Product Lines of Product Lines

D. Batory and C.T. Shepherd
Department of Computer Science
University of Texas at Austin
Austin, TX 78712 USA
batory.cts@cs.utexas.edu

Abstract—A Software Product Line (SPL) is a set of related programs. A SPL of a SPL (SPL²), or a product line of a product line, is a fundamental scaling of product line concepts. A member of a SPL² is not a program, but a SPL. The idea recurses: a SPL³ is a SPL of a SPL², and so on. The depth of recursion is the rank of the product line. We present an algebra to understand and build feature-based product lines of arbitrary rank that is consistent with our experiences in building several SPLs of rank two and higher.

I. INTRODUCTION

A Software Product Line (SPL) is a family of related programs constructed from a common set of assets [1]. Variations in programs are explained by features — increments in program functionality. The assets of a SPL are modules that implement features. These modules are the building blocks of SPL programs [2].

A fundamental scaling of product lines is to apply its concepts recursively. Just as a SPL expresses variations in a family of programs, a SPL of a SPL (SPL²) expresses variations in a family of SPLs. That is, a SPL can itself have variations and a family of related SPLs can be constructed from a common set of assets. The idea recurses: a SPL³ is a SPL of SPL², etc. A SPLⁿ has rank n. Product lines of rank 2 and higher arise both in theory and practice, which we illustrate shortly [3]-[4][5][6][7][8][9][10]-[11].

Years ago, we built SPL² and SPL³ and noticed an unusual property that their modules exhibited a Cartesian relationship [12][3]-[2]. For example, the assets of a SPL² could be arranged as a 2D array of modules. Module expressions — another name for a composition of modules — are produced by eliminating unnecessary rows and columns of this array, and composing the remaining modules. An analog holds for SPLⁿ and n-dimensional arrays. We never understood why this was so.

Feature Oriented Software Development (FOSD) has two intertwined goals. The first is to show how features can be modularized and how programs can be synthesized by composing modules. The second is to define an algebraic foundation for composition and synthesis. Programs are structures and software engineers use tools to manipulate and transform these structures. Compilers, for example, transform source structures to bytecode structures. Refactoring tools transform source structures to (refactored) source structures. Model Driven Engineering (MDE) uses models (structures) to describe programs and transformations to map models-to-models and models-to-code. Software development is replete with structure creation and manipulation, and mathematics is the language of such activities.

In this paper, we integrate recent advances in FOSD [13][14] with operations on n-dimensional arrays to explain SPLⁿ. Briefly, a SPL¹ partitions a code base into modules that implement features. A SPL² is created by overlaying two different SPL¹ partitions of the same code base, yielding a 2D array of modules; a SPLⁿ is an overlaying of n different SPL¹ partitions, yielding a n-D array of modules. We present an algebra of SPL overlays to show how SPLs of arbitrary rank can be understood and built, and that is consistent with our prior implementation experiences. Doing so reveals new algebraic laws of FOSD construction. It is the explanation and use of these laws that is the contribution of this paper.

To give readers a concrete grounding for SPLⁿ abstractions, we use the Expression Problem (EPL²) [4][15][6][8][16][11] as a running example and the Operation Product Line (OPL¹) as its precursor. We begin by showing how SPLⁿ arise in practice.

II. ORIGINS OF SPLⁿ IN PRACTICE

Consider a product line of electric shavers that is offered to Europeans (Figure 1a). F₁ is the base product with optional features F₂, F₃, and F₄. Each box represents a module that implements a feature; ignore box shading for now.

![Figure 1. Two Different SPL's](image)

An Asian market for these shavers arises, leading to a slightly different product line (Figure 1b). F₁ is the base product with optional features F₂, F₄, and F₅. But as typically happens, the European feature F₂ is not quite the same as the Asian feature F₂, but there is clearly some commonality in their modules (the clear part is shared and the shaded parts are different). Further, features F₁ and F₄ are identical, and features F₃ and F₅ are market-specific.

In these circumstances, a single integrated product line is created for both markets. The reason is pragmatics: (a) to
eliminate redundancy, (b) to provide simpler specifications — identify the region and then select shaver features, instead of selecting individual modules, and (c) to manage complexity [17][5][9].

The integrated product line gives rise to a 2D array of modules (Figure 2). Rows represent regions and columns represent product features. The modules of the common row are shared by both European and Asian SPLs. Modules that are specific to Europe or Asia are in separate rows.

To specify a product for Europe, the common and Europe rows are composed into a single row, reforming the Europe SPL of Figure 1a, at which point, European products can be configured. The same holds for Asian products.

As different regions (North America) enlarge the market for shavers, new rows and new columns appear, still observing the Cartesian relationships described above. Further, new axes of variability, such as user type and new product types, extend the 2D array into arrays of higher dimensionality. Examples identical the above scenario are well-known (at least informally) in the product line literature [17][5][7][9][10].

A simple way to understand a SPL is as an overlaying two orthogonal partitionings of a single code base: one partitions the code base by features depicted as rows, the other by features depicted as columns. A 2D array of modules results by superimposing partitions, representing the modules of the SPL. There is a direct generalization to SPL.

To give these ideas a formal description, we review the basics of SPL using a concrete example.

III. PRODUCT LINES OF RANK 1

The Operation Product Line (OPL) implements arithmetic expressions as trees. OPL programs differ in the operations allowed on trees. Trees can be printed and evaluated. The expression \((\text{add } 4 \ 5)\) is a tree with add as the root and 4 and 5 as leaves. Printing \((\text{add } 4 \ 5)\) yields the string “4+5” and evaluating it returns 9.

The code for the entire OPL product line is listed in Figure 3. Code fragments that are unshaded appear in all OPL programs; shaded fragments appear in some programs. OPL has three features: core, print, and eval. A single module implements each feature and is given its name. (We remove this restriction in Section V to allow a feature to be implemented by several modules.) The core module contains uncolored code. The print module contains all green-colored fragments, and the eval module contains all red-colored fragments.

The position at which a code fragment is inserted is called a variation point (VP) [18]. A VP with name \(n\) is designated \(n\) in Figure 3. All VPs in OPL belong to the core module. Every variation point \(n\) is paired with a single fragment whose name is \(\text{vpn}\). The odd-numbered VPs are paired with fragments from the print module, such as \(\text{print}\). Even-numbered VPs are paired with fragments from the eval module, such as \(\text{eval}\).

Although OPL modules consist of code fragments, these fragments have meaning. The core module contains a class hierarchy rooted by the abstract class Exp with concrete subclasses Int, Plus, and Times. No class in core has methods. The print module adds a print() method to
to each class. The \texttt{eval} module adds an \texttt{eval()} method to each class. Figure 5 shows three programs of OPL, where \( \oplus \) is the module composition operation. Each of these programs can create expression trees that add and multiply integers. The \texttt{core} program can neither print nor evaluate expressions; the program labeled “\texttt{print}\oplus\texttt{core}” can print expressions, and program “\texttt{eval}\oplus\texttt{print}\oplus\texttt{core}” can both print and evaluate expressions.

