Optimal Variability Selection in Product Line Engineering

Rafael Pinto Medeiros*, Uéverton dos Santos Souza†, Fábio Protti† and Leonardo Gresta Paulino Murta†
*Universidade do Estado do Rio de Janeiro
Rio de Janeiro, Rio de Janeiro, Brazil
Email: rafaelmedeiros@uerj.br
†Instituto de Computação, Universidade Federal Fluminense
Niterói, Rio de Janeiro, Brazil
Email: {usouza,fabio,leomurta}@ic.uff.br

***Abstract*—Software Configuration Management is being adopted with success in the development of individual products, mainly supporting the creation of revisions and sporadically supporting the creation of variants via branches. However, some emerging software engineering methods, such as product line engineering, demand a sound support for variants to generate customized products from product lines. The adoption of branches in this scenario does not scale due to the huge number of interdependent variants. The main goal of this paper is to systematize a way to select the best variants of a product line to generate products according to specific user needs. Moreover, our paper contributes on providing an algorithm for product generation from product lines. Our algorithm can be easily implemented in the existing product line approaches as an alternative for product selection.**

***Keywords*-And/Or Graphs, Software Configuration Management, Software Versioning.**

I. Introduction

Software Configuration Management (SCM) is a discipline applied during the software development process to control the software evolution [9]. As changes can happen anytime during the software development process, SCM activities are developed to identify the changes; to assure the changes are being correctly implemented; and to inform the changes to people who have interests on it [6]. Due to that, it is possible to conclude that the main objective of SCM is not to avoid changes, but provide control and coordination over the changes. Moreover, it is also concerned on providing consistency among interdependent components and allowing the reconstruction of previous states of the software.

As an important system of SCM, the Version Control System (VCS) is responsible for managing different versions of a product. Usually, VCS are developed through models that define the objects to be versioned, version identification and organization, as well as operations to retrieve previous versions and create new ones. However, versions can serve for different purposes [5]. Versions that are used to replace other versions of the same component are called revisions. On the other hand, versions that live together with other versions of the same component, acting as alternatives, are called variants.

SCM is being adopted with success in the development of individual products, mainly supporting the creation of revisions and sporadically supporting the creation of variants via branches. However, some emerging software engineering methods, such as product line engineering, demand a sound support for variants to generate customized products from product lines. The adoption of branches in this scenario does not scale due to the huge number of interdependent variants.

In product line engineering, a software product line is composed to represent the commonalities and variabilities of a software family. According to [4] a product line is usually combined with feature models and configuration knowledge, responsible to identify the possible product features and how these features interplay. The derivation process is fundamental in product line engineering. This process consists in composing specific products from the product line according to some user requirements. According to [8] the existing approaches to model product line architectures are predominantly focused on enumerating the available component versions for each possible product that can be generated from the product line. However, conceptual differences in product features and their interrelationships are not easily expressed in the available modeling constructs. On the other hand, a goal-based approach provides a natural mapping to modeling product line architectures, considering the user needs during the product generation.

The main goal of this paper is to systematize a way to select the best variants of a product line to generate products according to specific user needs. This can result in VCS that are better prepared to support product line engineering and other methodologies that focus on the conception of families of products. Moreover, our approach formalizes this product composition according to the SCM terminology. Product line researchers can build upon our approach to implement their derivation process according to their specific technologies (i.e., features model, architecture description languages, etc.).

This paper is organized into 5 sections besides this introduction. Section 2 presents some background related to software versioning concepts. Section 3 introduces the combinatorial problem that emerges from the product line scenario. Section 4 presents our approach to generate the optimal product from a product line. Section 5 presents some related works on software versioning. Finally, section 6 presents final considerations

and future works.

II. Background on Software Versioning

During the development process, software engineers need to build specific versions of the software. A software version is structured by components, and each component also have specific versions. However, the selection of different components or different versions of the same components leads to different versions of the software as a whole. At an exponential rate, different software versions start to become possible of building, even versions that are not aligned to the user desires or requirements [5].

A version model identifies and organizes items that should be versioned and items that should not be versioned. Each SCM system provides its own version model according to the target domain and builds over its own formalism. According to Conradi and Westfechtel [5], there are many ways to represent a version model, such as file-based models, where versioning is applied on files and directories, and data-based models, where versioning systems manage versions of objects stored in a database. Other resources are commonly applied to express versioning rules, such as textual languages.

