LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

MANAGING MULTIPLE SOFTWARE PRODUCT LINES USING MERGING TECHNIQUES

Mathieu Acher, Philippe Collet, Philippe Lahire, Robert France

Equipe MODALIS

Rapport de recherche
ISRN I3S/RR–2010-06–FR

Mai 2010

Laboratoire d'Informatique de Signaux et Systèmes de Sophia Antipolis - UNSA-CNRS
2000, rte.des Lucioles – Les Algorithmes – Bât Euclide B – B.P. 121 – 06903 Sophia-Antipolis Cedex – France
Tél.: 33 (0)4 92 94 27 01 – Fax: 33 (0)4 92 94 28 98 – www.i3s.unice.fr
UMR6070

Managing Multiple Software Product Lines Using Merging Techniques

Mathieu Acher[1], Philippe Collet[1], Philippe Lahire[1], and Robert France[2]

[1] University of Nice Sophia Antipolis, I3S Laboratory (CNRS UMR 6070)
{acher,collet,lahire}@i3s.unice.fr
[2] Colorado State University, Computer Science Department
france@cs.colostate.edu

Abstract. A growing number of organizations produce and maintain multiple Software Product Lines (SPLs) or design software products that utilize features in SPLs maintained by competing suppliers. Manually building monolithic Feature Models (FMs) to help manage features described across different SPLs is error-prone and tedious and the resulting FMs can be difficult to understand and use. In this paper we propose a compositional approach to managing multiple SPLs that involves automatically merging FMs defined across the SPLs. We illustrate how the approach can be used to create FMs that support selection of products from among sets of competing products provided by different companies or suppliers. The merging techniques can also manage features from different SPLs which are then combined to form products. We show that the proposed approach results in more compact FMs, and we provide some empirical results on the complexity and scalability of the composition operators used in the approach.

1 Introduction

Software Product Line (SPL) engineering is concerned with systematically reusing development assets in an application domain [1, 2]. Products of an SPL are distinguished by *features*, where a feature is a domain abstraction relevant to stakeholders. Feature Models (FMs) [3, 4, 5] can be used in SPL engineering to compactly represent product commonalities and variabilities in terms of optional, alternative and mandatory features. An FM can be viewed as a characterization of all valid combinations of features in an SPL.

In some SPL environments, support for manipulating multiple SPLs may be needed. For example, in the consumer electronics domain, the reuse of software components from different SPLs is commonplace [6]. Some of these SPLs may be developed and maintained by external suppliers, some of which may compete to deliver similar products. The same observation can be made in the semiconductor industry where a set of hardware components from several suppliers has to be integrated into a product [7]. There is thus a need for SPL engineering approaches that support defining and managing variability across different SPLs [2, 8]. Managing variabilities across multiple SPLs is especially challenging when the SPLs are owned by different companies [2, 7, 9]. Support for composing multiple FMs can help domain engineers produce coherent characterizations of valid combinations of features taken from multiple SPLs. Product (application) engineers also need support for producing valid product configurations that belong to one or several SPLs. There is thus a need to determine which SPLs are

able to provide a specific (combination of) feature(s) or not. Product engineers should also be able to derive products using features from different SPLs and suppliers.

Manually creating FMs to support the above activities can be a cumbersome and error-prone task. Recently proposed techniques support automatic creation of FMs that integrate features from multiple SPLs [9, 10]. They provide operational solutions in which the resulting FM is complemented with some constraints or references to manage the relations between, possibly competing, input SPLs. In this paper we provide some evidence that these techniques are not scalable and that the FMs they produce are difficult to understand and use. The goal of our research in this area is to provide support for managing large, multiple SPLs. We propose an incremental, scalable technique for automatically producing compact FMs that integrate features from multiple SPLs. The technique uses *merge* operations to compose FMs defined in different SPLs.

In Section 2 we define key terms and motivate our work. In particular, we distinguish *competing* from *compositional* multiple SPLs. We then show in Section 3 how the management of multiple SPLs based on FMs can be defined in terms of sets of configurations. In Section 4, we give an overview of merge operators [11] used to compose FMs and provide a sound formal basis for the operators in Boolean logic. We identify interesting properties of the composition technique as it notably allows independent reasoning and incremental evolution of multiple SPLs. We also show how to efficiently extract configurations that lead to a single-source supplier. In Section 5, we present some empirical results we performed on a large number of different FMs to evaluate the merging techniques in terms of scalability. Finally, we discuss and compare our approach in Section 5.2 and explain how the technique overcomes the limitations of earlier attempts. Section 6 concludes this paper.

2 Multiple Software Product Line

One of the main ideas behind SPL engineering is to develop reusable artifacts (or core assets) that are then reused extensively during the development of final products [1,2]. Central to SPL engineering is the management of variability. FMs are widely used to capture SPL requirements in terms of common and variable features and compactly represent the valid feature combinations of an SPL. The validity of a configuration is determined by the semantics of FMs (e.g., x32 and x64 are mutually exclusive and cannot be selected at the same time in Fig. 1).

Definition 1 (SPL and Feature Model). *A software product line SPL_i is a set of products described by a feature model FM_i. Each product of SPL_i is a combination of features and corresponds to a valid configuration of FM_i. A configuration c of FM_i is defined as a set of features selected, i.e., $c = \{f_1, f_2, \ldots, f_m\}$ with f_1, f_2, ..., f_m features of FM_i. $[\![FM_i]\!]$ denotes the set of valid configurations of the feature model FM_i.*

FMs can be used to first perform domain analysis and scoping so that the variation to be supported in the production line is identified. Then, software

assets to be reused are implemented. In such top-down (or proactive) approach, SPL design and implementation is accomplished for all products in the foreseeable horizon. For example, the valid combinations of features supported by the family of laptop are first specified and products can then be derived. FMs can also be used in a bottom-up (or extractive) approach when an organization first considers existing custom software and extract the common and varying source codes into an SPL. In Fig. 1, four existing laptops exhibit some characteristics (OS, Processor, Monitor, etc.) and can be organized within a *family of laptop*. Customers can select a set of features from the family that fulfills their requirements such that the combination of features corresponds to an actual laptop product.

