Collaborative Product Configuration: Formalization and Efficient Algorithms for Dependency Analysis

Marcílio Mendonça[(1)], Donald Cowan[(1)], William Malyk[(1)], Toacy Oliveira[(2)]

[(1)] David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, ON, Canada
Email: {marcilio, dcowan, wmalyk}@csg.uwaterloo.ca

[(2)] Departamento de Informática,
PUC-RS, Rio Grande do Sul, RS, Brazil
Email: toacy@inf.pucrs.br

Abstract - **In the Software Product Line approach, product configuration is a key activity in which stakeholders choose features for a product. This activity is critical in the sense that careless feature selections might lead to undesirable products. Even though product configuration is seen as a team activity in which divergent interests and views are merged into a single consistent product specification, current configuration technology is essentially single-user-based. This configuration approach can be error-prone and time-consuming as it usually requires numerous interactions between the product manager and the stakeholders to resolve decision conflicts. To tackle this problem we have proposed an approach called "Collaborative Product Configuration" (CPC). In this paper, we extend the CPC approach by providing efficient dependency analysis algorithms to support the validation of workflow-based descriptions called CPC plans. In addition, we add to previous work by providing a formal description of the approach's concepts, an augmented illustrated example, and a discussion covering several prototype tools now available.**

Index Terms— **Software Product Lines, Collaborative Product Configuration, Feature Models, Dependency Analysis Algorithms, Logics in Product Lines.**

I. INTRODUCTION

Software Product Lines (SPLs) [1] is a software development approach that capitalizes on reusable assets as a means to improve software quality, shorten time-to-market and substantially lower production costs. In the context of SPLs, product configuration is a key activity in which stakeholders choose features for a product. Product configuration is critical in the sense that careless feature selections might lead to an undesirable product. A feature model [2] is commonly used to guide product configuration as it breaks down the commonalities and variabilities of product line members in terms of a hierarchy of configurable parts, i.e., product features. In the last decade or so, feature models have raised significant interest within the SPL community as demonstrated by the ever-growing number of supporting approaches [3][4][5] and tools [6][7][8].

Even though product configuration is seen as a team activity in which divergent interests and views are merged into a single consistent product specification, current configuration technology is essentially single-user-based. That is, a single role called product manager[1] is usually accountable for interpreting and translating requirements into feature selections. As a consequence, the involvement of stakeholders in the configuration process is essentially *passive;* they are limited to providing requirements and hope that useful features are added to the product. We claim that single-user-configuration is error-prone and time-consuming requiring numerous interactions between the product manager and the stakeholders to resolve eventual decision conflicts. In addition, requirements misinterpretations might lead to products that do not fully satisfy the customers. Therefore, there is an urge for approaches that provide an explicit support for what we refer to as *Collaborative Product Configuration* (CPC).

To tackle this problem, we have already proposed a preliminary approach to CPC [9]. The approach introduced basic CPC concepts and provided an algorithm to support the automatic generation of process models to describe the configuration decision-making process. In this paper, we show an improved version of the approach. That is, rather than relying on a process generation algorithm the approach now encourages the product manager to create a configuration plan (*CPC plan*) on her own and yet provides efficient dependency analysis algorithms that support the validation of the plan. The belief is that because the product manager has an overall view of the stakeholders' expectations and their expertise she is in a much better position to create plans to coordinate the stakeholders' interactions than a less flexible generation algorithm. The paper also adds to the previous work by providing a formal description of the CPC approach, an augmented illustrated example, and a discussion of several prototype tools now available.

The remainder of this paper is organized as follows. Section II provides background on feature models and hypergraphs. An overview of the concepts behind the CPC approach is presented in Section III and Section IV where they are more rigorously described. Section V discusses dependency analysis algorithms for feature models and how such algorithms can be used to derive constraints to validate CPC plans. Performance analysis of those algorithms is discussed in Section VI. Section VII covers an illustrated example of the CPC approach for a web portal product line. In section VIII related work is discussed and section IX concludes the paper.

[1] *Also referred to as the* ***application engineer***

© 2008 ACADEMY PUBLISHER

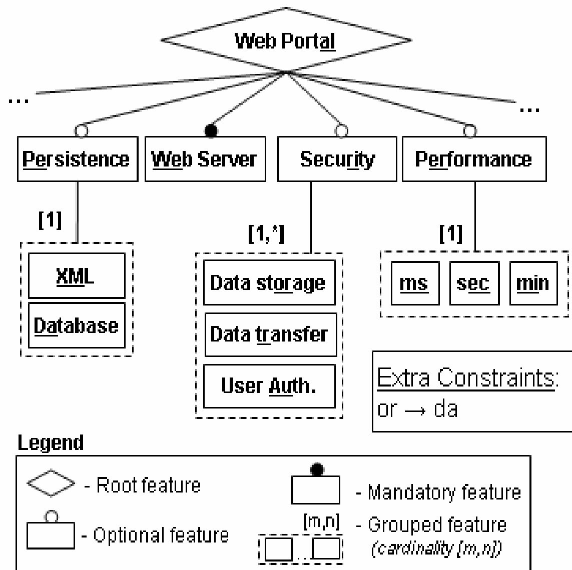

Figure 1. Partial feature model for a web portal product line. (feature names have been abbreviated as underlined)

II. Background

This section provides a background discussion on feature models and hypergraphs.

A. Feature Models

Feature models were originally proposed in a domain analysis method called FODA (Feature-Oriented Domain Analysis) [2] as a means to represent the commonalities and variabilities of system families. Since then, many extensions have been proposed to feature models in attempts to improve properties such as succinctness and naturalness [10]. Nevertheless, there seems to be a consensus that at a minimum feature models should be able to represent optional and mandatory features as well as feature groups. Figure 1 shows a partial feature model of a web portal product line. The root feature (diamond shaped) is called the *concept* node. Rectangles represent features. White circles on top of rectangles indicate optional features (e.g. *Persistence*, *Performance*) while mandatory feature rectangles are decorated with black circles on top (e.g. *Web Server*). Feature groups are represented by dashed rectangles enclosing two or more grouped features. Cardinality relations containing lower and upper bounds are indicated for grouped features (e.g. [1,*]). Extra arbitrary feature relations can be attached to the feature model. For instance, constraint (*or* → *da*) in Figure 1 indicates that a database must be available in the product if data storage security is a requirement (see box *Extra Constraints*). A product specification can be obtained by selecting features in the feature model. For instance, a valid product specification S_1 for the partial feature model in Figure 1 could be (using abbreviated feature names): S_1 = {*pe, we, er, ec*}. S_2 = {*pe, er, ec, xm, da*} is an example of an invalid specification since feature *we* is mandatory but not included in the specification and features *xm* and *da* are mutually exclusive but appear together in the specification.

B. Hypergraphs

A hypergraph is a generalization of a graph in which a single edge can connect multiple nodes. Formally, a hypergraph is defined as a pair (X,E), where X is a set of vertices, and E is a set of hyperedges where each edge can connect any number of vertices. Hypergraphs have already been used in SPLs as a means to support the automatic extraction of feature models from propositional formulas [11].

In CPC, we are interested in two specific hypergraph operations: *merge* and *projection*. In addition, we want to use hypergraphs to support variable dependency analysis on propositional logic formulas. The next sections discuss these aspects in more detail.

1) Constraints

A constraint in propositional logic can be written using the binary operators ∧ (conjunction), ∨ (disjunction), → (implication), ↔ (biconditional), the unary operator ¬ (not), the Boolean values *true* and *false*, and a set of Boolean variables. Formulas (1) and (2) below are examples of propositional formulas.

(1) $a \vee b \vee \neg c$
(2) $(a \rightarrow x) \wedge (b \rightarrow x) \equiv (\neg a \vee x) \wedge (\neg b \vee x)$

We say that two variables depend on each other if a particular assignment to one of the variables might cause the automatic instantiation of the other variable through propagation in order to maintain the truth of the formula. For example, in formula (1), if variables *a* and *b* are set to *false*, variable *c* must be *false*, otherwise the formula will evaluate to *false*. The same relation holds for all variables in formula (1). Thus, we say that *a*, *b*, and *c* are interdependent variables. *We use hypergraphs to represent clusters of interdependent variables such that vertices represent variables and hyperedges indicate variable dependency sets. Hypergraphs (a) and (b) encode variable dependency for formulas (1) and (2), respectively.*

(a) $V=\{a,b,c\}$, $E=\{\{a,b,c\}\}$
(b) $V=\{a,b,x\}$, $E=\{a,x\},\{b,x\}\}$

2) Merge

We define the merge of hypergraphs as the function: *merge*_hg: H_1:(V_1, E_1) *x* H_2:(V_2, E_2) → H_3:(V_3, E_3), such that the resultant hypergraph H_3 contains the set of vertices $V_3 = V_1 \cup V_2$, and the set of hyperdeges E_3 that merges (union set) the hyperdedge sets in E_1 and E_2 that share a common vertex. Hyperedges can be labeled such as in hypergraph H_3.

