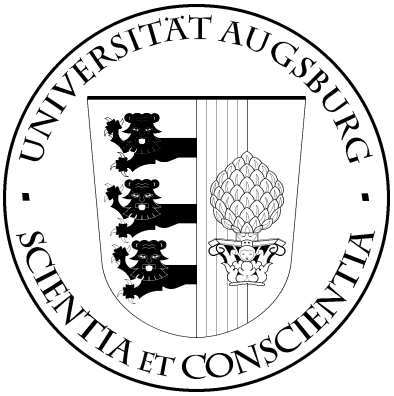
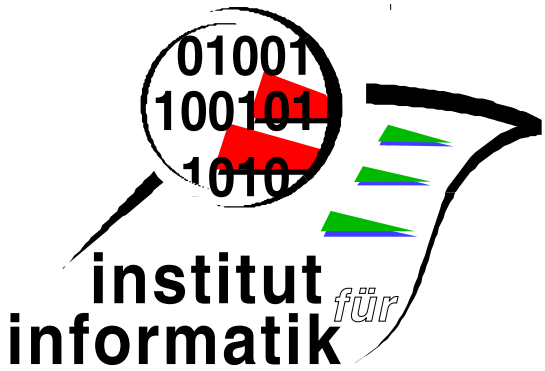
Universität Augsburg



Formal Product Families for Abstract Machines

Andreas Zelend

Report 2011-06 März 2011



Institut für Informatik
D-86135 Augsburg

Contents

1 Introduction **1**

2 Product Family Algebra **3**
2.1 Semirings 4
2.2 Product Family Algebra 7
2.3 The Set Model 7
2.4 Refinement 11
2.5 Requirements 12

3 Abstract Machines **14**
3.1 B-Method 14
3.2 Abstract Machine Notation 15
3.2.1 Machine Header 18
3.2.2 Static Data 19
3.2.3 The State of an Abstract Machine 21
3.2.4 Operations 23
3.2.5 Inclusion Clauses 31
3.3 A Basic Calculator 33

4 Product Families for Abstract Machine **44**
4.1 A Semiring for Abstract Machines 44
4.2 Products and Features 49
4.3 An Algebraic Model for Abstract Machines 53
4.4 A Basic Calculator Revisited 63

5 Extensions **68**
5.1 Requirements 68
5.2 EXTENDS and Machine Composition 70
5.3 Proper Machines 73

6 Conclusion and Outlook **76**

Appendix **77**

A Deferred Files for the Calculator from Section 3.3 **78**
A.1 Abstract Machines and their Implementations 78
A.2 Generated Source Files 83

B Deferred Files for the Calculator from Section 4.4 **92**
B.1 Abstract Machines and their Implementations 92
B.2 Generated Source Files 96

Bibliography **100**

Chapter 1

Introduction

Since the size of software systems is increasing—just think of database systems used by search engines—it is desirable to develop these systems on the basis of smaller subsystems. These subsystems are more manageable and easier to maintain than the whole system. As an approach to reach this goal, the concept of *product families*, stemming from hardware industry, was adopted for software development by Parnas [14]. Products of such families are built up from *features*. This kind of software delopment is called *feature oriented software development* (FOSD). Over the past years, quite a lot of informal definitions of what a feature is, or what it represents, have been given. Among these informal definitions we prefer the one from Kang et al. [10]:

> "features are distinctively identifiable functional abstractions that must be implemented, tested, delivered, and maintained".

Other definitions can be found in [2].

Further research and development of the concepts of products and features led to the *feature-oriented domain analysis* (FODA) [9]. In FODA features are linked to each other with so called *feature trees*, a special kind of OR/AND trees. These trees can be used to express optional and mandatory presence of features in a product.

Since the terms features, products and product families lack formal definitions, Höfner et al. [4] developed an algebra in 2006. Further research led to a "*product family algebra*" [6]. This algebra is based on the well-known concept of semirings and gives mathematical precise definitions of the above terms. Furthermore it provides mechanisms to solve important task of FOSD, e.g. 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 specification of *abstract machines*

written in the specification language *Abstract Machine Notation* (AMN). The intentions behind B can be compared to the ones that led to product family algebra. While the latter stems form a lack of a mathematical precise base of FOSD, the former was developed to offer a formal method to specify and verify software on a solid mathematical foundation. Since the B-Method is tool-supported it not only aroused interest in academia, but also in industry [12].

The goal of this thesis is to underlay the formal product family algebra with the semantics of abstract machines. To reach this goal, we develop a formal product family algebra for abstract machines and investigate the properties of this algebra. Afterwards we present a model of this algebra. In this model we can compose families of machines to build larger ones in an incremental way. We then compare the development of a system using B, to the development of this system using the presented algebraic model. Furthermore we briefly discuss how rules for correct machines given in B can be expressed in this model.

This thesis is organized as follows. In Chapter 2, we present the basics of product family algebra and discuss a standard model of the algebra. In Chapter 3, we give a brief overview of the B-Method and discuss abstract machines in detail. Afterwards we develop a calculator system using B. In Chapter 4, we give the mathematical basics of product families for abstract machines, discuss a model for abstract machines and revisit the calculator system. In Chapter 5 we show how requirements of the B-Method can be expressed in the algebra for abstract machines. In Chapter 6, we present a conclusion and give an outlook on future research. Finally Appendix A and Appendix B present deferred abstract machines and automatically generated source code for the developed calculator systems.

Chapter 2

Product Family Algebra

In this chapter we present the basics of product family algebra. This concept was introduced by Höfner et al. in [6], where a more detailed discussion can be found. As a motivation we give an example that informally introduces *features*, *products* and *product families*.

Product family	Mandatory	Optional	Commonalities
MP3 Player	– Play MP3 files	– Record MP3 files	– LCD Display – USB Connector
Video Player	– MP3 Player – Play video files	– E-Book Reader – Games	
E-Book Reader	– Show E-Books	– Play MP3 files – Games	
Mediaplayer	– MP3 Player – Video Player – E-Book Reader		

Table 2.1: Product families of a consumer electronics company

Assume a consumer electronics company which assembles several different media players. Table 2.1 shows all possible devices the company may produce. For example, the company offers two different MP3 players. One that can only play MP3 files and one that can record MP3 files additionally. These *products* are summarized in the MP3 Player *product family*. More precisely the table states which functionalities a product of a certain product family must have (columns *"Mandatory"* and *"Commonalities"*), which it may have (column *"Optional"*) and which it has in common with other

products (column *"Commonalities"*). The terms given in these columns are called *features*. Features that each product has, are called *common features*.

To shorten the upcoming calculations, we declare the following abbreviations:

Play MP3 files	`p_mp3`
Record MP3 files	`r_mp3`
Play video files	`v_alg`
Show E-Books	`e-book`
Games	`games`
Display	`lcd`
USB connector	`usb`

These expressions are the (basic) features from which the products are built. We want to express the products and product families described in the table algebraically. For Example, the MP3 Player product family can be written as

$$\texttt{mp3\_player} = \texttt{p\_mp3} \cdot \texttt{lcd} \cdot \texttt{usb} + \texttt{p\_mp3} \cdot \texttt{r\_mp3} \cdot \texttt{lcd} \cdot \texttt{usb} \ .$$

The operation + means choice between two products and $\cdot$ means (mandatory) presence of a feature within a product. This kind of structure is offered by semirings. Therefore we want to present these first.

2.1 Semirings

Semirings are an algebraic structure that have been investigated quite well. They are based on the combination of two monoids. Thus we will give a definition of monoids first.

Definition 2.1.1 A monoid is a triple $M = (S, \cdot, 1)$ with a set S, an inner operation $\cdot : S \times S \to S$, which is associative, i.e.,:

$$\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c) \ ,$$

and an identity (or neutral element) 1, i.e.,:

$$\forall a \in S : a \cdot 1 = a = 1 \cdot a \ .$$

Since we will use a special kind of monoids later on, we will give another definition.

Definition 2.1.2 A monoid $M = (S, \cdot, 1)$ has an *irreducible identity* iff the following holds:

$$\forall a, b \in S : a \cdot b = 1 \Rightarrow a = b = 1 \ .$$

It is *commutative* iff the following holds:

$$\forall a, b \in S : a \cdot b = b \cdot a \ .$$

It is *idempotent* iff the following holds:

$$\forall a \in S : a \cdot a = a \ .$$

Example 2.1.3 An example of an idempotent and commutative monoid with irreducible identity is $M = (\wp(S), \cup, \emptyset)$, where $\wp(S)$ is the power set over an arbitrary set S. It is easy to check that M is idempotent and commutative. To show that $\emptyset$ is irreducible we use contradiction. Let $A, B \in \wp(S)$ be two arbitrary sets. Assume that at least one set is non-empty. Then $A \cup B$ is also non-empty. Thus $A \cup B \neq \emptyset$ holds.

The set of all $n \times n$ matrices over $\mathbb{R}$ with the matrix multiplication as operation and the identity matrix I_n as neutral element forms the monoid $(\mathbb{R}^{n \times n}, \cdot, I_n)$. This monoid is neither commutative nor idempotent nor has it an irreducible identity.

Consider the 2×2 matrices $I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ and $B = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$. Then $A \cdot A = I_2$ holds, but $A \neq I_2$ which violates the properties *irreducible identity* and *idempotence*. Furthermore $A \cdot B \neq B \cdot A$ holds which violates commutativity. □

Let us now define a semiring.

Definition 2.1.4 A *semiring* is a quintuple $(S, +, 0, \cdot, 1)$ such that

(1) $(S, +, 0)$ is a commutative monoid with identity element 0 ,

(2) $(S, \cdot, 1)$ is a monoid with identity element 1 ,

(3) $\cdot$ distributes over $+$, i.e., $\forall a, b, c \in S : (a + b) \cdot c = a \cdot c + b \cdot c$
as well as $\forall a, b, c \in S : c \cdot (a + b) = c \cdot a + c \cdot b$,

(4) 0 is an annihilator, i.e., $\forall a \in S : 0 \cdot a = 0 = a \cdot 0$.

The semiring is *commutative* iff $\cdot$ is commutative and it is *idempotent* iff $+$ is idempotent. An idempotent semiring is called *i-semiring*. In an i-semiring 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 $\leq$.

Example 2.1.5 A commutative semiring is $(\mathbb{N}, +, \cdot, 0, 1)$ where $\mathbb{N}$ is the natural numbers set with the usual addition and multiplication as operations. Any Kleene algebra is an i-semiring [8]. □

Most often $+$ can be seen as a choice between elements and $\cdot$ as their composition. We will now characterize special elements of a commutative i-semiring, which we call *products* and *features*.

Definition 2.1.6 Assume a commutative i-semiring A. An element $a \in A$ is called a *product* if $a = 1$ or if it satisfies the following laws:

$$\forall\, b \in A : b \leq a \implies (b = 0 \vee b = a)\ , \tag{2.1}$$

$$\forall\, b, c \in A : a \leq b + c \implies (a \leq b \vee a \leq c)\ . \tag{2.2}$$

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

Having product families, and $+$ as the choice between them in mind (see Table 2.1), Implication (2.1) states, that products cannot be decomposed w.r.t. $+$. This fits well with the fact that a product does not offer a choice between alternatives, only product families can do. Implication (2.2) states that if a is included, w.r.t. $\leq$, in the sum of the elements b and c, it has to be included in b or c already. In other words, a has not been formed by summing up b and c. Again this fits well with the interpretation that the addition offers a choice between products and not a kind of mixture of them.

Since we want to build products from features, we define features as a special kind of products.

Definition 2.1.7 An element a is called a *feature* if it is a proper product different from 1 satisfying the following laws:

$$\forall\, b : b \,|\, a \implies b = 1 \vee b = a\ , \tag{2.3}$$

$$\forall\, b, c : a \,|\, (b \cdot c) \implies (a \,|\, b \vee a \,|\, c)\ , \tag{2.4}$$

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

The definition of features is quite similar to the one of products. Implication (2.3) states, that features cannot be decomposed regarding to the semiring operation $\cdot$ and $|$ (apart from the trivial decomposition into the neutral element and the element itself). Implication (2.4) states, that if a feature a splits an element $b \cdot c$, it also has to divide b or c. That means a has to be included in b or c regarding $\cdot$ and $|$. In other words features cannot be generated by multiplying one feature by another one.

Mathematically products are *irreducible* regarding $+$ and $\leq$ and features are *irreducible* regarding $\cdot$ and $|$. A discussion of these properties is given the Appendix of [5].

2.2 Product Family Algebra

Having a mathematical precise definition for products and features, we can now specify an algebra that deals with these elements. The algebra is called *product family algebra.*

Definition 2.2.1 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 $g \leq f$, where $\leq$ is the natural semiring order.

In the context of product family algebra we will interpret addition $+$ as choice between families and multiplication $\cdot$ as composition of families. In a product family algebra, products are made up of features, by composing them, and families are made up of products, by adding them. As seen in Table 2.1 product families can have common features. This leads to the following definition.

Definition 2.2.2 Assume a product family algebra. A feature c is called a *common feature* of the product families a and b iff the following holds:

$$\exists x, y : a = c \cdot x \ \wedge \ b = c \cdot y \ .$$

In the remainder we will only discuss product family algebras made up from a (finite) set of features. Formally they are defined as follows.

Definition 2.2.3 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.

The relation between feature diagrams (OR/AND trees), mentioned in Section 1, is given in Section 3 of [6]. In short, every algebraic term of a product family algebra, i.e., every family, can be translated into an OR/AND tree and vice versa.

2.3 The Set Model

As an example of a feature-generated product family algebra, we will present the so called set model. In this model families are sets of products, and products are sets of features. This model originates from [6], where a more detailed discussion is given.

Definition 2.3.1 Let $\mathbb{F}$ be a finite 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 $\mathbb{P} =_{df} \wp(\mathbb{F})$, the power set or set of all subsets of $\mathbb{F}$. A collection of products (an element of $\wp(\mathbb{P})$) is called *product family*. A special family is $1 = \{\emptyset\}$ consisting just of the empty product that has no features.

Based on $\wp(\mathbb{P})$ we define two operations to link up families. The operation $\cdot$ is intended to compose families:

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

The operation $+$ is intended to offer a choice between families:

$$\begin{aligned} &+ : \wp(\mathbb{P}) \times \wp(\mathbb{P}) \to \wp(\mathbb{P}) \\ &P + Q =_{df} P \cup Q . \end{aligned}$$

It is easily checked that with these definitions the structure

$$\mathbb{PFS} =_{df} (\wp(\mathbb{P}), +, \emptyset, \cdot, \{\emptyset\})$$

forms a product family algebra, called the *set model* over $\mathbb{F}$. It does not allow multiple occurrences of the same feature in a product nor does it enforce products of the same family to have common features. In the set model a product is a singleton set and a feature is a singleton set which only element is a singleton set again. To make things clearer and to show the use of these operations we will have a closer look at the media player product families from Table 2.1.

Example 2.3.2 Let $\mathbb{F} = \{\texttt{p\_mp3}, \texttt{r\_mp3}, \texttt{v\_alg}, \texttt{e-book}, \texttt{games}, \texttt{lcd}, \texttt{usb}\}$ be the set of all features/function given in Table 2.1. Then we can formalize the media player product families in the set model $\mathbb{PFS}$. Elements of $\mathbb{PFS}$ are sets. Thus the MP3 Player product family looks like $\texttt{mp3\_player} = \{\{\texttt{p\_mp3}, \texttt{lcd}, \texttt{usb}\}, \{\texttt{p\_mp3}, \texttt{r\_mp3}, \texttt{lcd}, \texttt{usb}\}\}$. It consists of the two products $\{\{\texttt{p\_mp3}, \texttt{r\_mp3}, \texttt{lcd}, \texttt{usb}\}\}$ and $\{\{\texttt{p\_mp3}, \texttt{r\_mp3}, \texttt{lcd}, \texttt{usb}\}\}$. A feature, e.g., is $\{\{\texttt{p\_mp3}\}\}$. As explained after the Definitions 2.1.6 and 2.1.7, products cannot be decomposed regarding $+$ and features cannot be decomposed regarding $\cdot$. Thus, and for better readability, we will omit set parentheses and will denote $\emptyset$ by 0 and $\{\emptyset\}$ by 1. Expressions involving $+$ describe product families. The MP3 Player product families can therefore be expressed as $\texttt{mp3\_player} = \texttt{p\_mp3} \cdot \texttt{lcd} \cdot \texttt{usb} + \texttt{p\_mp3} \cdot \texttt{r\_mp3} \cdot \texttt{lcd} \cdot \texttt{usb}$ in this model.

The expressions for every product family described in Table 2.1 are given by

$$\begin{aligned}
\texttt{mp3_player} &= \texttt{p_mp3}\cdot\texttt{lcd}\cdot\texttt{usb} + \texttt{p_mp3}\cdot\texttt{r_mp3}\cdot\texttt{lcd}\cdot\texttt{usb}\,,\\
\texttt{v_player} &= \texttt{mp3_player}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\\
&\quad + \texttt{mp3_player}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{e-book}\\
&\quad + \texttt{mp3_player}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{games}\\
&\quad + \texttt{mp3_player}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{e-book}\cdot\texttt{games}\,,\\
\texttt{e-b_reader} &= \texttt{e-book}\cdot\texttt{lcd}\cdot\texttt{usb}\\
&\quad + \texttt{e-book}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{games}\\
&\quad + \texttt{e-book}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{p_mp3}\\
&\quad + \texttt{e-book}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{p_mp3}\cdot\texttt{games}\,,\\
\texttt{media_player} &= \texttt{mp3_player} + \texttt{v_player} + \texttt{e-b_reader}\,.
\end{aligned}$$

Now we have a look at one of the main tasks of FOSD, that is "Determine common features". In a product family algebra we can perform this task by expanding the expressions describing the families, summing them up and using distributivity afterwards. For example, we will analyze which features the families `mp3_player` and `v_player` have in common—if any. Since the "Commonalities" column is nonempty, it is clear that the families share features, but are there other ones? The MP3 Player expression is already expanded, the video player is given by

$$\begin{aligned}
\texttt{v_player} &= \texttt{mp3_player}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\\
&\quad + \texttt{mp3_player}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{e-book}\\
&\quad + \texttt{mp3_player}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{games}\\
&\quad + \texttt{mp3_player}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{e-book}\cdot\texttt{games}\\
&= \texttt{p_mp3}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\\
&\quad + \texttt{p_mp3}\cdot\texttt{r_mp3}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\\
&\quad + \texttt{p_mp3}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{e-book}\\
&\quad + \texttt{p_mp3}\cdot\texttt{r_mp3}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{e-book}\\
&\quad + \texttt{p_mp3}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{games}\\
&\quad + \texttt{p_mp3}\cdot\texttt{r_mp3}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{games}\\
&\quad + \texttt{p_mp3}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{e-book}\cdot\texttt{games}\\
&\quad + \texttt{p_mp3}\cdot\texttt{r_mp3}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{v_alg}\cdot\texttt{lcd}\cdot\texttt{usb}\cdot\texttt{e-book}\cdot\texttt{games}\,.
\end{aligned}$$

Now we can add the MP3 Player family and use the distributivity laws to "sort" the common parts. As mentioned above, in the set model a feature cannot occur multiply in a product since $\cdot$ is idempotent in this model. Thus

we will simplify the expressions with the use of the equation $\texttt{lcd} \cdot \texttt{usb} \cdot \texttt{lcd} \cdot \texttt{usb} = \texttt{lcd} \cdot \texttt{usb}$ and commutativity in one step.

$$\begin{aligned}
&\texttt{mp3_player}+\texttt{v_player} \\
&\quad = \big(1 + \texttt{r_mp3} + \texttt{v_alg} + \texttt{r_mp3} \cdot \texttt{v_alg} \\
&\quad + \ \texttt{v_alg} \cdot \texttt{e-book} + \texttt{r_mp3} \cdot \texttt{v_alg} \cdot \texttt{e-book} \\
&\quad + \ \texttt{v_alg} \cdot \texttt{games} + \texttt{r_mp3} \cdot \texttt{v_alg} \cdot \texttt{games} \\
&\quad + \ \texttt{v_alg} \cdot \texttt{e-book} \cdot \texttt{games} + \texttt{r_mp3} \cdot \texttt{v_alg} \cdot \texttt{e-book} \cdot \texttt{games}\big) \\
&\quad \cdot \texttt{p_mp3} \cdot \texttt{lcd} \cdot \texttt{usb}
\end{aligned}$$

The product families have the features `p_mp3`, `lcd` and `usb` in common. Looking at Table 2.1 this is maybe unsurprising, since the MP3 Player family is stated as mandatory for the video player family. But note that `r_mp3` is not a common feature which is not easy to catch from the table. This is because we are interested in features which every product of the corresponding family has. Furthermore what we also recognize is that a video player needs no "new" LCD display nor a "new" USB connector, since the ones from the MP3 Player are already present and can be reused. With the same method we can also determine the common parts within one product family, e.g. the e-book reader family:

$$\texttt{e-b\_reader} = \texttt{e-book} \cdot \texttt{lcd} \cdot \texttt{usb} \cdot (1 + \texttt{p\_mp3}) \cdot (1 + \texttt{games}) \ .$$

In this form we see that the features `e-book`, `lcd` and `usb` are mandatory for an e-book reader, whereas `p_mp3` and `games` are optional. □

Looking at Example 2.3.2 we see that, even if there is quite a small amount of features and we omitted some steps, calculating common features can become pretty complicated. This is one of the scenarios where an algebraic approach shows its strengths. Due to its mathematical foundation FOSD tasks can easily be done by software tools. Especially the task presented above can be efficiently solved. Höfner states in [6] that finding the common features can be replaced by finding the *greatest common divisor (gcd)* which can be efficiently solved [3] if we identify features with prime numbers. To "Determine new products" we can build all possible combinations of the given features. This approach may also generate products we are not interested in or we even want to avoid for some reasons, e.g. security concerns. This leads us to the task "Avoid unwanted feature combinations". We can solve this problem by stating equations of the form $a \cdot b = 0$ as axioms, where a, b are features. This implies that any product in which both features occur are equal to the empty product 0. Algebraically this works because of commutativity

and 0 being an annihilator. To make things clearer assume a product p of a feature-generated algebra consisting of the features $f_1, \ldots, f_n$ ($n \geq 2$) and an axiom $f_k \cdot f_l = 0$ (w.l.o.g. $k \leq l$). Then we can conclude:

$$\begin{array}{ll} & p \\ = & f_1 \cdots f_k \cdots f_l \cdots f_n \\ = & \quad \{\![\text{ commutativity }]\!\} \\ & f_1 \cdots f_k \cdot f_l \cdots f_n \\ = & \quad \{\![\text{ axiom } f_k \cdot f_l = 0 \ ]\!\} \\ & f_1 \cdots 0 \cdots f_n \\ = & \quad \{\![\ 0 \text{ is an annihilator }]\!\} \\ & 0 \end{array}$$

"Build new product families" can be achieved by adding or composing families. Furthermore we can add/remove features from certain products within a family or remove a whole product from a family.