Module composition is simple to compute. Let \( A \) and \( B \) be modules. \( A \oplus B \) is the union of the code fragments in \( A \) and \( B \); fragments are paired with their VPs. If pairing is not possible, fragments are retained for later pairing. So the composite module \( \texttt{print} \oplus \texttt{eval} \) is the set of all \texttt{print} and \texttt{eval} fragments, none of which are paired (as neither \texttt{print} or \texttt{eval} fragments have VPs). The composition \( \texttt{print} \oplus \texttt{core} \) pairs every \texttt{print} fragment with a VP in \texttt{core}; only the even-numbered VPs of \texttt{core} have no fragments. Note that the order in which modules are summed does not matter; summation is both commutative and associative.

\( A \). The Axioms of Summation (\( \oplus \))

Let \( R, S, T \) be modules as described above. Three axioms govern summation [13]:

\[
\begin{align*}
\text{Identity} : \quad & R \oplus 0 = R \\
\text{Commutativity} : \quad & R \oplus S = S \oplus R \\
\text{Associativity} : \quad & R \oplus (S \oplus T) = (R \oplus S) \oplus T
\end{align*}
\]

The empty module \( (0) \) contains no fragments. Summing 0 with another module yields that module. Summation is commutative and associative, as it is based on set union.

\( B \). Feature Models

Not all combinations of features are meaningful. Some features require the presence or absence of other features. Meaningful combinations are defined by a feature model, which is a context-sensitive grammar [19]. A feature model consists of (1) a context-free grammar with no recursion or repetition and whose sentences define a superset of all legal feature expressions, and (2) a set of constraints (the context-sensitive part) that eliminates nonsensical sentences.

We write product line \( M \) has feature model \( \hat{M} \). Suppose the grammar of \( \hat{M} \) is a single production with optional features indicated in [brackets]. \( \hat{M} \) below defines eight sentences (features \( x, y, z \) are optional; \( w \) is mandatory). The constraints of \( \hat{M} \) limit legal sentences to those that have at least one optional feature, and if feature \( x \) is selected, so too must \( z \):

\[
\hat{M} : \quad \{ [x] [y] [z] w \} \quad \text{context free grammar} \\
\{ x \lor y \lor z \} \quad \text{constraints} \\
x \implies z;
\]

For now, assume a simple universe where there are no feature interactions. (We remove this restriction in Section V-E.) Given a sentence of \( \hat{M} \) (such as \( \{xzw\} \)) a sum is taken of its terms to map it to a module expression \( (x \oplus z \oplus w) \). A program of \( M \) is synthesized by evaluating this expression. Returning to OPL, its feature model OPL is:

\[
\text{OPL} : \quad \{ \text{eval} \} \{ \text{print} \} \{ \text{core} \}; (4)
\]

which is a compact way of writing all four OPL sentences, one for each program in OPL: core, print \( \oplus \) core, eval \( \oplus \) core, and eval \( \oplus \) print \( \oplus \) core.

\( C \). Arrays

Instead of informally saying OPL has the modules \texttt{core}, \texttt{print}, and \texttt{eval}, we list its modules in a 1-dimensional array OPL\(_j\). (Here we use a standard naming convention: the name of the array is OPL and its rank is indicated by the number of subscripts, which in this case is one). The order in which features are listed does not matter:

\[
\text{OPL}_j = [\text{ eval } \text{ print } \text{ core } ]
\]

There are two operations on arrays: projection and contraction. \textit{Projection} is the elimination of unnecessary elements. Let \( D_j \) be the projection of OPL\(_j\) that retains only the \texttt{core} and \texttt{print} modules. Projection is written as a membership constraint per index:

\[
D_j = \text{OPL}_j \{ \text{print,core} \} = [\text{ eval } \text{ print } \text{ core }]_{j \in \text{print,core}} = [\text{ print } \text{ core }]
\]

\textit{Contraction} aggregates elements of an array. The summation-contraction of a module array \( A_j \) along dimension \( j \) is written \( \sum_j A_j \), or simply \( \sum A_j \) when all modules are to be summed. (We have only one dimension now, but there are more later). So program \( P1 = \text{print} \oplus \text{core} \) can be written as a projection and contraction of OPL\(_j\):

\[
P1 = \sum \text{OPL}_j \{ \text{print,core} \} = \sum [\text{ eval } \text{ print } \text{ core }]_{j \in \text{print,core}} = \sum [\text{ print } \text{ core }] = \text{print} \oplus \text{core}
\]

The legal projections of \( \text{OPL}_j \) – that is, the legal subsets of features – are defined by feature model OPL\(_j\). The contraction of each legal projection of \( \text{OPL}_j \) is a module expression, which when evaluated synthesizes a program of OPL. All programs of OPL are produced in this manner.

We conclude this section by formally defining a product line \( S^j \) as the ordered pair \((S, S_j)\), its feature model and its module array. OPL\(_1^j = (\text{OPL}, \text{OPL}_j)\).
### IV. Product Lines of Rank 2 and Higher

Suppose we want OPL to have variations. The `plus` and `times` functionalities are mandatory now. We want them to be optional. To accomplish this, we start with the members of OPL (the top row of programs in Figure 6). We then remove all code that is specific to the `times` functionality (namely the `Times` class), forming the set of programs in the middle row of Figure 6. Then we eliminate all code that is specific to the `plus` functionality (namely the `Plus` class), forming the set of programs in the bottom row of Figure 6. The resulting set of nine programs are members of the Expression Problem Product Line (EPL) [4][5][6][8][16][11].

#### A. Feature Models of Rank 2 and Higher

EPL has two axes of variability. One axis, defined by column features, specifies whether the `print` or `eval` functionalities are in programs. The legal combinations of column features is expressed by OPL, replicated below:

$$\text{OPL} : \text{[eval]} \text{[print]} \text{core}$$

The second axis, defined by row features, specifies whether the `plus` or `times` functionalities are present in OPL programs. The feature model, CPL, expresses the legal combinations of row features:

$$\text{CPL} : \text{[times]} \text{[plus]} \text{base}$$

A feature model of EPL expresses both variabilities (that of selecting which operations can appear in a tree and operations that can be performed on a tree):

$$\text{EPL} : \text{CPL OPL}$$

Using the two axes of variability, we can express the legal combinations of programs. To express this on a tree, we use a notation of `@` (``#``) to indicate the presence (absence) of a feature. We return to this observation shortly.