Example: Merge: H_1 *x* H_2 → H_3

H_1: $V_1 = \{t_1,t_2,t_3,t_4,t_5\}$ $E_1 = \{\{t_1\},\{t_2\},\{t_3\},\{t_4,t_5\}\}$
H_2: $V_2 = \{t_3,t_4,t_6,t_7\}$ $E_2 = \{\{t_3,t_6,t_7\},\{t_4\}\}$
H_3: $V_3 = \{t_1,t_2,t_3,t_4,t_5,t_6,t_7\}$,
$E_3 = \{A=\{t_1\},B=\{t_2\},C=\{t_3,t_6,t_7\},D=\{t_4, t_5\}\}$

The merge operation will be used later to perform dependency analysis on feature model extra constraints.

© 2008 ACADEMY PUBLISHER

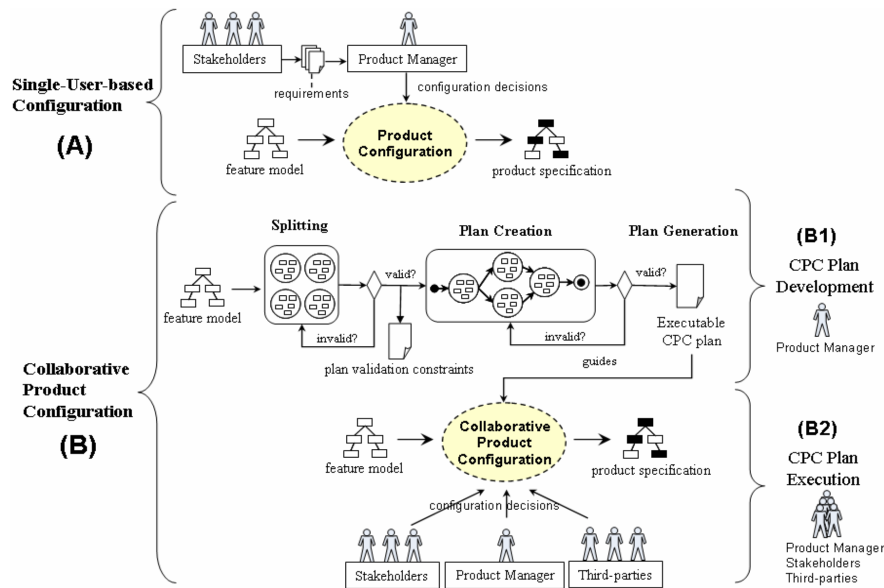

Figure 2. Single-User-based (A) and Collaborative (B) Product Configuration Scenarios

3) Projection

We define the projection of hypergraphs as a function: *projection_hg*: H_1:(V_1,E_1) *x* H_2:(V_2,E_2) → H_3:(V_3,E_3), such that the resultant hypergraph H_3 contains the set of vertices $V_3 = E_2$, and the set of hyperedges E_3 that uses the hyperedge elements in E_1 to search hyperedges in E_2 and group them together as hyperedges in E_3. An example will help to clarify this operation. Consider hypergraphs H_3 and H_4 with labeled hyperedges below. The projection of H_4 on H_3 results on a hypergraph H_5. Notice that the vertices in H_5 (*A, B, C* and *D*) correspond to the labeled hyperedges of H_3 and that the hyperedges of H_4 were used to locate the corresponding hyperedges in H_3 and group them as hyperedges in H_5.

Example: Projection: (H_4 *x* H_3) → H_5

H_3: $V_3 = \{t_1,t_2,t_3,t_4,t_5,t_6,t_7\}$,
$E_3 = \{A=\{t_1\},B=\{t_2\},C=\{t_3,t_6,t_7\},D=\{t_4, t_5\}\}$
H_4: $V_4 = \{t_1,t_2,t_3,t_4,t_5,t_6,t_7\}$
$E_4 = \{W=\{t_1,t_5\},X=\{t_2,t_7\},Y=\{t_3,t_6\},Z=\{t_4\}\}$
H_5: $V_5 = \{A,B,C,D\}$,
$E_5 = \{\{A,D\},\{B,C\},\{C\},\{D\}\}$

By projecting one hypergraph on another we create a relation among their hyperedges with the hope of finding a meaningful outcome. For instance, suppose that hypergraph H_4 above represents a set of tasks (vertices) and dependencies among those tasks (hyperedges) such that each group of dependent tasks (e.g. {t1,t5}, {t2,t7}) has to be performed at a specific time. Suppose hypergraph H_3 indicates the people (*A, B, C,* and *D*) in charge of carrying out each task. In order to identify *which people need to work together because of tasks dependencies,* we perform a projection of H_4 (interdependent tasks) on H_3 (task assignments) and obtain hypergraph H_5. The result of the projection shows that *A* and *D* need to work together because their tasks t_1 and t_5 are interdependent. The same happens with *B* and *C* for tasks t_2 and t_7, respectively. Hypergraph projections will be used later to support dependency analysis on feature models in the context of CPC.

THE CPC APPROACH

Figure 2 depicts two possible configuration scenarios. Scenario-A illustrates a traditional non-collaborative configuration process in which stakeholders provide requirements to the product manager who in turn interprets and translates them into configuration decisions. A feature model serves as the input to the configuration process and the result is a complete valid product specification.

Scenario-B describes the CPC approach consisting of two distinct phases. In phase-1 (scenario-B1) the goal is to produce a plan to coordinate configuration decision-making. The product manager commonly leads phase-1 as she has an overall view of the participants in the configuration process (called *configuration actors*), and their expertise. The first step in phase-1 is called *Splitting*. Splitting partitions the universe of configuration decisions into more fine-grained units called *configuration spaces* based on a particular criterion (e.g. knowledge domain). At this time, the product manager also defines who is to be accountable for which decisions. There are some rules to constrain a splitting that will be shown later.

Once the feature model splitting is complete and validated a step known as *plan creation* starts. The product manager needs to devise a plan to that will guide collaborative configuration-decision making (CPC plan). The CPC plan is a workflow-like structure that specifies a set of configuration tasks and their order of execution. For instance, in a *configuration session* a group of configuration actors work together on one or more configuration spaces. The order in which the configuration sessions are conducted is defined by the

© 2008 ACADEMY PUBLISHER

product manager and described in the plan. Merging sessions are used to resolve conflicting situations, i.e., when decisions made in concurrent configuration sessions violate a constraint in the feature model. CPC plans are subject to automatic validation since invalid plans can lead to incorrect product specifications. The last step in phase-1 is *plan generation* in which an executable encoding to represent the CPC plan is generated, i.e., the high-level plan description is converted into a machine-executable format (e.g. a workflow described in XML).

Once the CPC plan is validated and generated, phase-2 can be started. Phase-2 represents the actual product configuration process that is similar to the single-user-based configuration (scenario-A) and that aims at configuring an initial feature model and producing a valid product specification. The major difference is that multiple configuration actors are now directly involved in the configuration decision-making and hence need to follow a prescribed plan. Tool support in phase-2 is desirable as a means to enforce that the CPC plan is strictly followed.

III. A Rigorous Description of CPC

Up to this point, the CPC approach has been discussed in terms of high-level concepts. In the following, a more rigorous notation is used to describe such concepts and their relations.

Definition 1 *(Boolean Constraint Satisfaction Problem or Boolean-CSP)*: A *Boolean-CSP* or SAT is a triple $<X,D,C>$ where X is a set of variables of domain $D = \{0,1\}$, and C is a set of constraints on those variables. Every constraint $C_i \in C$ restricts the combined values of the variables in C_i, denoted by $V(C_i)$. An assignment $A(C_i)$ to constraint C_i is a set of tuples $<X_i, V_i>$ such that $X_i \in V(C_i)$, $V_i = 0$ or 1. We say that $A(C_i)$ satisfies C_i if it causes the constraint to evaluate to 1 *(true)*. A solution to the problem is an assignment to the variables in X that satisfies all the constraints in C.

Definition 2 *(Feature Tree or FT)*: A *FT* is a tree-like structure where nodes represent product features and edges indicate feature relations. The root of the tree is a special node called *concept node*. The other kinds of nodes are *optional*, *mandatory* and *grouped* features. Except for the root node that is always *true*, features can be *true,* indicating that they should be included in the product specification, or *false,* otherwise. Optional features are always *false* if their parents are *false*. Mandatory features always assume the same truth value of their parents (*true* if their parent node is the root of the tree). Grouped features represent a mutual exclusion relation according to a given cardinality [m ,n], where m and n indicate the minimum and maximum number of features in the group that can be set to *true*, respectively. If $m > 0$, at least m grouped features must be *true* whenever the parent feature is *true*. Moreover, regardless of the value of m and n, if any grouped feature is set to *true* so is its parent feature. Similarly, if any grouped feature is set to *false* so are its descendant features.