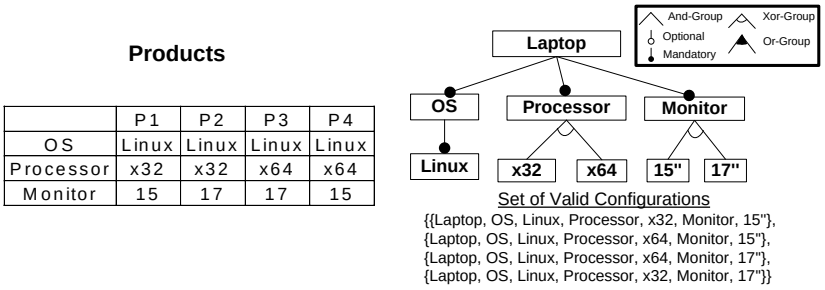


Fig. 1: Products and Product Line

In several domains, a growing diversity of products due to market demands more product variations and shorter introduction times. For example, several families of laptop are proposed to customers: laptops dedicated to public sector, small and medium business, large enterprise, etc. Customers select a laptop product among different (sub-)families of laptop such that the features of the laptop product fit well with the customers' requirements. Each laptop family can be configured according to different options (e.g., Wifi, Accessories such as Keyboard or Mouse, as shown in Fig. 2) and alternatives (e.g., the choice of a Processor or a GraphicCard). Nowadays, technology companies or suppliers do not only sell personal computers but also produce chipsets, processors, graphic cards, keyboards, etc. All of these products are software intensive, and each group of products forms a product family which can be assembled to construct a new set of laptop products.

In such context, there is opportunity to define and manage a large amount of products across different product lines and across all software development artifacts. In addition, some product family can be owned by different companies or provided by different suppliers. For example, the GraphicCard of a laptop can be chosen from Supplier_4 whereas the Processor of a laptop can be chosen among Supplier_3 products (cf. lower part of Fig. 2). The same remark applies for other components of a laptop (Motherboard, Keyboard, etc.). Achieving systematic reuse within these product families is a challenging problem: There is more and more need to combine several SPLs and manage variability across SPLs. We introduce here the term multiple SPL according to the following definition:

Definition 2. *A multiple SPL M_{SPL} is an SPL that manages a set of SPLs $\{SPL_1, SPL_2, \ldots, SPL_n\}$. Its set of products is described by a feature model $FM_{M_{SPL}}$.*

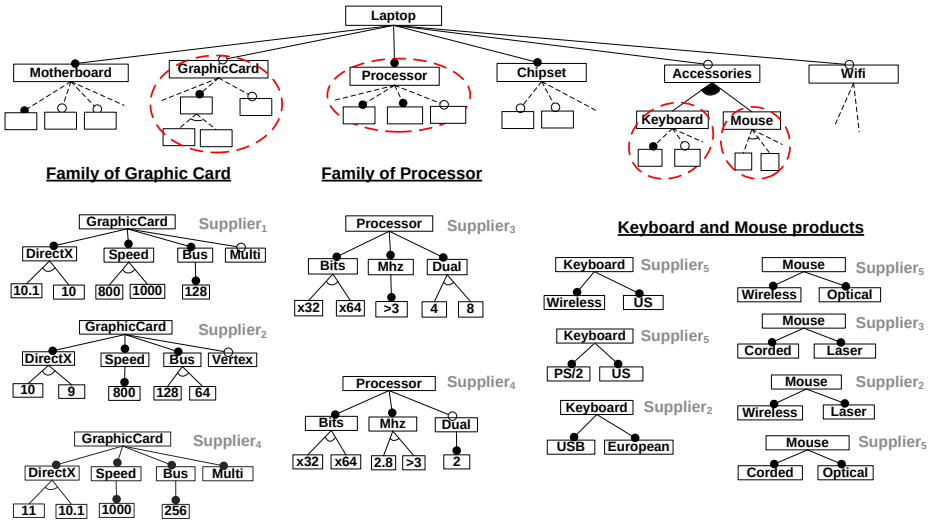


Fig. 2: Competing and Compositional Multiple Product Line

We distinguish competing from compositional multiple SPL. In a *competing* multiple SPL, different sets of competing products in the same market segment are usually provided by each SPL such that there is a fierce competition between SPLs. For example, three existing SPLs propose Graphic Cards with different features (see Fig. 2). Each SPL belongs to different suppliers or companies. The SPL of Supplier_2 is the only one to propose Graphic Cards that support Vertex but cannot propose Multi cores like SPLs of Supplier_1 and Supplier_3. In a competing multiple SPL, each combination of features should correspond to an actual product of at least one SPL. A product is a combination of features in which all features are provided by one and only one supplier. It is possible that for a given combination of features, more than one corresponding product exists. The configuration {GraphicCard, DirectX, 10, Speed, 800, Bus, 128} corresponds to two products provided by Supplier_1 and Supplier_2. It is also possible that no product exists for a given configuration, e.g., {GraphicCard, DirectX, 10.1, Speed, 1000, Bus, 256}. In a *compositional* multiple SPL, each part of the SPL may correspond to several SPLs, built by different parties, which are then combined to form an integrated software system. A product is a combination of features in which features can be provided by several different SPLs. For example, the Processor and the Graphic Card of an SPL of Laptops can be provided by two SPLs, i.e., an SPL of processors and an SPL of graphic cards. The same observation applies at the product level: Accessories products consist of Keyboard products and Mouse products.

As in the majority of real usage of SPLs, the family of laptop in Fig. 2 can be seen as a compositional multiple SPL in which several competing multiple

SPLs (Processor, GraphicCard, Keyboard, etc.) are assembled. The design process of a competing or a compositional multiple SPL can be realized in two different ways.

From SPLs to Multiple SPL. In a bottom-up approach, a multiple SPL is built from existing SPLs, in the same way an SPL is bootstrapped from existing products (see Fig. 1). For example, a competing multiple SPL, say M_{SPL} can be deduced from SPLs of $Supplier_1$, $Supplier_2$ and $Supplier_4$ such that each product of M_{SPL} corresponds to at least one product of an SPL of GraphicCard. Similarly, a compositional multiple SPL can be deduced from the market supply, e.g., a family of accessories including a set of existing Keyboards *and* a set of existing Mouses can be built.

