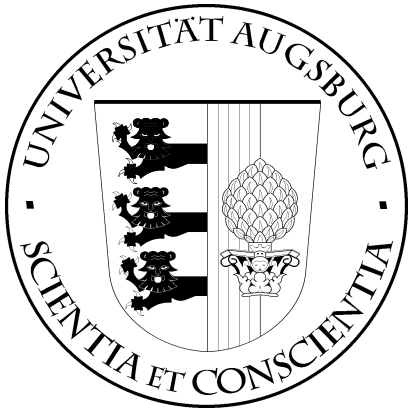
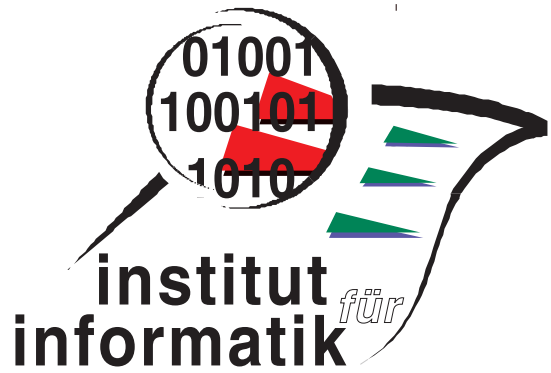
Universität Augsburg



Feature Algebra

P. Höfner **R. Khedri** **B. Möller**

Report 2006-04 June 2006



Institut für Informatik
D-86135 Augsburg

Copyright © P. Höfner R. Khedri B. Möller
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

Feature Algebra*

Peter Höfner[1]**, Ridha Khedri[2]***, and Bernhard Möller[1]

[1] Institut für Informatik, Universität Augsburg
D-86135 Augsburg, Germany
{hoefner,moeller}@informatik.uni-augsburg.de
[2] Department of Computing and Software, McMaster University
Canada L8S 4L7, Hamilton, Ontario
khedri@mcmaster.ca

Abstract. Based on experience from the hardware industry, *product families* have entered the software development process as well, since software developers often prefer not to build a single product but rather a family of similar products that share at least one common functionality while having well-identified variabilities. Such shared commonalities, also called *features*, reach from common hardware parts to software artefacts such as requirements, architectural properties, components, middleware, or code. We use idempotent semirings as the basis for a *feature algebra* that allows a formal treatment of the above notions as well as calculations with them. In particular models of feature algebra the elements are sets of products, i.e. product families. We extend the algebra to cover product lines, refinement, product development and product classification. Finally we briefly describe a prototype implementation of one particular model.

1 Introduction

Software development models relate, in general, to the development of single software systems from the requirements stage to the maintenance one. This classical method of developing software is described in [12] as *sequential completion*. There, a particular system is developed completely to the delivery stage; only after that similar systems are developed by keeping large parts of the working system and changing relatively small parts of it. Contrarily, in [12], Parnas introduces the notion of *program family* and defines it as follows:

> "We consider a set of programs to constitute a *family*, whenever it is worthwhile to study programs from the set by *first* studying the common properties of the set and *then* determining the special properties of the individual family members."

* Abridged version to appear in T. Nipkow, J. Misra (eds.): FM 2006. LNCS, 2006
** This research was supported by DFG (German Research Foundation)
*** This research was supported by Natural Sciences and Engineering Research Council of Canada

Parnas also proposes a design process for the concurrent development of the members of a program family. Since his paper [12], the notion of product family has gained a lot of attention and has found its way into the software development process in industry [13]. Indeed, software developers that are pressured by the increase in the speed of time-to-market and the necessity of launching new products do not build a single product but a family of similar products that share at least one common functionality and have well identified variabilities [5]. Their goal is to target many market segments or domains. Also, in the competitive market of today, they cannot afford to decline a request from a client who wants a special variant that is slightly different from the company's other products. In this situation, the company would have advantage in gathering the requirements for and designing families of software systems instead of single software systems. For example, in embedded system development, software depends on hardware and the developer needs to change software specifications frequently because of hardware specification changes. Hence, the developer ends up with many variations of the intended system that need to be managed. Prioritising development tasks and planning them become very challenging. A model that helps to capture the variabilities and commonalities of the members of a system family would be very helpful in dealing with these difficulties.

The concept of software product family comes from the hardware industry. There, hardware product lines allow manufacturing several variants of products, which leads to a significant reduction of operational costs. The paradigm of *product line* has been transferred to the software embedded in the products. To cope with a large number of software variants needed by an industrial product line, the software industry has been organising its software assets in software product families [13]. Hence, plainly, a *product family* can be defined as a set of products that share common hardware or software artefacts such as requirements, architectural properties, components, middleware, or code. In the remainder, we denote by *feature* any of these artefacts. We note that, according to [14], a feature is a conceptual characteristic that is visible to stakeholders (e.g., users, customers, developers, managers, etc.). A subfamily of a family F is a subset whose elements share more features than are shared by all the members of F. Sometimes, for practical reasons, a specific software subfamily is called a *product line*. For instance, in a context of software development based on the family approach, a subfamily is called a *product line* when its members have a common managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of a core assets in a prescribed way [5, 14]. Therefore, factors other than the structure of the members of a family are involved in defining a product line.

The family-oriented software development is based on the assumption that it is possible to predict the changes that are likely to be performed on a system. This assumption is true in most of the cases. For instance, the manufacturers of robots (and their embedded software) know from the start that customers will want to have robots with several basic means of locomotion, such as treads,

wheels, or legs and with several navigation systems which are more or less sophisticated.

The aim of the present paper is to underpin these ideas with a formalism that allows a mathematically precise description of product families as well as calculations with them. To this end we propose an algebra that we use to describe and analyse the commonalities and variabilities of a system family.

Since systems are characterised by their features, we call our approach *feature algebra*. We will present models where elements of feature algebras are sets of products, i.e. product families. Starting from idempotent semirings, we will define feature algebra in Section 4, extend it to cover product lines, refinements, product development and product classification. This approach allows compact and precise algebraic manipulations and calculations on these structures.

2 Literature Review

In the literature, we find several feature-driven processes for the development of software system families that propose models to describe the commonalities and variabilities of a system family. For brevity, we focus on the key processes relevant to the family description technique that we propose: Feature-Oriented Domain Analysis (FODA) [9], Feature-Oriented Reuse Method (FORM) [10], Featured Reuse-Driven Software Engineering Business (FeatuRSEB) [8] and Generative Programming (GP) [6]. The reader can find other feature modelling techniques in [2].

FODA uses feature models which are the means to give the mandatory, optional and alternative concepts within a domain [9, 14]. For example, in a car, we have a transmission system as a mandatory feature, and an air conditioning as an optional feature. However, the transmission system can either be manual or automatic. These two feature-options (manual and automatic) are said to be alternative features. The part of the FODA feature model most related to our work is the *feature diagram*. It constitutes a tree of features and captures the above relationships (i.e., mandatory, optional, and alternative) among features.

In [14], the authors propose the use of feature diagrams which are trees. Each feature may be annotated with a weight giving a kind of priority assigned to it. Then, they use basic concepts of fuzzy set theory to model variability in software product lines.