The version model can be described in terms of a product space and a version space. In order to achieve proper existence of a software, it is necessary to define what composes the software itself. In other words, the software components, their role to the final product, their functions inside the software, and their relationships to each other should be defined. This arrangements of components is defined in [5] as the product space. The product space represents a universe of items and their relationships, without considering their versions.

On the other hand, based on the definition in [5], the version space represents the universe of possible versions of a software, highlighting the common traits and differences of the versions of an item. The transformation of a non versioned item (in the product space) into a single, double, or multi versioned item can be seen as a binding of an specific moment in time of the product space with the version space. Hence, a software version is composed by versions of the software components, and generated by the combination of the product space with a moment of existence of each component in the version space. An item without this moment of existence in the version space is a non versioned item, with its changes implemented through overwriting.

A set of versions can be defined basically in two ways [5]: extensional versioning and intensional versioning. We can differ extensional from intensional versioning due to the reasons that demand the generation of a new version. Extensional versioning is realized through enumerating its components' versions; from this point, the user is able to retrieve a version v_x, apply changes over v_x, and generate a new version v_z. Intensional versioning is capable of generating versions from larger spaces to satisfy a set of goals established by the user.

As a consequence, extensional versioning only allow the retrieval of previously created versions, while intensional versioning allows the retrieval of versions on demand, combining component versions that may never worked together before and that can potentially generate inconsistent software versions in terms of the user needs. This is one of the main reasons why current SCM systems usually adopt extensional versioning.

III. The Optimal Intensional Version Problem

An *object base* is defined as a combination of product space and version space, comprehending all the versions of a software [5]. This base contains all the software components, all their versions, non versioned objects and their relationships.

The arrangement of an intensional version can be seen as a selection of objects inside the base in a way that the selected objects are enough to build the product version. This selection is structured to satisfy the needs that motivated the product development. During the versioning process, these needs are transformed into affirmatives, named versioning rules. Therefore, the selection is directed by a set of versioning rules.

This method leads to a combinatorial problem inside the intensional versioning. From a large number of potential versions, only a few of them sustain the consistency needed to satisfy the set of versioning rules. In summary, the configuration process is based on satisfying restrictions and demands to lead to a functional resulting software version. The versioning rules in this article represent restrictions and demands, and the object base is the universe of all possible versions, including inconsistent ones.

According to Conradi and Westfechtel[5], the most difficult factor when facing the combinatorial problem on intensional versioning is to eliminate inconsistent versions. After that, it relays on the configurator to build the version that matches a certain query.

We present a formalization to the optimal intensional versioning problem as follows:

Problem: OIV – Optimal Intensional Version
Input: An object base and a set of versioning rules
Output: To find in the object base, if possible, an optimized version of the software that satisfies the set of versioning rules.

In this article the software's object base is represented through an And/Or graph, according to the representation introduced in [5]. And/Or graphs [12] provide a general model for integrating product space and version space. An And/Or graph is a directed graph G, such that every vertex $v \in V(G)$ possesses a label $f(v) \in \{And, Or\}$. In this graph, the directed edges represent dependency relationship among the vertices: And-type vertices (represented through an arc between its out-edge) depend strictly on all its out-neighbors; Or-type vertices depend only on one of its out-neighbors.

To represent the object base the source vertex maps to the software as a whole, and the other vertices map to software modules or components and its versions. In this graph, the And out-edge represent composition relationship and the Or out-edge represent possible versions of an item.

According to [5] a distinction is made between And and Or

edges, which emanate from And and Or nodes, respectively. An unversioned product can be represented by an And/Or graph consisting exclusively of And nodes/edges. A versioned product is modeled by introducing Or nodes. Versioned objects and their versions are represented by Or nodes and And nodes, respectively.

For example, Figure 1 illustrates an And/Or graph representing a base of objects. It is important to notice that fine-grained visibilities of an object base can also be represented by And/Or graphs using the same formalisms discussed in this paper. However, we focused our examples on a coarse-grained visibility to allow a better understanding of the problem and our proposed solution. The main difference of coarse and fine granularity is the number of components and their versions, which enforces the necessity of faster algorithms to solve the problem when fine-grained components are in place.

