Int J Software Informatics, Vol.v, No.n, month 2010, pp. 1–31 E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org

Supplementing Product Families with Behaviour

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

[1] (Institut für Informatik, Universität Augsburg, Germany,
{hoefner,moeller}@informatik.uni-augsburg.de)

[2] (Department of Computing and Software, McMaster University, Hamilton, Canada,
khedri@mcmaster.ca)

It is our pleasure to dedicate this paper to Manfred Broy at the occasion of his 60th birthday. With it, we are trying to touch on a number of Manfred's wide-spread interests. The main theme of the paper is software engineering, in particular its formal foundations, something Manfred has been working on more and more intensively for the last years. The particular topic is a contribution to a formalisation of product lines, and Manfred has been active in that area, too. The tool we are using is a specification language based on guarded commands; this relates to quite early work by Manfred and others on the formal semantics of the so-called general correctness notion for non-deterministic programs. It allows the construction of an algebra of product families, and program algebra also is one of Manfred's many interests. With this sort of round trip through specification, semantics and algebra we hope to illustrate, but also complement his comprehensive and excellent work. So, best wishes, Manfred, for many further successful years!

Abstract A common approach to dealing with software requirements volatility is to define product families instead of single products. In earlier papers we have developed an algebra of such families that, roughly, consists in a more abstract view of FODA-like and/or trees of features. A product family is represented by an algebraic term over the feature names that can be manipulated using equational laws such as associativity or distributivity.

Initially, only "syntactic" models of the algebra were considered, giving more or less just the names of the features used in the various products of a family and certain interrelations such as mandatory occurrence and implication between or mutual exclusion of features, without attaching any kind of "meaning" to the features. While this is interesting and useful for determining the variety and number of possible members of such a family, it is wholly insufficient when it comes to talking about the correctness of families in a semantic sense.

In the present paper we define a class of "semantic" models of the general abstract product family algebra that allow treating very relevant additional questions. In these models, the features of a family are requirements scenarios formalised as pairs of relational specifications of a proposed system and its environment.

However, the paper is just intended as a proof of feasibility; we are convinced that the approach can also be employed for different semantic models such as general denotational or stream-based semantics.

‡ Corresponding author: Bernhard Möller, Email: moeller@informatik.uni-augsburg.de

Key words: Formal model driven software development, software family, product family algebra, functional requirements, requirements scenarios

P Höfner, R Khedri, B Möller. Supplementing Product Families with Behaviour. *Int J Software Informatics*, 2010, v(n): 1–31. **http://www.ijsi.org/xxx**

1 Introduction

Software developers are pressed to produce, in a relatively short period of time, many variations of a software product that exhibit high system qualities (such as reliability, availability, and maintainability). Moreover, they need to handle volatility in the requirements of these variations, while they have to struggle to be ahead of the competition in an ever changing market. Two main techniques for dealing with these challenges have been proposed. The first deals with the focus of attention in software development processes while the second relates to the methods employed along the development process.

- The first technique proposes that instead of focusing our attention onto a single software system to be built, one takes predictable changes into account. This amounts to the analysis and design of a family of software systems, called a *software product line*, that share a core part (commonality among all the members). Software product line engineering, which is a family-oriented software production process and method, seems to be adopted by both practitioners and researchers to deal with changes in the requirements and thereby revisions of the corresponding designs. The idea behind product line engineering is to take advantage of the commonality of systems that are developed for a specific domain.
  Faulk [Fau01] points out that much of the research related to product line processes and techniques has focused on the development stages that go from the architectural design to the coding and has dealt essentially with enhancing the reuse of software artefacts or paradigms related to these stages. However, one should expect that family-based software development should start at the earliest stage of the adopted software development process. A software development system tailored for a non-monolithic software development process should take into account the modern reality of software production: expected and unexpected changes in the requirements (both functional and non-functional) are unavoidable and must consequently be reflected and accommodated in software development processes and techniques. Only a few studies have combined the software family approach with requirements analysis [MYC05].
  The limitation of the family-based approach to software development is captured in one of its underlying assumptions, namely the *oracle hypothesis* from [WL99, page 11]: "It is possible to predict the changes that are likely to be needed to a system over its lifetime". The rapid change in the user needs and in market trends makes it hard to consider this hypothesis as tenable. Hence, another technique is required to deal with unpredictable changes.
  In [Bro06b], Broy highlights the main challenges that the automotive software industry faces. He points to the importance of dependences between different functions of a car. In particular, he shows several kinds of feature interactions. He also stresses that one of the biggest problems in automotive industry is the lack of

more appropriate requirements engineering and that modelling and understanding the requirements lie in the centre of software challenges. We believe that these problems are not limited to automotive software. They are challenges in nearly all industries such as Mobile Phone Industry or banking.

- The second technique proposed in the literature for dealing with, among others, changes and the volatility of some aspects of the users requirements is *Model Driven Engineering* (MDE), which is a general approach to the automation of model processing. By the above discussion, this technique has to work in absence of the oracle assumption. The MDE approach consists in systematic transition from a set of *initial models*, that constitute the starting points for the MDE process of a software system, to its executable code. However, the current techniques for this transition approach lack formality. To allow trust in the obtained code, the transformations need to be based on a well-defined syntax and semantics grounded in established mathematical theories (e.g., languages, set theory, algebras, etc.). Bézivim et al. [BBJ07] indicate that since 2001 model driven software development has taken different forms. However, they all share the same principle: for each domain of application a *meta-model* (or *abstract model*) is constructed, to which then all models used within that domain (the so-called *derived models*) must "conform" [BBJ07]. The initial family models are the result of requirements engineering processes. In other terms, they are the result of elicitation and formalisation activities. These activities need to be performed in a systematic and rigourous manner, but not necessary formally. Once one reaches formal models, then formal transformations should be adopted when possible. One can envisage transformations of abstract models to models that carry more details, or models of views of potential functional architectures of the system. The derived models give the specifications of both the system and the environment in which it is supposed to operate. Thus, it helps in presenting exactly what the system is expected to do in reaction to stimuli from its environment.

Despite several decades of research on developing techniques and methodologies for specifying and verifying software-intensive systems, we are still faced with many challenges in this area. In [Bro06a], Broy writes: "Developing a methodology for specifying and verifying software-intensive systems poses a grand challenge that a broad stream of research must address".

The results presented in this paper set up a mathematical framework to combine the software family approach with model driven software development. The aim is to tackle building and maintaining systems that consist of many parts or are performing diverse functionalities that are continuously changing and constantly being maintained.

We present a transformation of a software family requirements model into detailed requirements models of its members. This transformation is based on *product family algebra* and *relation algebra*. We give the mathematical foundation for this transformation system.

In earlier papers [HKM06a, HKM09] we have developed an algebra of product families that, roughly, consists in a more abstract view of FODA-like and/or trees of features. A product family is represented by an algebraic term over the feature names that can be manipulated using equational laws such as associativity or distributivity.

Initially, only "syntactic" models of the algebra were considered, giving more or less just the names of the features used in the various products of a family and certain interrelations such as mandatory occurrence and implication between or mutual exclusion of features, without attaching any kind of "meaning" to the features that has the form of descriptions, specifications, or models. While this is interesting and useful for determining the variety and number of possible members of such a family, it is wholly insufficient when it comes to talking about the correctness of families in a semantic sense.

In these models, the features of a family are requirements scenarios formalised as pairs of relational specifications of a proposed system and its environment.

However, the paper is just intended as a proof of feasibility; we are convinced that the approach can also be employed for different semantic models such as general denotational or stream-based semantics.

The paper is structured as follows: In Section 2 the underlying concepts and theory are recapitulated. In particular, we give the definition of product family algebra as well as a small example. This example illustrates also what is meant by a system's behaviour and its environment. After that, we formalise a command language for scenarios in Section 3. Its semantics is based on a transition relation that describes the connection from starting states to their possible successor states. Based on that we derive a product family algebra for formal scenarios in Section 4. In Section 5 the theory is underpinned by an illustrative example. Moreover, further applications of our approach are briefly mentioned. The paper concludes with a discussion concerning related work (Section 6) and future work (Section 7).

2 Background

2.1 A Brief Review of Program Family Algebra

To specify a software family, we use the language of a product family algebra which an idempotent and commutative semiring.

Definition 2.1. (e.g. [HW98])

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

In the context of product family specification, $+$ can be interpreted as a choice between options of products and $\cdot$ as their composition or mandatory presence. This motivates the following definition.

Definition 2.2. An idempotent commutative semiring is called a *product family al-*

gebra [HKM09]. Its elements are termed *product families* and can be considered as abstractly representing sets of products each of which is composed via $\cdot$ from a number of features.

Example 2.3. We describe a family of simple banking services: a bank has several software products that differ by the options they provide to a customer for opening a new account directly at a branch, via a web page or by e-mail. The latter two activities add some functionality to the basic opening activity at a branch. Moreover, there are some further standard activities involved in account opening that are subsumed by `restOfCoreBnkgSystem`. We may then specify our family of services by the following algebraic expression:

$$\begin{aligned}\texttt{BankingFamily} = {} & \texttt{openAccAtBranch} \\ & \cdot\ (1 + \texttt{openAccountOnline} + \texttt{openAccountByMail}) \\ & \cdot\ \texttt{restOfCoreBnkgSystem}\end{aligned}$$

By commutativity of the $\cdot$ operator this term is equal to

$$\begin{aligned}\texttt{BankingFamily} = {} & \texttt{openAccAtBranch} \cdot \texttt{restOfCoreBnkgSystem} \\ & \cdot\ (1 + \texttt{openAccountOnline} + \texttt{openAccountByMail});\end{aligned}$$

Hence commonality of the family is described by the subterm

$$\texttt{openAccAtBranch} \cdot \texttt{restOfCoreBnkgSystem}$$

while its variability is given by

$$1 + \texttt{openAccountOnline} + \texttt{openAccountByMail}$$

which adds to the commonality either nothing (summand 1) or `openAccountOnline` or `openAccountByMail`. The variability states that either `openAccountOnline` or `openAccountByMail` is possible, but not both. If one wants not only both features in conjunction ($\texttt{openAccountOnline} \cdot \texttt{openAccountByMail}$) but also optionality of each, the expression has to be rewritten into

$$\begin{aligned}& 1 + \texttt{openAccountOnline} + \texttt{openAccountByMail} + \\ & \texttt{openAccountOnline} \cdot \texttt{openAccountByMail}\end{aligned}$$

By distributivity this equals

$$(1 + \texttt{openAccountOnline}) \cdot (1 + \texttt{openAccountByMail})\ .$$

□

These algebraic expressions are closely related to FODA-like and/or trees (see [HKM09]). More precisely, they relate to *feature diagrams* of Feature-Oriented Domain Analysis (FODA) [KCH$^+$.90]. These diagrams capture the commonalities and mandatory features as well as the optional ones of a feature algebra. The leaf nodes contain the basic features of the describes product family. In the domain dictionary each basic feature is specified.

We exemplify this correspondence for our example. We assume that there are constants, such as `openAccountOnline` or `openAccountByMail` for every basic feature.