From Multiple SPL to SPLs. In a top-down approach, the design of a multiple SPL is first formulated with no assumptions about the set of SPLs available. For example, a multiple SPL of Processor can be designed to attack a market and propose a new generation of Processors. Once designed, such a multiple SPL is then related to other SPLs. In a competing multiple SPL, the set of SPLs should be able to provide the entire set of products and cover all combinations of features. Similarly, in a compositional multiple SPL, at least one combination of SPLs should be able to provide the entire set of products.

Currently, we have introduced and described in an informal way the notion of multiple SPL. In Section 3, we first define the semantics of competing and compositional multiple SPL. Then, we show in Section 4 how merge operators can realize the semantics and can be used to manage multiple SPL using a bottom-up or a top-down approach.

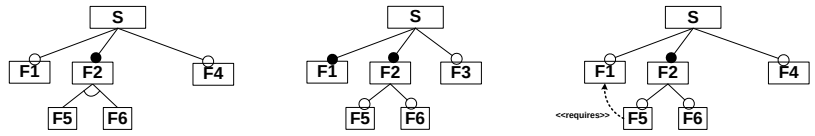
3 Semantics

We define the semantics of a multiple SPL in terms of the set of products of $SPL_1, SPL_2, \ldots, SPL_n$. Since FMs are used to represent the set of products of an SPL, it boils down to define the relationship between $FM_{M_{SPL}}$and FM_1, FM_2, ..., FM_n.

Definition 3 (Competing Multiple SPL). *Any product of a competing multiple SPL M_{SPL} is a product belonging to either SPL_1, SPL_2, ..., or SPL_n, i.e., any configuration of $FM_{M_{SPL}}$ should correspond to at least one valid configuration of FM_1, FM_2, ..., FM_n. Formally:* $\forall c \in [\![FM_{M_{SPL}}]\!], c \in [\![FM_1]\!] \vee c \in [\![FM_2]\!] \vee \ldots \vee c \in [\![FM_n]\!]$

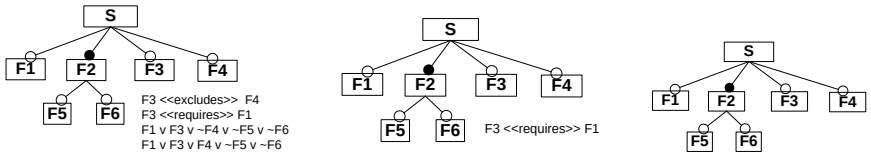
For example, the FM $FM_{M_{SPL}}$of Fig. 4a accurately represents the sets of configuration of the competing multiple SPL that manages the set of SPL represented by FMs FM_{supp_1}, FM_{supp_2} and FM_{supp_3} of Fig. 3a, 3b and 3c. This is not the case for the FM FM_{SIFM} of Fig. 4c. A counter example is given by { S, F2, F3, F4} which is a valid configuration of FM_{SIFM} but is not a valid configuration of either FM_{supp_1}, FM_{supp_2} or FM_{supp_3}.

In a competing multiple SPL, some suppliers or companies generate individual, unique, and user-specific products that others do not. Therefore, there is an interest to determine which products of a competing multiple SPL are unique.



(a) FM_{supp_1} (Supplier$_1$) (b) FM_{supp_2} (Supplier$_2$) (c) FM_{supp_3} (Supplier$_3$)

Fig. 3: Three SPLs and FMs from different suppliers (extracted from [9])



(a) $FM_{M_{SPL}}$(competing) (b) $FM_{M_{SPL}}$(compositional) (c) FM_{SIFM}

Fig. 4: Multiple SPL and FMs

Definition 4 (Product Uniqueness (Competing)). *A product p of a competing multiple SPL M_{SPL} is unique if p belongs exclusively to either SPL_1, SPL_2, ..., or SPL_n. Let p a product of M_{SPL} described by a configuration c. p is unique if and only if* $\exists i, \forall j \in 1...n, j \neq i, c \in [\![FM_{M_{SPL}}]\!] \wedge\ c \in [\![FM_i]\!] \wedge\ c \notin [\![FM_j]\!]$. *By extension, an SPL SPL_i of a competing multiple SPL M_{SPL} is unique if all products of SPL_i are unique:* $\forall c \in [\![FM_i]\!], \forall j \in 1...n, j \neq i, c \notin [\![FM_j]\!]$

For example, SPL_1 of Fig. 3a has two unique products: { S, F2, F5} and { S, F2, F4, F5} ; SPL_2 of Fig. 3b has three unique products: { S, F1, F2, F3}, { S, F1, F2, F3, F5}, { S, F1, F2, F3, F6} ; SPL_3 of Fig. 3c has four unique products: { S, F2}, { S, F2, F4}, { S, F2, F4, F1} and { S, F1, F2, F4, F5, F6}.

Similarly, we can define the commonality of a product, i.e., when products belong to every SPL of the competing multiple SPL.

Definition 5 (Commonality of a Product (Competing)). *A product p of a competing multiple SPL M_{SPL} is common if p belongs to SPL_1, SPL_2, ..., and SPL_n. Let p a product of M_{SPL} described by a configuration c. p is common if and only if* $c \in [\![FM_{M_{SPL}}]\!]\ \wedge\ \forall i \in 1...n, c \in [\![FM_i]\!]$

In the example of Fig. 3, there are two common products: { S, F1, F2, F5} and { S, F1, F2, F6}.

Definition 6 (Number of products). *The number of products of a multiple SPL M_{SPL} is denoted $|M_{SPL}|$ and defined as the cardinality of its set of products, i.e.,* $|M_{SPL}| = |[\![FM_{M_{SPL}}]\!]|$.

In a competing multiple SPL, $|[\![FM_{M_{SPL}}]\!]| \leq |[\![FM_1]\!]| + |[\![FM_2]\!]| + \ldots + |[\![FM_n]\!]|$ since the Inclusion-Exclusion principle of combinatorial mathematics applies. For

example, $|[\![FM_{M_{SPL}}]\!]| = 18 \leq |[\![FM_{supp_1}]\!]| + |[\![FM_{supp_2}]\!]| + |[\![FM_{supp_3}]\!]| = 28$. The reader can verify that the number of valid configurations of FM $FM_{M_{SPL}}$ depicted in Fig. 4a is 18 whereas the number of valid configurations of FM FM_{SIFM} (see Fig. 4c) is 32 (43% of the FM_{SIFM} configurations are not valid in FM_{supp_1}, FM_{supp_2}, or FM_{supp_3}).

