Pair-wise Component Compatibility Testing

Abstract: The concept of reusability has attributed to modern software systems, which are built with multiple third-party off-the-shelf components, each of which can have different versions. These interdependencies are complex and subject to changes in small time intervals, which creates a huge combination space of all the components involved, thereby making it infeasible to test all potential configurations of components. In the previous work we have modelled the entire configuration space of a multi-component based software system and tested all direct inter-component dependencies by sampling and testing in parallel.

In this paper we propose to extend the configuration sampling strategy to not just the direct dependencies but also include the indirect dependencies, while keeping the number of sampled configurations small, since indirect dependencies can affect the functionality of components. In the new strategy, we sample configurations to test pair-wise combinations of components under direct and indirect dependencies. We believe that the approach will be useful to reveal compatibility bugs that exist between specific component versions in a configuration.

Keywords: pair-wise testing, compatibility, component

1. Introduction

Testing component-based software system is a complex and time consuming task. Particularly, it is challenging to test the compatibility of the software on diverse environments that can consist of different versions of software components, including multiple compilers and third-party components (libraries or tools). That is, the build and execution environment are extremely heterogeneous. Each possible combination of components and their versions is a configuration that might contain unique errors [1]. Restricting the user environments (configurations) to a limited set of tested environments and releasing the software with undetected compatibility bugs, can degrade overall user experience.

Compatibility testing is performed to ensure that a software system can build and function properly in heterogeneous environments [2]. However, the large configuration space leads developers to test only a few popular configurations, leaving bugs to escape to the field. In our previous work [1] we modelled the entire configuration space of a software system and developed a system called Rachet to produce and test configurations that cover all direct dependencies between components. Given a set of component versions, their dependencies and a set of constraints set, Rachet can automatically generate the configuration space and test whether components can be built correctly over all configurations on which the components are designed to run. This approach was also extended to incrementally test components [3].

Testing direct dependencies could reduce the number of test cases drastically when compared to exhaustive testing. However, testing direct dependencies are not enough to test the correct behavior of components over diverse configurations, since the behavior can also be affected by indirect dependencies, especially when the components are reused by binding them dynamically at run time. If a compatibility bug exists between two components under an indirect dependency in a configuration, it cannot be detected by our previous work.

Pairwise testing has laid its basis on the fact that many faults are not due to complex configurations, but due to simple pair-wise interactions. It has the advantage of reducing the number of tests to be executed, because pair-wise bugs represent majority of combinatorial bugs. In this paper, we present an algorithm that samples configurations that test all pair-wise relationships between components under direct and also indirect dependencies, so as to validate both the correct build and functional behavior of components on user configurations, under the assumption that compatibility bugs arise in many cases from mismatch between two components deployed in a configuration.

The rest of the paper is organized as follows. We give a brief description on the pair-wise test case generation strategy in Section 2 and then explain the algorithm to sample configurations in Section 3. The last section compares the number of configurations produced incrementally and exhaustively.

2. Background: Pair-Wise Test Generation

Pair-wise test case generation is a combinatorial testing method in which each pair of input parameters to a system is tested for all the possible discrete combinations of those parameters. It selects a subset of

test cases that covers all possible pairs of combinations with the constraints of reducing the number of total test cases to be run, increasing the test effectiveness by spreading the range of coverage, while still maintaining a small set of test cases [4]. It has the advantage of reducing the number of tests to be executed, because pairwise bugs represent majority of combinatorial bugs [1].

There are several studies, both in computational and algebraic way[4], for pairwise test case generation, like Automatic Efficient Test Generator (AETG) [5][6] and IPO (In-Parameter-Order) [7, 8]. The IPO algorithm has lesser space and time complexity, is deterministic in nature and gives comparable results to the other methods. This is because it generates test set's one column (parameter) at a time. The ATEG algorithm is commercial and needs an expensive license if used [11]. The nature of IPO, combined with the above advantages explains its utility for this research problem and in this paper we modified the IPO algorithm to generate configurations that test all pair-wise relationships between component versions under dependencies. For a system with two or more input parameters, the IPO first generates the pairwise test set for the first two parameters and extends the test set to the other additional parameters. The process consists of two steps: (1) Horizontal growth and (2) Vertical growth. We briefly explain the steps with an example component-based system model given in Figure 1. Figure 1 shows that component A requires component D and one of either B or C, as direct prerequisite for building A. Similarly, D requires F and F requires G.

Horizontal Growth extends test cases by adding a new parameter to the existing test set. For example: Component A has version A_1 and Component B has three versions B_1, B_2, and B_3. From the versions, we can consider (A_1, B_1), (A_1, B_2) and (A_1, B_3) as a pair-wise test set for A and B. If we extend this test set by adding component C that has C_1 and C_2 as its versions, we get (A_1, B_1, C_1), (A_2, B_2, C_2) respectively. But there are several uncovered pairs {(B_1, C_2), (B_2, C_1), (B_3, C_1) and (B_3, C_2)} [3].

