Universität Augsburg

Algebraic View Reconciliation

P. Höfner **R. Khedri** **B. Möller**

Report 2007-13 December 2007

Institut für Informatik
D-86135 Augsburg

Algebraic View Reconciliation

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

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

Abstract. Embedded systems such as automotive systems are very complex to specify. Since it is difficult to capture all their requirements or their design in one single model, approaches working with several system views are adopted. The main problem there is to keep these views coherent; the issue is known as *view reconciliation*. This paper proposes an algebraic solution. It uses sets of integration constraints that link (families of) system features in one view to other (families of) features in the same or a different view. Both families and constraints are formalized using a feature algebra. Besides presenting a constraint relation and its mathematical properties, the paper shows in several examples the suitability of this approach for a wide class of integration constraint formulations.

1 Introduction

The adoption of the product family paradigm in software development aims at recognising a reality in software development industry pointed to by Parnas [15] decades ago. The research work about software product families aims at studying the commonalities/variability occurring among the products in order to have a better management of and processes for software production. Also, the family approach to software development proposes that, instead of focusing our attention on a single software system to be built, we take into account predictable changes. Thereby, the analysis and design of a family of software systems that share a core part (commonalities among all the members) is considered. Software product line engineering, which is a family-oriented software production process and technique, seems to be adopted by both practitioners and researchers to deal with changes in the requirements and thereby a reconsideration of the corresponding designs. The idea behind product line engineering is to take advantage of the commonality of systems that are developed for a specific domain. Weiss and Lai [16, Preface, page 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%.

Embedded systems such as automotive systems are very difficult to specify using one single model that takes into consideration the software and the hardware of the system. However, in engineering and in such a situation, it is common to adopt a multi-view approach. For instance, at the construction of a building, the specifiers elaborate many views of the building: structure view, plumbing view, electrical wiring view, etc. These views need to be coherent. When we take this view-approach to software product families, the complexity of the problem increases: each member of the family of each view needs to be coherent with some members of each of the other views.

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

There is a wide literature on the reconciliation of non-functional requirements such as security and performance [3]. For instance security requires careful scrutinizing of data, which could affect system's performance. Also, we find approaches to resolve architectural mismatches resulting from integrating commercial off-the-shelf (COTS) components. The mismatches are essentially between the services required and provided that might arise in the interaction of a component and its environment. However these approaches do not directly relate to the problem we are tackling in this 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 [2]. Also, a similar problem occurs when merging views of a database and it is called *view reconciliation problem* [11]. The above cases are considering the development of a single software system and not a software family. The mismatches that we are concerned with are 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 [4, 17]. However, they deal with the integration of partial descriptions such as scenarios, use-cases, and viewpoints of requirements for a single system. In this paper, we introduce a technique for an overall integration of descriptions of a product family from several views/perspectives. The proposed formalism allows the integration of descriptions of a family from views that can be either orthogonal (e.g. software and hardware) or may overlap. Some views might impose constraints on others. We aim at integrating product family descriptions to obtain a specification of the considered family that does not include members violating the constraints of integration which relate features from one view with others from the same or a different view. To perform such an integration, mainly two problems need to be resolved. The first is how to articulate the constraints of integration, and the second is to perform the integration of partial family descriptions taking into account these constraints and leading to a coherent specification of the considered family. A review of the literature of product family based software development reveals a wide set of notions and terms used without formal definitions. However, in [6, 7] a clear and simple mathematical setting for the usage of this paradigm is proposed. In the present paper, we extend that approach to cover the view reconciliation problem.

In Section 2, we define feature algebra, products and features and present a refinement relation. In Section 3, we introduce a requirement relation and elaborate on its properties and its use to formally capture informal integration constraints. In Section 4, we sketch the multi-view reconciliation problem. In Section 5, we present a larger case study. We conclude and point to future research in Section 6.

2 Feature Algebra

In this section, we introduce the algebraic structure of feature algebra. Since it is based on semirings we will first present these. Afterwards, we will give an idea on some notions defined within this mathematical structure.

Definition 2.1 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 $\leq$.