FORM starts with an analysis of commonalities among applications in a particular domain in terms of services, operating environments, domain technologies and implementation techniques. Then a model called *feature model* is constructed to capture commonalities as an AND/OR graph [11, pages 40-41& 99-100]. The AND nodes in this graph indicate mandatory features and OR nodes indicate alternative features selectable for different applications. The model is then used to derive parameterised reference architectures and appropriate reusable components instantiable during application development [10].

In FeatuRSEB, the feature model is represented by a graph (not necessary a tree) of features. The edges are mainly UML dependence relationships: *com-*

posed_of, *optional_feature* and *alternative_relationship*. The graph enables to specify the *requires* and *mutual exclusion* constraints. The feature model in FeatuRSEB can be seen as an improvement of the model of FODA.

GP is a software engineering paradigm based on modelling of software system families. Its feature modelling aims to capture commonalities and variation points within the family. A feature model is represented by a hierarchically arranged diagram where a parent feature is composed of a combination of some or all of its children. A vertex parent feature and its children in this diagram can have one of the following relationships [6]:

- And: indicates that all children must be considered in the composition of the parent feature;
- Alternative: indicates that only one child forms the parent feature;
- Or: indicates that one or more children features can be involved in the composition of the parent feature (a cardinality (n, m) can be added where n gives a minimum number of features and m gives the maximum number of features that can compose the parent);
- Mandatory: indicates that children features are required;
- Optional: indicates that children features are optional.

3 Example of a Simple Product Family

The following example is adapted from a case study given in [4]. An electronic company might have a family of three product lines: mp3 Players, DVD Players and Hard Disk Recorders. Table 1 presents the commonalities and the variability of this family. All its members share the list of features given in the *Commonalities* column. A member can have some mandatory features and might have some optional features that another member of the same product line lacks. For instance, we can have a *DVD Player* that is able to play music CDs while another does not have this feature. However, all the DVD players of the *DVD Player* product line must have the *Play DVD* feature. Also, it is possible to have a DVD player that is able to play several DVDs simultaneously.

We see that there are at least two different models of DVD players described. But how many different models are described in Table 1? And what are the properties/features of these products? Later on we will give the answer to these two questions. If we had a model which gives us all combinations of features we would be able to build new products. Vice versa, such a model would allow us to calculate commonalities of a given set of products.

4 Algebraic Structure and Basic Properties

In this section we introduce the algebraic structure of feature algebra. Since it is based on semirings we will first present these. Afterwards, we will define product families, feature algebra, a refinement relation on feature algebra and, in a set based model, features and products. In Section 6 the latter two are defined in general.

Table 1. Commonalities and variability of a set of product lines

Product line	Mandatory	Optional	Commonalities
mp3 Player	– Play mp3 files	– Record mp3 files	– Audio equaliser – Video algorithms for DVD players and hard disk recorders – Dolby surround (advanced audio features)
DVD Player	– Play DVD	– Play music CD – View pictures from picture CD – Burn CD – Play n additional DVDs at the same time	
Hard Disk Recorder		– mp3 player – organise mp3 files	

Definition 4.1 A *semiring* is a quintuple $(S, +, 0, \cdot, 1)$ such that $(S, +, 0)$ is a commutative monoid and $(S, \cdot, 1)$ is a monoid such that $\cdot$ distributes over $+$ and 0 is an annihilator, i.e., $0 \cdot a = 0 = a \cdot 0$. The semiring is *commutative* if $\cdot$ is commutative and it is *idempotent* if $+$ is idempotent, i.e., $a + a = a$. In the latter case the relation $a \leq b \Leftrightarrow_{df} a + b = b$ is a partial order, i.e., a reflexive, antisymmetric and transitive relation, called the *natural order* on S. It has 0 as its least element. Moreover, $+$ and $\cdot$ are isotone with respect to $\leq$.

In our current context, $+$ can be interpreted as a choice between optionalities of products and features and $\cdot$ as their composition or mandatory presence. An important example of an idempotent (but not commutative) semiring is REL, the algebra of binary relations over a set under relational composition. More details about (idempotent) semirings and examples of their relevance to computer science can be found,e.g., in [7].

For abbreviation and to handle the given case studies, we call an idempotent commutative semiring a *feature algebra*. Its elements are termed *product families* and can be considered as abstractly representing sets of products each of which is composed of a number of features. On every feature algebra we can define a relation that expresses that one product family refines another in a certain sense.

Example 4.2 Let $\mathbb{F}$ be a set of arbitrary elements that we call *features*. Often, features can be seen as basic properties of products. Therefore we call a collection (set) of features a *product*. The set of all possible products is $\mathbb{P} =_{df} \mathcal{P}(\mathbb{F})$, the power set or set of all subsets of $\mathbb{F}$. A collection of products (an element of $\mathcal{P}(\mathbb{P})$) is called *product family*. Note that according to this general definition the members of a product family need not have common features. Commonalities will be discussed in Section 6.

For example, looking at the DVD example of Table 1, an mp3 player is a product with the features 'play mp3 files', 'record mp3 files', 'audio visualiser' and so on.

We use the following abbreviations:

Abbreviations:	
p_mp3	Play mp3 files
r_mp3	Record mp3 files
c_1	Audio equaliser
c_2	Video algorithms
c_3	Dolby surround

Now we can describe the mp3 players algebraically as

$$mp3_player \;=\; p_mp3 \cdot (r_mp3 + 1) \cdot c_1 \cdot c_2 \cdot c_3 \,.$$

Here $1 = \{\emptyset\}$ denotes the family consisting just of the empty product that has no features, so that $(r_mp3 + 1)$ expresses optionality of r_mp3. For clarity the algebraic notation omits the set brackets.

We now formally define the operation $\cdot$ which is a composition or a merging operator for all features:

$$\begin{aligned} \cdot : \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P}) &\to \mathcal{P}(\mathbb{P}) \\ P \cdot Q \;&=\; \{p \cup q \;:\; p \in P, q \in Q\} \,. \end{aligned}$$

The second operation $+$ offers a choice between products of different product families:

$$\begin{aligned} + : \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P}) &\to \mathcal{P}(\mathbb{P}) \\ P + Q \;&=\; P \cup Q \,, \end{aligned}$$

With these definitions the structure

$$\mathbb{P}\mathrm{FS} \;=_{df}\; (\mathcal{P}(\mathbb{P}), +, \emptyset, \cdot, \{\emptyset\})$$

forms a feature algebra called *product family algebra*. The set-based model does not allow multiple occurrences of the same feature in a product. If this is desired, one can use an analogous model that employs multisets (also called bags) of features. This bag-based model is denoted by $\mathbb{P}\mathrm{FB}$. □

Using feature algebra offers abstraction from set-theory. On the one hand it provides a common structure that subsumes $\mathbb{P}\mathrm{FB}$ and $\mathbb{P}\mathrm{FS}$ and on the other hand it avoids many set-theoretic notations, like accumulations of braces, and emphasises the relevant aspects like commonalities.