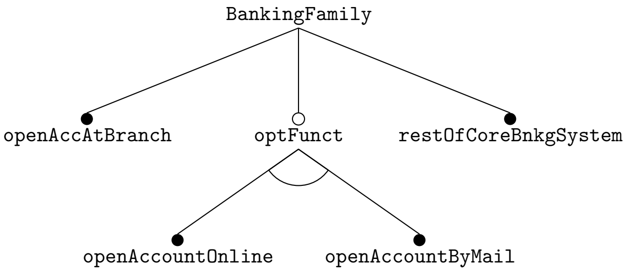


Fig. 1. Feature diagram for `BankingFamily`

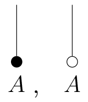
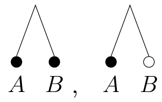
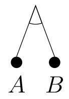
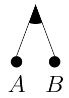
Base construct (feature diagram)	Description	Algebraic counterpart
	mandatory and optional feature	A and $(1 + A)$, resp.
 etc.	multiple features	$A \cdot B$, $A \cdot (1 + B)$, etc.
	alternative features	$A + B$
	or-group	$A + B + A \cdot B$

Table 1. FODA feature diagrams and their corresponding algebraic terms

Example 2.4. Figure 1 shows a possible feature diagram for the product family `BankingFamily` introduced in Example 2.3. We can only give possible diagrams since feature diagrams are not unique and there are several and-or trees corresponding to one single algebraic expression. □

The translation rules for the basic parts of an arbitrary and/or tree into an algebraic term are given in Table 1.

Using these rules every feature diagram can be transformed into an algebraic expression using a bottom-up traversal. This recursive method translates each subtree into an algebraic expression, starting from the leaf nodes going up to the root. When the basic constants are not interpreted, the result is unique up to commutativity and associativity of the semiring operators.

In sum, these expressions could be read purely syntactically as stating what basic features are involved in the services and how the overall services are composed from them. Still, the expressions can be transformed using laws of product family algebra, like associativity, commutativity or distributivity. However, it is much more important to attach meaning to the feature identifiers so that certain properties of the specified service family can be proved. This is what we will do in the next section.

2.2 A Command Model of Requirements Specifications

As our sample for a semantic model of product family algebra we use the idea of *formal scenarios* as defined in [DFK$^+$98]. In that approach, an informal scenario is first translated into an imperative notation (for which we will give a relational semantics in the next section). The result is split into two parts: one describes the expected behaviour of the system according to the scenario and the other the behaviour of its environment. Hence, a formal scenario is a pair (C_e, C_s) of commands that describe the possible actions of environment and system, respectively. The operation of the whole system then essentially consists in a finite or infinite repetition of the nondeterministic choice between C_e and C_s.

Example 2.5. Let us exemplify this again with our banking service family. Here is an informal specification of the program unit `openAccAtBranch`.

The customer shows up at a branch of the bank and requests to open an account. The bank through its representative at the branch analyses the conditions for opening an account. If the customer is eligible for that, the bank representative asks for one of her identification documents. The representative enters into the system the customer's identification number and the type of identification document used. If the customer is an existing customer, the system displays the remaining needed information and proposes a personalised account privileges. Otherwise, the system displays that the customer is a new customer, asks for her full name and address, and assigns to the account the standard banking privileges. If the customer accepts the privileges and pays the standard account opening fees, then the system issues a card that allows the customer to access her newly created account.

As shown in [DFK$^+$98, DKM05], the above informal scenario gives a partial description of the behaviour of the system as well as of its environment. Following these articles, in this paper we adopt the approach that these two behaviours are described by two separate relations $\mathtt{openAccAtBranch}_s$ and $\mathtt{openAccAtBranch}_e$, respectively, and that the set of states from which the environment is able to make an action is disjoint with that from which the system is able to make an action. Since a scenario is supposed to describe the environment-system interactions, it should contain only a description of the actions that originate in the domain of the function of the environment (resp. system) and terminate in a state in the domain of the function of the system (resp. environment). Therefore, the above condition indicates that, according to a scenario, at each state of the space exclusively either the environment or of the system can make an action, which puts a clear separation between the system and its environment.

Scenarios might not prescribe an action at each observable state of the system's state space. In this case we say that the scenario is *space incomplete*. When a requirements scenario is not space complete, the scenario is silent on what needs to be performed at some states of its space. Scenarios are inherently partial descriptions and therefore it is seldom that they are space complete. □

The command `openAccAtBranch` describes the behaviour of both the system and its environment perceived as forming together a closed system. Hence, at each state a choice is made between commands from $\texttt{openAccAtBranch}_s$ or $\texttt{openAccAtBranch}_e$.

As the notation for the concrete description of such relations we use a slight variation of Dijkstra's guarded command [Dij76] (see Lemma 3.7 for the relation with the original version) in the form

$$B_1 \rightarrow C_1 \;[\!]\; \cdots \;[\!]\; B_n \rightarrow C_n$$

where the B_i are predicates specifying the preconditions for execution of the commands C_i and $[\!]$ denotes non-deterministic choice. The semantics is that an arbitrary S_i for which B_i is true is executed. If none of the B_i is true, the execution of the command fails.

Example 2.6. We now give a part of the specifications of $\texttt{openAccAtBranch}_s$ and $\texttt{openAccAtBranch}_e$; their full specifications as well as that of the whole scenario can be found in Appendix A. In the code, `fld` stands for "field".

Two clauses of the behaviour of the banking system as given by the above scenario are the following:

$$
\begin{array}{l}
\quad\texttt{openAccAtBranch}_s\\
=_{df}\\
\qquad\Big(\texttt{cstmerEligOpnAcc} \;\wedge\; \texttt{fldIdNum} = \texttt{idNum}\\
\qquad\quad\wedge\; \texttt{fldIdType} = \texttt{idDocType} \;\wedge\; \neg\texttt{newCstmer}\\
\qquad\quad\wedge\; \neg\texttt{acctCreated} \;\wedge\; \neg\texttt{prvlgesAccepted}\\
\qquad\quad\wedge\; \neg\texttt{feesPayed} \;\wedge\; \neg\texttt{crdIssued}\\
\qquad\quad\longrightarrow \texttt{fldCsrmerName} := \texttt{getCstmerName(idNum)}\\
\qquad\quad\;\;;\texttt{fldCstmerAddress} := \texttt{getCstmerAddress(idNum)}\Big)\\
\qquad[\!]\;\Big(\texttt{cstmerEligOpnAcc} \;\wedge\; \texttt{fldIdNum} = \texttt{idNum}\\
\qquad\quad\wedge\; \texttt{fldIdType} = \texttt{idDocType} \;\wedge\; \neg\texttt{newCstmer}\\
\qquad\quad\wedge\; \texttt{fldCsrmerName} = \texttt{getCstmerName(idNum)}\\
\qquad\quad\wedge\; \texttt{fldCstmerAddress} = \texttt{getCstmerAddress(idNum)}\\
\qquad\quad\longrightarrow \texttt{acctprivileges} := \texttt{personalized}\;;\\
\qquad\qquad\quad \texttt{outputMssge} := \texttt{msgeAccptPrvlges?}\Big)\\
\qquad[\!]\ldots
\end{array}
$$

The first case describes the situation when a customer is eligible (`cstmerEligOpnAcc`) and she had specified an id (`idNum`) by some type of document (`idDocType`). Moreover the system's information also includes that the customer is already known (she is not a new person) and some more information (e.g., that the customer has not yet paid her fees). If these conditions are satisfied, the system determines the name and the address of the customer. The second case is read in a similar way. Here, the customer has to specify her name and her address.

The following clauses partially specify the users' behaviour or the system's environment. They are similar to the above scenario.

$$
\begin{array}{l}
\quad \mathtt{openAccAtBranch}_e \\
=_{df} \\
\qquad \Big(\mathtt{cstmerEligOpnAcc} \ \wedge\ \neg(\mathtt{fldIdNum} = \mathtt{idNum}) \\
\qquad\quad \wedge\ \neg(\mathtt{fldIdType} = \mathtt{idDocType})\ \wedge\ \neg\mathtt{acctCreated} \\
\qquad\quad \wedge\ \neg\mathtt{prvlgesAccepted}\ \wedge\ \neg\mathtt{feesPayed}\ \wedge\ \neg\mathtt{crdIssued} \\
\qquad\quad \longrightarrow \mathtt{fldIdNum}\ :=\ \mathtt{idNum}; \mathtt{fldIdType}\ :=\ \mathtt{idDocType}\Big) \\
\qquad [\!]\ \Big(\mathtt{cstmerEligOpnAcc}\ \wedge\ \mathtt{fldIdNum} = \mathtt{idNum} \\
\qquad\quad \wedge\ \mathtt{fldIdType} = \mathtt{idDocType}\ \wedge\ \mathtt{newCstmer} \\
\qquad\quad \wedge\ \neg\mathtt{acctCreated}\ \wedge\ \neg\mathtt{prvlgesAccepted} \\
\qquad\quad \wedge\ \neg\mathtt{feesPayed}\ \wedge\ \neg\mathtt{crdIssued} \\
\qquad\quad \longrightarrow \mathtt{fldCsrmerName}\ :=\ \mathtt{csrmerName}\ ; \\
\qquad\qquad\quad \mathtt{fldCstmerAddress}\ :=\ \mathtt{csrmerAddress}\Big) \\
\qquad [\!] \ldots
\end{array}
$$

□

2.3 Formal Scenarios

The command of the system part described by a scenario combined with those of the rest of the gathered scenarios provide the specification of the system to be constructed. Usually, we do not construct the environment of the system. One might ask why we then keep the command of the environment. In [KB04], we show that the specification of the environment enables us to test the system in order to assess whether it behaves as prescribed in its intended environment. The description of the environment specifies the behaviour that ought to trigger reactions from the system. In other terms the command of the environment is the specification of the behaviour of the tester of the system; the tester executes the command specifying the environment of the system as described by the scenarios of its requirements. In summary, we need the command of the system to build the system and the command of the environment to assess its behaviour (system acceptance testing). Hence, both parts are needed because they play different roles in the life cycle of a system.

For example, the above scenario is $(\mathtt{openAccAtBranch}_e, \mathtt{openAccAtBranch}_s)$, defined over a state space $\Sigma_{\mathtt{openAccAtBranch}}$. Variables in $\Sigma_{\mathtt{openAccAtBranch}}$ are for example `cstmerEligOpnAcc`, `idNum`, `newCstmer` or `crdIssued`. We cast these phenomena into a general definition.

Definition 2.7. A *(formal) scenario* over a state space Σ is a pair (C_e, C_s), where C_e and C_s are two domain-disjoint commands on Σ, called the *command of the environment* and the *command of the system*, respectively.

In a later section we will use scenarios to formalise product families and to attach meaning (semantics) to them.

3 Relational Semantics of Commands

3.1 Basic Commands and Feasibility

We now turn to the formalisation of our command language. Basically, a command defines a transition relation from starting states to their possible successor states.

However, as is well known, this purely relational view is not adequate if commands have the possibility of aborting.

If from a given starting state s there is the possibility of reaching some successor state t, the transition relation will contain the pair (s, t). But if additionally from s there is the possibility of aborting this is "ignored" by the transition relation, since it already has the "positive" information (s, t) about s.

There are various remedies to this situation. One, taken in Z (e.g. [Spi88]), is to add a pseudo-value $\perp$ to the state space that stands for abortion and to use relations over that extended state space. Another solution is the demonic relational semantics of [DBS$^+$95, DMN97] that models a total correctness view: if a state s has the possibility of leading to abortion it is considered as unsafe and no "proper" transitions (s, t) are included into the transition relation either.

