An Efficient Algorithm for Combining Verification & Validation Methods

Isela Mendoza[1], Uéverton Souza[1], Marcos Kalinowski[2], Ruben Interian[1], Leonado Gresta Paulino Murta[1]

[1]Computer Institute, Fluminense Federal University,
Niterói, Rio de Janeiro, Brazil
{imendoza, ueverton, rinterian, leomurta}@ic.uff.br
[2]Informatics Department, Pontifical Catholic University of Rio de Janeiro,
Rio de Janeiro, Rio de Janeiro, Brazil
kalinowski@inf.puc-rio.br

Abstract. An adequate combination of verification and validation (V&V) methods is important to improve software quality control throughout the development process and to reduce costs. However, to find an appropriate set of V&V methods that properly addresses the desired quality characteristics of a given project is a NP-hard problem. In this paper, we present a novel approach that combines V&V methods efficiently in order to properly cover a set of quality characteristics. We modelled the problem using a bipartite graph to represent the relationships between V&V methods and quality characteristics. Then we interpreted our problem as the Set Cover problem. Although Set Cover is considered hard to be solved, through the theoretical framework of Parameterized Complexity we propose an FPT-Algorithm (fixed-parameter tractable algorithm) that effectively solves the problem, considering the number of quality characteristics to be covered as a fixed parameter. We conclude that the proposed algorithm enables combining V&V methods in a scalable and efficient way, representing a valuable contribution to the community.

Keywords: combination, verification, validation, software quality, FPT, set cover, parameterized complexity.

1 Introduction

Studies suggest high costs related to quality assurance activities in software development projects [1]. The appropriate combination of verification and validation (V&V) methods is seen in the literature as a way to reduce these costs and increase product quality [2]. Over the years, some knowledge has been generated regarding V&V methods when observed in isolation [3]. However, the selection of different V&V methods as well as the interdependencies among them are still not well-understood [4].

A significant part of the software industry is made up of small and medium-sized companies that, given the lack of guidelines for performing the right combination of V&V methods, have difficulties in optimizing this combination for their context, increasing the costs of resources and time and mainly harming the quality of the produced software.

According to the Guide to the Software Engineering Body of Knowledge (SWEBOK) [5], verification is used to ensure that the software product is built in the correct way, that is, it complies with the previously defined specifications. On the other hand, validation guarantees that the product is adherent to the user needs. It is known that an adequate combination of V&V methods outperforms any method alone [6]. Most of the studies presented in a systematic mapping [9] do not clearly specify which V&V methods cover which quality characteristics (e.g., considering the quality characteristics described in the ISO 25010 quality model standard).

Finding a set of methods that together properly addresses all quality characteristics of interest can be seen as a Set Cover Problem (SCP) [11]. The SCP is a classic NP-hard problem in the computational complexity area, whose decision version belongs to the list of the 21 Karp's NP-complete problems [11]. This means that when the number of methods or quality characteristics increase, the performance of an algorithm to aiming at combining them in an optimal way would drastically decrease.

The existence of efficient algorithms to solve NP-complete, or otherwise NP-hard, problems is unlikely, if the input parameters are not fixed; all known algorithms that solve these problems require exponential time (or at least super-polynomial time) in terms of the input size. However, some problems can be solved by algorithms for which we can split the running time into two parts: one exponential, but only with respect to the size of a fixed parameter, and another polynomial in the size of the input. Such algorithms are denoted FPT (fixed-parameter tractable) in the *Parameterized Complexity* field, because the problem can be solved efficiently for small values of the fixed parameter [13][14][15]. This field emerged as a promising alternative for working with NP-hard problems [12].

In this paper, we propose an algorithm to obtain an optimal combination of methods covering software quality characteristics of interest in reasonable computational time. In order to find an optimal solution for the problem (based on the desired software quality characteristics and the relation between those and V&V methods, provided as input), we adopted a parameterized approach, considering the set of quality characteristics as fixed parameter, and obtaining an algorithm classified as FPT. The implemented FPT algorithm is the first of its kind that solves the SCP.

Our proposed algorithm reached its goals: it runs in $O(f(k) \times n)$, where the constant k is the number of quality characteristics, n is the number of methods, and $f(k)$ is some function of k. Considering that the number of quality characteristics of a given quality standard is always constant, the algorithm runs in polynomial time in terms of the number of V&V methods to be combined. As a result, it provides the minimum set of V&V methods addressing all quality characteristics of interest. While this information is surely useful, we are aware that companies may choose to complement these methods with others to further assure the quality of the product (or even chose others) and that other factors, such as cost, should be considered when taking the final decision.

The remainder of this paper is organized in the following sections: *Section 2* presents the background and related work concerning quality characteristics, V&V methods, and the combination of V&V methods. In *Section 3* the problem is modeled as a SCP. *Section 4* briefly introduces parameterized complexity theory. *Section 5* presents the FPT–Algorithm that obtains the optimal combination. *Section 6* contains a computational experiment analysis. *Section 7* discusses the contributions and limitations of our approach. *Section 8* presents the concluding remarks.