The *refinement relation* $\sqsubseteq$ on a feature algebra is defined as

$$a \sqsubseteq b \;\Leftrightarrow_{df}\; \exists\, c \;:\; a \leq b \cdot c \;.$$

As an example we use again the DVD product line. A standard mp3-player that can only play mp3 files is refined by a mp3-recorder that can play and record mp3 files. In the algebraic setting this behaviour is expressed by

$$p_mp3 \cdot r_mp3 \cdot c_1 \cdot c_2 \cdot c_3 \sqsubseteq p_mp3 \cdot c_1 \cdot c_2 \cdot c_3 \;.$$

It is easy to see that the refinement relation is a preorder, i.e., a reflexive and transitive relation. Informally, $a \sqsubseteq b$ means that every product in a has at least all the features of some product in b, but possibly additional ones.

Further examples for feature algebras are all lattices with join as $+$ and meet as $\cdot$ operation. In this case the refinement relation is the same as the natural order (which coincides with the lattice order).

Until now we have not made use of the commutativity of multiplication. Most of the following basic properties hold only if $\cdot$ is commutative. In the context of our case studies and the corresponding algebras IPFS and IPFB the commutativity is significant, since products and product families should not depend on the ordering of features.

Lemma 4.3 *Let a, b, c be elements of a feature algebra, then we have*

$$a \leq b \Rightarrow a \sqsubseteq b \;, \tag{1}$$
$$a \cdot b \sqsubseteq b \;, \tag{2}$$
$$a \sqsubseteq a + b \;, \tag{3}$$
$$a \sqsubseteq b \Rightarrow a + c \sqsubseteq b + c \;, \tag{4}$$
$$a \sqsubseteq b \Rightarrow a \cdot c \sqsubseteq b \cdot c \;, \tag{5}$$
$$a \sqsubseteq 0 \Leftrightarrow a \leq 0 \;, \tag{6}$$
$$0 \sqsubseteq a \sqsubseteq 1 \;. \tag{7}$$

Proof. (1) Set $c = 1$ in the definition of $\sqsubseteq$.
(2) $a \cdot b \sqsubseteq b \Leftrightarrow \exists c : a \cdot b \leq b \cdot c \Leftarrow a \cdot b \leq b \cdot a \Leftrightarrow \mathsf{true}$.
The last step only holds if $\cdot$ is commutative.
(3) Immediate from $a \leq a + b$ and (1).
(4) Suppose $a \sqsubseteq b$, say $a \leq b \cdot d$. Then by isotony

$$a + c \leq b \cdot d + c \leq b \cdot d + c + c \cdot d + b = (b + c) \cdot (d + 1) \;,$$

i.e., $a + c \sqsubseteq b + c$.
(5) By definition, isotony w.r.t. $\leq$ and commutativity we get
$a \sqsubseteq b \Leftrightarrow \exists d : a \leq b \cdot d \Rightarrow \exists d : a \cdot c \leq b \cdot c \cdot d \Leftrightarrow a \cdot c \sqsubseteq b \cdot c$.
(6) By annihilation, $a \sqsubseteq 0 \Leftrightarrow \exists c : a \leq 0 \cdot c \Leftrightarrow a \leq 0$.
(7) Set $a = 0$ and $b = 1$, resp., in (2). □

In IPFS and IPFB, (2) describes the situation that adding features (multiplying by an element in our algebra) refines products. (3) offers an alternative product on the right hand side. So we have a choice. But this does not affect that a refines itself ($a \sqsubseteq a$). (4) and (5) are standard isotony laws. (7) says that the empty set of products 0 refines all families — all its products indeed have at least as many features as some product in a. Moreover, (7) reflects that the product without any features (which is represented by 1) is refined by any family.

Lemma 4.4 *If a feature algebra contains $a \leq$-greatest element $\top$, we have*

$$a \sqsubseteq b \Leftrightarrow a \leq b \cdot \top \Leftrightarrow a \cdot \top \leq b \cdot \top \;.$$

Proof. First we show $a \sqsubseteq b \Leftrightarrow a \leq b \cdot \top$.

($\Rightarrow$) $a \sqsubseteq b \Leftrightarrow \exists c : a \leq b \cdot c \Rightarrow a \leq b \cdot \top$.
($\Leftarrow$) Set $c = \top$.

Now, we show $a \leq b \cdot \top \Leftrightarrow a \cdot \top \leq b \cdot \top$.

($\Leftarrow$) By isotony and $a \leq a \cdot \top$.
($\Rightarrow$) By isotony and $\top \cdot \top = \top$ (which follows by $\top \cdot \top \leq \top$). □

E.g., in IPFS the greatest element is $\mathcal{P}(\mathbb{P})$, whereas in IPFB there is no greatest element.

As already mentioned, $\sqsubseteq$ is a preorder. We now show that $\sqsubseteq$ forms a partial order only in a very special case.

Lemma 4.5 *$\sqsubseteq$ is antisymmetric if and only if it is the identity relation, i.e., iff $a \sqsubseteq b \Rightarrow a = b$.*

Proof. First, the identity relation clearly is antisymmetric.

Now suppose that $\sqsubseteq$ is antisymmetric and assume $a \sqsubseteq b$. Then by isotony (4) and idempotence of $+$ we get $a + b \sqsubseteq b$. By (3) we also have $b \sqsubseteq a + b$. Now antisymmetry shows $a = b$. □

As the last property of $\sqsubseteq$, we show that the choice operator can be split w.r.t. $\sqsubseteq$ or, in other words, that $+$ produces a supremum w.r.t. $\sqsubseteq$ as well.

Lemma 4.6

$$a + b \sqsubseteq c \Leftrightarrow a \sqsubseteq c \wedge b \sqsubseteq c\,.$$

Proof.

($\Rightarrow$) By the definition of $\sqsubseteq$, lattice algebra and the definition again
$a+b \sqsubseteq c \Leftrightarrow \exists d : a+b \leq c \cdot d \Leftrightarrow \exists d : a \leq c \cdot d \wedge b \leq c \cdot d \Rightarrow a \sqsubseteq c \wedge b \sqsubseteq c$.
($\Leftarrow$) By isotony and distributivity
$a \leq c \cdot d \wedge b \leq c \cdot e \Rightarrow a + b \leq c \cdot d + c \cdot e = c \cdot (d + e)$.
Hence, $a \sqsubseteq c \wedge b \sqsubseteq c \Rightarrow a + b \sqsubseteq c$. □

5 Example of a More Complex Product Family

Our next case study is borrowed from [15] where it is used to illustrate a set-theoretic approach to reasoning about domains of what is called *n-dimensional and hierarchical* product families. It consists of a product family of mobile robots that reflect different hardware platforms and several different behaviours. The robot family is constructed using two hardware platforms: a *Pioneer* platform and a *logo-bot* platform. The behaviour of the robots ranges from a random exploration of an area to a more or less sophisticated navigation inside an area that is cluttered with obstacles. More details about the case study can be found in Thompson et al. [16], where the platforms are thoroughly described.

Similar to Thompson et al. [16] we present the members of the family according to two perspectives: hardware and behaviour. Table 2 gives a description of the robot family from a hardware perspective and Table 3 describes it from a behaviour perspective.

As described in Table 2, the robot family includes three product lines: Basic Platform, Enhanced Obstacle Detection and Environmental Vision. All the members of the Basic Platform product line share the following features:

- basic means of locomotion that could be *treads*, *wheels*, or *legs*;
- ability to turn an angle α from the initial heading;
- ability to move forward;
- ability to move backward;
- ability to stay inactive.