2.4 Refinement

As mentioned above we can generate new product families in certain ways, e.g. by adding features. Since the new and the old family correlate, we can compare them. To cover this in the algebra we introduce the *refinement relation.*

Definition 2.4.1 The *refinement* relation is defined as

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

it is a preorder.

Informally, $a \sqsubseteq b$ means that every product in a has (at least) all the features of some product in b The refinement relation is related with the divisibility relation in the following ways.

Lemma 2.4.2 *Let a, b, p be elements of a product family algebra A, than divisibility implies refinement:*
$b \,|\, a \implies a \sqsubseteq b$.

If A is feature-generated and p a product, then refinement and divisibility coincide, i.e., $a \sqsubseteq p \iff p \,|\, a$.

A proof can be found in [6]. In feature-generated algebras $a \sqsubseteq p$ means that a includes the product p .

Example 2.4.3 Looking at the consumer electronic company and Example 2.3.2 we see that the video player family refines the MP3 Player family, i.e., $\texttt{v\_player} \sqsubseteq \texttt{mp3\_player}$. That is since each product of the video player family "includes" one product of the MP3 Player family. An element c as required in Definition 2.4.1, is $\texttt{v\_player}$, since $\texttt{v\_player} \cdot \texttt{mp3\_player} = \texttt{v\_player}$ (idempotence of $\cdot$ in the set model) and therefore $\texttt{v\_player} + (\texttt{v\_player} \cdot \texttt{mp3\_player}) = (\texttt{v\_player} \cdot \texttt{mp3\_player})$. In other words, each product of the video player family has every feature of one product of the MP3 Player family, which are $\texttt{p\_mp3}, \texttt{lcd}, \texttt{usb}$ and $\texttt{p\_mp3}, \texttt{r\_mp3}, \texttt{lcd}, \texttt{usb}$ respectively. It is not necessary that every product has all features occurring in the refined family. The symmetric relation $\texttt{mp3\_player} \sqsubseteq \texttt{v\_player}$ does not hold since every video player product includes the $\texttt{v\_alg}$ feature, as seen in Example 2.3.2.
Another refinement is given by $\texttt{v\_player} \sqsubseteq \texttt{p\_mp3} \cdot \texttt{lcd} \cdot \texttt{usb}$, since $\texttt{p\_mp3} \cdot \texttt{lcd} \cdot \texttt{usb}$ is a product and $\texttt{p\_mp3} \cdot \texttt{lcd} \cdot \texttt{usb} \,|\, \texttt{v\_player}$ (have a look at the expanded $\texttt{v\_player}$ family in Example 2.3.2).

But also discarding a feature or product can refine a family. Assume that the company makes the decision to discontinue the production of an MP3 player which can record MP3 files. The new product family is then $\texttt{mp3\_player\_new} = \texttt{p\_mp3} \cdot \texttt{lcd} \cdot \texttt{usb}$. Now the new family refines the old one: $\texttt{mp3\_player\_new} \sqsubseteq \texttt{mp3\_player}$.

□

2.5 Requirements

In Section 2.3 we have seen that we can avoid unwanted feature combinations by introducing equations of the form $a \cdot b = 0$. In this section we will present another way to reach this goal as well as a way to express requirements. Höfner et al. state the goals in [6] informally as:

> "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".

Formally the relation is defined as follows.

Definition 2.5.1 Assume a feature-generated algebra. For elements a, b, c, d and a product p we define the *requirement relation* $\xrightarrow{a}$ by

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

For example, the requirement that in the MP3 Player family every product that includes the feature `r_mp3` needs also `p_mp3` can be expressed by

$$\texttt{r\_mp3} \xrightarrow{\texttt{mp3\_player}} \texttt{p\_mp3} \ .$$

Feature exclusion can be reached by expressions of the form

$$a \cdot b \xrightarrow{c} 0 \ ,$$

where a, b are features and c is a family. This means, that in the family c every product must not contain both features a and b together. When comparing the two possibilities providing feature exclusion, there is a big difference. Equations of the form $a \cdot b = 0$ apply in all families of the domain. This yields that a and b are mutual exclusive in every possible product. Whereas $a \cdot b \xrightarrow{c} 0$ only applies within the family c. This allows a more fine granular use of feature exclusion.

We will now present a lemma, which states some useful properties of the requirement relation. It deals with the requirement relation in connection with choice $(+)$ and composition $(\cdot)$ respectively. The proofs are given in Appendix A of [6], from where it is adopted.

Lemma 2.5.2 *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) $a + b \xrightarrow{e} c \iff a \xrightarrow{e} c \wedge b \xrightarrow{e} c$.

Chapter 3

Abstract Machines

In this chapter we present *abstract machines*. The concept of abstract machines was introduced by Abrial. It is the basic mechanism in the *B-Method* (B) [1]. The intention of B is to specify and verify software on a solid mathematical foundation. Therefore it is mainly used for the development of safety-critical systems. We give a short overview of the B-method. Afterwards we discuss abstract machines in detail.

3.1 B-Method

The *B-Method* is a formal method for software development, also called *B*. It covers mathematical techniques to specify, design and implement software systems. For each step mathematical proofs are mandatory. Thus the result of each phase of the development cycle is guaranteed to be correct.

The *specification phase* is done with the help of abstract machines. Abstract machines can be compared to the concept of classes or modules in various programming languages. In this phase elements identified in the requirements analysis (not part of the method) are formalized by abstract machines written in a kind of pseudo-code notation. The consistency of these machines is verified by proof obligations, e.g. the variables have to fulfill a certain invariant formalized as part of the machine. In Section 3.2 we discuss abstract machines in detail. As a result of this phase a *formal specification* is produced.

In the *design* step abstract machines can be decomposed into smaller ones if necessary. Basically the machines are transformed by several refinement steps towards executable modules. All *refinements* are checked to be correct and to satisfy the specification of the refined abstract machine by generating again proof obligations. In this phase a *formal design* is produced.

The last phase consists of two steps. First the refinements are transformed into *implementations* which are very close to low-level, executable modules. Again, these implementations are proved to be correct. Secondly, and thus in the last step of the B-Method, code is generated, Abrial [1, p. xvii] calls this "the ultimate refinement of a machine". The B-Method offers a way to translate the implementations into imperative programming languages.

All these steps may yield iterations in the process. For example, in the design phase it may be useful to decompose the system into subsystems and therefore to adjust the specification. Proof attempts of a refinement can also reveal errors in the specification of the refined machines. Every development step has its own kind of machine. Abstract machines are used for specifications, refinement machines for the design/refinement step and implementation machines for the implementation step. They are all written in the *Abstract Machine Notation* (AMN) that we present in Section 3.2. However, some constructs are not available in every type of machine. For example, recursive operations are only allowed in implementation machines and therefore at code level. Since abstract machines are the main objects we discuss, we concentrate on the part of the notation used for them.

The B-Method features 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 [12].

3.2 Abstract Machine Notation

Abstract machines are the main components of the B-Method. In [1], Abrial states that an abstract machine can informally be seen as a calculator. A calculator 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 [13]. Abstract machines are used to specify software systems or more precisely, modules of a software system. They describe only what modules/system have to be developed but not how—implementations are responsible for this. Abstract machines are written in the *Abstract Machine Notation (AMN)* that we introduce in this section. For a comprehensive description of AMN and B we refer to [1].

Formally an abstract machine consists of different clauses. We discuss only those clauses we need later on, in fact there are some more. An overview

[1] see http://www.b-core.com/btoolkit.html

[2] Atelier B can be obtained at http://www.bmethod.com

of all possible clauses is given in Appendix B of [1]. 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*. A detailed description and theoretical discussion is given in [1].

Syntax	Semantic
MACHINE `Name(parameters)` **INCLUSION CLAUSES**	Machine header The **parameters** are a list of scalars or finite, non-empty sets offering instantiation. Inclusion clauses may be listed here.

CONSTRAINTS
`Predicate`
SETS
`Sets`
CONSTANTS
`Constants`
PROPERTIES
`Predicate`
	Static data
	
	The **CONSTRAINTS** predicate is used to restrict the machine parameters.
	
	The **PROPERTIES** clause is given as a predicate involving **CONSTANTS** and **SETS**.

VARIABLES
`Variables`
INVARIANT
`Predicate`
INITIALIZATION
`Substitution`
	State
	
	The **VARIABLES** are initialized by the **INITIALIZATION** clause. They must satisfy the **INVARIANT** predicate during the whole lifetime of the machine.

OPERATIONS `Operations`	Dynamics **OPERATIONS** are responsible for input/output tasks as well as for modifying the **VARIABLES**.

Table 3.1: Structure of an abstract machine

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 clauses are given in Table 3.1. They are all optional, except the **MACHINE** clause. However, there are certain dependences between them. We give the dependences when we discuss the clauses in the following subsections.

To type or restrict parameters, constants, sets and variables some constants and sets are predefined in AMN.

`NATURAL`	$\mathbb{N}$
`INTEGER`	$\mathbb{Z}$
`maxint : NATURAL`	$\texttt{maxint} \in \mathbb{N}$
`maxint : INTEGER`	$\texttt{minint} \in \mathbb{Z}$
`NAT = minint..maxint`	$\texttt{INT} = \{z \in \mathbb{N} \ : \ 0 \leq z \leq \texttt{maxint}\}$
`NAT1 = minint..maxint`	$\texttt{INT} = \{z \in \mathbb{Z} \ : \ 1 \leq z \leq \texttt{maxint}\}$
`INT = minint..maxint`	$\texttt{INT} = \{z \in \mathbb{Z} \ : \ \texttt{minint} \leq z \leq \texttt{maxint}\}$
`INT1 = INT - NAT`	$\texttt{INT1} = \mathbb{Z} \setminus \mathbb{N}$
`CHAR = 0..255`	$\texttt{CHAR} = \{z \in \mathbb{N} \ : \ 0 \leq z \leq 255\}$
`BOOL = TRUE, FALSE`	$\texttt{BOOL} = \{0, 1\}$

Table 3.2: Predefined constants of AMN

Table 3.2 gives the predefined constants and sets in the ASCII compatible AMN and in the corresponding mathematical notation. The constants `minint` and `maxint` depend on the architecture of the target platform.

3.2.1 Machine Header

The machine header consists of the mandatory **MACHINE** clause and the optional inclusion clauses. In the **MACHINE** clause the name of the machine is given. If necessary, a list of `parameters` can be declared. Parameters are intended to offer instantiation by other machines. Therefore using parameters produces a set of machines rather than just one. Parameters are given as a comma-separated list of identifiers surrounded by brackets directly after the machine name. Only numeric data types, called *scalars*, and sets are allowed as parameters. By convention scalars are denoted in lower case and sets in upper case. The various inclusion clauses relate data from different machines to each other. Therefore we present this data first. Afterwards we discuss the inclusion clauses in Subsection 3.2.5.

3.2.2 Static Data

The static data of an abstract machine consists of the data which cannot be modified within the machine. In particular this data is given in the clauses:

- **CONSTRAINTS**
- **SETS**
- **CONSTANTS**
- **PROPERTIES**

The optional **CONSTRAINTS** clause is associated with the **parameters**. It consists of a number of predicate conjunctions. These constraints allow to determine the type of a scalar parameter or to state that it is a member of a parameter set. Further properties, like the cardinality of sets or a maximum value of a scalar can also be stated here. However, parameter sets cannot be "typed" in terms of being a subset of another set. Therefore they are called *independent types*. If no constraint is given for a particular parameter, it is assumed to be a scalar or a non-empty set.

Example 3.2.1 For example have a look at the following abstract machine.

MACHINE
```
Params(scalar_a, scalar_b, SET_A)
```

CONSTRAINTS
```
scalar_a : NAT &
scalar_a < 100 &
card(SET_A) < 10
```

The abstract machine `Params` has two scalar parameters and one set parameter. The parameter `scalar_a` is not only typed as a natural number, but also enforced to be less than 100. Whereas for `scalar_b` no predicate occurs in the **CONSTRAINTS** clause. Thus, `scalar_b` is implicitly typed as being an integer. Furthermore the cardinality (number of elements) of the set parameter `SET_A` is restricted to 10. □

The **SETS** clause allows us to define given sets, and therefore own types, within an abstract machine. There are two kinds of sets, namely *enumerated sets* and *deferred sets*. Enumerated sets are sets whose elements are given explicitly. They must have distinct elements. If no members are specified a

set is called *deferred set*. Deferred sets are implicitly assumed to be finite and non-empty. Like `parameter` sets, deferred sets are independent types. A set is declared by an upper case identifier, optionally followed by an enumeration of its members in curly braces. We give an example of this clause at the end of this subsection.

The **CONSTANTS** clause consists of an identifier list. If constants are given, they must be typed in the **PROPERTIES** clause. However, they do not necessarily have a (explicit) value. This is since abstract machines are used for specification.

The **PROPERTIES** clause can be compared to the **CONSTRAINTS** clause and the later mentioned **INVARIANT** clause but with respect to constants and sets. All constants have to be typed in this clause and further restrictions can be given. This is done by stating a conjunction of predicates involving the constants. Possible types of constants are scalar constants of sets, a total function from a set to a set or a subset of a scalar set. Since functions are relations, they are also sets. However, sets must not have a related predicate stated in the **PROPERTIES** clause. Furthermore no values can be assigned to deferred sets. They may be valued in the last refinement step which is an **IMPLEMENTATION** machine.

We now give an example of the just discussed clauses.

Example 3.2.2

```
MACHINE
        Bool_Ar

SETS
        INDEX_SET;
        WEEK_DAYS = { Mon, Tue, Wed, Thu, Fri, Sat, Sun }

CONSTANTS
        const_a, func

PROPERTIES
        const_a : INDEX_SET &
        func : INDEX_SET --> BOOL
```

The abstract machine `Bool_Ar` has two constants and two sets. The set `INDEX_SET` is a deferred set; it is not further restricted in the **PROPERTIES** clause. The set `WEEK_DAYS` is an enumerated set. The constant `const_a` is

typed as a member of the set INDEX_SET and the func constant is typed as a total function from INDEX_SET to the (predefined) set of Boolean values BOOL. Without going into detail, we give a possible implementation of this machine.

```
IMPLEMENTATION
        Bool_Ar_i

REFINES
        Bool_Ar

VALUES
        const_a = 1;
        INDEX_SET = 0..2;
        func = { 0 |-> TRUE, 1 |-> FALSE, 2 |-> TRUE }
```

The **VALUES** clause can only appear in an implementation machine. It is used to assign values to given constants and sets. We see that the enumerated set WEEK_DAYS does not occur in the **VALUES** clause. This is because it can be translated directly into a programming language construct, e.g. `enum WEEK_DAYS{Mon, Tue, Wed, Thu, Fri, Sat, Sun}` in C. The set INDEX_SET is mapped to the integer interval 0..2. Since func was declared as a total function in the specification machine, a mapping for each value of the domain (INDEX_SET) to a value of the codomain (BOOL) must be specified. Using a B to C translator, it may become a constant bool array, e.g. `const bool func[3] = {true, false, true}`. □

3.2.3 The State of an Abstract Machine

The *state of an abstract machine* consists of the clauses

- **VARIABLES**,
- **INVARIANT**,
- **INITIALIZATION**.

The main components are the variables that are listed in the **VARIABLES** clause. They can be compared to fields, or class variables, in programming languages such as Java. If a **VARIABLES** clause is given, the **INVARIANT** and **INITIALIZATION** clauses are mandatory.

The **INITIALIZATION** clause is used to assign initial values to the given variables. This is done by a combination of substitutions, each dealing with exactly one variable, called multiple substitution. Substitutions are discussed in Subsection 3.2.4.

The **INVARIANT** clause fixes the boundaries in which the variables can be modified. More precisely, the variables must fulfill the predicate stated in the **INVARIANT** clause during the whole "lifetime" of the abstract machine. This means that the invariant must hold after they got their initial values (**INITIALIZATION**) and also after they got new values. The latter is done by operations. Operations must also preserve the invariant in terms of: "If the invariant holds before the execution of the operation, it must also hold afterwards". From outside the machine, the state cannot be changed directly. But one might call internal operations the machine offers to modify the state. This is due to the hiding principle mentioned in Section 3.2. In Java this can be simulated by declaring all fields `private`. The **INVARIANT** clause is stated by a conjunction of predicates. For each variable at least a typing predicate has to be given; further restrictions are optional.

We give an abstract machine containing the presented clauses as an example.

Example 3.2.3

```
MACHINE
        Var(max_index)

CONSTRAINTS
        max_index : NAT

SETS
        INDEX_SET;
        WEEK_DAY = { Mon, Tue, Wed, Thu, Fri, Sat, Sun }

VARIABLES
        key, ind, day, sub, pair

INVARIANT
        key : NAT
        & key <= max_index
        & ind : INDEX_SET
        & day : WEEK_DAY
        & sub <: INDEX_SET
        & pair : (INT <-> INT)
```

```
        & ({0,99} <| pair) = {}

INITIALIZATION
        key := 0
        || ind :: INDEX_SET
        || day := Mon
        || sub := {}
        || pair := {(1|->2)}
```

The given abstract machine `Var` contains five different variables. The variable `key` is typed as a natural number, restricted to be less than or equal to the machine parameter `max_index` and initialized with 0. The variable `ind` is declared as a member of the set `INDEX_SET`. Since `INDEX_SET` is a deferred set which is not typed—the type of its members is unknown—we cannot assign a specific value to `ind`. Therefore we state in the clause **INITIALIZATION** that `ind` "becomes a member of" `INDEX_SET` by the substitution `ind ::INDEX_SET`. The variable `day` has the type `WEEK_DAY` and is initialized with the enumerated set element `Mon`.

The variable `pair` represents a relation as an example for specifying complex data types. It is initialized by `pair := {(1|->2)}`, i.e., by the relation containing the tuple $(1, 2)$. In the **INVARIANT** clause the predicate `({0,99} <| pair) = {}` restricts this relation. The symbol `<|` here means domain restriction which is mathematically defined by $S \lhd R = \{(a, b) | (a, b) \in R \wedge a \in S\}$ for an arbitrary set S and an arbitrary relation R. Thus, the given predicate states that the relation `pair` must not contain any tuple where the first component is 0 or 99. This is equivalent to `dom(pair)/\{0,99}= {}` where `dom(pair)` is the domain of `pair` and `/\` is the usual set intersection.

Sets stated in the `parameters` can be used for typing variables also. In this, they are treated in the same way as deferred sets (e.g. `INDEX_SET`). □

3.2.4 Operations

The operations of an abstract machine are intended to handle the modification of the machine state. Because of the hiding principle [13] this modification cannot be done directly from "outside", but only within the machine. This principle is very important for the B-Method, since it allows a *refinement* from an abstract machine towards an implementation. During these refinement steps the change of variables and operation bodies is usually necessary while the operation signatures remain constant. Thus the operations are a kind of figurehead for an abstract machine. Their role is to specify a pos-

sibility to modify and inquiry data of the abstract machine from the outside. Since abstract machines are used in the specification phase, we do not have to worry about poor performance or how the operations are implemented in detail.

All modifications of the machine state have to preserve the invariant mentioned above. For this purpose we have to state a *proof obligation* for each operation and prove them afterwards. These proof obligations can be generated by a B toolkit. The toolkit also tries to prove them automatically. We give an example of such a proof after discussing the structure of operations.

Table 3.3 shows the structure of an abstract machine operation. It consists of an operation header followed by a substitution. There are four different types of operation headers. The most general operation header is `Id_list <-- Identifier(Id_list)`; it offers a list of input parameters and a list of output parameters. The other types can be derived by omitting one or both parameter list, which corresponds to stating an empty list.

