

An Algebra of Product Families^{*}

Peter Höfner¹, Ridha Khedri², Bernhard Möller¹

¹ Institut für Informatik, Universität Augsburg, Germany,
e-mail: {hoefner,moeller}@informatik.uni-augsburg.de

² Department of Computing and Software, McMaster University,
Hamilton, Canada, e-mail: khedri@mcmaster.ca

March 2, 2010

Abstract Experience from recent years has shown that it is often advantageous 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 *product families* are built from elementary *features* that reach from hardware parts to software artefacts such as requirements, architectural elements or patterns, components, middleware, or code. We use the well established mathematical structure of idempotent semirings as the basis for a *product family algebra* that allows a formal treatment of the above notions. A particular application of the algebra concerns the *multi-view reconciliation problem* that arises when complex systems are modelled. We use algebraic integration constraints linking features in one view to features in the same or a different view and show in several examples the suitability of this approach for a wide class of integration constraint formulations. Our approach is illustrated with a HASKELL prototype implementation of one particular model of product family algebra.

Key words Product family, product line, idempotent semiring, multi-view reconciliation, formal family specification, feature modelling.

1 Introduction

The concepts of product families originally stem from hardware industry. They allow manufacturing several variants of products, which leads to a significant reduction of development and maintenance costs. As early as 1976 Parnas [31] realised that the adoption of the product family paradigm is also useful in software development. The research on product families aims

^{*} Revised and enlarged version of [17,20]

at studying the commonality/variability occurring in a family of products in order to have better management of and processes for software production. This family approach proposes that, instead of focusing attention on a single immutable system to be built, one takes into account predictable changes to it. To this end one performs analysis and design of a family of systems that share a core part (commonality in all the members). Such a family is called a *product line* or an *f-carrying* family (“f” standing for “common features”) when its members have a common set of features that satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way [6,33].

By now, the notion of *product family* has gained a lot of attention and has found its way into the software development process in industry [32]. Weiss and Lai [41, Preface, p. xvii] report that applying family-based processes at *Lucent Technologies* led to decreases in development time and costs for family members by 60% to 70%.

With the current strong emphasis on embedded systems, product families consisting of both hardware and software components also become of rising interest. Hence, more generally, a product family can be defined as a set of products that share common hardware or software artefacts such as requirements [34], architectural properties [35], components [28], middleware [13], or code [40]. In the remainder, we call such artefacts *features*. This fits well with the widely accepted view that a feature is a conceptual characteristic visible to stakeholders (e.g., users, customers, developers, managers, etc.). At the requirements level, a feature encapsulates a set of related system-environment interactions. For instance, the IEEE standard [36, p. 19] states the following:

“A feature is an externally desired service by the system that may require a sequence of inputs to effect the desired result. For example, in a telephone system, features include local call, call forwarding, and conference call.”

Also, in [34], Savolainen et al. write “A feature is specified by a set of requirements; this set may contain one or more requirements”. Therefore, a feature can be described using scenarios (or use-cases) that provide the system-environment interactions.

The aim of the present paper is to underpin the ideas of family-based development with a formalism that allows a mathematically precise description and manipulation of product families. To this end we propose a *product family algebra* that we use to describe and analyse the commonalities and variabilities of a system family. We will extend the standard notion of an idempotent semiring to cover product families, refinements, product development and classification and, finally, multi-view reconciliation. In its main models the elements are sets of products, i.e. product families.

The multi-view reconciliation problem is of particular importance for embedded systems such as automotive systems. These are very difficult to specify using one single model that takes the software and the hardware

perspective of the system into consideration. For engineering tasks, it is common to adopt multi-view approaches. For instance, when constructing a building, the specifiers elaborate many views of it: structure view, plumbing view, electrical wiring view, etc. These views need to be coherent. When we carry this approach over to product families, the complexity of the problem increases: each product of every view needs to be coherent with at least one in the other views.

Each view gives a partial description of the considered family, including a mixture of optional and required features/properties. Reconciling these views when integrating them helps to eliminate contradictory features/properties of the family, which leads to convergence towards a comprehensive overall specification of the family. It is worth noting that this specification might not be complete; it depends on the domain coverage of the views.

After view reconciliation, the obtained family model comprises potential products. Some of them are theoretically possible in the family model but correspond to inconsistent scenarios that cannot be realised jointly; at a more detailed level, the functional requirements associated with the features clash. In [26], the reader finds a discussion on the transformation from an algebraic model of a family given as a product family algebra term into a more concrete model of its members, of the commonality of its members, or of a simple combination of requirements features.

To eliminate the inconsistent products we will introduce requirement constraints as additional axioms formulated in terms of the algebra.

The paper is organised as follows. We start in Section 2 by giving a simple example of a product family. In Section 3, we present the basics of product family algebra and discuss the relationship to feature oriented domain analysis. Section 4 reports on a HASKELL prototype implementation of a model of that algebra. In Section 5, as an example we discuss a product family that exhibits a sophisticated structure of subfamilies. In Section 6, we discuss building product families and generating desirable products. In Section 7, we extend the algebra by a requirement relation and elaborate on its properties and its use in formally capturing informal integration constraints. In Section 8, we present our approach to the multi-view reconciliation problem. In Section 9, we give a case study of a two-view specification of a driver assisting system and its reconciliation using our technique. Section 10 contains a comprehensive review of the literature of product family based software development. We conclude and point to future research in Section 11; in particular, we indicate how the reconciled views can be used for further analysis at more concrete levels of requirements employing already known requirement analysis techniques. The paper is rounded off by Appendix A, which gives proofs for our mathematical results, Appendix B, which presents a cross-section of our HASKELL implementation, and Appendix C, which sketches the use of an automated theorem prover in the context of product family algebra.

2 Example of a Simple Product Family

The following example is adapted from a case study given in [5]. An electronics 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 handle several DVDs simultaneously.

Product family	Mandatory	Optional	Commonalities
MP3 Player	– Play MP3 files	– Record MP3 files	– Audio equaliser – Dolby surround (advanced audio features)
DVD Player	– Play DVD	– Play music CD – View pictures from picture CD – Burn CD – Handle additional DVDs	
Hard Disk Recorder		– MP3 player – organise MP3 files	

Table 1 Commonalities and variability of a set of product lines

We see that there are different models of DVD Players. For example there is one that is able to play music CDs and one that does not have this feature. But how many different models are actually described in Table 1? And what are the properties/features of these products? Later on we will give the answers to these two questions. We will develop a model that gives us all combinations of features; with its help we will be able to envision new products. Vice versa, the model will allow us to calculate the commonalities of a given set of products.

3 Product Family Algebra

In this section, we introduce the foundations of our approach. Mathematically, it is based on semirings, hence we will first present these. After that we show the connection to the well established area of feature-oriented domain analysis (FODA). Finally, we will give precise definitions of some characteristic notions around product families.

3.1 Semirings

Definition 3.1 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 $a \leq b \iff_{df} a + b = b$ 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 .

An important example of an idempotent (but not commutative) semiring is REL, the algebra of binary relations over a set under union and relational composition.

Note that multiplication is not assumed to be commutative or idempotent; hence, unlike addition, it does not induce a partial order. Also, no absorption laws between addition and multiplication are assumed, so that in general there is no lattice structure on a semiring. However, an idempotent semiring induces an upper semilattice in which addition coincides with the supremum operator. More details about (idempotent) semirings and examples of their relevance to computer science can be found, e.g., in [15,10]. In the present paper, addition $+$ can be interpreted as a choice between families of products (or, equivalently, as their union), and multiplication \cdot as their composition or mandatory presence. The element 0 represents the empty family of products while 1 represents a family consisting just of a single pseudo-product with no features. Consequently, the term $\text{opt}[a] =_{df} a + 1$ denotes a product family that offers a choice between the products in a and the empty product; hence it will be used to express optionality of a -products. Optionality of a list of products p_1, \dots, p_n is denoted by

$$\text{opt}[p_1, \dots, p_n] =_{df} (1 + p_1) \cdots (1 + p_n),$$

a choice allowing any subset of the products p_1, \dots, p_n (or none of them).

The natural order \leq is the abstract counterpart of the inclusion order \subseteq between sets (e.g. sets of products).

3.2 From FODA to Feature Algebra

To further clarify the relevance and usefulness of the semiring approach we link it to the probably most prominent and widespread tool, viz. Feature-Oriented Domain Analysis (FODA) [24]. It uses feature models to give the mandatory, optional and alternative concepts within a domain [24,33]. These are closely related to our work and are given by *feature diagrams* combined with a *domain dictionary*. Feature diagrams are basically OR/AND trees that can capture the commonalities and mandatory features as well as the optional ones of a feature algebra. The leaf nodes contain the basic

features of the describes product family. In the domain dictionary each basic feature is specified.

We will now show that OR/AND trees correspond very directly to algebraic terms over the semiring operations. We assume that every basic feature is denoted by a constant, such as **transmission**, **horsepower** or **aircondition** when we want to specify a family of cars. The translation rules for the basic parts of an OR/AND tree into an algebraic term are given in Table 2.

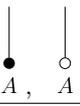
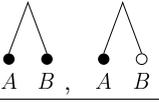
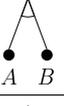
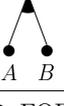
Base construct (feature diagram)	Description	Algebraic counterpart
	mandatory and optional feature	A and $\text{opt}[A]$, resp.
	multiple features	$A \cdot B$, $A \cdot \text{opt}[B]$, etc.
	alternative features	$A + B$
	or-group	$A + B + A \cdot B$

Table 2 FODA feature diagrams and their corresponding algebraic terms

Using these rules every feature diagram can be transformed into an algebraic expression using a bottom-up traversal. This recursive method translates each subtree into an algebraic expression, starting from the leaf nodes going up to the root. The result is unique up to commutativity and associativity of the semiring operators.

Example 3.2 The standard example of a feature diagram describes features of a car and was originally introduced in [24]. The feature diagram as well as a stepwise bottom up transformation into an algebraic term is given in Figure 1. \square

It is evident that the term representation is much more compact than the tree representation. The converse direction (building a tree from a given algebraic term) is straightforward, too.

Feature diagrams may additionally be equipped with composition rules, like exclusions. These requirements are also called cross-tree constraints. We will show in Section 7 how to model such constraints.

To model requirements like “air condition requires horsepower > 100 HP”, one can just use more refined basic features.