The variability among the members of a product line is due in part to the use of a variety of hardware. For instance, if we take the robotic collision sensors that protect robots from being damaged when they approach an obstruction or contact, then we obtain members with different sensing technologies. In our case, there are three main methods to sense contact with an obstruction: pneumatic, mechanical and a combination of mechanical and pneumatic. A member of the Basic Platform can have more than one collision sensor. The sensors could be of different types. The optional features of the members of *Basic Platform* product line are given in the third column of Table 2.

The DVD example of Section 3 was chosen for its simplicity to illustrate basic notions. The present example illustrates a family of products that exposes a more sophisticated structure of its subfamilies. It emphasises the fact that products can be defined from more than one perspective. Within a given perspective, subfamilies are defined based on other subfamilies. For instance, in the robot example the subfamily *Enhanced Obstacle Detection* is constructed on top of *basic platform* subfamily. The specification of the robot family as described in Table 2 can be found in Appendix A.

Table 2. Commonalities and variability of a robot family (a hardware perspective)

Product line	Mandatory	Optional	Commonalities
Basic Platform		– Speed of locomotion - Limited to low speed of locomotion - Extended to high speed of locomotion – Locomotion control system - Basic control (only *on* or *off*) - Digital valued indication of locomotion speed and direction – Platform size - Small - Medium - Large – Type of collision sensors - Pneumatic - Mechanical - Combination of mechanical and pneumatic – Number of collision sensors - between 0 and 3 for a small platform - between 0 and 7 for a medium platform - between 0 and 11 for a large platform	– Basic means of locomotion that could be *treads*, *wheels*, or *legs* – Ability to turn an angle α from the initial heading – Ability to move forward – Ability to move backward – Ability to stay inactive
Enhanced Obstacle Detection	– Basic Platform with at least ONE collision sensor	– Type of range finder - Small Ultrasonic Range Finder - Low-cost Ultrasonic Ranger - Compact High Performance Ultrasonic Ranger – Number of range finders - between 0 and 1 for a small platform - between 0 and 2 for a medium platform - between 0 and 3 for a large platform	– One range finder
Environmental Vision	– Enhanced Obstacle Detection	– Environmental vision system - Back and white vision - Primary colors vision	– Sensor capable of determining the color of objects in the robot's environment

Table 3. Commonalities and variability of a robot family (a behavior perspective)

Product line	Mandatory	Optional	Commonalities
Random Exploration	– Attempt to avoid colliding with obstacles	– Obstacle detection for treads-mode of locomotion – Obstacle detection for wheels-mode of locomotion – Obstacle detection for legs-mode of locomotion – Two successive collision recovery (can tolerate another collision during the recovery from a previous collision) – Three successive collision recovery – Dictated normal behavior in the absence of an obstacle	– Avoid colliding – First collision recovery – Normal behavior in the absence of an obstacle, collision, or any other specified behavior (move forward at maximum speed)
Random Exploration with the Ability to Negotiate Doors	– Random Exploration – Navigate through doors	– Small platform navigation through a door – Medium platform navigation through a door – Large platform navigation through a door	– Locate doors in its environment
Random Exploration with the Ability to Signal when it Encounters Objects of a Particular Color	– Random Exploration – Environmental Vision		– Detect an object of a specified color

6 Further Notions and Properties

In the literature, terms like product family and subfamily are used without any exact definition. Therefore, we want to make these terms formally precise. Already in Section 4 we have defined some notions like feature and product in the special models of $\mathbb{P}$FS and $\mathbb{P}$FB, however, in terms of these particular models and not in general algebraic terms. In the remainder let $F = (S, +, 0, \cdot, 1)$ be a feature algebra.

Definition 6.1 An element a is said to be a *product*, if $a \neq 0$ and

$$\forall\, b : b \leq a \;\Rightarrow\; b = 0 \vee b = a \quad \wedge \quad \forall\, b, c : a \leq b + c \;\Rightarrow\; (a \leq b \vee a \leq c)\ . \tag{8}$$

The set of all products is denoted by $\mathbb{P}$.

Intuitively, this means that a product cannot be split using the choice operator $+$. In $\mathbb{P}$FS and $\mathbb{P}$FB an element is a product iff it contains only one element, i.e., it is a singleton set.

In Example 4.2 we have also given a definition of features for the concrete case. Again we want to give an abstract algebraic counterpart. Analogously to Definition 6.1, we ask for indecomposability, but this time w.r.t. multiplication rather than addition.

Definition 6.2 An element a is called *feature* if it is a product and

$$\forall\, b : b \,|\, a \;\Rightarrow\; b = 0 \vee b = a \quad \wedge \quad \forall\, b, c : a \,|\, (b \cdot c) \;\Rightarrow\; (a \,|\, b \vee a \,|\, c)\ , \tag{9}$$

where the divisibility relation $|$ is given by $x \,|\, y \;\Leftrightarrow_{df}\; \exists\, z : x = y \cdot z$. The set of all features is denoted by $\mathbb{F}$.

From the mathematical point of view, the characteristics of products (8) and features (9) are similar and well known. We give a uniform treatment of both notions in Appendices B and C, where we also discuss the order-theoretic background.

As a special kind of products we have the *generated products* (for short g$\mathbb{P}$), i.e., those products that are obtained by multiplication of features:

$$\mathrm{g}\mathbb{P} \;=_{df}\; \mathbb{P} \cap \mathbb{F}^*,$$

where $\mathbb{F}^* \;=_{df}\; \Big\{ \prod_{i=1}^{n} x_i : n \in \mathbb{N}, x_i \in \mathbb{F} \Big\}$ is the set of all elements that arise by multiplying an arbitrary finite number of features. Over a finite set $\mathbb{F}$ of features, in $\mathbb{P}$FS as well as in $\mathbb{P}$FB the set of generated products is equal to the set of all products, i.e., $\mathrm{g}\mathbb{P} = \mathbb{P}$.

Definition 6.3 A *product family* or *family* ($\mathbb{I}$PFam) is a set of generated products that have at least one common feature, i.e,

$$a \in \mathbb{I}\text{PFam} \Leftrightarrow \exists\, f \in \mathbb{IF} : \exists\, I \subseteq \text{g}\mathbb{IP} : a = f \cdot \sum_{x_i \in I} x_i\,.$$

We call b a *subfamily* of a iff $b \leq a$.

Of course, the family a may have more common features than just f; they could be extracted from the sum by distributivity. But in our definition we wanted to emphasise that there is *at least one*. It is obvious that each subfamily of a forms a family again, since it has f as a common feature.

Sometimes, for practical reasons, a specific subfamily is called a *product line*. For instance, in a context of software development based on the family approach, a subfamily that needs to be developed in the same production site or by the same development team is called a product line. Therefore, factors other than the structure of its members can be involved in defining a product line.

To get a measure for *similarity* we give the following definitions:

Definition 6.4 Let $k \in \mathbb{N}$. The family f_1 is said to be k-*near* the family f_2, if

$$\exists\, g \neq 0 : \exists x, y \in \mathbb{IF}^{\leq k} : x \neq y \wedge f_1 = x \cdot g \wedge f_2 = y \cdot g,$$

where $\mathbb{IF}^{\leq k} =_{df} \big\{ \prod_{i=1}^{n} x_i : k \in \mathbb{N}, n \leq k, x_i \in \mathbb{IF} \big\}$.