#### B. Feature Modules of Rank 2 and Higher

EPL has more variability than either OPL and CPL. This extra variability is achieved by adding more VPs to the OPL (or CPL) code base. Figure 7 shows the updated code base. VP `@` is paired with a code fragment that contains the `Plus` class, `5`, and `6`. VP `♦` is paired with a code fragment that contains the `Times` class, `7`, and `8`. In general, the codebase of a SPL² (or SPL⁰ for that matter) are indistinguishable from feature models of SPL¹: they are still context-sensitive grammars. As we will see next, there is nothing particularly special about the code base of a SPL² (or SPL⁰) either.

Simply put, a program in EPL is uniquely specified by two sentences: one from CPL and another from OPL. We return to this observation shortly.

EPL is a feature model of rank 2. In general, feature models of rank 2 are not always the concatenation of rank 1 models. There can be constraints that preclude or require certain row-column combinations that are not part of their rank 1 models. EPL does not exhibit this complexity. Models in [5][9] do. Also note that feature models of SPL² (or SPL⁰ for that matter) are indistinguishable from feature models of SPL¹: they are still context-sensitive grammars. As we will see next, there is nothing particularly special about the code base of a SPL² (or SPL¹) either.
Contracting the array produces a \( \text{EPL} \) print functionalities. We project (remove) the \( \text{let} \) sentences: one from \( \hat{\text{CPL}} \) in Section IV-C a simple way to compute the contents of \( \hat{\text{SPL}} \). We will see in Section V that features are selected, the fragments in the module \( i \) modules.

Recall that any program in \( \text{EPL} \) is specified by two sentences: one from \( \text{CPL} \) and another from \( \text{OPL} \). In general, let \( S_{ij} \) be the module array of \( S^2 \). A program \( P \) of \( S^2 \) is uniquely specified by a set of rows \( \alpha \) and a set of columns \( \beta \). This specification enables us to infer the modules of \( S_{ij} \) to sum: module \( r \# c \) is present in \( P \) iff \( r \in \alpha \) and \( c \in \beta \). Stated differently, programs of \( \text{SPL}^2 \) (and in general, \( \text{SPL}_n \)) can be produced by the projection and contraction of its module array.

Program \( P_2 \) of \( \text{EPL} \) supports the \text{core} and \text{eval} column functionalities and the \text{base}, \text{plus}, and \text{times} row functionalities. We project (remove) the \text{print} column of \( \text{EPL}_{ij} \), yielding a 2D array of modules that comprise \( P_2 \). (These are all the interaction modules that contain fragments that implement the selected row and column features). Contracting the array produces a \( \oplus \)-expression for \( P_2 \):

\[
P_2 = \sum_{\text{EPL}_{1j}} \text{EPL}_{ij} \]

\[
EPL_{ij} = \begin{bmatrix}
\text{times} \# \text{eval} & \text{times} \# \text{print} & \text{times} \# \text{core} \\
\text{plus} \# \text{eval} & \text{plus} \# \text{print} & \text{plus} \# \text{core} \\
\text{base} \# \text{eval} & \text{base} \# \text{print} & \text{base} \# \text{core}
\end{bmatrix}
\]

Removing the green-colored fragments (but retaining their VPs) in Figure 3 yields the code for \( P_2 \).

C. Axis Symmetries and Computing Array Entries

A 2D array has symmetry: its rows and columns can be interpreted and manipulated identically. This differentiates arrays from hierarchies, where rows or columns are given a dominant or privileged role in decomposition. Arrays have no dominant decomposition and thus can uniformly express decomposition by rows then columns, or decomposition by columns and then rows.

We presented \( \text{EPL} \) as a \( \text{SPL}^2 \) where each row of Figure 6 is a different variation of \( \text{OPL} \). Another interpretation is each \text{column} of Figure 6 is a \( \text{SPL}^1 \) and the set of all columns is a \( \text{SPL}^2 \). In this case, the variations among columns is the presence or absence of the \text{print} and \text{eval} functionalities. The ability to treat a row (or column) as a \( \text{SPL}^1 \) and rows (columns) as a \( \text{SPL}^2 \) is called \text{axis symmetry}. A \( \text{SPL}^2 \) has two axis symmetries: rows and columns. \( \text{SPL}_n \) has \( n \) axis symmetries. Symmetries lead to a simple way to understand rank 2 and higher SPLs and compute the contents of interaction modules.

As mentioned in Section II, a \( \text{SPL}^2 \) is an overlay of two different partitionings of a single code base.

To see this visually, Figure 9a depicts \( \text{OPL}^1 \) as a partitioning of a code base (the square) into features depicted as columns, making some of these columns optional. Figure 9b depicts \( \text{CPL}^1 \) as a partitioning of the same code base into an orthogonal set of features drawn as rows, making some of these rows optional. Figure 9c shows \( \text{EPL}_{ij} \) to be an overlay of both partitionings, yielding a 2D grid of interaction modules. Recall a module is a set of code fragments. Given module \( r \in \text{CPL}_i \) and module \( c \in \text{OPL}_j \), \( r \# c \) is the intersection of \( r \) and \( c \). The analog for \( \text{SPL}_n \) is an overlay of \( n \) partitionings.

Axis symmetries are useful: they allow us to reconstruct the SPL 1D array of each axis. \( \text{OPL}_i \) is reconstructed by summing each column of \( \text{EPL}_{ij} \):

\[
\text{OPL}_{\text{core}} = \times \text{times} \# \text{eval} \oplus \text{times} \# \text{core} \oplus \text{base} \# \text{core}
\]

\[
\text{OPL}_{\text{print}} = \times \text{times} \# \text{eval} \oplus \text{times} \# \text{print} \oplus \text{base} \# \text{print}
\]

\[
\text{OPL}_{\text{eval}} = \times \text{times} \# \text{eval} \oplus \text{times} \# \text{eval} \oplus \text{base} \# \text{eval}
\]

### Figure 8. EPL Array of 2-Dimensional Feature Modules

\[
\begin{array}{c|c|c|c}
\text{eval} & \text{print} & \text{core} \\
\hline
\text{times} & \text{times} & \text{times} \\
\hline
\text{plus} & \text{plus} & \text{plus} \\
\hline
\text{base} & \text{base} & \text{base}
\end{array}
\]

\( i \# j \) is the name of the module whose coordinates are \((i, j)\). Note: no code is associated with individual \( \text{OPL} \) or \( \text{CPL} \) features, an observation which is important in Section A. All code of \( \text{EPL}_{ij} \) belongs to the above \( i \# j \) modules.

What do these modules mean? \( i \# j \) modularizes the interaction of features \( i \) and \( j \). That is, when both \( i \) and \( j \) features are selected, the fragments in the module \( i \# j \) are added to the target program. We will see in Section V that \# is another operation on features, along with \&. We explain in Section IV-C a simple way to compute the contents of the \( \text{EPL}_{ij} \) modules.

