

Department of Computing and Software

Faculty of Engineering — McMaster University

Formal Model Driven Approach to Deal with Requirements Volatility

by

Ridha Khedri

C.A.S. Report Series

CAS-08-03-RK

Department of Computing and Software

January 2008

Information Technology Building

McMaster University

1280 Main Street West Hamilton, Ontario, Canada L8S 4K1

Copyright © 2008, Khedri

Formal Model Driven Approach to Deal with Requirements Volatility

*

Technical Report CAS-08-03-RK
Department of Computing and Software
McMaster University

January 2008

Abstract

To deal with software requirements volatility, we present a model transformation of a software family requirements model into detailed requirements models of its members. This transformation is based on *feature algebra* and *relation algebra*. We give the mathematical foundation for this transformation system. The initial models are the result of engineering requirements processes. In other terms, they are the result of elicitation and formalization activities. A family model is represented by a feature algebra term. The features of the family are requirements scenarios formalized into relational scenarios, which are 2-tuples of relations. Throughout the transformation of the initial models into the detailed models, the consistency of the specified system as well as that of its environment are verified.

Keywords: Model driven software development, formal methods, feature algebra, relational algebra, functional requirements, requirements scenarios, software family, product line.

*Department of Computing and Software, McMaster University, khedri@mcmaster.ca

1 Introduction

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

The first solution proposes that instead of focusing our attention on a single software system to built, we take into account predictable changes. Thereby, the analysis and design of a family of software systems that share a core part (commonality among all the members) is considered. As far as we know, this solution was proposed since 1976 by David Parnas in [22] where he introduced the notion of *program family*. However, we find in [27, page 12] that family-oriented software development was suggested as early as 1968 without giving the source of this fact. Software product line engineering, which is a family-oriented software production processes and techniques, seems to be adopted by both practitioners and researchers to deal with changes in the requirements and thereby a reconsideration of the corresponding designs. The idea behind product line engineering is to take advantage of the commonality of systems that are developed for a specific domain. Weiss and Lai [27, Preface, page xvii] report that applying family-based processes at *Lucent Technologies* lead to decreases in development time and costs for family members of 60% to 70%. This observation has been endorsed by other similar reports later on.

Faulk [5] pointed 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 artifacts or paradigms related to these stages. However, one should expect that family-based software development should start at the earliest of the adopted software development process. A software development system tailored for non-monolithic software development process should take into account the modern reality of today's software: 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 that have combined software family to requirements analysis [20].

The limitation of the family-based approach to software development is captured in one of its underlying assumptions: the *oracle hypothesis* which considers that "it is possible to predict the changes that are likely to be needed to a system over its lifetime" [27, page 11]. The rapid change in the user needs and in market trends makes this hypothesis hard to consider as tenable. Then, another solution is required to deal with the unpredictable changes.

The second solution proposed in the literature to deal, among others, with 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. In other terms, this solution is to face up the affect of the rejection of the *oracle assumption*. This approach

anticipates systematic transitions from a set of initial models¹ of a software system to its executable code. However, the techniques for these proposed transitions lack formality. To trust the obtained code, these transformations need to be based on well defined syntax and semantics that are well grounded in established mathematical theories (e.g., languages, set theory, algebras, etc.).

From a historical perspective, MDE is a continuation of the initiative launched by the OMG project [1]. The focus at the beginning was around *Model Driven Architecture* (MDA). Since then, a wide area of research has been developing to investigate the evolution from initial models to an operational system. Bézivim et al. [9] indicate that since OMG proposal [1], model driven software development took different forms. However, they all share the same principle: A meta-model (an abstract model) is constructed, then any model used, within a domain of application, must be “conform” to the metamodel [9]. These models are called *derived models*.

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

We present a model transformation of a software family requirements model into detailed requirements models of its members. This transformation is based on *feature algebra* and *relation algebra*. We give the mathematical foundation for this transformation system. The initial models are the result of engineering requirements processes. In other terms, they are the result of elicitation and formalization activities. Once one reaches formal models, then formal transformations should be adopted when possible. The derived models give the specifications of both the system and the environment in which it is specified to operate. Thus, it helps in presenting exactly what the system is expected to do in reaction to stimuli from its environment.