In our current context, $+$ can be interpreted as a choice between options of products and features and $\cdot$ as their composition or mandatory presence. An important example of an idempotent (but not commutative) semiring is REL, the algebra of binary relations over a set under relational composition. More details about (idempotent) semirings and examples of their relevance to computer science can be found,e.g., in [5].

In the literature, terms like product family and subfamily lack exact definitions. Following [7, 6] we use the following algebraic definitions for these terms.

Definition 2.2 A *feature algebra* is an idempotent and commutative semiring. Its elements are termed *product families*.

These elements can be considered as abstractly representing sets of products, each of which is composed of a number of features. In the remainder let $F = (S, +, 0, \cdot, 1)$ be a feature algebra.

Definition 2.3 An element a is said to be a *product* if

$$\forall\, b : b \leq a \implies b = 0 \vee b = a \;\wedge\; \forall\, b, c : a \leq b + c \implies (a \leq b \vee a \leq c)\;. \quad (1)$$

Note that 0 is a product. A product a is *proper* if $a \neq 0$.

Intuitively, this means that a product cannot be divided using the choice operator $+$. Or in other terms, it does not offer optional or alternative features. With this definition we deviate slightly from the one in [7, 6] to avoid tedious case analyses.

Analogously to Definition 2.3, indecomposability can be required, but this time w.r.t. multiplication rather than addition.

Definition 2.4 An element a is called *feature* if it is a proper product and

$$\forall\, b : b \,|\, a \implies b = 1 \vee b = a \ \wedge\ \forall\, b, c : a \,|\, (b \cdot c) \implies (a \,|\, b \vee a \,|\, c)\ , \qquad (2)$$

where the divisibility relation $|$ is given by $x \,|\, y \iff_{df} \exists\, z : y = x \cdot z$. The algebra is *feature-generated* if every element is a finite sum of finite products of features. In this case, the *size* of element a is the minimum number n such that $a = \sum\limits_{i<n} p_i$ for suitable products p_i.

From the mathematical point of view, the characteristics of products (1) and features (2) are similar and well known. A uniform treatment of both notions is given in the Appendix of [6], where also the order-theoretic background is discussed. Other notions and similarity measures among families, like *generated products*, *refinement* of families, *k-near similarity* of two families, *weak zero*, and *subfamily*, are also defined and discussed. An important tool is the

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 $P(b) \wedge P(c) \implies P(b + c)$ (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.

A particular feature-generated algebra over a set of basic features can be constructed in the following way. Take as products finite bags (or multisets) of basic features and as elements finite sets of such bags. Use set union for $+$ and (essentially) bag union for $\cdot$. This yields a feature algebra in which $\cdot$ is not idempotent, since bags record the multiplicities of features. We will refer to this algebra as the *bag model*. If one does not want to distinguish multiple occurrences of features, one can use sets rather than bags of basic features; this yields an algebra with idempotent $\cdot$ to which we will refer as the *set model*. In both models the size of an element is its cardinality.

Example 2.5 We assume a small company which has a family of two product lines: *DVD Players* and *MP3 Players*. All its members share a list of common features (audio equaliser (`a_eq`) and dolby surround (`dbs`)). Members can also 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 able to *play mp3-files* (`p_mp3`) while another does not have this feature. However, all the DVD players must have the *play DVD* (`p_dvd`) feature. Similarily, some but not all MP3 players are able to `record mp3-files` (`r_mp3`).

Therefore we can characterise the product lines as

$$\begin{aligned} \texttt{dvd\_player} &= \texttt{p\_dvd} \cdot (1 + \texttt{p\_mp3}) \cdot \texttt{a\_eq} \cdot \texttt{dbs} \ , \\ \texttt{mp3\_player} &= \texttt{p\_mp3} \cdot (1 + \texttt{r\_mp3}) \cdot \texttt{a\_eq} \cdot \texttt{dbs} \ . \end{aligned}$$

The whole product family is the combination of both players via choice:

$$(\texttt{p\_dvd} \cdot (1 + \texttt{p\_mp3}) + \texttt{p\_mp3} \cdot (1 + \texttt{r\_mp3})) \cdot \texttt{a\_eq} \cdot \texttt{dbs} \ . \qquad \square$$