Recall that any program in \( \text{EPL} \) is specified by two sentences: one from \( \text{CPL} \) and another from \( \text{OPL} \). In general, let \( S_{ij} \) be the module array of \( S^2 \). A program \( P \) of \( S^2 \) is uniquely specified by a set of rows \( \alpha \) and a set of columns \( \beta \). This specification enables us to infer the modules of \( S_{ij} \) to sum: module \( r \# c \) is present in \( P \) iff \( r \in \alpha \) and \( c \in \beta \). Stated differently, programs of \( \text{SPL}^2 \) (and in general, \( \text{SPL}_n \)) can be produced by the projection and contraction of its module array.

Program \( P_2 \) of \( \text{EPL} \) supports the \text{core} and \text{eval} column functionalities and the \text{base}, \text{plus}, and \text{times} row functionalities. We project (remove) the \text{print} column of \( \text{EPL}_{ij} \), yielding a 2D array of modules that comprise \( P_2 \). (These are all the interaction modules that contain fragments that implement the selected row and column features). Contracting the array produces a \( \oplus \)-expression for \( P_2 \):

\[
P_2 = \sum_{\text{EPL}_{1j}} \text{EPL}_{ij} \]

\[
EPL_{ij} = \begin{bmatrix}
\text{times} \# \text{eval} & \text{times} \# \text{print} & \text{times} \# \text{core} \\
\text{plus} \# \text{eval} & \text{plus} \# \text{print} & \text{plus} \# \text{core} \\
\text{base} \# \text{eval} & \text{base} \# \text{print} & \text{base} \# \text{core}
\end{bmatrix}
\]
or more compactly:

$$\text{OPL}_j = \sum_i \text{EPL}_{ij}$$

These identities can be seen: Figures 4 and 8 are identical except that the horizontal lines in Figure 8 are absent in Figure 4. This simply reflects the fact that we partitioned (refactored) each module of \(\text{OPL}_j\) across rows, and axis symmetry says that we can reconstruct \(\text{OPL}_j\) by undoing the partitioning through summation.

By symmetry, let \(\text{CPL}_i\) be the 1D module array listing the rows of \(\text{EPL}_{ij}\):

$$\text{CPL}_i = \begin{bmatrix} \text{times} & \text{plus} & \text{base} \end{bmatrix}$$

The values of the row modules of \(\text{CPL}_j\) can be reconstructed by summing columns:

$$\text{CPL}_i = \sum_j \text{EPL}_{ij}$$

The corresponding analogs exist for \(\text{SPL}^n\).

### D. Array Terminology

We use the language of tensors to describe and manipulate \(n\)-dimensional arrays [20]. The dimensionality of an array is its rank, which is indicated by the number of indices it has. \(A_i\) is rank 1, \(A_{i,j}\) is rank 2, \(A_{i,j,k}\) is rank 3, and so on. Array names are in upper-case and their scalar elements are in lower-case.

Arrays of higher rank are produced by taking the outer product of arrays of lesser rank. Let \(A_i = [a_1 \cdots a_n]\) and \(B_j = [b_1 \cdots b_n]\) denote a pair of rank 1 arrays. The \(\theta\) outer-product of \(A_i\) and \(B_j\) yields a 2D array, where element \((r,c)\) has the value \(a_i \theta b_c\):

$$A_i \theta B_j = \begin{bmatrix} a_1 \theta b_1 & \cdots & a_1 \theta b_n \\ \vdots & \ddots & \vdots \\ a_n \theta b_1 & \cdots & a_n \theta b_n \end{bmatrix}$$

(8)

In general, if the rank of array \(X\) is \(r\) and the rank of array \(Y\) is \(s\), the rank of \(X \theta Y\) is \(r + s\). Outer-products are associative:

$$A_i \theta (B_j \theta C_k) = (A_i \theta B_j) \theta C_k$$

(9)

The simple theorem below says that projection and outer-products commute: that is, a projection of a \(\theta\)-product equals the \(\theta\)-outer-product of a projection.

$$\sum_{i \in \mathbb{R}_{\theta \in \mathbb{C}}} A_i \theta B_j = \sum_{A \in \mathbb{R}} \min A \theta \max B$$

(10)

We use this identity in Section A3.

Observe that a product of \(A_i\) and \(B_j\) produces all combinations of \((a_i,b_j)\) elements. This is what \(\text{EPL}_{ij}\) encodes: it is the \#-outer-product of \(\text{CPL}_i\) and \(\text{OPL}_j\):

$$\text{EPL}_{ij} = \begin{cases} \text{CPL}_i \# \text{OPL}_j \\ \text{times plus base} \ # \ \text{eval print core} \\ \text{times eval times print times core} + \text{eval plus print plus core} + \text{base eval base print base core} \end{cases}$$

Like tensors, calculating array element names in \(\text{EPL}_{ij}\) is easy, but assigning values (in our case, modules) to its terms is harder. It takes time to first determine the modules of \(\text{OPL}_j\) and a bit more to determine the modules of \(\text{EPL}_{ij}\).

In general, the modules of a \(\text{SPL}^n\) are the elements of an array \(M_{i_1 \cdots i_n}\) of rank \(n\). If the \(n\) \(\text{SPL}^2\) module arrays that form its axes are \(A_{i_1} \cdots Z_{i_n}\) we know the following outer-product identity holds:

$$M_{i_1 \cdots i_n} = A_{i_1} \# \cdots \# Z_{i_n}$$

(12)

We conclude this section by formally defining a product line \(S^n\) as the ordered pair \((\hat{S}, S_{i_1 \cdots i_n})\), its feature model and its module array. \(\text{EPL}^n = (\text{EPL}, \text{EPL}_{ij})\).

### V. Feature Interactions

Feature interactions are ubiquitous. We review a classical example of interactions to motivate further discussions.

#### A. The Fire and Flood Control Problem

Let \(b\) denote the design of a building. A flood control feature adds water sensors to every floor of \(b\). If standing water is detected, the water main to \(b\) is turned off to prevent further water damage. A fire control feature adds fire sensors to every floor of \(b\). If fire is detected, sprinklers are turned on. Adding flood or fire control to the building (flood \(\oplus\) b and fire \(\oplus\) b) is straightforward. However, adding both (flood \(\oplus\) fire \(\oplus\) b) is problematic: if fire is detected, the sprinklers turn on, standing water is detected, the water main is turned off, and the building burns down. This is not the intended semantics of the composition of the flood, fire, and b features.

The fix is to include an additional module, labeled flood#fire, which is the interaction of flood and fire. That is, flood#fire represents a module of changes that are needed to make the flood and fire modules work correctly together, i.e. flood#fire modifies possibly both flood and fire modules. The correct building design is flood#fire \(\oplus\) flood \(\oplus\) fire \(\oplus\) b, where flood#fire is an interaction module.