The semantics of a compositional multiple SPL differs from the semantics of a competing multiple SPL.

Definition 7 (Compositional Multiple SPL). *Any configuration of a compositional multiple SPL M_{SPL} corresponds to either a configuration of FM_1, FM_2,..., or FM_n or any composition of configurations where each configuration belongs to either FM_1, FM_2,..., or FM_n. A composition of configurations c_1, c_2, ..., c_n produces a configuration $c_{n'}$ such that $c_1 \cup c_2 \cup \ldots \cup c_n = c_{n'}$.*

The FM $FM_{M_{SPL}}$ of Fig. 4b accurately represents the sets of configuration of a compositional multiple SPL which manages the set of SPL represented by FMs FM_{supp_1}, FM_{supp_2} and FM_{supp_3} of Fig. 3a, 3b and 3c. Note that the set of products of a compositional multiple SPL includes the set of products of a competing multiple SPL. For example, the configuration { S, F2, F4} which corresponds to a valid configuration in FM_{supp_3} is also a valid configuration in $FM_{M_{SPL}}$. The configuration { S, F1, F2, F3, F4} can be obtained by composing the configurations { S, F1, F2, F3} of FM_{supp_2} and { S, F2, F4} of FM_{supp_3}. Similarly, the configuration { S, F1, F2, F4, F5, F6} can be obtained by composing the configurations { S, F2, F5} of FM_{supp_1}, { S, F1, F2} of FM_{supp_2} and { S, F2, F6} of FM_{supp_3}. The FM FM_{SIFM} of Fig. 4c does not respect the semantics previously defined, e.g., counter examples are given by { S, F2, F5, F6, F3} and { S, F2, F5, F6, F3, F4} which are valid configurations of FM_{SIFM} but do not correspond to any configuration of FM_{supp_1}, FM_{supp_2} and FM_{supp_3} or any composition of configurations.

4 Merging Techniques to Manage Multiple SPL

4.1 Merge Operators

When two FMs share several features and are different viewpoints of a system, there is need to *merge* the overlapping parts of the two FMs to obtain an integrated model of the system. In prior work [11], we introduced a merge operator and we detailed the preserved properties when two FMs are merged.

Merge Operator Semantics. Several modes are defined for the merge operator and the properties of the merged FM is formalized with respect to the sets of configurations of input FMs. The *intersection* mode is the most restrictive option: the merged FM, FM_r, expresses the common valid configurations of FM_1 and FM_2. The merge operator in the intersection mode is noted: $FM_1 \oplus_{\cap} FM_2 = Result$. The relationship between a merged FM *Result* in intersection mode and two input FMs FM_1 and FM_2 can be expressed as follows:

$$[\![FM_1]\!] \cap [\![FM_2]\!] = [\![Result]\!] \qquad (M_1)$$

The *union* mode is the most conservative option: the merged FM, can express either valid configuration of first input FM or second input FM. In the union mode, we want to obtain a new FM where each configuration that is valid *either* in FM_1 *or* FM_2, is also valid:

$$[\![FM_1]\!] \cup [\![FM_2]\!] \subseteq [\![Result]\!] \quad (M_2)$$

A more restrictive property, called *strict union*, is defined as follows:

$$[\![FM_1]\!] \cup [\![FM_2]\!] = [\![Result]\!] \quad (M_3)$$

The merge operator in the strict union mode is noted $FM_1 \oplus_{\cup} FM_2 = Result$ and respects the property M_3. Another restrictive property in union mode, called *cross product*, is defined as follows:

$$[\![FM_1]\!] \otimes [\![FM_2]\!] = [\![Result]\!] \quad (M_4)$$

where the cross product is defined as $A \otimes B = \{a \cup b \mid a \in A, b \in B\}$ (A and B being a set of sets). The merge operator in the cross product mode is noted: $FM_1 \oplus_{\otimes} FM_2 = Result$. Compared to [11], we introduce a new operator, called *diff*, and noted $FM_1 \oplus_{\setminus} FM_2 = Result$ such that the following property holds:

$$[\![FM_1]\!] \setminus [\![FM_2]\!] = \{x \in [\![FM_1]\!] \mid x \notin [\![FM_2]\!]\} = [\![Result]\!]$$

Note that all merge operators are associative and commutative except the diff operator.

Implementation. In [11], a reference algorithm for merge operators was proposed but we observe a lack of scalability and input FMs that are not sharing the same hierarchy *cannot* be merged. We present here a novel and more efficient implementation which relies on the use of Boolean logic and the algorithm proposed in [12] to construct a FM from propositional formula. The use of logic also raises the limitations of our previous work where we do not consider constraints.

In a FM, listing all selected features for each possible product individually is not feasible if the number of products is very large. Fortunately, the set of configurations represented by a FM can be compactly described by a propositional formula defined over a set of Boolean variables, where each variable corresponds to a feature [4, 12]. For instance, the strict union of two sets of configurations represented by two FMs, FM_1, and FM_2, can be computed as follows. First, FM_1 (resp. FM_2) FMs are encoded into a propositional formula ϕ_{FM_1} (resp. ϕ_{FM_2}). Then, the following formula is computed:
$\phi_{Result} = (\phi_{FM_1} \wedge not(\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1})) \vee (\phi_{FM_2} \wedge not(\mathcal{F}_{FM_1} \setminus \mathcal{F}_{FM_2}))$

with $\mathcal{F}_{FM_1}$ (resp. $\mathcal{F}_{FM_2}$) the set of features of FM_1 (resp. FM_2) FM. $\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1}$ denotes the complement (or difference) of $\mathcal{F}_{FM_2}$ with respect to $\mathcal{F}_{FM_1}$. *not* is a function that, given a non-empty set of features, returns the Boolean conjunction of all negated variables corresponding to features:

$not(\{f_1, f_2, ..., f_n\}) = \bigwedge_{i=1..n} \neg f_i$

Computing the intersection of two sets of configurations represented by two FMs, FM_1, and FM_2, follows the same principles and we obtain:

$\phi_{Result} = (\phi_{FM_1} \wedge not(\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1})) \wedge (\phi_{FM_2} \wedge not(\mathcal{F}_{FM_1} \setminus \mathcal{F}_{FM_2}))$

At the moment, for all merge mode considered, ϕ_{Result} is solely a compact representation of the sets of configurations of the expected FM. The hierarchy of

the FM and the structuring information are still to be constructed. Czarnecki et al. propose an algorithm to construct a FM from propositional formula [12]. More precisely, the algorithm constructs a tree with additional nodes for feature groups that can be translated into a basic FM. Importantly, the algorithm can restore the hierarchy of input FMs by indicating parent-child relationships (mandatory or optional features) and Xor- or Or-groups.