To introduce the concepts related to the model transformation system that we propose, we present, in Section 2, the mathematical background as well as some notions taken from the literature and related to the use of relations to model requirements scenarios. In Section 3, we present our approach to requirements models transformation. In Section 4, we present a case study where we illustrate the use of the proposed model transformation system. In Section 5, we discuss the strengths and shortcomings of our formal model transformation to deal with software requirements analysis and evolution, and we point to future research work.

2 Background

2.1 A Brief Overview of Relations

For a simple presentation of our work, we present it within a homogeneous abstract relation algebra instead a heterogeneous one. The later is closer to the reality. Axiomatisations

¹The models that constitute the starting point to the MDE process are called initial models.

of both homogeneous and heterogeneous abstract relation algebra can be found in [25, pages 270–272]. The relational concepts needed to the formal background of the model transformation that we propose are elementary and can be found in any textbook on discrete mathematics. Nevertheless, we briefly recall the necessary notions. For more advanced material on relations and demonic operators, we refer the reader to [2, 6, 25].

A (binary) relation R on a set T is a subset of the Cartesian product of T with itself, that is, $R \subseteq T \times T$. Among the possible operations on relations, we select the following where Q and R are relations on a set T :

Empty relation	$\emptyset$	$\stackrel{\text{def}}{=} \{(t, t') \mid \text{false}\},$
Universal relation	$\mathbb{L}$	$\stackrel{\text{def}}{=} \{(t, t') \mid \text{true}\},$
Complement	$\overline{R}$	$\stackrel{\text{def}}{=} \{(t, t') \mid (t', t) \notin R\},$
Converse	$R^\sim$	$\stackrel{\text{def}}{=} \{(t, t') \mid (t', t) \in R\},$
Union	$Q \cup R$	$\stackrel{\text{def}}{=} \{(t, t') \mid (t, t') \in Q \vee (t, t') \in R\},$
Intersection	$Q \cap R$	$\stackrel{\text{def}}{=} \{(t, t') \mid (t, t') \in Q \wedge (t, t') \in R\},$
Composition	$Q;R$	$\stackrel{\text{def}}{=} \{(t, t'') \mid \exists(t' \mid t' \in T : (t, t') \in Q \wedge (t', t'') \in R)\},$
Domain	$\text{dom}(R)$	$\stackrel{\text{def}}{=} \{t \mid \exists(t' \mid t' \in T : (t, t') \in R)\},$

2.2 Relations as Requirements Specifications

The concept of relation is fairly simple, and relates well to software requirements as viewed by a user (mappings between inputs and outputs). Consider the following informal functional requirements scenario:

Scenario_Basic_Personal_Tasks:

Only authorized HR staff can perform this task. Once the authentication of the HR staff is established, (s)he can input the identification of the employee concerned by the changes. Changes can be made to name, address, home/work telephone number, emergency contacts, next of kin, marriage details, or births or adoption details. A change to the address or a telephone number requires the input of the date on which the change takes effect. The default effective date is the date on which the change is made. When a change is made to the marriage details, the system displays a message alerting the user that the change might affect the next of kin information.

This requirements scenario (partial description of environment-system interactions) can be formalized into the predicate **Scenario_Basic_Personal_Tasks** presented in Appendix A. This predicate is the characteristic predicate of the relation on the space formed by the Cartesian product of the types of all the variables that occur in the text of the considered scenario. For more details on how to work out the state space from informal scenarios, we refer the reader to [16, 17].