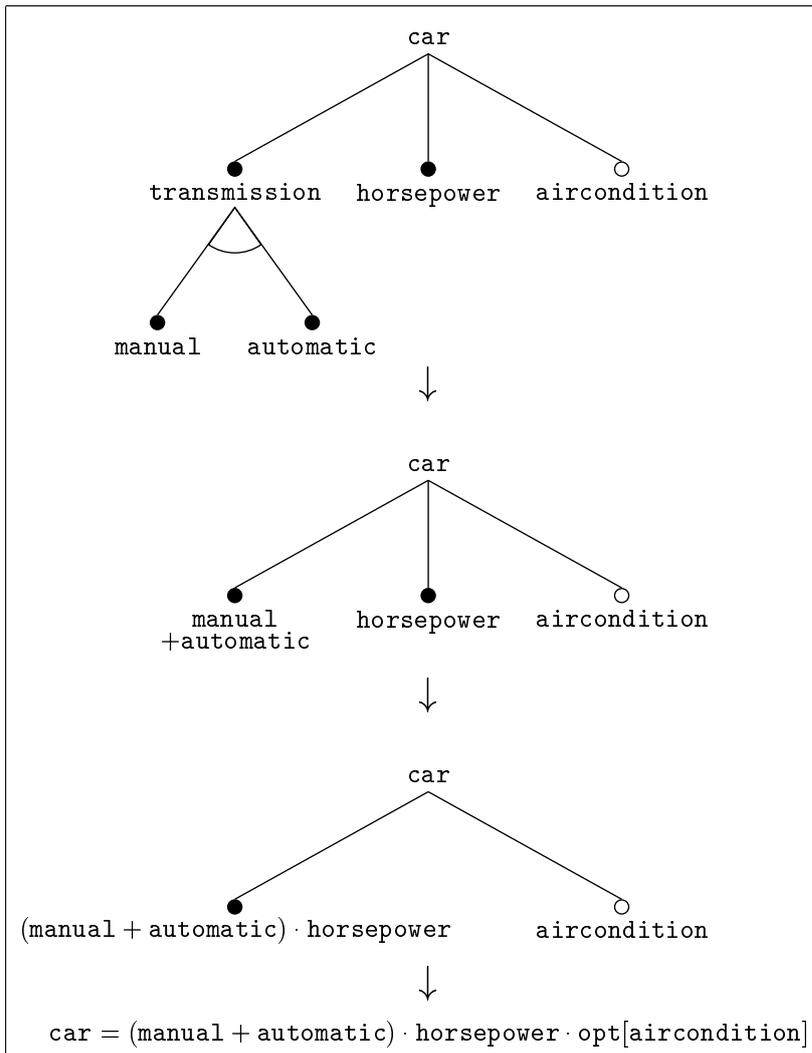


Fig. 1 Step-wise transformation

3.3 Set and Bag Models of Semirings

To further illustrate the roles of the semiring operations we construct a concrete model that consists of sets of products, each of which, in turn, is a set of basic features.

Let \mathbb{F} be a set of arbitrary elements that we call *features*. Then 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*. A special family is $1 = \{\emptyset\}$ consisting just of the empty product that has no features. Note that

according to this general definition the members of a product family need not have common features. Commonalities will be discussed in Section 3.7.

Example 3.3 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 equaliser` and so on. To shorten expressions we use the following abbreviations.

<i>Play MP3 files</i>	<code>p_mp3</code>
<i>Record MP3 files</i>	<code>r_mp3</code>
<i>Audio equaliser</i>	<code>a_eq</code>
<i>Dolby surround</i>	<code>dbs</code>
<i>Burn CD</i>	<code>b_cd</code>
<i>Additional DVDs</i>	<code>a_dvd</code>

Then we can describe the MP3 players algebraically as

$$\text{mp3_player} = \text{p_mp3} \cdot \text{opt}[\text{r_mp3}] \cdot \text{a_eq} \cdot \text{dbs} .$$

This expression matches Table 1 as follows: the features `p_mp3`, `a_eq` and `dbs` are classified as mandatory or as common to all products (implicitly mandatory). Therefore, in the above expression defining `mp3_player`, they are combined using the \cdot operator. The feature `r_mp3` is classified as optional in Table 1 and it is expressed above by including the term `opt[r_mp3]`. \square

We now formally define the operation \cdot of our set model which is a composition or merging operator on products:

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

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

$$\begin{aligned} + : \mathcal{P}(\mathbb{F}) \times \mathcal{P}(\mathbb{F}) &\rightarrow \mathcal{P}(\mathbb{F}) \\ P + Q &=_{df} P \cup Q , \end{aligned}$$

It is easily checked that with these definitions the structure

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

forms a commutative idempotent semiring, called the *set-based model* over \mathbb{F} . It does not allow multiple occurrences of the same feature in a product. If duplication of features is desired, one can use an analogous model that employs multisets (also called bags) of features as products. This *bag-based model* over \mathbb{F} forms also a commutative idempotent semiring and is denoted by $\mathbb{F}\text{FB}$.

Let us also give a small example of the use of the semiring laws. It is required that \cdot distributes through $+$. Using this and neutrality of 1 we can rewrite the expression for `mp3_player` into

$$\text{p_mp3} \cdot \text{r_mp3} \cdot \text{a_eq} \cdot \text{dbs} \quad + \quad \text{p_mp3} \cdot \text{a_eq} \cdot \text{dbs} ,$$

which clearly exhibits that `mp3_player` is a family consisting of exactly two products. Similarly one can determine the number of products inside the family `dvd_player`. Conversely, using commutativity of \cdot , we can rearrange the expression for `mp3_player` into

$$\text{p_mp3} \cdot \text{a_eq} \cdot \text{dbs} \cdot (\text{r_mp3} + 1) ;$$

this form lists the commonality of the products at the beginning and then gives the variability which in this case is just an optional MP3 recorder. This means that the MP3 player can be described as `p_mp3 · a_eq · dbs · opt[r_mp3]`.

Using product algebraic terms offers an abstraction from set-theory. On one hand it provides a common structure that subsumes IPFB and IPFS and on the other hand it avoids many set-theoretic notations, like accumulations of braces, and emphasises the relevant aspects like commonalities.

3.4 Products, Features and Product Feature Algebras

In the literature, terms like product family and subfamily lack exact definitions. We now show how to achieve this using semiring terminology.

Intuitively, a single product cannot be decomposed using the choice operator $+$. In other terms, it does not offer optional or alternative parts. Let us characterise this formally.

Definition 3.4 Assume a commutative idempotent semiring A . An element $a \in A$ is called a *product* if it satisfies the following laws:

$$\forall b : b \leq a \implies (b = 0 \vee b = a) , \quad (1)$$

$$\forall b, c : a \leq b + c \implies (a \leq b \vee a \leq c) . \quad (2)$$

In particular, 0 is a product.¹ A product a is *proper* if $a \neq 0$.

Implication (1) indicates that a product must not have non-trivial subfamilies, i.e., its only subfamilies are the empty one and the product itself. Mathematically such elements are called *atoms*. The second formula (2) states that if a is a subfamily of the family formed by two families b and c , it must be a subfamily of one of them. In mathematics this property is also known as *primeness*. As a consequence of both requirements we have that a product a is *irreducible*, i.e., that no product can be split into two different proper products. In signs this reads $\forall b, c : a = b + c \implies (a = b \vee a = c)$.

In IPFS and IPFB an element is a product if and only if it contains only one element, i.e., it is a singleton set. With respect to our running example, the MP3-Player family `p_mp3 · a_eq · dbs` is a product, whereas the family `p_mp3 · (r_mp3 + 1) · a_eq · dbs` is not — it can be decomposed into the two products `p_mp3 · a_eq · dbs` and `p_mp3 · r_mp3 · a_eq · dbs`.

¹ To avoid tedious case analyses we deviate here slightly from the definition in [17], where we explicitly excluded 0 from being a product.

Furthermore, in \mathbb{PFS} and \mathbb{PFB} the neutral element $1 = \{\emptyset\}$ satisfies (1) and (2) and hence is a product. However, in general semirings this need not be the case.

Analogously to Definition 3.4, features can be defined as indecomposable elements, but this time w.r.t. multiplication rather than addition.

Definition 3.5 An element a is called a *feature* if it is a proper product different from 1 satisfying the following laws:

$$\forall b : b | a \implies b = 1 \vee b = a , \quad (3)$$

$$\forall b, c : a | (b \cdot c) \implies (a | b \vee a | c) , \quad (4)$$

where the divisibility relation $|$ is given by $x | y \iff_{df} \exists z : y = x \cdot z$.

As for products, implications (3) and (4) state that no feature can be decomposed into two non-trivial features. In particular, implication (3) states that for every product family b , if we have b as mandatory for the construction of a (i.e., $\exists c : a = b \cdot c$, which indicates as well that b divides a), it implies that either b is the special product 1 or b is identical to a . (4) states that for every product families b and c , if we have a as mandatory to forming $b \cdot c$, it implies that either a is mandatory to forming b or it mandatory to forming c . To illustrate this definition, we use again our running example. The product (family) $\mathbf{a_eq} \cdot \mathbf{dbs}$ is not a feature since it is made up from the features $\mathbf{a_eq}$ and \mathbf{dbs} . Again these laws axiomatise an element a to be irreducible and prime, this time with respect to multiplication.

Therefore, from a mathematical point of view, the characteristics of products and features are similar and well known. A uniform treatment of both notions is given in the Appendix of [18], where also the order-theoretic background is discussed.

It should also be noted that we can represent the multiset model isomorphically in the following way: view a natural number as the multiset of its prime factors. Now number the (atomic) features by primes and represent a product of features as the corresponding natural number. The above definition of features is just a generalisation of the property of primality, so that things fit well together.

We now are in the position to give our main definition.

Definition 3.6 A *product family algebra* is an idempotent and commutative semiring in which 1 is a product. Its elements are called *product families* or briefly *families*. A family g is a *subfamily* of family f iff $g \leq f$, where \leq is the natural semiring order.

Lemma 3.7 Both \mathbb{PFS} and \mathbb{PFB} are product family algebras.

Since variants of semirings have already successfully been combined with automated theorem provers [21, 22], we implemented product family algebra axiomatically in the first-order theorem prover Prover9 and the counterexample generator Mace4 [27]. The encoding can be found in Appendix C.

Using this encoding we can prove all the presented theorems and lemmas fully automatically. For the sake of readability we do not display the input/output files and machine proofs. They all can be found at a web site [16]. Proofs by hand can be found in Appendix A.

3.5 Feature-Generated Algebras

For practical reasons it is useful to assume that every product is built from a given set of features, as in IPFS and IPFB. Vice versa this means that every product can again be split into its features. To express this behaviour mathematically, we introduce the notion of feature-generated algebras.

Definition 3.8 A product family algebra is *feature-generated* if every element is a finite sum of finite products of features, where a *product of features* is a composition $f_1 \cdots f_m$ of features that itself is a product, and the multiplication of two products of features is a product of features again. In this case, single features are the “smallest” components from which products and product lines are built. The *size* of element a is the minimum number n such that $a = p_1 + \cdots + p_n$ for suitable products p_i of features.