An algebra of feature interactions was recently proposed that introduced new operations on features [13]. Besides sequential composition (\(\ominus\)), there is also the product (\(\times\)) and interaction (\(#\)) operations. The essential idea is this: When architects want features \(R\) and \(S\), they are really asking for their product, \(R \times S\), which is governed by the axiom:

$$\text{Product} : R \times S = (R \# S) \oplus R \oplus S$$

(13)
That is, architects want not only the composition of feature modules \( R \) and \( S \), but also an interaction module \( R \# S \) that modifies and/or integrates \( R \) and \( S \) so that they work together correctly. In the following sections, we review axioms of \# and \( \times \) from [13] that are relevant to our needs and that should be self-evident.

**B. Axioms of Interaction (#)**

Let \( R, S, \) and \( T \) be modules. \( R \# S \) is the module that contains the set of changes to \( R \) and \( S \) that are needed to make them work together correctly. Three axioms of \# are:

\[
\begin{align*}
\text{No Interaction} & : R \# 0 = 0 \\
\text{Commutativity} & : R \# S = S \# R \\
\text{Associativity} & : R \# (S \# T) = (R \# S) \# T
\end{align*}
\]

(14) states the elementary fact that 0 cannot be changed and that it does not change other modules. \# is commutative since the order in which modules \( R \) and \( S \) are \( \oplus \)-composed does not matter, the \( R \# S \) module that contains their repairs should not depend on their \#-composition order either. A similar argument justifies \#-associativity.

Further, interaction (#) binds stronger than summation (\( \oplus \)). Both are related by a distributivity law:

\[
\begin{align*}
\text{Distributivity I} & : R \# (S \oplus T) = (R \# S) \oplus (R \# T) \\
\text{Distributivity II} & : (S \oplus T) \# R = (S \# R) \oplus (T \# R)
\end{align*}
\]

and a more verbose version, which we use in Section A3:

\[
\begin{align*}
\text{Distributivity III} & : (r_1 \oplus \cdots \oplus r_n) \# (c_1 \oplus \cdots \oplus c_n) = (r_1 \# c_1) \oplus \cdots \oplus (r_n \# c_n)
\end{align*}
\]

**C. Axioms of Product (\( \times \))**

There is but one axiom of \( \times \), which we listed earlier (13). Three theorems can be derived from the axioms already presented:

\[
\begin{align*}
\text{Identity} & : R \times 0 = R \\
\text{Commutativity} & : R \times S = S \times R \\
\text{Associativity} & : (R \times S) \times T = R \times (S \times T)
\end{align*}
\]

The meaning of these theorems should be self-evident. Their proofs and the consistency of these axioms are given in [13].

**D. The Product of Feature Models**

Consider the following illustration of products. One of the tenets of FOSD is the application of its ideas to all program representations. This means that feature models \( CPL \) and \( OPL \) can be themselves interpreted as modules and thus can be composed. Their product equals \( EPL \):

\[
EPL = CPL \times OPL = (CPL \# OPL) \oplus CPL \oplus OPL
\]

We know the values of \( EPL, CPL, \) and \( OPL \) to be (4), (5), and (6). From this, we can determine the value of module \( CPL \# OPL \) to be the uncolored code in Figure 10, which includes \( c \) and \( o \). Variation point \( c \) is paired with \( CPL \) and \( o \) is paired with \( OPL \).

![EPL Feature Model Modules](image)

**E. Feature Models and Feature Products**

Until now, we assumed a 1-1 mapping between features and modules. In general, a feature is implemented by a sum of modules. We now revisit the connection among models, sentences, and module expressions.

Recall feature model \( \hat{M} \) from Section III-B. Assuming feature interactions arise, a sentence of this model (‘kjb’) is mapped to a module expression by multiplying its terms \((k \times j \times b)\). Interaction and product axioms reduce an expression with \( \times \) operations to an expression summing only primitive modules, as below:

\[
\begin{align*}
& k \times j \times b \quad // \text{def of p} \\
& = k \times (j\#b \oplus j \oplus b) \quad // (13) \\
& = k\#(j\#b \oplus j \oplus b) \oplus k \oplus j\#b \oplus j \oplus b \quad // (13) \\
& = k\#j\#b \oplus k\#j \oplus k\#b \oplus k \oplus j\#b \oplus j \oplus b \quad // (17)
\end{align*}
\]

Again, module \( k\#j\#b \) is primitive, like \( j \). The resulting \( \oplus \)-expression includes all possible interaction modules, which is what we want. Evaluating this expression generates the ‘kjb’ program [13].

A consequence of the interaction and product axioms is an increase in the size of module arrays. For \( M^1 \), module array \( M \) would include not only modules for individual features \( k, i, j, \) and \( b \), but also all interaction modules:

\[
M = [ k \quad i \quad j \quad b \quad k\#i \quad k\#j \quad k\#i\#j \quad \ldots ]
\]

The same holds for products of such arrays that arise in \( SPL^n \).

**F. Recap**

We conclude this section by explaining general identities for constructing \( SPL^n \). These are the new algebraic laws that this paper introduces to FOSD.

Product lines of higher rank are produced by overlaying distinct product lines of the same code base. We express the overlay operation \( \odot \) in terms of products and interactions. Let \( A^m \) and \( B^n \) be SPLs (distinct partitionings) of the same code base. Their overlay \( A^m \odot B^n \) is \( AB^{m+n} \):
AB^{n\times m} = A^n \otimes B^m
= (\vec{A}, A_{11 : \mathcal{I}, \ldots : \mathcal{J}}) \otimes (\vec{B}, B_{11 : \mathcal{I}, \ldots : \mathcal{J}})
= (\vec{A} \times \vec{B}, A_{11 : \mathcal{I}, \ldots : \mathcal{J}} \# B_{11 : \mathcal{I}, \ldots : \mathcal{J}})
(24)
= (\vec{A}, AB_{11 : \mathcal{I}, \ldots : \mathcal{J}})
(25)

That is, the overlay of two SPLs equals the product of their feature models and the #-outer-product of their arrays. Recall CPL^1, OPL^1 and EPL^2 are each defined as an ordered pair:

\[ \text{CPL}^1 = (\text{CPL}_1, \text{CPL}_1) \]
\[ \text{OPL}^1 = (\text{OPL}_1, \text{OPL}_1) \]
\[ \text{EPL}^2 = (\text{EPL}_1, \text{EPL}_1) \]

We know that \( \text{EPL} = \text{CPL} \times \text{OPL} \) from (23) and \( \text{EPL}^{i+} = \text{CPL}^{1} \# \text{OPL}^{I} \) from (11). It follows that EPL^2 is the overlay of CPL^1 and OPL^1:

\[ \text{EPL}^2 = \text{CPL}^1 \otimes \text{OPL}^1 \]
\[ = (\text{CPL}_1, \text{CPL}_1) \otimes (\text{OPL}_1, \text{OPL}_1) \]
\[ = (\text{CPL}_1 \times \text{OPL}_1, \text{CPL}_1 \# \text{OPL}_1) \]
\[ = (\text{EPL}_1, \text{EPL}_1) \]

Finally, suppose we have a set of modules that we could both (i) treat as modules of a SPL^1 and (ii) arrange into a n-D module array of a SPL^m. The specification for a program \( P \) either way would be as a set of features. In the appendix, we prove that the algebra of [13] (which knows nothing about arrays and deals only with SPL^1) and our SPL^m algebra produce equivalent \( \oplus \)-expressions for \( P \). More on this topic in Section VII.