4.2 Using the Merge Operators

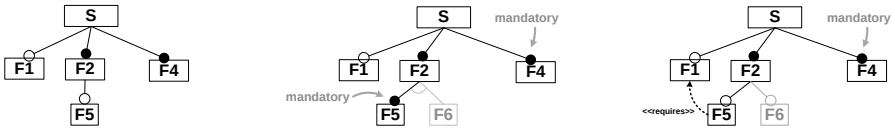
We show how the merge operators defined above can be used to manage both compositional and competing multiple SPL.

Competing Multiple SPL

From SPLs to competing multiple SPL. Let us consider a competing multiple SPL M_{SPL} that manages a set of SPLs $\{SPL_1, SPL_2, \ldots,$ or $SPL_n\}$. The computation of $FM_{M_{SPL}}$ using the merge operator in strict union mode can be realized as follows: $FM_{M_{SPL}} = FM_1 \oplus_{\cup} FM_2 \oplus_{\cup} \ldots \oplus_{\cup} FM_n$. For example, when considering FMs of Fig. 3, we obtain $FM_{M_{SPL}}$ of Fig. 4a by computing $FM_{M_{SPL}} = FM_{supp_1} \oplus_{\cup} FM_{supp_2} \oplus_{\cup} FM_{supp_3}$. $FM_{M_{SPL}}$ of Fig. 4a is synthesized using the following Boolean formula: $\phi_{result} = (S \wedge (F1 \Rightarrow S) \wedge (S \Leftrightarrow F2) \wedge (F4 \Rightarrow S) \wedge (F5 \Rightarrow F2) \wedge (F6 \Rightarrow F2) \wedge (F5 \Rightarrow F1) \wedge \neg F3) \vee (S \wedge (F1 \Rightarrow S) \wedge (F2 \Leftrightarrow S) \wedge (F4 \Rightarrow S) \wedge (F5 \Rightarrow F2) \wedge (F6 \Rightarrow F2) \wedge (\neg F5 \vee \neg F6) \wedge ((F2 \Rightarrow F5) \vee (F2 \Rightarrow F6)) \wedge \neg F3) \vee (S \wedge (F1 \Leftrightarrow S) \wedge (F2 \Leftrightarrow S) \wedge (F3 \Rightarrow S) \wedge (F5 \Rightarrow F2) \wedge (F6 \Rightarrow F2) \wedge \neg F4)$ The number of products of a competing multiple SPL (see Definition 6) can be computed by counting the number of satisfying variable assignments of ϕ_{result}.

From Products to SPL. Constructing an SPL from products can be done by applying the merge operator in strict union mode. For example, the FM of Fig. 1 is computed using the products description table and where each product description is represented as a FM with no variability, i.e., in which all features are mandatory.

From competing multiple SPL to SPLs. In a top-down approach, when stakeholders design a new competing multiple SPL, the set of products is first specified. Then, there is need to determine, for each product, which SPLs are suitable to provide the given product. For example, we want to determine which suppliers between Supplier$_1$, Supplier$_2$ and Supplier$_3$ (see Fig. 3) can provide a subset of the products of the FM depicted in Fig. 5a. In this case, the merge operator in intersection mode is applied. For example, the computation of $FM_{new} \oplus_{\cap} FM_2$ gives the empty set so that we know Supplier$_2$ cannot provide any product. Supplier$_1$ can provide two products represented by the two following configurations: { S, F2, F4, F5, F1} and { S, F2, F4, F5}. Supplier$_3$ can provide three products represented by the following configurations: { S, F2, F4, F5, F1}, { S, F2, F4, F1}, { S, F2, F4}. As a result, the competing multiple SPL is an SPL represented by FM_{new} and which manages the SPLs provided by Supplier$_1$ and Supplier$_3$. The set of products provided by Supplier$_1$ is described by $FM_{supp'_1} = FM_{new} \oplus_{\cap} FM_{supp_1}$ of Fig. 5b in which the feature F6 is no longer proposed while



(a) FM_{new} (b) $FM_{supp'_1}$ (Supplier$_1$) (c) $FM_{supp'_3}$ (Supplier$_3$)

Fig. 5: A new competing multiple SPL and suppliers' FM updates of Fig. 3

F4 and F5 are now mandatory features. Similarly, the set of products provided by Supplier$_3$ is described by $FM_{supp'_3} = FM_{new} \oplus_\cap FM_{supp_3}$ of Fig. 5c.

It is then necessary to determine whether the set of suppliers is able to provide *all* products of FM_{new}. We can compute the union of each set of products that belong to FM_{new} *and* that are provided by each supplier. In this case, the union set should be equal to the set of products of FM_{new} and the following relation should hold: $(FM_{new} \oplus_\cap FM_{supp_1}) \oplus_\cup (FM_{new} \oplus_\cap FM_{supp_2}) \oplus_\cup (FM_{new} \oplus_\cap FM_{supp_3}) = FM_{new}$. Using properties of the merge operators, the relation can be rewritten: $FM_{new} \oplus_\cap (FM_{supp_1} \oplus_\cup FM_{supp_2} \oplus_\cup FM_{supp_3}) = FM_{new}$. We know that $FM_{supp_1} \oplus_\cup FM_{supp_2} \oplus_\cup FM_{supp_3} = FM_{M_{SPL}}$ so that $FM_{new} \oplus_\cap FM_{M_{SPL}} = FM_{new}$. According to set theory, this is equivalent to $[\![FM_{new}]\!] \subseteq [\![FM_{M_{SPL}}]\!]$. As a result, when $FM_{M_{SPL}}$ is computed (see Fig. 4a), ensuring that the set of products of FM_{new} is provided by all suppliers is equivalent to determine if FM_{new} is a specialization or a refactoring[3] of $FM_{M_{SPL}}$. It can dramatically reduce the amount of time needed when the number of suppliers to consider is important since there is no more need to check the two-by-two compatibility between each supplier and the new specification of the SPL.