Definition 3 *(Feature Model or FM)*: A *FM* is composed by a *FT* and a set of additional relations that constrain the features in *FT* (extra constraints). Such relations are usually expressed using a constraint language (e.g. propositional logic).

Definition 4 *(Product Configuration Problem or PCP)*: A *PCP* can be encoded as a *CSP*. Features are variables, the domain of values is {0,1}, where 1 (*true*) and 0 (*false*) indicate that the feature has been included or excluded from the product specification, respectively, and the feature tree relations and the extra constraints represent the constraints of the problem. A product specification is said to be *valid* if it is a solution to the *CSP*.

Definition 5 *(Configuration Space or CS)*: A *CS* is a sub-tree of the FT, i.e., contains a single root node and several internal and leaf nodes. Every *CS* contains the root node plus at least one *open decision*, i.e., an optional or grouped feature. A *CS* must enclose either all grouped features of a feature group or none of them. The root of the CS tree is *never* an open decision internal to the space, i.e., it has always been previously instantiated.

Definition 6 *(Junction Node or JN)*: A *JN* is a feature node shared by two or more CSs. Say C_{jk} is the set of CSs containing junction node *jk*. If *jk* is not the root of the *FT*, one and only one *CS* in C_{jk} is the *parent configuration space* and all the others are the *children configuration spaces*. Notice that children configuration spaces are siblings. *jk* is always a leaf node in the parent *CS* and the root node in all children *CSs* in C_{jk}. If *jk* is the root node of the FT, all CSs in C_{jk} are just *siblings*.

Definition 7 *(Configuration Actor or CA)*: A *CA* is a configuration decision-making role that is assigned one or more *CSs*. *CAs* decide whether or not a feature in the CS is to be included in the product specification. The same *CS* can be assigned to multiple *CAs*.

Definition 8 *(Splitting)*: A splitting of a feature model F is a partitioning P containing a set of configuration spaces S such that the union of all $S_i \in S$ corresponds to F, and the intersection of all $S_i \in S$ contains only *JNs*. That is, the splitting breaks down the *FM* in a set of parent-child and/or sibling *CSs* connected by *JNs*.

Definition 9 *(Configuration Space Dependency)*: A configuration space S_1 depends on a configuration space S_2 if an assignment in S_2 automatically instantiates features in S_1 (an instantiated feature has a truth value assigned to it). If only *some* of the features in S_1 are automatically instantiated, we say that *S_1 is weakly-dependent on S_2*. Oppositely, if *all* features in S_1 are instantiated we say that S_1 *is strongly-dependent* on S_2.

Definition 10 *(Configuration Session or CN)*: A *CN* contains one or more non-repeated *CSs* such that feature assignments in all those spaces are made concurrently by the same team of *CAs*. Automatic propagations and

© 2008 ACADEMY PUBLISHER

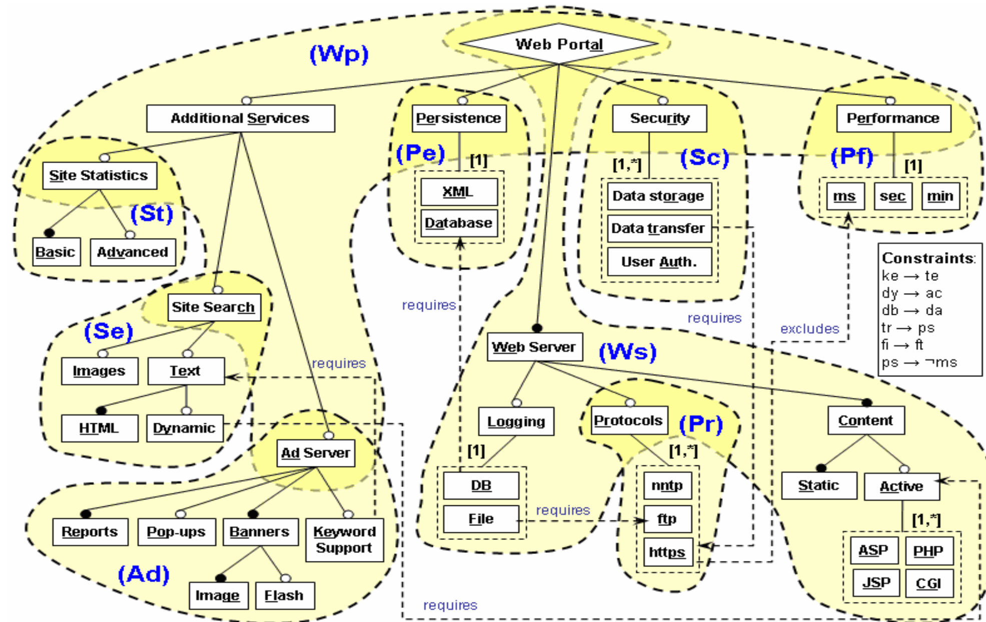

Figure 3. Feature model for a web portal product line decorated with configuration spaces

conflicts are handled by the team. Every *CS* in the feature model must appear in one and only one *CN*.

Definition 11 *(Merging Session or MS)*: A *MS* contains one or more *CSs* for which the actual feature assignments conflict, because they violate one or more global feature constraints. The *MS* is used to resolve the conflict by finding an assignment that satisfies all global constraints. In manual merges humans are responsible for re-assigning truth values to features, whereas in an automatic merge an algorithm is used to find a globally satisfiable assignment.

Definition 12 *(Configuration Plan or CP)*: A *CP* for a split feature model *F* is a *DAG* (directed acyclic graph) that contains single starting and ending nodes and possibly several nodes representing *CNs* and *MSs* encompassing all configuration spaces of *F*. A CP is valid if:

(1) Whenever CSs C_1 and C_2 are such that C_1 is strongly-dependent on C_2 they must either be placed in the same CN, or have their sessions arranged sequentially such that C_2 precedes C_1, and;

(2) Whenever CSs C_1 and C_2 are such that C_1 is weakly-dependent on C_2, they must be placed in the same CN, or have their CNs arranged in sequence, or have their CNs arranged in parallel but immediately followed by a MS to resolve possible conflict decisions in those CSs.

Definition 13 *(Collaborative Product Configuration - Plan Creation Process or CPC-PCP)*: *CPC–PCP* is the process of finding a triple *<F,S,P>* such that *F* is a feature model, *S* is a *valid* splitting of *F*, and *P* is a *valid* plan for configuring *F* according to splitting *S*.

IV. Dependency Analysis & CPC

In the context of product configuration, *dependency analysis* (DA) is the process of reasoning on feature dependencies with the purpose of improving specific aspects of the configuration process. For instance, a domain analyst supported by a DA tool is able to identify implicit undesirable relations among features (e.g. a mutual-exclusion relation between two features) and decide to refactor a feature model by removing the relation or moving features around. In CPC, DA plays a key role in supporting the generation of constraints to validate CPC plans (see Figure 2, scenario-B1, *plan validation constraints*). By analyzing dependencies among features DA supports the automatic identification of strong and weak dependencies among configuration spaces that can be used further to constrain the order and the type of sessions in CPC plans. A particular strategy can be used to map feature dependencies to validation constraints. Constraints derived from strong dependencies should enforce a sequential arrangement of the configuration sessions (*order constraints*). Weak dependencies translate to constraints that require either the sequencing of configuration sessions or the placement of a mandatory merging session immediately after such sessions (*dependency constraints*). Therefore, now we can use DA to identify strong and weak dependencies among configuration spaces and consequently derive validation constraints for CPC plans.

In order to investigate dependencies among features we must provide an efficient algorithm to answer the following question: "*Does feature A depends on feature B?*", i.e., "*Is there any situation in which assigning a particular truth value to feature A forces feature B to assume a particular truth value*?". More generally, "*What features are instantiated and how (which truth value) by an assignment to a particular feature?*" Notice in order to provide an accurate answer to these questions a meticulous analysis of feature dependencies is required.

A possible approach for DA is to use logics. For instance, previous works have discussed the use of *Binary Decision Diagrams* (BDDs) [13] and SAT Solvers [12] as a means to reason on feature models. Such approaches are interesting in the sense that they enable the use of

© 2008 ACADEMY PUBLISHER

well-know techniques and algorithms to work with logic-based structures. However, the use of logics can impose several practical challenges. For instance, the building phase of a BDD can be very costly in terms of space and time demands. In addition, BDDs are very sensitive to factors such as variable ordering and thus can grow exponentially in size on the number of variables in a propositional formula. Unfortunately, finding an optimal variable ordering is an NP-complete problem. Similarly, the use of SAT solvers generally requires translating an arbitrary formula into CNF which is also a time-consuming process. As well, SAT is well-known as the first NP-complete problem to be widely discussed in the literature. To overcome such limitations, we advocate that domain-specific algorithms can be very effective in supporting dependency analysis on feature trees since they exploit particular aspects of the problem space normally invisible to logic-based approaches.