VI. VALIDATION: THE ATS CASE STUDY

The AHEAD Tool Suite (ATS) is an example of FOSD-based product-line development [22][22]. It is a set of languages and tools that support the definition and composition of feature modules to generate programs. ATS was bootstrapped, so ATS itself is generated from a pair of SPL^2’s.

AHEAD represents features as program transformations, so that a program is defined as a composition of transformations. The order in which AHEAD transformations are composed matters. In a VP-fragment implementation, modules can also be interpreted as transformations, but with a key difference: VP-fragment transformations are commutative.

In this section, we explain the mapping of AHEAD concepts to a VP-fragment implementation, and then describe a pair of SPL^2’s from which AHEAD tools were generated [3][23].

A. Mapping AHEAD to Variation Points

The mapping of AHEAD concepts to VPs and code fragments is straightforward. Figure 11a shows a class declaration (which is identical to that in Java). Figure 11b shows a refinement of this class that introduces a boolean variable y and wraps method m(). Figure 11c displays the result of composing the original declaration with its refinement.

Here is how the same design is expressed in terms of VPs: Figure 12a shows \( \oplus \) as the place to insert fragment “@y bool y;”. \( \oplus \) has the fragment “@x x++;”. And m\( \oplus \) is paired with a wrapping fragment which wraps the fragment of x\( \oplus \). Wrapping fragments have been implemented in CIDE [24][14] and Paan [13], both of which are FOSD tools that implement VPs.

B. The JT^2 and BT^2 Product Lines of ATS

JT^2 is a family of Java-based tools (consisting of 140K LOC) in ATS and is the overlay of two rank 1 product lines, Java^1 and JTool^1. A simplified version of their feature models are:

\[ \text{JT}^2 : \text{Java, JTool} ; \]
\[ \text{Java} : \text{[ant]} \text{[smRef]} \text{[ref]} \text{[sm]} \text{common} \text{[quote]} \text{java} ; \]
\[ \text{JTool} : \text{[mixin]} \text{[jampack]} \text{[jak2java]} \text{toolcommon} ; \text{ref} \text{sm} \Rightarrow \text{smRef} ; \text{mixin} \Rightarrow \neg\text{jampack} \wedge \neg\text{jak2java} ; \text{jampack} \Rightarrow \neg\text{mixin} \wedge \neg\text{jak2java} ; \text{jak2java} \Rightarrow \neg\text{mixin} \wedge \neg\text{jampack} ; \]

Java specifies dialects of Java, whose optional features include Lisp quotes for meta-programming (quote), state machines (sm), class and interface refinements (ref), state machine refinements (smRef), and Apache ant options for tool invocation (ant). The first constraint listed above restricts combinations of these options. JTool specifies which tool in ATS to produce. All tools share a common feature (toolcommon); only one optional tool feature (mixin, jampack, or jak2java) can be specified at a time. The remaining three constraints listed above impose these
limitations. There are more tools in JTool that could be listed; but the essence of JTool is captured above.

The JTool array has a module for each Tool feature; there are no interaction modules. The total number of modules in JTool is 9 (although the simplified version of Tool suggests fewer). The Java array has a module for each of its features, plus a handful of pairwise interaction modules. The total number of modules in Java is 16 (although again Java suggests fewer). The JT array has 13 \times 9 = 117 elements, but only 84 out 117 interaction modules (or 72%) are non-zero. Projections and contractions of JT generates the Java-based tools in ATS, which produces about 140K LOC. For more details, see [23], [3].

ATS also uses BT to generate grammar manipulation tools. BT is an overlay of two rank 1 product lines, Bali and BTool. A simplified version of their feature models are:

\[
\begin{align*}
\text{BT} & : \text{Bali BTool;} \\
\text{Bali} & : [\text{require} \text{ bali}; \\
\text{BTool} & : [b2l] [\text{bcomp} [b2jcc] \text{ [codegen base];} \text{ bcomp V b2jcc \Rightarrow codegen;} \\
& \text{ b2l + bcomp} \land \neg \text{b2jcc;} \\
& \text{ bcomp} \Rightarrow \neg \text{b2l} \land \neg \text{b2jcc;} \\
& \text{b2jcc} \Rightarrow \neg \text{b2l} \land \neg \text{bcomp}; \\
\end{align*}
\]

Bali is a small SPL that specifies a dialect of the Bali language. There are two features: the core language itself (bali) and an optional requires statement (require). BTool specifies which Bali tool to produce. All Bali tools share (base), most use (codegen), and a legal BTool sentence requires one tool to be selected. All the constraints listed above belong to BTool.

Bali has 2 modules. BTool has 6 modules, one per feature in BTool (although BTool suggests fewer tools). BT is a 2 \times 6 array, with all but two of its 12 entries (83%) are non-zero. Projections and contractions of BT generates the Bali-based tools in ATS, which is about 35K LOC. For more details, see [23], [3]. An elementary AHEAD SPL, based on EPL, is presented in [12].

VII. OPEN PROBLEMS

We have answered an open question in FOSD: how do SPL arise, what is a mathematics behind them, and how can they be understood and developed. Still, fundamental problems remain. For example, when can a SPL be mapped to a SPL and vice versa? Mapping EPL to a SPL whose feature model is:

\[
\text{EPL} : [\text{times}] [\text{plus}] [\text{eval}] [\text{print}] \text{ corebase;}
\]

is not difficult. (We have done it, but do not have a general algorithm to map a SPL to a SPL). The modules are unchanged; only their names change.

However, the reverse may be different. We do not understand how a SPL can be mapped to a SPL. Moreover, it is not clear that all SPL can be mapped to a SPL. And if not, what qualifies those that can be mapped? When can a n-D array be mapped to a 1D array and vice versa? We do not know. This paper gives a clue: SPL are produced by the overlaying of n SPL partitionings of the same code base.

Multiple dimensional SPLs arise naturally in implementations and in the literature. It seems awkward and unnatural not to think of SPL in terms of n-D arrays of modules; the literature is replete with examples. Rhetorically, why would these authors use multi-dimensional arrays if it did not give them some conceptual advantage? Our experience in building SPL's, the topic of the prior section, gave us a stark answer: we initially could not understand SPL without 2D arrays. Axis symmetries played an essential role in our understanding of SPL.