The predicate that corresponds to the formalisation of the scenario is considered as the characteristic predicate of the relation of the scenario `Scenario_Basic_Personal_Tasks`. The above scenario gives a partial description of the behavior of the system as well as that of its environment. The predicate obtained from its formalisation can be split into two predicates: One describes the expected behavior of the system according to the scenario and the other the behavior of its environment. In the following, dashed names like `Authentication'` are used to denote the values of the same components of the state after the operation (at the ending state). This notation is similar to that of **Z**. We denote by `i` the input by the user to the system that the system in its turn sets it to empty string ϵ . We denote by A an alphabet. The predicate $OnlyChge(x_i)$ indicates that we have $x'_j = x_j$ for all the other variables x_j .

The following is the predicate of the relation of the environment obtained from the informal scenario `Scenario_Basic_Personal_Tasks`.

$$\begin{aligned}
& \text{MenuOption} = \text{BasicPersonalTskUpdate} \wedge i = \epsilon \\
& \wedge (\text{Authentication} = \text{false} \wedge i' \in A^+ \vee \text{Authentication} = \text{true} \wedge i' \in A^+) \\
& \quad \wedge OnlyChge(i) \\
& \vee \text{MenuOption} = \text{UpdateData} \wedge i = \epsilon \wedge \text{EffectDateNeeded} = \text{true} \wedge i' \in A^+ \\
& \quad \wedge OnlyChge(i)
\end{aligned}$$

The relation of the system has the following characteristic predicate.

$$\begin{aligned}
& \text{MenuOption} = \text{BasicPersonalTskUpdate} \\
& \wedge \left(i \in A^+ \wedge (i' = \epsilon \wedge OnlyChge(i) \vee \text{Authentication}' = \text{true} \wedge i = \epsilon \right. \\
& \quad \left. \wedge OnlyChge(i, \text{Authentication})) \right) \\
& \vee \text{Authentication} = \text{true} \wedge i \in A^+ \wedge i \in \text{SetOfEmployees} \\
& \quad \wedge i' = \epsilon \wedge \text{MenuOption}' = \text{UpdateData} \\
& \quad \wedge OnlyChge(i, \text{MenuOption}) \\
& \vee \text{Authentication} = \text{true} \wedge i \in A^+ \wedge i \notin \text{SetOfEmployees} \wedge i' = \epsilon \\
& \quad \wedge OnlyChge(i) \Big) \\
& \vee \text{MenuOption} = \text{UpdateData} \wedge i \in A^+ \\
& \quad \wedge \left(\text{Address}' = i \wedge \text{EffectDateNeeded}' = \text{true} \wedge i' = \epsilon \right. \\
& \quad \quad \wedge OnlyChge(\text{Address}, i, \text{EffectDateNeeded}) \\
& \quad \vee \text{HomeTel}' = i \wedge \text{EffectDateNeeded}' = \text{true} \wedge i' = \epsilon \\
& \quad \quad \wedge OnlyChge(\text{HomeTel}, i, \text{EffectDateNeeded}) \\
& \quad \vee \text{WorkTel}' = i \wedge \text{EffectDateNeeded}' = \text{true} \wedge i' = \epsilon \\
& \quad \quad \left. \wedge OnlyChge(\text{WorkTel}, i, \text{EffectDateNeeded}) \right) \\
& \vee \text{MenuOption} = \text{UpdateData} \wedge \text{EffectDateNeeded} = \text{true} \wedge i \in A^+
\end{aligned}$$