Now we return to general feature algebras. The refinement relation is defined as $a \sqsubseteq b \iff_{df} \exists c : a \leq b \cdot c$ and forms a preorder, i.e., it is reflexive and transitive. 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. It is easy to see that divisibility implies refinement:

$$a \,|\, b \implies b \sqsubseteq a \ . \tag{3}$$

The reverse implication need not hold for the following reason: $b \sqsubseteq a$ allows that some variants of b may be discarded, whereas $a \,|\, b$ means that *all* variants of b can be extented to have an a. However, since products are defined to be elements with only one variant, refinement and divisibility coincide in particular feature algebras if the refinee is a product:

Lemma 2.6 *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 $b \sqsubseteq p$ as "b has p (as a subproduct)". The representations of the elements in sum-of-products form corresponds to or/and trees of features.

We list a few further useful properties of the refinement relation.

Lemma 2.7 *Let a, b, c, p be elements of a feature algebra such that p is a product.*

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

The paper [7] also gives some useful applications of feature algebras concerning finding common features, building up product families, finding new products and excluding special feature combinations.

Moreover, finding the commonalities of a given set of products is a very relevant issue, since the identification of common artifacts within systems (e.g. chips, software modules, etc.) enhances hardware/software reuse. Within feature algebras like the set-based model and the bag-based model, this problem can be formalised as finding "the greatest common divisor" or to factor out the features common to all given products. It is a direct use of "classical" algorithms which shows an advantage of using an algebraic approach. Solving gcd (greatest common divisor) is well known and easy, whereas finding commonalities using diagrams (e.g., FODA [12]) or trees (e.g., FORM [13]) is more complex. Other properties such as

"If a product has feature f_1 it also must have feature f_2"

can be modelled using feature algebra and are discussed in the next section. To check the adequacy of the proposed definitions a prototype implementation of the bag model[3] has been written 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.

3 Requirements: Implications and Exclusions

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

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

Such integration constraints can easily be formulated in feature algebra. To achieve this goal we introduce the following *requirement relation*.

Definition 3.1 Assume a feature-generated algebra. For elements a, b and product p we define, in a family-induction style,

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

Now $a \xrightarrow{e} b$ is well defined for all e, since by assumption e can be written as a finite sum of products. Informally $a \xrightarrow{e} b$ means that if e has a then it also has b. From a mathematical point of view, 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 $\mathtt{wheel}^4 \cdot \mathtt{axis}$ and $\mathtt{wheel}^3 \cdot \mathtt{axis}^2$ is $\mathtt{wheel}^4 \cdot \mathtt{axis}^2$. The requirement that in a product line a two wheels need an axis is expressed by $\mathtt{wheel}^2 \xrightarrow{a} \mathtt{axis}$. Later on we present more examples of the requirement relation.

Lemma 3.2 *Let a, b, c, d be elements of a feature-generated algebra.*

[3] The program and a short description can be found at: http://www.informatik.uni-augsburg.de/∼hoefnepe/featurealgebra .

(a) $\xrightarrow{a}$ *is a preorder.*
(b) *Let* $b \sqsubseteq c$*, then* $c \xrightarrow{a} d \implies b \xrightarrow{a} d$ *and* $(d \xrightarrow{a} b) \implies (d \xrightarrow{a} c)$.
In particular, $b \sqsubseteq c \implies b \xrightarrow{a} c$.
(c) *Let* $b \leq c$*, then* $c \xrightarrow{a} d \implies b \xrightarrow{a} d$ *and* $d \xrightarrow{a} b \implies d \xrightarrow{a} c$.
In particular, $p \leq q \implies p \xrightarrow{a} q$.

Since variants of semirings are already successfully combined with automated theorem provers [9, 10], we implemented feature algebra axiomatically in the first-order theorem prover Prover9 and the counterexample generator Mace4 [14]. Using this encoding we can prove all the theorems and lemmas presented 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 [1]. Proofs by hand can be found in Appendix A.