The condition that the set of products of features should be closed under multiplication ensures that irreducible elements do not become reducible by combining them.

Lemma 3.9 *The set and bag models over a finite set \mathbb{F} of basic features are feature-generated; in both cases the size of a family is its cardinality.*

Example 3.10 The size of `mp3_player = p_mp3 · (r_mp3 + 1) · a_eq · dbs` is two. \square

In a feature-generated algebra the natural order can be expressed as an inclusion between generating sets:

Lemma 3.11 *Consider proper products p_1, \dots, p_m and products q_1, \dots, q_n in a feature-generated algebra. Then $p_1 + \cdots + p_m \leq q_1 + \cdots + q_n$ iff $\{p_1, \dots, p_m\} \subseteq \{q_1, \dots, q_n\}$.*

This implies that in a feature generated product family algebra the set of generating proper products of every element is unique:

Lemma 3.12 *Suppose that in a feature-generated algebra we have $p_1 + \cdots + p_m = q_1 + \cdots + q_n$ for proper products p_1, \dots, p_m and q_1, \dots, q_n . Then $m = n$ and $\{p_1, \dots, p_m\} = \{q_1, \dots, q_n\}$.*

As we have seen, the representation of the elements in sum-of-products form corresponds to OR/AND trees of features [29]. It may also be viewed as a commutative variant of the well known Backus-Naur form of grammars where $+$ corresponds to the variant stroke $|$ and \cdot to commutative juxtaposition. It yields the following fundamental

Principle of Family Induction Assume a feature-generated algebra A and a predicate P on A . If $P(p)$ holds for all products $p \in A$ (induction base) and is preserved by addition, i.e., satisfies $P(b) \wedge P(c) \Rightarrow P(b + c)$ (induction step) then $P(a)$ holds for all $a \in A$.

The soundness of this principle is shown by a straightforward induction on the size of the elements of A . Note that the induction base also requires establishing $P(0)$, since 0 is an (improper) product.

3.6 Refinement of Product Families

Often new product families are derived from existing ones by adding features. In this section we develop a corresponding comparison relation between product families. We motivate this by an extension of the MP3 Player example.

Example 3.13 We look again at the electronics company of Section 2. For the moment we are only interested in the two product families `dvd_player` and `mp3_player`.

$$\begin{aligned} \text{dvd_player} &= \text{p_dvd} \cdot \text{a_eq} \cdot \text{dbs} \cdot \text{opt}[\text{p_mp3}] , \\ \text{mp3_player} &= \text{p_mp3} \cdot \text{a_eq} \cdot \text{dbs} \cdot \text{opt}[\text{r_mp3}] . \end{aligned}$$

To find all common parts we look at the sum of the two products and by a simple calculation using distributivity obtain

$$\begin{aligned} \text{dvd_player} + \text{mp3_player} &= \\ (\text{p_dvd} \cdot \text{opt}[\text{p_mp3}] + \text{p_mp3} \cdot \text{opt}[\text{r_mp3}]) \cdot \text{a_eq} \cdot \text{dbs} . \end{aligned}$$

Next, we define an “older” product family of DVD players that does not have the capability for Dolby surround:

$$\text{old_dvd_player} = \text{p_dvd} \cdot \text{opt}[\text{p_mp3}] \cdot \text{a_eq} .$$

Since each product (or member) of `dvd_player` has at least the same features as a product of `old_dvd_player`, we say that `dvd_player` is a refinement of `old_dvd_player`. \square

For this kind of situation we introduce the relation $a \sqsubseteq b$ between elements a, b of a product family algebra. 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, or, conversely, that a is a specialisation of b , which explains the notation $a \sqsubseteq b$.

Definition 3.14 Formally, the *refinement* relation is defined as

$$a \sqsubseteq b \iff_{df} \exists c : a \leq b \cdot c ;$$

it is a preorder, i.e., it is reflexive and transitive.

Lemma 3.15 *Divisibility implies refinement: $b|a \implies a \sqsubseteq b$.*

The reverse implication need not hold for the following reason: $b|a$ means that *all* products of b can (uniformly) be extended to products of a , whereas $a \sqsubseteq b$ allows that some products of b may be disregarded in the extension. However, if the refined element is a product p there is nothing to disregard unless one wants to end up with the empty product family, and hence p has to be contained in every product of a so that in this case also p divides a :

Lemma 3.16 *Let a, p be elements of a feature-generated algebra such that p is a product. Then refinement and divisibility coincide, i.e., $a \sqsubseteq p \iff p|a$.*

Because of this lemma, in such algebras we may pronounce $a \sqsubseteq p$ as “ a has p (as a subproduct)”.

We list a few useful properties of the refinement relation.

Lemma 3.17 *Let a, b, c be elements of a product family algebra. Then*

- (a) $a \leq b \implies a \sqsubseteq b$.
- (b) $a \cdot b \sqsubseteq b$.
- (c) $a \sqsubseteq a + b$.
- (d) $a \sqsubseteq b \implies a + c \sqsubseteq b + c$.
- (e) $a \sqsubseteq b \implies a \cdot c \sqsubseteq b \cdot c$.
- (f) $a \sqsubseteq 0 \iff a \leq 0$.
- (g) $0 \sqsubseteq a \sqsubseteq 1$.

In IPFS and IPFB, Part (b) describes the situation that adding features (multiplying by an element in our algebra) refines products. Part (c) offers an alternative product on the right hand side. Part (d) and Part (e) are standard isotony laws. Part (g) 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, Part (g) reflects that the product without any features (which is represented by 1) is refined by any family.

Next we give rules for splitting refinements that involve sums.

Lemma 3.18 *Let a, b, c, p be elements of a product family algebra such that p is a product.*

- (a) $a + b \sqsubseteq c \iff a \sqsubseteq c \wedge b \sqsubseteq c$.
- (b) $p \sqsubseteq a + b \iff p \sqsubseteq a \vee p \sqsubseteq b$.

Part (a) means that a choice refines an element if and only if both parts do. Part (b) say that a product refines a choice if it refines either of the parts. One half of this follows for general elements from Part (c) of Lemma 3.17, the other from irreducibility of products.

If the algebra contains a greatest element the refinement relation can be represented without an existential quantification:

Lemma 3.19 *If a product family algebra contains a \leq -greatest element \top , we have*

$$a \sqsubseteq b \iff a \leq b \cdot \top \iff a \cdot \top \leq b \cdot \top .$$

E.g., in \mathbb{PFS} and \mathbb{PFB} the greatest element is the product family that contains every product built from the features of \mathbb{F} . In \mathbb{PFS} that is $\top = \mathcal{P}(\mathbb{F})$.

3.7 Further Notions and Properties

We now show how in principle the notion of a product line, i.e., a family with at least one common feature, can be formalised in product family algebra. We do this for the case of at least one common feature in a family; more liberal notions can be defined analogously.

Definition 3.20 An *f-carrying family* a is a family in which all products share at least one common feature f , i.e, there are a feature f and products p_1, \dots, p_n such that $a = f \cdot (p_1 + \dots + p_n)$.

Of course, the f -carrying 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 then each subfamily of a forms an f -carrying family again, since it, too, has f as a common feature.

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

Definition 3.21 Assume a set \mathbb{F} of features and $k \in \mathbb{N}$.

- (a) The set of all products with at most k features is defined by $\mathbb{F}^{\leq k} =_{df} \{x_1 \cdots x_n : k \in \mathbb{N}, n \leq k, x_i \in \mathbb{F}\}$.
- (b) A family a_1 is *k-near* to the family a_2 , if $\exists g \neq 0 : \exists x, y \in \mathbb{F}^{\leq k} : x \neq y \wedge a_1 = x \cdot g \wedge a_2 = y \cdot g$.

Part (b) describes the situation where the product families a_1 and a_2 differ in at most k features. Since every product is also a product family (which has only one member), we also have a notion for measure similarity of products.

Example 3.22 We resume the product line of Section 2 and Example 3.3. Assume two products: an MP3 player defined as $p_mp3 \cdot a_eq \cdot dbs$ and an MP3 recorder given by $p_mp3 \cdot r_mp3 \cdot a_eq$. As above we can find all common parts by factoring out:

$$p_mp3 \cdot a_eq \cdot (dbs + r_mp3) .$$

Thus the common parts are $p_mp3 \cdot a_eq$. In particular, the player and the recorder are 1-near. \square

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

$$II =_{df} \{\mathbb{F}\} .$$

This element contains only one product, namely the product that has all possible features. From an intuitive perspective, it is the “greatest” product, and hence refines every other product. In this case we have $a \cdot II = II$ if $a \neq 0$. Then, by setting $c = II$ in the definition of the refinement relation \sqsubseteq in Section 3 we get

$$II \sqsubseteq a \text{ if } a \neq 0 .$$

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

Lemma 3.23

- (a) *A weak zero is unique if it exists.*
- (b) *A weak zero z refines everything except 0, i.e., $z \sqsubseteq a \iff a \neq 0$.*
- (c) *If z is a weak zero then $a \sqsubseteq z \iff a \leq z$.*

Note that in \mathbb{PFB} there is no weak zero, since multiple occurrences of features are allowed.

4 A Prototype Implementation in HASKELL

To check the adequacy of our definitions we have written a prototype implementation of the \mathbb{PFB} model² 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. We have also implemented the refinement relation.

Example 4.1 We extend products of the electronics company of Section 9 by some more features.

```
-- 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"
```

² The program and a short description can be found at <http://www.informatik.uni-augsburg.de/forschung/reports>.

```

v_cd = bf "view picture CD"
b_cd = bf "burn CD"
a_cd = bf "play additional CD"
a_eq = bf "audio equaliser"
dbs  = 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 = a_eq .* dbs .*
        (mp3_player .+. dvd_player .+. hd)

```

In the above code, we are using the HASKELL notation of our implementation, i.e., `.*` and `.*` denote multiplication and addition, respectively.

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

```

=====
                        Common Parts
-----
audio equaliser
dolby surround
=====
                        Variability
-----
burn CD
play CD
play DVD
play additional CD
-----
burn CD
play CD
play DVD
play additional CD
view picture CD
-----
burn CD
...

```

The common parts are audio equaliser and dolby surround. Besides these parts, the first product has the capabilities of burning and playing CDs, playing DVDs and last it can handle an additional CD. The second product has all the features of the first one (including the common ones). As an add-on it is also able to handle picture CDs. \square

5 A More Complex Product Family

The DVD/CD example of Section 2 was chosen for its simplicity to illustrate basic notions. We now define a family of products that exhibits 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. The example is borrowed from [38] where it is used to illustrate a set-theoretic approach to reasoning about domains of what is called *n-dimensional and hierarchical* product families³. 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. [37], where the platforms are thoroughly described. The full specification in terms of our HASKELL prototype can be found in Appendix B.