2 Background and Related Work

2.1 Quality Characteristics

Concerning software product quality characteristics, the product quality model defined in the ISO 25010 standard includes eight characteristics, for which quality requirements may be defined and measured during software development [10]. The characteristics and short descriptions for them, based on the ISO 25010 standard, can be found in Table 1.

Table 1. ISO 25010 Quality Characteristics.

Characteristic	***Short Description***
Function Suitability	Degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions.
Performance Efficiency	Represents the performance relative to the amount of resources used under stated conditions.
Compatibility	Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment.
Usability	Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.
Reliability	Degree to which a system, product or component performs specified functions under specified conditions for a specified period.
Security	Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.
Maintainability	Degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements.
Portability	Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another.

2.2 V&V Methods

Several V&V methods have been proposed over the years. In this paper we concentrate on a subset of V&V methods extracted mainly from the SWEBOK [5] and some other sources [19][20] to compose our corpus. The list of methods can be found in Table 2. Due to space constraints, a short description of the methods is not provided, but it can be easily obtained in the cited sources. It is noteworthy that there are variations for each of these methods (e.g., different control flow-based criteria, different inspection techniques). Nevertheless, we use this more generic classification as a starting point, given that characterizing all possible variations to obtain a representative input for our algorithm would be hard to accomplish. Thus, a method covering a quality characteristic, in the context of this paper, means that there are ways of appropriately addressing it using the method.

Table 2. V&V Methods.

Classification	***Method***
Based on Intuition & Experience	*Ad Hoc* Testing Exploratory Testing
Input Domain-Based	Equivalence Partitioning Pair wise Testing Boundary-Value Analysis Random Testing Cause-Effect Graphing
Code-based	Control Flow-Based Criteria Data Flow-Based Criteria
Fault-based	Error Guessing Mutation Testing
Usage-based	Operational Profile Usability Inspection Methods
Model-based	Finite-State Machines Workflow Models
Reviews	Walkthrough Peer Review or desk checking Technical Review Inspection

2.3 Combination of V&V Methods

It is known that the quality of software products is strongly dependent on the appropriate combination of V&V methods employed during development [2].

Experimental studies have long demonstrated that the use of combinations of different V&V methods to ensure the quality of a software is more effective than using isolated methods [7][8].

Elbertzhager *et al.* [9] conducted a mapping study concerning the combination of V&V methods. They describe two fundamental approaches: *Compilation* and *Integration*. We focus on the *Compilation* approach since our purpose is purely to combine existing V&V methods (*Compilation* process). We are not focusing on *creating* new techniques by combining different methods into one, nor in using the results of the application of some technique as an instance to apply another one (*Integration* of V&V methods).

In order to establish how other works perform the combination of V&V methods in the *Compilation* approach, Elberzhager *et al.* [9] created a categorization to classify and organize these studies into three subgroups.

In the first subgroup, static and dynamic techniques are combined, focusing on thread escape analysis, atomicity analysis, protocol analysis, vulnerability analysis, concurrent program analysis or on defects in general. All these combinations are supported by open-source or proprietary tools.

The second subgroup compares different testing and inspection techniques discussing advantages and disadvantages among them. In most cases, two or three techniques are compared to each other. Several studies initially perform inspections, followed by some tests, corroborating then the effectiveness of the combination of both techniques.

The last subgroup describes other combinations, such as testing techniques and inspections combined with formal specifications, bug-finding tools, comprehensive quality control processes in industrial environments, comprising several inspections and technical tests, requirements and static analysis, tutorials, simulations, and vision-based approaches.

The most cited papers in the systematic mapping [9] regarding the *Compilation* approach are: Basili [22], Kamsties and Lott [23], and Wagner *et al.* [24]. Basili [22] makes a comparison of three software testing techniques: reading of code by gradual abstraction, functional testing using equivalence partition and border value analysis, and structural testing using total coverage of criticism, according to efficiency, cost, and fault detection classes. Kamsties and Lott [23], evaluate three techniques through a controlled experiment: reading of code by gradual abstraction, functional (black-box) testing and structural (white-box) testing. Wagner [24] describes a case study where several projects are analyzed in an industrial environment. In this project, automatic static analysis, testing, and reviews are used to detect defects. Their results show that these techniques complement each other and that they should be combined.

In the systematic mapping [9], papers were analyzed until 2010. This led us to carry out an update regarding the compilation approach, with the aim of finding more relevant and recent papers from 2010 to present. Due to space constraints the details of our mapping update will not be provided in this paper and we focus directly on the recent related work.