$$\begin{aligned}
& \wedge \text{EffectDate}' \geq \text{CurrentDate} \wedge i' = \epsilon \wedge \text{OnlyChge}(\text{EffectDate}, i) \\
\vee & \text{MenuOption} = \text{UpdateData} \wedge i \in A^+ \\
& \wedge \left(\text{Name}' = i \wedge \text{EffectDateNeeded}' = \text{false} \wedge i' = \epsilon \right. \\
& \quad \wedge \text{OnlyChge}(\text{Name}, i, \text{EffectDateNeeded}) \\
& \vee \text{EmrgceCntcts}' = i \wedge \text{EffectDateNeeded}' = \text{false} \wedge i' = \epsilon \\
& \quad \wedge \text{OnlyChge}(\text{EmrgceCntcts}, i, \text{EffectDateNeeded}) \\
& \vee \text{NextKin}' = i \wedge \text{EffectDateNeeded}' = \text{false} \wedge i' = \epsilon \\
& \quad \wedge \text{OnlyChge}(\text{NextKin}, i, \text{EffectDateNeeded}) \\
& \vee \text{BirthAdoption}' = i \wedge \text{EffectDateNeeded}' = \text{false} \wedge i' = \epsilon \\
& \quad \wedge \text{OnlyChge}(\text{BirthAdoption}, i, \text{EffectDateNeeded}) \\
& \vee \text{MariageDtls}' = i \wedge \text{EffectDateNeeded}' = \text{false} \\
& \quad \wedge \text{MessageToDisplay}' = \text{ChangeMightAffectNextkin} \wedge i' = \epsilon \\
& \quad \left. \wedge \text{OnlyChge}(\text{MariageDtls}, \text{MessageToDisplay}, i, \text{EffectDateNeeded}) \right)
\end{aligned}$$

2.3 Relational Scenarios

The above example illustrates that a scenario can be formalized into a 3-tuple: the space of the scenario, a relation that specifies the behavior of the environment according to the scenario, and a relation that specifies the behavior of the system according to the scenario as well. The space of a scenario corresponds to the Cartesian product of the types of all the variables that occur in its formal specifications. To have a simple presentation, we omit the state space by assuming that all the scenarios have a same space. Consequently, we represent a scenario as a 2-tuple (R_e, R_s) , where R_e is its relation of the environment and R_s is its relation of the system. We note that for any scenario, we have $R_e \cap R_s = \emptyset$ and the relation of the scenario is $R_e \cup R_s$. In [17, 16], (R_e, R_s) is called a *relational scenario*. For more details on this technique of formalizing scenarios using relations, we refer the reader to [13, 14, 16, 17]. When the functional requirements of a system or that of a family is given in term of scenarios, one should expect inconsistency among the given scenarios. In [17, 16], a technique based on relational algebra is proposed to detect inconsistency among scenarios and to integrate them when they are consistent. The technique enables verification for dictionary inconsistency (i.e., naming inconsistency) and more importantly verification of behavior (functional) inconsistency. The latest verification can be automated and a prototype tool is presented in the literature [4, 12, 15].

2.4 A Brief Overview of Feature Algebra

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

Definition 2.1 ([8, 7]). A *semiring* is a quintuple $(S, +, 0, \cdot, 1)$ such that $(S, +, 0)$ is a commutative monoid and $(S, \cdot, 1)$ is a monoid such that $\cdot$ distributes over $+$ and 0 is an annihilator, i.e., $0 \cdot a = 0 = a \cdot 0$. The semiring is *commutative* if $\cdot$ is commutative and it is *idempotent* if $+$ is idempotent, i.e., $a + a = a$. In the latter case the relation $a \leq b \stackrel{\text{def}}{\iff} a + b = b$ 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 software family specification, $+$ can be interpreted as a choice between options of products and features and $\cdot$ as their composition or mandatory presence. In [7, 8], an idempotent commutative semiring is called a *feature algebra*. Its elements are termed *product families* and can be considered as abstractly representing sets of products each of which is composed of a number of features.

In the literature, terms like product family and subfamily lack exact definition. In [8, 7], we find the algebraic definitions for these terms. Also, within specific models (such as set-based model and bag-based model), they provide the corresponding definitions of these notions in terms of these particular models and not in general algebraic terms. In the remainder let $F = (S, +, 0, \cdot, 1)$ be a feature algebra. We note that Boolean algebras and lattices are feature algebras.

Definition 2.2 ([8, 7]). An element a is said to be a *product*, if $a \neq 0$ and

$$\begin{aligned} & \forall(b \mid b \leq a \implies b = 0 \vee b = a) \\ & \wedge \forall(b, c \mid a \leq b + c \implies (a \leq b \vee a \leq c)) . \end{aligned} \quad (1)$$

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.