In a later section, we discuss the analysis of configuration space dependencies based on efficient in-place algorithms for identifying weak dependencies on the feature tree in combination with a hypergraph-based technique to tackle the extra constraint space. In the next sections, we describe these algorithms and how they support the generation of order and dependency constraints to validate CPC plans.

TABLE I.
ALGORITHM TO GENERATE ORDER CONSTRAINTS FROM STRONG DEPENDENCIES

```
⌂ pre: M is a non-null feature model
⌂ post: R stores order constraints for conf. spaces of M
gen_order_constraints(FeatureModel:M):
        R:{<node,node,op>}
1.  begin
2.  | R ← {}   ⌂ stores order constraints relations
3.  | P ← conf_spaces(M) ⌂ conf. spaces to process
4.  | for each configuration space C1 ∈ P
5.  | | P ← P - {C1}
6.  | | r ← root(C1)    ⌂ r is a junction point
7.  | | if (¬is_root(r,M) ) ⌂ r is not root of f. model
8.  | | | S ← {C1}  ⌂ stores children conf. spaces
9.  | | | T ← NIL   ⌂ stores the parent conf. space
10. | | | for each conf. space C2 ∈ M, C2 ≠ C1
11. | | | | if ( is_ root(r, C2) )
12. | | | | | S ← S ∪ {C2}
13. | | | | | P ← P - {C2}
14. | | | | else if ( r ∈ C2 )
15. | | | | | T ← C2
16. | | | ⌂ set parent-child relationships
17. | | | for each configuration space C3 in S
18. | | | | R = R ∪ <T, C3, ↦ >
19. end
```

A. Searching Strong Dependencies

TABLE I shows an algorithm named *gen_order_constraints* that traverses the configuration spaces in the feature model, identifies strong (parent-child) relationships among them, and generates a set *R* of order constraints. Constraints are expressed as triples <C_1, C_2, ↦> to indicate that configuration space C_1 *precedes* C_2 whenever they are placed in distinct configuration sessions. Line 2 of the algorithm defines an empty set *R* that will store order constraint relations. Set *P,* in line 3, stores all the configuration spaces of the feature model *M.* In line 4, the configuration spaces of *P* are traversed and removed to indicate they have been processed (line 5). In line 6, the root node *r* of the current configuration space C_1 is recovered to allow skipping the root configuration space since it has no parents and thus does not need to be processed (line 7). Lines 8-9 define two variables *S* and *T*, respectively, a set of children configuration spaces, i.e., that contains *r* as the root node (lines 11-12), and the parent configuration space that contains *r* as a leaf node (lines 14-15). Configuration spaces in *S* are siblings. Lines 17-18 update set *R* with order constraints based on the parent-child relationships found. Finally, line 14 removes children configuration spaces from *P* to indicate that they do not need further processing.

B. Searching Weak Dependencies

One way to tackle the problem of identifying weak dependencies is to use a common logic structure to represent the feature tree and the extra constraints such as BDDs and CNF formulas. However, as discussed this approach involves overcoming serious space and time constraints and dealing with NP-hard algorithms. Hence, we propose a different more-scalable approach to finding weak dependencies that distinguish feature tree relations from extra constraints. The approach takes advantage of the fact that a large number of constraints in a configuration problem is concentrated in the feature tree and the extra constraints usually correspond to a fraction of these relations. As well, since the kinds of relations in the tree are well-known (optional, mandatory, and feature groups) efficient recursive algorithms can be provided.

The approach uses two techniques to examine feature dependencies. For feature tree relations in-place recursive algorithms are used to traverse the tree structure and examine feature dependencies. Hypergraphs and operations such as *merge* and *projection* (as seen in section II) are used to support the analysis of the extra constraints. Even though distinct strategies are used to examine dependencies, a uniform and integrated dependency model is produced that can be used to derive dependency constraints for configuration spaces. It is important to notice that many other techniques can be used to tackle the extra constraints space. Hypergraphs were chosen because they are simple enough not to distract the reader's attention from the main issue of performing efficient dependency analysis in the feature tree.

1) DA algorithms for examining feature tree relations

We propose efficient in-place recursive algorithms to examine feature tree relations starting at a particular feature node. Given a tuple <*f,v*> where *f* is a feature and *v* is a truth value for *f*, the algorithm finds a list of tuples <*node,value*> representing all *features* and the truth *value*

© 2008 ACADEMY PUBLISHER

they can assume if *f* is assigned *v*. That is, the list corresponds to all potential propagations of *f* when *v* is assigned to it. For instance, suppose that feature *f* is a child optional feature of a mandatory feature *g*. The tuple <*f,true*> is part of the dependency list of <g, *true*> because whenever *g* is *true* so are its mandatory children features. Similarly, the tuple <g, *true*> is part of the dependency list of <f, *true*> since parent features are always selected (*true*) whenever one of their descendants is selected. In fact, the relation between features *f* and *g* can be described in propositional logic as $f \leftrightarrow g$ (bi-conditional).

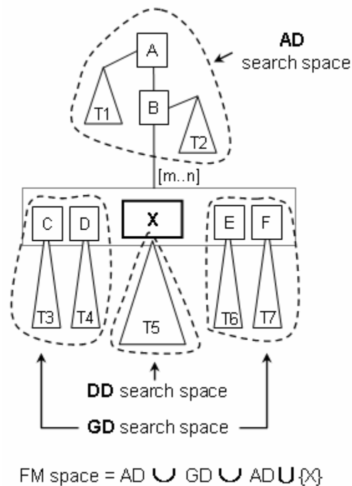

Figure 4. Search spaces for dependency analysis on feature trees

For the sake of simplicity and to make proper use of stepwise refinement in our algorithms, we divided the feature model in three disjoint *dependency search spaces*: *AD*, *DD*, and *GD* and developed specific algorithms to search dependencies in each space. In Figure 4 search spaces *AD*, *DD*, and *GD* for feature *X* are shown. Space *AD* covers the ancestors of feature *X*, and their children (except for the tree rooted by *X* and its siblings C,D,E, and F). The descendants of *X* are represented by space *DD*. Finally, if *X* is a grouped feature (as in the case of the example), space *GD* covers *X*'s siblings in the group and their descendants. Notice that the entire feature model space is obtained by the union of all search spaces and feature *X*. For each combination of search space and truth value, algorithms were developed to find feature dependencies (see TABLE II). In addition, three other convenient operations *D(n)*, *DT(n)* and *DF(n)* were provided.

TABLE II
OPERATIONS TO FIND FEATURE DEPENDENCIES IN FEATURE TREES

Operation	Search Space	Truth value of *n*
D(n)	Entire feature model - {n}	*true* and *false*
DT(n)	Entire feature model - {n}	*true*
DF(n)	Entire feature model - {n}	*false*
ADT(n)	AD	*true*,
ADF(n)	AD	*false*
DDT(n)	DD	*true*
DDF(n)	DD	*false*
GDT(n)	GD	*true*
GDF(n)	GD	*false*

Operation D(*n*) finds a list of tuples <*node, value*> in the feature model that depend on a *true* or *false* assignment to feature *n*. D(*n*) is the union of all tuples found by operations *DT(n)* and *DF(n)*, i.e., *D(n) = DT(n) ∪ DF(n)*. Operations *DT(n)* and *DF(n)* find all tuples <*node, value*> in all dependency search spaces that depend on a *true* and *false* assignment to *x*, respectively. DT(n) combines the results of *ADT(n), DDT(n),* and *GDT(n)*, i.e., *DT(n) = ADT(n) ∪ DDT(n) ∪ GDT(n)*. The same applies to *DF(n)* and support operations *ADF(n), DDF(n)* and *GDF(n).*

```
⌂ Ancestor-space nodes dependent on n:true
ADT(n: Node): R:{<node,value>}
1. begin
2.  R ← {<node,value>}
3.  if (n ≠ NIL)
4.  | p ← parent(n)
5.  | if (p ≠ NIL)
6.  | | R ← R ∪ <p, true>
7.  | | if (is_grouped_node(n) )
8.  | | | R = R ∪ DDT(p, n ∪ sib(n))
9.  | | else
10. | | | R = R ∪ DDT(p, n)
11. | | if (is_grouped_node(p) )
12. | | | R = R ∪ GDT(p)
13. | | R = R ∪ ADT(p)
14. end
```

The *ADT(n)* operation illustrated above searches for dependencies in the *AD* space, that is, the ancestors of feature *n* and their children except for the tree rooted by *n* and its siblings (e.g. features *A* and *B* and sub-trees *T1* and *T2* in Figure 4, for *n = X*). In the algorithm, line 2 creates an empty set *R* of tuples <*node, value*>. Lines 3-6 check if node *n* and its parent node are non-null and adds tuple <*p,true*> to *R* indicating that whenever *n* is *true* so is *p*. Lines 7-10 includes the descendants of *p* in the search space except for the tree rooted by *n* and its siblings (In Figure 4, *X* is *n, B* is *p,* and *T2* is the descendant of *p* also included in the search). Furthermore, if *n* is a grouped node, the siblings of *n* in the group (*sib(n)*) and their descendants are eliminated from the search space (line 8). Lines 11-12 check if *p* is a grouped node and if that is the case the *GDT(p)* operation is called to examine the impact of assigning *true* to *p* on the other grouped features. Finally, line 13 makes a recursive call to *ADT* using *p* as a parameter. All intermediate tuples found are added to set *R* the result of the dependency analysis for feature *n*.