Operation declaration	`Operation_header = Substitution`

`Operation_header`	`Id_list <-- Identifier(Id_list)`
	`Identifier(Id_list)`
	`Id_list <-- Identifier`
	`Identifier`

Table 3.3: Structure of an abstract machine operation

The "body" of an operation is a generalized substitution, which is an extension of simple substitutions. The notion of *substitution* is formally defined as follows. Let x be a *variable*, E be an *expression* and F be a *Formula*, then the *simple substitution*

$$[x := E]F$$

denotes the *Formula* obtained by replacing all *free* occurrences of x in F by E. To substitute more than one variable at once, a *parallel* or *multiple* substitution can be stated by

$$[x, y := C, D]F \ .$$

It is defined by

$$[x, y := C, D]F \Leftrightarrow_{df} [z := D][x := C][y := z]F \quad \text{if } x \backslash z \text{ and } z \backslash (x, y, C, D, F) \ ,$$

where x, y, z are pairwise distinct variables, C, D expressions, F a formula and $\backslash$ denotes non-freeness, e.g. $z \backslash F$ means that z does not occur in F without being bound by an quantifier. The definitions are adopted from [1], where a detailed discussion of substitutions is given. In AMN the parallel substitution can be written in one of the following equivalent forms:

$$\begin{aligned} & x, y := C, D \\ & \Leftrightarrow \\ & x := C \ || \ y := D \ . \end{aligned}$$

As mentioned above, every abstract machine operation must fulfill the invariant. This is one of the proof obligations to be proved when verifying an abstract machine. If I denotes the invariant and S is a substitution, this proof obligation can be stated by

$$I \Rightarrow [S]I \ .$$

This can be read as "If the invariant I holds then the substitution S establishes the predicate/invariant I". If this implication can be proved, then the operation or, more precisely the substitution is guaranteed to correctly modify the machine state.

We give an example of an abstract machine that offers two operations, one to decrement an internal variable and one to increment it.

```
MACHINE
        Counter

VARIABLES
        count

INVARIANT
        count : NATURAL

INITIALIZATION
        count := 0

OPERATIONS
```

```
increment =
   count := count + 1;

decrement =
   count := count - 1;
```

The term `count := count + 1` is a (simple) substitution of the form `Variable := Expression` that substitutes the expression on the right-hand side for the variable on the left-hand side. The proof obligation for the operation `increment` is:

$$count \in \mathbb{N} \Rightarrow [count := count + 1](count \in \mathbb{N}) \ .$$

That means if *count* is a natural number, it is still a natural number after increasing it by 1. After replacing all free occurrences of *count* in the predicate $(count \in \mathbb{N})$, we get:

$$count \in \mathbb{N} \Rightarrow count + 1 \in \mathbb{N} \ ,$$

which clearly holds. Thus `increment` preserves the invariant.

If we transform the proof obligation for `decrement` we get:

$$count \in \mathbb{N} \Rightarrow count - 1 \in \mathbb{N} \ .$$

This statement is obviously not valid if *count* is equal to 0. Therefore this operation breaks the invariant and may lead to a system crash in a final implementation. One solution is to introduce a *pre-condition* to the substitution. This yields generalized substitutions, cf. Table 3.4.

Substitution	`Variable_list := Expression_list`
	`skip`
	`Predicate | Substitution`
	`Substitution [] Substitution`
	`Predicate ==> Substitution`
	`@Variable .Substitution`

Table 3.4: Structure of a generalized substitution

We will discuss them one by one, starting with the substitution `Predicate | Substitution` . Let P, R be predicates and S, T, U be substitutions from now on. This substitution is called *pre-conditioned substitution* and denoted by

$$P \mid S \ ,$$

with the property

$$[P \mid S]R \Leftrightarrow P \wedge [S]R \ .$$

It is pronounced "P pre S" and is valid iff P is valid and S establishes R. P and R are called *pre-condition* and *post-condition* respectively. Note that if the substitution is executed even though the pre-condition is not valid, the substitution may lead to an undefined state. In AMN it is stated by

`PRE` P `THEN` S `END` .

The pre-condition P indicates exactly those cases where the substitution S takes place. As mentioned above, we can "fix" the operation `decrement` by this kind of substitution. Therefore we modify the operation as follows.

```
decrement =
PRE
 0 < count
THEN
 count := count - 1
END
```

The proof obligation for this version is

$$(count \in \mathbb{N} \wedge 0 < count) \Rightarrow [0 < count \mid count := count - 1](count \in \mathbb{N}) \ .$$

This can be simplified to

$$(count \in \mathbb{N} \wedge 0 < count) \Rightarrow [count := count - 1](count \in \mathbb{N}) \ .$$

This statement clearly holds. For the goal of a valid machine we can also use a *conditional substitution* which is a combination of a *guarded substitution* and a *bounded choice substitution*. The *guarded substitution*

$$P \Rightarrow S \ ,$$

pronounced "P guards S", is defined by

$$[P \Rightarrow S]R \Leftrightarrow (P \Rightarrow [S]R) \ .$$

Informally $P \Rightarrow S$ reads as: "S is performed under the assumption P". Note that there is a difference between "P pre S" and "P guards S". If we want to prove that in the former case S establishes a post-condition, we first have to check if P holds. If P does not hold, S is said to "abort". In the latter case P can be assumed to hold (implication) for proving that S establishes a post-condition. If P does not hold, S may establish anything.

The next substitution we present, is the *bounded choice substitution*

$$S \ [] \ T \ .$$

It is pronounced "S choice T"and is defined by

$$[S \ [] \ T]R \Leftrightarrow ([S]R \wedge [S]T) \ .$$

In AMN it is written as

$$\texttt{CHOICE}\ S\ \texttt{or}\ T\ \texttt{or}\ \cdots\ \texttt{or}\ U\ \texttt{END} \ .$$

It offers a *choice* between two or more substitutions and therefore introduces a kind of non-determinism. It is meant to allow specifications of alternative behavior rather than letting the implemented operation decide which statement will be executed. In order to establish the post-condition any substitution of the choice has to establish it.

Having "P guards S" and "S choice T" we can define a *conditional substitution* by

$$(P \Rightarrow S) \ [] \ (\neg P \Rightarrow T) \ .$$

In AMN it is written as

$$\texttt{IF}\ P\ \texttt{THEN}\ S\ \texttt{ELSE}\ T\ \texttt{END}$$

and has the property

$$[\texttt{IF}\ P\ \texttt{THEN}\ S\ \texttt{ELSE}\ T\ \texttt{END}]R \Leftrightarrow (P \Rightarrow [S]R) \wedge (\neg P \Rightarrow [T]R) \ .$$

Another useful substitution is `skip` with the property $[\texttt{skip}]R \Leftrightarrow R$. In other words `skip` does nothing. Using `skip`, a "short" conditional substitution is give by

$$\texttt{IF}\ P\ \texttt{THEN}\ S\ \texttt{END} \quad =_{df} \quad \texttt{IF}\ P\ \texttt{THEN}\ S\ \texttt{ELSE skip END} \ .$$

It has the property

$$[\texttt{IF}\ P\ \texttt{THEN}\ S\ \texttt{END}]R \Leftrightarrow (P \Rightarrow [S]R) \wedge (\neg P \Rightarrow R)\ .$$

This is neither equivalent to the pre-conditioned substitution nor to the guarded substitution presented earlier, since `IF P THEN S END` performs an "identity substitution" if `P` does not hold. This means that all variables are substituted for themselves. Therefore this substitution has a deterministic behavior.

The short conditional version of `decrement` looks like this:

```
decrement =
IF
 count > 0
THEN
 count := count - 1
END
```

In this version no pre-condition has to be proved as it was necessary for the "P pre S" version.

A situation in which a pre-condition must be stated is given if an operation has input parameters. Consider the following operation `increase` which increases the counter by a given value.

```
increase(value) =
          count := count + value;
```

When no type is given for the parameter `value` it is neither possible to check if the given invariant `count : NAT` is preserved nor whether the addition is defined at all. For example, the invariant is violated if `value` $<$ $-$`counter`. That is because the result of the addition is lesser than zero in this case. Therefore input parameters have to be typed in the pre-condition at least. Additional restrictions can be stated. A correct specification of `increase` is

```
increase(value) =
PRE
 value : INT & value > 0
THEN
 count := count + value
END
```

The pre-condition states, that `increase` can only be called with an integer input parameter which is greater than zero.

The last remaining generalized substitution from Table 3.4 is the so called *unbounded choice substitution*

$$@z \ .S \ ,$$

where z is a variable which does not appear in the invariant. It defines all possible substitutions S, whatever the value of z is. Therefore it is pronounced "any z S". Establishing a post-condition is defined by

$$[@z \ .S]R \Leftrightarrow \forall z.[S]R \ .$$

In AMN this substitution is combined with the guarded substitution and is used in two shapes. The first one is

$$\texttt{ANY} \ z \ \texttt{WHERE} \ P \ \texttt{THEN} \ S \ \texttt{END} \ ,$$

with the formal definition

$$@z \ .(P \Rightarrow S) \ .$$

The second one is

$$x :: E \ .$$

It is an abbreviation of

$$\texttt{ANY} \ z \ \texttt{WHERE} \ z : E \ \texttt{THEN} \ x := E \ \texttt{END} \ ,$$

where x and z are variables and E is a set. The operator $::$ is pronounced "becomes a member of", since it means that an arbitrary element of E should be substituted for x.

The usage of operations is restricted as follows. It is forbidden to specify a recursive operation in an abstract machine. However this is possible in an implementation machine. It is not allowed to call an operation of the same abstract machine within an operation. Furthermore it is not allowed to call two or more operations of an included machine within a parallel substitutions, since it may break the invariant. Such inclusions are discussed in Subsection 3.2.5, where we given an example for such a situation.

3.2.5 Inclusion Clauses

In this subsection we introduce the inclusion clauses used in an abstract machine. These clauses are located in the machine header and can be compared to import statements in Java or header files in C. They are used to structure the development of a system. In an abstract machine the following clauses are possible:

- **SEES**
- **USES**
- **INCLUDES**
- **EXTENDS**
- **PROMOTES**

These clauses consist of a (distinct) list of machine names. The main differences between them are which components of the listed machines can be used by the abstract machine stating the specific clause and how the components can be used. We will briefly outline the use of the clauses. Assume three abstract machines M_1, M_2 and M_3.

If M_1 **SEES** M_2 the constants and sets of M_2 are visible in M_1; variables of M_2 are only visible as read-only in the operations of M_1, i.e., they cannot be modified in M_1. Operations and parameters are not visible.

If M_1 **USES** M_2 the constants and sets of M_2 are visible in M_1; variables and parameters of M_2 are visible in the invariant and in the operations of M_1; variables are read-only in operations. Operations are not visible.

If M_1 **INCLUDES** M_2 the constants and sets of M_2 are visible in M_1; variables of M_2 are visible in the invariant and in the operations of M_1; variables are read only in operations. Parameters are invisible, since they have been instantiated in M_1. Operations of M_2 are visible in operations of M_1, but do not become operations of M_1, i.e., they are invisible for machines which lists M_1 in any of the inclusion clause.

The **PROMOTES** clause requires the **INCLUDES** clause, since it enlists operations (by name) of an included machine. The listed operations become part of the including machine.

The **EXTENDS** clause is a combination of the clauses **INCLUDES** and **PROMOTES**. If M_1 extends M_2, it includes M_2 and promotes all operations of M_2.

SEES and **USES** are used to support shared access to read-only variables, common types and constants, but not operations, between abstract machines.

Both clauses are *intransitive*, i.e., if M_1 sees/uses M_2 and M_2 sees/uses M_3 then M_1 does not have access to data of M_3. Whereas **INCLUDES** and **EXTENDS** are used for exclusive access, since a particular machine can be included/extended only by one abstract machine in a coherent system. This is due to *transitivity* of these clauses, i.e., if M_1 includes/extends M_2 and M_2 includes/extends M_3 then M_1 implicitly includes/extends also M_3.

As mentioned in Subsection 3.2.4 an including machine must not call two operations of an included machine within one parallel substitution. Otherwise, the associated operation might break the invariant. We give an example of such a situation, adopted from [1].

Example 3.2.4 Assume the following abstract machine providing two preconditioned substitutions.

```
MACHINE
        M2

VARIABLES
        v,w

INVARIANT
        v : NAT & w & NAT & v <= w

INITIALIZATION
        v := 0 || w := 0



OPERATIONS
        increment =
           PRE v < w THEN v := v + 1;

        decrement =
           PRE v < w THEN w := w - 1;
```

Let the following machine include the machine `M2` now.

```
MACHINE
    M1

INCLUDES
    M2
```

```
OPERATIONS
    inc_decr=
      PRE v < w THEN increment || decrement END
```

The operation `inc_decr` violates the invariant `v <= w` if `w` is equal to `v+1`. Even though each of the pre-conditions stated in `M2` hold, `v` is equal to `w+1` after the parallel substitution `inc_decr` is performed. □

3.3 A Basic Calculator

In this section we develop a core system for a basic calculator using the B-Method. The development steps, especially the proofs, are done with Atelier B. The automated translation from implementation machines to C source code is done by the "ComenC"[3] translator. This translator can automatically generate C code from a subset of the B language called *B0*. Implementations written in B0 are subject to restrictions. For example, machine parameters and renaming of machines are not allowed. Therefore we avoid these concepts, although they might be useful.

In the requirements analysis the following requirements for the core system were identified:

1. The system has to offer integer arithmetic. Now overflow can occur.
2. Basic operations are: addition, subtraction, multiplication and division.
3. Since the display can show at most eight symbols, the operations have to indicate unrepresentable results.
4. Modular structure of the system to allow a possible reuse.
5. The system has to provide a possibility to recall the last valid result.

In the design phase we determined the structure of the calculator core system as given in Figure 3.1. The required machines are given in boxes. The labeled arrows indicate the inclusion relations between them.

[3]ComenC can be obtained at http://www.comenc.eu

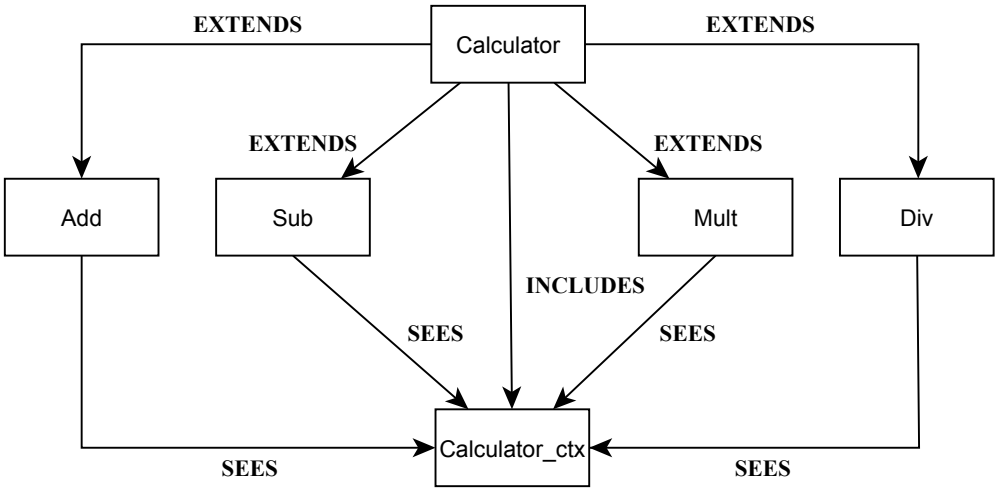


Figure 3.1: Architecture of a basic calculator core system

The calculator core system consists of six machines. The abstract machines `Add`, `Sub`, `Mult` and `Div` deal with the arithmetics as well as error indication. Additionally they store the last valid result calculated by their operations.

The abstract machine `Calculator_ctx` is responsible for the distribution of contextual information. Therefore, every other machine **SEES** this machine. The provided information covers the minimal/maximal value the display can show, as well as the error codes. This way each machine can use the same codes to indicate an error.

The abstract machine `Calculator` acts as a facade, or interface, to the "outside", e.g. another machine dealing with input/output tasks. Therefore it bundles the abstract machines `Add`, `Sub`, `Mult` and `Div` by extending them. Since the later given **IMPLEMENTATION** `Calculator_i` will import `Calculator_ctx`, the abstract machine `Calculator` has to include this machine.

In the remainder of this section, we present that part of the system, that consists of the machines `Calculator`, `Add` and `Calculator_ctx`. The remaining machines are very similar to `Add`. Furthermore we omit the header of the generated files in this section. The full specification, its implementation and the generated code is given in Appendix A.

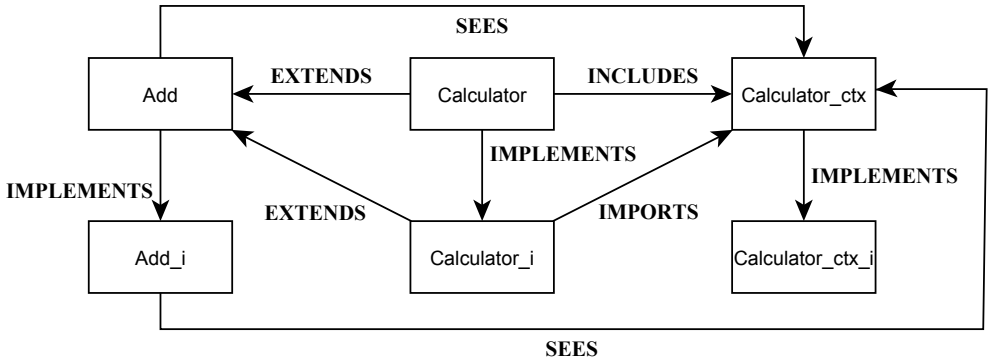


Figure 3.2: The calculator machines and their **IMPLEMENTATIONS**

Figure 3.2 shows the abstract machines and their implementations of the subsystem we consider. Each of the abstract machines is refined by their **IMPLEMENTATION** denoted by `Calculator_i`, `Calculator_ctx_i` and `Add_i` respectively. The dependences between them are illustrated by labeled arrows. Next, we present the machines shown in Figure 3.2 in detail.

```
MACHINE
    Calculator_ctx

SETS
    STATE = { ok, overflow, underflow, div_by_zero }

CONSTANTS
    min_dsp_number, max_dsp_number

PROPERTIES
    min_dsp_number : INT
    & min_dsp_number < 0
    & max_dsp_number : NAT
```

The "ComenC" translator cannot handle formal parameters, therefore the abstract machine `Calculator_ctx` contains two constants `min_dsp_number` and `max_dsp_number` representing the minimum and the maximum number the display can show. One constant is not sufficient, since the display has also to show the sign of negative numbers. Thus for negative numbers there are only seven digits available. With respect to the reusability requirement (4.) from the listing above, it is not a good idea to "hard-code" these values

into the arithmetic machines—it is usually a bad idea to do so, anyway. A benefit of this approach is, if the properties of the display change, e.g. in new calculator generation, the minimal/maximal representable value has to be modified only in the `Calculator_ctx` machine.

Furthermore the machine contains an enumerated set `STATE`. Its elements are the error-codes intended to indicate the validity of an operation result. We can refine this machine directly by an implementation machine.

```
IMPLEMENTATION
    Calculator_ctx_i

REFINES
    Calculator_ctx

VALUES
    max_dsp_number = 99999999;
    min_dsp_number = -9999999
```

In the implementation the final values are assigned to the constants. For a description of the **VALUES** clause see Example 3.2.2. The enumerated set `STATE` needs not to be valued, since it can be translated into a enum directly.

```
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
#include "Calculator_ctx.h"

/*------------------------
   INITIALISATION Clause
  ------------------------*/
void Calculator_ctx__INITIALISATION(void) {
}
```

Listing 3.1: Generated source file for the context machine

```
#ifndef _Calculator_ctx_h
#define _Calculator_ctx_h
/*--------------------------
   Added by the Translator
  --------------------------*/
```

```
#include <stdbool.h>
/*------------------------------
   SETS Clause: enumerated sets
  ------------------------------*/
typedef enum {
   Calculator_ctx__ok,
   Calculator_ctx__overflow,
   Calculator_ctx__underflow,
   Calculator_ctx__div_by_zero
} Calculator_ctx__STATE;


/*-----------------------------------------------
   CONCRETE_CONSTANTS Clause: scalars and arrays
  -----------------------------------------------*/
const int Calculator_ctx__max_dsp_number = 99999999;
const int Calculator_ctx__min_dsp_number = -9999999;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
extern void Calculator_ctx__INITIALISATION(void);

#endif
```

Listing 3.2: Generated header file for the context machine

The automatically generated C source file as well as the corresponding header file for the `IMPLEMANTAION Calculator_ctx_i` is given in Listing 3.2 and Listing 3.1 respectively. As expected the constants have been translated into `const int` types. The set `STATE` has been translated into an `enum` of the type `Calculator_ctx__STATE`.

Next we present the abstract machine `Add` and afterwards its implementation `Add_i`.

```
MACHINE
    Add

SEES
    Calculator_ctx

VARIABLES
    last_result, status

INVARIANT
    last_result : INT & status : STATE
```

```
INITIALIZATION
    last_result := 0 || status := ok

OPERATIONS
    result <-- add(op1, op2) =
    PRE
        op1 : INT & op2 : INT & (op1 + op2) : INT
    THEN
        IF
            (op1 + op2) <= max_dsp_number
            & (op1 + op2) >= min_dsp_number
        THEN
            status := ok || last_result := op1 + op2
            || result := op1 + op2
        ELSE
            status :: STATE - ok || result := 0
        END
    END;

    last_res <-- ans =
        last_res := last_result

    ret_status <-- get_status =
        ret_status := status
```

The abstract machine `Add` lists the machine `Calculator_ctx` in its **SEES** clause. Hence it can read the data stated there. The operation `add` has a pre-condition that states that the calculation of the sum does not cause an over/-underflow. This is derived from Requirement 3.; specifying an overflow detection is not in the scope of this thesis. In the operation `add` the two constants `min/max_dsp_number` are used to decide whether the sum of the operand can be displayed or not. If it can be displayed, the variable `status` is replaced by `ok` and the `result`, as well as the `last_result`, is reaplaced by the sum of `op1` and `op2`. If not, the variable `status` is replaced by an element of (`STATE - {ok}`) where `-` is the usual set difference. This is an example of the *unbounded choice substitution*. We leave it to the implementer to set the appropriate status. The variable `result` is set to be zero. Note that the variable `last_result` is not modified in this case, since we want to store only "valid", in terms of representable, results.