Analogously to Definition 2.2, indivisibility is required, but this time w.r.t. multiplication rather than addition.

Definition 2.3 ([8, 7]). An element a is called *feature* if it is a product and

$$\begin{aligned} & \forall(b \mid b \div a \implies b = 0 \vee b = a) \\ & \wedge \forall(b, c \mid a \div (b \cdot c) \implies (a \div b \vee a \div c)) , \end{aligned} \quad (2)$$

where the divisibility relation $\div$ is given by $x \div y \stackrel{\text{def}}{\iff} \exists(z \mid x = y \cdot z)$.

On every feature algebra we can define a relation that expresses that one product family refines another in a certain sense.

Definition 2.4 ([8, 7]). The *refinement relation* $\sqsubseteq_f$ on a feature algebra is defined as

$$a \sqsubseteq_f b \stackrel{\text{def}}{\iff} \exists(c \mid c \in S : a \leq b \cdot c) ,$$

where $\leq$ is the natural order on S .

The refinement relation is a preorder, i.e., a reflexive and transitive relation. Roughly speaking, $a \sqsubseteq_f b$ means that every product in a has at least all the features of some product in b , but possibly additional ones. In [7, 8] other notions and similarity measures among families are defined and discussed. We list the notions of *generated products*, *k-near similarity* of two families, *weak-zero*, and *sub-family*. In [7, 8], the authors propose some useful properties of feature algebras concerning finding common features, building up product families, finding new products and excluding special feature combinations. Finding the commonalities of a given set of products which is a very relevant issue since the identification of common artifacts within systems (e.g. chips, software modules, etc.) enhances hardware/software reuse. Within feature algebras like the set-based model and the bag-based model, this problem can be formalized as finding “the greatest common divisor” or to factor out the features common to all given products. It is a direct use of “classical” algorithms which shows an advantage of using an algebraic approach. Solving gcd (greatest common divisor) is well known and easy, whereas finding commonalities using diagrams (e.g., FODA [10]) or trees (e.g., FORM [11]) is more complex.

2.5 Demonic Meet and Demonic Join

In our approach, features are relational scenarios. Hence, to be able to elaborate further on the operations on relational scenarios in order to integrate them and detect inconsistency among them, we introduce two relational operation: the demonic join $\sqcup$, and its dual the demonic meet $\sqcap$.

Definition 2.5 ([3, 21]). We say that a relation Q *refines* a relation R , denoted by $Q \sqsubseteq_r R$, iff $Q \cap R : \mathbb{L} \subseteq R : \mathbb{L} \wedge R : \mathbb{L} \subseteq Q : \mathbb{L}$, or, equivalently, iff $Q \cup \overline{Q} : \mathbb{L} \subseteq R \cup \overline{R} : \mathbb{L} \wedge \overline{Q} : \mathbb{L} \subseteq \overline{R} : \mathbb{L}$.

The relation $\sqsubseteq_r$ is a partial ordering [3]. Let $\mathcal{B}_R$ be a set of relations. With respect to $\sqsubseteq_r$, a join operation on $(\mathcal{B}_R, \sqsubseteq_r)$ is defined. The structure $(\mathcal{B}_R, \sqsubseteq_r)$ is a complete join semilattice. In Appendix C, Proposition C.1 articulates the general definition of the join operation, however for the need of this report we present only the definition of the join of two relations R and Q :

$$Q \sqcup R = (Q \cup R) \cap Q : \mathbb{L} \cap R : \mathbb{L} \quad (3)$$

Also, we have

$$(Q \sqcup R) : \mathbb{L} = Q : \mathbb{L} \cap R : \mathbb{L} \quad (4)$$

Example 2.1. If we take the relations $Q = \{(0, 3), (2, 5)\}$ and $R = \{(0, 3), (1, 5)\}$, then $Q \sqcup R = \{(0, 3)\}$. For the relations $S = \{(0, 3), (0, 4), (1, 5)\}$ and $T = \{(0, 5), (0, 6), (2, 5)\}$, we have $S \sqcup T = \{(0, 3), (0, 4), (0, 5), (0, 6)\}$. So, $(a, b) \in S \sqcup T \iff ((a, b) \in S \vee (a, b) \in T) \wedge a \in \text{dom}(S) \wedge a \in \text{dom}(T)$.