The *DDT(n, E)* operation searches for dependencies in the *DD* space, that is, the descendants of feature *n* except for nodes in set *E* (In Figure 4, it corresponds to sub-tree *T5* for *n = X*). In the algorithm, line 2 creates an empty set *R* of tuples <*node, value*>. For each mandatory child *c* of *n* a tuple <c, *true*> is added to set *R* to indicate than whenever *n* is *true* so are its mandatory feature children. If *c* is part of a group it means that, whenever the lower bound of the group cardinality is greater than zero, a *true*

© 2008 ACADEMY PUBLISHER

assignment to *n* requires some of its grouped-node children to be *true* as well (lines 9-13). Lines 8 and 14 recursively call *DDT(n)* for descendants of *n*. Suppose node *B* in Figure 4 is the feature being analyzed. If B is set to *true*, feature *X* would be included in the DD search space as a descendant of *B* and because *X* is grouped, if the lower bound of the group cardinality *m* is greater than zero, tuples for nodes *C, D, E,* and *F* containing the truth value *true* are added to *R* to indicate that if B is set to *true* some of these nodes may also be set to *true* (at least *m* nodes to be more precise). All intermediate tuples found are added to set *R* the result of the dependency analysis for feature *n*.

```
⌂ Descendant-space nodes dependent on n:true
DDT(n:Node, E:{Node}) R: {<node, value>}
1. begin
2.  R ← {}
3.  if ( n≠ NIL)
4.  | for each c ∈ children(n)
5.  | | if ( c ∉ E )
6.  | | | if (mandatory(c))
7.  | | | | R ← R ∪ <c, true>
8.  | | | | R ← R ∪ DDT(c, E)
9.  | | | else if (is_grouped_node(c) )
10. | | | | g ← group(x)
11. | | | | if min(g) > 0
12. | | | | | for each g ∈ sib(c)
13. | | | | | | R ← R ∪ <g, true>
14. | | | | | | R ← R ∪ DDT(g, NIL)
15. end
```

```
⌂ Group-space nodes dependent on n:true
GDT( n:Node ) R: {<node, value>}
1. begin
2.  R ← {}
3.  if ( n≠ NIL ∧ is_grouped(n))
4.  | g ← group(n)
5.  | if max(x) < count_nodes(g)
6.  | | for each s ∈ g
7.  | | | if ( s ≠ n )
8.  | | | | R ← R ∪ <s, false>
9.  | | | | R ← R ∪ DDF(s, NIL)
10. end
```

The *GDT(n)* operation searches for dependencies in the *GD* space, that is, the nodes that are grouped together with feature *n*, and their children. In the algorithm, line 2 creates an empty set *R* of tuples <*node, value*>. Line 3 checks if *n* is non-null and is a grouped node. If that is the case, for each sibling *s* of *n* in the group a tuple <s, *false*> is added to *R* whenever the upper bound of the group cardinality is lower than the number of features in the group. In fact, whenever it is the case, assigning *true* to *n* may imply falsifying other grouped features in order to abide by the cardinality constraint. Line 9, calls function DDF in order to include the descendants of each grouped node (except for *n*) in the search space. In Figure 4, if *X* is set to *true* and the upper bound of the group cardinality *n* is lower than 5 (the total number of grouped features) tuples for nodes *C,D,E,* and *F* containing a *false* assignment are included in *R.* All intermediate tuples found are added to set *R* that is the result of the dependency analysis for feature *n*.

For brevity, the remaining operations *ADF(n), DDF(n, E)* and *GDF(n)* will not be discussed but pseudo code follows. Such operations share similar logic with the algorithms described thus making the understanding of their structure straightforward.

```
⌂ Descendant-space nodes dependent on n:false
DDF( n:Node, E:{Node} ) R: {<node, value>}
1. begin
2.  R ← {}
3.  if ( n ≠ NIL)
4.  | for each c ∈ children(n)
5.  | | if ( c ∉ E )
6.  | | | R ← R ∪ <c, false>
7.  | | | R ← R ∪ DDF(c, E)
8. end
```

```
⌂ Group-space nodes dependent on n:false
GDF( n:Node ) R: {<node, value>}
1. begin
2.  R ← {}
3.  if ( n ≠ NIL ∧ is_grouped(n))
4.  | g = group(n)
5.  | if (min(g) > 0)
6.  | | for each s ∈ g
7.  | | | if ( s ≠ n )
8.  | | | | R ← R ∪ <s, true>
9.  | | | | R ← R ∪ DDT(s, NIL)
10. end
```

```
⌂ Ancestor-space nodes dependent on n:false
ADF( n : Node ) R: {<node, value>}
1. begin
2.  R ← {}
3.  if ( n≠ NIL)
4.  | p = parent(n)
5.  | if (p ≠ NIL)
6.  | | if (mandatory(n) ∨
7.  | | |    (is_grouped (n) ∧ min(group(n)) > 0)
8.  | | | R ← R ∪ <p, false>
9.  | | | if (is_grouped(n) )
10. | | | | R ← R ∪ DDF(p, n ∪ sib(n))
11. | | | else
12. | | | | R ← R ∪ DDF(p, n)
13. | | | if (is_grouped(p) )
14. | | | | R ← R ∪ GDF(p)
15. | | R ← R ∪ ADF(p)
16. end
```

2) DA algorithm for deriving dependency constraints from weak dependencies

An algorithm to identify weak dependencies in feature models called *gen_dependency_constraints(M)* is depicted in TABLE III. The algorithm combines distinct techniques to examine feature dependencies. Hypergraph-

© 2008 ACADEMY PUBLISHER

based operations support extra constraint analysis while the algorithms shown in the previous section cover the feature tree space. The algorithm produces a single consistent output that consists of a complete list of weak dependencies for the features of a feature model. Dependencies are represented by tuples *<node, node, op>* where *node* indicates feature nodes and *op* (represented by the symbol ≈) a weak dependency relation between the features. The tuples are further used to derive weak dependencies among feature model configuration spaces.

TABLE III.
ALGORITHM TO GENERATE DEPENDENCY CONSTRAINTS FROM WEAK DEPENDENCIES

⌂ *pre: M is a feature model with conf. spaces*
⌂ *post: R contains dependency constraints for M*
gen_dependency_constraints(FeatureModel: M):
    R: {<node,node,op>}
1. begin
2. ⌂ *builds hypergraph hc based on*
3. ⌂ *M's extra constraints*
4. $H_D \leftarrow$ (V={},E={})
5. for each c ∈ extra_constraints(M)
6. │ H_D = merge_hg(H_D, hg_from_constraint(c))
7. ⌂ *expands* H_D *based on feature tree relations*
8. H_D = expand_constraint_hg(H_D, tree(M))
9. ⌂ *build conf. space hypergraph*
10 H_C = build_conf_space_hypergraph(H_D, M)
11. ⌂ *projection:* $H_D \; x \; H_C \rightarrow H_F$
12. H_F = project_hg(H_D, H_C)
13. add_parent_child_relations(H_F)
14. ⌂ *generate dependency constraints in R*
15. ⌂ *based on* H_F*'s hypedeges*
16. R ← {}
17. for each H ∈ hyperedge(H_F)
18. │ R ← R ∪ <H, ≈ >
19. end

The web portal feature model in Figure 3 will be used to illustrate the algorithm. Lines 4-6 create and update a hypergraph H_D that represents dependencies among features found in the extra constraints. Operation *hg_from_constraint(c)* converts each extra constraint of the feature model into a hypergraph as shown in section II. Following this step, constraint hypergraphs are merged with hypergraph H_D so that hyperdeges sharing a common vertex are combined (set union). Notice that feature *ps* in the web portal feature model appears in two constraints *(tr → ps)* and *(ps → ¬ms)* and thus the variables of such constraints are combined in a single hyperedge of H_D (in bold below).