There is a third variant which we will use in this paper because of its pleasant algebraic properties. This is the general correctness semantics as defined in various forms in [BGW79, Par83, BZ86, Mor87, Mor88, Bac89, Nel89, Doo94]. The idea is to model commands as pairs consisting of a transition relation and a set of states from which no abortion is possible. This semantics was also used in [BN94] to discuss an operation of fair non-deterministic choice. In the present paper we will follow the definitions in [MS06]. In this section we use a concrete relational semantics; a more general semantics in terms of so-called modal semirings is given in Appendix B.

Definition 3.1. Consider a set Σ of *states*; the exact nature of its elements does not matter.

1. A *command* over Σ is a pair (R, P) where $R \subseteq \Sigma \times \Sigma$ is a transition relation and P is a subset of Σ.
2. The *restriction* of a transition relation $R \subseteq \Sigma \times \Sigma$ to a subset $Q \subseteq \Sigma$ is $Q \downarrow R =_{df} R \cap Q \times \Sigma$.

The set P is intended to characterise those states for which the command cannot lead to abortion.

Now we define a number of basic commands and command-forming operators that correspond to programming constructs.

Definition 3.2.

1. The worst command abort is the one that offers no transitions and does not exclude abortion for any state:

   $$\mathsf{abort} =_{df} (\emptyset, \emptyset) \ .$$

2. The program skip does not do anything: it leaves the state unchanged and cannot lead to abortion for any state:

   $$\mathsf{skip} =_{df} (I, \Sigma) \ ,$$

   where $I =_{df} \{(s, s) \mid s \in \Sigma\}$ is the *identity relation* on states.

3. The command fail is quite peculiar: it does not offer any transitions but guarantees that no state may lead to abortion:

   $$\mathsf{fail} =_{df} (\emptyset, \Sigma) \ .$$

We will comment a bit more on it below.

4. The command $\mathsf{chaos} =_{df} (\Sigma \times \Sigma, \emptyset)$ is completely unpredictable.

We now define the operator $[\!]$ of non-deterministic choice.

Definition 3.3. Let $C = (R, P)$ and $D = (S, Q)$ be commands. The command $C \,[\!]\, D$ is intended to behave as follows. For a starting state s non-deterministically a transition under R or S is chosen (if there is any). Absence of abortion can be guaranteed for s iff it can be guaranteed under both C and D, i.e., iff $s \in P \cap Q$. Therefore we define

$$(R, P) \,[\!]\, (S, Q) =_{df} (R \cup S, P \cap Q) \ .$$

From Definition 3.3 it is easy to see that $[\!]$ is associative, commutative and idempotent and that fail is its neutral element. The intuition behind taking set union in the first and intersection in the second is the following: if there is a greater choice of transitions the set of states from which no abortion is possible obviously gets smaller.

Let us now see that these definitions solve the original problem of a naïve relational semantics in that they distinguish commands which may lead to abortion from those which have the same transitions but exclude abortion. A command of the first kind is skip $[\!]$ abort, one of the second kind skip by itself. Definition 3.3 yields

$$\mathsf{skip} \,[\!]\, \mathsf{abort} \;=\; (I, \Sigma) \,[\!]\, (\emptyset, \emptyset) \;=\; (I \cup \emptyset, \Sigma \cap \emptyset) = (I, \emptyset) \neq (I, \Sigma) \;=\; \mathsf{skip} \ .$$

On the other hand, the approach has the property that it admits all kinds of “counterintuitive” commands such as fail or $(I, \emptyset)$ that arose in our previous example. Therefore it is reasonable to distinguish a subclass of commands which assert absence of abortion only for those states for which they actually offer transitions. This is captured by the following definition.

Definition 3.4. A command (R, P) is *feasible* [Par83] when $P \subseteq \mathit{dom}(R)$.

It is easy to check that feasible commands are closed under $[\!]$. The role of feasibility for specification purposes will become clear later. Note that fail is not feasible.

Feasible commands are precisely the ones for which we can use the above-mentioned demonic semantics. If $C = (R, P)$ is feasible then $P \downarrow R$ is that part of the transition of C for which abortion is excluded for its starting states, namely the ones in $P \cap \mathit{dom}(R)$. In other words, if abortion is excluded, a successful transition is guaranteed. Conversely, every transition R that is intended to model such a behaviour can be represented by the feasible command $(R, \mathit{dom}(R))$. These connections are elaborated further in Appendix B.

We now prepare for the semantics of the if fi-construct.

Definition 3.5. Let (R, P) be a command and $Q \subseteq \Sigma$ be a set of states, e.g., of the ones that satisfy a Boolean expression as occurring within an if fi statement. Then the *guarded command* $Q \rightarrow (R, P)$ (where Q is called the *guard*) is defined by

$$Q \rightarrow (R, P) =_{df} (Q \downarrow R, \neg Q \cup P) \ ,$$

where $\neg Q$ is the complement of Q w.r.t. Σ.

In a starting state s this command can lead to a transition only if s is both in Q

and in the domain of R; if so, all possible transitions under R are allowed. Hence, abortion can be excluded if s is not in Q or in P, which explains the expression for the second component of the command. Note that in general $Q \rightarrow (R, P)$ is not feasible even if (R, P) is. Hence, the iterated choice $B_1 \rightarrow C_1 [\!] \cdots [\!] B_n \rightarrow C_n$ will generally also not be feasible and hence, by itself, is not adequate for modelling the general non-deterministic branching construct. This is remedied by the following definition.

Definition 3.6. Given a command (R, P), then the if fi-statement is defined by

$$\mathsf{if}\,(R, P)\,\mathsf{fi} \;=_{\mathit{df}}\; (R, P \cap \mathit{dom}(R)) \;,$$

where $\mathit{dom}(R) =_{\mathit{df}} \{s \in \Sigma \mid \exists\, t \in \Sigma :(s, t) \in R\}$ is the *domain* of R, i.e., the set of states from which transitions under R emanate.

The purpose of surrounding a command with if fi transforms it into a feasible command. This is used to define the semantics of the general non-deterministic branching construct as follows.

Lemma 3.7. *Given sets Q_i of states and commands (R_i, P_i), $(1 \leq i \leq n)$. Then*

$$\mathsf{if}\,Q_1 \rightarrow (R_1, P_1) [\!] \cdots [\!] Q_n \rightarrow (R_n, P_n)\,\mathsf{fi} \;=$$
$$\Big(\bigcup(Q_i \downarrow R_i),\; \big(\bigcup(Q_i \cap \mathit{dom}(R_i))\big) \cap \big(\bigcap(\neg Q_i \cup P_i)\big)\Big)$$

This now has Dijkstra's original semantics: if none of the guards opens, the command aborts rather than fails; in particular, it is feasible by construction.

Using if fi we can now give a formal semantics to scenarios.

Definition 3.8. The command $\mathsf{if}\,C_e \;[\!]\; C_s\,\mathsf{fi}$ is called the *command of the scenario* (C_e, C_s).

A sequential composition and, based on that, finite and infinite iteration of commands can also be defined in this style. Since we do not need them here, we refer to [MS06] for details.

3.2 Refinement and the Lattice of Commands

We now define an algebraic analogue of the refinement relation as introduced by [Bac78].

Definition 3.9. We set

$$(R, P) \sqsubseteq (S, Q) \;\Leftrightarrow_{\mathit{df}}\; Q \subseteq P \;\wedge\; Q \downarrow R \subseteq S \;.$$

This relation is reflexive and transitive and hence a pre-order. However, it is not antisymmetric. The associated equivalence relation is given by $C \equiv D \;\Leftrightarrow_{\mathit{df}}\; C \sqsubseteq D \;\wedge\; D \sqsubseteq C$. Componentwise, it works out to

$$(R, P) \equiv (S, Q) \;\Leftrightarrow\; P = Q \;\wedge\; P \downarrow R = P \downarrow S \;.$$

In a sense, the if fi-construct provides the "closest feasible refinement" of a command:

Lemma 3.10. $\mathsf{if}\,(R, P)\,\mathsf{fi}$ *is the $\sqsubseteq$-least refinement of (R, P) that preserves the transition R.*

We have the following connection between the refinement relation and non-deterministic choice.

Lemma 3.11. *For commands C, D we have $C \sqsubseteq D \Leftrightarrow C \mathbin{[\!]} D \equiv D$.*

As is known from order theory, the relation $\sqsubseteq$ can be transferred to the equivalence classes under $\equiv$, namely, two classes are related by $\sqsubseteq$ if any of their representatives are. This defines now a partial order on equivalence classes of commands. In the sequel we will work with such equivalence classes, but always denote them by suitable representatives.

The above lemma implies that (the equivalence class of) $C \mathbin{[\!]} D$ is the least upper bound of (the equivalence classes of) C and D w.r.t. $\sqsubseteq$.

However, it turns out that, for commands, there is also a greatest-lower-bound operator which will be important for the combination operator on scenarios we are going to define.

Lemma 3.12. *The greatest lower bound of commands (R, P) and (S, Q) w.r.t. $\sqsubseteq$ is*

$$(R, P) \sqcap (S, Q) \;=\; ((R \cap S) \cup (\neg P \downarrow S) \cup (\neg Q \downarrow R), P \cup Q) \; .$$

Moreover, $[\!]$ and $\sqcap$ distribute over each other, i.e., the commands form even a distributive lattice.

The proof can be found in Appendix B.

As we already have pointed out, the feasible commands are of particular interest. However, unlike in the case of $[\!]$, they are not closed under the $\sqcap$ operator. However, it turns out that, given two transition relations R and S, the meet of the feasible commands $(R, \mathit{dom}(R))$ and $(S, \mathit{dom}(S))$ is feasible again iff

$$\mathit{dom}(R \cap S) = \mathit{dom}(R) \cap \mathit{dom}(S) \; .$$

This means that for every state in the intersection of their domains R and S have to offer at least one common transition. This property is central for allowing an integration of R and S into a common specification; hence we introduce a name for it.

Definition 3.13. Two relations R, S are *integrable* iff $\mathit{dom}(R \cap S) = \mathit{dom}(R) \cap \mathit{dom}(S)$.

When the functional requirements of a system or a family are given in terms of scenarios, one has to reckon with inconsistency among the given scenarios. *Functional inconsistency* arises when the transition relations of the scenarios are not integrable [DFK$^+$98]. A further source of inconsistency is *dictionary inconsistency* (i.e., naming inconsistency). The detection of functional inconsistency can be partially automated, and a prototype tool called SCENATOR is presented in [DKM05, KWS03].

4 Formal Scenarios as a Product Family Algebra

As stated in the introduction, our objective is to provide a semantic model of product family algebra in terms of scenarios. To achieve this, we need to provide concrete definitions for the two product family algebra operators $\cdot$ and $+$ and to provide explicit definitions of 0 and 1. As we have seen, a single scenario provides a specification of one particular system. If we identify a single scenario with a possible/feasible product then sets of scenarios can be used to argue about product families and product lines.

In the remainder we assume that all scenarios work on a common state space. If the state spaces of the scenarios are not the same one can extend them to a common one [DFK+98].

Next we define an operator for combining two scenarios.

Definition 4.1. Let $S_C =_{df} (C_e, C_s)$ and $S_D =_{df} (D_e, D_s)$ be two scenarios on a common state space. The scenario $S_C \cdot S_D$ is defined by

$$S_C \cdot S_D =_{df} (C_e \mathbin{[\!]} D_e, C_s \sqcap D_s) \ . \tag{4.1}$$

If C_s and D_s are not integrable, we say that the scenarios S_C and S_D are *system-inconsistent*.