Dwyer and Elbaum [25] suggest an approach based on dividing V&V methods into two main classes: those that make dynamic analyses (or focused on behavior of the system, e.g., testing) and those that use static analysis (typically focused on a single property of the system at a time). Runeson *et al.* [27] compare code inspections and structural unit tests by analyzing three replications of an experiment in order to know which method finds more faults. Olorisade *et al.* [28] investigate the effectiveness of two test techniques (partition of equivalence class and decision coverage) and one review technique (code by abstraction) in terms of their ability to detect faults. Cotroneo *et al.* [29] combine testing techniques adaptively, based on machine learning, during the testing process, by learning from past experience and adapting the technique selection to the current testing session. Bishop *et al.* [30] combine a monotonicity analysis with a defined set of tests, showing that, unlike "independent" dynamic methods, this combination provides a full error coverage. Solari and Matalonga [31] study the behavior of two techniques, equivalence partition and decision coverage, to determine the types of defects that are undetectable for either of them. Finally, Gleirscher *et al.* [32] analyze three different techniques of automated static analysis: code clone detection, bug pattern detection, and architecture conformance analysis. They claim that this combination tends to be affordable in terms of application effort and cost to correct defects.

It is noteworthy that none of the related work has implemented something similar to our proposal, since we focus on covering a set of quality characteristics with few methods, thus obtaining an optimal combination of V&V methods. While applying all available methods represents a solution, this option might not be applicable due to cost constraints.

3 Modeling the Problem

In this section we describe how the problem of finding the smallest combination of methods that cover a specific set of quality characteristics can be modelled as a *Set Cover Problem*.

Consider C as the set of characteristics, and $N(m)$ as the subset of C that is covered by a specific method m. We need to *find the smallest set of subsets that cover* C. The problem is NP-hard in general. The relation between the characteristics and the methods can be modeled as an undirected bipartite graph as shown in Figure 1.

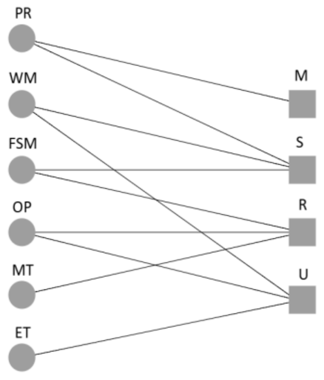

Fig. 1. Undirected bipartite graph. In the left-hand side the methods are positioned, and in the right-hand side the characteristics. Edges reflect the relationship between methods and characteristics. In the depicted instance, the set of methods contains the following elements: Peer Review (*PR*), Workflow Models (*WM*), Finite-State Machines (*FSM*), Operational Profile (***OP***), Mutation Testing (*MT*), and Exploratory Testing (*ET*). The set of characteristics is composed by four characteristics: Usability (**U**); Reliability (**R**); Security (**S**), and Maintainability (**M**). The graph shows a scenario in which method *PR* covers characteristics **M** and **S**, *WM* covers **S** and **U**, *FSM* covers **S** and **R,** *OP* covers **R** and **U,** *MT* covers **R**, and *ET* covers **U**.

The example instance was obtained from the results of a survey [21] that collected the opinion of experts about these V&V methods. The experts answered about their agreement on the suitability of the methods to address the different quality attributes of the ISO 25010 standard. The relationship between some method m and some characteristic c is obtained from the median of the survey answers (1 – disagree, 2 – partially disagree, 3 – partially agree, 4 – agree). In this example we considered that m properly covers c if the median is bigger or equal to 3. I.e., only methods that cover a quality characteristic to a certain degree will have edges in the graph. The outcome of the survey relating the quality characteristics to the V&V method can be seen in more details in [21].

For illustrative purposes we built this graph instance taking a subset of our real data, considering only four quality characteristics and some of the methods that can be used to properly address them according to the answers of the respondents (19 experts from 7 different countries, all with PhDs in software engineering, active in major software engineering and V&V venue committees, and with relevant publications in the area of V&V). The example perfectly serves our illustrative purposes to present the V&V method combination algorithm. Actually, this smaller example allows providing a better understanding of the algorithm's execution and correctness.

4 Parameterized Complexity

The *Parameterized Complexity* field emerged as a promising way to deal with NP-hard problems [12][26]. It is a branch of the Computational Complexity Theory that focuses on classifying computational problems according to their hardness with respect to different parameters of the input. The complexity of a problem is mainly expressed through a function of these parameters.

The theory of NP-completeness was developed to identify problems that cannot be solved in polynomial time if $P \neq NP$. However, several NP-complete and NP-hard problems still need to be solved in practice.

For many problems, only super-polynomial time algorithms are known when the complexity is measured according to the size of the input, and in general, they are considered "intractable" from the theoretical point of view assuming that P is different from NP. Nevertheless, for several problems we can develop algorithms in which we can split the running time into a part computed in polynomial time with respect to the size of the input and another part computed in at least exponential time, but only with respect to a parameter k. Consequently, if we set the parameter k to a small value and its growth is relatively small we could consider these problems as "manageable" and not "intractable" [13][14][15].

Thus, an important question arises: *"Do these hard problems admit non-polynomial time algorithms whose exponential complexity part is a function of merely some aspects of the problem?"* [12]. The existence of such algorithms was analyzed by Downey and Fellows in [13], and is briefly discussed in the next section.

4.1 Fixed-Parameter Tractable (FPT) Approach

The fixed-parameter tractable (FPT) approach [13] considers the following format for the problems: *"Given an object x and a non-negative integer k, the goal is to determine whether x has some property that depends on k?"* The parameter k is considered small compared to the size of x. The relevance of these parameters lies precisely in the small range of values they can take, being a very important factor in practice [12].