IV. Solution to the OIV Problem

In this section we introduce a transformation of the OIV problem into a combinatorial problem related to And/Or graphs (MIN–AND/OR). Through this transformation we systematize a process where the versioning rules are converted into weights at the edges of the object base's graph. Thus, we present an approach that enables the development of algorithms for solving the problem of intentional versioning. In addition, we present a backtracking algorithm for MIN–AND/OR which in turn solves the OIV problem.

The MIN–AND/OR problem consists on finding a subgraph (*solution subgraph*) that matches a set of restrictions considering a weighted And/Or graph with a source vertex s. We introduce the definition of the MIN–AND/OR Problem[11] as follows:

Problem: MIN–AND/OR

Input: An acyclic And/Or graph G=(V, E) properly connected and weighted with a source vertex $s \in V$, where each vertex v possesses a label $f(v) \in \{And, Or\}$ and each edge possesses a weight $\tau(e) \geq 1$.

Output: A subgraph $H = (V', E')$ of G such as the sum of the weights of its arcs is minimum, and satisfies:

1) $s \in V'$;
2) if $v \in V'$ and $f(v) = And$ then all out-arcs of v must belong to E';
3) if $v \in V'$ and $f(v) = Or$ then exactly one of the out-arcs of v must belong to E'.

To transform the OIV Problem into the MIN–AND/OR Problem, it is necessary to weight the And/Or graph G (representing the object base) according to the query built from the versioning rules.

A. Weighting the And/Or graph G

Considering the And/Or graph G it is possible to add weights to its edges using two types of versioning rules:

- Configuration Rules: a series of affirmative statements originated from questions asked to the stakeholders of the software; from these affirmatives, it is possible to exclude some versions from the space of possible versions. As an example of the formulation of the configuration rules, the configuration manager can ask how the user communicates with the system (local access, remote access). It becomes possible then to formulate the configuration rule: the software must provide local access support. This configuration rule discards the components that provide remote access or portability features, because they are not demanded. These rules consider the *functional* requirements of the product being generated, and can be replaced by an existing feature-based approach if the product line engineering process already has one in place.
- Classification/Qualification rules: a set of items classified/qualified considering their priority to the stakeholders according to the needs and demands of the software. For instance, the stakeholders can ask for a product that allows high-end control capabilities and is efficient. The configuration manager then is able to formulate the Classification/Qualification rules: Control Capability, Efficiency. Versions of components that best excel at those two items are preferred. These rules consider the *non-functional* requirements of the product being generated. They are important to solve situations where there are open alternatives even after imposing the rules related to *functional* requirements.

1) Applying Configuration Rules: The set of configuration rules characterizes the first process of weighting the graph of the base of objects. The possibilities that do not accomplish the defined configuration rules are unable of existing in a solution-subgraph with non-infinite cost. Along the weighting process, the configuration rule analysis dictates if an edge receives an infinite weight or not, sometimes excluding a large amount of versions (by excluding one component, all its versions are automatically excluded).

The configuration rules are therefore defined as affirmative statements, originated from a set of questions answered in cooperation with the stakeholders. The software engineers should create questions with a high capability of constraining the version space.

When applying configuration rules, the questions work as a way to formulate demanding affirmatives - the configuration rule itself. All edges of G pointing to vertices that oppose the affirmative receive infinite weight; all other vertices have their edges set with weight 1. According to this algorithm, a version with non-infinite weight is a version that matches the configuration rule not necessarily by accomplishing the configuration rule, but by not going directly against it.

2) Applying Classification/Qualification Rules: With the application of the classification/qualification rules, edge values bounded to each component will emerge. By doing so, some versions shall protrude, therefore distinguishing themselves from the others. The stakeholder must classify some criteria, according to his/her priority, aiming to reflect what is expected in the final version of the software. In other words, the

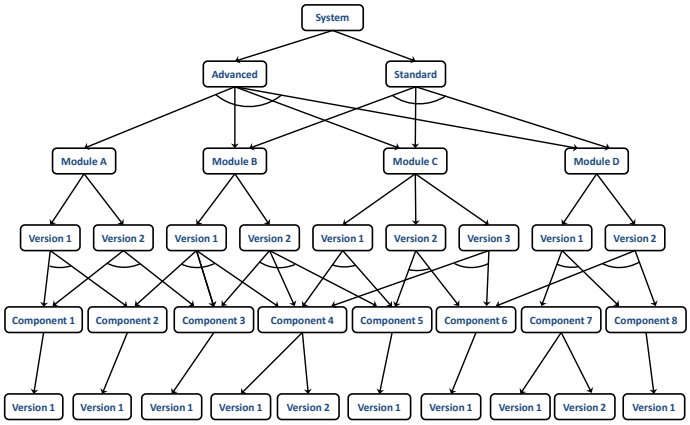

Fig. 1. Example of a object base represented by an And/OR graph.

stakeholder is demanded to prioritize non-functional aspects of the software.