Informally, the operation $\cdot$ is justified as follows. The environment (in our example the the customer of the bank) acts in an arbitrary way. Hence we model its choice between the actions of C_e and D_e as non-deterministic choice. In contrast, the system (in our example the bank) has to react to each action of the environment. In order to be consistent, the reaction to an action a must be the same in C_s and D_s if both commands specify a reaction for a.

This definition explains why we are using the more complex setting of commands rather than that of pure relations with a demonic interpretation: the demonic meet is a partial operation, whereas a product family algebra needs a total operation. And the demonic meet is faithfully represented by the meet $\sqcap$ of commands, which is total.

The formal scenario $1_{\mathrm{SC}} =_{df} (\mathsf{fail}, \mathsf{chaos})$ can be viewed as the closed system that can be built from all the given scenarios. Its environment does not have any effect on any environment of the given formal scenarios. It is the neutral element w.r.t. the combination operator $\cdot$. Moreover, it is easy to see that the operation is associative, commutative and idempotent.

If one wants to model the whole specification from the user's perspective, one might argue that the system behaves more or less arbitrarily. Hence one can define the symmetric (dual) operation $\cdot_\delta$ by

$$S_C \cdot_\delta S_D =_{df} (C_e \sqcap D_e, C_s \mathbin{[\!]} D_s) \ .$$

If C_e and D_e are not integrable, we say that the scenarios S_C and S_D are *environment-inconsistent*. All the presented theory works also for this operation.

The formal scenario $1^\delta_{\mathrm{SC}} =_{df} (\mathsf{chaos}, \mathsf{fail})$ specifies a system that involves all the consistent commands of the system corresponding to all the scenarios given to us. Its environment command can be refined by all the commands of the environment of all the scenarios.

For two sets $\mathcal{S}$ and $\mathcal{T}$ of scenarios, the operator $\cdot$ is lifted pointwise to sets of scenarios, i.e., $\mathcal{S} \cdot \mathcal{T} =_{df} \{S_C \cdot S_D \mid S_C \in \mathcal{S}, S_D \in \mathcal{T}\}$. Based on that we can now define a product family algebra for scenarios.

Theorem 4.2. *Let M be a set of scenarios that is closed under $\cdot$ and contains 1_{SC}. Then the structure* SC $=_{df} (\mathcal{P}(M), \cup, \emptyset, \cdot, \{1_{SC}\})$ *is a product family algebra. Under analogous conditions $(\mathcal{P}(M), \cup, \emptyset, \cdot_\delta, \{1^\delta_{SC}\})$ is a product family algebra.*

By this we have defined a product family algebra which now allows semantic reasoning. Its natural order corresponds to set inclusion $\subseteq$.

In the literature, terms like product, feature and subfamily lack an exact definition. In [HKM06a, HKM06, HKM09], we find the algebraic definitions for these terms based on product family algebra. For example a product is defined as follows.

Definition 4.3. [HKM09] Assume a product family algebra $F = (S, +, 0, \cdot, 1)$. An element $a \in S$ is said to be a *product* if it satisfies the following laws:

$$\forall\, b \in S : b \leq a \;\Rightarrow\; (b = 0 \,\vee\, b = a) \;, \tag{4.2}$$

$$\forall\, b, c \in S : a \leq b + c \;\Rightarrow\; (a \leq b \,\vee\, a \leq c) \;. \tag{4.3}$$

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

Intuitively, this means that a product cannot be divided using the choice operator $+$. Or in other terms, it does not offer optional or alternative features. In SC, exactly the sets with at most one member are products.

Analogously to Definition 4.3, a feature can be defined by indivisibility; this time w.r.t. multiplication rather than addition [HKM09]. Unfortunately, the definition is not useful in the present context: e.g., an indivisible part of a transition relation would be a single pair of states; it is not realistic to describe a complete system as the dot-integration of the respective commands. Further details on this are beyond the scope of the paper.

5 Illustrative Example and Further Applications

5.1 Informal Description of a Product Family

We now make our simple banking example into a proper family. For reasons of space we only give the informal description and merely sketch the formalisation; the principles should be clear from the earlier examples.

Let us assume that a software development department of a bank operating world-wide has a software product family to address its specific banking operations in several countries. The family enables several ways of opening accounts. All the products of the banking software family include a feature that allows customers to open an account when they visit a branch of the bank; this is formally described by the scenario `openAccAtBranch` from Section 2.2. Certainly, a product will contain several additional features related to other core banking activities, described by a scenario `restOfCoreBnkgSystem`.

Our product family allows the optionality of a feature `openAccountOnline` to open an account online and a feature `openAccountByMail` to open an account by sending the application and the needed documents by mail.

The scenario corresponding to `openAccountOnline` is the following: The customer logs into the web site of the bank corresponding to her country of residence. She then selects the open account operation. The system retrieves the appropriate eForm for opening an account. The customer fills in the field corresponding to her identification number and the type of identification document. If she is already recorded in the system, it displays the remaining needed information and proposes personalized account privileges. Otherwise, the system displays that the customer is new, asks for her full name and address, and assigns to the account the standard banking privileges. If the customer accepts the privileges, the system asks the customer to enter the data of a valid major credit card to pay for the opening fees. If the data are valid, the system issues a receipt containing the account number and a message stating that the card associated with the opened account will be handed to her at her first visit to one of the bank branches. Otherwise, after the third attempt the system aborts the operation and goes back to the main bank webpage.

The scenario corresponding to `openAccountByMail` reads as follows. If the country of residence of the customer is a member of the *Universal Postal Union* and the mail services of that country are considered as reliable by the bank, a customer can open an account using the mail. She sends an application for opening an account with one original identification document and the fees for opening the account. When the bank receives the application, an appropriate identification document, and the opening fees, it proceeds to the opening of the account and issues a card associated to the account. The process is similar to that for opening an account at a branch. It then returns by mail the identification document and the issued card to the customer. The account is considered open from the time the bank posts the card.

5.2 Formal Specification of the Family

The *product family algebra model* (FAM) of the above family is the following:

$$\begin{aligned}&\texttt{BankingFamily} = \\ &\qquad \texttt{openAccAtBranch} \cdot (1_{\mathrm{SC}} + \texttt{openAccountOnline} + \texttt{openAccountByMail}) \\ &\qquad \cdot\, \texttt{restOfCoreBnkgSystem}\end{aligned}$$

The scenarios `openAccAtBranch` and `restOfCoreBnkgSystem` are integrable with `openAccountOnline` and with `openAccountByMail`. However, `openAccountOnline` and `openAccountByMail` are not integrable since they treat e.g., the issued bank card differently.

5.3 Model Transformations

For instance, to generate the specification of the commonality of the above family, we proceed as follows:

1. We identify the product (according to the understanding given by Definition 4.3) that is common to all the members of the family. This product is formed as the dot-integration of the features common to all the members. From the above expression we can derive, by associativity and commutativity, that this is `openAccAtBranch` $\cdot$ `restOfCoreBnkgSystem`. Of course, this extraction of the commonality can also easily be automated; see for instance the prototype tool described in [HKM06a]. If the detailed expressions for the scenarios are analysed further, parts common to just two of them may be identified (see the phrase "The process is similar to that for opening an account at a branch" in the informal specification of `openAccountByMail`); this provides a way of refactoring specifications and implementations.
2. We replace each of the scenarios that occur in the above commonality specification by its corresponding formal scenarios to perform, if possible (i.e., when all the relations of the system of all the scenarios are consistent), their dot-integration. We note that dot-integration is associative and commutative and therefore the order in which we integrate the scenario does not matter.

3. In the same way, we can build the specification of each potential member of this example family when that is possible. Also, the specification of any sub-family can be generated in the same way.
4. Since we are building on an algebra of commands, efficiency-increasing transformations using the relation $\equiv$ are also semantics-preserving and hence admissible. However, the definition of product family so far does not take a refinement relation like $\sqsubseteq$ into account; this will be the subject of further work.

5.4 Application to Other Semantic Models

We have now seen how the approach works in a concrete semantic algebra of basic features. To show that it is more generally applicable we sketch three envisaged other semantic algebras.

A first idea is to define something in between the purely syntactic algebra, where products are just strings of feature names, and the purely semantic command algebra without a useful set of atomic features. In the new algebra one might use triples (x, C_e, C_s) as atomic features, where x is a feature name and (C_e, C_s) is a scenario. The elements of a corresponding product family algebra could then be sets of bags of such triples, where every bag has for a given name x only identical triples, if any. This allows identification and counting and still offers a semantic interpretation of features and products.

A second idea is to use as elements of a product family algebra sets consisting of unordered feature structure forests (FSFs) in the sense of [ALM+10] with commutative superimposition as the interpretation for composition $\cdot$.

A third idea is to form a product family algebra based on stream processing functions (e.g. [BS01]) using the $\otimes$ operator of component composition as the interpretation for composition $\cdot$.

All such applications would open the possibility of an algebraic treatment of view reconciliation and feature interaction along the lines of [HKM09] in those areas.

6 Related Work

It is a common belief in the requirements community that scenario-based or use case based descriptions or requirement specifications help to reduce the effort of model construction [UBC09]. The CREWS[1] group has visited twelve projects in Germany and Switzerland that used scenarios in their software engineering process in one way or another [AEG+98]. The survey revealed that scenarios are flexible and broadly applicable. The excessive complexity of typical software systems makes monolithic software specifications beyond the grasp of most software engineers and most software users. Scenarios allow us to structure complex specifications as aggregates of simple scenarios describing the user/environment system interactions. It allows us to address the inability of typical users to understand formal requirements specifications by allowing them provide the specifier with informal descriptions of the system in response to a business event. By virtue of its provision for covering systematically all relevant aspects of given requirements, scenarios help addressing the difficulty to elicit and capture user requirements in a systematic, verifiable manner.

[1] Co-operative Requirements Engineering With Scenarios

Model-driven development is a paradigm that helps address several problems related to composition and integration of systems from parts. Recently several attempts were made to formally extend the use of model-driven development to product families. Schätz [Sch07] proposes an integrated approach for variability modeling and model-based development and he illustrates a possible tool-support based on it. Thaker et al. [TBK$^+$07] introduce a technique to synthesize programs of a software product line by composing modules that implement features. They focus mainly on low-level implementation constraints such as features referencing elements that are defined in other feature modules and on assuring that all programs in a product line do not reference to undefined classes, methods, and variables.

7 Conclusion and Future Work

We have presented a mathematical framework that enables the transition from a family model and a set of initial models into derived models of the family members, or of that of the commonality of the family or of any of its sub-families. The family-model is a product family algebra term. Each of the other initial models is a formal scenario that captures environment-system interaction. Obtaining the model of a member is done through dot-integration of all its formal scenarios. Through this integration, inconsistency can be detected upon verification. When a concrete model of a family is derived, an additional verification is performed on the consistency of alternative features (formal scenarios). Indeed, +-integration requires that the environments of alternative formal scenarios need to be consistent; otherwise they cannot be taken as alternatives. Through our work the analysis activities that are performed in monolithic software development can be seen as a special case of that of a family: the case of a singleton family. The usual verification of requirements consistency and completeness are basic activities in the model transformation process that we propose.

In [MPK$^+$10], Méndez Fernández et al. point to the need for precise structure, syntax and semantics of requirements documents in order to ensure precise requirements. They propose a meta model for artefact-orientation. We can see that the language of product family algebra can be used in articulating the artefact abstraction model that they propose. The family model obtained after instantiating the features with their corresponding formal scenarios provides a part of the artefact content. We envision a requirements documentation technique that uses the language of product families, scenarios, and commands. It would be a significant step towards attaining a precise requirements that can evolve despite the volatility of some of the requirements of the documented product family.