Next, we present the implementation machine `Add_i` which implements its specification `Add`. In an implementation machine only concrete variables

are allowed, since these are meant to be translated (directly) into variables of a programming language without being refined first. Furthermore operations have to be stated by sequenced substitutions; parallel substitutions as in abstract machines are not allowed. Sequencing is denoted by a semicolon.

```
IMPLEMENTATION
    Add_i

REFINES
    Add

SEES
    Calculator_ctx

CONCRETE_VARIABLES
    last_result, status

INITIALIZATION
    status := ok ;
    last_result := 0

OPERATIONS
    last_res <-- ans =
    BEGIN
        last_res := last_result
    END;

    ret_status <-- get_status =
    BEGIN
        ret_status := status
    END;

    result <-- add ( op1 , op2 ) =
    BEGIN
        VAR temp IN
            temp := op1 + op2;
            IF temp <= max_dsp_number
                & temp >= min_dsp_number
            THEN
                status := ok;
                result := temp;
                last_result := result
            ELSE
```

```
                IF
                    temp < min_dsp_number
                THEN
                    result := 0;
                    status := underflow
                ELSE
                    result := 0;
                    status := overflow
                END
            END
        END
    END
```

In the refined operation `add` the former unbounded choice substitution `status :: STATE - {ok}` is now given by a conditional substitution. This determines the `status` variable more precisely. The files generated with the "ComenC" translator are given in Appendix A.

The last machines we present are `Calculator` and its implementation `Calculator_i`. As mentioned above the machines are reduced by the parts dealing with the abstract machines `Sub, Mult, Div`.

```
MACHINE
    Calculator

EXTENDS
    Add/* ,Sub,Mult,Div */ /* cf. Appendix A */

INCLUDES
    Calculator_ctx

VARIABLES
    answer, state

INVARIANT
    answer : INT & state : STATE

INITIALIZATION
    answer := 0 || state := ok

OPERATIONS
    last_res <-- get_answer =
        last_res := answer;
```

```
res <-- addition(op1,op2)=
PRE op1 :INT & op2 :INT & op1+op2 :INT
THEN
    res <-- add(op1,op2) || answer :: INT || state :: STATE
END;
/* for the operations sub, mult, div cf. Appendix A */
```

The variable `answer` is used to store the last valid result of an operation; here only of the operation `addition`. The variable `state` indicates whether the last operation was successful or not; successful in terms of representable by the display. The values of `answer` and `state` are determined during the execution of `add`. Since sequencing is not allowed in an abstract machine we use the unbounded choice substitution "becomes a member". This substitution states that the variables may get modified by the operation. The operation `get_answer` allows the user to "ask" for the value of `answer`. The `IMPLEMENTATION` is given by

```
IMPLEMENTATION
    Calculator_i

REFINES
    Calculator

EXTENDS
    Add/* ,Sub,Mult,Div */ / *cf. Appendix A */

IMPORTS
    Calculator_ctx

CONCRETE_VARIABLES
    answer, state

INITIALIZATION
    answer := 0; state := ok

INVARIANT
    answer : INT & state : STATE

OPERATIONS
    last_res <-- get_answer =
        last_res := answer;
```

```
    res <-- addition(op1,op2)=
    BEGIN
        res <-- add(op1,op2);
        state <-- get_status;
        IF  state = ok
        THEN
            answer := res
        END
    END
    /* for the operations sub, mult, div cf. Appendix A */
```

According to [1] a **seen** abstract machine should be imported (only once) somewhere in the system by an `IMPLEMENTATION`. In the calculator system this is done by `Calculator_i` because it extends all other machines. Since basic substitutions are linked by sequencing in an implementation, we can now assign the correct values to the variables `answer` and `state`. The variable `answer` is only modified if the `state` of the calculation performed before, is valued by `ok`. Thus `answer` always stores the last representable result. We given an excerpt of the generated source code; full details can be found in Appendix A.

```
/*----------------------------
   CONCRETE_VARIABLES Clause
  ----------------------------*/
int Calculator__answer;
Calculator__STATE Calculator__state;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
void Calculator__INITIALISATION(void) {
   Calculator__answer = 0;
   Calculator__state = Calculator__ok;
}

/*--------------------
   OPERATIONS Clause
  --------------------*/
void Calculator__get_answer(
   int *Calculator__last_res) {
   *Calculator__last_res = Calculator__answer;
}
```

```
void Calculator__addition(
   int Calculator__op1,
   int Calculator__op2,
   int *Calculator__res) {
   Calculator__add (Calculator__op1,
      Calculator__op2,
      Calculator__res);
   Calculator__get_status (&Calculator__state);
   if (Calculator__state ==
      Calculator__ok)
   {
      Calculator__answer = *Calculator__res;
   }
}
```

Chapter 4

Product Families for Abstract Machine

In Chapter 2 we have presented an algebra that deals with product families. This algebra has been used to precisely define notions around product families,e.g. features.

In chapter 3, we have discussed abstract machines. Abstract machines are used for mathematical based specification and verification of software systems. The goal of chapter thesis is to underlay the formal product family algebra with the semantic of abstract machines. Therefore we develop a semiring abstract machines first, and present a model of this algebra afterwards.

4.1 A Semiring for Abstract Machines

In this section our goal is to establish the basis for a product family algebra that is, according to Chapter 2, a commutative idempotent semiring. We will form this semiring on the power set over a direct product of monoids.

Definition 4.1.1 Let $((M_i, \cdot_i, 1_i))_{i \in I}$ be a finite family of monoids, indexed by $I = \{1, 2, \ldots, n\}$, and $(\mathbb{M}, \cdot, \mathbf{1}) = (\bigtimes_{i \in I} M_i, \cdot, (1_1, \ldots, 1_n))$ their direct product, where $\cdot$ is a componentwise operation. We define a lifted operation $\circ$:

$$\begin{aligned} &\circ : \wp(\mathbb{M}) \times \wp(\mathbb{M}) \to \wp(\mathbb{M}) \\ &\mathcal{A} \circ \mathcal{B} =_{df} \{a \cdot b \ : \ a \in \mathcal{A}, b \in \mathcal{B}\}. \end{aligned}$$

Remark 4.1.2 By the HSP-theorem, e.g. [16], $(\mathbb{M}, \cdot, \mathbf{1})$ is again a monoid. Therefore the operation $\cdot$ is totally defined and $\circ$ in turn too. □

Lemma 4.1.3 *The operation $\circ$ is associative; it is commutative if $\cdot$ is. Multiplication $\cdot$ in turn is commutative if every monoid is.*

Proof Assume $\mathcal{A}, \mathcal{B}, \mathcal{C} \in \wp(\mathbb{M})$, then we can conclude:

Commutativity:
Since the operation $\cdot$ is defined componentwise, it inherits the properties of $\cdot_i$ in the corresponding component. Therefore, if every $\cdot_i$ is commutative, $\cdot$ is obviously too. Regarding $\circ$ we assume $\cdot$ being commutative and conclude:

$$
\begin{array}{ll}
 & \mathcal{A} \circ \mathcal{B} \\
= & \quad \{\![\text{ definition of } \circ \,]\!\} \\
 & \{a \cdot b \ : \ a \in \mathcal{A}, b \in \mathcal{B}\} \\
= & \quad \{\![\text{ commutativity of } \cdot \,]\!\} \\
 & \{b \cdot a \ : \ a \in \mathcal{A}, b \in \mathcal{B}\} \\
= & \quad \{\![\text{ definition of } \circ \,]\!\} \\
 & \mathcal{B} \circ \mathcal{A}
\end{array}
$$

Associativity can be shown similarly and straight forwardly. □

Lemma 4.1.4 *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 $(\wp(\mathbb{M}), \cup, \circ, \emptyset, \mathbb{1})$ with $\mathbb{1} =_{df} \{(1_1, \ldots, 1_n)\}$ is an idempotent semiring, called* Cartesian i-semiring. *The Cartesian i-semiring is commutative if every monoid is.*

Using monoids with irreducible identity elements is important later in this chapter.

Proof According to the definition of an i-semiring, we need to check that

(1) $(\wp(\mathbb{M}), \cup, \emptyset)$ is an idempotent and commutative monoid with identity element $\emptyset$,

(2) $(\wp(\mathbb{M}), \circ, \mathbb{1})$ is a monoid with identity element $\mathbb{1}$,

(3) $\circ$ distributes over $\cup$,

(4) $\emptyset$ is an annihilator, i.e., $\forall \mathcal{A} \in \wp(\mathbb{M}) : \emptyset \circ \mathcal{A} = \emptyset = \mathcal{A} \circ \emptyset$.

Part (1) needs no further review as this structure is well known.
Lemma 4.1.3 yields the associativity of $\circ$. Now we show that $\mathbb{1}$ is an identity element, assume $\mathcal{A} \in \wp(\mathbb{M})$:

$$
\begin{array}{ll}
 & \mathbb{1} \circ \mathcal{A} \\
= & \quad \{\!\![\text{ definition of } \mathbb{1} \ ]\!\!\} \\
 & \{(1_1 \ldots, 1_n)\} \circ \mathcal{A} \\
= & \quad \{\!\![\text{ definition of } \circ \ ]\!\!\} \\
 & \{(1_1, \ldots, 1_n) \cdot (a_1, \ldots, a_n) \ : \ (a_1, \ldots, a_n) \in \mathcal{A}\} \\
= & \quad \{\!\![\text{ definition of } \cdot \ ]\!\!\} \\
 & \{(1_1 \cdot_1 a_1, \ldots, 1_n \cdot_n a_n) \ : \ (a_1, \ldots, a_n) \in \mathcal{A}\} \\
= & \quad \{\!\![\ 1_i \text{ is the identity element regarding to } \cdot_i, i = 1 \ldots n \ ]\!\!\} \\
 & \{(a_1, \ldots, a_n) \ : \ (a_1, \ldots, a_n) \in \mathcal{A}\} \\
= & \mathcal{A}
\end{array}
$$

The equation $\mathcal{A} \circ \mathbb{1} = \mathcal{A}$ can be shown similarly. Thus Part (2) holds. To show Part (3) we choose three arbitrary elements $\mathcal{A}, \mathcal{B}, \mathcal{C} \in \wp(\mathbb{M})$ and conclude:

$$
\begin{array}{ll}
 & (\mathcal{A} \cup \mathcal{B}) \circ \mathcal{C} \\
= & \quad \{\!\![\text{ definition of } \cup \ ]\!\!\} \\
 & \{x \ : \ x \in \mathcal{A} \vee x \in \mathcal{B}\} \circ \mathcal{C} \\
= & \quad \{\!\![\text{ definition of } \circ \ ]\!\!\} \\
 & \{x \cdot c \ : \ (x \in \mathcal{A} \vee x \in \mathcal{B}) \wedge c \in \mathcal{C}\} \\
= & \quad \{\!\![\text{ distributivity of } \wedge \text{ over } \vee \ ]\!\!\} \\
 & \{x \cdot c \ : \ (x \in \mathcal{A} \wedge c \in \mathcal{C}) \vee (x \in \mathcal{B} \wedge c \in \mathcal{C})\} \\
= & \quad \{\!\![\text{ definition of } \cup \ ]\!\!\} \\
 & \{x \cdot c \ : \ (x \in \mathcal{A} \wedge c \in \mathcal{C})\} \cup \{x \cdot c \ : \ (x \in \mathcal{B} \wedge c \in \mathcal{C})\} \\
= & \quad \{\!\![\text{ definition of } \circ \ ]\!\!\} \\
 & (\mathcal{A} \circ \mathcal{C}) \cup (\mathcal{B} \circ \mathcal{C})
\end{array}
$$

We can show the dual distributivity law, i.e., $\mathcal{A} \circ (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \circ \mathcal{B}) \cup (\mathcal{A} \circ \mathcal{C})$, similar to the former proof. We see the correctness of Part (4) almost directly. Assume $\mathcal{A} \in \wp(\mathbb{M})$:

$$
\begin{array}{ll}
 & \emptyset \circ \mathcal{A} \\
= & \quad \{\!\![\text{ definition of } \circ \ ]\!\!\} \\
 & \{x \cdot a \ : \ x \in \emptyset \wedge a \in \mathcal{A}\} \\
= & \quad \{\!\![\text{ definition of } \emptyset \ ]\!\!\} \\
 & \emptyset \\
= & \quad \{\!\![\text{ definition of } \emptyset \ ]\!\!\} \\
 & \{a \cdot x \ : \ a \in \mathcal{A} \wedge x \in \emptyset\} \\
= & \quad \{\!\![\text{ definition of } \circ \ ]\!\!\} \\
 & \mathcal{A} \circ \emptyset
\end{array}
$$

The commutativity of $\circ$, in the case of the commutativity of every monoid, follows directly from Lemma 4.1.3.

□

Remark 4.1.5 The case that $n = 1$, i.e., the family of monoids consists of only one monoid, is considered in Chapter 2. We will omit the indices of neutral elements 1_i and operations $\cdot_i$ of the monoids M_i for better readability, whenever possible. □

We give a short example of a Cartesian i-semiring basing on two monoids.

Example 4.1.6 Assume a programming language having a class `String` with a `string` field `str`, an `integer` field `len` to store the length of `str` and a method to concatenate another string to `str`. We choose the two monoids $INT = (\mathbb{N}, +, 0)$ and $STR = (S, , \lambda)$. Here S is the set of all finite strings, juxtaposition the common string concatenation and λ the empty string `""`. Then $(\wp(\mathbb{M}), \cup, \circ, \emptyset, \mathbb{1})$, with $\mathbb{M} = INT \times STR$, forms a non-commutative Cartesian i-semiring. Now it is possible to concatenate two strings or combine two sets of strings with each other and update their length at the same time. Assume $\mathcal{S} = \{(1,\texttt{" "})\}$, $\mathcal{A} = \{(5,\texttt{"hello"})\}$, $\mathcal{B} = \{(5,\texttt{"world"})\}$, $\mathcal{C} = \{(13,\texttt{"federal state"}), (7,\texttt{"federal"})\}$ and
$\mathcal{D} = \{(10,\texttt{"government"}), (3,\texttt{"law"})\}$. Then we can combine these elements in the following way, e.g. :
$\mathcal{A} \circ \mathcal{S} \circ \mathcal{B} = \{(6,\texttt{"hello "})\} \circ \{(5,\texttt{"world"})\} = \{(11,\texttt{"hello world"})\}$,
$\mathcal{C} \circ \mathcal{S} \circ \mathcal{D} =$
$\{(24,\texttt{"federal state government"}), (17,\texttt{"federal state law"}),$
$(11,\texttt{"federal law"}), (18,\texttt{"federal government"})\}$.

□

Often it is necessary to get certain components from tuples belonging to an element of a Cartesian i-semiring. In relational algebra this concept is well known as projection, see e.g. [15]. We give a slightly different definition of the unary operation projection as the one in relational algebra.

Definition 4.1.7 Assume a Cartesian i-semiring $(\wp(\mathbb{M}), \cup, \circ, \emptyset, \mathbb{1})$, an element $\mathcal{B} \in \wp(\mathbb{M})$ and an element $a \in \mathbb{M}$. We define the *projection* $\pi_i(a)$ by

$$\pi_i(a) = \pi_i((a_1, \ldots, a_i, \ldots, a_n)) = a_i \ . \tag{4.1}$$

In the same way we define the *projection* $\pi_i(\mathcal{B})$ by

$$\pi_i(\mathcal{B}) = \{\pi_i(b) \ : \ b \in \mathcal{B}\} \ , \tag{4.2}$$

which maps its elements (tuples) to a set of their i-th components. Correspondingly we define the inverse mapping for $b_i \in M_i$ and $\mathcal{B}$ by

$$\pi_i^{-1}(b_i, \mathcal{B}) = \{x \in \mathcal{B} \ : \ \pi_i(x) = b_i\} \ . \tag{4.3}$$

Furthermore we define a substitution of the i-th component of a by x, by

$$a[i \mapsto x] = (a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_n) \ , \tag{4.4}$$

$$\text{i.e., } \pi_j(a[i \mapsto x]) = \begin{cases} \pi_i(a), & \text{if } j \neq i \\ x, & \text{if } j = i \end{cases}$$

Example 4.1.8 For the element $\mathcal{C} \circ \mathcal{S} \circ \mathcal{D}$ from the example above the projection to its first components is $\pi_1(\mathcal{C} \circ \mathcal{S} \circ \mathcal{D}) = \{11, 17, 18, 24\}$. An example for the substitution of a component is given by
$(5, \text{"}\texttt{world}\text{"})[2 \mapsto \text{"}\texttt{Peter}\text{"}] = (5, \text{"}\texttt{Peter}\text{"})$. □

We now give some useful properties of the projections and the substitution that we will use later.

Lemma 4.1.9 *Assume a Cartesian i-semiring* $(\wp(\mathbb{M}), \cup, \circ, \emptyset, \mathbb{1})$. *For elements* $\mathcal{A}, \mathcal{B} \in \wp(\mathbb{M})$ *and elements* $a, b \in \mathbb{M}$ *the following holds:*

$$(a \cdot b)[i \mapsto x \cdot y] = a[i \mapsto x] \cdot b[i \mapsto y] \tag{4.5}$$

$$a = b \Rightarrow \pi_i(a) = \pi_i(b) \tag{4.6}$$

$$\pi_i(a \cdot b) = \pi_i(a) \cdot \pi_i(b) \tag{4.7}$$

$$\pi_i(\mathcal{A} \cup \mathcal{B}) = \pi_i(\mathcal{A}) \cup \pi_i(\mathcal{B}) \tag{4.8}$$

$$\bigcup_{a_i \in \pi_i(\mathcal{A})} \pi_i^{-1}(a_i, \mathcal{A}) = \mathcal{A} \tag{4.9}$$

Proof We only show the Equations (4.5) and (4.9). The other properties can be checked easily.
Equation (4.5) :
Assume two elements $a, b \in \mathbb{M}$, then we conclude:

$$\begin{aligned} & a[i \mapsto x] \cdot b[i \mapsto y] \\ = & \quad \{\![\text{ Definition 4.1.7, Part (4.4) }]\!\} \\ & (a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_n) \cdot (b_1, \ldots, b_{i-1}, y, b_{i+1}, \ldots, b_n) \\ = & \quad \{\![\text{ Definition of multiplication }]\!\} \\ & (a_1 \cdot b_1, \ldots, a_{i-1} \cdot b_{i-1}, x \cdot y, a_{i+1} \cdot b_{i+1}, \ldots, a_n \cdot b_n) \\ = & \quad \{\![\text{ Definition 4.1.7, Part (4.4) }]\!\} \\ & (a \cdot b)[i \mapsto x \cdot y] \end{aligned}$$

Equation (4.9) :
Assume an element $\mathcal{A} \in \wp(\mathbb{M})$, then we conclude:

$$
\begin{aligned}
& \bigcup_{a_i \in \pi_i(\mathcal{A})} \pi_i^{-1}(a_i, \mathcal{A}) \\
= \quad & \{\![\text{ Definition 4.1.7, Part (4.3) }]\!\} \\
& \bigcup_{a_i \in \pi_i(\mathcal{A})} \{x \in \mathcal{A} \ : \ \pi_i(x) = a_i\} \\
= \quad & \{\![\text{ Set theory }]\!\} \\
& \{x \in \mathcal{A} \ : \ \pi_i(x) \in \pi_i(\mathcal{A})\} \\
= \quad & \{\![\text{ Tautology }]\!\} \\
& \{x \in \mathcal{A}\} \\
= \quad & \mathcal{A}
\end{aligned}
$$

□

4.2 Products and Features

Before presenting a concrete model of a product family algebra based on a Cartesian i-semiring we can generally characterize products and features. Having a look at Definition 2.1.6, the structure of products in such a semiring is easy to see.

Corollary 4.2.1 *An element $\mathcal{A}$ of a Cartesian i-semiring is a* product *iff it is a singleton set.*

Definition 2.1.7 yields that a feature is a product with special properties. Looking at Implication (2.3) of Definition 2.1.7 we can determine the structure of a feature in a Cartesian i-semiring more precisely. In preparation for the next theorem we will first give a definition for special elements of the underlying monoids which is quite similar to the definition of a feature in an i-semiring. Since there is only one operation in monoids, a distinction between products and feature makes no sense. But since these elements are pre-stages of features, in some degree, we call them *prefeatures*.

Definition 4.2.2 Assume a monoid $(M, \cdot, 1)$. An element a is called a *prefeature* iff it is different from 1 satisfying the following laws:

$$\forall\, b : b \,|\, a \implies b = 1 \ \vee \ b = a \ , \tag{4.10}$$

$$\forall\, b, c : a \,|\, (b \cdot c) \implies (a \,|\, b \ \vee \ a \,|\, c) \ , \tag{4.11}$$

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

As done in Corollary 4.2.1 for products, we want to identify features in a Cartesian i-semiring. Therefore we need, additionally to prefeatures, a lemma dealing with the divisibility relation and the case that the involved elements are of a special structure.