In this paper we named these criteria as Classification/Qualification rules. As an example of classification rules, the stakeholders can be asked to classify the following items according to their priority: Control Capability, Efficiency, and Support. It becomes possible, after this step, to choose the version that best matches the stakeholder's priorities.

The Classification/Qualification rules initiates by considering a set of criteria $\{s_1,\ s_2,\ s_3,...,s_k\}$, in general, some of these criteria can be obtained from the ISO/IEC 9126 for the evaluation of software quality [2], such as: usability, efficiency, maintainability and portability. These criteria are classified by the stakeholder with the tags *high relevancy*, *regular relevancy*, *low relevancy*. Next, the software engineer classifies each component of the object base according to the quality (*bad*, *regular*, *good*, *excellent* or *does not interfere/ not relevant*) of each s_j criterion.

After that, it is proposed a weighting process of the And/Or graph as follows:

To each component c_i do:

1) Add
   - weight 0 to the criterion classified as excellent or not relevant;
   - weight 1 to the criterion classified as good;
   - weight 2 to the criterion classified as regular;
   - weight 3 to the criterion classified as bad.
2) Calculate
   - the sum of the weights of the high relevancy criteria as HR;
   - the sum of the weights of the regular relevancy criteria as RR;
   - the sum of the weights of the low relevancy criteria as LR;
3) Sum W to the weight of the vertex in-edge that represents the component c_i ($W = 3HR + 1.5RR + LR$).

The W value is calculated to ensure that the weight of a high relevancy criterion is the double of the regular relevancy criterion weight and the triple of the low relevancy criterion weight. Consequently, a high relevancy criterion classified/qualified as good corresponds to a regular relevancy criterion classified as regular or to a low relevancy criterion classified as bad. Clearly the higher relevancy criteria weights more, what is justified because of the MIN-AND/OR problem structure. The MIN-AND/OR Problem consists on the weight minimization in which the higher relevance criteria weight is expected to be the smallest. Figure 2 illustrates the object base shown in Figure 1 after applying a possible set of versioning rules.

At this point it is important to highlight that other possible versioning, with different versioning rules, would produce another weighting of the G graph; however, as the object base remains the same, the graph itself is not rebuilt. In addition, even when there are changes on the object base, the G graph is capable of easy adapting and still has not to be rebuilt.

With the And/Or graph properly weighted, it is possible to state that:

1) a version of the software corresponds to a subgraph solution-subgraph) of G, such as:
   - The source vertex (representing the software) belongs to solution-subgraph.
   - If a And-type vertex belongs to the solution-subgraph then all its out-edges do as well.
   - If an Or-type vertex belongs to the solution-subgraph then exactly one of its out-edges does as well.

   This observation is easily verifiable. The out-edges of an And vertex represents composition, that is, if a module belongs to a version z so do all of its components. On the other hand, out-edges of an Or vertex represents version options of a component; as a software uses at most one version of each component, this observation is true.
2) To find the optimal intensional version of the OIV Problem corresponds to find the solution-subgraph with minimum cost of the G graph. In our example, the

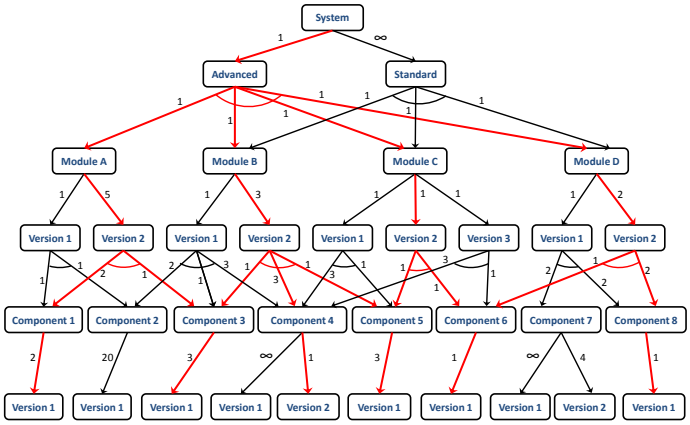

Fig. 2. And/Or graph G representing an object base after applying the versioning rules.

optimal intensional version highlighted in red in Figure 2.