We briefly explain the main parts of the robots' behaviours. The robot family includes three product lines: **Basic Platform**, **Enhanced Obstacle Detection** and **Environmental Vision**. All the members of the Basic Platform share the following features:

- basic means of locomotion that could be *treads*, *wheels*, or *legs*;
- ability to turn by 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 in part due 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 of different types. The optional features of the members of Basic Platform concern their locomotion abilities as well as their locomotion means (treads, wheels or legs).

For instance, in the robot example the subfamily Enhanced Obstacle Detection is constructed on top of basic platform subfamily. For more details we refer the reader to [37,18]. The specification of the robot family is given in Tables 3 and 4.

³ Thompson and Heimdahl [38] say that a product family is multi-dimensional if a hierarchical decomposition is not sufficient to capture its structure. In other terms, it needs to be specified from n perspectives: one family-hierarchy per perspective/view such as software or hardware. A dimension can be perceived as a view in our context.

Product line	Mandatory	Optional	Commonalities
Basic Platform		<ul style="list-style-type: none"> - Speed of locomotion - Limited to low speed of locomotion - Extended to high to high speed of locomotion - Locomotion control system - Basic control (only <i>on</i> or <i>off</i>) - Digital valued indication of locomotion speed and direction - Platform size <ul style="list-style-type: none"> - Small - Medium - Large - Type of collision sensors <ul style="list-style-type: none"> - Pneumatic - Mechanical - Combination of mechanical and pneumatic - Number of collision sensors <ul style="list-style-type: none"> - between 0 and 3 for a small platform - between 0 and 7 for a medium platform - between 0 and 11 for a large platform 	<ul style="list-style-type: none"> - Basic means of locomotion that could be <i>treads, wheels, or legs</i> - 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	<ul style="list-style-type: none"> - Basic Platform with <i>at least one</i> collision sensor 	<ul style="list-style-type: none"> - Type of range finder <ul style="list-style-type: none"> - Small Ultrasonic Range Finder - Low-cost Ultrasonic Ranger - Compact High Performance Ultrasonic Ranger - Number of range finders <ul style="list-style-type: none"> - between 0 and 1 for a small platform - between 0 and 2 for a medium platform - between 0 and 3 for a large platform 	<ul style="list-style-type: none"> - One range finder
Environmental Vision	<ul style="list-style-type: none"> - Enhanced Obstacle Detection 	<ul style="list-style-type: none"> - Environmental vision system - Back and white vision - Primary colour vision 	<ul style="list-style-type: none"> - Sensor capable of determining the colour of objects in the robot's environment

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

Product line	Mandatory	Optional	Commonalities
Random Exploration	<ul style="list-style-type: none"> - Attempt to avoid colliding with obstacles 	<ul style="list-style-type: none"> - 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 behaviour in the absence of an obstacle 	<ul style="list-style-type: none"> - Avoid colliding - First collision recovery - Normal behaviour in the absence of an obstacle, collision, or any other specified behaviour (move forward at maximum speed)
Random Exploration with Ability to Negotiate Doors	<ul style="list-style-type: none"> - Random exploration - Navigate through doors 	<ul style="list-style-type: none"> - Small platform navigation through a door - Medium platform navigation through a door - Large platform navigation through a door 	<ul style="list-style-type: none"> - Locate doors in its environment
Random Exploration with Ability to Signal Encounter of Objects with Particular Color	<ul style="list-style-type: none"> - Random exploration - Environmental vision 		<ul style="list-style-type: none"> - Detect an object of a specified colour

Table 4 Commonalities and variability of a robot family (a behaviour perspective)

6 Building Product Families and Generating Desirable Products

In this section we show the use of product family algebra in finding common features, building up product families, finding new products and excluding undesirable 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 artefacts within systems (e.g. chips, software modules, etc.) enhances hardware/software re-use. If we look at product family algebras like \mathbb{PFS} and \mathbb{PFB} 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, easy and efficient⁴, whereas finding commonalities using diagrams is more complex.

Such calculations can easily be done by our prototype. 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 line. 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 f -carrying subfamilies. 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 from Section 4, we calculate that the Basic Platform 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 product family algebra by postulating the additional equation

$$\text{wheels} \cdot \text{legs} = 0 .$$

This exclusion functionality is also implemented in our prototype. For the robot example all the required exclusion postulates are

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

⁴ A standard algorithmic result is $\text{gcd}(m, n) \in \mathcal{O}(\ln(n))$ for natural numbers m and n . When using the \mathbb{PFB} -isomorphic model of natural numbers as described at Page 10, this result directly transfers to our setting.

```

.+ small_pltfrm .* c_sensor .^ 4
.+ medium_pltfrm .* c_sensor .^ 5
.+ large_pltfrm .* c_sensor .^ 6

```

Here \cdot^{\wedge} is the power operator. For example “ $c_sensor \cdot^{\wedge} 4$ ” is an abbreviation for the term $c_sensor \cdot c_sensor \cdot c_sensor \cdot c_sensor$. Due to the fact that 0 is an annihilator for \cdot , the last line excludes large platforms with more than 5 collision sensors. We refer the reader to Appendix B for the specification of the robot family. Altogether we are left with a Basic Platform consisting of 1539 products; this means a reduction by about 90%.

In the next section we will discuss exclusion in a more general setting.

7 Requirements: Implications and Exclusions

When the specification of a product or a family of products is tackled by adopting a multi-view approach, constraints on the integration of the views need to be elicited as well. These constraints very often link the presence of a feature in a partial description in one view to that of another feature in the same or another view. They can link subproducts or subfamilies as well. A common informal formulation of these constraints reads as follows:

“If a member of a product family has property p_1
it must also have property p_2 ” or
“If a member of a product family has property p_1
it must not have property p_2 ”.

An example for the latter type was already given in the previous section, where we excluded robots with wheels and legs from our production pipeline.

To formulate such integration constraints in our algebra we introduce the following *requirement relation* $a \xrightarrow{e} b$ for elements a, b and e . Informally, $a \xrightarrow{e} b$ means that if e has a then it also has b , or, in other words, that a implies b within e , whence our notation.

Definition 7.1 Assume a feature-generated algebra. For elements a, b, c, d and product p we define, in a family-induction style,

$$\begin{aligned}
 a \xrightarrow{p} b &\iff_{df} (p \sqsubseteq a \implies p \sqsubseteq b) , \\
 a \xrightarrow{c+d} b &\iff_{df} a \xrightarrow{c} b \wedge a \xrightarrow{d} b .
 \end{aligned}$$

Now $a \xrightarrow{e} b$ is well defined for all e , since by assumption e can be written as a finite sum of products. If a and b are products then $a \xrightarrow{e} b$ coincides with $a \xrightarrow{e} l$ where l is the least common multiple of a and b . In the bag model the least common multiple of two bags p and q is the “smallest” bag refined by p and q . For example, assume the features `wheel` and `axis`. Then the least common multiple of `wheel4·axis` and `wheel3·axis2` is `wheel4·axis2` (where a^n denotes the n th power of a). The requirement that in a product line a two wheels need an axis is expressed by `wheel2 \xrightarrow{a} axis`. Later on, we present more examples of the requirement relation.

We now establish some connections between our various relations.

Lemma 7.2 *Let a, b, c, d be elements of a feature-generated algebra.*

- (a) $\overset{a}{\rightarrow}$ is a preorder.
- (b) Let $b \sqsubseteq c$, then $c \overset{a}{\rightarrow} d \implies b \overset{a}{\rightarrow} d$ and $d \overset{a}{\rightarrow} b \implies d \overset{a}{\rightarrow} c$.
In particular, $b \sqsubseteq c \implies b \overset{a}{\rightarrow} c$.
- (c) Let $b \leq c$, then $c \overset{a}{\rightarrow} d \implies b \overset{a}{\rightarrow} d$ and $d \overset{a}{\rightarrow} b \implies d \overset{a}{\rightarrow} c$.

Lemma 7.3 *Let a, b, c, d, p be elements of a feature-generated algebra.*

- (a) $b \overset{a}{\rightarrow} b + c$.
- (b) $b \cdot c \overset{a}{\rightarrow} b$.
- (c) $b \overset{a}{\rightarrow} c \implies b \overset{a}{\rightarrow} c + d$.
- (d) $b \overset{a}{\rightarrow} d \implies b \cdot c \overset{a}{\rightarrow} d$.
- (e) If p is a product, then $b \overset{p}{\rightarrow} c \implies b + d \overset{p}{\rightarrow} c + d$.
- (f) $a \overset{e}{\rightarrow} b \wedge c \overset{e}{\rightarrow} d \implies a \cdot c \overset{e}{\rightarrow} b \wedge a \cdot c \overset{e}{\rightarrow} d$.
- (g) $a + b \overset{e}{\rightarrow} c \iff a \overset{e}{\rightarrow} c \wedge b \overset{e}{\rightarrow} c$.

Let us explain these properties informally. Part (a) means that if a family a has b then it also has a realisation of the choice between between b and an arbitrary c . According to Part (b), if a family a has $b \cdot c$, i.e., both b and c are mandatory for a , then it has to have b (and also c , since \cdot is commutative). Parts (c), (d) and (e) allow embedding already derived feature implications into larger contexts. Part (f) means that mandatory inclusion of two families implies all families that are implied by the included ones separately. Finally, Part (g) expresses that a choice guarantees a feature if and only if both its parts do.

Before looking at the multi-view reconciliation problem we will give some small examples of how the above relation can be used.

Example 7.4 In the remainder we assume that a vehicle is built up from the components (features) `speed_indicator`, `steering_wheel`, `wheel`, `axis`, `engine`, `standard_transmission` and `automatic_transmission`.

- The requirement `engine` $\overset{\text{car}}{\rightarrow}$ `speed_indicator` enforces that every motor vehicle of the family `car` has also a speed indicator.
- By `wheel` \cdot `wheel` $\overset{\text{car}}{\rightarrow}$ `steering_wheel` and `engine` $\overset{\text{car}}{\rightarrow}$ `steering_wheel` we require that there is at least one steering wheel if the vehicle has at least two wheels or one engine.
- To exclude more than one steering wheel, one can use the requirement `(steering_wheel) \cdot (steering_wheel)` $\overset{\text{car}}{\rightarrow}$ `0`.
- By `(wheel \cdot wheel)^n` $\overset{\text{car}}{\rightarrow}$ `axis^n` (for all $n \in \mathbb{N}$) we enforce that each pair of wheels can be connected by its own axis.
- To express that a car has to have an even number of wheels we can use `wheel^{2n+1}` $\overset{\text{car}}{\rightarrow}$ `wheel^{2n+2}`. \square

So far we have used only requirements for products. However, our requirement relation can also be used more generally. For instance, we may wish to express the following:

“If a member of product family a has feature p_1 it also needs to have feature p_2 or p_3 ”.