Lemma 4.2.3 *Assume a Cartesian i-semiring* $S = (\wp(\mathbb{M}), \cup, \circ, \emptyset, \mathbb{1})$ *and* $\mathcal{P}, \mathcal{B}, \mathcal{C} \in S$*. Furthermore let* $j \in \{1, \ldots, n\}$ *be a fixed index and* $\mathcal{P} = \{p\}$ *a product where* $\pi_i(p) = 1\ \forall i \neq j$*,* $\pi_j(p) = p_j$ *and* p_j *a prefeature, then the following holds:*

$$\mathcal{P} \mid \mathcal{B} \iff \forall b \in \mathcal{B} : p \mid b \iff \forall b_j \in \pi_j(\mathcal{B}) : p_j \mid b_j \ , \tag{4.12}$$

$$\begin{aligned} \forall b_j \in \pi_j(\mathcal{B}), c_j \in \pi_j(\mathcal{C}) : p_j \mid (b_j \cdot c_j) \implies \\ (\forall b_j \in \pi_j(\mathcal{B}) : p_j \mid b_j \vee \\ \forall c_j \in \pi_j(\mathcal{C}) : p_j \mid c_j) \ . \end{aligned} \tag{4.13}$$

Proof Note that because of the given properties of $\mathcal{P}$, we can write $\mathcal{P}$ also as $\{\mathbf{1}[j \mapsto p_j]\}$. By Definition 4.1.1, $\mathbf{1}$ is the neutral element of $(\mathbb{M}, \cdot, \mathbf{1})$ and $\mathbb{1} = \{\mathbf{1}\}$. We will prove the Equivalences (4.12) in three steps:

(1) $\mathcal{P} \mid \mathcal{B} \implies \forall b \in \mathcal{B} : p \mid b$

(2) $\forall b \in \mathcal{B} : p \mid b \implies \forall b_j \in \pi_j(\mathcal{B}) : p_j \mid b_j$

(3) $\forall b_j \in \pi_j(\mathcal{B}) : p_j \mid b_j \implies \mathcal{P} \mid \mathcal{B}$

Step (1):

$$\begin{array}{ll} & \mathcal{P} \mid \mathcal{B} \\ \Leftrightarrow & \quad \{\!\![\text{ Definition of } \mid \ ]\!\!\} \\ & \exists \mathcal{D} : \mathcal{P} \circ \mathcal{D} = \mathcal{B} \\ \Leftrightarrow & \quad \{\!\![\ \mathcal{P} \text{ is a product and definition of } \circ \ ]\!\!\} \\ & \exists \mathcal{D} : \{p \cdot d \ : \ d \in \mathcal{D}\} = \mathcal{B} \\ \Rightarrow & \quad \{\!\![\text{ set identity }]\!\!\} \\ & \forall b \in \mathcal{B} : \exists d \in \mathcal{D} : p \cdot d = b \\ \Rightarrow & \quad \{\!\![\ \mathcal{D} \subseteq \mathbb{M} \ ]\!\!\} \\ & \forall b \in \mathcal{B} : \exists d \in \mathbb{M} : p \cdot d = b \\ \Rightarrow & \quad \{\!\![\text{ definition of } \mid \text{ in } \mathbb{M} \ ]\!\!\} \\ & \forall b \in \mathcal{B} : p \mid b \end{array}$$

Step (2):
Assume $b_j \in \pi_j(\mathcal{B})$, we have to show $p_j \mid b_j$.

$$\begin{array}{ll} & b_j \in \pi_j(\mathcal{B}) \\ \Rightarrow & \quad \{\!\![\ \forall b \in \mathcal{B} : p \mid b \text{ and } \pi_i^{-1}(b_i, \mathcal{B}) \subseteq \mathcal{B} \text{ by Equation (4.9) }]\!\!\} \\ & \forall b \in \pi_j^{-1}(b_j, \mathcal{B}) : p \mid b \\ \Rightarrow & \quad \{\!\![\text{ Definition of } \mid \text{ in } \mathbb{M} \ ]\!\!\} \\ & \forall b \in \pi_j^{-1}(b_j, \mathcal{B}) \ \exists d \in \mathbb{M} : p \cdot d = b \end{array}$$

$$
\begin{aligned}
\Rightarrow\quad & \{\![\text{ Implication (4.6) }]\!\} \\
& \forall\, b \in \pi_j^{-1}(b_j, \mathcal{B})\ \exists d \in \mathbb{M} : \pi_j(p \cdot d) = \pi_j(b) \\
\Rightarrow\quad & \{\![\text{ Equation (4.7) }]\!\} \\
& \forall\, b \in \pi_j^{-1}(b_j, \mathcal{B})\ \exists d \in \mathbb{M} : \pi_j(p) \cdot \pi_j(d) = \pi_j(b) \\
\Rightarrow\quad & \{\![\text{ Logic }]\!\} \\
& \exists d_j \in M_j : p_j \cdot d_j = b_j \\
\Rightarrow\quad & \{\![\text{ Definition of } \mid \text{ in } M_j \,]\!\} \\
& p_j \mid b_j
\end{aligned}
$$

Step (3):
By definition $\mathcal{P} \mid \mathcal{B}$ iff there is a $\mathcal{D} : \mathcal{P} \circ \mathcal{D} = \mathcal{B}$. Therefore we choose $\mathcal{D} = \bigcup_{b_j \in \pi_j(\mathcal{B})} \{b[j \mapsto d_j] \ :\ b \in \pi_j^{-1}(b_j, \mathcal{B}) \wedge p_j \cdot d_j = b_j\}$ and show that it suffices the requirement.
Note that because of the premise $\forall\, b_j \in \pi_j(\mathcal{B}) : p_j \mid b_j$ there is a $d_j \in M_j$ with $p_j \cdot d_j = b_j$. We now calculate $\mathcal{P} \circ \mathcal{D}$:

$$
\begin{aligned}
& \mathcal{P} \circ \mathcal{D} \\
= \quad & \{\![\ \mathcal{P} = \{\mathbf{1}[j \mapsto p_j]\} \text{ and definition of } \circ\ ]\!\} \\
& \bigcup_{b_j \in \pi_j(\mathcal{B})} \{\mathbf{1}[j \mapsto p_j] \cdot b[j \mapsto d_j] \ :\ b \in \pi_j^{-1}(b_j, \mathcal{B}) \wedge p_j \cdot d_j = b_j\} \\
= \quad & \{\![\text{ Equation (4.5) and } \mathbf{1} \cdot b = b\ ]\!\} \\
& \bigcup_{b_j \in \pi_j(\mathcal{B})} \{b[j \mapsto p_j \cdot d_j] \ :\ b \in \pi_j^{-1}(b_j, \mathcal{B}) \wedge p_j \cdot d_j = b_j\} \\
= \quad & \{\![\ p_j \cdot d_j = b_j \text{ and } b[j \mapsto b_j] = b\ ]\!\} \\
& \bigcup_{b_j \in \pi_j(\mathcal{B})} \{b \ :\ b \in \pi_j^{-1}(b_j, \mathcal{B}) \wedge p_j \cdot d_j = b_j\} \\
= \quad & \{\![\text{ Equation (4.9) }]\!\} \\
& \mathcal{B}
\end{aligned}
$$

Implication (4.13) can be shown by contradiction:

$$
\begin{aligned}
& \neg(\forall\, b_j \in \pi_j(\mathcal{B}) : p_j \mid b_j \ \vee\ \forall\, c_j \in \pi_j(\mathcal{C}) : p_j \mid c_j) \\
\Leftrightarrow\quad & \exists b_j \in \pi_j(\mathcal{B}) : p_j \nmid b_j \ \wedge\ \exists c_j \in \pi_j(\mathcal{C}) : p_j \nmid c_j \\
\Leftrightarrow\quad & \exists b_j \in \pi_j(\mathcal{B}), c_j \in \pi_j(\mathcal{C}) : p_j \nmid b_j \ \wedge\ p_j \nmid c_j \\
\Rightarrow\quad & \{\![\ p_j \text{ is a prefeature and negation of (4.11) }]\!\} \\
& \exists b_j \in \pi_j(\mathcal{B}), c_j \in \pi_j(\mathcal{C}) : p_j \nmid (b_j \cdot c_j) \\
\Leftrightarrow\quad & \neg(\forall\, b_j \in \pi_j(\mathcal{B}), c_j \in \pi_j(\mathcal{C}) : p_j \mid (b_j \cdot c_j))
\end{aligned}
$$

□

Theorem 4.2.4 *Assume a commutative Cartesian i-semiring S. An element $\mathcal{A} \in S$ is a feature iff $\mathcal{A} = \{\mathbf{1}[j \mapsto a_j]\}$ and a_j is a prefeature in the corresponding monoid.*

Proof

($\Rightarrow$) We will prove the first implication by contradiction. Assume $\mathcal{A}$ is a feature and one of the following conditions hold:

(a) $\mathcal{A} \neq \{\mathbf{1}[j \mapsto a_j]\}$;

(b) a_j is not a prefeature.

Since $\mathcal{A}$ is a feature, and therefore a product, it is a singleton set by Corollary 4.2.1. Thus we can write $\mathcal{A}$ as $\{a\}$.
In Case (a) there are two possible structures of $\mathcal{A}$. The first one is, that all components are 1, i.e., $\mathcal{A} = \{\mathbf{1}\} = \mathbb{1}$. But by Definition 2.1.7 $\mathbb{1}$ is not a feature. The second one is, that at least two components, assume a_k and a_l (w.l.o.g., $k < l$), of a are not equal to $\mathbf{1}$. But then we can decompose $\mathcal{A} = \{a\}$ into two elements $\mathcal{B} = \{a[l \mapsto 1]\}$ (Note that $\mathcal{B} \neq \mathbb{1}$ since $\pi_k(a[l \mapsto 1]) \neq 1$) and $\mathcal{C} = \{1[l \mapsto a_l]\}$ with $\mathcal{B}, \mathcal{C} \neq \mathbb{1}$ and $\mathcal{B}, \mathcal{C} \neq \mathcal{A}$:

$$
\begin{aligned}
& \mathcal{B} \circ \mathcal{C} \\
= \; & \{a[l \mapsto 1]\} \circ \{\mathbf{1}[l \mapsto a_l]\} \\
= \; & \quad \{\!\![\text{ definition of } \circ \;]\!\!\} \\
& \{a[l \mapsto 1] \cdot \mathbf{1}[l \mapsto a_l]\} \\
= \; & \{a[l \mapsto a_l]\} \\
= \; & \quad \{\!\![\; a[l \mapsto a_l] = \text{a} \;]\!\!\} \\
& \mathcal{A}
\end{aligned}
$$

This clearly violates Implication (2.3) from Definition 2.1.7, i.e., $\mathcal{A}$ is not a feature.

Case (b) : If a_j is not a prefeature there are elements $b_j, c_j \in M_j$ with $\pi_j(a) = b_j \cdot c_j \wedge b_j, c_j \neq 1 \wedge b_j, c_j \neq \pi_j(a)$. Hence we can again decompose $\mathcal{A} = \{a\}$ into two elements $\{a[j \mapsto b_j]\}$ and $\{\mathbf{1}[j \mapsto c_j]\}$, both unequal to $\mathbb{1}$ and unequal to $\mathcal{A}$, no matter how the other components look like:

$$
\begin{aligned}
& \{a\} \\
= \; & \{a[j \mapsto b_j] \cdot \mathbf{1}[j \mapsto c_j]\} \\
= \; & \quad \{\!\![\text{ definition of } \circ \;]\!\!\} \\
& \{a[j \mapsto b_j]\} \circ \{\mathbf{1}[j \mapsto c_j]\}
\end{aligned}
$$

Similar to Case (a) this shows that $\mathcal{A}$ is not a feature.

($\Leftarrow$) Assume $\mathcal{A} = \{\mathbf{1}[j \mapsto a_j]\}$ and a_j is a prefeature. We have to show that $\mathcal{A}$ satisfies Definition 2.1.7. $\mathcal{A}$ is different from $\mathbb{1}$ since a_j is a prefeature. To show Implication (2.3) we assume $\mathcal{B} \in S$ with $\mathcal{B}|\mathcal{A}$. Now we conclude:

$$
\begin{aligned}
& \mathcal{B}|\mathcal{A} \\
\Leftrightarrow \quad & \{\![\text{ Definition of } | \ ]\!\} \\
& \exists \mathcal{C} : \mathcal{B} \circ \mathcal{C} = \mathcal{A} \\
\Leftrightarrow \quad & \exists \mathcal{C} : \mathcal{B} \circ \mathcal{C} = \{\mathbf{1}[j \mapsto a_j]\} \\
\Leftrightarrow \quad & \exists \mathcal{C} : (\forall b \in \mathcal{B}, c \in \mathcal{C} : b \cdot c = \mathbf{1}[j \mapsto a_j]) \\
\Rightarrow \quad & \{\![\text{ all monoids } M_i \text{ have a irreducible identity }]\!\} \\
& \exists \mathcal{C} : (\forall b \in \mathcal{B}, c \in \mathcal{C} : \mathbf{1}[j \mapsto b_j] \cdot \mathbf{1}[j \mapsto c_j] = \mathbf{1}[j \mapsto a_j]) \\
\Leftrightarrow \quad & \{\![\text{ Equation (4.5) }]\!\} \\
& \exists \mathcal{C} : (\forall b \in \mathcal{B}, c \in \mathcal{C} : \mathbf{1}[j \mapsto b_j \cdot c_j] = \mathbf{1}[j \mapsto a_j] \\
\Rightarrow \quad & \{\![\ b_j \cdot c_j = a_j \text{ and } a_j \text{ is a prefeature }]\!\} \\
& \exists \mathcal{C} : (\forall b \in \mathcal{B}, c \in \mathcal{C} : (\mathbf{1}[j \mapsto b_j \cdot c_j] = \mathbf{1}[j \mapsto a_j] \\
& \qquad \wedge (b_j = 1 \wedge c_j = a_j) \vee (b_j = a_j \wedge c_j = 1)) \\
\Rightarrow \quad & \exists \mathcal{C} : (\mathcal{B} = \{\mathbf{1}[j \mapsto 1]\} = \mathbb{1} \wedge \mathcal{C} = \mathcal{A}) \\
& \qquad \vee (\mathcal{B} = \{\mathbf{1}[j \mapsto a_j]\} = \mathcal{A} \wedge \mathcal{C} = \mathbb{1})
\end{aligned}
$$

We will show Implication (2.4) with the help of Lemma 4.2.3.

$$
\begin{aligned}
& \mathcal{A} \,|\, (\mathcal{B} \circ \mathcal{C}) \\
\Rightarrow \quad & \{\![\text{ Lemma 4.2.3 (4.12) }]\!\} \\
& \forall\, (b \cdot c) \in (\mathcal{B} \circ \mathcal{C}) : a \,|\, (b \cdot c) \\
\Rightarrow \quad & \{\![\text{ definition of } \circ \ ]\!\} \\
& \forall\, b \in \mathcal{B}, c \in \mathcal{C} : a \,|\, (b \cdot c) \\
\Rightarrow \quad & \{\![\text{ Lemma 4.2.3 (4.12) }]\!\} \\
& \forall\, b_j \in \pi_j(\mathcal{B}), c_j \in \pi_j(\mathcal{C}) : a_j \,|\, (b_j \cdot c_j) \\
\Rightarrow \quad & \{\![\text{ Lemma 4.2.3 (4.13) }]\!\} \\
& \forall\, b_j \in \pi_j(\mathcal{B}) : a_j \,|\, b_j \vee \forall\, c_j \in \pi_j(\mathcal{C}) : a_j \,|\, c_j \\
\Rightarrow \quad & \{\![\text{ Lemma 4.2.3 (4.12) }]\!\} \\
& \mathcal{A} \,|\, \mathcal{B} \vee \mathcal{A} \,|\, \mathcal{C}
\end{aligned}
$$

□

Theorem 4.2.4 states that, in such semirings, features are exactly those elements, which are singleton sets with a specific element. This element is a tuple whose components are all equal to the identity element from the corresponding monoid, expect one. Furthermore, this component is a prefeature, i.e., it is irreducible.

4.3 An Algebraic Model for Abstract Machines

In this section we present an algebraic model for abstract machines based on the semiring discussed in Section 4.1.

Looking at Table 3.1, we see that abstract machines examined by us can be decomposed into nine clauses. We will describe these parts algebraically and combine them afterwards. In fact, there are eighteen clauses a machine can consist of. For example, we omit the inclusion clauses, described in Subsection 3.2.5, in the presented model. The model can easily be extended by the remaining nine clauses by adding their corresponding monoids to the direct product defined in Definition 4.3.1.

abstract machine clause	Monoid
parameter	$(S_{id}, \cup, \emptyset)$
CONSTRAINTS	$(P, \wedge, \texttt{true})$
SETS	$(S_S, \cup, \emptyset)$
CONSTANTS	$(S_{id}, \cup, \emptyset)$
PROPERTIES	$(P, \wedge, \texttt{true})$
VARIABLES	$(S_{id}, \cup, \emptyset)$
INVARIANT	$(P, \wedge, \texttt{true})$
INITIALIZATION	$(S_I, \cup, \emptyset)$
OPERATIONS	$(S_O, \cup, \emptyset)$

Table 4.1: abstract machine clauses and their appropriate monoid

Table 4.1 shows the mapping between the abstract machine clauses and their algebraic counterparts. S_{id}, P, S_S, S_I and S_O are sets of all allowed expressions for the corresponding machine clause as defined in [1].

For example, S_{id} is the power set of all possible identifiers. Identifiers are built up from `ASCII` letters, lower or upper case, numbers and the underscore character. They have to be at least two characters long and start with a letter. More precisely the language $\mathcal{L}(identifier)$ of identifiers is specified by the regular expression $identifier = (a|\ldots|z|A|\ldots|Z)^{+}\cdot(\_|a|\ldots|z|A|\ldots|Z|0|\ldots|9)^{*}$. Thus S_{id} is equal to $\wp(\mathcal{L}(id))$. In combination with the usual set union $\cup$ and the empty set $\emptyset$, S_{id} builds the monoid $(S_{id}, \cup, \emptyset)$.

By convention only upper case identifiers are allowed for sets. Therefore S_S is defined similarly to S_{id}, but without lower case letters. $(P, \wedge, \texttt{true})$ is a monoid over the set P of all possible predicates of the Abstract Machine Notation. Predicates are based on the *predicate calculus* and *set theory*, which are discussed in Section 1.3 and Section 2.1 of [1]. By Remark 4.1.2 we can combine these monoids to a new monoid.

Definition 4.3.1 The monoid $(\mathbb{AM}, \cdot, \mathbf{1})$, where $\mathbb{AM} = S_p \times P \times S_S \times S_C \times P \times S_V \times P \times S_I \times S_O$ and $\mathbf{1} = (\emptyset, \texttt{true}, \emptyset, \emptyset, \texttt{true}, \emptyset, \texttt{true}, \emptyset, \emptyset)$ is called *abstract machine monoid*. The componentwise multiplication $\cdot$ is called *abstract machine composition*.

It is now possible to write an abstract machine as a nine-tuple of its clauses.

Remark 4.3.2 Identifiers are treated as GUIDs (globally unique identifiers). This fits well with the chosen monoid of identifiers, especially the operation $\cup$, see Table 4.1. This means if we compose two or more abstract machines having common identifiers they are treated as being identical not only syntactically but also semantically. If this is not intended the identifier under consideration can be renamed, e.g. by prefixing the machine name it belongs to. Furthermore operation overloading (same name, different signature and body) is forbidden, therefore we can abbreviate operations by their names. □

We give an example of the *abstract machine composition* of two machines with common identifiers.