The FPT-algorithms sacrifice the execution time, which can be exponential, but guarantee that the exponential dependency is restricted to the parameter k, which means that the problem can be solved efficiently for small values of that fixed parameter. The use of these algorithms provides a more rigorous analysis of problem's time complexity since this complexity is generally obtained from the size of the input [12].

Formally, a problem Π belongs to the class FPT (it is fixed-parameter tractable) with respect to a parameter k if it admits an algorithm to solve it whose running time is of the form: $f(k) \times n^a$, where a is a constant, and f is an arbitrary computable function. Note that whenever k is bounded by a constant we have $f(k) = O(1)$, hence the running time of the algorithm will be polynomial.

Finally, for the problem of this paper, we present a fixed-parameter tractable algorithm where the size k of the set of characteristics to be covered is the parameter. I.e., we are limiting the complexity by the number of relevant product characteristics to be considered when developing the software.

4.2 Scalability of the FPT-algorithms

Scalability is the ability of a system or process to handle an increasing amount of data [16]. Computer algorithms can be called scalable if they are efficient when applied to large instances, i.e., instances with a large size of the input [17].

We can say that FPT-algorithms are scalable because they are efficient when executed in large instances. These algorithms take advantage of the specific structure of the instances, which is a differential when compared to exact or exhaustive search algorithms that require high computational time.

It is important to note that the studied problem can handle a large number of V&V methods, given that the number of quality characteristics tends to be relatively small. Therefore, an FPT-algorithm with respect to the number of characteristics to be covered will produce a tool for combination of V&V methods with high scalability.

Indeed, in our problem, the number of quality characteristics is already a known small integer (in the ISO standard this number is 8). Therefore, scalability relies on the ability to find the optimal solution even if the number of considered methods is growing. Our initial set comprises 19 methods, but additional methods have been reported by the survey respondents and our algorithm allows to efficiently work, for example, with 30, 50, or 100 methods.

5 FPT–Algorithm to Combine V&V Methods

The goal of the algorithm, shown in the Figure 2, is to obtain the optimal combination (smallest number) of V&V methods that properly cover all the relevant quality characteristics for the product to be developed. Certainly, a software organization could complement the resulting set with other V&V methods that cover similar quality characteristics to find more defects and to further enhance quality, but at least they would know about the minimum set of methods to consider in order to address all the quality characteristics that are relevant for the product to be developed. I.e., a combination such that there is a method properly addressing (i.e., with an edge in the graph for) each relevant quality characteristic and none of them remains uncovered.

The objective function is the number of selected methods that properly cover all the characteristics. The parameter to be set is the number of the selected quality characteristics. In this way, we are parameterizing the *Set Cover Problem* by the number of characteristics to be covered by the V&V methods.

Coming up next, we present some definitions that are used in the algorithm presented in Figure 2:

C – set of characteristics.
M – set of methods.
$N(m)$ – set of characteristics covered by the method m.
$P(c) = \{x \in M : c \in N(x)\}$ – set of methods that cover the characteristic c.
$R(m) = \{x \in M : N(x) \subseteq N(m)\}$ – set of methods that cover a subset of $N(m)$.

The input parameters are the set of characteristics C and the set of methods M. The redundant methods are removed in line 6 by using a simple preprocessing step. It removes methods that cover a subset of characteristics covered by any other method. A characteristic c is selected from the set of characteristics in line 7. The algorithm then focuses on selecting the method that will cover c in the optimal solution. In line 8, the variable C_t that contains the characteristics to be covered is initialized. A loop runs through all the methods that cover c in lines 9-25. The set M' that stores the methods that will be part of a feasible solution is initialized with method m in line 10. The set of characteristics to cover C_t is updated in line 11 by removing the characteristics already covered by m. The set M_t, containing the methods available to cover C_t, is initialized in line 12 with all methods of M except those covering a subset of $N(m)$. In lines 13-18 a loop is executed while there are characteristics c' that are covered by a single method m'. The variables C_t, M_t and M' are updated in lines 15-17. The available M_t methods and the characteristics that have not been covered until now are used to obtain an optimal sub-problem solution by recursively calling the *SetCover* algorithm. In line 19, the obtained optimal solution is stored in M'^{*}. If the methods selected in M' together with the optimal solution M'^{*} of the sub-problem improve the optimum value found so far (f^{*}), then $f^{*}$and M^{*} are updated in lines 20-23. The value of C_t is reinitialized in line 24. The best solution found (M^{*}), is returned as the optimal solution to the problem in line 27.

```
Algorithm 1 Set Cover Algorithm,  Parameters: sets C, M
1:  $M^* \leftarrow M$
2:  $f^* = |M|$
3:  if $C = \emptyset$ then
4:      return $\emptyset$
5:  else
6:      $M \leftarrow RemoveSubsets(M)$
7:      $c \leftarrow SelectCharacteristic(C)$
8:      $C_t \leftarrow C \setminus \{c\}$
9:      for all $m \in P(c)$ do
10:         $M' \leftarrow \{m\}$
11:         $C_t \leftarrow C_t \setminus N(m)$
12:         $M_t \leftarrow M \setminus R(m)$
13:         while $\exists c' \in C_t : |P(c') \cap M_t| = 1$ do
14:             Let $m' \in P(c')$
15:             $C_t \leftarrow C_t \setminus N(m')$
16:             $M_t \leftarrow M_t \setminus \{m'\}$
17:             $M' \leftarrow M' \cup \{m'\}$
18:         end while
19:         $M'^* \leftarrow SetCover(C_t, M_t)$
20:         if $|M'^*| + |M'| < f^*$ then
21:             $f^* \leftarrow |M'^*| + |M'|$
22:             $M^* \leftarrow M'^* \cup M'$
23:         end if
24:         $C_t \leftarrow C \setminus \{c\}$
25:     end for
26: end if
27: return $M^*$
```