Commonality of a product. The set of common products of a competing multiple SPL corresponds to the set of configurations represented by FM_{common} which can be computed as follows: $FM_1 \oplus_\cap FM_2 \oplus_\cap \ldots \oplus_\cap FM_2 = FM_{common}$. Note that the set of common products can be rendered as a feature diagram.

Uniqueness of a product. According to Definition 4, a unique product belongs necessary to only one SPL. The unique products of an SPL_i corresponds to the set of configurations of FM_{unique_i} where $FM_{uniquei} = FM_i \oplus_\setminus (FM_1 \oplus_\cup \ldots FM_{i-1} \oplus_\cup FM_{i+1} \oplus_\cup \ldots \oplus_\cup FM_n$. The set of unique products can then be deduced by computing $FM_{unique_1} \oplus_\cup FM_{unique_2} \oplus_\cup \ldots FM_{unique_n}$.

Configuration process. In a competing multiple SPL, a bottom-up approach consists in first computing $FM_{M_{SPL}}$ according to FM_1, FM_2, ..., FM_n. In application engineering, stakeholders then derive a product and select/deselect features. For example, when a user selects the feature F3 of $FM_{M_{SPL}}$ (see Fig. 4a), the Boolean formula $\phi_{result} \wedge F3$ can be used to check the consistency of the feature selection or to deduce the possible values (e.g. selected/deselected) for features that have not been previously configured by the user. This is equivalent to activate the variable $F3$ in each Boolean formula $\phi_{FM_{supp_1}}$, $\phi_{FM_{supp_2}}$ and $\phi_{FM_{supp_3}}$ (corresponding resp. to FM_{supp_1}, FM_{supp_2} and FM_{supp_3}) and then

[3] We rely on the terminology used in [13]. Let f and g be FMs. f is a *generalization* of g if $[\![f]\!] \subset [\![g]\!]$; f is a *specialization* of g if $[\![g]\!] \subset [\![f]\!]$; f is a *refactoring* of g if $[\![g]\!] = [\![f]\!]$.

compute the merge in strict union mode: $F3 \wedge \phi_{result} = F3 \wedge ((\phi_{FM_{supp_1}} \wedge \neg F3) \vee (\phi_{FM_{supp_2}} \wedge \neg F4) \vee (\phi_{FM_{supp_3}} \wedge \neg F3)) = ((\phi_{FM_{supp_1}} \wedge F3) \wedge \neg F3) \vee ((\phi_{FM_{supp_2}} \wedge F3) \wedge \neg F4) \vee ((\phi_{FM_{supp_3}} \wedge F3) \wedge \neg F3)$. As a result, $FM_{M_{SPL}}$ can be used *independently* during configuration process without considering FM_1, FM_2, ..., FM_n.

Evolution of multiple SPL. A competing multiple SPL M_{SPL} can evolve during time. For example, the set of products of a new supplier can be taken into account. When an SPL SPL_{n+1} is added into a competing multiple SPL M_{SPL}, we should obtain a new competing multiple SPL $M_{SPL'}$ such that $[\![FM_{n+1}]\!] \cup [\![FM_{M_{SPL}}]\!] = [\![FM_{M_{SPL'}}]\!]$. Since the merge operator in strict union mode is commutative, the evolution of a multiple SPL can be realized in an incremental manner and there is no need to compute again the merge of all FMs managed by M_{SPL}. Interestingly, when FM_{n+1} is a specialization (resp. generalization) of $FM_{M_{SPL}}$, then $FM_{M_{SPL'}} = FM_{M_{SPL}}$ (resp. $FM_{M_{SPL'}} = FM_{n+1}$).

Compositional Multiple SPL. The management of a compositional multiple SPL follows the same principles as previously described: Merging operators are applied on the set of FMs. Let us consider a compositional multiple SPL M_{SPL} that manages a set of SPLs $\{SPL_1, SPL_2, \ldots, \text{or } SPL_n\}$. The computation of $FM_{M_{SPL}}$ can be realized by using a compositional operator which combines the merge operator in the strict union mode and in the cross product mode. The compositional operator, noted $\oplus_{comp}$, processes two input FMs FM_1 and FM_2 as follows: $FM_1 \oplus_{comp} FM_2 = (FM_1 \oplus_{\cup} FM_2) \oplus_{\cup} (FM_1 \oplus_{\otimes} FM_2)$. Then we can apply the compositional operator on the set of FMs, i.e., $FM_{M_{SPL}} = FM_1 \oplus_{comp} FM_2 \oplus_{comp} \ldots \oplus_{comp} FM_n$. For example, when considering FMs of Fig. 3, we obtain $FM_{M_{SPL}}$ of Fig. 4b by computing $FM_{M_{SPL}} = FM_{supp_1} \oplus_{comp} FM_{supp_2} \oplus_{comp} FM_{supp_3}$.

From compositional multiple SPL to SPLs. In a top-down approach, determining whether all products of a compositional multiple SPL, described by FM_{comp}, can be derived can be done as previously described by checking if FM_{comp} is a specialization or a refactoring of $FM_{M_{SPL}}$, i.e., the following relation should hold: $[\![FM_{comp}]\!] \subseteq [\![FM_{M_{SPL}}]\!]$. A second issue is to determine which products of SPLs can be composed to build products of a compositional multiple SPL. In this case, all features that are *not* in $FM_{M_{SPL}}$ but are in FM_i (for $i \in 1..n$) are deselected in FM_i. Intuitively, all valid configurations of FM_i which contain a feature `F` not present in any configuration of $FM_{M_{SPL}}$ are removed. For example, when we want to determine which suppliers between Supplier_1, Supplier_2 and Supplier_3 (see Fig. 3) can provide a subset of the products of the FM depicted in Fig. 5a, feature `F6` is deselected in FM_{supp_1}, FM_{supp_2} and FM_{supp_3} and feature `F3` is deselected in FM_{supp_2}.