Example 4.3.3 Suppose the following abstract machine.

```
MACHINE
    Square

VARIABLES
    area, length

INVARIANT
    area : NATURAL & length : NATURAL
    & area = length**2

INITIALIZATION
    area := 0 || length := 0

OPERATIONS
    set_length(len) =
    PRE len : NATURAL
    THEN
        length := len || area := len**2
    END
    out <-- get_area =
       out := area
```

The machine `Square` represents a (geometrical) square. If the side of the square is updated, its (new) area is stored in the variable `area`. In the algebraic context of the monoid $(\mathbb{AM}, \cdot, \mathbf{1})$ this abstract machine is the tuple

$$\begin{aligned} \mathit{Square} = (&\emptyset, \mathtt{true}, \emptyset, \emptyset, \mathtt{true}, \{\mathtt{area}, \mathtt{length}\}, \\ &\mathtt{area} : \mathtt{NATURAL} \wedge \mathtt{length} : \mathtt{NATURAL} \\ &\wedge\ \mathtt{area} = \mathtt{length} * *2, \\ &\{\mathtt{area} := 0,\ \mathtt{length} := 0\}, \\ &\{\mathtt{set\_length}, \mathtt{get\_area}\})\ . \end{aligned}$$

Now assume another machine `Rectangle` which represents a (geometrical) rectangle.

```
MACHINE
    Rectangle

VARIABLES
    area, length, width

INVARIANT
    area : NATURAL & length : NATURAL & width : NATURAL
    & area = length * width

INITIALIZATION
    area := 0 || length := 0 || width := 0

OPERATIONS
    set_l_w(len, wid) =
    PRE len : NATURAL & wid : NATURAL
    THEN
        length := len || width := wid
        ||area := len*wid
    END;
    out <-- get_area =
            out := area
```

The corresponding element in $\mathbb{AM}$ is

$$\begin{aligned} \mathit{Rectangle} = (&\emptyset, \mathtt{true}, \emptyset, \emptyset, \mathtt{true}, \{\mathtt{area}, \mathtt{length}, \mathtt{width}\}, \\ &\mathtt{area} : \mathtt{NATURAL} \wedge \mathtt{length} : \mathtt{NATURAL} \\ &\wedge \mathtt{width} : \mathtt{NATURAL} \wedge \mathtt{area} = \mathtt{length} * \mathtt{width}, \\ &\{\mathtt{area} := 0,\ \mathtt{length} := 0,\ \mathtt{width} := 0\}, \\ &\{\mathtt{set\_l\_w}, \mathtt{get\_area}\})\ . \end{aligned}$$

We may want to combine all abstract machine dealing with geometric objects by the use of abstract machine composition. The composition of *Square* and *Rectangle* is then given by

$$\begin{aligned} \mathit{Square} \cdot \mathit{Rectangle} = (&\emptyset, \mathtt{true}, \emptyset, \emptyset, \mathtt{true}, \{\mathtt{area}, \mathtt{length}\}, \\ &\mathtt{area} : \mathtt{NATURAL} \wedge \mathtt{length} : \mathtt{NATURAL} \\ &\wedge \mathtt{area} = \mathtt{length} * *2, \\ &\{\mathtt{area} := 0,\ \mathtt{length} := 0\}, \\ &\{\mathtt{set\_length}, \mathtt{get\_area}\}) \\ &. \\ &(\emptyset, \mathtt{true}, \emptyset, \emptyset, \mathtt{true}, \{\mathtt{area}, \mathtt{length}, \mathtt{width}\}, \\ &\mathtt{area} : \mathtt{NATURAL} \wedge \mathtt{length} : \mathtt{NATURAL} \\ &\wedge \mathtt{width} : \mathtt{NATURAL} \wedge \mathtt{area} = \mathtt{length} * \mathtt{width}, \\ &\{\mathtt{area} := 0,\ \mathtt{length} := 0,\ \mathtt{width} := 0\}, \\ &\{\mathtt{set\_l\_w}, \mathtt{get\_area}\}) \\ = \\ &(\emptyset, \mathtt{true}, \emptyset, \emptyset, \mathtt{true}, \{\mathtt{area}, \mathtt{length}, \mathtt{width}\}, \\ &\mathtt{area} : \mathtt{NATURAL} \wedge \mathtt{length} : \mathtt{NATURAL} \\ &\wedge \mathtt{area} = \mathtt{length} * *2 \wedge \mathtt{width} : \mathtt{NATURAL} \\ &\wedge \mathtt{area} = \mathtt{length} * \mathtt{width}, \\ &\{\mathtt{area} := 0,\ \mathtt{length} := 0,\ \mathtt{width} := 0\}, \\ &\{\mathtt{set\_length}, \mathtt{set\_l\_w}, \mathtt{get\_area}\})\ . \end{aligned}$$

For a better overview we translate it back into an abstract machine.

```
MACHINE
    Square_Rectangle

VARIABLES
    area, length, width

INVARIANT
    area : NATURAL & length : NATURAL & width : NATURAL
    & area = length **2 & area = length * width

INITIALIZATION
    area := 0 || length := 0 || width := 0

OPERATIONS
    set_length(len) =
    PRE len : NATURAL
    THEN
        length := len || area := len**2
    END;

    set_l_w(len, wid) =
    PRE len : NATURAL & wid : NATURAL
    THEN
        length := len || width := wid
        ||area := len*wid
    END;
    out <-- get_area =
            out := area
```

The abstract machine `Square_Rectangle` is syntactically correct, type checking is not a problem and the initialization preserves the invariant. But there are some problems. First both operations do not preserve the invariant in general; they only do if `length = width` holds. Secondly, assume that we had omitted the formula(s) of the area(s) in the invariant. Then both operations do preserve the invariant. However, they modify the `area` of the square and the `area` of the rectangle. This is not the intention. Thirdly, if another machine (or the user) calls the operation `get_area` it is not possible to determine which `area` it returns.

□

We do not want to restrict the use of abstract machine composition to machines with distinct identifiers. This is since the formal correctness of the resulting machine has to be checked by proof obligations anyway. But from a semantical point of view it is always up to the one who states an abstract machine to guarantee that this machine fulfills the intended needs. In Section 4.4 we present a situation in which it is intended that several machines have common identifiers.

We can use the abstract machine composition not only to merge "full-blown" machines, but also to add a certain operation to a machine. The abstract machine monoid can be seen as a kind of a library providing every possible machine. Therefore we know that there is at least one machine offering a special modification for a given variable. We give an example.

Example 4.3.4 Assume that we want to add a possibility to scale the variable `area` of the abstract machine `Rectangle` from Example 4.3.3. This can be achieved by multiplying `Rectangle` by an appropriate machine. So we may have the following operation.

```
OPERATIONS
    scale_area(factor) =
    PRE factor : NATURAL
    THEN
        length := length*factor || width := width*factor
        ||area := area*factor*factor
    END;
```

Note that this machine is not correct in terms of proof obligations, since the variables `length`, `width` and `area` have neither been declared nor typed. In our monoid, this machine is given by the element:

$$scale\_area\_m = (\emptyset, \texttt{true}, \emptyset, \emptyset, \texttt{true}, \emptyset, \texttt{true}, \emptyset, \{\texttt{scale\_area}\}) \ .$$

Or even more compact:

$$scale\_area\_t = \mathbf{1}[9 \mapsto \{\texttt{scale\_area}\}] \ .$$

Thus the composition is given by

$$\begin{aligned} \mathit{Rectangle} \cdot \mathit{scale\_area\_t} &= \mathit{Rectangle} \cdot \mathbf{1}[9 \mapsto \{\texttt{scale\_area}\}] \\ &= (\emptyset, \texttt{true}, \emptyset, \emptyset, \texttt{true}, \{\texttt{area}, \texttt{length}, \texttt{width}\}, \\ &\quad \texttt{area} : \texttt{NATURAL} \wedge \texttt{length} : \texttt{NATURAL} \\ &\quad \wedge \texttt{width} : \texttt{NATURAL} \wedge \texttt{area} = \texttt{length} * \texttt{width}, \\ &\quad \{\texttt{area} := 0,\ \texttt{length} := 0,\ \texttt{width} := 0\}, \\ &\quad \{\texttt{set\_l\_w}, \texttt{get\_area}, \texttt{scale\_area}\}) \ . \end{aligned}$$

□

Since all monoids given in Table 4.1 are commutative and have irreducible identities, we can build a product family algebra from the monoid $(\mathbb{AM}, \cdot, \mathbf{1})$. The structure $(\wp(\mathbb{M}), \cup, \circ, \emptyset, \mathbb{1})$, with $\circ$ as in Definition 4.1.1, is a commutative i-semiring by Lemma 4.1.4. Furthermore it is a product family algebra since $\mathbb{1} = \{(\emptyset, \texttt{true}, \emptyset, \emptyset, \texttt{true}, \emptyset, \texttt{true}, \emptyset, \emptyset)\}$ is a product by Corollary 4.2.1.

Definition 4.3.5 The product family algebra $(\wp(\mathbb{AM}), \cup, \circ, \emptyset, \mathbb{1})$ is called *product family algebra for abstract machines* . Elements of $\wp(\mathbb{AM})$ are called *abstract machine families* or *families* for short. The operation $\circ$ is called *abstract machine families composition* or *family composition* for short. The operation $\cup$ is called *abstract machine families union* or *family union* for short.

Having a product family algebra for abstract machines, all algebraic laws given in Chapter 2, Section 4.1 and Section 4.2 can be applied to abstract machine families.

In the remainder, we denote elements of a family, i.e., tuples, by their corresponding machine name followed by *_t*, e.g. *Square_t*. This way we can abbreviate tuples and have a distinction for families and tuples simultaneously.

In Section 3.3 we have presented the machines `Add`, `Calculator_ctx` and `Calculator`. For the abstract machines `Sub`, `Mult` and `Div` we refer to Appendix A. Because of the limitations of the translator, we have named all identifiers of the given machines differently. From now, all identifiers are shortened by the suffix derived from their related machine. Furthermore, all inclusion clauses are omitted, since they are not available in this model.

We give an example of determining the common parts of two families.

Example 4.3.6 First of all we present a mapping between the old and new identifier names.

old identifier name	new identifier name
last_result_sub last_result_mult last_result_div	last_result
status_sub status_mult status_div	status
ans_sub ans_mult ans_div	ans
get_status_sub get_status_mult get_status_div	get_status

Table 4.2: Mapping between old and new identifier names

Compared to Example 4.3.3 we now have sets instead of tuples. Therefore the abstract machine add is the element

$$\begin{aligned} add\_prod = \Big\{ & (\emptyset, \texttt{true}, \emptyset, \emptyset, \texttt{true}, \{\texttt{last\_result}, \texttt{status}\}, \\ & \texttt{last\_result} : \texttt{INT} \wedge \texttt{status} : \texttt{STATE}, \\ & \{\texttt{last\_result} := 0, \texttt{status} := \texttt{ok}\}, \\ & \{\texttt{add}, \texttt{ans}, \texttt{get\_status}\}) \Big\} . \end{aligned}$$

The machine Sub is the family

$$\begin{aligned} sub\_prod = \Big\{ & (\emptyset, \texttt{true}, \emptyset, \emptyset, \texttt{true}, \{\texttt{last\_result}, \texttt{status}\}, \\ & \texttt{last\_result} : \texttt{INT} \wedge \texttt{status} : \texttt{STATE}, \\ & \{\texttt{last\_result} := 0, \texttt{status} := \texttt{ok}\}, \\ & \{\texttt{sub}, \texttt{ans}, \texttt{get\_status}\}) \Big\} . \end{aligned}$$

Since these families cannot be decomposed w.r.t. family union, they are *products* in our algebraic context. Therefore we suffixed "*_prod*". Whenever

possible we will abbreviate the tuples, as mentioned above. We can offer a choice between the two families by applying family union to them. We denote the resulting family by *little_calcs_fam*:

$$little\_calcs\_fam = add\_prod \ \cup \ sub\_prod = \{add\_t, sub\_t\} \ .$$

The commonalities of the two elements from the product family *little_calcs* can be determined by calculating their greatest common divisor (cf. Section 2.3). As a result, we have the rearranged family

$$\begin{aligned} little\_calcs\_fam = &\Big(\{\mathbf{1}[9 \mapsto \{\texttt{add}\}]\} \cup \{\mathbf{1}[9 \mapsto \{\texttt{sub}\}]\}\Big) \\ &\circ \Big\{(\emptyset, \texttt{true}, \emptyset, \emptyset, \texttt{true}, \{\texttt{last\_result}, \texttt{status}\}, \\ &\quad \texttt{last\_result} : \texttt{INT} \wedge \texttt{status} : \texttt{STATE}, \\ &\quad \{\texttt{last\_result} := \texttt{0}, \texttt{status} := \texttt{ok}\}, \\ &\quad \{\texttt{ans}, \texttt{get\_status}\})\Big\} \end{aligned}$$

We see that the two families have quite a lot in common. In particular, they have the same static data and the same state (cf. Subsections 3.2.2 and 3.2.3). They only differ in their `OPERATIONS` part. We will use this fact in Section 4.4. □

As stated in Theorem 4.2.4, features are exactly those elements in a Cartesian i-semiring which are irreducible with respect to $\circ$. In case of the product family algebra for abstract machines these are elements whose corresponding abstract machine consists at most of one nonempty clause. Therefore the families $\{\mathbf{1}[9 \mapsto \{\texttt{add}\}]\}$ and $\{\mathbf{1}[9 \mapsto \{\texttt{sub}\}]\}$ are *abstract machine features*, or *features* for short.

We give a short example in which we list all features of the product *add_prod*, cf. Example 4.3.6.

Example 4.3.7 The element *add_prod* is a product and not a feature. It is built up from the features

$$\begin{aligned} f\_add\_var\_last\_result &= \{\mathbf{1}[6 \mapsto \{\texttt{last\_result}\}]\}, \\ f\_add\_var\_status &= \{\mathbf{1}[6 \mapsto \{\texttt{status}\}]\}, \\ f\_add\_inv\_last\_result &= \{\mathbf{1}[7 \mapsto \{\texttt{last\_result} : \texttt{INT}\}]\}, \\ f\_add\_inv\_status &= \{\mathbf{1}[7 \mapsto \{\texttt{status} : \texttt{STATE}\}]\}, \\ f\_add\_init\_last\_result &= \{\mathbf{1}[8 \mapsto \{\texttt{last\_result} := \texttt{0}\}]\}, \end{aligned}$$

$$
\begin{aligned}
f\_add\_init\_status &= \{\mathbf{1}[8 \mapsto \{\texttt{status} := \texttt{ok}\}]\},\\
f\_add\_op\_ans &= \{\mathbf{1}[9 \mapsto \{\texttt{ans}\}]\},\\
f\_add\_op\_get\_status &= \{\mathbf{1}[9 \mapsto \{\texttt{get\_status}\}]\},\\
f\_add\_op\_add &= \{\mathbf{1}[9 \mapsto \{\texttt{add}\}]\}\ .
\end{aligned}
$$

The product *add_prod* can be completely decomposed into nine parts:

$$
\begin{aligned}
add\_prod = & f\_add\_var\_last\_result \circ f\_add\_var\_status\\
& \circ f\_add\_inv\_last\_result \circ f\_add\_inv\_status\\
& \circ f\_add\_init\_last\_result \circ f\_add\_init\_status\\
& \circ f\_add\_op\_ans \circ f\_add\_op\_get\_status \circ f\_add\_op\_add\ .
\end{aligned}
$$

□

If we add features to a family, it may happen that the result is not correct w.r.t the B-Method, cf. Example 4.3.4. We present an approach to weaken this effect in Chapter 5.

4.4 A Basic Calculator Revisited

In Section 3.3 we have developed a basic calculator using the B-Method. In this section we discuss how we can "build" this calculator with the help of the *product family algebra for abstract machines*.

The `Calculator_ctx` machine is mapped to the element *calculator_ctx_t* of the the abstract machine monoid. It is given by

$$
\begin{aligned}
&calculator\_ctx\_t =\\
&\quad (\emptyset, \texttt{true}, \{\texttt{STATE} = \{\texttt{ok}, \texttt{overflow}, \texttt{underflow}, \texttt{div\_by\_zero}\}\},\\
&\quad \{\texttt{min\_dsp\_number}, \texttt{max\_dsp\_number}\},\\
&\quad \texttt{min\_dsp\_number} : \texttt{INT} \wedge \texttt{min\_dsp\_number} < 0\\
&\quad \wedge\ \texttt{max\_dsp\_number} : \texttt{NAT}, \emptyset, \texttt{true}, \emptyset, \emptyset)\ .
\end{aligned}
$$

For the whole system we use the following products:

$$
\begin{aligned}
calculator\_ctx\_prod &= \{calculator\_ctx\_t\}\\
add\_prod &= \{add\_t\}\\
sub\_prod &= \{sub\_t\}\\
mult\_prod &= \{mult\_t\}\\
div\_prod &= \{div\_t\}\ .
\end{aligned}
$$

The products *add_prod* and *sub_prod* have been defined in Example 4.3.6. The products *mult_prod* and *div_prod* are defined similarly. Their corresponding abstract machines, w.r.t. Table 4.2, are given in Appendix A. Note that there is no family for the abstract machine `Calculator`. We give the reason for this later.

Now we combine these products using family composition. As result we get the product *calc_prod*:

$$\begin{aligned} calc\_prod &= calculator\_ctx\_prod \circ add\_prod \\ &\quad \circ sub\_prod \circ mult\_prod \circ div\_prod \\ &= \left\{ calculator\_ctx\_t \cdot add\_t \cdot sub\_t \cdot mult\_t \cdot div\_t \right\} . \end{aligned}$$

After performing the machine composition

$$\begin{aligned} calc\_prod = (&\emptyset, \\ &\texttt{true}, \\ &\{\texttt{STATE} = \{\texttt{ok}, \texttt{overflow}, \texttt{underflow}, \texttt{div\_by\_zero}\}\}, \\ &\{\texttt{min\_dsp\_number}, \texttt{max\_dsp\_number}\}, \\ &\texttt{min\_dsp\_number} : \texttt{INT} \wedge \texttt{min\_dsp\_number} < 0 \\ &\wedge \texttt{max\_dsp\_number} : \texttt{NAT}, \\ &\{\texttt{last\_result}, \texttt{status}\}, \\ &\texttt{last\_result} : \texttt{INT} \wedge \texttt{status} : \texttt{STATE}, \\ &\{\texttt{last\_result} := 0, \texttt{status} := \texttt{ok}\}, \\ &\{\texttt{Add}, \texttt{Sub}, \texttt{Mult}, \texttt{Div}\}) \ , \end{aligned}$$

we translate the expression back into an abstract machine. Since the translation is basically done by string-concatenation, it can be easily automated by a simple tool. The complete machine is given in Appendix B; here we skip most of the bodies of the operations.

```
MACHINE
    calc_prod

SETS
    STATE = { ok, overflow, underflow, div_by_zero }

CONSTANTS
    min_dsp_number, max_dsp_number
```

```
PROPERTIES
    min_dsp_number : INT
    & min_dsp_number < 0
    & max_dsp_number : NAT

VARIABLES
    last_result,
    status

INVARIANT
    last_result : INT &
    status : STATE

INITIALISATION
    last_result := 0 || status := ok

OPERATIONS
    result <-- add(op1, op2) =
    PRE
        op1 : INT & op2 : INT & (op1 + op2) : INT
    THEN
        IF
            (op1 + op2) <= max_dsp_number
            & (op1 + op2) >= min_dsp_number
        THEN
            status := ok || last_result := op1 + op2
            || result := op1 + op2
        ELSE
            status :: STATE - ok || result := 0
        END
    END;

    result <-- sub(op1, op2) = /* operation body */
    result <-- mult(op1, op2) = /* operation body */
    result <-- div(op1, op2) = /* operation body */

    last_res <-- ans =
    last_res := last_result;

    ret_status <-- get_status =
    ret_status := status
```

All proof obligations for this abstract machine can be proved without any problems. For example the operation `add` preserves the typing invariant for the variable `state`. This means, that `state` is an element of the set `STATE` before and after the "execution" of `add`.

This is an example for a situation in which common identifiers are quite useful. Each of the arithmetic machines fulfills every requirement listed in Section 3.3—except Requirement 2., of course. Therefore we can develop each machine separately and "link" them afterwards, just as we did in Section 3.3. But there is one major difference between the B-Method approach and the algebraic approach. Using the B-Method we had to introduce an additional abstract machine `Calculator` for this "linking". Whereas using the algebraic approach this kind of machine is not necessary.

Let us explain why this is necessary in the B-Method approach. Assume that the `Calculator` does not provide any operation itself. Furthermore assume a machine, which is responsible for I/O tasks, that **EXTENDS** the `Calculator`. Then all operations specified by the arithmetic machines can be called directly. Now a user may call the operation `add`. Next he/she clears the display of the calculator for some reason. Afterwards he/she wants to recall the last calculated result. Since the last called operation cannot be determined, it is ambiguous which of the operations `ans`, `ans_sub`, `ans_mult`, `ans_div` is responsible for this task. Therefore the last result cannot presented to the user. This problem can be solved in two ways. The machine `Calculator` stores either the last called operation or the last (valid) result. For both solutions a new variable has to be specified. To assign the correct values to this variable, some kind of wrapping operations are necessary. We used the second solution in Section 3.3.

For the algebraic approach the machine `Calculator` is redundant. This is since we allow the composition of machines with identical identifiers. The "linking", mentioned above, is done (automatically) by machine composition. From the machine `calc_prod` the last valid result can be directly retrieved by the operation `ans`. This works because we are not interested in the last valid result of a specific operation, but in the result of the last successful operation. This algebraic approach slenderizes the core system compared to the B-Method approach. Furthermore, all requirements have been met, especially the modularized design which allows a reuse of parts of the system. If we want to extend the core system later by new arithmetic operations, we can simply add them by using family composition. In contrast to that, the B-Method would have to modify the machine `Calculator` by adding the corresponding wrapping operations. This leads to even more duplicated code.

Furthermore the machine `Calculator` cannot be reused, since this recurring changes. In our model all elements can be used.

So far, we extended only products, but we can also extend a family which offers a choice of products using family composition. By this we can add an extension to each product of the family.

Example 4.4.1 Assume the family *calculators* give by

$$calculators = calc\_prod \cup calc\_prod\_ext \ ,$$

where *calc_prod_ext* is an extension of the product *calc_prod* given above. For example, it may have an additional operation to calculate the square root of a given number. The family *calculators* offers a choice between the basic calculator and the extended one. Furthermore the product *memory* is given by

$$\begin{aligned} memory = \Big\{ & (\emptyset, \texttt{true}, \emptyset, \emptyset, \texttt{true}, \{\texttt{memory}\}, \texttt{memory} : \texttt{INTEGER}, \\ & \{\texttt{memory} := \texttt{0}\}, \{\texttt{memory\_save}, \texttt{memory\_recall}, \\ & \texttt{memory\_add}, \texttt{memory\_clear}\}) \Big\} \ . \end{aligned}$$

This product provides the possibility to save and recall a given number. Furthermore the memorized number can be deleted or a given value can be added to it. To add this product to both products of the *calculators* family, we simply use family composition:

$$\begin{aligned} calculators \circ memory &= (calc\_prod \cup calc\_prod\_ext) \circ memory \\ &= (calc\_prod \circ memory) \ \cup \ (calc\_prod\_ext \circ memory) \ . \end{aligned}$$

□

Chapter 5

Extensions

In this chapter we present further properties of the product family algebra for abstract machines (cf. Chapter 4). Furthermore we investigate the relation between the abstract machine clause **EXTENDS** and algebraic family composition. When combining families it may happen that the result is not correct w.r.t. the B-Method. In Section 5.3 we present an approach to avoid at least syntactical incorrectness.