One of the inherent risks with modelling is that by raising the level of abstraction one might over simplify to such an extent that no details are left for answering useful questions. However, by adopting several levels of abstraction such that each lower level is derived from a higher one by instantiating its elements, one can use only the model at the appropriate level of abstraction to answer a question without dealing with unnecessary details. In our case, our high level is the family specification expressed in terms of black boxes called features. Then, formal scenarios expressed using commands instantiate the features of the family. The obtained detailed model can answer questions such as system correctness and environment adequacy.

The contributions to model driven development of the requirements that we have reported in this paper are two-fold. First, we set up a mathematical background for formal derivation of high level requirements to a more concrete level. Second, our approach deals with expected changes through the adoption of a family approach as well as with unexpected ones by having a mathematical setting that enhances automation. Indeed, based on the mathematics presented one can easily construct tools to perform model transformation and the verification of family-models. A change to a software family touches either the product family algebra term that specifies it or its set of formal scenarios. Then, through automation, the models of the members, sub-families, commonality, and other possible submodels are updated. Once the generated models are obtained, a further analysis needs to be performed in order to assess the effect of the constraints placed upon them by an existing product line architecture. Therefore, some of the derived models are possible but not viable due to these constraints.

The proposed approach improves quality by encouraging reuse of already existing formal scenarios, building on the family commonalities, and easily coping with change in the requirements. The reuse is enhanced through the verification allowed by dot-integration, which enables verifying whether a scenario can be composed from already existing ones. The approach enhances consistency verification not only of the behaviour of the system but also of that of its environment. For instance, if two scenarios are alternative, the proposed approach enables verifying whether their respective required environments are consistent or not. The consistency verification can be automated [KWS03], which enhances the scalability of the approach, since many of the tasks of the verification are repetitive and can be delegated to mechanized mathematics machinery such as theorem provers and computer algebra systems.

Once a requirements model of a family member is obtained, the work presented in [KB04] shows how it can be used to derive the member's functional architectural design. The relation of the environment is used for acceptance and system testing [KB04]. It constitutes the specification of the tester; the tester needs to act according to the relation of the environment.

The proposed technique is confined to functional requirements. Aspects such as a system's performance are not addressed. We simply focus on the models that capture the business functionality and behaviour, which commonly are called *Platform Independent Model* (PIM). Our early work on view reconciliation [HKM09] can be used in some typical applications to generate *Platform Specific Models* (PSM). However, additional investigation is needed to develop techniques to incorporate non-functional requirements (overall qualities) of the system in the model transformation.

Some systems by their nature exhibit an inherent architecture that might involve several agents that act concurrently. Our future work aims at involving the inherent architecture of the family in the integration of the features.

Acknowledgment We are grateful to Christian Lengauer for valuable comments.

References

[ALM$^+$10] S. Apel, C. Lengauer, B. Möller, C. Kästner: An Algebraic Foundation for Automatic Feature-Based Program Synthesis. Sci. Computer Programming 75, 1022–1047 (2010)

[AEG$^+$98] M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, and K. Weidenhaupt: Survey on the scenario use in twelve selected industrial projects. Technical report, Aachener Informatik Berichte (AIB), No. 98-07, RWTH Aachen, Fachgruppe Informatik, 1998.

[Bac78] R.-J. Back: On the correctness of refinement steps in program development PhD thesis, Åbo Akademi, Department of Computer Science, 1978.

[Bac89] R.-J. Back: A method for refining atomicity in parallel algorithms. In E. Odijk, M. Rem and J.-C. Syre (eds.): Proc. Parallel Architectures and Languages Europe, Volume II: Parallel Languages. LNCS 366, 199–216, Springer, 1989.

[BW93] R. Backhouse, J. van der Woude: Demonic operators and monotype factors. Mathematical Structures in Computer Science 3(4), 417–433 (1993).

[BZ86] R. Berghammer, H. Zierer: Relational algebraic semantics of deterministic and non-deterministic programs. Theoretical Computer Science 43, 123–147 (1986).

[BBJ07] J. Bézivin, M. Barbero, F. Jouault: On the applicability scope of model driven engineering. In J. Fernandes, R. Machado, R. Khedri, S. Clarke (eds.): *Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 2007).* 3–7, IEEE, 2007.

[Bro06a] M. Broy: Challenges in automotive software engineering. In L.J. Osterweil, H.D. Rombach and M.L. Soffa (eds.): *International Conf. Software engineering (ICSE '06).* 33–42, ACM, 2006.

[Bro06b] M. Broy: The 'Grand Challenge' in informatics: engineering software-intensive systems. IEEE Computer 39(10), 72 – 80 (2006).

[BGW79] M. Broy, R. Gnatz, M. Wirsing: Semantics of nondeterministic and non-continuous constructs. In F.L. Bauer, M. Broy (eds.): *Program construction.* LNCS 69, 553–592, Springer, 1979.

[BN94] M. Broy, G. Nelson: Adding fair choice to Dijkstra's calculus. ACM Transactions on Programming Languages and Systems 16, 924–938 (1994).

[BS01] M. Broy, K. Stølen: Specification and Development of Interactive Systems — Focus on Streams, Interfaces, and Refinement. Springer 2001

[DBS$^+$95] J. Desharnais, N. Belkhiter, S. Sghaier, F. Tchier, A. Jaoua, A. Mili, N. Zaguia: Embedding a demonic semilattice in a relation algebra. Theoretical Computer Science 149, 333–360 (1995).

[DFK$^+$98] J. Desharnais, M. Frappier, R. Khedri, A. Mili: Integration of sequential scenarios. IEEE Transactions on Software Engineering 24(9), 695–708 (1998).

[DKM05] J. Desharnais, R. Khedri, A. Mili: Representation, validation and integration of scenarios using tabular expressions. Formal Methods in System Design (accepted 2005, in press).

[DMN97] J. Desharnais, A. Mili, T.T. Nguyen: Refinement and demonic semantics. In C. Brink, W. Kahl, G. Schmidt (eds): *Relational methods in computer science*, Advances in Computer Science, 166–183, Springer, 1997.

[DMS06] J. Desharnais, B. Möller, G. Struth: Kleene algebra with domain. ACM Transaction on Computational Logic 7(4), 798–833 (2006).

[DMT04] J. Desharnais, B. Möller, F. Tchier: Kleene under a modal demonic star. Journal of Logic and Algebraic Programming 66(2), 127–160 (2006).

[Dij76] E. Dijkstra: *A discipline of programming.* Prentice-Hall, 1976.

[Doo94] H. Doornbos: A relational model of programs without the restriction to Egli-Milner-monotone constructs. In E.-R. Olderog (ed.): *Programming concepts, methods and calculi.* IFIP Transactions, 363–382, North-Holland, 1994.

[Fau01] S. Faulk: Product-line requirements specification (PRS): an approach and case study. In *IEEESymposium on Requirements Engineering (RE 2001)*, 2001. 48–55, IEEE, 2001.

[GM06] W. Guttmann, B. Möller: Modal design algebra. In S. Dunne, W. Stoddart (eds.): *Unifying Theories of Programming.* LNCS 4010, 236–256, Springer 2006.

[HW98] U. Hebisch and H. Weinert: *Semirings — Algebraic Theory and Applications in Computer Science.* World Scientific, 1998.

[HH98] T. Hoare, J. He: *Unifying theories of programming.* Prentice Hall, 1998.

[HKM06] P. Höfner, R. Khedri, B. Möller: Feature algebra. Technical Report Report 2006-04, Institut für Informatik, Universität Augsburg, 2006.

[HKM06a] P. Höfner, R. Khedri, B. Möller: Feature algebras. In J. Misra, T. Nipkow, E. Sekerinski (eds.): *FM 2006: Formal Methods.* LNCS 4085, 300–315, Springer, 2006.

[HKM09] P. Höfner, R. Khedri, B. Möller: An algebra of product families. Software and Systems Modeling, 2009.

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

[KB04] R. Khedri, I. Bourguiba: Formal derivation of functional architectural design. In *IEEE Conf. Software Engineering and Formal Methods*, 356–365, IEEE, 2004.

[KB04] R. Khedri, I. Bourguiba: Requirements scenaros based system-testing. In F. Maurer and G. Ruhe (eds.): *Software Engineering and Knowledge Engineering (SEKE'04).* 252–257, Knowledge Systems Institute Graduate School, 2004.

[KWS03] R. Khedri, R. Wu, B. Sanga: SCENATOR: a prototype tool for requirements inconsistency detection. In F. Wang and I. Lee, (eds.): *Automated Technology for Verification and Analysis*, 75–86, National Taiwan University, 2003.

[Koz97] D. Kozen: Kleene algebra with tests. ACM Transactions on Programming Languages and Systems 19(3), 427–443 (1997).

[MPK$^+$10] D. Méndez Fernández, B. Penzenstadler, M. Kuhrmann, and M. Broy: A Meta Model for Artefact-Orientation: Fundamentals and Lessons Learned in Requirements Engineering. In D.C. Petriu, N. Rouquette, O. Haugen (eds.): *MODELS 2010.* LNCS 6395, 183–197, Springer, 2010.

[Möl04] B. Möller: Kleene getting lazy. Science of Computer Programming 65, 195–214 (2007).

[MS06] B. Möller, G. Struth: wp is wlp. In W. MacCaull, M. Winter and I. Duentsch (eds.): *Relational Methods in Computer Science.* LNCS 3929, 121–133, Springer, 2006.

[MYC05] M. Moon, K. Yeom, H. S. Chae: An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. IEEE Transactions on Software Engineering 31(7), 551–569 (2005).

[Mor88] C. Morgan: The specification statement. ACM Transactions on Programming Languages and Systems 10(3), 403–419 (1988).

[Mor87] J.M. Morris: A theoretical basis for stepwise refinement and the programming calculus. Sci. Computer Programming 9(3), 287-306 (1987)

[Nel89] G. Nelson: A generalization of Dijkstra's calculus. ACM Transactions on Programming Languages and Systems 11, 517–561 (1989).

[Par83] D. Parnas: A generalized control structure and its formal definition. Communications of ACM 26, 572–581 (1983)

[Sch07] B. Schätz: Combining product lines and model-based development. Electronic Notes in Theoretical Computer Science 182 , 171–186, Elsevier, 2007.

[SOM$^+$05] J. Savolainen, I. Oliver, M. Mannion, H. Zuo: Transitioning from product line requirements to product line architecture. In *International Computer Software and Applications Conference (COMPSAC 2005)*, 186–195, IEEE, 2005.

[Spi88] M. Spivey: *Understanding Z.* Cambrigde University Press 1988.

[TBK$^+$07] S. Thaker, D. Batory, D. Kitchin, W. Cook: Safe composition of product lines. In C. Consel and J.L. Lawall (eds.): *Generative programming and component engineering*, 95 – 104, ACM, 2007.

[UBC09] S. Uchitel, G. Brunet, and M. Chechik. Synthesis of Partial Behaviour Models from Properties and Scenarios. IEEE Transactions on Software Engineering 35(3), 384-406 (2009).

[WL99] D. Weiss, C.T. R. Lai: *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison Wesley Longman, 1999.