Lemma 3.3 *Let* a, b, c, d, p *be elements of a feature-generated algebra.*

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

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

Example 3.4 In the sequel we assume that a vehicle is built up from the following components (features): `wheel`, `axis`, `steering−wheel`, `speed_indicator`, `engine`, `standard_transmission` and `automatic_transmission`.

- $\texttt{engine} \xrightarrow{\texttt{car}} \texttt{speed\_indicator}$ guarantees that every motor vehicle of the family a has also a speed indicator.
- $\texttt{wheel} \xrightarrow{\texttt{car}} \texttt{steering}-\texttt{wheel}$ and $\texttt{engine} \xrightarrow{\texttt{car}} \texttt{steering}-\texttt{wheel}$ means that there is at least one steering-wheel if the vehicle has at least one wheel or one engine.
- To exclude more than one steering-wheel, one can use the requirement $(\texttt{steering}-\texttt{wheel}) \cdot (\texttt{steering}-\texttt{wheel}) \xrightarrow{\texttt{car}} 0$.
- $(\texttt{wheel} \cdot \texttt{wheel})^n \xrightarrow{\texttt{car}} \texttt{axis}^n$ (for all $n \in \mathbb{N}$) guarantees that each pair of wheels can be connected by a single axis.
- To express that a car has to have an even number of wheels we can use $\texttt{wheel}^{2n+1} \xrightarrow{\texttt{car}} \texttt{wheel}^{2n+2}$.

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 a product family has feature p_1 it also needs to have feature p_2 or feature p_3".

For this we may simply write $p_1 \xrightarrow{\texttt{car}} (p_2 + p_3)$.

– Thus, by $\texttt{engine} \xrightarrow{\texttt{car}} \texttt{standard\_transmission} + \texttt{automatic\_transmission}$ we require that if a car has an engine it also needs to have a standard transmission or an automatic one.

An application of such an integration contraint occurs when sensors are used, 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.

– For example $1 \xrightarrow{\texttt{car}} \texttt{engine}$ enforces that each car has (at least) one engine.

□

The third item above 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 [7].

Finally, to express global product implication, one might define

$$b \xrightarrow{*} c \iff_{df} \forall\, a : b \xrightarrow{a} c\ . \tag{4}$$

However, this relation is uninteresting by the following result.

Lemma 3.5 *Let b, c be elements of a feature-generated algebra. Then $b \xrightarrow{*} c \iff b \sqsubseteq c$. In particular, $b \xrightarrow{*} 0 \iff b \sqsubseteq 0 \iff b = 0$.*

Since the proof only uses reflexivity and transitivity of $\sqsubseteq$, it generalises to arbitrary preorders. In fact, we have the following result.

Lemma 3.6 *For an arbitrary binary relation Q define the relation R_Q by*

$$x\, R_Q\, y \iff_{df} (\forall\, x : xQy \implies xQz)\ .$$

(a) R_Q is a preorder.

(b) A relation $\preceq$ is a preorder iff it satisfies the principle of indirect inequality*, i.e., coincides with $R_{\preceq}$.*

4 Multi-View Reconciliation Problem

In this section we sketch the multi-view reconciliation problem. Later on, we illustrate the problem with a small example. In Section 5 we will present a larger case study.

When we approach the specification of a product family from different perspectives, we can easily show that these perspectives are somehow interdependent. When this interdependence is known, how can we integrate them taking into account their interdependence, which can be captured by 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 lines 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.

A a simple example we assume a company which produces computers. In particular, it builds machines with a harddisk 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 feature algebra[4]:

```
hw  =  hd .*. scr .*. opt[scr , prn , scn]
```

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

```
sw  =  hd_drv .*. scr_drv .*. prn_drv
   .+. hd_drv .*. scr_drv .*. scn_drv
```

The first one contains drivers for harddisks, screens and printers; the second for harddisks, screens and scanners. The Multi-View Reconciliation Problem asks for all products that satisfy the following requirements:

$$
\begin{aligned}
\mathtt{hd} &\xrightarrow{\mathtt{hw.*.sw}} \mathtt{hd\_drv}\\
\mathtt{scr} &\xrightarrow{\mathtt{hw.*.sw}} \mathtt{scr\_drv}\\
\mathtt{prn} &\xrightarrow{\mathtt{hw.*.sw}} \mathtt{prn\_drv}\\
\mathtt{scn} &\xrightarrow{\mathtt{hw.*.sw}} \mathtt{scn\_drv}
\end{aligned}
$$

[4] We are using the Haskell notation of our implementation, i.e., `.*.` and `.+.` denote multiplication and addition, resp.