This chapter is not intended to cover all possible extensions of the algebraic approach for abstract machines. It is intended as a proof of feasibility; we are convinced that the algebraic approach will yield further improvements. However, for doing so one has to model/research things like Requirements and the clause **EXTENDS** in much more detail. This is left for further research.

5.1 Requirements

In Section 2.5 we have presented the so called *requirement relation*. This relation makes it possible to express requirements for product families. For example, we can postulate that each product of a product family must have a certain feature or that two features are mutually exclusive. To express the mandatory presence of a family $\mathcal{A}$ in a family $\mathcal{B}$ we can state

$$\mathbb{1} \xrightarrow{\mathcal{B}} \mathcal{A}$$

in the algebra for abstract machines.

Requirements occur naturally during the development of a software system. For example these requirements can be feature requests, safety require-

ments or technical requirements. We have seen such requirements in Section 3.3 and 4.4, where we have developed a calculator core system.

To show how the requirement relation can be used, we give an example.

Example 5.1.1 For a better overview we repeat the requirements for the calculator core system.

1. The system has to offer integer arithmetic. Now overflow can occur.
2. Basic operations are: addition, subtraction, multiplication and division.
3. Since the display can show at most eight symbols, the operations have to indicate unrepresentable results.
4. Modular structure of the system to allow a possible reuse.
5. The system has to provide a possibility to recall the last valid result.

We have identified the following machines the system consists of:

- The arithmetic machines `Add`, `sub`, `Mult` and `Div`
- The `Calculator_ctx` to provide contextual informations

The machines have been mapped to the products add_prod, sub_prod, $mult\_prod, div\_prod$ and $calculator\_ctx\_prod$. With the requirement relation we can state that these products are mandatory for the product $calc\_prod$:

$$\begin{aligned}
\mathbb{1} &\xrightarrow{calc_prod} add_prod\ , \\
\mathbb{1} &\xrightarrow{calc_prod} sub_prod\ , \\
\mathbb{1} &\xrightarrow{calc_prod} mult_prod\ , \\
\mathbb{1} &\xrightarrow{calc_prod} div_prod\ , \\
\mathbb{1} &\xrightarrow{calc_prod} calculator_ctx_prod\ .
\end{aligned}$$

□

Requirements like the ones of Example 5.1.1 help to guarantee that all required components have been included into a system. This can be particularly useful in larger systems.

To express mutual exclusion of a family $\mathcal{A}$ and a family $\mathcal{B}$ in a family $\mathcal{C}$ we can state

$$(\mathcal{A} \circ \mathcal{B}) \xrightarrow{\mathcal{C}} \emptyset$$

in our algebra.

Example 5.1.2 Assume that we have a family *crypto* which offers a choice between various cryptographic products. Most of these products have a feature $f\_var\_key = \{\mathbf{1}[6 \mapsto \{\mathtt{key}\}]\}$. The variable *key* represents a secret key used for encryption tasks. Needless to say, it should not be possible to retrieve this key from outside. By convention the names of inquiry operations are formed by *get_* followed by the variable name. The inquiry operation for the variable *key* is then *get_key* and the corresponding feature *f_ops_get_key*. Since we want to prevent a disclosure of the *key*, we state :

$$(f\_var\_key \circ f\_ops\_get\_key) \xrightarrow{crypto} \emptyset \ .$$

□

5.2 EXTENDS and Machine Composition

In this section we investigate if there is a relation between the **EXTENDS** clause (cf. Subsection 3.2.5) and the machine composition (cf. Definition 4.3.1). Let us explain why we compare these two mechanisms.

Any attempt to express an inclusion clause, different from **EXTENDS**, by machine composition is foredoomed to fail. This is since the resulting machine of a composition effectively consists of the objects from the composed machines. For example, each operation stemming from one of the composed machines, can be called from outside. The only inclusion clause that offers this, is **EXTENDS**.

Therefore we compare the clause **EXTENDS** with abstract machine composition in this section. The main difference between these concepts are visibility rules regarding objects of the involved machines. An overview of these visibility rules is given in Appendix D of [1]. We give a short repetition of the properties of **EXTENDS**.

The **EXTENDS** clause is a combination of the **INCLUDES** clause and a **PROMOTES** clause that lists every operation of the included machine. All identifiers of the involved machines have to be distinct. Furthermore, the following visibility rules between objects/identifiers of the included machine and clauses of the including machine apply.

- Parameters are invisible.
- Constants and sets are visible in every clause, but read-only.
- Variables are visible in the **INVARIANT** clause and read-only in the **INITIALISATION** and **OPERATIONS** clauses.

- Operations are visible in the clause **OPERATIONS**.

In our algebraic model for abstract machines we have explicitly allowed the composition of machines with identical identifiers. We have discussed the reasons in Section 4.3. Therefore it is obvious that the extension of a machine by another one is not equivalent in general to the composition of corresponding elements in our algebraic model. Even if we consider distinct identifiers, different visibility rules apply for a composed machine and an extending machine. This is because the objects of the composed machines become genuine objects of the resulting machine. Whereas the objects of an extended machine do not become a "physical" part of the extending. The visibility rules for a machine w.r.t. to its own objects are:

- Parameters are visible in each clause, except in the **PROPERTIES** clause.
- Constants and sets are visible in every clause, except in the **CONSTRAINTS** clause.
- Variables are visible in the **INVARIANT**, **INITIALISATION** and **OPERATIONS** clause.
- Operations are invisible.

For example, the visibility regarding operations differs. Operations of an included machine can be called within an operation of the including machine, whereas an operation of the same machine cannot be called.

However, if certain conditions hold, the resulting machines of the two mechanisms can be "used equivalently", e.g. by another machine. By "used equivalently" we mean, that both machines provide the same facility, e.g. operations.

Conjecture 5.2.1 Let `M1`, `M2`, `M3` be machines and $m1\_t, m2\_t, m3\_t$ the corresponding elements in the abstract machine monoid. Furthermore, assume that `M1` (exclusively) lists `M2` in its **EXTENDS** clause and `M3` is the machine which results from a back translation of $m3\_t = m1\_t \cdot m2\_t$. Then `M1` and `M3` can be "used equivalently", if the following conditions hold:

1. `M1` and `M2` are are correct w.r.t. the B-Method.
2. No parameters are given in `M2`.
3. No operation of `M1` calls an operation of `M2`.

□

Note that the back translation of $m1\_t$ is not equal to `M1`, since $m1\_t$ has no component to "store" names of extended machines.

Condition 1. guarantees, among other things, that all identifiers of `M1` and `M2` are distinct and that `M2` does not use any identifier of `M1`. The latter is possible in our algebra, but not in the B-Method. The former is mandatory for the use of the **EXTENDS** clause. The reason for Condition 2. is, that parameters of an extended machine have to be instantiated in the extending machine. In `M3` an instantiation of (its own) parameters is not possible. Condition 3. guarantees that in `M3` no operation call of an own operation occurs, which is forbidden.

We give an example that suggests Conjecture 5.2.1.

Example 5.2.2 Assume the machine `M_1`.

```
MACHINE
    M1

INCLUDES
    M2

OPERATIONS
    out <--incr(in)=
      PRE in : INTEGER THEN out := in + 1 END

END
```

Assume that machine `M1 EXTENDS` a machine `M2` given by

```
MACHINE
    M2

OPERATIONS
    out <--decr(in)=
      PRE in : INTEGER THEN out := in - 1 END

END
```

Let the back translation of the composition of the related monoid elements be the machine `M1_M2` given by

```
MACHINE
    M1_M2

OPERATIONS
    out <--incr(in)=
      PRE in : INTEGER THEN out := in + 1 END

    out <--decr(in)=
      PRE in : INTEGER THEN out := in - 1 END

END
```

Then M1 and M1_M2 provide the same operations to the outside. □

5.3 Proper Machines

Families of the product family algebra for abstract machines are built up from a set of features. Since these features correspond to an abstract machine that consists of just a single clause, a feature most likely not forms a valid machine w.r.t B. The B-Method states several requirements that a machine has to meet to be valid. One of these is that a machine has to fulfill dependency requirements w.r.t. abstract machine clauses. In this section we map these requirements to our algebra.

Each of the clauses we presented in Section 3.2 is optional in principle. However, some of them require the presence of another one. The dependency requirements are the following ones:

1. **Parameters** require a **CONSTRAINTS** clause and vice-versa.
2. a **CONSTANTS** clause requires a **PROPERTIES** clause and vice-versa.
3. a **VARIABLES** clause requires an **INVARIANT** clause and vice-versa.
4. a **VARIABLES** clause requires an **INITIALIZATION** clause and vice-versa.
5. a **VARIABLES** clause requires an **OPERATIONS** clause. Not vice-versa.

We have adopted this listing in a slightly expanded form from [1]. When using family composition it happens that these requirements are not fulfilled.

Example 5.3.1 Assume the two features of Example 4.3.7:

$$\begin{aligned} f\_add\_var\_last\_result &= \{\mathbf{1}[6 \mapsto \{\texttt{last\_result}\}]\} \\ f\_add\_op\_get\_status &= \{\mathbf{1}[9 \mapsto \{\texttt{get\_status}\}]\} \ . \end{aligned}$$

The composition of them yields

$$\begin{aligned} comp = f\_add\_var\_last\_result \circ f\_add\_op\_get\_status = \\ \Big\{(\emptyset, \texttt{true}, \emptyset, \emptyset, \texttt{true}, \{\texttt{last\_result}\}, \\ \texttt{true}, \emptyset, \{\texttt{get\_status}\})\Big\} \ . \end{aligned}$$

The result *comp* does not meet the dependences regarding variables. For example, although *comp* has the variable *last_result*, it does not have an invariant for this particular variable. □

Of course, there are further requirements regarding the correctness of an abstract machine, e.g. each variable has to be typed in the **INVARIANT** clause. In this section the goal is only to map the dependences from the listing above. This can be seen as pre-stage to the mapping of other correctness requirements.

To give an idea of how we can achieved that an element of the abstract machine monoid fulfills the dependency requirements, we give an short example.

Example 5.3.2 Assume the machine `M2` from Example 5.2.2. The element $m2 \in \mathbb{AM}$ that is related to `M2` is given by

$$m2 = \mathbf{1}[9 \mapsto \{\texttt{decr}\}]$$

□

We see from the example above, that the **OPERATIONS** clause yields the non-empty set $\{\texttt{decr}\}$ in the ninth component of $m2$. We will take advantage of this fact, since this applies to every clause of an abstract machine. To enforce that a family $\mathcal{A}$ fulfills the dependences given above, we state an appropriate predicate for every element of this family. These predicates have the form

$$\pi_1(a) \neq 1 \Leftrightarrow \pi_2(a) \neq 1 \ ,$$

where a is an element of the considered family. For example, the above predicate maps the first dependency from the above listing. Just a reminder: 1 is the identity of the related monoid and we omit the index, since it is clear from the projection which identity is meant.

Definition 5.3.3 A family $\mathcal{A}$ fulfills the dependency requirements iff the following holds:

$$\begin{aligned}\forall a \in \mathcal{A} : \ & \pi_1(a) \neq 1 \Leftrightarrow \pi_2(a) \neq 1 \\ & \wedge \pi_4(a) \neq 1 \Leftrightarrow \pi_5(a) \neq 1 \\ & \wedge \pi_6(a) \neq 1 \Leftrightarrow \pi_7(a) \neq 1 \\ & \wedge \pi_6(a) \neq 1 \Leftrightarrow \pi_8(a) \neq 1 \\ & \wedge \pi_6(a) \neq 1 \Rightarrow \pi_9(a) \neq 1\end{aligned}$$

Chapter 6

Conclusion and Outlook

In this thesis we have equipped the formal product family algebra with the semantics of abstract machines. This approach combines the advantages of both techniques. Abstract machines are used for the specification and the verification of software systems. They are the main concept of the formal B-Method. Product family algebra focuses on FOSD. Problems like finding common features or the expression of dependences between families are major tasks. This algebra bases on the well-known concept of semirings. By the use of a product family algebra for abstract machines, software specification with abstract machines gains the advantages of this algebra.

In this thesis we have presented product family algebra first and discussed how goals of FOSD can be achieved using this mathematical approach. Then we have discussed abstract machines in detail and presented a development of a small calculator system using the B-Method. The main work has been done in the investigation of an appropriate algebraic structure for abstract machines. It turns out that the structure which fits best is a monoid for abstract machines. Since the elements of this monoid are tuples, an abstract machine can be directly mapped to an element of this monoid, and vice-versa. Each component of an element represents one of the abstract machine clauses. To compose these elements a componentwise operation has been defined. With respect to B, this operation basically joins the machine clauses. To deal with families of algebraically represented abstract machines, we have defined a product family algebra for abstract machines. This algebra is based on a Cartesian i-semiring. We precisely stated the structure of products and features in this semiring. Afterwards we revisited the calculator system, showed how it could be developed with product families and discussed the differences to the B-Method approach.

At the end of the thesis we pointed at some possibilities future work can yield. We have sketched how requirements can be modeled. Using the

requirement relation one can for example guarantee that a family does not lack certain features.

Moreover we described conditions that have to be met in order that family composition is equivalent to extending a machine. Finally we gave a predicate that maps the dependences which hold between machine clauses to the presented algebra.

All these topics need certainly further research. 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. From an algebraic point of view update has already been modeled, e.g. [7, 11]. However updating yields often non-commutative operations, e.g. simple overriding. Since commutativity of multiplication is one of the basic concepts in product family algebra, one might change the underlying algebraic structure. In case of abstract machines this change might be avoidable. Since abstract machine operations are based on generalized substitution it is conceivable to use parallel substitution or bounded choice substitution for an update. These substitutions preserve the commutativity of product family algebra. Furthermore it is of interest to investigate how to integrate the requirements that the B-Method states for a correct abstract machine in the algebra for abstract machines.

Acknowledgments I am deeply grateful to P. Höfner for his superior supervision. He has the patience of a saint.

Appendix A

Deferred Files for the Calculator from Section 3.3

Appendix A and Appendix B give deferred abstract machines and generated source code for the developed calculator systems.

A.1 Abstract Machines and their Implementations

In this section we present the deferred abstract machines from Section 3.3 where we developed a calculator system using B-Method.

```
MACHINE
    Sub

SEES
    Calculator_ctx

VARIABLES
    last_result_sub,
    status_sub

INVARIANT
    last_result_sub : INT &
    status_sub : STATE

INITIALISATION
    last_result_sub := 0 || status_sub := ok

OPERATIONS
    result <-- sub(op1, op2) =
    PRE
        op1 : INT & op2 : INT & (op1 - op2) : INT
    THEN
        IF
            (op1 - op2) <= max_dsp_number
```

```
            & (op1 - op2) >= min_dsp_number
        THEN
            status_sub := ok || last_result_sub := op1 - op2
            || result := op1 - op2
        ELSE
            status_sub :: STATE - ok || result := 0
        END
    END;

    last_res <-- ans_sub =
            last_res := last_result_sub;

    ret_status <-- get_status_sub =
            ret_status := status_sub
END
```

```
IMPLEMENTATION
    Sub_i

REFINES
    Sub

SEES
    Calculator_ctx

CONCRETE_VARIABLES
    last_result_sub, status_sub

INITIALISATION
    status_sub := ok;
    last_result_sub := 0

OPERATIONS
    last_res <-- ans_sub =
    BEGIN
        last_res := last_result_sub
    END;

    ret_status <-- get_status_sub =
    BEGIN
        ret_status := status_sub
    END;

    result <-- sub(op1 , op2) =
    BEGIN
        VAR temp IN
            temp := op1 - op2;
            IF temp <= max_dsp_number
                & temp >= min_dsp_number
            THEN
                status_sub := ok;
                result := temp;
                last_result_sub := result
            ELSE
                IF
                    temp < min_dsp_number
                THEN
                    result := 0;
                    status_sub := underflow
```

```
                ELSE
                    result := 0;
                    status_sub := overflow
                END
            END
        END
    END
END
```

```
MACHINE
    Mult

SEES
    Calculator_ctx

VARIABLES
    last_result_mult,
    status_mult

INVARIANT
    last_result_mult : INT &
    status_mult : STATE

INITIALISATION
    last_result_mult := 0 || status_mult := ok

OPERATIONS
    result <-- mult(op1, op2) =
    PRE
        op1 : INT & op2 : INT & (op1 * op2) : INT
    THEN
        IF
            (op1 * op2) <= max_dsp_number
            & (op1 * op2) >= min_dsp_number
        THEN
            status_mult := ok
            ||result := op1*op2
            ||last_result_mult := op1*op2
        ELSE
            status_mult :: STATE - ok || result := 0
        END
    END;

    last_res <-- ans_mult =
            last_res := last_result_mult;

    ret_status <-- get_status_mult =
            ret_status := status_mult

END
```

```
IMPLEMENTATION
    Mult_i

REFINES
```

```
    Mult

SEES
    Calculator_ctx

CONCRETE_VARIABLES
    last_result_mult, status_mult

INITIALISATION
    status_mult := ok;
    last_result_mult := 0

OPERATIONS
    last_res <-- ans_mult =
    BEGIN
        last_res := last_result_mult
    END;

    ret_status <-- get_status_mult =
    BEGIN
        ret_status := status_mult
    END;

    result <-- mult(op1 , op2) =
    BEGIN
        VAR temp IN
            temp := op1 * op2;
            IF temp <= max_dsp_number
                & temp >= min_dsp_number
            THEN
                status_mult := ok;
                result := temp;
                last_result_mult := result
            ELSE
                IF
                    temp < min_dsp_number
                THEN
                    result := 0;
                    status_mult := underflow
                ELSE
                    result := 0;
                    status_mult := overflow
                END
            END
        END
    END
END
```

```
MACHINE
    Div

SEES
    Calculator_ctx

VARIABLES
    last_result_div,
    status_div

INVARIANT
```

```
    last_result_div : INT &
    status_div : STATE

INITIALISATION
    last_result_div := 0 || status_div := ok

OPERATIONS
    result <-- div(op1, op2) =
    PRE
        op1 : INT & op2 : INT & (op1 / op2) : INT
    THEN
        IF
            op2 = 0
        THEN
            status_div := div_by_zero || result := 0
        ELSE
            status_div := ok || result:= op1/op2
            ||last_result_div := op1/op2
        END
    END;

    last_res <-- ans_div =
            last_res := last_result_div;

    ret_status <-- get_status_div =
            ret_status := status_div
END
```

```
IMPLEMENTATION
    Div_i
REFINES
    Div

SEES
    Calculator_ctx

CONCRETE_VARIABLES
    last_result_div,
    status_div

INITIALISATION
    last_result_div := 0;
    status_div := ok

OPERATIONS
    last_res <-- ans_div =
    BEGIN
        last_res := last_result_div
    END;

    ret_status <-- get_status_div =
    BEGIN
        ret_status := status_div
    END;

    result <-- div ( op1 , op2 ) =
    BEGIN
        IF op2 = 0
        THEN
```

```
            status_div := div_by_zero;
            result := 0
        ELSE
                        status_div := ok;
            result := op1/op2;
            last_result_div := result
        END
    END
END
```