Fig. 2 Pseudocode of the Set Cover FPT-Algorithm.

5.1 Execution of the Set Cover Algorithm

Taking the graph represented in Figure 1 as the entry of the algorithm, we now illustrate the execution of the pseudocode. After initialization steps 1-5, line 6 removes redundant methods. In this case, methods *MT* and *ET* are removed, because they cover only one characteristic, already covered by other methods. The result is shown in Figure 3.

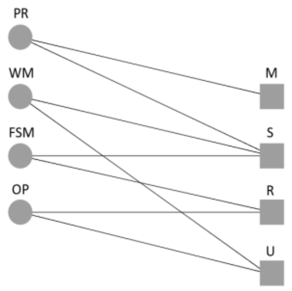

Fig.3 Algorithm execution. State of the graph after the preprocessing step.

Afterwards, the first characteristic **M** is chosen as c, and all the methods that cover **M** must be considered in the loop that begins on line 9. Therefore, method PR is selected. In line 11, we remove all the characteristics already covered by PR, that is, **M** and **S**. The variable M_t gets the set of methods {*WM, FSM, OP*} in line 12. Since there are no characteristics covered by only one method, the loop on lines 13-18 does not perform any action, and the algorithm is called recursively in line 19 with set of characteristics {**R**, **U**}, and set of methods {*WM, FSM, OP*} as parameters. Figure 4 illustrates the graph at this stage.

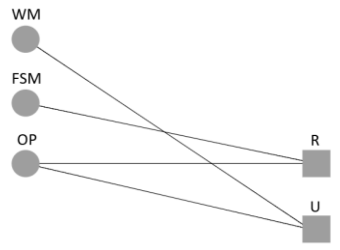

Fig. 4 Algorithm execution. State of the graph after the recursive call.

Finally, the algorithm is executed again from the beginning. Methods *WM* and *FSM* are immediately removed as redundant, and the remaining method *OP* is selected to cover the last two characteristics. The variables C_t and M_t became empty, and in the next recursive call, the stopping criterion is reached. The *OP* method is returned as a solution of the instance represented in Fig. 4, forming the final solution of the whole instance together with already selected method *PR*. The smallest set of methods M^* is set as {*PR, OP*}, and the optimal value f^* is set to 2.

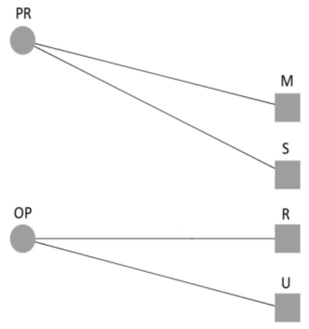

Fig.5. Methods that form the optimal solution returned by the algorithm when executed in the graph. If **M** is selected as the first characteristic at the beginning of the execution, then the optimal set of methods returned by the algorithm is {*PR, OP*}.

As can be observed from the execution, the algorithm considers all the possible ways of covering the quality characteristics, keeping the most efficient ones. In this sense, the obtained solution can be considered optimal for the problem and model we pose. In fact, the algorithm is able to determine the optimal combination (smallest number) of V&V methods that properly cover quality characteristics of interest for a product to be developed based on any initial graph configuration connecting V&V methods to the quality characteristics they properly address.

5.2 Running Time Analysis

Suppose that there are n methods in the set M, and there are k characteristics in the set C, being k some small integer. We note that a naive (brute force) algorithm would test all solutions (subsets of the set M) and chose which of them cover C having smaller size. Because there are 2^n subsets of the set M, this naive algorithm has a time complexity of $O(2^n)$. This exponential order is intractable even for some relatively small values of n, like 30 or 40 (which could easily be achieved when including specific variations of the V&V methods as input).

Instead, the proposed algorithm tries to determinate which method is the best option to cover each characteristic. After choosing some characteristic c, the algorithm tries to select each method that properly covers c, covering the rest of characteristics recursively. Because the number of methods that cover each characteristic is at most n, the order of this algorithm can be initially bounded by $O(n^k)$. In general, this order is already better than the 'naive' solution.