On a complete join semilattice $(\mathcal{B}_R, \sqsubseteq_r)$, the greatest lower bound of the set of relations (where X is a set of indices) $\{R_i \mid i \in X\}$ exists under a special condition. We give in Appendix C (Proposition C.2) the general condition under which the greatest lower bound

$\sqcap$ exists. The operator $\sqcap$ is called *demonic meet*. A special case of the definition of the demonic meet of two relations is the following:

Corollary 1 ([16]). *Let Q and R be two relations. The demonic meet of Q and R is defined as follows:*

$$\begin{aligned} Q \sqcap R &\stackrel{\text{def}}{=} (Q \cup \overline{Q}:\mathbb{L}) \cap (R \cup \overline{R}:\mathbb{L}) \cap (Q:\mathbb{L} \cap R:\mathbb{L}) \\ &= Q \cap R \cup Q \cap \overline{R}:\mathbb{L} \cup R \cap \overline{Q}:\mathbb{L} \\ &\text{defined if } \mathbb{L} \subseteq ((Q \cup \overline{Q}:\mathbb{L}) \cap (R \cup \overline{R}:\mathbb{L})):\mathbb{L}. \end{aligned}$$

The definiteness condition can be written as follows:

$$\mathbb{L} \subseteq ((Q \cup \overline{Q}:\mathbb{L}) \cap (R \cup \overline{R}:\mathbb{L})):\mathbb{L} \iff Q:\mathbb{L} \cap R:\mathbb{L} = (Q \cap R):\mathbb{L} \quad (5)$$

The above condition means as well $\text{dom}(Q \cap R) \subseteq \text{dom}(Q) \cap \text{dom}(R)$. It indicates that, for any argument in their common domain, Q and R must have at least one image in common. Thus, the relations Q and R of Example 2.1 satisfy the condition because, on their common domain (i.e., $\{0\}$), they agree on result 3. The demonic meet of Q and R is then $Q \sqcap R = \{(0, 3), (1, 5), (2, 5)\}$. For the relations S and T , the condition $\text{dom}(S) \cap \text{dom}(T) = \text{dom}(S \cap T)$ is not satisfied (the relations do not agree on a common result for input 0) and their demonic meet is not defined. If we take S and T for two specifications, it means that the specifications contradict each other on the result for input 0 and cannot be satisfied together. In other terms, the consistency condition between two relations is the condition under which they can be refined by a common relation. For more information on the use of demonic meet to detect inconsistency in scenarios, we refer the reader to [17, 16]. Simple cases worth noticing regarding the demonic meet operator are given, in Appendix C, by Proposition C.3.

3 Formal Transformation of Initial Requirements Models

In the literature, we find various relationships between features and requirements. For instance IEEE standard [26, page 19] gives among others the following relationship between requirements and features:

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

Therefore, a feature is described using scenarios (system-environment interactions). Also, in [24], Savolainen et al. write “A feature is specified by a set of requirements; this set may contain one or more requirements”. Hence, features can be expressed using a requirements

scenario. Since a scenario is a partial description of a system-environment interactions subjectively presented, therefore what constitutes several scenarios to one specifier can be seen as one scenario by another specifier. We will introduce, in Definition 3.3, a notion of atomic relational scenario with regard to a system specification. In summary, at the requirements level a feature encapsulates a set of related system-environment interactions. We will as well introduce, in Definition 3.2, the notion of a *genuine product*. The family model gives us only potential products. Some products are possible in the family model but they are formed by inconsistent scenarios that cannot be satisfied together.