H_D={V={ke ,te, dy, ac, db, da, tr, ps, fi, ft ,ps, ms},
*E={**{ps, tr, ms}**{ke ,te},{dy, ac},{db, da},{fi, ft}}}*

The hyperedges of H_D now represent dependencies among the features of the web portal product line. However, the DA process is not complete as the relations in the feature tree were not taken into account. The next step in the algorithm (line 8), expands hypergraph H_D based on feature tree relations. For each unique variable (representing features) in extra constraints a call to operation *D(n) = DT(n) ∪ DF(n)* is made. Whenever the result of the call contains a variable that can be found in the extra constraint a new relation is derived. For the web portal feature model the following calls are performed:

*(1) **DT**(fi) = {<db, false>}, **DF**(fi) = {<db, true>}*
*(2) **DT**(db) = {<fi, false>}, **DF**(db) = {<fi, true>}*
*(3) **DT**(te) = {}, **DF**(te) = {<dy, false>}*
*(4) **DT**(dy) = {<te, true>}, **DF**(dy) = {}*
*(5) **DT**(ft) = {}, **DF**(ft) = {<ps, true>}*
*(6) **DT**(ps) = {}, **DF**(ps) = {<ft, true>}*

The new relations found are {*fi, db*} (from (1) and (2)), {*te, dy*} (from (3) and (4), and {*ft, ps*} (from (5) and (6)). Now, those relations sets are merged with the H_D hyperedges.

H_D={V={ke, te, dy, ac, db, da, tr, ps, fi, ft, ps ,ms}
E={A:{ke, te, dy, ac}, B:{db, da, fi, ft, tr, ps, ms}}}

Hyperedges labeled *A* and *B* represent two independent sets consisting of interdependent features. In fact, since there is no constraint that links these sets they are completely independent from each other.

The next step requires the creation of a hypergraph H_S to represent the configuration spaces depicted in Figure 3. H_S contains the same vertices as H_D but its hyperedges group vertices according to the configuration spaces that enclose them. For instance, features *te* and *dy* are enclosed by configuration space *Se*, thus giving rise to the labeled hyperedge set *Se:*{*te*, *dy*}. The final configuration for hypergraph H_S is:

H_S ={ V={ke, te, dy, ac, db, da, tr, ps, fi, ft, ps, ms},
E={ Ws:{db, fi, st, ac }, Se:{te, dy}, Ad:{ke},
Pr:{ft, ps}, Pe:{da}, Sc:{tr},Pf:{ms}} }

Hypergraphs H_D and H_S have identical vertex sets. Moreover, because H_D represents feature dependencies and H_S represents configuration spaces, a projection ($H_D \; x \; H_S \rightarrow H_F$) would provide an answer to the question: *Which configuration spaces depend on each other?* In line 12, such a projection is performed thus giving rise to hypergraph H_F.

H_F = {V={Ws,Pe,Pr,Se,Ad,Sc,Pf}
E={{Se,Ad,Ws),{ {Ws,Pe,Pr,,Sc,Pf}}}

The final update to hypergraph H_F comes from the observation that whenever two configuration spaces depend on each other, they also depend on each other's parent/children spaces. Thus, those spaces must be included in H_F as follows (in bold):

H_F = {V={Ws, Pe, Pr, Se, Ad, Sc, Pf}
*E={{**Wp,** Se, Ad, Ws),{ {**Wp**, Ws, Pe, Pr, Sc, Pf}}}*

Figure 5 shows a merged view of the strong and weak dependencies of the configuration spaces of the web portal product line. Notice that strong dependencies

© 2008 ACADEMY PUBLISHER

always prevail over weak dependencies whenever they happen in the same configuration space.

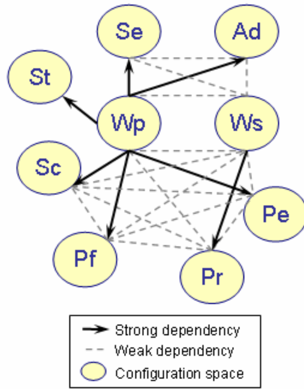

Figure 5. Strong and weak dependencies merged view

One possible alternative to translate configuration space dependencies into constraints to validate CPC plan is to use logic formulas as follows:

(1)(*Session(Wp) = Session(Se)* ∨
O(Session(Wp)) < O(Session(Se)))

(3) (*(Session(Se) = Session(Ad))* ∨
(O(Session(Se)) < O(Session(Ad))) ∨
(O(Session(Ad)) < O(Session(Se))) ∨
(O(Session(Se)) = O(Session(Ad)) ∧
(∃Merge(Ad,Se)
· O(Merge(Ad, Se)) = O(Session(Ad))+1)))

Operation *O(S)* returns the placement order of the configuration or merging session *S* in the CPC plan. Formula (1) reads: *Configuration spaces Wp and Se must be part of the same configuration session OR* the *configuration session of configuration space Wp must precede the configuration session of configuration space Se*, since *Se* strongly depends on *Wp*. Formula (2) was derived from a weak dependency between configuration spaces *Ad* and *Se* and reads: *Configuration spaces Ad and Se must be part of the same configuration session OR the configuration session of configuration space Ad must precede the configuration session of configuration space Se OR the configuration session of configuration space Se must precede the configuration session of configuration space Ad OR they must be configured in concurrent sessions but a merging session involving Ad and Se must follow the configuration sessions immediately*.

V. Performance Analysis

This section examines algorithmic complexity for the dependency analysis operations introduced in section IV. In additional, the results of a preliminary experiment to evaluate the performance of algorithms to generate BDDs for feature models are shown.

A. Algorithmic Complexity

The algorithm *gen_order_constraints(M)* contains three loops in lines 4, 10, and 17. All other lines consist of operations that can be computed in constant time. The loop in line 10 iterates over all configuration spaces of the feature model. Hence, its complexity is $O(|S|)$, where $|S|$ is the total number of configuration spaces decorating the feature model. In line 17, some of these configuration spaces are traversed again to derive order constraints. This keeps the added running-time of both loops (lines 10 and 17) proportional to $O(|S|)$. The loop in line 4 also traverses available configuration spaces but as they are processed, other configuration spaces found are removed from the yet-to-process list *P*. In fact, for each configuration space in *P* all its siblings are processed simultaneously and removed from *P*. Thus, we conclude that the loop runs *p* times, where *p* is the number of parent configuration spaces in the feature model. Thus the algorithm can compute order constraints in $O(p·|S|)$ time, where *p* is the total number of parent configuration spaces, i.e., have at least one child, and *s* is the number of configuration spaces of the feature model. In the worst-case, characterized by a sequential top-down arrangement of the configuration spaces, the algorithm performs in $O(|S|^2)$, i.e., $p = |S|$. Therefore, the algorithm can be highly efficient even for very large feature models as the number of configuration spaces is usually a fraction of the number of features.

The running-time of algorithms *ADT(n), DDT(n), GDT(n), ADF(n), DDF(n), and GDF(n)* depends on the kinds of feature relations described in the feature model (e.g., optional, mandatory, groups) and is especially sensitive to the feature picked for analysis. For instance, if feature *X* in Figure 4 is a top-level feature, i.e., near to the tree root node, when *DDF(n)* is performed on *X* a potentially high number of features will be found in the search (in fact, all features in sub-tree *T5*). However,, if *DDT(n)* is applied to *X*, a short list of dependent nodes might be expected. Notice that if *X* is a grouped feature (as it is the case in Figure 4) it becomes hard to predict the path of the search as it depends highly on the cardinality upper and lower bounds of *X's* group and the number of ancestors and descendants of *X*'s grouped siblings.

Despite the unpredictability of the search path, calculating the combined worst-case scenario for those algorithms is straightforward. Because search spaces *AD*, *DD* and *GD* are disjoint, in the worst-case all features in these spaces are traversed, i.e., a total of $|AD| + |DD| + |GD|$ features. Therefore, we can conclude that the worst-case running time for operations *DT(n)* and *DF(n)* is $O(|F|-1)$, where $|F|$ is the total number of features in the feature model. Additionally, the worst-case time for *D(n)* is $O(2·(|F|-1))$ as it would have to traverse twice the entire feature model (one for *DT(n)* and again for *DF(n)*).

The operation *gen_dependency_constraints(M)* makes use of different dependency analysis techniques to examine configuration space dependencies. More specifically, we proposed the use of distinct (yet integrated) systems to analyze dependencies in the extra constraints and in the feature tree. For instance, operations *DT(n)* and *DF(n)* were used to exploit feature tree relations, and expand hypergraphs representing

© 2008 ACADEMY PUBLISHER

dependencies among extra constraint variables. As argued earlier, other data structures and algorithms can be used to analyze dependencies on arbitrary constraints such as BDDs or propositional formulas. However, one important contribution of our work is the delivery of efficient in-place algorithms to analyze dependencies in the feature tree that can be easily integrated with specific techniques to analyze the extra constraints. By doing so, we also significantly improved the efficiency and scalability of dependency analysis on feature models since the size of the problem is now proportional to the size of the extra constraints.