Since every product is also a product family (which has only one member), we also have a notion for measure similarity of products. In particular, each product of a family is at least 1-near any other element of the same family (they have the common feature f).

Finally, we discuss the case of a finite set of features $\mathbb{IF}$. Then we have an additional special element in $\mathbb{I}$PFS, which is characterised by

$$\mathit{\Pi} =_{df} \Big\{ \big\{ \prod_{x_i \in \mathbb{IF}} x_i \big\} \Big\}\,.$$

This element contains only one product, namely the product that has all possible features. In this case we have $a \cdot \mathit{\Pi} = \mathit{\Pi}$ if $a \neq 0$. Then, by setting $c = \mathit{\Pi}$ in the definition of the refinement relation $\sqsubseteq$ Section 4)

$$\mathit{\Pi} \sqsubseteq a\,.$$

In general, we call an element $p \neq 0$ satisfying, for all $a \in S \backslash \{0\}$, $a \cdot p = p$ ($= p \cdot a$ by commutativity) a *weak zero*, since it annihilates *almost* all elements.

Lemma 6.5 *(i) A weak zero is unique if it exists.*
(ii) A weak zero p refines everything except 0, i.e., $p \sqsubseteq a \Leftrightarrow a \neq 0$.
(iii) If p is a weak zero then $a \sqsubseteq p \Leftrightarrow a \leq p$.

Proof. (i) Assume p and q to be weak zeros. Then, by definition, $p = p \cdot q = q$.
(ii)($\Rightarrow$) Assume $a = 0$. Then by definition of weak zero and annihilation $p = 0$, which contradicts the definition of p.
($\Leftarrow$) By definition $p \leq p \cdot a$ if $a \neq 0$ and hence, $p \sqsubseteq a$.
(iii) By definition of $\sqsubseteq$ and weak zero,
$a \sqsubseteq p \Leftrightarrow \exists\, c : a \leq p \cdot c \Leftrightarrow a \leq 0 \vee a \leq p \Leftrightarrow a \leq p$. □

Note that in IPFB there is no weak zero, since multiple occurrences of features are allowed.

7 Building Product Families and Generating Product Lines

In this section we present some useful properties of feature algebras concerning finding common features, building up product families, finding new products and excluding special feature combinations.

We first address the issue of finding the commonalities of a given set of products. This is a very relevant issue since the identification of common artifacts within systems (e.g. chips, software modules, etc.) enhances hardware/software reuse. If we look at feature algebras like IPFS and IPFB we can formalise this problem as finding "the greatest common divisor" or to factor out the features common to all given products. This relation to "classical" algorithms again shows an advantage of using an algebraic approach. Solving gcd (greatest common divisor) is well known and easy, whereas finding commonalities using diagrams (e.g., FODA) or trees (e.g., FORM) is more complex.

Example 7.1 Resuming the product line of Section 3 and Example 4.2, we give an explicit example. Assume two different products: An mp3-player defined as

$$p_mp3 \cdot c_1 \cdot c_2 \cdot c_3$$

and an mp3-recorder given by

$$p_mp3 \cdot r_mp3 \cdot c_1 \cdot c_2 \ .$$

To find all common parts we look at the sum of the two products, i.e., we create a set of products, and by simple calculations using distributivity we get

$$p_mp3 \cdot c_1 \cdot c_2 \cdot (c_3 + r_mp3) \ .$$

Thus the common parts are p_mp3, c_1 and c_2. □

Such calculations can easily done by a program; we will briefly describe a prototype in the next section. Of course one can calculate the common parts of any set of products. If there is at least one common feature, all the products form a product family. After factoring out the common parts, we can iterate this procedure for a subset of the given products and find again common parts. In this way

we can form *subproduct families* if necessary. Hence, using the algebraic rules in different directions, we can both structure and generate product families and product lines.

Starting with a set of features, we can create new products just by combining these features in all possible ways. This can easily be automated. For example, using our prototype which is described in Section 8, we calculate that the *Basic Platform* subfamily consists of 13635 products.

However, there are products with combinations of features that are impossible or undesirable. For example, it is unreasonable to have a robot that has both wheels and legs as basic means of locomotion. This requirement can be coded in feature algebra by postulating the additional equation

$$wheels \cdot legs \;=\; 0 \;.$$

This exclusion property is also implemented in our prototype. For the robot example we also exclude combinations of impossible or undesirable features (see next section) from the *Basic Platform* subfamily and are left with 1539 products.

There are many other properties like:

"If a product has feature f_1 it also needs to have feature f_2".

Most of these requirements can easily be modelled and implemented using our algebra.

8 A Prototype Implementation in Haskell

To check the adequacy of our definitions we have written a prototype implementation of the PFB model[1] in the functional programming language *Haskell*. Features are simply encoded as strings. Bags are represented as ordered lists and $\cdot$ as bag union by merging. Sets of bags are implemented as repetition-free ordered lists and $+$ as repetition-removing merge.

This prototype can normalise algebraic expressions over features into a sum-of-products-form. A small pretty-printing facility allows us to display the results as the sequence of all products described by such an expression.

As an example we give the code corresponding to Table 1 of Section 3.

```
-- basic features:
p_mp3 = bf "play mp3-files"
r_mp3 = bf "record mp3-files"
o_mp3 = bf "organise mp3-files"
p_dvd = bf "play DVD"
p_cd  = bf "play CD"
v_cd  = bf "view picture CD"
b_cd  = bf "burn CD"
a_cd  = bf "play additional CD"
```

[1] The program and a short description can be found at: http://www.informatik.uni-augsburg.de/lehrstuehle/dbis/pmi/publications/all_pmi_tech-reports .

```
c1    = bf "audio equaliser"
c2    = bf "video algorithms"
c3    = bf "dolby surround"

-- composed features
mp3_player = p_mp3 .*. (opt [r_mp3])
dvd_player = p_dvd .*. (opt [p_cd , v_cd , b_cd , a_cd])
hd  = opt [mp3_player, o_mp3]

--whole product line
p_line = c1 .*. c2 .*. c3 .*. (mp3_player .+. dvd_player .+. hd)
```

The product line contains 22 products, printed out as follows:

```
========================================================================
              Common Parts
------------------------------------------------------------------------
audio equaliser
dolby surround
video algorithms
------------------------------------------------------------------------
========================================================================
              Variabilities
------------------------------------------------------------------------
burn CD
play CD
play DVD
play additional CD
------------------------------------------------------------------------
burn CD
play CD
play DVD
play additional CD
view picture CD
------------------------------------------------------------------------
burn CD
...
```

Feature exclusion as discussed in the previous section, can be also encoded using an algebraic expression. For instance, all the required exclusion properties of the robot example are given by

```
excludes =     treads .*. wheels
           .+. treads .*.  legs
           .+. wheels .*. legs
           .+. limited_spd   .*. extended_spd
           .+. basic_ctrl    .*. digital_ctrl
           .+. small_pltfrm  .*. large_pltfrm
           .+. medium_pltfrm .*. large_pltfrm
           .+. small_pltfrm  .*. c_sensor .^. 4
```

```
.+. medium_pltfrm .*. c_sensor .^. 5
.+. large_pltfrm  .*. c_sensor .^. 6
```

Here ^ is the exponentiation operator. Due to the fact that 0 is an annihilator for $\cdot$, the last line excludes large platforms with more than 5 collision sensors.