Our objective here is to move from a *feature algebraic* model of a family (which is the feature model given as a feature algebra term) into a more concrete model of its members, of the commonality of its members, or of a simple combination of requirements features. First, we need to relate formal scenarios to feature algebra by providing concrete definitions for the two feature algebra operators $\cdot$ and $+$, and provide explicit 0 and 1.

Let $(\mathcal{B}_e, \sqsubseteq_r)$ (resp., $(\mathcal{B}_s, \sqsubseteq_r)$) be the semilattice ordered by $\sqsubseteq_r$. Let $C_{\mathcal{E}} = \{Q_i \mid i \in X\} \subseteq \mathcal{B}_e$ (resp., $C_{\mathcal{S}} = \{R_i \mid i \in Y\} \subseteq \mathcal{B}_s$). We assume the following:

Assumption 3.1.

1. $\forall (i, j \mid (i \in X \wedge j \in X) \vee (i \in Y \wedge j \in Y)) : R_i \cdot \mathbb{L} \cap R_j \cdot \mathbb{L} = (R_i \cap R_j) \cdot \mathbb{L}$
2. $\sqcap$ is closed over $C_{\mathcal{E}}$ and over $C_{\mathcal{S}}$.

Assumption 3.1(1) states that the demonic meet of two relations in $C_{\mathcal{E}}$ (resp. $C_{\mathcal{S}}$) is defined (i.e., the relations are consistent) and assumption 3.1(2) indicates that the result belongs to $C_{\mathcal{E}}$ (resp. $C_{\mathcal{S}}$). We denote by $1_{C_{\mathcal{E}}}$ (resp. $1_{C_{\mathcal{S}}}$) the supremum of $C_{\mathcal{E}}$ (resp. $C_{\mathcal{S}}$), and by $0_{C_{\mathcal{E}}}$ (resp. $0_{C_{\mathcal{S}}}$) the infimum of $C_{\mathcal{E}}$ (resp. $C_{\mathcal{S}}$). These element exist due Assumptions 3.1. Let $\mathcal{C} \stackrel{\text{def}}{=} C_{\mathcal{E}} \times C_{\mathcal{S}}$. We define the following relation on $\mathcal{C}$.

Definition 3.1 (Relational Scenario ordering). For every (Q_e, Q_s) and (R_e, R_s) in $\mathcal{C}$, we define

$$(Q_e, Q_s) \sqsubseteq_{sc} (R_e, R_s) \stackrel{\text{def}}{\iff} Q_e \sqsubseteq_r R_e \wedge R_s \sqsubseteq_r Q_s$$

We say that relational scenario (Q_e, Q_s) refines the relational scenario (R_e, R_s) .

Proposition 3.1. *The relation $\sqsubseteq_{sc}$ on $\mathcal{C}$ is an order (or partial order).*

Proof. • $\sqsubseteq_{sc}$ is reflexive due to the reflexivity of $\sqsubseteq_r$.

- $\sqsubseteq_{sc}$ is antisymmetric due to the antisymmetry of $\sqsubseteq_r$.

$$\begin{aligned}
& (Q_e, Q_s) \sqsubseteq_{sc} (R_e, R_s) \wedge (R_e, R_s) \sqsubseteq_{sc} (Q_e, Q_s) \\
& \iff \langle \text{Definition 3.1} \rangle \\
& \quad Q_e \sqsubseteq_r R_e \wedge R_e \sqsubseteq_r Q_e \wedge R_s \sqsubseteq_r Q_s \wedge Q_s \sqsubseteq_r R_s \\
& \implies \langle \sqsubseteq_{sc} \text{ is antisymmetric} \rangle \\
& \quad Q_e = R_e \wedge R_s = Q_s \\
& \iff \langle \text{Tuple equality definition} \rangle \\
& \quad (Q_e, Q_s) = (R_e, R_s)
\end{aligned}$$

- $\sqsubseteq_{sc}$ is transitive due to the transitivity of $\sqsubseteq_r$.