A.2 Generated Source Files

In this section we present the generated source files for the calculator system we developed in Section 3.3 using B-Method.

```
/*************************************************
 File Name               : Calculator.c
*************************************************/
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
#include "Calculator.h"

/*-----------------
   IMPORTS Clause
  -----------------*/
#include "Calculator_ctx.h"

/*-----------------
   EXTENDS Clause
  -----------------*/
#include "Add.h"
#include "Div.h"
#include "Mult.h"
#include "Sub.h"

/*---------------------------
   CONCRETE_VARIABLES Clause
  ---------------------------*/
int Calculator__answer;
Calculator__STATE Calculator__state;

/*-------------------------
   INITIALISATION Clause
  -------------------------*/
void Calculator__INITIALISATION(void) {
   Calculator__answer = 0;
   Calculator__state = Calculator__ok;
}

/*--------------------
   OPERATIONS Clause
  --------------------*/
void Calculator__get_answer(
```

```
   int *Calculator__last_res) {
   *Calculator__last_res = Calculator__answer;
}

void Calculator__addition(
   int Calculator__op1,
   int Calculator__op2,
   int *Calculator__res) {
   Calculator__add (Calculator__op1,
      Calculator__op2,
      Calculator__res);
   Calculator__get_status (&Calculator__state);
   if (Calculator__state ==
      Calculator__ok)
   {
      Calculator__answer = *Calculator__res;
   }
}

void Calculator__subtraction(
   int Calculator__op1,
   int Calculator__op2,
   int *Calculator__res) {
   Calculator__sub (Calculator__op1,
      Calculator__op2,
      Calculator__res);
   Calculator__get_status_sub (&Calculator__state);
   if (Calculator__state ==
      Calculator__ok)
   {
      Calculator__answer = *Calculator__res;
   }
}

void Calculator__multiplication(
   int Calculator__op1,
   int Calculator__op2,
   int *Calculator__res) {
   Calculator__mult (Calculator__op1,
      Calculator__op2,
      Calculator__res);
   Calculator__get_status_mult (&Calculator__state);
   if (Calculator__state ==
      Calculator__ok)
   {
      Calculator__answer = *Calculator__res;
   }
}

void Calculator__division(
   int Calculator__op1,
   int Calculator__op2,
   int *Calculator__res) {
   Calculator__div (Calculator__op1,
      Calculator__op2,
      Calculator__res);
   Calculator__get_status_div (&Calculator__state);
   if (Calculator__state ==
      Calculator__ok)
   {
      Calculator__answer = *Calculator__res;
   }
```

```
}
```

```
/*************************************************
 File Name               : Calculator.h
*************************************************/
#ifndef _Calculator_h
#define _Calculator_h
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
/*-----------------
   IMPORTS Clause
  -----------------*/
#include "Calculator_ctx.h"

/*-----------------
   EXTENDS Clause
  -----------------*/
#include "Add.h"
#include "Div.h"
#include "Mult.h"
#include "Sub.h"

/*-------------------------------
   SETS Clause: enumerated sets
  -------------------------------*/
#define Calculator__STATE Calculator_ctx__STATE
#define Calculator_ctx__ok Calculator_ctx__ok
#define Calculator_ctx__overflow Calculator_ctx__overflow
#define Calculator_ctx__underflow Calculator_ctx__underflow
#define Calculator_ctx__div_by_zero Calculator_ctx__div_by_zero


/*-----------------------------------------------
   CONCRETE_CONSTANTS Clause: scalars and arrays
  -----------------------------------------------*/
#define Calculator__max_dsp_number Calculator_ctx__max_dsp_number
#define Calculator__min_dsp_number Calculator_ctx__min_dsp_number

/*----------------------------
   CONCRETE_VARIABLES Clause
  ----------------------------*/
extern int Calculator__answer;
extern Calculator__STATE Calculator__state;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
extern void Calculator__INITIALISATION(void);

/*-------------------------------
   PROMOTES and EXTENDS Clauses
  -------------------------------*/
#define Calculator__add Add__add
#define Calculator__ans Add__ans
#define Calculator__ans_div Div__ans_div
#define Calculator__ans_mult Mult__ans_mult
#define Calculator__ans_sub Sub__ans_sub
#define Calculator__div Div__div
```

```
#define Calculator__get_status Add__get_status
#define Calculator__get_status_div Div__get_status_div
#define Calculator__get_status_mult Mult__get_status_mult
#define Calculator__get_status_sub Sub__get_status_sub
#define Calculator__mult Mult__mult
#define Calculator__sub Sub__sub

/*--------------------
   OPERATIONS Clause
  --------------------*/
extern void Calculator__addition(
   int Calculator__op1,
   int Calculator__op2,
   int *Calculator__res);
extern void Calculator__division(
   int Calculator__op1,
   int Calculator__op2,
   int *Calculator__res);
extern void Calculator__get_answer(
   int *Calculator__last_res);
extern void Calculator__multiplication(
   int Calculator__op1,
   int Calculator__op2,
   int *Calculator__res);
extern void Calculator__subtraction(
   int Calculator__op1,
   int Calculator__op2,
   int *Calculator__res);

#endif
```

```
/*************************************************
 File Name               : Sub.h
*************************************************/
#ifndef _Sub_h
#define _Sub_h
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
/*--------------
   SEES Clause
  --------------*/
#include "Calculator_ctx.h"

/*---------------------------
   CONCRETE_VARIABLES Clause
  ---------------------------*/
extern int Sub__last_result_sub;
extern Calculator_ctx__STATE Sub__status_sub;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
extern void Sub__INITIALISATION(void);

/*--------------------
   OPERATIONS Clause
  --------------------*/
extern void Sub__ans_sub(
```

```
   int *Sub__last_res);
extern void Sub__get_status_sub(
   Calculator_ctx__STATE *Sub__ret_status);
extern void Sub__sub(
   int Sub__op1,
   int Sub__op2,
   int *Sub__result);

#endif
```

```
/*************************************************
 File Name               : Sub.c
*************************************************/
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
#include "Sub.h"

/*--------------
   SEES Clause
  --------------*/
#include "Calculator_ctx.h"

/*---------------------------
   CONCRETE_VARIABLES Clause
  ---------------------------*/
int Sub__last_result_sub;
Calculator_ctx__STATE Sub__status_sub;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
void Sub__INITIALISATION(void) {
   Sub__status_sub = Calculator_ctx__ok;
   Sub__last_result_sub = 0;
}

/*--------------------
   OPERATIONS Clause
  --------------------*/
void Sub__ans_sub(
   int *Sub__last_res) {
   *Sub__last_res = Sub__last_result_sub;
}

void Sub__get_status_sub(
   Calculator_ctx__STATE *Sub__ret_status) {
   *Sub__ret_status = Sub__status_sub;
}

void Sub__sub(
   int Sub__op1,
   int Sub__op2,
   int *Sub__result) {
   {
      int Sub__temp;

      Sub__temp = Sub__op1 -
         Sub__op2;
```

```
      if ((Sub__temp <=
         Calculator_ctx__max_dsp_number) &&
         (Sub__temp >=
         Calculator_ctx__min_dsp_number))
      {
         Sub__status_sub = Calculator_ctx__ok;
         *Sub__result = Sub__temp;
         Sub__last_result_sub = *Sub__result;
      }
      else {
         if (Sub__temp <
            Calculator_ctx__min_dsp_number)
         {
            *Sub__result = 0;
            Sub__status_sub = Calculator_ctx__underflow;
         }
         else {
            *Sub__result = 0;
            Sub__status_sub = Calculator_ctx__overflow;
         }
      }
   }
}
```

```
/************************************************
 File Name               : Mult.h
************************************************/
#ifndef _Mult_h
#define _Mult_h
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
/*--------------
   SEES Clause
  --------------*/
#include "Calculator_ctx.h"

/*----------------------------
   CONCRETE_VARIABLES Clause
  ----------------------------*/
extern int Mult__last_result_mult;
extern Calculator_ctx__STATE Mult__status_mult;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
extern void Mult__INITIALISATION(void);

/*--------------------
   OPERATIONS Clause
  --------------------*/
extern void Mult__ans_mult(
   int *Mult__last_res);
extern void Mult__get_status_mult(
   Calculator_ctx__STATE *Mult__ret_status);
extern void Mult__mult(
   int Mult__op1,
   int Mult__op2,
   int *Mult__result);
```

```
#endif
```

```
/*************************************************
 File Name               : Mult.c
*************************************************/
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
#include "Mult.h"

/*--------------
   SEES Clause
  --------------*/
#include "Calculator_ctx.h"

/*----------------------------
   CONCRETE_VARIABLES Clause
  ----------------------------*/
int Mult__last_result_mult;
Calculator_ctx__STATE Mult__status_mult;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
void Mult__INITIALISATION(void) {
   Mult__status_mult = Calculator_ctx__ok;
   Mult__last_result_mult = 0;
}

/*--------------------
   OPERATIONS Clause
  --------------------*/
void Mult__ans_mult(
   int *Mult__last_res) {
   *Mult__last_res = Mult__last_result_mult;
}

void Mult__get_status_mult(
   Calculator_ctx__STATE *Mult__ret_status) {
   *Mult__ret_status = Mult__status_mult;
}

void Mult__mult(
   int Mult__op1,
   int Mult__op2,
   int *Mult__result) {
   {
      int Mult__temp;

      Mult__temp = Mult__op1 *
         Mult__op2;
      if ((Mult__temp <=
         Calculator_ctx__max_dsp_number) &&
         (Mult__temp >=
         Calculator_ctx__min_dsp_number))
      {
         Mult__status_mult = Calculator_ctx__ok;
         *Mult__result = Mult__temp;
```

```
         Mult__last_result_mult = *Mult__result;
      }
      else {
         if (Mult__temp <
            Calculator_ctx__min_dsp_number)
         {
            *Mult__result = 0;
            Mult__status_mult = Calculator_ctx__underflow;
         }
         else {
            *Mult__result = 0;
            Mult__status_mult = Calculator_ctx__overflow;
         }
      }
   }
}
```

```
/***********************************************
 File Name               : Div.h
***********************************************/
#ifndef _Div_h
#define _Div_h
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
/*--------------
   SEES Clause
  --------------*/
#include "Calculator_ctx.h"

/*----------------------------
   CONCRETE_VARIABLES Clause
  ----------------------------*/
extern int Div__last_result_div;
extern Calculator_ctx__STATE Div__status_div;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
extern void Div__INITIALISATION(void);

/*--------------------
   OPERATIONS Clause
  --------------------*/
extern void Div__ans_div(
   int *Div__last_res);
extern void Div__div(
   int Div__op1,
   int Div__op2,
   int *Div__result);
extern void Div__get_status_div(
   Calculator_ctx__STATE *Div__ret_status);

#endif
```

```
/***********************************************
 File Name               : Div.c
```

```
*********************************************/
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
#include "Div.h"

/*--------------
   SEES Clause
  --------------*/
#include "Calculator_ctx.h"

/*---------------------------
   CONCRETE_VARIABLES Clause
  ---------------------------*/
int Div__last_result_div;
Calculator_ctx__STATE Div__status_div;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
void Div__INITIALISATION(void) {
   Div__last_result_div = 0;
   Div__status_div = Calculator_ctx__ok;
}

/*--------------------
   OPERATIONS Clause
  --------------------*/
void Div__ans_div(
   int *Div__last_res) {
   *Div__last_res = Div__last_result_div;
}

void Div__get_status_div(
   Calculator_ctx__STATE *Div__ret_status) {
   *Div__ret_status = Div__status_div;
}

void Div__div(
   int Div__op1,
   int Div__op2,
   int *Div__result) {
   if (Div__op2 ==
      0)
   {
      Div__status_div = Calculator_ctx__div_by_zero;
      *Div__result = 0;
   }
   else {
      Div__status_div = Calculator_ctx__ok;
      *Div__result = Div__op1 /
         Div__op2;
      Div__last_result_div = *Div__result;
   }
}
```

Appendix B

Deferred Files for the Calculator from Section 4.4

B.1 Abstract Machines and their Implementations

In this section we present the deferred abstract machines from Section 4.4 where we developed a calculator system using product family algebra for abstract machines.

```
MACHINE
    calc_prod

SETS
    STATE = ok, overflow, underflow, div_by_zero

CONSTANTS
    min_dsp_number, max_dsp_number

PROPERTIES
    min_dsp_number : INT
    & min_dsp_number < 0
    & max_dsp_number : NAT

VARIABLES
    last_result,
    status

INVARIANT
    last_result : INT &
    status : STATE

INITIALISATION
    last_result := 0 || status := ok

OPERATIONS
    result <-- add(op1, op2) =
    PRE
```

```
    op1 : INT & op2 : INT & (op1 + op2) : INT
THEN
    IF
        (op1 + op2) <= max_dsp_number
        & (op1 + op2) >= min_dsp_number
    THEN
        status := ok || last_result := op1 + op2
        || result := op1 + op2
    ELSE
        status :: STATE - ok || result := 0
    END
END;

result <-- sub(op1, op2) =
PRE
    op1 : INT & op2 : INT & (op1 - op2) : INT
THEN
    IF
        (op1 - op2) <= max_dsp_number
        & (op1 - op2) >= min_dsp_number
    THEN
        status := ok || last_result := op1 - op2
        || result := op1 - op2
    ELSE
        status :: STATE - ok || result := 0
    END
END;

result <-- mult(op1, op2) =
PRE
    op1 : INT & op2 : INT & (op1 * op2) : INT
THEN
    IF
        (op1 * op2) <= max_dsp_number
        & (op1 * op2) >= min_dsp_number
    THEN
        status := ok
        ||result := op1*op2
        ||last_result := op1*op2
    ELSE
        status :: STATE - ok || result := 0
    END
END;

result <-- div(op1, op2) =
PRE
    op1 : INT & op2 : INT & (op1 / op2) : INT
THEN
    IF
        op2 = 0
    THEN
        status := div_by_zero
        || result := 0
    ELSE
        status := ok
        ||result:= op1/op2
        ||last_result := op1/op2

    END
END;

last_res <-- ans =
```

```
    last_res := last_result;

    ret_status <-- get_status =
    ret_status := status
END
```

```
IMPLEMENTATION
    calc_prod_i
REFINES
    calc_prod

VALUES
    max_dsp_number = 99999999;
    min_dsp_number = -9999999

CONCRETE_VARIABLES
    last_result ,
    status

INITIALISATION
    last_result := 0;
    status := ok

OPERATIONS
    last_res <-- ans =
    BEGIN
        last_res := last_result
    END;

    ret_status <-- get_status =
    BEGIN
        ret_status := status
    END;

    result <-- add(op1 , op2) =
    BEGIN
        VAR temp IN
            temp := op1 + op2;
            IF temp <= max_dsp_number
                & temp >= min_dsp_number
            THEN
                status := ok ;
                result := temp ;
                last_result := temp
            ELSE
                IF
                    temp < min_dsp_number
                THEN
                    result := 0;
                    status := underflow
                ELSE
                    result := 0;
                    status := overflow
                END
            END
        END
    END;

    result <-- sub(op1 , op2) =
```

```
    BEGIN
        VAR temp IN
            temp := op1 - op2;
            IF temp <= max_dsp_number
                & temp >= min_dsp_number
            THEN
                status := ok;
                result := temp;
                last_result := result
            ELSE
                IF
                    temp < min_dsp_number
                THEN
                    result := 0;
                    status := underflow
                ELSE
                    result := 0;
                    status := overflow
                END
            END
        END
    END;

    result <-- mult(op1 , op2) =
    BEGIN
        VAR temp IN
            temp := op1 * op2;
            IF temp <= max_dsp_number
                & temp >= min_dsp_number
            THEN
                status := ok;
                result := temp;
                last_result := result
            ELSE
                IF
                    temp < min_dsp_number
                THEN
                    result := 0;
                    status := underflow
                ELSE
                    result := 0;
                    status := overflow
                END
            END
        END
    END;

    result <-- div ( op1 , op2 ) =
    BEGIN
        IF op2 = 0
        THEN
            status := div_by_zero;
            result := 0
        ELSE
            status := ok;
            result := op1/op2;
            last_result := result
        END
    END
END
```

B.2 Generated Source Files

In this section we present the generated source files for the calculator system we developed in Section 4.4 using product family algebra for abstract machines.

```
/*************************************************
 File Name               : calc_prod.h
*************************************************/
#ifndef _calc_prod_h
#define _calc_prod_h
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
/*------------------------------
   SETS Clause: enumerated sets
  ------------------------------*/
typedef enum {
   calc_prod__ok,
   calc_prod__overflow,
   calc_prod__underflow,
   calc_prod__div_by_zero
} calc_prod__STATE;


/*-----------------------------------------------
   CONCRETE_CONSTANTS Clause: scalars and arrays
  -----------------------------------------------*/
const int calc_prod__max_dsp_number = 99999999;
const int calc_prod__min_dsp_number = - 9999999;

/*---------------------------
   CONCRETE_VARIABLES Clause
  ---------------------------*/
extern int calc_prod__last_result;
extern calc_prod__STATE calc_prod__status;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
extern void calc_prod__INITIALISATION(void);

/*--------------------
   OPERATIONS Clause
  --------------------*/
extern void calc_prod__add(
   int calc_prod__op1,
   int calc_prod__op2,
   int *calc_prod__result);
extern void calc_prod__ans(
   int *calc_prod__last_res);
extern void calc_prod__div(
   int calc_prod__op1,
```

```
   int calc_prod__op2,
   int *calc_prod__result);
extern void calc_prod__get_status(
   calc_prod__STATE *calc_prod__ret_status);
extern void calc_prod__mult(
   int calc_prod__op1,
   int calc_prod__op2,
   int *calc_prod__result);
extern void calc_prod__sub(
   int calc_prod__op1,
   int calc_prod__op2,
   int *calc_prod__result);

#endif
```

```
/*************************************************
 File Name               : calc_prod.c
*************************************************/
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
#include "calc_prod.h"

/*---------------------------
   CONCRETE_VARIABLES Clause
  ---------------------------*/
int calc_prod__last_result;
calc_prod__STATE calc_prod__status;

/*-------------------------
   INITIALISATION Clause
  -------------------------*/
void calc_prod__INITIALISATION(void) {
   calc_prod__last_result = 0;
   calc_prod__status = calc_prod__ok;
}

/*--------------------
   OPERATIONS Clause
  --------------------*/
void calc_prod__ans(
   int *calc_prod__last_res) {
   *calc_prod__last_res = calc_prod__last_result;
}

void calc_prod__get_status(
   calc_prod__STATE *calc_prod__ret_status) {
   *calc_prod__ret_status = calc_prod__status;
}

void calc_prod__add(
   int calc_prod__op1,
   int calc_prod__op2,
   int *calc_prod__result) {
   {
      int calc_prod__temp;

      calc_prod__temp = calc_prod__op1 +
         calc_prod__op2;
```

```
      if ((calc_prod__temp <=
         calc_prod__max_dsp_number) &&
         (calc_prod__temp >=
         calc_prod__min_dsp_number))
      {
         calc_prod__status = calc_prod__ok;
         *calc_prod__result = calc_prod__temp;
         calc_prod__last_result = calc_prod__temp;
      }
      else {
         if (calc_prod__temp <
            calc_prod__min_dsp_number)
         {
            *calc_prod__result = 0;
            calc_prod__status = calc_prod__underflow;
         }
         else {
            *calc_prod__result = 0;
            calc_prod__status = calc_prod__overflow;
         }
      }
   }
}

void calc_prod__sub(
   int calc_prod__op1,
   int calc_prod__op2,
   int *calc_prod__result) {
   {
      int calc_prod__temp;

      calc_prod__temp = calc_prod__op1 -
         calc_prod__op2;
      if ((calc_prod__temp <=
         calc_prod__max_dsp_number) &&
         (calc_prod__temp >=
         calc_prod__min_dsp_number))
      {
         calc_prod__status = calc_prod__ok;
         *calc_prod__result = calc_prod__temp;
         calc_prod__last_result = *calc_prod__result;
      }
      else {
         if (calc_prod__temp <
            calc_prod__min_dsp_number)
         {
            *calc_prod__result = 0;
            calc_prod__status = calc_prod__underflow;
         }
         else {
            *calc_prod__result = 0;
            calc_prod__status = calc_prod__overflow;
         }
      }
   }
}

void calc_prod__mult(
   int calc_prod__op1,
   int calc_prod__op2,
   int *calc_prod__result) {
   {
```

```
      int calc_prod__temp;

      calc_prod__temp = calc_prod__op1 *
         calc_prod__op2;
      if ((calc_prod__temp <=
         calc_prod__max_dsp_number) &&
         (calc_prod__temp >=
         calc_prod__min_dsp_number))
      {
         calc_prod__status = calc_prod__ok;
         *calc_prod__result = calc_prod__temp;
         calc_prod__last_result = *calc_prod__result;
      }
      else {
         if (calc_prod__temp <
            calc_prod__min_dsp_number)
         {
            *calc_prod__result = 0;
            calc_prod__status = calc_prod__underflow;
         }
         else {
            *calc_prod__result = 0;
            calc_prod__status = calc_prod__overflow;
         }
      }
   }
}

void calc_prod__div(
   int calc_prod__op1,
   int calc_prod__op2,
   int *calc_prod__result) {
   if (calc_prod__op2 ==
      0)
   {
      calc_prod__status = calc_prod__div_by_zero;
      *calc_prod__result = 0;
   }
   else {
      calc_prod__status = calc_prod__ok;
      *calc_prod__result = calc_prod__op1 /
         calc_prod__op2;
      calc_prod__last_result = *calc_prod__result;
   }
}
```

Bibliography

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July 2009.

[3] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[4] P. Höfner, R. Khedri, and B. Möller. Feature algebra. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2006.

[5] P. Höfner, R. Khedri, and B. Möller. Feature algebra. Technical Report 2006-04, Institute of Computer Science, University of Augsburg, 2006.

[6] P. Höfner, R. Khedri, and B. Möller. An algebra of product families. *Software and Systems Modeling*, 2009.

[7] P. Höfner and B. Möller. An extension of feature algebra. *Science of Computer Programming*, 2010. (submitted).

[8] P. Höfner and G. Struth. Non-termination in idempotent semirings. In R. Berghammer, B. Möller, and G. Struth, editors, *Relations and Kleene Algebra in Computer Science*, volume 4988 of *Lecture Notes in Computer Science*, pages 206–220. Springer Berlin / Heidelberg, 2008.

[9] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon Software Engineering Institute, Carnegie Mellon University, 1990.

[10] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.

[11] B. Möller. Towards pointer algebra. *Science of Computer Programming*, 21(1):57–90, 1993.

[12] S. Motré and C. Téri. Using B Method to Formalize the Java Card Runtime Security Policy for a Common Criteria Evaluation. *Proc. of 23rd National Information Systems Security Conference (NISSC 2000), Baltimore, USA, October 16-19, 2000.*, 2000.

[13] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, 1972.

[14] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE2(1):1–9, 1976.

[15] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[16] W. Wechler. *Universal Algebra for Computer Scientists.* Springer, 1992.