Formal Product Families for Abstract Machines

Andreas Zelend
Institut für Informatik, Universität Augsburg, Germany
zelend@informatik.uni-augsburg.de

Abstract. We present a model of product family algebra using the semantics of abstract machines as defined in the B-Method. In this model we can compose families of machines to build larger ones in an incremental way.

1 Introduction

Since subsystems are more manageable and easier to maintain than large software systems as a whole, Parnas [6] adopted the concept of product families, stemming from hardware industry, for software development. Products of such families are build up from features. This kind of software development is called feature oriented software development (FOSD). Since the terms features, products and product families lacked formal definitions, Höfner et al. [3] developed a product family algebra that gives mathematical precise definitions of the above terms. Furthermore it provides mechanisms to solve important tasks of FOSD such as finding common features of several product. Another concept based on solid mathematical foundations is the B-Method (B). It is a formal method of software development which was introduced by Abrial [1]. The basic concept of B is the formal specification of systems with abstract machines. The goal of this work is to develop a product family algebra for abstract machines.

2 Product Family Algebra

As a motivation we give an example that informally introduces features, products and product families.

Assume a consumer electronics company which assembles several different media players. The company offers two MP3 players with different features. One that can only play MP3 files (p_{mp3}) and one that can additionally record MP3 files (r_{mp3}). Both players have a LCD display (lcd) and a USB connector (usb). These products are summarized in a MP3 Player product family (mp3_player).

The expressions in parentheses are the (basic) features from which the products are built. We want to express products and product families algebraically. The MP3 Player product family can be written as

\[ mp3\_\text{player} = p_{mp3} \cdot \text{lcd} \cdot \text{usb} + p_{mp3} \cdot r_{mp3} \cdot \text{lcd} \cdot \text{usb}. \]

The operation + is interpreted as choice between two products and \cdot as (mandatory) presence of a feature within a product. This kind of structure is offered by semirings.
2.1 Semirings

Semirings are a well-known algebraic structure that offers these kind of operations. They are the base of the Product Family Algebra introduced in [3].

An i-semiring $S$ is a semiring $(S, +, 0, \cdot, 1)$ such that $+$ is idempotent. In an i-semiring the relation $a \leq b \iff a + b = b$ is a partial order, called the natural order on $S$. It has 0 as its least element. Moreover, $+$ and $\cdot$ are isotone with respect to $\leq$. The i-semiring is commutative iff $\cdot$ is commutative.

An element $p \in S$ is called a product if $p = 1$ or if it satisfies

$$\forall a \in S : a \leq p \implies (a = 0 \lor a = p),$$

$$\forall a, b \in S : p \leq a + b \implies (p \leq a \lor p \leq b).$$

A product $p$ is proper if $p \neq 0$.

An element $f \in S$ is called a feature if it is a proper product different from 1 satisfying the following laws:

$$\forall a \in S : a \mid f \implies a = 1 \lor a = f,$$

$$\forall a, b \in S : f \mid (a \cdot b) \implies (f \mid a \lor f \mid b),$$

where the divisibility relation $\mid$ is given by $x \mid y \iff \exists z : y = x \cdot z$.

Mathematically, products are irreducible w.r.t. $+$ and $\leq$ and features are irreducible w.r.t. $\cdot$ and $\mid$. A discussion of these properties is given in the appendix of [2].

2.2 Product Family Algebra

We state only the main definition of a product family algebra here. A summary of important properties is given in [7]; for a detailed discussion see [3].

A product family algebra is a commutative i-semiring in which 1 is a product. Its elements are called product families or families for short. A family $g$ is a subfamily of family $f$ iff $g \leq f$.

A product family algebra is feature-generated iff 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 set of products is closed under multiplication. In this case, single features are the “smallest” components from which products and product families are made.

3 Abstract Machines

In this section we briefly present the structure of abstract machines. The concept of abstract machines was introduced by Abrial. It is the basic mechanism of the B-Method (B) [1], intended to specify and verify software on a solid mathematical foundation. It is mainly used for the development of safety-critical systems.
The B-Method offers commercial and non-commercial tool support, e.g. the B-Toolkit\(^1\) or Atelier B\(^2\). Atelier B has been used in various industrial and academic projects, mainly security related ones, e.g. smart card development [4]. We give a brief overview of the structure of abstract machines.

In [1], Abrial states that an abstract machine can informally be seen as a calculator, that has an internal memory and buttons to manipulate this memory. Using the buttons is the only way to modify the state of the calculator. This behavior is one of the principles known as information hiding stated by Parnas [5]. Abstract machines are used to specify software systems or more precisely, modules of a software system. They only describe what modules/systems have to be developed but not how—implementations are responsible for this.

Formally an abstract machine consists of different clauses. We treat only those clauses needed later on. An overview of all possible clauses is given in Appendix B of [1]. An abstract machine can, roughly spoken, be described by four parts: *machine header*, *static data*, *state* and *dynamics*. Each part consists of one or more clauses.