Nevertheless, we improve the upper bound of our algorithm's running time by refining the actual number of methods it will analyze. In fact, there are only 2^k different ways of covering a set C of k elements. If there is more than 2^k methods, then necessarily there will be two of them that cover exactly the same set of characteristics. That means that these two methods would be indistinguishable to our algorithm; that is, if they cover the same characteristics, any one of them can be used. Using this fact, we can successively preprocess the input, improving the algorithm performance from $O(n^k)$ to $O(f(k) \times n)$, where $f(k) \leq (2^k)^k = 2^{k^2}$. In our case, $k = 8$ and this means that $f(k)$ is bounded above by a constant, i.e., $f(k) = O(1)$. Then, we have a linear algorithm for the problem instead of an exponential or even a $O(n^8)$-time algorithm.

Once again, the upper bound for $f(k)$ is improved (decreased) using the fact that if some method m_1 covers a subset of characteristics covered by some other method m_2, then m_1 can be removed from the set of methods. This is because if m_1 is actually chosen, you can instead choose m_2, since m_2 is 'better' method in the sense that it covers all that m_1 covers, and possibly more. Lubell [18] showed that there are no more than $\binom{k}{\lfloor k/2 \rfloor}$ combinations with the property that no one is a subset of the other. This implies that for $k = 8$, there can be much less than 2^k different methods with the property that there is no method that covers a subset of characteristic of some other method. In particular, for $k = 8$ there can be at most $\binom{8}{4} = 70$ methods satisfying this property, and there can be at most $\binom{7}{3} = 35$ of these methods covering one common characteristic. Therefore, the redundant methods are removed by using a simple preprocessing step that searches for methods that cover a subset of characteristics covered by any other method. At each iteration of the algorithm, the number of characteristics to be covered decreases and the previous steps of the algorithm are repeated considering a decremented k value.

Summarizing, it holds that:

$$f(k) < \binom{k-1}{\lfloor (k-1)/2 \rfloor} \times \binom{k-2}{\lfloor (k-2)/2 \rfloor} \times \cdots \times \binom{2}{1} \quad \textbf{(1)}$$

For $k = 8$, it follows that $f(k) < 35 \times 20 \times 10 \times 6 \times 3 \times 2$, then our $O(f(k) \times n)$-time algorithm is an efficient (linear) algorithm where $f(k)$ is bounded above by a constant. In practice, this constant is even lower, and does not depend on the number of existing methods, which produces scalability with respect to the number of methods to be worked.

6 Computational Experiments

Several experiments were performed to assess the algorithm presented above. The algorithm was implemented in C# programming language and compiled by Roslyn, a reference C# compiler, in an Intel Core i3 machine with a 2.0 GHz processor and 4 GB of random-access memory, running under the Windows 10 operating system.

A number of test problems created by a random generator is considered. Each test problem has two parameters: the number of vertices n and the probability p of a method to cover a characteristic.

The FPT–algorithm is also executed on the instance obtained from the survey described in [21], according to the criteria explained in the *Section 3* when we build the example with a subset of this same data.

Table 3 shows the optimal solution sizes and execution times (in seconds) for the FPT-Algorithm solver with and without instance preprocessing (Fig. 2, line 6) and a naive algorithm (brute force), for each instance. The name of the instance indicates the number of methods, followed by the probability of a characteristic to be covered by a method, in percent. The optimal solution sizes (number of methods returned) are equal for all instances, indicating the correctness of the Algorithms.

The FTP–Algorithm with the preprocessing is more efficient than without preprocessing and the Naive Algorithm, obtaining the result in less than 0:01 seconds in all cases even for the biggest instances, unlike the other algorithms in which for some instance sizes the solution is not found in a reasonable waiting time (-----). The FTP–Algorithm without processing proves to be, in turn, more efficient than the Naive Algorithm, executing more instances with better runtime.

For the instance obtained from the survey [21], the FPT–Algorithm with the processing of the instance returned the methods: 12, 19 and the FPT–Algorithm without the processing and the Naive Algorithm returned the methods: 12, 18. The solution returned by the FPT–Algorithm with the processing contains the method 19 that covers a superset of the characteristics covered by the method 18 in the others algorithm solutions, showing that FPT–Algorithm with preprocessing performs better when concerning coverage, by using this additional comparison criterion.

Table 3. Computational Experiments

Instance	Runtime FPT-Alg (with pre-processing)	Optimal Solution FPT-Alg (with pre-processing)	Runtime FPT-Alg (no pre-processing)	Optimal Solution FPT-Alg (no pre-processing)	Runtime Alg-Naive (brute force)	Optimal Alg-Naive (brute force)
Instance_20_10	0.00031	5	0.00375	5	0.14907	5
Instance_20_20	0.00047	3	0.01094	3	0.17703	3
Instance_20_50	0.00062	2	0.08859	2	0.22297	2
Instance_50_10	0.00031	5	0.14359	5	-----	-----
Instance_50_20	0.00110	2	4.97453	2	-----	-----
Instance_50_50	0.00094	2	11.4025	2	-----	-----
Instance_100_10	0.00094	2	64.7025	2	-----	-----
Instance_100_20	0.00125	3	-----	-----	-----	-----
Instance_100_50	0.00110	2	-----	-----	-----	-----
Instance_200_10	0.00344	3	-----	-----	-----	-----
Instance_200_20	0.00188	2	-----	-----	-----	-----
Instance_200_50	0.00094	1	-----	-----	-----	-----
Instance_500_10	0.00359	3	-----	-----	-----	-----
Instance_500_20	0.00469	2	-----	-----	-----	-----
Instance_500_50	0.00234	1	-----	-----	-----	-----
Instance_1000_10	0.00766	3	-----	-----	-----	-----
Instance_1000_20	0.00578	2	-----	-----	-----	-----
Instance_1000_50	0.00500	1	-----	-----	-----	-----