From these statements it becomes possible to adopt the existing results in the literature regarding the MIN–AND/OR Problem to achieve results to the OIV Problem. Among the results of the MIN–AND/OR Problem, it is known that the problem is $NP\text{-}Hard$ in general. However, it becomes polynomial when the G graph is a tree. In addition, it is known that finding a viable solution (not necessarily optimal) is polynomial [11], which means finding a version that satisfies the configuration rules of the OIV Problem is also polynomial.

B. Backtracking algorithm to The MIN–AND/OR Problem

At this point, it is known that the And/Or Graph represents the software object base; consequently a solution subgraph of G corresponds to a specific version of the software. We present the Algorithm 1 that, considering an And/Or graph G, returns its optimal solution-subgraph, if there is such. In other words, this algorithm finds the optimal solution to the MIN–AND/OR problem and consequently to the OIV problem.

The algorithm is divided in 3 parts; in the first part it realizes a topological sorting of the G vertices, which is possible because G is an acyclic graph. The sorting assures that a vertex v_j $(1 \leq j \leq n)$ is always before its out-neighbors on the vertices arrangement.

Next, the procedure Generate is called to enumerate all possible $solution\text{-}subgraphs$ of G. These subgraphs are stored in V, where:

1) The vertices with the register flag set to 1 belong to the current solution-subgraph.
2) If the position i in V represents an Or-type vertex that belongs to the current subgraph, then exactly one index j is stored in V[i], where v_j is an out-neighbor of v_i.
3) If the position i in V represents an And-type vertex that belongs to the current subgraph, then a set with the index j of each vertex v_j which is an out-neighbor of v_i is stored in V[i].
4) If the position i in V represents a $sink$ then an empty set is stored in V[i].

With the V array properly populated, it becomes possible to go through it building the solution-subgraph and calculating its weight, which is done by the $Cost$ procedure. When the algorithm terminates, the result is the solution-subgraph with the smallest weight.

The complexity of the algorithm is $O(n.K)$ where K is the number of possible $solution\text{-}subgraphs$, and the time to generate each solution-subgraph is $O(n)$.

V. Related Work

In [1], Ghose and Sundararajan presented a work for measuring software quality using pricing and demand data, quantifying the degradation associated with software versioning.

Conradi and Westfechtel [5] introduced a uniform method to represent version models using graphs, providing to configuration managers a more flexible and illustrative way to work with intensional and extensional versioning. In the same direction, in this paper, we show that intensional versioning rules can be represented by weights on the graph's edges, and the main problem on intensional versioning can be seen as a classic combinatorial problem.

According to [7], product derivation is the process of making decisions to select a particular product from a product line and to customize it for a particular purpose. In product derivation, the variability provided by the product line is communicated to the users of the product line and based on customers' requirements, variants are selected from the product line thus resolving the available variability. The most important requirement for tool-supported product derivation is obviously to support resolving variability. Users need tools that present the available variability and let users decide about choices interactively. Many SPLE tools for variability resolution are model-based, e.g., they visualize feature or decision models and allow users resolving variability by selecting features [3] or making decisions [10]. The approach presented in this paper can be adapted for variability resolution merging selecting features and making decisions.

Algorithm 1: Backtracking for MIN–And/Or
input : An And/Or graph G; two arrays V and SS of n positions (inicially empty).
output: SS storing the optimal solution-subgraph of G.

```
begin
    Assume v_1, v_2, v_3, ..., v_n an arrangement of the vertex of
    G, given by an topological sorting;
    for i:=1 to n do
        if v_i is an And-type vertex then
            V[i].out:=O_i;
            (O_i is the set of index of the out-neighbours of v_i);
        else
            V[i].out:={};
        if i = 1 then
            V[i].flag:=1;
        else
            V[i].flag:=0;
        V[i].in:={}
    smallest:=∞;
    Generate(1,smallest,V,SS);
end
procedure Generate(i, smallest:integer,V, SS:array)
    if i ≠ n then
        if V[i].flag=0 or v_i is sink then
            Generate(i+1,smallest,V,SS);
        else
            if v_i is an Or-type vertex then
                foreach out-neighbor v_j of v_i do
                    V[i].out := {j};
                    V[j].flag:=1;
                    V[j].in := V[j].in ∪ {i};
                    Generate(i+1,smallest,V,SS);
                    Clear(i,{j});
            else
                foreach out-neighbor v_j of v_i do
                    V[j].flag:=1;
                    V[j].in := V[j].in ∪ {i};
                Generate(i+1,smallest,V,SS);
    else
        if Cost(V,1) ≤ smallest then
            SS:=V;
            smallest := Cost(V, 1);
end
procedure Cost(V: array i: integer)
    if V[i].out ≠ {} then
        foreach j ∈ V[i].out do
            value:= weight of edge(v_i, v_j) + Cost(V,j)
        Cost:=value;
    else
        Cost:=0;
end
procedure Clear(k: integer O_k: set of integer)
    foreach j ∈ O_k do
        V[j].in := V[j].in \ {k};
        if V[j].in = { } then
            if V[j].out ≠ { } then
                Clear(j,V[j].out);
            V[j].flag:=0;
end
```