These requirements guarantee that each hardware component has an appropriate driver. For this we use the function `reconc` that 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 procedure described above. Therefore the call

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

determines all desired products, 8 in number:

```
--------------------------------------------------------------------
    harddisk,       harddisk driver
    printer,        printer driver
    screen (2x),    screen driver

--------------------------------------------------------------------
    harddisk,       harddisk driver
    printer,        printer driver
    screen,         screen driver

--------------------------------------------------------------------
    harddisk,       harddisk driver
                    printer driver
    screen (2x),    screen driver

--------------------------------------------------------------------
    harddisk,       harddisk driver
                    printer driver
    screen,         screen driver

--------------------------------------------------------------------
    harddisk,       harddisk driver
    scanner,        scanner driver
    screen (2x),    screen driver

--------------------------------------------------------------------
    harddisk,       harddisk driver
    scanner,        scanner driver
    screen,         screen driver

--------------------------------------------------------------------
    harddisk,       harddisk driver
                    scanner driver
    screen (2x),    screen driver

--------------------------------------------------------------------
    harddisk,       harddisk driver
                    scanner driver
    screen,         screen 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 harddisk and screen(s) only. The versions offer software for scanners and printers , respectively. 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.

Finally, it should be mentioned that, symmetrically to the combination of product lines, one can extract views of product families by simple projection on the respective feature sets.

5 Illustrative Case Study of the Multi-View Reconciliation Problem

Due to lack of space we only point out the interesting bits of the case study. The whole specification and the corresponding Haskell code can be found in [8].

We consider a family of *Driver Assisting Systems* described from a functional perspective and from a sensor perspective. The latter perspective includes only the sensors needed by the products.

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

"Road sign recognition and indication" (`rd_sgn_rcg`),
"Far Infra-Red detection" (`fir`),
"Thermal imaging detection" (`tid`),
"Line departure warning" (`ldw`),
"Blind spot monitoring" (`bsm`),
"Adaptive Cruise Control following control" (`acc_f_c`),
"Emergency braking" (`e_braking`),
"Urban Adaptive Cruise Control (stop & go)" (`u_acc`),
"Automatic steering and braking" (`aut_str_brk`),
"Automatic line keeping" (`aut_lane`),
"Obstacle avoidance" (`obst_avd`) and
"Obstacle warning" (`obst_wrng`).

More advanced products are combinations of other components. For example, the functional description of night vision consists of far infra-red technology (`fir`) or a simple thermal infra-red imaging technology (`tid`). The component for *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.

```
n_vision      =  fir .+. tid .+. (fir .*. tid)
driver_i_w    =  rd_sgn_rcg .*. ldw .*. n_vision .*. bsm
```

To describe the complete product line for the driver assisting system from the functional perspective we use further components for *automatic longitude control* (`aut_long_ctrl`) and *automatic lateral control* (`aut_ltrl_ctrl`). The details are described in [8]. The whole product line is then characterized by