Appendix

A The Banking Example Completed

To describe the whole banking system, we start to explain members of the state space:

`cstmerEligOpnAcc`	a customer is eligible to open an account
`idNum`	an arbitrary identification number provided by the customer
`idDocType`	type of the identification document
`fldIdNum`	an arbitrary identification number provided by the customer
`fldIdType`	an arbitrary type of the identification document provided by the customer
`newCstmer`	a customer is new
`acctCreated`	an account is created
`acctprivileges`	a type of privileges that the bank assign to a customer, which can be *standard* or *personalized*
`prvlgesAccepted`	the customer accepts the privileges
`feesPayed`	the fees for opening the account are payed
`crdIssued`	a card associated with the account is issued
`outputMssge`	a message from the system to the user
`csrmerName`	an arbitrary customer's name
`csrmerAddress`	an arbitrary customer's address
`fldCsrmerName`	customer's name as entered to the system
`fldCstmerAddress`	customer's address as entered to the system
`getCstmerName(idNum)`	system function to get the name of the custfrom an internal date store
`getCstmerAddress(idNum)`	system function to get customer's address from an internal date store
`personalized`	the *personalized* privilege
`standard`	the *standard* privilege
`msgeAccptPrvlges?`	a message from the system asking whether the customer accepts the proposed privilege

Now we are able give the complete formal specification of the banking system:

$$\begin{array}{l}
\quad \texttt{openAccAtBranch}_s \\
=_{df} \\
\qquad \Big(\texttt{cstmerEligOpnAcc} \ \wedge \ \texttt{fldIdNum} = \texttt{idNum} \\
\qquad\quad \wedge \ \texttt{fldIdType} = \texttt{idDocType} \ \wedge \ \neg\texttt{newCstmer} \\
\qquad\quad \wedge \ \neg\texttt{acctCreated} \ \wedge \ \neg\texttt{prvlgesAccepted} \\
\qquad\quad \wedge \ \neg\texttt{feesPayed} \ \wedge \ \neg\texttt{crdIssued} \\
\qquad\quad \longrightarrow \texttt{fldCsrmerName} \ := \ \texttt{getCstmerName}(\texttt{idNum}) \\
\qquad\quad\ \ ; \texttt{fldCstmerAddress} \ := \ \texttt{getCstmerAddress}(\texttt{idNum})\Big) \\
\qquad [] \Big(\texttt{cstmerEligOpnAcc} \ \wedge \ \texttt{fldIdNum} = \texttt{idNum} \\
\qquad\quad \wedge \ \texttt{fldIdType} = \texttt{idDocType} \ \wedge \ \neg\texttt{newCstmer} \\
\qquad\quad \wedge \ \texttt{fldCsrmerName} = \texttt{getCstmerName}(\texttt{idNum}) \\
\qquad\quad \wedge \ \texttt{fldCstmerAddress} = \texttt{getCstmerAddress}(\texttt{idNum}) \\
\qquad\quad \longrightarrow \texttt{acctprivileges} \ := \ \texttt{personalized}; \texttt{outputMssge} \ := \ \texttt{msgeAccptPrvlges?}\Big)
\end{array}$$

$$\begin{array}{l}
[\!]\Big(\mathtt{cstmerEligOpnAcc} \;\wedge\; \mathtt{fldIdNum} = \mathtt{idNum} \\
\quad \wedge\; \mathtt{fldIdType} = \mathtt{idDocType} \;\wedge\; \neg\mathtt{newCstmer} \\
\quad \wedge\; \mathtt{fldCsrmerName} = \mathtt{getCstmerName}(\mathtt{idNum}) \\
\quad \wedge\; \mathtt{fldCstmerAddress} = \mathtt{getCstmerAddress}(\mathtt{idNum}) \\
\quad \wedge\; \mathtt{acctprivileges} = \mathtt{personalized} \;\wedge\; \mathtt{outputMssge} = \mathtt{msgeAccptPrvlges?} \\
\quad \wedge\; \mathtt{prvlgesAccepted} \\
\quad \longrightarrow \mathtt{acctCreated} \;:=\; \mathit{true}; \mathtt{crdIssued} \;:=\; \mathit{true}\Big) \\
[\!]\Big(\mathtt{cstmerEligOpnAcc} \;\wedge\; \mathtt{fldIdNum} = \mathtt{idNum} \\
\quad \wedge\; \mathtt{fldIdType} = \mathtt{idDocType} \;\wedge\; \mathtt{newCstmer} \;\wedge\; \mathtt{fldIdNum} = \mathtt{idNum} \\
\quad \wedge\; \mathtt{fldIdType} = \mathtt{idDocType} \\
\quad \longrightarrow \mathtt{acctprivileges} \;:=\; \mathtt{standard}; \mathtt{outputMssge} \;:=\; \mathtt{msgeAccptPrvlges?}\Big) \\
[\!]\Big(\mathtt{cstmerEligOpnAcc} \;\wedge\; \mathtt{fldIdNum} = \mathtt{idNum} \\
\quad \wedge\; \mathtt{fldIdType} = \mathtt{idDocType} \;\wedge\; \neg\mathtt{newCstmer} \\
\quad \wedge\; \mathtt{fldIdNum} = \mathtt{idNum} \;\wedge\; \mathtt{fldIdType} = \mathtt{idDocType} \\
\quad \wedge\; \mathtt{acctprivileges} = \mathtt{personalized} \;\wedge\; \mathtt{outputMssge} = \mathtt{msgeAccptPrvlges?} \\
\quad \wedge\; \mathtt{prvlgesAccepted} \\
\quad \longrightarrow \mathtt{acctCreated} \;:=\; \mathit{true}; \mathtt{crdIssued} \;:=\; \mathit{true}\Big)
\end{array}$$

Next, we give the complete specification of user and environment:

$$\begin{array}{l}
\quad \mathtt{openAccAtBranch}_e \\
=_{df} \\
\quad \Big(\mathtt{cstmerEligOpnAcc} \;\wedge\; \neg(\mathtt{fldIdNum} = \mathtt{idNum}) \\
\quad\quad \wedge\; \neg(\mathtt{fldIdType} = \mathtt{idDocType}) \;\wedge\; \neg\mathtt{acctCreated} \\
\quad\quad \wedge\; \neg\mathtt{prvlgesAccepted} \;\wedge\; \neg\mathtt{feesPayed} \;\wedge\; \neg\mathtt{crdIssued} \\
\quad\quad \longrightarrow \mathtt{fldIdNum} \;:=\; \mathtt{idNum}; \mathtt{fldIdType} \;:=\; \mathtt{idDocType}\Big) \\
\quad [\!]\Big(\mathtt{cstmerEligOpnAcc} \;\wedge\; \mathtt{fldIdNum} = \mathtt{idNum} \\
\quad\quad \wedge\; \mathtt{fldIdType} = \mathtt{idDocType} \;\wedge\; \mathtt{newCstmer} \\
\quad\quad \wedge\; \neg\mathtt{acctCreated} \;\wedge\; \neg\mathtt{prvlgesAccepted} \\
\quad\quad \wedge\; \neg\mathtt{feesPayed} \;\wedge\; \neg\mathtt{crdIssued} \\
\quad\quad \longrightarrow \mathtt{fldCsrmerName} \;:=\; \mathtt{csrmerName}; \mathtt{fldCstmerAddress} \;:=\; \mathtt{csrmerAddress}\Big) \\
\quad [\!]\Big(\mathtt{cstmerEligOpnAcc} \;\wedge\; \mathtt{fldIdNum} = \mathtt{idNum} \\
\quad\quad \wedge\; \mathtt{fldIdType} = \mathtt{idDocType} \;\wedge\; \neg\mathtt{newCstmer} \\
\quad\quad \wedge\; \mathtt{fldCsrmerName} = \mathtt{getCstmerName}(\mathtt{idNum}) \\
\quad\quad \wedge\; \mathtt{fldCstmerAddress} = \mathtt{getCstmerAddress}(\mathtt{idNum}) \\
\quad\quad \wedge\; \mathtt{acctprivileges} = \mathtt{personalized} \;\wedge\; \mathtt{outputMssge} = \mathtt{msgeAccptPrvlges?} \\
\quad\quad \longrightarrow \mathtt{prvlgesAccepted} \;:=\; \mathit{true}\Big) \\
\quad [\!]\Big(\mathtt{cstmerEligOpnAcc} \;\wedge\; \mathtt{fldIdNum} = \mathtt{idNum} \\
\quad\quad \wedge\; \mathtt{fldIdType} = \mathtt{idDocType} \;\wedge\; \mathtt{newCstmer} \\
\quad\quad \wedge\; \mathtt{fldIdNum} = \mathtt{idNum} \;\wedge\; \mathtt{fldIdType} = \mathtt{idDocType} \\
\quad\quad \wedge\; \mathtt{acctprivileges} = \mathtt{standard} \;\wedge\; \mathtt{outputMssge} = \mathtt{msgeAccptPrvlges?} \\
\quad\quad \longrightarrow \mathtt{prvlgesAccepted} \;:=\; \mathit{true}\Big)
\end{array}$$

B Algebraic Semantics of Commands

B.1 Modal Semirings

We will model programs algebraically using elements of a so-called modal semiring S; the precise definitions will be given below. The idea is that the elements of S model sets of transitions from program states to program states. A particular subset of the elements, the tests, model sets of states or, equivalently, assertions about program states.

Formally, an *idempotent semiring* is a structure $(S, +, 0, \cdot, 1)$ satisfying the following axioms.

- The substructure $(S, +, 0)$ is a commutative and idempotent monoid. This induces the *natural order* $a \leq b \Leftrightarrow_{df} a + b = b$ w.r.t. which 0 is the least element and $a + b$ is the join of a and b.
- The substructure $(S, \cdot, 1)$ is a monoid.
- Composition $\cdot$ distributes over sum in both arguments, i.e., $(a+b) = a{\cdot}c+b{\cdot}c$ and $a \cdot (b + c) = a \cdot b + a \cdot c$.
- The element 0 is a left and right annihilator w.r.t. composition, i.e., $0 \cdot a = 0 = a \cdot 0$.

In most applications these operators are interpreted as follows:

$$\begin{array}{lll@{\qquad}lll} + & \leftrightarrow & \text{choice,} & \cdot & \leftrightarrow & \text{sequential composition,} \\ 0 & \leftrightarrow & \text{empty choice,} & 1 & \leftrightarrow & \text{null action,} \\ \leq & \leftrightarrow & \multicolumn{4}{l}{\text{increase in information or in choice possibilities.}} \end{array}$$

A prominent idempotent semiring is the set of all binary relations over a set W with union as $+$ and relational composition as $\cdot$.

A *test* in an idempotent semiring is a subidentity $p \leq 1$ that has a complement $\neg p$ relative to 1, i.e., $p \cdot \neg p = 0 = \neg p \cdot p$ and $p + \neg p = 1$. If p characterises a set S of states then $\neg p$ characterises its complement. Note that the complement operation $\neg$ is required only for tests, not for general semiring elements, which allows a much wider class of models. The set of all tests of S is denoted by $\mathsf{test}(S)$.

In the relation semiring, the tests are the subidentities of the form $\Delta_V =_{df} \{(x, x) \mid x \in V\}$ for subsets $V \subseteq W$. So Δ_V can represent V as a relation and hence model the predicate characterising V.

The above definition of tests deviates slightly from that in [Koz97] in that it does not allow an arbitrary Boolean algebra of subidentities as $\mathsf{test}(S)$ but only the maximal complemented one. The reason is that the axiomatisation of domain to be presented below forces this maximality anyway (see [DMS06]).