For this we may simply write $p_1 \xrightarrow{a} p_2 + p_3$.

Example 7.5 The formula

`engine` $\xrightarrow{\text{car}}$ `standard_transmission + automatic_transmission`

requires a car with an engine to also have a standard transmission or an automatic one. \square

An application of such an integration constraint occurs, e.g., when sensors are used (see Section 9), because then very often several technologies are adopted. We can have requirements demanding that either of the technologies be used. Last, but not least, one can use the product family 1 consisting just of the empty product to guarantee the existence of other elements.

Example 7.6 The requirement $1 \xrightarrow{\text{car}} \text{engine}$ enforces that each car has (at least) one engine. \square

The third item in Example 7.4 shows how to describe exclusion using \xrightarrow{a} . While a global mutual exclusion of products p and q can be expressed by the additional axiom $p \cdot q = 0$, practically expressing exclusion using \xrightarrow{a} is more suitable. Very often we exclude combination of features only within a particular product (or family) a . The exclusion using \xrightarrow{a} has scope a , whereas $p \cdot q = 0$ does not have an explicit scope. Therefore our requirement relation fits well with the exclusion concept of Section 5.

Finally, to express global product implication, one might define

$$b \xrightarrow{*} c \iff_{df} \forall a : b \xrightarrow{a} c . \quad (5)$$

However, this relation is redundant by the following result.

Lemma 7.7 *Let b, c be elements of a feature-generated algebra. Then $b \xrightarrow{*} c \iff b \sqsubseteq c$, i.e., $\xrightarrow{*}$ coincides with \sqsubseteq .*

In particular, $b \xrightarrow{} 0 \iff b \sqsubseteq 0 \iff b = 0$.*

8 Multi-View Reconciliation

In this section we sketch the multi-view reconciliation problem. Later on, we illustrate the problem with a small example. In Section 9 we will present a larger case study.

When we approach the specification of a product family from different perspectives these are usually somehow interdependent. When this interdependence is known, how can we describe it in the form of a set of integration constraints?

We will show that simple multiplication, i.e., the Cartesian product, of families, combined with the requirement relation yields the desired behaviour. On the basis of our algebra we can tackle the feature reconciliation problem in the following way:

- Take two product families a and b and a set of implication clauses of the form $c \xrightarrow{a \cdot b} d$.
- Write a and b in sum-of-products form.
- Now form $a \cdot b$, multiplying out and removing all products from the resulting sum that do not respect the implication clauses.

Example 8.1 We assume a company that produces computers. In particular, it builds machines with a hard disk and a screen. Additionally, a second screen, a printer or a scanner can be ordered. Of course, it is possible to have more than one extension for the basic computer. Using the abbreviations `hd`, `scr`, `prn` and `scn`, this yields the following element in product family algebra:

$$\text{hw} = \text{hd} \cdot \text{scr} \cdot \text{opt}[\text{scn}, \text{prn}, \text{scr}] .$$

In fact the company produces exactly 8 different machines. Next to the company producing hardware, we assume a software company implementing drivers. At the moment it offers only two different software packages.

$$\begin{aligned} \text{sw} &= \text{hd_drv} \cdot \text{scr_drv} \cdot \text{prn_drv} \\ &+ \text{hd_drv} \cdot \text{scr_drv} \cdot \text{scn_drv} \end{aligned}$$

The first one contains drivers for hard disks, screens and printers; the second for hard disks, screens and scanners. The multi-view reconciliation problem asks for all products that satisfy the following requirements:

$$\begin{array}{l} \text{hd} \xrightarrow{\text{hw} \cdot \text{sw}} \text{hd_drv} \\ \text{scr} \xrightarrow{\text{hw} \cdot \text{sw}} \text{scr_drv} \\ \text{prn} \xrightarrow{\text{hw} \cdot \text{sw}} \text{prn_drv} \\ \text{scn} \xrightarrow{\text{hw} \cdot \text{sw}} \text{scn_drv} \end{array}$$

These requirements guarantee that each hardware component has an appropriate driver. For this, in our HASKELL implementation the function `reconc` takes two product families a and b and a list of pairs (c, d) that represent requirements $c \xrightarrow{a \cdot b} d$ and solves the multi-view reconciliation problem by the above procedure. Hence the call

```
reconc hw sw
  [(hd, hd_drv), (scr, scr_drv), (prn, prn_drv), (scn, scn_drv)]
```

determines all desired products, 8 in number.

Hardware	Software
hard disk	hard disk driver
printer	printer driver
screen (2x)	screen driver

hard disk printer screen	hard disk driver printer driver screen driver
hard disk screen	hard disk driver printer driver screen driver
hard disk screen	hard disk driver scanner driver screen driver
hard disk ...	hard disk driver

Let us have a closer look at the result set. First, there is no machine with scanner and printer. This is due to the fact that there is no software package having drivers for both components. Furthermore, there are two different versions of the hardware product consisting of hard disk and screen(s) only. The versions offer software for scanners and printers, resp. Such products can be seen as hardware with an upgrade option. That means that the customer can add a hardware component without changing the software.

□

Symmetrically to the combination of product lines, one can extract a view of a product family by simple projection to the respective feature set F using a product family algebra homomorphism that sends all features outside F to the empty product 1.

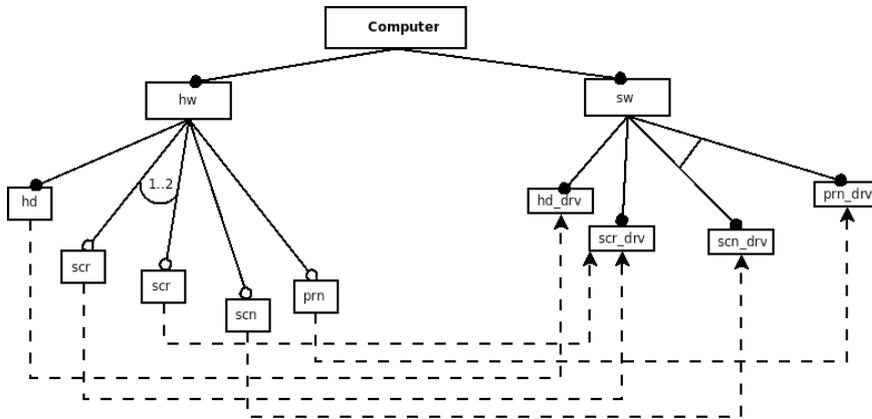


Fig. 2 FORE feature diagram corresponding to the system of Example 8.1

To illustrate again the relationship between our approach to specify product families and the graphical feature modelling approaches, we provide in Figure 2 the graphical representation of the product family subject

of Example 8.1. The graphical notation used is that of FORE [11,12], an extension of pure FODA feature diagrams (cf. Section 3.2). The root is labelled with the name given to the overall family. The edges are used to progressively decompose it into smaller components. A leaf corresponds to a *feature*. The features indicated with hollow circles are optional. Vertices with black circles are considered as mandatory. The features `prn_drv` and `scn_drv` are related with an xor graphical connector. Multiplicities are represented with two integers giving the lower and the upper bound. For instance, in Figure 2, the number of screens for each product ranges between 1 and 2. Alternatively and without using multiplicity, expressing the possibility of having at least one screen and at most two screens can be graphically represented with one mandatory `scr` and another optional.

The subtree rooted at `hw` does not share basic features with that having `sw` as root. Both subtrees give orthogonal views of the *Computer* family: `hw` and `sw`. However, they are related with *requires* constraints represented as dashed arrows. They express the same constraints given above. From this simple example, we notice that to express dependences between elements of several views, we need to represent all the views needed in a same diagram. Therefore, for systems with several views the obtained diagram can be extremely large. Our approach allows to specify each view separately. Then, the constraints that link several views are articulated.

9 Case Study: Driver Assisting Systems

The following case study is based on a simplified family of *Driver Assisting Systems* (DAS). It is partially adapted from a discussion on control problems in vehicle system design given in [30]. Applying the separation of concerns principle, we present the family DAS from two perspectives: a functional perspective and a sensor/actuator perspective. The first articulates the family using the main functional features of DAS. The latter perspective includes only the sensors and actuators needed by the family.

9.1 Functional View

The functional description is built up from the following basic functional components (features):

<i>Road sign recognition and indication</i>	<code>rd_sgn_rcg,</code>
<i>Far Infra-Red (FIR) detection</i>	<code>fir,</code>
<i>Near Infra-Red (NIR) detection</i>	<code>nir,</code>
<i>Thermal imaging detection</i>	<code>tid,</code>
<i>Line departure warning</i>	<code>ldw,</code>
<i>Blind spot monitoring</i>	<code>bsm,</code>
<i>Adaptive-Cruise-Control-(ACC-) following control</i>	<code>acc_f_c,</code>
<i>Emergency braking</i>	<code>e_braking,</code>
<i>Urban ACC (stop & go)</i>	<code>u_acc,</code>
<i>Automatic steering and braking</i>	<code>aut_str_brk,</code>
<i>Automatic line keeping</i>	<code>aut_line,</code>
<i>Obstacle avoidance</i>	<code>obst_avd,</code>
<i>Obstacle warning</i>	<code>obst_wrng.</code>

More advanced composite products are combinations of the above components. For example, the functional description of night vision consists of any combination of far infra-red technology (`fir`), near infra-red technology (`nir`), or a simple thermal imaging technology (`tid`). However, it has to have at least one of these features and `tid` is a mandatory feature of `n_vision`. We can express this requirement as

$$\text{n_vision} = \text{opt}[\text{fir}, \text{nir}] \cdot \text{tid}$$

The *driver information and warning* (`driver_i_w`) is the combination of the three mandatory basic features `rd_sgn_rcg`, `ldw` and `bsm`, and the ability of night vision.

$$\text{driver_i_w} = \text{rd_sgn_rcg} \cdot \text{ldw} \cdot \text{n_vision} \cdot \text{bsm}$$

To describe the complete product family for the driver assisting system from the functional perspective, we use further composite components for *automatic longitude control* (`aut_long_ctrl`) and *automatic lateral control* (`aut_ltrl_ctrl`). The component `aut_long_ctrl` allows to automatically adjust speed in order to maintain a proper distance between vehicles on a same line in urban environment or on highways. It is defined in terms of features as follows:

$$\text{aut_long_ctrl} = \text{acc_f_c} \cdot \text{opt}[\text{e_braking}, \text{u_acc}]$$

The main purpose of the component `aut_ltrl_ctrl` is to automatically steer or to assist the driver in steering the vehicle. It might include line keeping, line change, merging, or diverging. It is defined as follows:

$$\text{aut_ltrl_ctrl} = \text{opt}[\text{aut_str_brk}, \text{aut_line}]$$

The whole product line is then characterised by

$$\begin{aligned} \text{p_line_driver_assist_sys} = \\ \text{obst_wrng} \cdot \text{opt}[\text{obst_avd}, \text{driver_i_w}] \cdot \\ \text{opt}[\text{aut_long_ctrl}, \text{aut_ltrl_ctrl}] \end{aligned}$$

where `opt[...]` describes again optional features.