B. Tools and Running-times

In addition to algorithmic analysis, we conducted a preliminary experiment to evaluate the performance of algorithms to generate data structures to support the dependency analysis of feature models. For the purpose of the experiment, two support tools were developed. The *GenFM* tool generates random feature models based on a set of input parameters that control the number and types of features to be available in the feature tree. The generation of optional and mandatory features as well as feature groups with cardinality [1] and [1,*] was possible. The *FM2BDD* tool uses the JavaBDD library to translate feature trees into equivalent BDD structures. Both tools are available at [14]. The main goal of the experiment was to assess space and time requirements during the building phase of a BDD. Details about the resources used in the experiments and the results obtained are shown next.

Equipment: AMD Turion 64 x2, 1.61GHz, 960Mb RAM with Physical address extension
OS: Windows XP, SP2
Runtime Environment: Java JSE 1.6.0_02-b06
Library: JavaBDD, 1 million initial nodes, 1 million cache size
Command line: java –mx512m startup.FM2BB fm.xml
Other details: Feature tree nodes were visited in depth-first search to define the BDD variable order.

Four random feature trees (FT) were generated for the experiments containing 100, 500, 1000, and 2000 nodes. The tree included a balanced number of mandatory, optional and grouped features. The size and building time of the BDD for a 100-feature feature model was 200ms and 375 nodes, respectively. For feature models containing 500, 1000, and 2000 features the BDD size and building times were [400ms, 2810 nodes], [1.2sec, 5981 nodes], and [5.5secs, 15797 nodes], respectively. For feature trees containing more than 2500 features an *OutofMemoryException* exception was raised. This observation suggested that even though the ratio *BDD nodes/FT nodes* was not excessively high as we expected the amount of memory required to store and manipulate BDD nodes was considerably higher than that for feature trees.

At the same time, a feature model with 10,000 features was generated and the time to perform operation *DF(root)* remained under 20ms. Notice that *DF(root)* is an expensive operation since it needs to traverse the entire feature tree. *DT(n)* was also validated in many scenarios involving mandatory, optional and grouped features and never took more than 5 ms to complete. Since the running-time of such operations remained low for very large feature models it seems to be reasonable to think of using BDDs as a means to perform DA on extra constraints as the problem now is considerably reduced.

Finally, to make possible the assessment of operations such as *DF(n)* and *DT(n)* a tool called *CPC* was developed and is available at [14]. More importantly, the *CPC* tool implemented all the algorithms presented in this paper including the manipulation of hypergraphs. In fact, the tool was crucial to validate the major components of the CPC approach. In the next section, the web portal PL (see Figure 3) is used to illustrate the approach.

VI. ILLUSTRATED EXAMPLE

This section illustrates phase-1 of the CPC approach by taking the web portal product line depicted in Figure 3 as basis. Notice that the figure shows an expanded version of the feature model in Figure 1 in which constraint (*or* → *da*) was removed and several others were included.

A. Splitting

The product manager role is responsible for the splitting phase since she has an overview of the stakeholders and their expertise. In addition, the product manager can anticipate potential conflict situations and try to avoid them. Figure 3, shows a possible splitting for the web portal product line. Nine configuration spaces are depicted: *Wp, St, Pe, Sc, Pf, Se, Ad, Ws,* and *Pr.* Each configuration space groups features based on a particular criterion (e.g. knowledge domain). Notice that some features appear in more than one configuration space (e.g. *Ad Server, Protocols*) and are called *junction points*. Because a junction point is also a feature it needs to be assigned to a configuration actor. In fact, only a single configuration space contains this feature as a leaf node and this space represents the place in which the feature will be chosen. For instance, feature *pr* (*Protocols*) will be chosen in configuration space *Ws* rather than in configuration space *Pr.*

Notice that configuration spaces can be viewed as clusters of the feature model and their arrangement respect the hierarchy of features in the feature model. Hence, the concept of parent and children configuration spaces is applicable (e.g. configuration space *Wp* is parent of configuration space *St*). Two kinds of configuration space dependencies are relevant: *strong* and *weak* dependencies. A configuration space *A* is *strongly dependent on* a configuration space *B* when a single decision in *A* can impact all decisions in *B.* It can be easily observed that children configuration spaces are always strongly dependent or their parent spaces. For instance, child space *St* is strongly dependent on parent space *Wp* because when feature *si* (*Site Statistics*) is set to false all features in *St* are automatically falsified. Two

© 2008 ACADEMY PUBLISHER

configuration spaces *A* and *B* are *weakly dependent* if some decisions in *A* can impact some decisions in *B*, and vice-versa (e.g. *Ws* and *Pe* because of constraint $db \rightarrow da$). Weak dependencies are directly related to the extra constraints attached to the feature model.

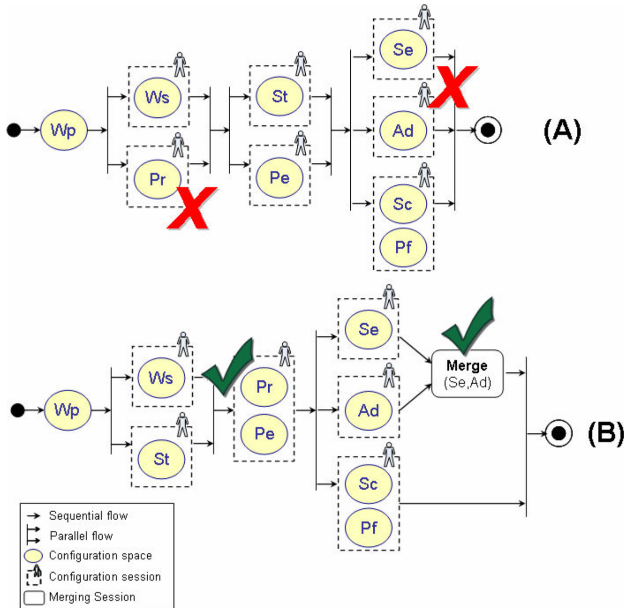

Figure 6. Invalid (A) and valid (B) CPC plans for the web portal product line

Figure 6 shows a merged view of the strong and weak dependencies among the configuration spaces of the web portal product line. Arrows indicate strong dependencies and weak dependencies are represented by dashed lines. For instance, configuration spaces *St, Se, Ad, Pe, Pf* and *Sc* strongly depend on configuration space *Wp*. In fact, Wp is the parent space of those configuration spaces.

B. Plan Creation

Once the feature model is split into several hierarchical configuration spaces a plan is specified to group configuration spaces in configuration sessions and to arrange the sessions in sequential and parallel flows. Notice that invalid plans are possible thus validation rules are required to enforce the correctness of plans. An invalid plan is one that leads to invalid product specifications, i.e., that contains incompatible features. To validate plans we will consider the following rules:

(1) Whenever a configuration space B is strongly dependent on a configuration space A, A must precede B

(2) If two configuration spaces A and B are weakly dependent they are to be arranged either in a sequence or in parallel but immediately followed by a merging session

It is important to notice that rules (1) and (2) only apply to configuration spaces placed in different configuration sessions. That is, configuration spaces of the same session are configured by the same team of configuration actors and eventual dependencies among them are resolved together during the session.

The CPC plan is a workflow-like structure that groups configuration spaces into configuration sessions and arranges configuration sessions in sequence or parallel. Merging sessions follow dependent configuration sessions, i.e., configuration sessions that contain interdependent configuration spaces. Figure 6 depicts two plans A and B for the web portal product line based on the splitting shown in Figure 3. Plan A is invalid because configuration spaces Ws and Pr are placed in parallel configuration sessions yet Pr is strongly dependent on Ws. Similarly, configuration spaces Se and Ad are also placed in parallel sessions but because they are weakly dependent on each other, a merging session is required to enforce that eventual decision conflicts in those spaces will be resolved.

Plan *B* fixes the problems of Plan *A* by moving configuration space *Pr* to a new configuration session that follows *Ws's* configuration session. Similarly, a merging session was added immediately after the configuration sessions of configuration spaces *Se* and *Ad*. Finally, note that configuration space *St* was moved to the same configuration session as configuration space *Ws* for optimization purposes since those spaces exhibit no dependencies on each other. The same optimization strategy could have been applied to configuration spaces *Ws* and *Wp*. It is up to the product manager to accept or reject optimizations.