Here is another problem which we have seen repeatedly: Give the same code base to different groups of people. Each group will partition the code base using different sets of features, yielding different modules. What is the relationship between these different partitionings and SPLs? Again, overlaying provides an important clue that shows how different SPL can be related. Answering these problems remain future work.

VIII. RELATED WORK

Algebras for FOSD are common [2][13][25][26]. Our work differs from prior algebras in that transformations (modules) are commutative. Another difference is the multi-dimensionality of our algebra. Feature interactions and the granularity of features are closely related [27]; this paper and [13] give a formal foundation for their study.

Axis symmetries were informally suggested in Multi-Dimensional Separation of Concerns (MDSoC) [28], where a code base could be partitioned along different axes or “dimensions”. Contracting along different axes provided different modularizations or views of this code base. Underlying MDSoC is consistent views or modularizations; if views of a common code base are inconsistent, then understanding, modifying, proving properties, etc. in one view may not hold in others, leading to development chaos. Axis symmetry is a natural way in which view consistencies are expressed. Our work is a feature-based formalization of MDSoC.

Lopez-Herrejon was the first to make the connection between the extensibility problem and software product lines [23] and giving an algebraic formulation of MDSoC. Subsequent work explored this connection both in features [3][2] and in aspects [15][29].

SPL have been recognized in industry for some time. Bühne et al. explored how commonalities and variabilities across different SPLs could be managed [17]. The refactoring that we presented in Section II is similar to the method they described. Unlike their work, we present a formalism
for $\text{SPL}^n$. Further, their goal was to explore the ability to retrieve information from integrated SPLs. In principle, this corresponds to the retrieval of a module array, which could itself be a projection and contraction of a larger array.

Thompson and Heimdahl examined the population relationships between different product lines, and explained how a multi-dimensional approach (different axes for orthogonal classifications) was useful in understanding and compactly describing the structure of a domain [9]. In particular, they explored a $\text{SPL}^2$ for robotics, where one axis encoded variations in environments, and the other encoded variations in robot behavior. No formal model (or array model) of their approach was given.

Reiser and Weber address the issue of feature model scalability in a hierarchical product line of feature models (FMs), each of which represents a distinct product line [7]. A child FM is derived from its parent or reference FM by a transformation that connects child features to parent features when possible, and permits parent features to be deleted, augmented or replaced by other child features. Dealing with large FMs is indeed an issue for $\text{SPL}^1$, and is likely to be more of a challenge for $\text{SPL}^n$. This raises an intriguing possibility of generalizing our algebra that uses arrays of scalars to arrays of scalars and/or arrays.

Variation points are among the oldest ideas in SPL design [18][30]. FOSD has historically been associated with class inheritance hierarchies, rather than VPs. Recent work in FOSD, in particular CIDE [14], has reinvigorated the use of VP implementations. Our use of VPs to explain SPLs allows module composition to be commutative. In a more general setting, where inserting code fragments into VPs is not sufficient to describe product lines, modules become more general transformations where commutativity does not apply. Future work should generalize our algebra to non-commutative module compositions.

Multiple axes of features (a.k.a. contexts) was described by Hartman and Trew [5]. Axes defined orthogonal contexts to classify products by regions, by product types, and by type of users. It is possible to select several contexts (read: features) along each axis in product specifications. The focus of their work was mapping context (axis feature) selections to define the feature model of a product line. Our work is complementary as it provides an algebraic foundation for casting such ideas.

In fact, their work provides independent supporting evidence for the consistency of our interaction and $\text{SPL}^n$ algebras. Context features in Hartman and Trew’s work have no concrete representation: it is only the interaction of context features from different axes that have concrete artifacts. (This is equivalent to our observation that modules in $R_i$ and $C_j$ in Section A3 have value 0; only interactions of modules in $R_i$, with modules in $C_j$ are non-zero). In general, modules along the axes of all $\text{SPL}^n$ are context features and their interactions; all such modules are empty (value 0). It is only their $n$-way interactions that have materializations.

IX. Conclusions

Just as programs have variations, which leads to product lines of rank 1 ($\text{SPL}^1$), product lines can themselves have variations, which leads to product lines of rank 2 ($\text{SPL}^2$), and so on. In this paper, we explained how product lines of rank $n$ ($\text{SPL}^n$) arise from an overlaying of $n$ different $\text{SPL}^1$ partitionings of a single SPL code base. Although in practice we encountered $n = 2$ and $n = 3$, larger values of $n$ are indeed possible. As our literature review revealed, SPLs are common both in theory and practice.

In this paper, new laws of FOSD construction were presented, which we summarize as follows: Let $A^n$ is a SPL of rank $n$. $A^n$ has a feature model $\hat{A}$ which defines the legal combinations of its features and an $n$-dimensional array of modules $A_{1,1,\ldots,1}$. We write this as: $A^n = (\hat{A}, A_{1,1,\ldots,1})$. Product lines of higher rank are created by overlaying $\odot$ product lines of lower rank, using the product $(\times)$ and interaction-outer-product $(\#)$ operations. Given $A^n$ and $B^m$ of the same codebase, their overlay $A^n \odot B^m$ is $AB^{n+m}$:

$$AB^{n+m} = A^n \odot B^m$$
$$= (\hat{A}, A_{1,1,\ldots,1}) \odot (\hat{B}, B_{j_1,j_2,\ldots,j_m})$$
$$= (\hat{A} \times \hat{B}, A_{1,1,\ldots,1} \odot B_{j_1,j_2,\ldots,j_m})$$
$$= (\hat{A} \times \hat{B}, AB_{1,1,\ldots,1,j_1,\ldots,j_m})$$

That is, overlaying two product lines equals the product of their feature models and the interaction-outer-product of their arrays. We proved that these laws are in agreement with a recent advance in feature interactions. Further, we validated our work by showing how existing $\text{SPL}^2$ development could be explained in its terms.

Having a formal foundation for feature-based software generation is important. It tells us how to build tools with solid conceptual foundations, it gives us strict guidelines on how to modularize domains of software to be amenable to program generation, and strengthens the belief that a mathematics-based science underlies automated software design.

Acknowledgments. This research was supported by NSFs Science of Design Project CCF 0724979. Thanks to S. Apel, C. Kästner, and P. Höfner for their helpful comments on earlier drafts of this paper.
Appendix

A. Agreement of Both Algebras

Here is a key question: suppose we have a set of modules that we could (i) simply treat as a SPL\(^1\) or (ii) arrange in as an n-D array of a SPL\(^n\). The specification for a program \(P\) for either interpretation is as a set of features. Either way, both algebras must produce equivalent \(\oplus\)-expressions (up to commutativity) for \(P\). Proving this is the goal of this section.