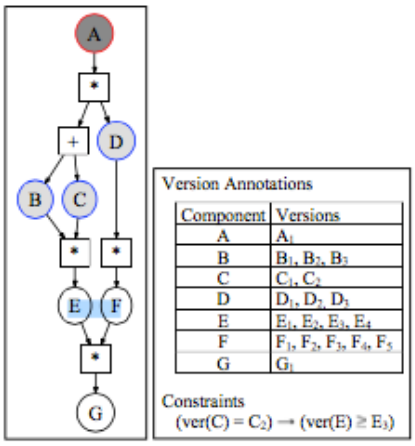

Figure 1 - An Example Component Dependency Model

Vertical Growth adds new tests to cover pairs not covered by horizontal growth. In the running example, to cover (B_1, C_2) we add a configuration (_, B_1, C_2) and then set a version for the component A, which gives (A_1, B_1, C_2) because the component A has only one version in the model. Similarly, for the pairs (B_2, C_1), (B_3, C_1) and (B_3, C_2) we can generate (A_1, B_2, C_1), (A_1, B_3, C_1) and (A_1, B_3, C_2) as the other tests. Thus, a total of 6 test sets are generated to cover all the pairs for components A, B and C. Although there is not an exemplary difference when compared to the 11 test sets generated by exhaustive testing, but it is seen that for a system with n components each having v versions the size of test set grows logarithmically for n and quadratically for v. This means that the difference would increase exponentially as the number of nodes or components to be tested increase. It is noteworthy that the minimum number of configurations are at least greater than the maximum number of version pairs between two components under dependency. In the example above, the number of configurations are greater or equal to 6, which is the version pairs between B and C.

3. Configuration Sampling Method

We describe in this section a method to generate configurations that test all version pairs of components under dependencies, given a software model represented as a component dependency model, which is an acyclic graph, as shown in Figure 1. Specifically, we apply a customized IPO algorithm to generate configurations. In the method, the component dependency model is traversed either horizontally, vertically or both. Horizontal traversal is to cover direct dependencies between components. We horizontally traverse a graph and find all the direct dependencies that have to be considered to test the correct build of a component over a set of prerequisite

components. Vertical traversal is to cover indirect dependencies that have to be tested for ensuring the correct functionality of a component in a configuration that contains other components under indirect dependencies. We traverse each directed path from each node(s) without any incoming edge to all node(s) without any outgoing edge.

In the algorithm 1 shown in figure 2, *rnode* is a relation node which defines the relationships between components, *path* is a Boolean variable that determines whether there is a directed path between two components, and *dpath* determines if two components are under direct dependency. π is the variable that represents the set of version pairs between different components and their versions (e.g., there are 4 pairs between A and B in the running example).

We apply the IPO strategy to traverse the dependency graph, starting from the root node of the graph. While traversing the graph in depth-first we check whether the relation node between components is AND or XOR.

```
if (rnode = "*") then
    if (path(n,p) && (!dpath(n , p))) then
        pairwise(n , p , π)
    end
    else
        do not include all pairwise combinations in π
    end
end
else if ( rnode = "+") then
    pairwise(n , p , π)
end
if (dpath(n , d)) then
    pairwise(n , d , π)
end
else
    do not include all pairwise combinations in π
end
```

Figure 2 – Configuration Sampling Algorithm

If the *rnode* is an AND, we check whether there is a directed path between components to be tested. In Figure 2, *n* is a component close to the root node and *p* is a set of components indirectly required for building *n* – that is, this is to test vertical relationships between components in the graph. If the condition is true, all pairwise combinations between the versions of these components are computed by the *pairwise* function and are kept in π. If the relation node is an XOR, any of the components can be used to build the dependent component as described earlier, but the pairwise testing needs to be applied with every XOR related component and the dependent component -- i.e., in the running example, A needs to be tested with both B and C, since one user might build A with B in the user machine and the other might build A with C. This is repeated to include every component in the dependency model. Next, we produce version pairs to cover all the direct dependencies between components – it tests horizontal relationships between components.