VI. Conclusion

In this paper we built upon the representations of version models introduced by Conradi and Wesfechtel [5] and proposed an approach of graph weighting and search for obtaining the optimal intensional version of a software system according to the preferences of the stakeholder. We showed that a query over the object base can be translated into a weighting of a And/Or graph G, where its edges are weighted according to specific criteria and priorities. In addition, we presented a transformation of the Optimal Intensional Versioning Problem to the combinatorial MIN–And/Or Problem, thus utilizing some of the MIN–And/Or existing results to solve the OIV Problem. Finally, we presented an algorithm that takes the weighted And/Or graph G and finds, if possible, the optimal version, that is, the version with the smallest cost that matches all the existing conditions over the referred object base.

For future work, we would like to apply the proposed algorithm in projects with different characteristics to evaluate how the algorithms behave and if there are situations where intensional versioning should be avoid at all. Moreover, we intend to investigate situations where the stakeholder preferences or even the criteria relevance change dynamically. These situations are common in dynamic software product lines, used in self-adaptive systems at runtime.

References

[1] A. Ghose, A. Sundararajan, Software versioning and quality degradation? An exploratory study of the evidence, Leonard N. Stern School of Business, Working Paper CeDER, New York, NY, USA, pp. 05-20, July 2005.

[2] ISO, ISO/IEC 9126 - Software engineering - Product quality, International Organization for Standardization, 2001.

[3] K. Czarnecki, S. Helson, U.W. Eisenecker, Staged configuration using feature models, Proc. of the 3rd International Software Product Line Conference (SPLC 2004), Springer, Berlin/Heidelberg, Boston, MA, USA, 2004, pp. 266-283.

[4] K.C. Kang, J. Lee, P. Donohoe, Feature-Oriented Product Line Engineering, IEEE Software, v. 19, issue 4, pp. 58-65, July/August 2002.

[5] R. Conradi, B. Westfechtel, Version models for software configuration management. ACM Computing Surveys, v. 30, issue 2, pp. 232-282, June 1998.

[6] R. Pressman, Software Engineering: A Practitioner's Approach. McGraw, 2009.

[7] R. Rabiser, P. Grünbacher, D. Dhungana, Requirements for product derivation support: Results from a systematic literature review and an expert survey, Information and Software Technology, v. 52, pp. 324-346, 2010.

[8] S. A. Hendrickson, A.van der Hoek, Modeling Product Line Architectures through Change Sets and Relationships, 29th International Conference on Software Engineering (ICSE'07), pp. 189-198, 2007.

[9] S. Dart, Concepts in configuration management systems, SCM 91 Proceedings of The 3rd International Workshop on Software Configuration Management, ACM Press, New York, NY, USA, pp. 1-18, June 1991.

[10] T. Asikainen, T. Soininen, T. Männistö, A Koala-based approach for modelling and deploying configurable software product families, Proc. of the 5th International Workshop on Product-Family Engineering (PFE 2003), Siena, Italy, Springer, Berlin/Heidelberg, 2003, pp. 225-249.

[11] U. dos S. Souza, A Parameterized Approach for And/Or Graphs and X-of-Y graphs. Master Thesis, Federal University of Rio de Janeiro, 2010.

[12] W. F. Tichy, A data model for programming support environments and its application, Proc. of the IFIP WG 8.1 Working Conference on Automated Tools for Information System Design and Development, New Orleans, North-Holland, pp. 31-48, Jan 1982.