7 Discussion

Our proposed algorithm is effective, being able to provide the optimal combination (smallest number) of V&V methods properly covering a set of chosen quality characteristics to be considered when developing a software product. Additionally, it is more efficient than brute-force or exhaustive search algorithms and its execution time properties match the particularities of the problem well. Indeed, the algorithm can be applied to instances of different sizes, making our approach scalable, i.e., suitable for larger case studies (for instance, considering more V&V methods, including specific variations of the more generic methods used for our sample).

There is, however, a basic assumption for applying the algorithm, which is having a defined input with information on which V&V methods properly address the different quality characteristics. For illustrative purposes, our example was based on initial outcomes of an expert survey. It is also noteworthy that our set of 19 V&V methods represents generic methods for which several variations are available (e.g., applying specific testing criteria or variations of inspection methods). While they perfectly fit our illustrative example and allowed us getting feedback from experts on whether they can be employed to properly address quality characteristics, information on more specific methods could be provided as input to combination algorithm. We highlight that this initial configuration is out of the scope of the intended contribution of this paper and that companies could use an initial configuration based on their own sets of evidence on the V&V methods they typically use or on their own elicited expert beliefs.

Moreover, from a practical point of view, companies might decide to complement the optimal solution provided by the algorithm by applying additional V&V methods that cover similar quality characteristics (e.g., aiming at finding additional defects and further enhancing product quality), in particular for critical projects. However, using our approach at least they would know about a minimum set of methods that would allow them avoiding neglecting quality characteristics that are relevant for the product to be developed.

Also, specialists on software engineering economics might argue that our solution providing the smallest number of V&V methods is not considering the cost of applying each method. However, to address this issue we would need to know the relative cost among the V&V methods and this information is extremely context specific and hard to generalize. We are aware of this limitation and further addressing it is part of our future work. A solution option to handle this issue when using the approach described in this paper would be removing the methods that are cost restrictive from the initial configuration.

8 Concluding Remarks

In this paper, we modeled the problem of finding a combination of V&V methods to cover software quality characteristics as the Set Cover problem, a NP-hard combinatorial optimization problem. We defined a parameterized FPT algorithm that is specially designed for our instances, since typically the number of considered quality characteristics is small. Provided by a valid input, the proposed algorithm is able to efficiently provide an optimal combination (smallest number) of V&V methods properly covering a set of chosen quality characteristics to be considered when developing a software product. Additionally, we showed that it is more efficient than Naive (brute-force) algorithms. Furthermore, the algorithm can be applied to instances of different sizes, making our approach scalable, i.e., suitable for larger studies (for instance, considering more V&V methods).

Our future works consist of development of a support tool that, given a set of selected quality characteristics and an initial configuration (e.g., from the survey results, or any other source such as within-company expert belief elicitation), provide the optimal combination of V&V methods. Finally, for now we focused on product quality, and a next step would be to integrate cost-related issues into the approach. Moreover, we believe that the Fixed-Parameter Tractable algorithm approach can be applied to solve other problems in the software engineering domain and that sharing our V&V method combination experience with the community could foster discussions towards other graph theory-based solutions for relevant software engineering problems.

Acknowledgment

The authors would like to thank CNPq and FAPERJ for the financial support (Project N^o. E-26/010.001578/2016, Title: "*Resolution of Critical Problems of the Software Industry through Graph Theory and its algorithms*"). Thanks also to the survey respondents, which provided us the initial configurations to test our approach.

References

[1] Meyers, G.J., Badgett, T., Thomas, T., Csandler, C.: The Art of Software Testing. Wiley, 3rd ed., ISBN: 978-1118031964, (2011).

[2] Feldt, R., Torkar, R., Ahmad E., Raza, B.: Challenges with Software Verification and Validation Activities in the Space Industry, Third International Conference on Software Testing, Verification and Validation (ICST), (2010).

[3] Boehm, B., Basili, V.: Software Defect Reduction Top 10 List. IEEE Software, Vol. 34 (1), January (2001) 135-137.

[4] Feldt, R., Marculescu, B., Schulte, J., Torkar, R., Preissing, P., Hult, E.: Optimizing Verification and Validation Activities for Software in the Space Industry. Data Systems in Aerospace (DASIA), Budapest (2010).

[5] Bourque, P., Fairley, R.E: SWEBOK Guide V3.0, Guide to the Software Engineering Body of Knowledge, IEEE Computer Society (2004).