It is easy to see that this product family contains products with both FIR and NIR technologies. From an industrial point of view, if both technologies occur, one of them is just redundant and would only generate extra costs. Therefore, we use the requirement

$$\text{fir} \cdot \text{nir} \xrightarrow{\text{p_line_driver_assist_sys}} 0.$$

The reduced result `res_p_line_driver_assist_sys` yields a size reduction from 200 to 160 members (i.e., size reduction of 20%). In real life the early detection of conflicting features prevents constructing dysfunctional products; otherwise, an immense increase of costs is incurred by attempting their construction. Moreover, by simple algebraic calculations done automatically by our prototype, we can list its common features.

```
printfeat(common res_p_line_driver_assist_sys)
```

shows that obstacle warning (`obst_wrng`) is the only common feature. This result shows that (a) every product of the driver assisting system must have such a warning system and (b) that the company can produce one single version of such a system for all its products.

In the next subsection, we focus on another view. Instead of discussing functional descriptions of our system we now focus on sensors and actuators.

9.2 Sensor/Actuator View

This view describes the kind of sensors and actuators needed by the family to gather the information necessary for the above functional features and for controlling the mechanisms of DAS products.

Similar to the functional view, we first list the basic features for the actuator and sensor view:

<i>Acceleration pulsator</i>	<code>acclrt_pulsator,</code>
<i>Acceleration-of-wheel sensor</i>	<code>acclrt_wheel,</code>
<i>Acceleration-of-vehicle-body sensor</i>	<code>acclrt_body,</code>
<i>Displacement-of-wheel sensor</i>	<code>dis_wheel,</code>
<i>Displacement-of-vehicle-body sensor</i>	<code>dis_body,</code>
<i>Brake temperature sensor</i>	<code>brk_temp,</code>
<i>CO₂ sensor</i>	<code>co_snsr,</code>
<i>Position sensor</i>	<code>position,</code>
<i>Load data sensor</i>	<code>load,</code>
<i>ACC radar</i>	<code>acc_radar,</code>
<i>ACC laser</i>	<code>acc_laser,</code>
<i>ACC video camera</i>	<code>acc_v_cam,</code>
<i>ACC IR camera</i>	<code>acc_ir_cam,</code>
<i>ACC Far-IR camera</i>	<code>acc_far_ir.</code>

To describe the complete on-board sensor configuration we use the following composite features:

<i>Acceleration Sensors</i>	<code>acclrt_sensors,</code>
<i>Displacement Sensors</i>	<code>dis_sensors,</code>
<i>Adaptive Cruise Control Sensors</i>	<code>acc_sensors.</code>

The composite feature `acclrt_sensors` is a collection of sensors that provide information on the acceleration of the moving (rotating) parts of the vehicle. It is specified in terms of basic features as follows:

$$\text{acclrt_sensors} = \text{acclrt_pulsator} \cdot \text{acclrt_wheel} \cdot \text{acclrt_body} .$$

The composite feature `dis_sensors` is formed by displacement sensors that can be placed, for instance, in pairs on either side of an automotive brake disc to dynamically monitor an assortment of performance parameters. They accurately measure position and displacement. Hence we have

$$\text{dis_sensors} = \text{dis_wheel} \cdot \text{dis_body}$$

The composite feature *adaptive cruise control sensors* (`acc_sensors`) specifies the combinations of sensors that supply the system with the information needed to automatically adjust a vehicle's speed to maintain a safe following distance. In the following specification of `acc_sensors`, constraints are put on the maximum number of occurrences of some features.

$$\text{acc_sensors} = \text{acc_radar}^{\leq 2} \cdot \text{opt}[\text{acc_laser}] \cdot \text{acc_v_cam}^{\leq 4} \cdot \text{acc_ir_cam}^{\leq 4} \cdot \text{acc_far_ir}^{\leq 4}$$

where for product family a and natural number n we set $a^{\leq n} = (\text{opt}[a])^n$. By definition of `opt[.]` and distributivity $a^{\leq n} = 1 + a + a^2 + \dots + a^n$. The complete description of all on-board sensors then is

$$\text{on_board} = \text{opt}[\text{co_snsr}] \cdot \text{opt}[\text{brk_temp}^4] \cdot \text{acclrt_sensors} \cdot \text{dis_sensors} \cdot \text{acc_sensors} \cdot \text{opt}[\text{position}^8]$$

Similar to the requirement constraint of the functional description view, we have the following exclusion constraints:

$$\begin{aligned} \text{acc_radar} \cdot \text{acc_laser} &\xrightarrow{\text{on_board}} 0 , \\ \text{acc_far_ir} \cdot \text{acc_ir_cam} &\xrightarrow{\text{on_board}} 0 . \end{aligned}$$

The restricted set `res_on_board` of possible sensor configurations is about 76% smaller than the unrestricted version (reduction from 6000 to 1440). Thus, adding simple restriction constraints can yield an immense and useful decrease of the variety of products, which is not surprising.

9.3 View Reconciliation

The functional view and the sensor/actuator view now form the basis for the multi-view reconciliation problem. For the normal behaviour of each product, we need to provide the functional products (as given in the functional view) with the sensor/actuator products needed. Thereby, we need to link these two perspectives by setting up the following requirements:

$$\text{driver_i_w} \xrightarrow{x} \text{acclrt_pulsator} + \text{co_snsr} + \text{position} , \quad (6)$$

$$\text{e_braking} \xrightarrow{x} \text{brk_temp} \cdot \text{position} , \quad (7)$$

$$\text{aut_str_brk} + \text{aut_line} \xrightarrow{x} \text{dis_wheel} \cdot \text{acclrt_body} \cdot \text{load} , \quad (8)$$

where x is the product consisting of the two restricted product lines, i.e.,

$$x = \text{res_p_line_driver_assist_sys} \cdot \text{res_on_board} .$$

The driver information and warning (`driver_i_w`) needs to be fed with streams of information collected from the system environment. This information allows making decisions, for example, on road line changing, and detecting obstacles. For this purpose, one of the `acclrt_pulsator`, `co_snsr`, or `position` is required. This requirement is expressed by (6) above.

In the case of an *emergency braking* (`e_braking`), the sensors have to collect the temperature of the brakes and at the same time the current position has to be checked to react if there is an obstacle in front. Linking `driver_i_w` to its needed sensors is given by requirement (7).

Requirement constraint (8) indicates that either one of `aut_str_brk` or `aut_line` calls for the mandatory presence of `dis_wheel`, `acclrt_body`, and `load`. According to Lemma 7.3(g), requirement (8) can be written as the conjunction of two simpler ones: `aut_str_brk` \xrightarrow{x} `dis_wheel` · `acclrt_body` · `load` and `aut_line` \xrightarrow{x} `dis_wheel` · `acclrt_body` · `load`.

Now we can use the described algorithm to solve the multi-view reconciliation problem. Simple algebraic calculations done automatically by our prototype show that reconciling the two considered view yields a general product family with 40320 products. The full HASKELL code can be found in [19].

10 Related Work

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) [24], Feature-Oriented Reuse Method (FORM) [25], Featured Reuse-Driven Software Engineering Business (FeaturSEB) [14] and Generative Programming (GP) [8]. Other feature modelling techniques are presented in [3].

Similar to feature diagrams, the authors of [33] propose the use of weighted feature diagrams. 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 OR/AND graph [29, pages 40–41 and 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 parametrised reference architectures and appropriate re-usable components instantiable during application development [25].

In FeatuRSEB, the feature model is represented by a graph (not necessarily a tree) of features. The edges are mainly UML dependence relationships: *composed_of*, *optional_feature* and *alternative_relationship*. The graph enables specifying 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 software system families. Its feature modelling aims to capture commonalities and variation points within the family. A feature model includes 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 [8]:

- *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.

To have a feature model, in addition to a feature diagram one needs to provide additional information such as a short description of each feature, constraints on feature combinations and information on the importance of some key features.

Concerning view reconciliation, there is a wide literature on integrating non-functional requirements such as security and performance [7]. For instance, security requires careful scrutinising of data, which could affect a system’s performance. Also, we find approaches to resolve architectural mismatches resulting from integrating commercial off-the-shelf (COTS) components. The mismatches essentially occur between the services required and provided, when a component and its environment interact. However these

approaches do not directly relate to the problem we are treating in the present paper. They tackle the reconciliation of two architectural models: one that is forward-engineered from the requirements specification and a second that is reverse-engineered from the COTS-based system implementation [1]. Also, a similar problem occurs when merging views of a database; it is called the *view reconciliation problem* [23]. The above cases are considering the development of a single software system and not a software family. The mismatches that we are concerned with occur at the level of the feature model in the initial phase of the software development process before the architectural design.

Also, the literature regarding the sequential completion method for the development of software systems proposes a variety of solutions to the view reconciliation problem [9, 39, 42]. The views considered there are partial descriptions (e.g., scenarios or use-cases) of the functional requirements of a system. To build up the overall specification of the system, these partial views are integrated and in this process any inconsistencies among them are detected. In [9] we find a relational approach to the integration of sequential scenarios, which are scenarios involving a single user. The integration is possible if the scenarios are consistent (from a behaviour perspective). Moreover, the proposed technique offers several opportunities for detecting possible sources of requirement incompleteness. In [39], an approach to scenario-based specification, integration and behaviour analysis is proposed. The approach proposes a Message Sequence Charts language that integrates existing approaches based on scenario composition by using high-level Message Sequence Charts. Also, it presents a synthesis algorithm which transforms scenarios into a behavioural specification in the form of Finite Sequential Processes. The obtained specification can be analysed with the Labeled Transition System Analyzer using model checking and animation. In [42] an algebraic framework for view consistency in the elaboration of functional requirements of a system is presented. Viewpoints are formalised as pairs of a syntactic and a semantic category linked by a model functor, while views are objects in the syntactic category. Consistency of views is defined by a heterogeneous pullback construction. The approaches of [9, 39, 42] deal with the integration of partial descriptions of the behaviour of a single system while interacting with its environment.

Formal requirements scenarios can be related to our product family algebra by providing concrete definitions for the two product family algebra operators \cdot and $+$, and providing explicit 0 and 1. For instance, the integration of features using the product is presented as a generalisation of the work presented in [9]. It can be as well represented as a scenario composition using high-level Message Sequence Charts (hMSCs) and then one uses the technique proposed by Uchitel et al. [39] to analyse the obtained concrete model.