Horizontal coverage information is computed after the vertical coverage as many version pairs in the horizontal coverage are subsumed in the vertical coverage, reducing the number of test cases to be generated. The direct dependency coverage involves checking if the components have a direct path between them – i.e., there is a path that has no other component between them and are just connected by relation nodes [1]. Other components which do not have a direct dependency relation are not included in the pairwise test set, i.e. not all the version combinations are included in the set of version pairs.

```
//Horizontal Growth(T, a_i)
if (|T| ≤ q) then
    for (1 ≤ j ≤ |T|) extend the j^th test by adding values of v_j
end
else
    for ( 1 ≤ j ≤ q) extend the j^th test in T by adding values of v_j
    for(q < j ≤ |T|) extend the j^th test in T by adding one value of a_i
    such that it covers maximum pairs in π.
end
// Vertical Growth(T, π)
while (T does not cover all pairs between a_i and each of a_1, a_2, ..., a_{i-1})
do
    for each pair in π
    if (T' has a pair which contains " -" as the value of a_k and v as the
    value of a_i) then
        Use w to replace "-"
    end
    else
        Add a new test to T' where w has a value for a_k and v for a_i and
        "-" as other parameter values
    end
    T= T U T'
end
```

Figure 3 - Horizontal and Vertical Growth

The graph is traversed breadth-wise, starting from the root node, as it helps cover all the nodes and maximize the coverage with a small set of configurations. This is achieved by adding nodes to a queue and generate configurations incrementally by adding components in the order in the queue. In the horizontal growth shown in Figure 3, T is the configuration set under construction, a_1, a_2, …, a_{i-1} are components added in the configuration, q is the total number of versions of the component a and v_1, v_2, …, v_q are the versions of the component a. T' is an empty set considered for the vertical growth.

The horizontal coverage as mentioned above adds new component, the first "if" condition determines the case where the new node to be added has fewer versions than the number of configuration in T -- e.g., in the running example, there must be 4 configurations that cover version pairs between A and B, and when we add C we just need to use three of the

configurations for three versions of the component C. If the number of versions of newly added component is greater than the number of configurations, we perform the vertical growth to cover the version pairs not included in *T* during the horizontal growth.

At each stage of the growth, it is important to find a configuration that covers the maximum number of uncovered version pairs -- i.e. before expanding a configuration set with the new component version, the effect it has on the number of uncovered pairs should be measured. The one which covers the maximum of the new pairs should be chosen. Since this is quite significant to keep the generated configurations low and also it is not computationally feasible to generate each such configuration and choose an optimal one, a greedy approach can be considered.

5. Conclusion

The success of the pair-wise test case generation technique is based on the hypothesis that most defects in combinatorial testing are either single-mode defects (the function under test simply does not work) or double-mode defects (the pairing of two functions fails even though all others perform correctly)[9]. This characteristic fit well for the component compatibility problem. In this paper we presented our initial effort for systematically generating configurations that test all pair-wise combinations between components under direct and indirect dependencies. We plan to further revise the algorithm and evaluate its benefit by conducting experiments with real-world software systems.

Acknowledgement

This research was supported by the National Research Foundation of Korea (NRF-2013010695).

References

[1] Ilchul Yoon, Alan Sussman, Atif Memon, and Adam Porter, "Effective and Scalable Software Compatibility Testing", International Symposium on Software Testing and Analysis, pp. 63-74, Jul. 2008.

[2] Maity, S., Nayak, A., Zaman, M., Srivastava, A., and Goel, N., "An Efficient Test Generating Algorithm for Pair-wise Testing", In Proc. of the 14th Int. Symposium on Software Reliability Engineering, 2003.

[3] Ilchul Yoon, Alan Sussman, Atif Memon, and Adam Porter, "Towards Incremental Software Compatibility Testing", International ACM SIGSOFT Symposium on Component Based Software Engineering, pp. 119-128, Jun. 2011.

[4] Cassiano B. A. L. Monteiro, Luiz Alberto Vieira Dias, and Adilson Marques da Cunha, "A case study on pairwise testing applications", In Proc. of the 11th International Conference on Information Technology, 2014.

[5] Mats Grindal, Birgitta Lindstrom, Jeff Offutt, and Sten F. Andler, "An Evaluation of Combination Strategies for test case selection", Empirical software Journal, vol. 11, no. 4, pp. 583-611 , 2006 .

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design", IEEE Transactions On Software Engineering, volume 23, 1997.

[7] Kuo-Chung Tai and Yu Lei, "A test generation strategy for pairwise testing", IEEE transactions on Software Engineering, 28(1), 2002.

[8] Y. Cui, L. Li, and S. Yao, "A New strategy for pairwise test case generation", In Proc. of the 3rd International Symposium on Intelligent Information Technology Application, 2009.

[9] R. Kuhn, Y. Lei and R. Kacker, "Practical combinatorial testing: Beyond pairwise", IEEE IT Professional, vol. 10, no. 3 , pp.19 -23 , 2008.

[10] James Bach and Patrick J. Schroeder, "Pairwise Testing: A Best Practice That Isn't", PNSQC, 2004.

[11] Soumen Maity and Amiya Nayak, "Improved test case generation algorithms for pairwise testing", In Proc. of the 16th IEEE International Symposium on Software Reliability Engineering, 2005.