[6] Endres, A., Rombach, D.: A Handbook of Software and Systems Engineering: AddisonWesley (2003).

[7] Myers, G. J.: A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections. Communications of the ACM, vol. 21, no. 9 (1978) 760–768.

[8] Wood, M., Roper, M., Brooks, A., Miller, J.: Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study, In: Proceedings of the 6th European Software Engineering Conference, Springer New York, Inc. (1997) 262– 277.

[9] Elberzhager, F., Münch J., Nha, V.T.N.: A systematic mapping study on the combination of static and dynamic quality assurance techniques, Inf. Softw. Technol., 54 (1) (2012) 1-15.

[10] ISO25000 Software Product Quality, ISO/IEC 25010, http://iso25000.com/index.php/en/iso-25000-standards/iso-25010, Official site (2011).

[11] Karp, R.M.: Reducibility Among Combinatorial Problems, In R. E. Miller and J. W. Thatcher (editors), Complexity of Computer Computations. New York: Plenum. (1972) 85–103.

[12] Santos, V.F., Souza, U.S.: Uma Introdução à Complexidade Parametrizada. Anais da 34º Jornada de Atualização em Informática, CSBC (2015) 232-273.

[13] Downey, R. G., Fellows, M. R.: Parameterized complexity, Monographs in Computer Science. Springer (1999).

[14] Flum, J., Grohe, M.: Parameterized complexity theory. Springer (2006).

[15] Niedermeier, R.: Invitation to fixed-parameter algorithms. Oxford Lecture Series in Mathematics and Its Applications, Oxford University Press (2006).

[16] Bondi, A.B.: Characteristics of scalability and their impact on performance. Proceedings Second International Workshop on Software and Performance WOSP (2000) 195-203.

[17] Laudon, K.C., Traver, C.G.: E-commerce: Business, Technology, Society. Stanford University, USA (2008).

[18] Lubell, D.: A short proof of Sperner's lemma. Journal of Combinatorial Theory, 1 (2) (1996) 299.

[19] Wagner, S.: Software Product Quality Control. Springer (2013).

[20] Wiegers, K.E.: Peer Reviews in Software: A Practical Guide. 1st Edition, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (2002).

[21] Mendoza I., Kalinowski M., Souza U., Felderer M.: Relating Verification and Validation Methods to Software Product Quality Characteristics: Results of an Expert Survey. 11th Software Quality Days (SWQD), Springer Lecture Notes on Business Information Processing, Vienna, Austria (2019). (to appear).

[22] Basili, V.R: Comparing the effectiveness of software testing strategies. IEEE Transactions on Software Engineering, SE-13 (12), USA (1987) 1278-1296.

[23] Kamsties, E., Lott, C.M.: An empirical evaluation of three defect-detection techniques. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 989, ESEC 1995, Sitges, Spain (1995) 362-383.

[24] Wagner, S., Jürjens, J., Koller, C., Trischberger, P.: Comparing bug finding tools with reviews and test. 17th IFIP TC6/WG 6.1 International Conference Testing of Communicating Systems (TestCom), Lecture Notes in Computer Science, 3502, Montreal, Que, Canada (2005) 40-55.

[25] Dwyer, M. B., Elbaum, S.: Unifying Verification and Validation Techniques: Relating Behavior and Properties through Partial Evidence. FSE/SDP workshop on Future of Software Engineering Research (FOSE), Santa Fe, New Mexico, USA (2010) 93-98.

[26] Cygan, M., Fomin, F.V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, Ma., Pilipczuk, Mi., Saurabh, S.: Parameterized Algorithms. Springer International Publishing, Switzerland (2015).

[27] Runeson, P., Stefik, A., Andrews, A., Grönblom, S., Porres, I., Siebert, S.: A comparative analysis of three replicated experiments comparing inspection and unit testing. Proceedings 2nd International Workshop on Replication in Empirical Software Engineering Research (RESER), art. no. 6148335, Banff, AB; Canada (2012) 35-42.

[28] Olorisade, B.K., Vegas, S., Juristo, N.: Determining the effectiveness of three software evaluation techniques through informal aggregation. Information and Software Technology, 55 (9) (2013) 1590-1601.

[29] Cotroneo, D., Pietrantuono, R., Russo, S.: A Learning-Based Method for Combining Testing Techniques. Proceedings 35th International Conference on Software Engineering (ICSE), art. no. 6606560, San Francisco, CA, USA (2013) 142-151.

[30] Bishop, P., Bloomfield, R., Cyra, L.: Combining Testing and Proof to Gain High Assurance in Software: A Case Study. IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), Article number 6698924, Pasadena, CA, USA (2013) 248-257.

[31] Solari, M., Matalonga, S.: A controlled experiment to explore potentially undetectable defects for testing techniques. Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE), Canada (2014) 106-109.
[32] Gleirscher, M., Golubitskiy, D., Irlbeck, M., Wagner, S.: Introduction of static quality analysis in small- and medium-sized software enterprises: experiences from technology transfer. Software Quality Journal, 22 (3) (2014) 499-542.