```
p_line_driver_assist_sys  =  obst_wrng .*. opt[obst_avd, driver_i_w]
                             .*. opt[aut_long_ctrl, aut_ltrl_ctrl]
```

where opt[...] describes again optional features. It is easy to see that this product family contains products with both far and thermal infra-red technology. From an industrial point of view one of the both technologies is just redundant if both occur and yields to extra costs. Therefore we use the requirement

$$\mathtt{fir}.*.\mathtt{tid} \xrightarrow{\mathtt{p\_line\_driver\_assist\_sys}} 0 \ .$$

The restricted result res_p_line_driver_assist_sys yields a size reduction of 25%. In real life this would lead to an immense decrease of costs. 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 sequel we focus on another view. Instead of discussing functional descriptions of our system we now focus on sensors and actuators. In particular, this view describes the kind of sensors needed by the family to gather the information necessary for the above functional features.

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

"Acceleration pulsator" (acclrt_pulsator),
"Acceleration of the wheel sensor" (acclrt_wheel),
"Acceleration of the body of the vehicle sensor" (acclrt_body),
"Displacement of the wheel sensor" (dis_wheel),
"Displacement of the body of the vehicle sensor" (dis_body),
"Brake temperature sensor" (brk_temp),
"CO2 sensor" (co_snsr),
"Position sensor" (position),
"Load data sensor" (load),
"Adaptive Cruise Control radar" (acc_radar),
"Adaptive Cruise Control laser" (acc_laser),
"Adaptive Cruise Control video camera" (acc_v_cam),
"Adaptive Cruise Control Infra-red camera" (acc_ir_cam) and
"Adaptive Cruise Control Far-Infra-red camera" (acc_far_ir).

To describe the complete on-board sensor configuration we use the following combined features (for details see again [8]):

"Acceleration Sensors" (acclrt_sensors),
"Displacement Sensors" (dis_sensors) and
"Adaptive Cruise Control Sensors" (acc_sensors).

The complete description of all on-board sensors then becomes

