that in real applications large number of attributes would not show up. Another way to improve effectiveness is calculate characteristics of composite testgroups in advance before analysis especially in cases when structure of testgroup hierarchy evolves over time. Interesting question is possibility to obtain exact result by the actual algorithm also if in the Propositions the first requirement is substituted by a weaker one.

Proposed analysis algorithm may be used during software development phase also even if we have testing results just for actual build: we can create mock testing results in advance and compare our actual testing results to them. If mock testing results describes test results according to the project plan then mock and real testing result comparison allows for various groups of users (testers, developers, managers) to see if development is happening according to the plan and if not then which areas are falling back the most.

Acknowledgements

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

References

1. Lee, M.L.: Change impact analysis of object-oriented software. PhD thesis, George Mason University (1998)
2. Li, W., Henry, S.: An empirical study of maintenance activities in two object-oriented systems. Journal of Software Maintenance: Research and Practice **7**(2) (1995) 131–147
3. Schneidewind, N.F.: The state of software maintenance. IEEE Transactions on Software Engineering (3) (1987) 303–310
4. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. 3 edn. John Wiley and Sons, Inc. (2012)
5. Wilson, B.: The Rootisserie – RCA Resources. `http://www.bill-wilson.net/root-cause-analysis/rca-resources` (2014) (Accessed August 19, 2015).
6. Rooney, J.J., Vanden Heuvel, L.N.: Root Cause Analysis For Beginners. Quality Progress **37** (july 2004) 45–53

7. Zeller, A.: Isolating cause-effect chains from computer programs. In: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, ACM (2002) 1–10
8. Bohner, S., Arnold, R.: Software Change Impact Analysis. Wiley – IEEE Computer Society Press (june 1996)