9 Conclusion and Outlook

The adoption of the product family paradigm in software development aims at recognising a reality in software development industry noticed decades ago [12]: economical constraints impose a concurrent approach to software development replacing the early sequential one. The research work about software product families aims at studying the commonalities/variability occurring among the products in order to have a better management of software production. However, a review of the literature reveals a wide set of notions and terms used without formal definitions. A clear and simple mathematical setting for the usage of this paradigm arises as a necessity.

In this paper we have introduced feature algebra as an idempotent commutative semiring. We have given a set-based and a bag-based model of the proposed algebra. To compare elements of our algebra, besides the natural order defined on an idempotent semiring we use a refinement relation and have established some of its basic properties. Then we have given formal definitions of common terms that are intuitively used in the literature such as product, feature, and family. We introduced as well new notions such as that of a weak zero, and a measure for similarity among products and families.

The proposed algebra not only allows us to express the basic notions used by the product family paradigm community, but also enables algebraic manipulations of families of specifications, which enhances the generation of new knowledge about them. The notions and relationships introduced in FODA [9], FORM [10], FeatuRSEB [8] and GP [6] and expressed with graphical notations can easily be stated within our algebra. For instance, the alternative is expressed using the $+$ operator, and we write $f = b \cdot (1 + a) \cdot c$ (where b, and c are families) to express that a feature a is optional in a family f.

In contrast to other product family specification formalisms, like FODA and FORM, there exists a large body of theoretical results for idempotent commutative semiring and for algebraic techniques in general with strong impact for research related to problems of consistency, correctness, compatibility and reusability.

Many items found in the literature support the potential scalability of algebraic approaches in specifying industrial-scale software product families [1, 3]. However, we think that empirical substantiation of the scalability of our approach is needed.

This work opens new questions and brings in new research horizons. One of the questions is how to generate the specification of individual members of a given family from the specifications of features and the feature-algebraic specification of a family. One can envisage that the specifications of all the features are stored

in a specification depository and the specification of a product is generated on the fly. There is no need to have rigid specifications of products that are members of a family. This flexibility in generating specifications on the fly eases coping with the changes that frequently affect specifications of features. The proposed feature algebra provides a solid base on which to build for answering these questions.

As illustrated in [15], a product family might need to be specified from several perspectives. For example, in embedded systems, a product family needs to be specified from hardware and software perspectives. We conjecture that these perspectives are somehow interdependent. When this interdependence is known, how can we model the global specification of a family (involves all the perspectives) within a super-structure (such as a product structure) of feature algebras? The aim of further work in this area is to tackle these questions.

References

1. D. Batory. The road to utopia: A future for generative programming. Keynote presentation at the Dagstuhl Seminar for Domain Specific Program Generation, March 2003.
2. D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Software Product Line Conference (SPLC-EUROPE 2005)*, 26-29 September 2005.
3. D. Batory, R. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product-families. In *Conference on Automated-Software Engineering*, September 2002.
4. S. Bühne, K. Lauenroth, and K. Pohl. Modelling requirements variability across product lines. In *13th IEEE International Requirements Engineering Conference*, pages 41–50. IEEE Computer Society, August 29–September 2 2005.
5. P. Clements, L. M. Northrop, and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Professional, 2002.
6. K. Czarnecki and U. Eisenecker. *Generative Programming, Methods, Tools and Applications*. Addison-Wesley, 2000.
7. J. Desharnais, B. Möller, and G. Struth. Modal Kleene Algebra and Applications – A Survey. *Journal on Relational Methods in Computer Science*, (1):93–131, 2004. http://www.cosc.brocku.ca/Faculty/Winter/JoRMiCS/.
8. M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In P. Devanbu and J. Poulin, editors, *Proceedings of the 5th International Conference on Software Reuse*, pages 76–85. IEEE Computer Society, 1998.
9. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon Software Engineering Institute, Carnegie Mellon University, 1990.
10. K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.
11. N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co., 1980.
12. D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE2(1):1–9, March 1976.
13. C. Riva and C. D. Rosso. Experiences with software product family evolution. In *Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 161–169. IEEE Computer Society, 2003.

14. S. Roback and A. Pieczynski. Employing fuzzy logic in feature diagrams to model variability in software product-lines. In *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03)*, pages 305–311. IEEE Computer Society, 2003.
15. J. M. Thompson and M. P. Heimdahl. Structuring product family requirements for n-dimensional and hierarchical product lines. *Requirements Engineering Journal*, 2002.
16. J. M. Thompson, M. P. Heimdahl, and D. M. Erickson. Structuring formal control systems specifications for reuse: Surviving hardware changes. Technical Report TR 00-004, Department of Computer Science and Engineering, University of Minnesota, February 2000.

A Specification of the robot family

```
------------------------------------------------------------------------
------------------------------------------------------------------------
------- Robot family example (Hardware perspective) --------------------
------------------------------------------------------------------------
------------------------------------------------------------------------

--basic features:
treads = bf "Moves around on treads"
wheels = bf "Moves around on wheels"
legs   = bf "Moves around on legs"
basic_means_of_locomotion = treads .+. wheels .+. legs


turn        = bf "Able to turn an angle from the initial heading"
move_frwrd  = bf "Able to move forward"
move_bckwrd = bf "Able to move backward"
stay_idle   = bf "Able to stay inactive"

limited_spd  = bf "Robot limited to low speed of locomotion"
extended_spd = bf "Robot extended to high speed of locomotion"

basic_ctrl   = bf "Robot equipped with basic control (only on or off)"
digital_ctrl = bf "Robot equipped with digital valued indication of
                   locomotion speed and direction"

small_pltfrm  = bf "Small size platform robot"
medium_pltfrm = bf "Medium size platform robot"
large_pltfrm  = bf "Large size platform robot"

c_s_pneumatic   = bf "Pneumatic collision sensor"
c_s_mechanical  = bf "Mechanical collision sensor"
c_s_combination = bf "Collision sensor is a combination of mechanical
                      and pneumatic"

sur_finder  = bf "Small Ultrasonic Range Finder"
lcur_finder = bf "Low-cost Ultrasonic Ranger"
chpu_finder = bf "Compact High Performance Ultrasonic Ranger"

v_s_color_vision      = bf "Sensor capable of determining the color of
                            objects in the robot's environment"
black_white_vision    = bf "Black and white environmental vision system"
primary_colors_vision = bf "Primary colors environmental vision system"

speed_of_locomotion = limited_spd .+. extended_spd

locomotion_ctrl_sys = basic_ctrl .+. digital_ctrl

c_sensor   = c_s_pneumatic .+. c_s_mechanical .+. c_s_combination
```

```
rng_finder = sur_finder    .+. lcur_finder    .+. chpu_finder

platform_size_snsor  =      small_pltfrm  .*. (c_sensor .^<=.  3)
                      .+.  medium_pltfrm .*. (c_sensor .^<=.  7)
                      .+.  large_pltfrm  .*. (c_sensor .^<=. 11)

platform_size_finder =      small_pltfrm  .*. (rng_finder .^<=. 1)
                      .+.  medium_pltfrm .*. (rng_finder .^<=. 2)
                      .+.  large_pltfrm  .*. (rng_finder .^<=. 3)

-- PRODUCT LINES

basic_platform =     basic_means_of_locomotion
                 .*. turn
                 .*. move_frwrd
                 .*. move_bckwrd
                 .*. stay_idle
                 .*. opt[speed_of_locomotion]
                 .*. opt[locomotion_ctrl_sys]
                 .*. opt[platform_size_snsor]

enhanced_obstacle_detection =     basic_platform
                              .*. c_sensor
                              .*. opt[platform_size_finder]

environmental_vision =     enhanced_obstacle_detection
                       .*. v_s_color_vision
                       .*. opt[black_white_vision]
                       .*. opt[primary_colors_vision]

-- Constraints on all the products to exclude the impossible
-- or undesirable combinations of features

excludes =     treads .*. wheels
           .+. treads .*.  legs
           .+. wheels .*. legs
           .+. limited_spd   .*. extended_spd
           .+. basic_ctrl    .*. digital_ctrl
           .+. basic_ctrl    .*. digital_ctrl
           .+. small_pltfrm  .*. large_pltfrm
           .+. medium_pltfrm .*. large_pltfrm
           .+. small_pltfrm  .*. c_sensor .^. 4
           .+. medium_pltfrm .*. c_sensor .^. 5
           .+. large_pltfrm  .*. c_sensor .^.6
```