11 Conclusion and Future Work

The adoption of the product family paradigm aims at recognising a reality in software development industry noticed decades ago [31]: economical constraints impose a concurrent approach to software development replacing the early sequential one. The research about software product families aims at studying the commonalities/variability occurring among the products in order to have a better management of system production. However, a review of the literature reveals a wide set of notions and terms used without formal definitions. Hence the development of a clear and simple mathematical basis for this paradigm became necessary.

In this paper we have introduced product family 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 [24], FORM [25], FeatuRSEB [14] and GP [8] and expressed with graphical notations can easily be stated within our algebra. For instance, the alternative is expressed using the $+$ operator, and we write $d = b \cdot (1 + a) \cdot c$ (where b , and c are families) to express that a feature a is optional in a family d .

In contrast to other product family specification formalisms, like FODA, FORM, and the other extensions to FODA, 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 re-usability.

Many items from the literature support the potential scalability of algebraic approaches in specifying industrial-scale software product families [2, 4]. However, we think that empirical substantiation of the scalability of our approach is needed.

As a major application of our algebraic framework we have presented a way of solving the multi-view reconciliation problem. The main ingredient is a set of integration constraints that link features or more generally sub-families in one view description to other features or sub-families in the same or another view description. The integration process leads to a more accurate specification of a product family by excluding the members that do not satisfy the integration constraints. The description of a family as well as the integration constraints are given within the same mathematical framework

of product family algebra. We have presented the mathematical properties of a requirement relation that we use to express the view integration constraints. Several examples have shown the capabilities of this approach for dealing with a wide class of integration constraint formulations.

The main characteristics of the proposed approach are the following:

- The conflict resolution among views is performed without modification on the initial views. It is a direct application of the principle of separation of concerns. Each specifier can concentrate on capturing a description of a product family from his own view without being constrained to conform to some other specifier’s view. In a second step one can focus the attention on the constraints that govern the integration of the considered views. The global view of the product family is then obtained by simple algebraic manipulations. This approach is suitable for graceful ageing and evolution of product family specifications: each time a view changes the global view can be automatically re-generated.
- The mathematical background needed to specify product family views as well as the integration constraints involves only simple concepts that we can realistically expect all stakeholders to understand and relate to.
- Due to the simplicity of the mathematical framework, the reasoning on product families as well as on view integration can be automated in provers such as Prover9 [27] and prototypically implemented over some useful models of product family algebra in HASKELL.

Our product family algebra is at a high level of abstraction. From a software perspective, a feature could be a requirement scenario/use-case or a partial description of the functionality. Our current research aims at investigating the derivation of the specifications of members of a family from its abstract feature algebra specification and the specifications of each of its features. This step would join the ongoing research efforts for formal model driven software development techniques. The algebraic model of a family and the specifications of the family’s features would be the initial models of the sought transformation.

Acknowledgements: We are grateful to A. Zelend and the anonymous referees for helpful discussions and remarks.

References

1. Paris Avgeriou and Nicolas Guelfi. Resolving architectural mismatches of COTS through architectural reconciliation. In *Lecture Notes in Computer Science*, volume 3412, pages 248–257. Springer, 2005.
2. Don Batory. The road to utopia: A future for generative programming. Keynote presentation at the Dagstuhl Seminar for Domain Specific Program Generation, March 2003.
3. Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Software Product Line Conference (SPLC-EUROPE 2005)*, 26-29 September 2005.

4. Don Batory, Roberto Lopez-Herrejon, and Jean-Philippe Martin. Generating product-lines of product-families. In *Conference on Automated-Software Engineering*, September 2002.
5. Stan Böhne, Kim Lauenroth, and Klaus 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.
6. Paul Clements, Linda M. Northrop, and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Professional, 2002.
7. L. M. Cysneiros and J. C. S. do Prado Leite. Non-functional requirements: From elicitation to conceptual models. *IEEE Transactions on Software Engineering*, 30(5):328–350, 2004.
8. Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming, Methods, Tools and Applications*. Addison-Wesley, 2000.
9. Jules Desharnais, Marc Frappier, and Ridha Khedri. Integration of sequential scenarios. *IEEE Transactions on Software Engineering*, 24(9):695–708, September 1998.
10. Jules Desharnais, Bernhard Möller, and Georg Struth. Modal Kleene algebra and applications — A survey. *Journal on Relational Methods in Computer Science*, 1:93–131, 2004.
11. Detlef Streitferdt. *Family-Oriented Requirements Engineering*. PhD thesis, Technical University Ilmenau, 2004.
12. Detlef Streitferdt and Matthias Riebisch and Ilka Philippow. Details of Formalized Relations in Feature Models Using OCL. In *The 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03)*, Huntsville, AL, USA, April 2003. IEEE Computer Society.
13. Wasif Gilani, Nabeel Hasan Naqvi, and Olaf Spinczyk. On adaptable middleware product lines. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware (ARM '04)*, pages 207–213. ACM, 2004.
14. Martin L. Griss, John Favaro, and Massimo 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.
15. Uwe Heibisch and Hanns J. Weinert. *Semirings — Algebraic Theory and Applications in Computer Science*. World Scientific, 1998.
16. P. Höfner. Database for automated proofs (laws for product families). <http://www.dcs.shef.ac.uk/~georg/ka>. (accessed March 1, 2009).
17. Peter Höfner, Ridha Khedri, and Bernhard Möller. Feature algebra. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2006.
18. Peter Höfner, Ridha Khedri, and Bernhard Möller. Feature algebra. Technical Report 2006-04, Institute of Computer Science, University of Augsburg, 2006.
19. Peter Höfner, Ridha Khedri, and Bernhard Möller. Algebraic view reconciliation. Technical Report 2007-13, Institute of Computer Science, University of Augsburg, 2007.
20. Peter Höfner, Ridha Khedri, and Bernhard Möller. Algebraic view reconciliation. In *Software Engineering and Formal Methods*, pages 85–94. IEEE Press, 2008.

21. Peter Höfner and Georg Struth. Automated reasoning in Kleene algebra. In Frank Pfennig, editor, *CADE 2007*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 279–294. Springer, 2007.
22. Peter Höfner and Georg Struth. Can refinement be automated? *Electronic Notes in Theoretical Computer Science*, 201:197–222, 2008.
23. Barry E. Jacobs. *Applied Database Logic: Fundamental Issues*, volume I. Prentice-Hall, Inc., 1985.
24. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon Software Engineering Institute, Carnegie Mellon University, 1990.
25. Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.
26. Ridha Khedri. Formal model driven approach to deal with requirements volatility. Computing and Software Technical Reports CAS-08-03-RK, Department of Computing and Software, McMaster University, 2008.
27. William McCune. Prover9 and Mace4. <http://www.prover9.org/>. (accessed March 1, 2009).
28. William Nace and Philip Koopman. A product family approach to graceful degradation. In *Proceedings of the IFIP WG10.3/WG10.4/WG10.5 International Workshop on Distributed and Parallel Embedded Systems: Architecture and Design of Distributed Embedded Systems*, pages 131–140. Kluwer, B.V. Denter, The Netherlands, The Netherlands, 2000.
29. Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co., 1980.
30. Laszlo Palkovics. Identification and control problems in vehicle system design. Knorr-Bremse Publication, 1991.
31. David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE2(1):1–9, 1976.
32. Claudio Riva and Christian Del 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.
33. Silva Roback and Andrzej 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.
34. Juha Savolainen, Ian Oliver, Mike Mannion, and Hailang Zuo. Transitioning from product line requirements to product line architecture. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, pages 186–195. IEEE Computer Society, 26–28 July 2005.
35. Charles P. Shelton. Using architectural properties to model and measure system-wide graceful degradation. In *In Workshop on Architecting Dependable Systems*, pages 267–289. Springer, 2002.
36. Software Engineering Standards Committee of the IEEE Computer Society. IEEE recommended practice for software requirements specifications, IEEE Std 830-1998 (revision of IEEE std 830-1993). <http://ieeexplore.ieee.org> (May 23, 2007).
37. Jeffrey M. Thompson, Mats P. Heimdahl, and Debra 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.

38. Jeffrey M. Thompson and Mats P.E. Heimdahl. Structuring product family requirements for n-dimensional and hierarchical product lines. *Requirements Engineering Journal*, 2002.
39. Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.
40. Andy Ju An Wang and Kai Qian. *Component-Oriented Programming*. Wiley, 2005.
41. David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison Wesley Longman, Inc., 1999.
42. Martin Wirsing and Alexander Knapp. View consistency in software development. In *Lecture Notes in Computer Science*, volume 2941, pages 341–357. Springer, 2004.

A Deferred Proofs

Proof of Lemma 3.11.

Since in a semiring $a + b$ is the least upper bound of a and b w.r.t. the natural order, we have the equivalence

$$a + b \leq c \iff a \leq c \wedge b \leq c. \quad (9)$$

Suppose now $p_1 + \dots + p_m \leq q_1 + \dots + q_n$ for distinct proper p_i and distinct proper q_j . Then by (9) we have $p_i \leq q_1 + \dots + q_n$ for all i . Since the p_i as products are irreducible (Formula (1)), we obtain $p_i \leq q_1 \vee \dots \vee p_i \leq q_n$. Since all p_i are assumed to be proper and all q_j are products, atomicity of the q_j (Formula (2)) tells us that this disjunction is equivalent to $p_i = q_1 \vee \dots \vee p_i = q_n$. But this means that $p_i \in \{q_1, \dots, q_n\}$. Hence $\{p_1, \dots, p_m\} \subseteq \{q_1, \dots, q_n\}$.

The converse implication is trivial. □

Proof of Lemma 3.12.

The equation $p_1 + \dots + p_m = q_1 + \dots + q_n$ is equivalent to $p_1 + \dots + p_m \leq q_1 + \dots + q_n \wedge q_1 + \dots + q_n \leq p_1 + \dots + p_m$. Now Lemma 3.11 shows $\{p_1, \dots, p_m\} \subseteq \{q_1, \dots, q_n\} \wedge \{q_1, \dots, q_n\} \subseteq \{p_1, \dots, p_m\}$ which is equivalent to the claim. □

Proof of Lemma 3.15.

Assume $a \mid b$, say $a \cdot c = b$. Then by reflexivity of \leq also $b \leq a \cdot c$, showing $b \sqsubseteq a$. □

Proof of Lemma 3.16.