- The *machine header* consists of a list of **parameters** and a list of **INCLUSIONS**. The **parameters** are a list of scalars or finite, non-empty sets offering instantiation.

- The *static data* consist of the clauses **CONSTRAINTS**, **SETS**, **CONSTANTS**, **PROPERTIES**. The **CONSTRAINTS** predicate is used to restrict the machine parameters. The clause **SET** can be used to define own types, for example. The **PROPERTIES** clause is given as a predicate involving **CONSTANTS** and **SETS**.

- The *state* is built from the clauses **VARIABLES**, **INVARIANT** and **INITIALIZATION**. The **VARIABLES** are initialized by the **INITIALIZATION**. They must satisfy the **INVARIANT** predicate during the whole "lifetime" of the machine.

- The *dynamics* of an abstract machine are the **OPERATIONS**. They are responsible for input/output tasks as well as for modifying the **VARIABLES**.

The clauses dealing with predicates are constituted by the use of the *Predicate Calculus* and *Set Theory*. The **INITIALIZATION** and the **OPERATIONS** are declared by the use of *generalized substitutions*. All clauses are optional, except the **MACHINE** clause. A detailed description and theoretical discussion is given in [7, 1].

4 A Semiring for Abstract Machines

The basis for our product family algebra of machines is the power set over a direct product of monoids.

---

\(^1\) [http://www.b-core.com/btoolkit.html](http://www.b-core.com/btoolkit.html)

\(^2\) [http://www.bmethod.com](http://www.bmethod.com)
Let \( (M_i, \cdot_i, 1_i)_{i \in I} \) be a finite family of monoids, indexed by \( I = \{1, 2, \ldots, n\} \), and \( (M, \cdot, 1) = (\prod_{i \in I} M_i, \cdot, (1_1, \ldots, 1_n)) \) their direct product. The operation \( \cdot \) is a componentwise operation, lifted pointwise to \( \circ \), defined by:

\[
\circ : \mathcal{P}(M) \times \mathcal{P}(M) \to \mathcal{P}(M)
\]

\[
A \circ B = \{ a \cdot b : a \in A, b \in B \}.
\]

By the HSP-theorem \((M, \cdot, 1)\) is again a monoid. The operation \( \circ \) is associative; it is commutative if \( \cdot \) is. Multiplication \( \cdot \) in turn is commutative if every monoid is.

**Lemma 4.1** Let \( (M_i, \cdot_i, 1_i)_{i \in I} \) be a finite family of monoids with irreducible identities, indexed by \( I = \{1, 2, \ldots, n\} \). Then \((\mathcal{P}(M), \cup, \emptyset, 1)\) is an i-semiring with \(
1 = \{1, 1, \ldots, 1\}\), called Cartesian i-semiring. The Cartesian i-semiring is commutative if every monoid is.

In Cartesian i-semiring the structure of products is easy to see; products are singleton sets. Features are exactly those products containing a specific element. The components of the tuple are all equal to the identity element from the corresponding monoid expect one, which is a pre-feature. A pre-feature of a monoid is defined similarly to a feature of a semiring, i.e., it is irreducible in particular. To describe features in a compact form, we define the substitution of \( x \) for the \( i \)-th component of \( a \) by \( a[i \mapsto x] = (a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_n) \).

**Theorem 4.2** Assume an element \( A \) of a commutative Cartesian i-semiring \( S \). \( A \) is a product iff it is a singleton set. \( A \) is a feature iff \( A = \{1[i \mapsto a_i]\} \) and \( a_i \) is a prefeature in the corresponding monoid.

## 5 An Algebraic Model for Abstract Machines

We present an algebraic model for abstract machines based on the semiring discussed before. We describe the parts of an abstract machine (clauses) algebraically and combine them afterwards. For each of the nine clauses we define an appropriate monoid:

- \((S_d, \cup, \emptyset)\) for parameter, CONSTANTS, VARIABLES
- \((P, \wedge, \text{true})\) for CONSTRAINTS, PROPERTIES, INVARIANT
- \((S_i \cup, \emptyset), i \in \{S, I, O\}\) for SETS, INITIALIZATION, OPERATIONS

Every set consists of the allowed expressions for the corresponding machine clause as defined in [1]. For example, \( S_d \) is the power set of all possible identifiers. \((P, \wedge, \text{true})\) is a monoid over the set \( P \) of all possible predicates of the Abstract Machine Notation.

**Definition 5.1** The monoid \((\mathcal{AM}, \cdot, 1)\), where \(\mathcal{AM} = S_p \times P \times S_S \times S_C \times P \times S_V \times P \times S_I \times S_O \) and \( 1 = (\emptyset, \text{true}, \emptyset, \emptyset, \text{true}, \emptyset, \text{true}, \emptyset, \emptyset) \) is called abstract machine monoid. The componentwise multiplication \( \cdot \) is called abstract machine composition.
In "full" B, there are eighteen clauses a machine can consist of. The model can easily be extended by the remaining nine clauses by adding their corresponding monoids to the direct product from Definition 5.1.