B Singles and Feature Propagation

In Section 6 we introduced products (8) and features (9). The definition of these structures have the same form. Therefore we give a common treatment of the notions of product and feature in this section.

Definition B.1 Consider a monoid $(S, \oplus, \mathbf{0})$ and $a \in S \backslash \{\mathbf{0}\}$. Define $a \preceq b \Leftrightarrow \exists\, c : b = a \oplus c$. Then a is a *single* iff

$$(\forall\, b : b \preceq a \;\Rightarrow\; (b = \mathbf{0} \vee b = a)) \;\wedge$$
$$(\forall\, b, c : a \preceq b \oplus c \;\Rightarrow\; (a \preceq b \vee a \preceq c)) \;.$$

The set of all singles of S is denoted by SG.

In Section C we relate this definitions to standard order-theoretic notions.

Hence, according to Definitions 6.1 and 6.2 products are singles w.r.t. $\leq$ as relation $\preceq$ and features are singles in the set of products w.r.t. $|$. Therefore, the relation $\preceq$ is the natural order $\leq$ in the case of $+$ and $|$ in the case of $\cdot$.

The following corollary shows that sums of singles can be uniquely decomposed into their single parts. For $\mathbb{P}$FS and $\mathbb{P}$FB this means that every (finite) set can be composed by singleton sets, i.e., for a set A, $A = \bigcup_{a \in A} a$, which is obviously true. In general we get

Corollary B.2 *Assume a monoid $(S, \oplus, \mathbf{0})$.*

1. *If a is a single and $J \subseteq \mathbb{N}$ is finite then*

$$a \preceq \bigoplus_{i \in J} b_j \;\Leftrightarrow\; \exists\, j \in J : a \preceq b_j \;.$$

2. *Let $I, J \subseteq \mathbb{N}$ be finite and a_i, b_j be singles. Then*

$$\bigoplus_{i \in I} a_i = \bigoplus_{j \in J} b_i \;\Rightarrow\; \{a_i : i \in I\} = \{b_j : j \in J\} \;.$$

Proof. 1. Straightforward induction on $|J|$.

2. Consider an a_i. By 1. there is some b_j with $a_i \preceq b_j$. Since a_i is a single we must have $a_i = b_j$, hence $a_i \in \{b_j : j \in J\}$. This shows $(\subseteq)$; the reverse inclusion is shown symmetrically. □

Part 2. says that the ordering of the factors in a product generated by singles is irrelevant.

In the remainder we show some properties for singles, which result in the feature propagation lemma, a lemma which is very interesting for the feature algebras $\mathbb{P}$FS and $\mathbb{P}$FB.

Definition B.3 In a partial order $(S, \preceq)$ we set

$$\mathsf{SG}(x) =_{df} \{a : a \preceq x,\, a \in \mathsf{SG}\} \;.$$

Lemma B.4 *Over a monoid S we have* $\mathsf{SG}(x \oplus y) = \mathsf{SG}(x) \oplus \mathsf{SG}(y)$.

Proof. For $a \in \mathsf{SG}$, we get by definition of $\mathsf{SG}(x)$, Corollary B.2.1 and definitoin again

$$\begin{aligned} & a \in \mathsf{SG}(x \oplus y) \\ \Leftrightarrow\ & a \preceq x \oplus y \\ \Leftrightarrow\ & a \preceq x \vee a \preceq y \\ \Leftrightarrow\ & a \in \mathsf{SG}(x) \vee a \in \mathsf{SG}(y) \\ \Leftrightarrow\ & a \in \mathsf{SG}(x) \cup \mathsf{SG}(y)\ . \end{aligned}$$

□

Corollary B.5 *Over a monoid S we have* $x \preceq y \Rightarrow \mathsf{SG}(x) \subseteq \mathsf{SG}(y)$.

Proof.
$$\begin{aligned} & x \preceq y \\ \Leftrightarrow\ & \exists\, c :\ x \oplus c = y \\ \Rightarrow\ & \exists\, c :\ \mathsf{SG}(x \oplus c) = \mathsf{SG}(y) \\ \Leftrightarrow\ & \exists\, c :\ \mathsf{SG}(x) \oplus \mathsf{SG}(c) = \mathsf{SG}(y) \\ \Leftrightarrow\ & \mathsf{SG}(x) \subseteq \mathsf{SG}(y)\ . \end{aligned}$$

□

Definition B.6 The partial order $(S, \preceq)$ is called *generated* if

$$\forall\, x \in S : x = \bigsqcup_{\mathsf{SG}(x)}\ .$$

We call it *finitely generated* if it is generated and for all $x \in S$ the set SG is finite.

The feature algebras $\mathbb{P}\mathrm{FS}$ and PLB have this property and therefore we have $\mathrm{g}\mathbb{P} = \mathbb{P}$ (cf. Page 13).

Corollary B.7 *Let S be generated. Then* $\mathsf{SG}(x) \subseteq \mathsf{SG}(y) \Rightarrow x \preceq y$ *and* $\mathsf{SG}(x) = \emptyset$ *iff* $x = \mathbf{0}$.

Proof. First,

$$\mathsf{SG}(x) \subseteq \mathsf{SG}(y) \Rightarrow \bigsqcup_{\mathsf{SG}(x)} \preceq \bigsqcup_{\mathsf{SG}(y)} \Leftrightarrow x \preceq y\ .$$

Second, by definition $\mathsf{SG}(\mathbf{0}) = \emptyset$. Moreover, if $\mathsf{SG}(x) = \emptyset$ then $x = \bigsqcup_{\mathsf{SG}(x)} = \bigsqcup_{\emptyset} = \mathbf{0}$.

□

Hence by Corollary B.5 and B.7 we have for a generated S that $\mathsf{SG}(x) \subseteq \mathsf{SG}(y) \Leftrightarrow x \preceq y$.