Straightforward calculations show that $\mathsf{test}(S)$ forms a Boolean algebra with $+$ as join, $\cdot$ as meet and 0 and 1 as its least and greatest elements. We will consistently write $a, b, c \ldots$ for arbitrary semiring elements and $p, q, r, \ldots$ for tests. When tests are viewed as predicates over a set W of possible worlds, the semiring operators play the following roles:

$$\begin{array}{rcl} 0\ /\ 1 & \leftrightarrow & \textit{false} \text{ (empty set) / } \textit{true} \text{ (full set } W\text{)}, \\ +\ /\ \cdot & \leftrightarrow & \text{disjunction (union) / conjunction (intersection)}, \\ \leq & \leftrightarrow & \text{implication (subsethood)}, \\ p \cdot a\ /\ a \cdot p & \leftrightarrow & \text{input / output restriction of } a \text{ by } p, \\ p \cdot a \cdot q & \leftrightarrow & \text{the part of } a \text{ taking } p\text{-elements to } q\text{-elements.} \qquad (*) \end{array}$$

An important property of tests is the following [Möl04]: if the meet $a \sqcap b$ exists then also $p \cdot a \sqcap b$ and $p \cdot a \sqcap p \cdot b$ with

$$p \cdot (a \sqcap b) \;=\; p \cdot a \sqcap b \;=\; p \cdot a \sqcap p \cdot b. \tag{2.4}$$

To ease reading, we will write $\wedge$ and $\vee$ instead of $\cdot$ and $+$ when both of their arguments are tests (metalogical conjunction and disjunction will be denoted with the larger $\bigwedge$ and $\bigvee$ to avoid confusion). Also, we will freely use the standard Boolean operations on $\mathsf{test}(S)$, for instance implication $p \rightarrow q \;=_{df}\; \neg p + q$.

To complete our setting we now introduce a *domain* operator. Given an element a the test $\ulcorner a$, the domain of a, characterises those states from which a-transitions are possible.

Formally, a *modal semiring* is a structure $(S, +, 0, \cdot, 1, \ulcorner)$ such that $(S, +, 0, \cdot, 1)$ is an idempotent semiring and the operator $\ulcorner : S \rightarrow \mathsf{test}(S)$ satisfies the axioms [DMS06]

$$a \leq \ulcorner a \cdot a, \qquad \ulcorner(p \cdot a) \leq p, \qquad \ulcorner(a \cdot b) = \ulcorner(a \cdot \ulcorner b).$$

In a modal semiring we can define the modal diamond and box operators $|\_\rangle, |\_] : S \rightarrow (\mathsf{test}(S) \rightarrow \mathsf{test}(S))$ as

$$|a\rangle p \;=_{df}\; \ulcorner(a \cdot p), \qquad |a]p \;=_{df}\; \neg|a\rangle\neg p.$$

This specifies $[a]q$ as the as the abstract counterpart of the weakest liberal precondition predicate transformer wlp [Dij76], with $p \leq |a]q$ representing the partial correctness semantics of the Hoare triple $\{p\}\, a\, \{q\}$.

B.2 Commands and Correctness

Over a modal semiring we can abstractly model programs with a general correctness view (i.e., "miraculous" behaviour is possible; we will later connect this to the total correctness view and demonic semantics.

Programs are modelled as *commands* [Nel89, MS06] taken from the set $\mathrm{COM(S)} =_{df} \mathrm{S} \times \mathsf{test}(\mathrm{S})$. In a command (a, p) the element $a \in S$ describes the possible state transitions and $p \in \mathsf{test}(S)$ characterises the states with guaranteed termination. All states characterised by $\neg p$ have the "result" of infinite looping besides any proper states that may be reached from them under a. The following definitions and properties are from [MS06].

The weakest (liberal) precondition can be defined as

$$\mathsf{wlp}.(a, p).q \;=_{df}\; |a]q\ , \qquad \mathsf{wp}.(a, p).q \;=_{df}\; p \wedge \mathsf{wlp}.(a, p).q\ .$$

This implies Nelson's *pairing condition* for commands k:

$$\mathsf{wp}.k.q \;=\; \mathsf{wp}.k.1 \wedge \mathsf{wlp}.k.q\ .$$

An important auxiliary concept is the *guard* of a command:

$$\mathsf{grd}.(a, p) \;=_{df}\; \neg\mathsf{wp}.(a, p).0 \;=\; \neg p \vee \ulcorner a \;=\; p \rightarrow \ulcorner a\ .$$

It characterises the set of states that, if non-diverging, allow a transition under a. A command is called *total* if its guard equals 1. The above formula links Parnas's condition [Par83] on termination constraints with totality:

$$\mathsf{grd}.(a,p) \;=\; 1 \;\Leftrightarrow\; p \le \ulcorner a \;.$$

Nelson remarks that totality of command k is also equivalent to Dijkstra's law $\mathsf{wp}.k.0 = 0$ of the excluded miracle.

The basic non-iterative commands are defined as

$$\begin{array}{c} \mathsf{fail} =_{df} (0,\top)\;, \qquad \mathsf{skip} =_{df} (1,\top)\;, \qquad \mathsf{abort} =_{df} (0,0)\;, \\ (a,p) \mathbin{[\!]} (b,q) =_{df} (a+b, p \wedge q)\;, \qquad (a,p)\,;(b,q) =_{df} (a \cdot b, p \wedge |a]q)\;. \end{array}$$

Here $p \wedge |a]q$ characterises those states for which a is guaranteed to terminate and which under a only lead to guaranteed termination states of b.

The commands form a *left semiring*, i.e., satisfy all semiring laws except for the right annihilation law for the zero element fail.

Theorem B.1. *The structure* $\mathrm{COM(S)} =_{df} (\mathrm{COM(S)}, [\!], \mathsf{fail}, ;, \mathsf{skip})$ *is an idempotent left semiring. The associated natural order on* $\mathrm{COM(S)}$ *is*

$$(a,p) \le (b,q) \;\Leftrightarrow\; a \le b \;\wedge\; p \ge q \;.$$

The proof can be found in [MS06]. It is essential that semiring S be a semiring and not only a left semiring.

As in [HH98] we say that command k is *(H4)* or *feasible* iff $k\,;\mathsf{abort} = \mathsf{abort}$. One calculates, using $|a]0 = \neg\ulcorner a$ and semiring properties,

$$(a,p)\,;\mathsf{abort} = (a \cdot 0, p\,|a]0) = (0, p\,\neg\ulcorner a)\;.$$

Corollary B.2. *Command* (a,p) *is feasible iff* $p \le \ulcorner a$.

So feasibility amounts exactly to Parnas's condition [Par83].

Therefore abort and skip are feasible, whereas fail is not. Moreover, $[\!]$ and ; preserve feasibility.

The feasible commands will give rise to demonic semantics (total correctness semantics) in Section B.4.

For the remainder of this chapter we will omit the operators $\wedge$ and $\cdot$ operator to simplify notation.

B.3 Refinement

Let us now look more closely at the natural order induced on the commands by the left semiring structure. By antitony of box we obtain for commands k, l

$$k \le l \;\;\Rightarrow\;\; \mathsf{wlp}.k \ge \mathsf{wlp}.l \;\;\wedge\;\; \mathsf{wp}.k \ge \mathsf{wp}.l\;,$$

where on the right hand side $\ge$ is the pointwise order between condition transformers. The second conjunct is the converse of the usual refinement relation. For it one obtains (see [MS06])

$$(\forall\, r : \mathsf{wp}.(a,p).r \ge \mathsf{wp}.(b,q).r) \;\Leftrightarrow\; p \ge q \;\wedge\; b \ge q\,a\;.$$

We use the latter formula as the refinement relation between commands:

$$(a,p) \sqsubseteq (b,q) \;\Leftrightarrow_{df}\; q \le p \;\wedge\; q\,a \le b\;.$$

Due to our generalised setting we only have $k \sqsubseteq l \;\Rightarrow\; \mathsf{wp}.k \geq \mathsf{wp}.l$. Equivalence holds if the underlying modal condition semiring S is *extensional*, i.e, if $|a\rangle \leq |b\rangle \;\Rightarrow\; a \leq b$ (the converse implication holds by isotony).

Unlike $\leq$ the relation $\sqsubseteq$ is only a pre-order with associated equivalence relation

$$k \equiv l \;\Leftrightarrow_{df}\; k \sqsubseteq l \;\wedge\; l \sqsubseteq k \ .$$

Componentwise, it works out to $(a,p) \equiv (b,q) \;\Leftrightarrow\; p = q \;\wedge\; p\,a \leq b \;\wedge\; p\,b \leq a$, which further simplifies to

$$(a,p) \equiv (b,q) \;\Leftrightarrow\; p = q \;\wedge\; p\,a = p\,b \ . \tag{eqc}$$

This agrees with the behaviour of designs described in [HH98]. For instance,

$$(p\,a,p) \equiv (a,p) \equiv (p \rightarrow a,p) \ .$$

Our relations between commands are put into perspective by

Lemma B.3.

1. $k \leq l \;\Rightarrow\; k \sqsubseteq l \;\Rightarrow\; \mathsf{wp}.k \geq \mathsf{wp}.l$.
2. $k \sqsubseteq l \;\Leftrightarrow\; k \,[\!]\, l \equiv l$.

The proof can be found in [MS06]. This lemma explains our choice for the direction of the $\sqsubseteq$ relation; in many texts on refinement it is used the other way around.

For calculations to work smoothly the following property is important:

Lemma B.4.

1. *The operations* $[\!]$ *and* ; *on commands are* $\sqsubseteq$*-isotone.*
2. *The equivalence* $\equiv$ *is a congruence w.r.t.* $[\!]$ *and* ;.

The proof can be found in [MS06].

Finally we look at the lattice structure of commands under $\sqsubseteq$. Note that join and meet can also be defined for pre-orders; they enjoy all the usual properties except that they are unique only up to the associated equivalence relation.

Lemma B.5.

1. *The join of commands* (a,p) *and* (b,q) *w.r.t.* $\sqsubseteq$ *is*

$$(a,p) \sqcup (b,q) \;=\; (a+b, p\,q) \;=\; (a,p) \mathbin{[\,]} (b,q)\ .$$

2. *If the meet* $a \sqcap b$ *exists then so does the meet of* (a,p) *and* (b,q) *w.r.t.* $\sqsubseteq$*, viz. (assuming that* $\sqcap$ *binds more strongly than* $+$*)*

$$(a,p) \sqcap (b,q) \;=\; (a \sqcap b + \neg p\,b + \neg q\,a + \neg p\,\neg q, p+q)\ .$$

3. *If* S *has a greatest element* $\top$ *then* $\mathsf{chaos} =_{df} (\top, 0)$ *is the* $\sqsubseteq$*-greatest element of* COM(S)*. Moreover,* chaos *is feasible.*

In the remainder we will work with the quotient set $\mathrm{C}(S) =_{df} \mathrm{COM(S)}/{\equiv}$ most of the time, but still abbreviate the classes $[(a,p)]_{\equiv}$ by their representatives (a,p).

We now prove two new results that are essential for our use of commands in a product family algebra.

Lemma B.6. *The equivalence* $\equiv$ *is a congruence w.r.t.* $\sqcup$,$\sqcap$, $p \to \_$ *and* $\mathsf{if\ fi}$.

Proof. We spell out the proofs for $\sqcup$ and $\sqcap$; the remaining ones are similar. Suppose $(a,p) \equiv (c,r)$, i.e., $p = r$ and $p\,a = p\,c$). We only show the congruence property for the first arguments of $\sqcup$ and $\sqcap$; for the second arguments it follows by commutativity of these operations.