C. Plan Generation

The last step prior to the actual product configuration process is to generate an *executable* representation for the CPC plan. Notice that plan *B* in Figure 6 is indeed a compact representation for the collaborative configuration process since many configuration sessions are in fact optional as they depend on decisions made on previous sessions. For instance, even though configuration space *St* follows configuration space *Wp*, *St's* configuration session will only be executed if feature *si* (*Site Statistics*) is selected during *Wp's* configuration session. In fact, prior to the execution of any configuration session the underlying workflow system needs to check whether at least one root feature of one configuration space in the session is *true*, otherwise the session is skipped. We say the CPC plan represents a *pessimistic view* of the collaboration process in which all sessions are executed and decision conflicts arise. In the actual configuration process, many configuration and merge sessions may be skipped as a consequence of previous decisions. We refer to the *expanded CPC workflow* as the actual executable workflow representation used to augment a CPC plan.

VII. RELATED WORK

In the last years, many attempts to improve product configuration have been made. However, there is still a significant gap between what is observed in practice and the kinds of techniques being developed. Generally, we argue that collaboration has not been tackled explicitly which makes configuration a difficult problem for humans to cope with as a team.

© 2008 ACADEMY PUBLISHER

A. Product Configuration as Constraint Satisfaction

Product configuration has also been addressed as a Constraint Satisfaction Problem (CSP) [15][16] in which configuration knowledge is described in terms of a component-port representation [17] that includes a set of constraints to restrict possible combinations of components. Constraints are usually written in formal notation (e.g., propositional logic). Similarly, user requirements are translated to a formal representation allowing the configuration problem to be solved by automated systems known as configurators. Alternative versions of the CSP approach support the notion of distributed configuration [17]. In distributed configuration, the configuration problem is translated into a distributed constraint satisfiability problem (DisCSP) [18] in which the constraints and variables are fragmented over multiple configuration environments. Each environment is controlled by an intelligent software agent that works as a local configuration system. DisCSP approaches build on distributed algorithms to support agent communication (e.g., message passing mechanisms) and coordination (e.g., enforcement of local and global constraints).

CSP and DisCSP focus on developing algorithmic support for solving constraint satisfaction problems. The assumption is that machines can quickly process thousands of instructions and perform efficient backtracking until a desirable solution is found. The involvement of humans in the process is limited to providing requirements to the configuration system in terms of logic formulas. Instead, in our approach the major goal is to support the coordination of human decision-making in product configuration. Tool support is provided not as a means to solve the problem but to provide assistance for humans to carry out their job.

B. Staged Configuration

Staged configuration [19] was an initial starting point for our work as it pinpointed various scenarios in which product configuration is carried out by multiple configuration actors in different stages. The authors introduced two configuration techniques called specialization and multi-level configuration to support the progressive configuration of products. The CPC approach relates to the notion of staged-configuration in two ways. First, it advances the discussion on coordination of configuration actors in collaborative configuration. Second, it provides effective tool support based on efficient algorithms for dependency analysis and formalizes the concepts in the approach.

VIII. CONCLUSION

Collaborative configuration is critical to the success of SPLs since in practice products are configured by several parties with possibly different interests and background. However, current configuration technology is essentially single-user-based in which a single role interprets and translates user requirements into configuration decisions. This process is error-prone and time-consuming since it may require several interactions between the product manager and the stakeholders to check whether the features selected for the product are indeed desirable. In this sense, stakeholders participate passively in the configuration process since they do not deal with configuration decisions directly.

We propose and formalize an approach that promotes stakeholders to *active* participants of the configuration process called *Collaborative Product Configuration* (CPC). The approach allows the product manager to describe the configuration process in terms of a workflow-like representation called *CPC plan*. The plan is used to support coordinating stakeholders in the product configuration decision-making. Because plans may be incorrect leading to inconsistent product specifications, constraints to validate plans were derived supported by efficient dependency analysis algorithms. It was shown that the algorithms can perform well even when very large feature models are considered. Finally, an illustration of the CPC approach in the context of a web portal product line was provided. We expect the approach to pave the way for a deeper understanding of how collaborative configuration can be properly supported.

Future research directions include the development of the execution environment for CPC plans, the conduct of larger case-studies, and the development of efficient algorithms to support feature model reasoning in the context of CPC.

ACKNOWLEDGEMENTS

The authors would like to thank the Computer Systems Group at the University of Waterloo for the financial support to this research work.

REFERENCES

[1] Software Engineering Institute, Software Product Lines, Link: http://www.sei.cmu.edu/productlines/

[2] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson: Feature-oriented domain analysis (FODA) feasibility study, SEI, CMU, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-21, Nov. 1990.

[3] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger. XML-based Feature Modelling. LNCS, Software Reuse: Methods, Techniques and Tools: 8th ICSR 2004. Proceedings, 3107:101–114, 2004.

[4] K. Kang, K. Lee, and J. Lee. FOPLE - Feature Oriented Product Line Software Engineering: Principles and Guidelines. Pohang University of Science and Technology, 2002

[5] K. Czarnecki and U.W. Eisenecker. Generative Programming. Addison Wesley, 2000. ISBN: 0201309777.

[6] C. Krueger. BigLever GEARS tool, BigLever Software Inc., link: http://www.biglever.com/extras/ Gears_data_sheet.pdf

[7] Pure-systems GmbH. Variant Management with Pure::Consul. Technical White Paper. Link: http://web.pure-systems.com, 2003.

© 2008 ACADEMY PUBLISHER

[8] M. Antkiewicz and K. Czarnecki, K. FeaturePlugin: Feature modeling plug-in for Eclipse. In: OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop. (2004) Link: http://www.swen.uwaterloo.ca/kczarnec/ etx04.pdf. Software available from gp.uwaterloo.ca/fmp.

[9] M. Mendonca, D. D. Cowan, T. Oliveira, A Process-Centric Approach for Coordinating Product Configuration Decisions, HICSS, p. 283a, 2007.

[10] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. 2007. Generic semantics of feature diagrams. Computer Networks 51, 2 (Feb. 2007), 456-479.

[11] K. Czarnecki, A. Wasowski: Feature Diagrams and Logics: There and Back Again. SPLC 2007, Kyoto, Japan.

[12] D. Batory. Feature Models, Grammars, and Propositional Formulas. SPLC 2005, Rennes, France.

[13] D. Benavides, S. Segura, P. Trinidad and A.Ruiz-Cortés. First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS): FAMA: Tooling a Framework for the Automated Analysis of Feature Models. January 2007.

[14] M. Mendonca, Tools for Dependency Analysis in Collaborative Product Configuration, 2007. Available at: *http://ww.csg.uwaterloo.ca/~marcilio/dacpc/index.html*

[15] E.P.K. Tsang. Foundations of Constraint Satisfaction. Academic Press, London and San Diego, 1993 ISBN 0-12-701610-4.

[16] V. Kumar. Algorithms for Constraint Satisfaction Problems: a Survey. AI Msg. 13 (1) (1992) 32-44.

[17] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. Towards Distributed Configuration. Proc. KI-2001, Joint German/Austrian Conference on AI, Vienna, Austria, Lecture Notes in AI, Springer Verlag.

[18] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. IEEE Transactions on Knowledge and Data Engineering, v.10 n.5, p.673-685, September 1998.

[19] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration through Specialization and Multi-level Configuration of Feature Models. Software Process Improvement and Practice, 10(2), 2005.

Marcilio Mendonca received the MSc degree in computer science (1996) from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil. He is currently a PhD student at the David R. Cheriton School of Computer Science at the University of Waterloo in Canada. He has recently received the prestigious Cheriton Scholarship Award reserved for the school's very top graduate students. Prior to the PhD he worked for 8 years in industry as a software architect, project manager, and university lecturer.

Donald Cowan is Distinguished Professor Emeritus in the School of Computer Science at the University of Waterloo. He was the founding Chair of Computer Science at the University of Waterloo and is currently Director of the Computer Systems Group at the same University. His current research interests include software engineering, software tools, Web-based systems for asset management and social networking, software processes, and hypermedia documentation. Dr. Cowan is the designer of forty unique Web-based information portals.

William Malyk is currently a PhD student at the David R. Cheriton School of Computer Science at the University of Waterloo where he previously received a MMath degree in computer science in 2006. His research focuses on software engineering, workflow technology, and health informatics.

Toacy Oliveira Toacy C. Oliveira received the BSc degree in electrical engineering (1991) and the MSc (1997) and PhD (2001) degrees in computer science from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil. He also spent two years as a postdoctoral fellow at the University of Waterloo, Canada. He is currently professor at the University of Liverpool, UK and at Pontifical Catholic University of Rio Grande do Sul, Brazil. Dr. Oliveira has participated in several projects in cooperation with industry and worked as a consultant to the United Nations Development Programs (UNDP). His current research interests include software design, software processes and tools.

© 2008 ACADEMY PUBLISHER