In the following, we focus on \(n = 2\) dimensions. We use both the \(\oplus\)-contraction of arrays and \(\times\)-contraction of arrays. The \(\times\)-contraction of module array \(A_j\) along dimension \(j\) is written \(\prod_j A_j\), or simply \(\prod A_j\) when the product of all modules in the array are to be taken. Further, let \(R^1\) denote a SPL\(^1\) of rows and \(C^1\) a SPL\(^1\) of columns. We are to compute programs in \(S^2\), where \(S^2 = R^1 \oplus C^1\).

1) With Arrays: An end-user specifies \(P\) in \(S^2\) as a set of row features \(\alpha\) and a set of column features \(\beta\) that form a sentence of \(S\). \(P\) is computed by projecting and contracting \(S_{ij}\), the rows to contract \((\text{row})\) and columns to contract \((\text{col})\) are determined by applying the interaction and product axioms:

\[
\prod_{r \in \alpha} r = \sum_{R \in \text{row}} R_{ij} \tag{26}
\]

\[
\prod_{c \in \beta} c = \sum_{C \in \text{col}} C_{ij} \tag{27}
\]

That is, a product of features can always be translated into a sum of modules. (However, the reverse is not true: not all module sums can be translated into a product of modules.)

The expression for \(P\) is given below, where \(\text{row}\) and \(\text{col}\) are defined by (26) and (27).

\[
P = \sum S_{ij} \text{row} \text{ col} \tag{28}
\]

2) Without Arrays: An end-user specifies \(P\) as the set of row features \(\alpha\) \(\cup\) \(\beta\) which forms a sentence of \(S\). The product of the selected features yields an expression for \(P\):

\[
P = \left(\prod_{r \in \alpha} r\right) \times \left(\prod_{c \in \beta} c\right) \tag{29}
\]

3) Agreement Proof: Here is a proof sketch for the equivalence of (28) and (29). We know that \(S_{ij}\) represents all modules from which \(P\) can be built. After all, \(S_{ij}\) slices and dices the code base of \(S^2\) in a grid-like manner. Modules that do not appear in that grid must have no code, and hence have value 0.

Every module in \(R^1\) is of the form \(r_{ij} \# \cdots \# r_{i,n}\), a \#-composition of one or more row features. The same for \(C^1\). But every module in \(S_{ij}\) is an interaction of an \(R^1\) module (a row) with a \(C^1\) module (a column). Modules that do not represent joint row and column interactions, such as every module in \(R^1\) and \(C^1\), must be 0.

We now rewrite (29) by the product axiom (13):

\[
P = \left(\prod_{r \in \alpha} r\right) \times \left(\prod_{c \in \beta} c\right)
\]

\[
= \left(\prod_{r \in \alpha} r\right) \# \left(\prod_{c \in \beta} c\right)
\]

The right-most two products above, \(\prod_{r \in \alpha} r\) and \(\prod_{c \in \beta} c\), reduce to a summation of modules from \(R_1\) and \(C_1\), respectively, according to (26) and (27).

From the previous paragraph, these summations are 0. Thus, if both algebras produce equivalent results, the following must be an identity:

\[
\sum S_{ij} \text{row} \text{ col} = \left(\prod_{r \in \alpha} r\right) \# \left(\prod_{c \in \beta} c\right)
\]

To prove this, we rewrite the Distributivity III axiom (19) in terms of array operations, which says the interaction of two sums equals the sum of their pairwise interactions:

\[
\left(\sum R_i\right) \# \left(\sum C_j\right) = \sum (R_i \# C_j) \tag{31}
\]

The equality that we seek follows:

\[
\sum S_{ij} \text{row} \text{ col} = \left(\prod_{r \in \alpha} r\right) \# \left(\prod_{c \in \beta} c\right)
\]

Thus, both algebras produce equivalent results. The same logic applies for agreement proofs for dimensionality \(n > 2\).

B. Extreme Examples

A key to understanding the \(\oplus\) operation is how products of feature models are defined. We illustrated a standard form for such products in earlier sections, where partitions are independent or orthogonal. But our results also apply to cases where “independence” or “orthogonality” clearly does not hold. Here are two extreme cases.

Case 1. Consider overlaying a SPL with itself. Let \(A^1\) be a product line of rank 1 (the rank need not be one, but it is simpler to visualize) and let \(B^1\) be identical to \(A^1\) except that the names of features are different. Both \(A^1\) and \(B^1\) have \(n\) features. For an overlay to make sense, \(A \times B\) would be \(A \hat{\oplus} B\):

\[
A \hat{\oplus} B : \hat{A} \hat{\oplus} \hat{B} ; \quad \hat{A} : \cdots ; \quad \hat{B} : \cdots ;
\]

\[
(A_1 \leftrightarrow B_1) \land (A_2 \leftrightarrow B_2) \land \cdots \land (A_n \leftrightarrow B_n)
\]

That is, equivalent sentences in \(A\) and \(B\) must be chosen. \(A_1 \# B_1\) is a \(n \times n\) matrix where only the diagonal has non-zeroes:

\[
A_i \# B_j = \begin{cases} 
0 & \text{if } i \neq j \\
A_i & \text{otherwise}
\end{cases}
\]

Thus, a sentence of \(A \hat{\oplus} B\) corresponds to a sentence in \(A\) (or \(B\)). Further, the module expression for each sentence in \(A \hat{\oplus} B\) is equivalent to the module expression for the corresponding sentence in \(A\) (or \(B\)).

Case 2. Again, let \(A^1\) and \(B^1\) be rank 1 SPLs of the same code base, but the relationship between \(A^1\) and \(B^1\) is not understood. Consequently, we want \(A \oplus B\) to be the union
of programs in $A^i$ and $B^j$. Array $A_{B_{ij}}$ is computed as in Section IV-C, and the product of $\hat{A} \times \hat{B}$ is $\hat{A}\hat{B}$:

$$\hat{A}\hat{B} : \hat{A} A_{1 \hat{B}}$$

$$\hat{A} : \cdots ;$$

$$A_{1 \hat{B}} : \cdots ;$$

$$\hat{B} : \hat{B} A_{\hat{1} \hat{B}} ;$$

$$\hat{A} : \cdots ;$$

$$A_{\hat{1} A} : A_1 A_2 \cdots A_n ; \text{// select all As}$$

$$\hat{B} : \cdots ;$$

$$A_{\hat{B}}: B_1 B_2 \cdots B_m ; \text{// select all Bs}$$

That is, the language of $\hat{A}\hat{B}$ is the union of the $\hat{A}$ and $\hat{B}$ languages. The 2D array $A_{B_{ij}}$ is computed exactly as described in Section IV-C.

REFERENCES