5 Assessment and Discussion

Managing (competing or compositional) multiple SPL essentially deals with the use of merging techniques. We implemented the merging operators presented in Section 4.1 using Binary Decision Diagrams (BDD) [14, 15]. BDDs are widely used in digital system design [15], model checking and even in the feature modeling community (e.g., [12, 16, 17]). A BDD can be seen as a compressed representation of a binary function (e.g., propositional formula) and can thus handle the sets of configurations of FMs.

We chose to use BDDs since computing the negation, conjunction or disjunction of two BDDs can be performed in at most polynomial time with respect to the sizes of the BDDs involved [14]. As presented in Section 4.1, these logical operations are intensively used during the merging of several FMs. Moreover efficient optimized implementations of these operations are provided by off-the-shelf BDD libraries (e.g., we use the open source JavaBDD library [18]). Another important reason is that the algorithm of Czarnecki et al. [12], which synthesizes FMs from propositional formulas, also relies on a BDD-based implementation. Hence the BDD resulting from the merge operations can be handled by their algorithm. Lastly, polynomial algorithms are available for computing valid domains during configuration process or counting the number of products.

5.1 Scalability

Experimental Setup. We evaluate how scaling the size of the models affects performance of merge operators implementation based on BDDs. We use randomly generated FMs to produce inputs with variations on i) the number of SPLs (noted nFM) involved in the multiple SPL, ii) the number of features *commonly* shared by SPLs (noted $nComm$), and iii) the percentage of features commonly shared by SPLs, (noted *per*). We make *per* vary between 100% (which corresponds to the case where all features are commonly shared) and 50%, so that we can analyze the merge behavior when all features are not necessary shared by FMs (we consider that 50% is a threshold under which it is not relevant to merge, at least for competing SPL). For each value of *per*, we experimented different values of nFM, (to determine how many FMs can be handled by the merge) and $nComm$ (to determine the manageable size of input FMs[4]) on the merging techniques and we measured the calculation time needed to perform logical operations on BDDs. About input FMs, we randomly create an initial FM and perform random edits (similarly as in [13]) such that we obtain new input FMs. Then, features are randomly added to these FMs. This way, we can parametrically control nFM, $nComm$ and *per*.

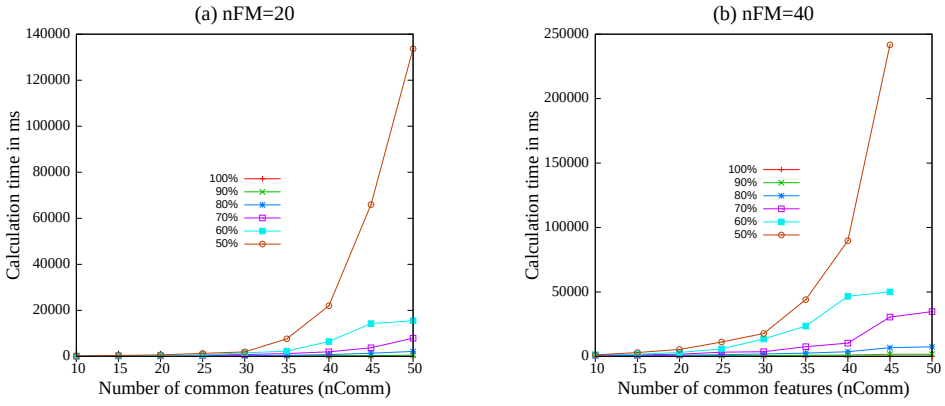


Fig. 6: Calculation time vs number of common features (strict union merge)

[4] We chose to focus on the common features of each FM since the goal of the merge operators is to group two or $nComm$ features with the same name into one feature.

Results. In Fig. 6, we report our results when $nFM = 20$ and $nFM = 40$, with merging performed in strict union mode[5]. Experiments show that when all features are commonly shared by SPLs ($per = 100\%$), the calculation time is almost linear to the number of features and the implementation scales even for nFM greater than 200 and for $nComm$ greater than 200. With $per \leq 60\%$ scalability issues occur when $nFMs = 40$ and for $nComm \geq 45$ (see Fig. 6). For these values, the logical operations fail since data overflows the main memory.

In Section 4.1, we explained that features with the same name are encoded into the same Boolean variable in each BDD. In our experimental setup, the total number of Boolean variables mainly depends on the number of features *not* commonly shared by FMs: $nBooleanVariable = nComm + (nFM * nComm(\frac{1-per}{per}))$ The experimental results show that the increase of the calculation time is proportional to the total number of Boolean variables manipulated in each BDD. As a result, the merge operations can efficiently manage a large number of SPLs even with a lot of features, especially when the SPLs share a large amount of common features (between 80% and 100%).

The experimental results consider only the use of logical operations on BDD, but the hierarchy of the FM and the structuring information are still to be built. Preliminary experiments indicate that on typical Boolean formulas, the algorithm presented in [12] scales up to 300 variables, e.g. the number of common features should not exceed 300 when $per = 100\%$. Moreover current techniques to compile FMs with BDD scale for a number of features lesser than 2000 [16]. Obviously, these limitations also apply to our approach, but rebuilding the complete FM is only needed when it will have to be manipulated by a user.

5.2 Related Work

This work is partially inspired by the work of Hartmann et al. [9], in which the authors introduce the *Supplier Independent Feature Model* (SIFM) in order to select products among the set of products described by several *Supplier Specific Feature Models* (SSFM). The SIFM contains the "super-set of the features" from all the suppliers and constraints are specified to inter relate features of SIFM and SSFMs. The approach proposed has the advantage to be realizable by current feature modeling tools and techniques. We now compare our proposals.