For the join we have by definition

$$(a,p) \sqcup (b,q) \;=\; (a+b, p\,q), \qquad (c,p) \sqcup (b,q) \;=\; (c+b, p\,q).$$

Now

$$p\,q\,(a+b) \;=\; p\,q\,a + p\,q\,b \;=\; q\,p\,c + p\,q\,b \;=\; p\,q\,c + p\,q\,b \;=\; p\,q\,(c+b),$$

which shows $(a,p) \sqcup (b,q) \equiv (c,p) \sqcup (b,q)$.

For the meet we have by definition

$$\begin{aligned}(a,p) \sqcap (b,q) \;&=\; (a \sqcap b + \neg p\,b + \neg q\,a + \neg p\,\neg q, p+q),\\ (c,p) \sqcap (b,q) \;&=\; (a \sqcap b + \neg p\,b + \neg q\,a + \neg p\,\neg q, p+q).\end{aligned}$$

Now

$$\begin{aligned}
&\quad (p+q)\,(a \sqcap b + \neg p\,b + \neg q\,a + \neg p\,\neg q)\\
&= \quad \{\!\![\ \text{distributivity and test algebra}\ ]\!\!\}\\
&\quad p\,(a \sqcap b) + q\,(a \sqcap b) + \neg p\,q\,b + \neg q\,p\,a\\
&= \quad \{\!\![\ \text{splitting } q \text{ in second summand}\ ]\!\!\}\\
&\quad p\,(a \sqcap b) + p\,q\,(a \sqcap b) + \neg p\,q\,(a \sqcap b) + \neg p\,q\,b + \neg q\,p\,a\\
&= \quad \{\!\![\ \text{second summand subsumed by first one, third one by fourth one}\ ]\!\!\}\\
&\quad p\,(a \sqcap b) + \neg p\,q\,b + \neg q\,p\,a\\
&= \quad \{\!\![\ \text{test and meet (2.4)}\ ]\!\!\}\\
&\quad p\,a \sqcap b + \neg p\,q\,b + \neg q\,p\,a\\
&= \quad \{\!\![\ \text{assumption } p\,a = p\,c\ ]\!\!\}\\
&\quad p\,c \sqcap b + \neg p\,q\,b + \neg q\,p\,c\\
&= \quad \{\!\![\ \text{reverse derivation}\ ]\!\!\}
\end{aligned}$$

$$(p+q)\,(c \sqcap b + \neg p\, b + \neg q\, c + \neg p\, \neg q),$$

which shows $(a,p) \sqcap (b,q) \equiv (c,p) \sqcap (b,q)$. □

Lemma B.7. *If the underlying semiring is a distributive lattice then join and meet of commands distribute over each other in the following way.*

1. $((a,p) \sqcup (b,q)) \sqcap (c,r) \;=\; ((a,p) \sqcap (c,r)) \sqcup (b,q) \sqcap (c,r))$.
2. $((a,p) \sqcap (b,q)) \sqcup (c,r) \;\equiv\; ((a,p) \sqcup (c,r)) \sqcap (b,q) \sqcup (c,r))$.

Proof. 1. Plugging in the definitions we obtain

$$\begin{array}{ll} & ((a,p) \sqcup (b,q)) \sqcap (c,r) \\ = & ((a+b, p\,q) \sqcap (c,r) \\ = & ((a+b) \sqcap c + \neg(p\,q)\, c + \neg r\,(a+b) + \neg(p\,q)\,\neg r,\;\; p\,q + r) \end{array}$$

and

$$\begin{array}{ll} & ((a,p) \sqcap (c,r)) \sqcup (b,q) \sqcap (c,r)) \\ = & (a \sqcap c + \neg p\, c + \neg r\, a + \neg p\, \neg r,\;\; p + r) \sqcup (b \sqcap c + \neg q\, c + \neg r\, b + \neg q\, \neg r,\;\; q + r) \\ = & (a \sqcap c + \neg p\, c + \neg r\, a + \neg p\, \neg r + b \sqcap c + \neg q\, c + \neg r\, b + \neg q\, \neg r,\;\; (p+r)\,(q+r)) \\ = & \quad \{\!\![\text{ rearrangement and distributivity }]\!\!\} \\ & ((a+b) \sqcap c + (\neg p + \neg q)\, c + \neg r\,(a+b) + (\neg p + \neg q)\,\neg r,\;\; p\,q + r)) \end{array}$$

and de Morgan shows the claim.

2. Plugging in the definitions we obtain

$$((a,p) \sqcap (b,q)) \sqcup (c,r) \;=\; (a \sqcap b + \neg p\, b + \neg q\, a + \neg p\, \neg q + c,\;\; (p+q)\,r)$$

and

$$\begin{array}{ll} & ((a,p) \sqcup (c,r)) \sqcap (b,q) \sqcup (c,r)) \\ = & (a+c, p\,r) \sqcap (b+c, q\,r) \\ = & ((a+c) \sqcap (b+c) + \neg(q\,r)\,(a+c) + \neg(p\,r)\,(b+c) + \neg(q\,r)\,\neg(p\,r), \\ & \;p\,r + q\,r) \\ = & \quad \{\!\![\text{ distributivity and omitting summands} \le c \;]\!\!\} \\ & ((a \sqcap b) + c + \neg(q\,r)\, a + \neg(p\,r)\, b + \neg(q\,r)\,\neg(p\,r), \\ & \;(p+q)\,r) \\ = & \quad \{\!\![\text{ de Morgan, distributivity and collecting} \\ & \quad\quad \text{all summands with a factor } \neg r \;]\!\!\} \\ & ((a \sqcap b) + c + \neg q\, a + \neg p\, b + \neg p\, \neg q + \neg r\,(a + b + 1), \\ & \;(p+q)\,r) \end{array}$$

Now (eqc) and $(p+q)\,r\,\neg r\,(a+b+1) = 0$ show the claim.

□

B.4 Demonic Semantics

We have already seen that command (a,p) is feasible if and only if $p \le \ulcorner a$ and thus define the set of feasible commands as $\mathrm{F}(S) = \{(a,p) | (a,p) \in \mathrm{C}(S) \wedge p \le \ulcorner a\}$. The aim of the present section is to establish a correspondence between feasible commands and elements of the underlying semiring S. It will be used to define the demonic operators on S and is given by two mappings [GM06]

$$\begin{array}{ll} E : \mathrm{F}(S) \to S \ , & D : S \to \mathrm{F}(S) \ , \\ E((a,p)) =_{df} p\,a \ , & D(a) =_{df} (a, \ulcorner a) \ . \end{array}$$

We will abbreviate $E((a,p))$ to $E(a,p)$. This function, which would make sense even for arbitrary pairs, describes the demonic view of (a,p) that discards all input states of a for which both termination and nontermination may occur, i.e., all those characterised by $\neg p \ulcorner a$. For the resulting semiring element, no extra termination information is needed; this is reflected in the definition of D. Moreover, from the definition and (eqc) it is clear that E respects the eqivalence $\equiv$, i.e., $(a,p) \equiv (b,q) \ \Rightarrow \ E(a,p) = E(b,q)$.

Lemma B.8. *E and D are inverse to each other, i.e., $D(E(a,p)) \equiv (a,p)$ and $E(D(a)) = a$.*

The proof can be found in [MS06].

We will now give a demonic ordering and demonic operations on S for modelling total correctness. In contrast to [DMT04], where such an ordering and operations are introduced by new definitions, we can derive these using the correspondence from Lemma B.8. The demonic refinement ordering is

$$a \mathrel{|\!\!\sqsubseteq} b \Leftrightarrow_{df} D(a) \sqsubseteq D(b) \ \Leftrightarrow \ (a, \ulcorner a) \sqsubseteq (b, \ulcorner b) \ \Leftrightarrow \ \ulcorner b \leq \ulcorner a \wedge \ulcorner b\, a \leq b.$$

By (eqc) and (cd1) $\mathrel{|\!\!\sqsubseteq}$ is antisymmetric, i.e., a partial order. Thus, by Lemma B.8, the mappings E and D are order isomorphisms between $(\mathrm{F}(S), \sqsubseteq)$ and $(S, \mathrel{|\!\!\sqsubseteq})$. Since chaos is the greatest element of COM(S), and therefore also of $\mathrm{F}(S)$, the $\mathrel{|\!\!\sqsubseteq}$-greatest element of S is $E(\mathsf{chaos}) = E(\top, 0) = 0$. In general, however, there is no $\mathrel{|\!\!\sqsubseteq}$-smallest element, since the corresponding least element fail of COM(S) is not feasible.

The demonic composition is

$$\begin{array}{rl} a \mathbin{\scriptstyle\square} b =_{df} & E(D(a)\,;D(b)) = E((a, \ulcorner a)\,;(b, \ulcorner b)) = E(a\,b, \ulcorner a\,|a]\ulcorner b) \\ = & (\ulcorner a\,|a]\ulcorner b)\,a\,b = (|a]\ulcorner b)\,a\,b \ . \end{array}$$

The unit skip of COM(S) is feasible, thus $E(\mathsf{skip}) = E(1,1) = 1$ is also the unit of demonic composition.

The demonic join (which is the $\mathrel{|\!\!\sqsubseteq}$-join and coincides with demonic choice) is

$$a \mathbin{|\!\!\sqcup} b =_{df} E(D(a) \sqcup D(b)) \ = \ \ulcorner a \ulcorner b\,(a+b) \ .$$

The demonic meet, whenever it exists, is

$$a \mathbin{|\!\!\sqcap} b =_{df} E(D(a) \sqcap D(b)) \ = \ a \sqcap b + \neg\ulcorner a\,b + \neg\ulcorner b\,a \ ;$$

the necessary and sufficient condition for its existence is the feasibility of $D(a) \sqcap D(b)$, which is equivalent to $\ulcorner(a \sqcap b) = \ulcorner a \ulcorner b$ (see again [MS06]).

Now we can establish properties analogous to the ones for $\sqcup$ and $\sqcap$.

Lemma B.9. *The operations $\mathbin{|\!\!\sqcup}$ and $\mathbin{|\!\!\sqcap}$ are associative and commutative and distribute over each other.*

Proof. As a sample we deal with distributivity, assuming that all demonic meets involved exist:

$$\begin{array}{ll}
 & (a \Sqcap b) \Sqcup c \\
= & \qquad \{\![\ \text{definitions}\ ]\!\} \\
 & E(D(E(D(a) \sqcap D(b))) \sqcup D(c)) \\
= & \qquad \{\![\ D(E(k)) \equiv k \text{ and } E \text{ respects } \equiv\ ]\!\} \\
 & E((D(a) \sqcap D(b)) \sqcup D(c)) \\
= & \qquad \{\![\ \text{by Lemmas B.7 and B.6 and since } E \text{ respects } \equiv\ ]\!\} \\
 & E((D(a) \sqcup D(c)) \sqcap (D(b) \sqcup D(c))) \\
= & \qquad \{\![\ k \equiv D(E(k)) \text{ and } E \text{ respects } \equiv\ ]\!\} \\
 & E(D(E(D(a) \sqcup D(c))) \sqcap D(E(D(b) \sqcup D(c)))) \\
= & \qquad \{\![\ \text{definitions}\ ]\!\} \\
 & (a \Sqcup c) \Sqcap (b \Sqcup c).
\end{array}$$

□