```
on_board  =  opt[co_snsr] .*. opt[brk_temp .^. 4] .*. acclrt_sensors
          .*. dis_sensors  .*. acc_sensors .*. opt[position .^. 8]
```

where `a.^.n = a.*. ... .*.a` denotes the standard power function. Similar to the requirement constaint of the functional description view, we have the following exclusion constraints:

$$\texttt{acc\_radar} .*. \texttt{acc\_laser} \xrightarrow{\texttt{on\_board}} 0 \, ,$$
$$\texttt{acc\_far\_ir} .*. \texttt{acc\_ir\_cam} \xrightarrow{\texttt{on\_board}} 0 \, .$$

The restricted set of possible sensor configurations `res_on_board` is about 76% smaller than the unrestricted version. Therefore adding simple restriction constraints can yield an immense and useful decrease of the variety of products.

The functional and the sensor view now form the basis for the multi-view reconciliation problem. To link these two perspectives we set up the following requirements:

$$\begin{aligned} \texttt{driver\_i\_w} &\xrightarrow{x} \texttt{acclrt\_pulsator} .+. \texttt{co\_snsr} .+. \texttt{position} \, , \\ \texttt{e\_braking} &\xrightarrow{x} \texttt{brk\_temp} .*. \texttt{position} \, , \\ \texttt{aut\_str\_brk} .+. \texttt{aut\_lane} &\xrightarrow{x} \texttt{dis\_wheel} .*. \texttt{acclrt\_body} .*. \texttt{load} \, , \end{aligned}$$

where $x = \texttt{res\_p\_line\_driver\_assist\_sys} .*. \texttt{res\_on\_board}$. Due to lack of space we cannot explain these requirements in detail; we only sketch the idea of the second one. In the case of an emergency break, the sensors have to control the temperature of the break and at the same time the current position has to be checked to react if there is an obstacle in front.

Now we can use the described algorithm to solve the multi-view reconciliation problem. This yields a general product family of 30240 different models.

6 Conclusion and Future Work

We have presented an algebraic framework for solving the multiview reconciliation problem. The main ingredient is a set of integration constraints that link features or more generally subfamilies 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 feature 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 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 aging 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 involve 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 [14] and prototypically implemented over some useful models of feature algebra in Haskell.

The algebraic model of features is at very 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 future 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 joint the ongoing research efforts for formal model driven software development techniques. The feature algebra model of a family and the specifications of the family's features would be the initial models of the sought transformation.

References

1. `http://www.informatik.uni-augsburg.de/~hoefnepe/kleene_db`.
2. P. Avgeriou and N. Guelfi. Resolving architectural mismatches of COTS through architectural reconciliation. In *LNCS*, volume 3412, pages 248–257. Springer.
3. 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.
4. J. Desharnais, M. Frappier, and R. Khedri. Integration of sequential scenarios. *IEEE Transactions on Software Engineering*, 24(9):695–708, 1998.
5. J. Desharnais, B. Möller, and G. Struth. Modal Kleene algebra and applications – A survey. *Journal on Relational Methods in Computer Science*, 1:93–131, 2004. http://www.cosc.brocku.ca/Faculty/Winter/JoRMiCS/.
6. P. Höfner, R. Khedri, and B. Möller. Feature algebra. Technical Report Report 2006-04, Institut für Informatik, Universität Augsburg, 2006.
7. P. Höfner, R. Khedri, and B. Möller. Feature algebra. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *LNCS*, pages 300–315. Springer, 2006.
8. P. Höfner, R. Khedri, and B. Möller. Algebraic view reconciliation. Technical Report Report 2007-XX, Institut für Informatik, Universität Augsburg, 2007. (to appear).

9. P. Höfner and G. Struth. Automated reasoning in Kleene algebra. In F. Pfennig, editor, *CADE 2007*, volume 4603 of *LNAI*, pages 279–294. Springer, 2007.
10. P. Höfner and G. Struth. Can refinement be automated? In E. Boiten, J. Derrick, and G. Smith, editors, *Refine 2007*, ENTCS. Elsevier, 2007. (to appear).
11. B. E. Jacobs. *Applied Database Logic: Fundamental Issues*, volume I. Prentice-Hall, 1985.
12. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon Software Engineering Institute, Carnegie Mellon University, 1990.
13. K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.
14. W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9`.
15. D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE2(1):1–9, 1976.
16. D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison Wesley Longman, 1999.
17. M. Wirsing and K. Alexander. View consistency in software development. In *LNCS*, volume 2941, pages 341–357. Springer, 2004.

A Omitted Proofs

Proof of Lemma 2.6. Assume $a \le c{\cdot}p$ for some element $c = \sum_{i\in I} q_i$ with finite index set I and products q_i. By distributivity, $a \le \sum_{i\in I} q_i \cdot p$. This means $a = \sum_{i\in J} q_i \cdot p$ for some subset $J \subseteq I$. Again, by distributivity, $a = (\sum_{i\in J} q_i) \cdot p$, showing $p|a$. □

Proof of Lemma 2.7.
(a) $a \le a + b = (a + b) \cdot 1$.
(b) Choose $c = b$ in the definition of refinement.
(c) See Lemma 4.6 of [7].
(d) ($\Leftarrow$) follows by (a) and transitivity of $\sqsubseteq$.
($\Leftarrow$) Assume $p \le (a + b) \cdot c = a \cdot c + b\cdot$. Since p is a product, we have $p \le a \cdot c \vee p \le b \cdot c$, which shows the claim. □

Proof of Lemma 3.2. 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 $b \le c \implies b \sqsubseteq c$. □

Proof of Lemma 3.3. Again the claims are shown by family induction for which we only do the base cases.

(a) By Lemma 2.7(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 3.2(b) by $c \sqsubseteq b + c$.

(d) Immediate from Lemma 3.2(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 2.7(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 2.7(d), predicate logic and definition of $\xrightarrow{p}$ again,

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

$\square$

Proof of Lemma 3.5. ($\Leftarrow$) 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 3.3(g), predicate logic, the induction hypothesis and Lemma 2.7(d),

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

$\square$

Proof of Lemma 3.6.

(a) Reflexivity: By definition, $y\, R_Q\, y \iff (\forall\, x : x\, Q\, y \implies x\, Q\, y)$ which is true by predicate logic.

Transitivity: Assume $y\, R_Q\, z \wedge z\, R_Q\, u$ and, for arbitrary x, that $x\, Q\, y$. Then, by the first assumption, also $x\, Q\, z$ and hence, by the second assumption, also $x\, Q\, u$, as required.

(b) ($\Rightarrow$) is shown as in the proof of Lemma 3.5.

($\Leftarrow$) is immediate from (a). $\square$