The definition of a competing multiple SPL corresponds to the supplier independent problem described in Hartmann et al. The *semantics* presented in Section 3 aimed at providing a sound framework for the management of multiple SPL and opens new perspectives to handle new properties (e.g., uniqueness of a product) not defined in [9]. The approach of Hartmann et al. leads to reasoning on a large set of features (i.e., SIFM and SSFMs' features) related by a large number of constraints. The number of variables to be generated may become an issue in terms of computational or space *complexity* and hinder some automated analysis operations of FMs [17]. For example, when nFM is equal

[5] We also performed experiments on other merging operators. In addition to the calculation time, we measured the memory space (i.e., the number of nodes in the resulting BDD). Due to the page limits, only an excerpt of figures is given. All empirical results for $10 \leq nFM \leq 250$, $50 \leq per \leq 100$ and $10 \leq nComm \leq 250$ are available in `http://modalis.polytech.unice.fr/softwares/mofm`

to 200, *nComm* is equal to 200 and *per* is equal to 100%, there is need to consider $200 * 200 = 40000$ variables which becomes intractable with state-of-the-art reasoning tools. In our approach, the FM representing the set of configurations of a multiple SPL is *independent* from the other FMs (see Section 4). This is not the case when using SIFM since when a feature is selected/deselected, reasoning tools have to consider every SSFM and all constraints before updating SIFM. Similarly, determining if a subset of products can be provided by suppliers cannot be done without considering all SSFMs. From a *user perspective*, the super-set of all supplier features over approximates the sets of configurations and hides some constraints to the user (see Fig. 4c in Section 3). For example, users cannot understand that features `F3` and `F4` are mutually exclusive until considering constraints between SIFM and SSFMs. In our approach, the set of configurations represented by the FM can be directly used. A limitation comes when hierarchies of input FMs to be merged are not the same. In this case, it is difficult do retrieve a satisfying hierarchy and refactoring operations must be performed manually. Finally, using SIFM and SSFMs, each feature is distinctly treated even when features have the same name so that users can define fine grained dependencies between features. To handle such situations, our merging techniques have to be extended.

Other Related Work. Reiser and Weber propose to use multi-level feature trees consisting of a tree of FMs in which the parent model serves as a reference FM for its children [10]. Their purpose is mostly to cope with large diagrams and large-scale organizations. They do not not provide a solution for managing competing multiple SPL or operators to merge FMs. Buhne et al. [8] describe an approach to model multiple product lines by using an extension of the OVM model [2]. Batory et al. have shown that SPL development using layered designs scales to product lines of program families [19]. The purpose of their work is to generate families of programs from a single code base and reasoning about program families. A few approaches consider the merging of FMs [20,5,21] but do not provide a solution to implement all merge operators defined in Section 4.1. In [22], the case of *synchronizing* existing configurations of a cardinality-based FM that have evolved over time is considered and can be seen as a merge. The composition operators used in this paper are restricted to basic [12] FMs.

6 Conclusion

Managing multiple SPL with FMs is a tedious and error-prone activity as software architects have to manually create and edit a large number of feature models. In this paper, we have introduced a comprehensive framework to represent and reason on several SPLs. We defined multiple SPL and distinguished competing from compositional multiple SPLs. Their semantics is defined through FMs and salient properties are described by relying on the expressed sets of configurations. We have shown that FM merging techniques can be directly reused to help in managing the different kinds of SPL. An implementation based on Boolean logic is also described, thus providing a both sound and fully mechanizable basis for the proposed framework. Experimentation with different kinds and sizes of FMs demonstrates that merging techniques scale even to a large number of FMs with hundred of features. In order to complement the proposed framework, a large scale validation in the medical imaging domain is about to

start [23]. Image analysis workflows and services will be managed through several SPLs, with both competing and compositional subparts. Future work also comprises improvement on the implementation, with focus on better performance and integration with a feature modeling tool.

References

1. Clements, P., Northrop, L.M.: Software Product Lines : Practices and Patterns. Addison-Wesley Professional (2001)
2. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag (2005)
3. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21 (1990)
4. Batory, D.S.: Feature models, grammars, and propositional formulas. In: SPLC '05. Volume 3714 of LNCS., Springer (2005) 7–20
5. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Comput. Netw. **51**(2) (2007) 456–479
6. van Ommering, R.: Building product populations with software components. In: ICSE '02, ACM (2002) 255–265
7. Hartmann, H., Trew, T.: Using feature diagrams with context variability to model multiple product lines for software supply chains. In: SPLC'08, IEEE (2008) 12–21
8. Buhne, S., Lauenroth, K., Pohl, K.: Modelling requirements variability across product lines. In: RE '05, IEEE (2005) 41–52
9. Hartmann, H., Trew, T., Matsinger, A.: Supplier independent feature modelling. In: SPLC'09, IEEE Computer Society (2009) 191–200
10. Reiser, M.O., Weber, M.: Multi-level feature trees: A pragmatic approach to managing highly complex product families. Requir. Eng. **12**(2) (2007) 57–75
11. Acher, M., Collet, P., Lahire, P., France, R.: Composing Feature Models. In: 2nd Int'l Conference on Software Language Engineering (SLE'09). LNCS (2009) 20
12. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: SPLC 2007. (2007) 23–34
13. Thüm, T., Batory, D., Kästner, C.: Reasoning about edits to feature models. In: ICSE'09, IEEE (2009)
14. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a bdd package. In: DAC '90: Design Automation Conference, ACM (1990) 40–45
15. Minato, S.i.: Binary decision diagrams and applications for VLSI CAD. Kluwer Academic Publishers, Norwell, MA, USA (1996)
16. Mendonca, M., Wasowski, A., Czarnecki, K., Cowan, D.: Efficient compilation techniques for large scale feature models. In: GPCE '08, ACM (2008) 13–22
17. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated Analysis of Feature Models 20 years Later: a Literature Review. Information Systems, Elsevier (2010)
18. JavaBDD: `http://javabdd.sourceforge.net/index.html`
19. Batory, D., Lopez-Herrejon, R.E., Martin, J.P.: Generating product-lines of product-families. In: ASE '02: Automated software engineering, IEEE (2002) 81–92
20. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: GPCE'06, ACM (2006) 201–210
21. Segura, S., Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Automated merging of feature models using graph transformations. Post-proceedings of the Second Summer School on GTTSE **5235** (2008) 489–505
22. Kim, C.H.P., Czarnecki, K.: Synchronizing cardinality-based feature models and their specializations. In: ECMDA-FA'05. LNCS Vol. 3748, Springer (2005) 331–348
23. Acher, M., Collet, P., Lahire, P., Montagnat, J.: Imaging Services on the Grid as a Product Line: Requirements and Architecture. In: Service-Oriented Architectures and Software Product Lines (SOAPL'08), workshop at SPLC'08, IEEE (2008)