Definition B.8 We call a semiring S *strongly generated* if it is generated and SG is closed under composition, i.e., if $\mathsf{SG} \cdot \mathsf{SG} \subseteq \mathsf{SG}$.

The presented feature algebras $\mathbb{P}$FS and $\mathbb{P}$FB as well as the language semiring have this property, whereas the relation semiring and the path semiring don't.

Corollary B.9 *A strongly and finitely generated semiring does not have proper divisors of* $\mathbf{0}$*, i.e.,* $x \cdot y = \mathbf{0} \Rightarrow x = \mathbf{0} \vee y = \mathbf{0}$*.*

Proof. We know $x = \bigoplus_{a \in \mathsf{SG}(x)} a$ and $y = \bigoplus_{b \in \mathsf{SG}(y)} b$ where both sums are finite. Hence by distributivity, associativity and commutativity,

$$x \cdot y = (\bigoplus_{a \in \mathsf{SG}(x)} a) \cdot (\bigoplus_{b \in \mathsf{SG}(y)} b) = \bigoplus_{a \in \mathsf{SG}(x)} \bigoplus_{b \in \mathsf{SG}(y)} a \cdot b \ .$$

By strong generatedness all products $a \cdot b$ (if any) are single again. So the whole sum can be $\mathbf{0}$ iff it is empty, i.e., iff $\mathsf{SG}(x) = \emptyset \vee \mathsf{SG}(y) = \emptyset$, i.e., iff $x = \mathbf{0} \vee y = \mathbf{0}$ by Corollary B.7. □

Now we are ready to prove the feature propagation lemma. This lemma describes the situation that in the case where a refines c, there exists an element d such that $c \cdot d$ is the same as a. In $\mathbb{P}$FS and $\mathbb{P}$FB this lemma says that we can build up a single product which includes all the missing features.

Lemma B.10 *Assume a strongly and finitely generated semiring* S *and let* $c \in \mathsf{SG}$*. Then*

$$a \sqsubseteq c \Rightarrow \exists\, d \in S : a = c \cdot d \ .$$

Proof. Assume $a \sqsubseteq c$, say $a \preceq c \cdot b$ for some $b \in S$. Then

$$c \cdot b = c \cdot \bigoplus_{e \in \mathsf{SG}(b)} e = \bigoplus_{e \in \mathsf{SG}(b)} c \cdot e \ .$$

By strong generatedness, all products $c \cdot e$ are singles, so that $\mathsf{SG}(c \cdot b) = \{c \cdot e : e \in \mathsf{SG}(b)\}$ by Corollary B.2.2. From $a \preceq c \cdot b$ we infer by Corollary B.5 that $\mathsf{SG}(a) \subseteq \mathsf{SG}(c \cdot b)$. So by Corollary B.2.2 there must be a subset $B \subseteq \mathsf{SG}(b)$ with $\mathsf{SG}(a) = \{c \cdot e : e \in B\}$. Now

$$a = \bigoplus_{f \in \mathsf{SG}(a)} f = \bigoplus_{e \in B} c \cdot e = c \cdot \bigoplus_{e \in B} e \ ,$$

so that we can set $d =_{df} \bigoplus_{e \in B} e$. □

C Order-Theoretic Background

In this section we want give an order-theoretic background for singles, features and products. In B we have already shown that features as well as products are singles. Now we show how singles can be classified in an algebraic manner using well-known definitions. Therefore we recapitulate these.

Definition C.1 Let $(S, \leq)$ be a partial order with a smallest element 0

1. An element $a \in S$ is *atomic* or an *atom* if

$$a \neq 0 \wedge \forall x : x \leq a \Rightarrow (x = 0 \vee x = a) .$$

2. An element $a \in S$ is *join-irreducible* if

$$\forall\, b, c : a = b + c \Rightarrow (a = b \vee a = c) .$$

3. An element $a \in S$ is *join-prime* if

$$\forall\, b, c : a \leq b + c \Leftrightarrow (a \leq b \vee a \leq c) .$$

So our singles are exactly the atomic and join-prime elements. We have the following relations between these notions.

Lemma C.2

1. *Every atom is join-irreducible.*
2. *Every join-prime element is join-irreducible.*
3. *If S is a distributive lattice then every join-irreducible element is join-prime.*
4. *If S is a Boolean lattice then every join-irreducible element is an atom. Hence in this case all three notions coincide.*

Proof. Let $a, b, c \in S$.

1. First, we assume $a \neq 0$

$$\begin{array}{ll} & b + c = a \\ \Rightarrow & \quad \{\![\text{ ordering }]\!\} \\ & b \leq a \\ \Rightarrow & \quad \{\![\, a \text{ atom }]\!\} \\ & b = 0 \vee b = a \\ \Rightarrow & \quad \{\![\, b + c = a \text{ and neutrality of } 0 \,]\!\} \\ & c = a \vee b = a \; . \end{array}$$

   Now we assume $a = 0$ and $b + c = 0$. In particular $b + c \leq 0$ and there for by lattice algebra $b \leq 0 \wedge c \leq 0$. The claim follows since 0 is the least element.

2. $$\begin{array}{ll} & b + c = a \\ \Rightarrow & \quad \{\![\text{ ordering }]\!\} \\ & a \leq b + c \\ \Rightarrow & \quad \{\![\, a \text{ join-prime }]\!\} \\ & a \leq b \vee a \leq c \; . \end{array}$$

   Since $b + c = a \Rightarrow b \leq a \wedge c \leq a$ we get from that by antisymmetry of $\leq$ that $a = b \vee a = c$.

3. $a \leq b + c$

$\Leftrightarrow$ {[ordering]}

$a \sqcap (b + c) = a$

$\Leftrightarrow$ {[distributivity]}

$(a \sqcap b) + (a \sqcap c) = a$

$\Rightarrow$ {[a join-irreducible]}

$a = a \sqcap b \vee a = a \sqcap c$

$\Leftrightarrow$ {[ordering]}

$a \leq b \vee a \leq c$.

4. Assume $b \leq a$, i.e. $b = a \sqcap b$.

TRUE

$\Leftrightarrow$ {[Boolean algebra]}

$a = (a \sqcap b) + (a \sqcap \overline{b})$

$\Rightarrow$ {[a join-irreducible]}

$a = a \sqcap b \vee a = a \sqcap \overline{b}$

$\Leftrightarrow$ {[$b = a \sqcap b$ and ordering]}

$a = b \vee a \leq \overline{b}$

$\Leftrightarrow$ {[Boolean algebra]}

$a = b \vee a \sqcap b = 0$

$\Leftrightarrow$ {[$b = a \sqcap b$]}

$a = b \vee b = 0$.

□

Hence we have shown that the definition of singles (products and features) fits well in the classical notion of algebra.

> Atomics is a very intricate theorem and can be worked out with algebra but you would want to take it by degrees because you might spend the whole night proving a bit of it with rulers and cosines and similar other instruments and then at the wind-up not believe what you had proved at all. If that happened you would have to go back over it till you got a place where you could believe your own facts and figures as delineated from Hall and Knight's Algebra and then go on again from that particular place till you had the whole thing properly believed and not have bits of it half-believed or a doubt in your head hurting you like when you lose the stud of your shirt in bed.
>
> Flann O'Brien, The Third Policeman