(\Leftarrow) follows immediately from Lemma 3.15. For the converse direction, assume $a \leq p \cdot c$ for some element c . If $a = 0$ then $p \mid a$ holds. Otherwise we must have $p \neq 0$ and $c \neq 0$. In particular, p is a proper product and there are

sets $\{p_1, \dots, p_m\}$ and $\{r_1, \dots, r_n\}$ of proper products with $a = p_1 + \dots + p_m$ and $c = r_1 + \dots + r_n$. By distributivity now the assumption $a \leq p \cdot c$ is equivalent to $p_1 + \dots + p_m \leq p \cdot r_1 + \dots + p \cdot r_n$. By Definition 3.8 all $p \cdot r_i$ are products again. Let $P =_{df} \{p \cdot r_i \mid p \cdot r_i \neq 0\}$ be the set of all non-trivial summands. Since $p \neq 0$ we must have $P \neq \emptyset$. This means that there are indices $j_1, \dots, j_m \in \{1, \dots, n\}$ with $\{p_1, \dots, p_m\} = \{p \cdot r_{j_1}, \dots, p \cdot r_{j_m}\}$ and hence $a = p_1 + \dots + p_m = p \cdot r_{j_1} + \dots + p \cdot r_{j_m} = p \cdot (r_{j_1} + \dots + r_{j_m})$ by distributivity, so that p divides a . \square

Proof of Lemma 3.17.

- (a) Set $c = 1$ in the definition of \sqsubseteq .
- (b) $a \cdot b \sqsubseteq b \iff \exists c : a \cdot b \leq b \cdot c \iff a \cdot b \leq b \cdot a \iff \text{true}$.
The last step only holds if \cdot is commutative.
- (c) Immediate from $a \leq a + b$ and Part (a).
- (d) 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., $a + c \sqsubseteq b + c$.

- (e) By definition, isotony w.r.t. \leq and commutativity we get
 $a \sqsubseteq b \iff \exists d : a \leq b \cdot d \implies \exists d : a \cdot c \leq b \cdot c \cdot d \iff a \cdot c \sqsubseteq b \cdot c$.
- (f) By annihilation, $a \sqsubseteq 0 \iff \exists c : a \leq 0 \cdot c \iff a \leq 0$.
- (g) Set $a = 0$ and $b = 1$, resp., in Part (b). \square

Proof of Lemma 3.18.

- (a) (\Leftarrow) follows by Lemma 3.17(a) and transitivity of \sqsubseteq .
 (\Rightarrow) Assume $p \leq (a + b) \cdot c = a \cdot c + b \cdot c$. Since p is a product, we have $p \leq a \cdot c \vee p \leq b \cdot c$, which shows the claim.
- (b) (\Rightarrow) Let $p \sqsubseteq a + b$, say $p \leq (a + b) \cdot c = a \cdot c + b \cdot c$. Since p is a product, this implies $p \leq a \cdot c \vee p \leq b \cdot c$, showing $p \sqsubseteq a \vee p \sqsubseteq b$.
 (\Leftarrow) follows by \sqsubseteq -isotony (Lemma 3.17(e)) and idempotence of $+$. \square

Proof of Lemma 3.19.

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

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

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

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

Proof of Lemma 3.23.

- (a) Assume y and z to be weak zeros. Then, by definition, $y = y \cdot z = z$.
- (b) (\Rightarrow) Assume $a = 0$. Then by definition of weak zero and annihilation $z = 0$, which contradicts the definition of z .
 (\Leftarrow) By definition $z \leq z \cdot a$ if $a \neq 0$ and hence, $z \sqsubseteq a$.

- (c) By definition of \sqsubseteq and weak zero,
 $a \sqsubseteq z \iff \exists c : a \leq z \cdot c \iff a \leq 0 \vee a \leq z \iff a \leq z.$ \square

Proof of Lemma 7.2.

The claims are shown by family induction. We only give the induction base cases; the induction steps are straightforward predicate logic.

- (a) Reflexivity follows immediately from the definition. Transitivity holds by transitivity of implication.
 (b) Let p be a product. First we show $c \xrightarrow{p} d \implies b \xrightarrow{p} d$. Therefore we assume $b \sqsubseteq c$, $c \xrightarrow{p} d$ and $p \sqsubseteq b$. Then by transitivity of \sqsubseteq we get $p \sqsubseteq c$ and hence also $p \sqsubseteq d$. The second claim is proved similarly.

For the third claim set $d = c$ in the first claim or $d = b$ in the second claim and use reflexivity of \xrightarrow{a} .

- (c) Immediate from (b) using $b \leq c \implies b \sqsubseteq c$. \square

Proof of Lemma 7.3.

Again the claims are shown by family induction for which we only do the base cases.

- (a) By Lemma 3.18(a) $q \sqsubseteq b$ implies $q \sqsubseteq b + c$ by $b \sqsubseteq b + c$ and transitivity of \sqsubseteq .
 (b) Assume $q \sqsubseteq b \cdot c$, i.e., $\exists f.q \leq b \cdot c \cdot f$. Setting $c' =_{df} c \cdot f$ shows $q \sqsubseteq b$.
 (c) Immediate from Lemma 7.2(b) by $c \sqsubseteq b + c$.
 (d) Immediate from Lemma 7.2(b) by $b \cdot c \sqsubseteq b$.
 (e) Assume $p \sqsubseteq b + d$. Since p is a product, this implies $p \sqsubseteq b$ or $p \sqsubseteq d$. In the first case, $p \sqsubseteq c \sqsubseteq c + d$ by $b \xrightarrow{p} c$ and Lemma 3.18(b). In the second case $p \sqsubseteq d \sqsubseteq c + d$.

Note that this property cannot be lifted to arbitrary elements using the sum of products form, since we use a special property of products.

- (f) Immediate from Part (c).
 (g) By definition of \xrightarrow{p} , Lemma 3.18(b), predicate logic and definition of \xrightarrow{p} again,

$$\begin{aligned} & (e + f \xrightarrow{p} c) \\ \iff & (p \sqsubseteq e + f \implies p \sqsubseteq c) \\ \iff & ((p \sqsubseteq e \implies p \sqsubseteq c) \wedge (p \sqsubseteq f \implies p \sqsubseteq c)) \\ \iff & (e \xrightarrow{p} c \wedge f \xrightarrow{p} c). \end{aligned} \quad \square$$

Proof of Lemma 7.7.

(\Leftarrow) We assume $b \sqsubseteq c$. Then, by Lemma 3.2(b), $b \xrightarrow{a} c$ for all a . By definition this is the same as $b \xrightarrow{*} c$.

(\Rightarrow) We use family induction on b .

Induction base, i.e., b a product: Spelling out the definition yields $b \xrightarrow{*} c \iff (\forall a : b \xrightarrow{a} c)$. Choosing $a = b$ implies $b \xrightarrow{b} c$ which is equivalent to $b \sqsubseteq b \implies b \sqsubseteq c$, since b is a product. This immediately yields the claim.

Induction step, i.e., $b = e + f$. We again set $a = b$ and reason as follows, using the definition of $\xrightarrow{e+f}$, Lemma 7.3(g), predicate logic, the induction hypothesis and Lemma 3.18(b),

$$\begin{aligned}
& e + f \xrightarrow{e+f} c \\
\iff & e + f \xrightarrow{e} c \wedge e + f \xrightarrow{f} c \\
\iff & e \xrightarrow{e} c \wedge f \xrightarrow{e} c \wedge e \xrightarrow{f} c \wedge f \xrightarrow{f} c \\
\iff & e \xrightarrow{e} c \wedge f \xrightarrow{f} c \\
\iff & e \sqsubseteq c \wedge f \sqsubseteq c \\
\iff & e + f \sqsubseteq c. \quad \square
\end{aligned}$$

B Haskell: Code Fragments and Complete Specification of the Robot Family

In this section we present some code fragments of our prototype implementation. We show the straightforward set-based implementation and not the more sophisticated one described on Page 10.

The main type of our implementation is `AlgFamily`, a pair consisting of a tree representation of an algebraic term and the product family it denotes.

```

type AlgFamily = (Term, ProdFamily)

type ProdFamily = [Product]           -- sum of products
type Product    = [BaseFeature]
type BaseFeature = String

data Term = Zero | One | Basic BaseFeature | Sum Term Term
          | Product Term Term | Excl Term Term

```

Using these types it is easy to implement the functions mentioned. For example `sum`, `multiplication` and `extracting common parts` can be encoded as follows:

```

(.,.), (.*.) :: AlgFamily -> AlgFamily -> AlgFamily
(x,xn) .+. (y,yn) = (Sum x y, sunion xn yn)
(x,xn) .* (y,yn) =
  (Product x y, mkset [[bunion bx by] | bx <- xn, by <- yn ])

common :: ProdFamily -> Product
common [] = []
common pf = foldl1 binter pf

```

Here `sunion` denotes set union, `bunion` bag union and `binter` bag intersection.

To encode a concrete example one can follow the code below where we give the whole specification for the robot family example of Section 5.

```

-----
----- 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_frwr = bf "Able to move forward"
move_bckwr = 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 with basic control (only on or off)"
digital_ctrl = bf "Robot 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 sensors"

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

v_s_colour_vision = bf "Sensor capable of determining the colour
                      of objects in the robot's environment"
black_white_vision = bf "Black and white environmental vision"
primary_colour_vision = bf "Primary colour environmental vision"

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_sensor =      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_sensor]

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

environmental_vision =      enhanced_obstacle_detection
                            .*. v_s_colour_vision
                            .*. opt[black_white_vision]
                            .*. opt[primary_colour_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
               .+. small_pltfrm .*. medium_pltfrm
               .+. medium_pltfrm .*. large_pltfrm
               .+. small_pltfrm .*. c_sensor .^. 4
               .+. medium_pltfrm .*. c_sensor .^. 5
               .+. large_pltfrm .*. c_sensor .^. 6

```

C Prover9 Script

In this section we show a typical input file for the theorem prover Prover9 encoding product family algebra. An automatic proof attempt can be started by `prover9 -f <file>` or via the graphical interface of Prover9; if that does not succeed (within reasonable time), the companion program Mace4 can try to find a counterexample to the conjectured theorem.

```

op(500, infix, "+").
op(490, infix, ";").
op(700, infix, "<=").
op(700, infix, "<<").

```

```

% axioms of product family algebra %%%%%%%%%%
formulas(sos).
% commutative additive monoid
x + y = y + x.
x + 0 = x.
x+(y+z) = (x+y)+z.
% commutative multiplicative monoid
x;y = y;x.
x;1 = x & 1;x = x.
x;(y;z) = (x;y);z.
% annihilation laws
0;x = 0 & x;0 = 0.
% idempotence
x + x = x.
% distributivity
x;(y + z) = x;y + x;z.
(x + y);z = x;z + y;z.

% natural order
x<=y <-> x+y=y.
% refinement relation
x<<y <-> (exists z (x<=y;z)).
end_of_list.

% conjecture/theorem/lemma... %%%%%%%%%%
formulas(goals).
% theorem to be proved should be added here;
% for example reflexivity of <<
x <= y -> x << y.

end_of_list.

```