Since all monoids we define for the abstract machine clauses are commutative and have irreducible identities, we can build a product family algebra from the monoid \( (\mathbb{AM},.,1) \). The structure \( (\mathcal{P}(M),\cup,\circ,\emptyset,1) \) is a commutative i-semiring by Lemma 4.1. Furthermore it is a product family algebra since \( \mathbb{I} = \{ (\emptyset,\text{true},\emptyset,\text{true},\emptyset,\emptyset,\emptyset) \} \) is a product by Theorem 4.2.

**Definition 5.2** The product family algebra \( (\mathcal{P}(\mathbb{AM}),\cup,\circ,\emptyset,1) \) is called product family algebra for abstract machines. Elements of \( \mathcal{P}(\mathbb{AM}) \) are called abstract machine families or families for short. The operation \( \circ \) is called abstract machine family composition or family composition for short. The operation \( \cup \) is called abstract machine family union or family union for short.

**6 Case Study**

In this section we investigate a simple calculator family. The family consists of two machines \texttt{Add} and \texttt{Sub}. The machine \texttt{Sub} is specified similarly to \texttt{Add}. In our algebra they are the elements given next to the abstract machine \texttt{Add}:

\[
\begin{align*}
\text{MACHINE} & \quad \texttt{Add} \\
\text{VARIABLES} & \quad \texttt{last_result, status} \\
\text{INVARIANT} & \quad \texttt{last_result : INT} \land \texttt{status : STATE} \\
\text{INITIALIZATION} & \quad \texttt{last_result := 0 \mid status := ok} \\
\text{OPERATIONS} & \quad \texttt{result <-- add(op1, op2) =} \\
& \quad \texttt{pre} \quad \texttt{op1 : INT} \land \texttt{op2 : INT} \land \texttt{(op1 + op2) : INT} \\
& \quad \texttt{then} \quad \texttt{last_result := op1 + op2 \mid result := op1 + op2} \\
& \quad \texttt{last_res <-- ans =} \\
& \quad \texttt{last_res := last_result} \\
\end{align*}
\]

\[
\begin{align*}
\text{add} = & \quad \{ (\emptyset,\text{true},\emptyset,\text{true},\emptyset,\text{true},\emptyset,\emptyset,\text{true},\emptyset,\emptyset,\emptyset) \} \\
& \quad \{ \texttt{last_result : INT} \land \texttt{status : STATE}, \\
& \quad \texttt{last_result := 0,} \\
& \quad \texttt{status := ok}, \\
& \quad \{ \texttt{add, ans, get_status} \} \}
\end{align*}
\]

Since these families cannot be decomposed w.r.t. family union, they are products in our algebraic context. We can offer a choice between the two families by applying family union to them. The resulting family is

\[
\text{simple_calc_fam} = \text{add} \cup \text{sub} = \{ \texttt{add}, \texttt{sub} \}.
\]

The commonalities of the two elements from this product family can be determined by distributivity. As a result, we have the rearranged family

\[
\begin{align*}
\text{simple_calcs_fam} = (\{1[9 \mapsto \{ \texttt{add} \}]\} \cup \{1[9 \mapsto \{ \texttt{sub} \}]\}) \circ \{ (\emptyset,\text{true},\emptyset,\text{true},\emptyset,\text{true},\emptyset,\emptyset,\text{true},\emptyset,\emptyset,\emptyset) \} \\
& \quad \{ \texttt{last_result : INT} \land \texttt{status : STATE}, \texttt{last_result := 0, status := ok}, \{ \texttt{ans, get_status} \} \}
\end{align*}
\]
We see that the two families have the same static data and the same state. They only differ in their OPERATIONS part. The families \(\{1[9 \mapsto \{\text{add}\}]\}\) and \(\{1[9 \mapsto \{\text{sub}\}]\}\) are features.

7 Conclusion and Outlook

Abstract machines are used to specify and verify software systems, whereas product family algebra focuses on FOSD where problems like finding common features are major tasks. By the use of a product family algebra for abstract machines, software specification with abstract machines gains the advantages of this algebra. To reach this we have mapped the structure of abstract machines to a monoid of tuples. Thus an abstract machine corresponds to an element of this monoid. Elements can be composed componentwise. To deal with families of algebraically represented abstract machines, we have defined a product family algebra for abstract machines and precisely stated the structure of products and features in this semiring.

Further research has to be done one the integration of the requirement relation [3] to guarantee that a family does not lack certain features, for example. Furthermore it is of interest to investigate how to integrate the requirements that the B-Method states for a correct abstract machine in our algebra. Moreover the treatment of operations in the algebra needs to be extended. For now, operations with the same name are considered to be identical. However it might be useful to allow a kind of update.

Acknowledgments. I thank R. Glück, P. Höfner and B. Möller for fruitful discussions and valuable comments. Research partially sponsored by DFG, Project FeatureFoundation—Algebra-Based Feature-Oriented Program Synthesis

References