□

Proposition 3.2. *The structure $(C_{\mathcal{E}} \times C_{\mathcal{S}}, \oplus, \odot, 0_{sc}, 1_{sc})$ is a feature algebra, where for Q_e and R_e elements of $C_{\mathcal{E}}$, and Q_s and R_s elements of $C_{\mathcal{S}}$, we have:*

- $(Q_e, Q_s) \oplus (R_e, R_s) \stackrel{\text{def}}{=} (Q_e \sqcap R_e, Q_s \sqcup R_s),$
- $(Q_e, Q_s) \odot (R_e, R_s) \stackrel{\text{def}}{=} (Q_e \sqcup R_e, Q_s \sqcap R_s),$
- $0_{sc} \stackrel{\text{def}}{=} (1_{C_{\mathcal{E}}}, 0_{C_{\mathcal{S}}}),$
- $1_{sc} \stackrel{\text{def}}{=} (0_{C_{\mathcal{E}}}, 1_{C_{\mathcal{S}}}),$

Proof. The $\sqcup$ of two relation is always defined. Due to Assumptions 3.1, the $\sqcap$ of two relations that both belong to $C_{\mathcal{E}}$ or $C_{\mathcal{S}}$ is defined. Hence, the $(Q_e, Q_s) \oplus (R_e, R_s)$ and $(Q_e, Q_s) \odot (R_e, R_s)$ exist for all (Q_e, Q_s) and (R_e, R_s) that belong to $C_{\mathcal{E}} \times C_{\mathcal{S}}$. It follows that $(C_{\mathcal{E}} \times C_{\mathcal{S}}, \oplus, \odot, 0_{sc}, 1_{sc})$ is a lattice. Every lattice is a feature algebra. □

In other terms, $(C_{\mathcal{E}} \times C_{\mathcal{S}}, \oplus, \odot, 0_{sc}, 1_{sc})$ is a model² of feature algebra.

If we take two formal scenarios (Q_e, Q_s) and (R_e, R_s) , the choice between them is the formal scenario that takes the demonic meet of their corresponding relations of the environment and the demonic join of their corresponding relations of the system. To offer two scenarios as alternatives (choice), their environments need to be consistent; otherwise they should not be taken as alternatives. The environment of an alternative feature could inhibit the normal behaviour of the other alternative. To illustrate this point in common terms, when we state that a person has two alternative means of transportation (car and bicycle) to travel from home to her work place, the environment of one (for instance, roads needed for a car) should allow the use of the other (bicycle). Otherwise, there is no real alternatives. This is why we take the demonic meet of the relations of the environment. On the system side, their $\oplus$ -integration takes only their demonic join, which consists into the union on their common domain.

Concerning the $\odot$ operation, the two argument scenarios are mandatory. Therefore, their common environment is represented by the demonic join of their relations of the environment. If the environment of a relational scenario forbids certain behaviours of the environment of another relational scenario involved in the $\odot$ -integration, then we need to take account of that by taking their demonic join. On system part, we require that the behaviour of the system as described by both scenarios must be consistent (checked by the definiteness of the demonic meet –see Corollary 1–). When the relations of the system of the $\odot$ -integrated scenarios are consistent, then we take their demonic meet as the behaviour of the system as described by both scenarios.

²In the mathematical sens i.e., Let $\mathcal{L}$ be a language. An $\mathcal{L}$ -theory T is simply a set of $\mathcal{L}$ -sentences. We say that $\mathcal{M}$ is a *model* of T if $\mathcal{M}$ satisfies Φ (or Φ is true in $\mathcal{M}$) for all sentences $\Phi \in T$. [19]

The relational scenario 0_{SC} specifies a system that involves all the consistent relations of the system corresponding to all the scenarios given to us (i.e., all the $R_s \in C_S$). Its environment relation can be refined by all the relations of the environment of all the scenarios. Moreover, we have $0_{\text{SC}} \sqsubseteq_{sc} (Q_e, Q_s)$, for every $(Q_e, Q_s) \in C_{\mathcal{E}} \times C_S$.

The relational scenario 1_{SC} can be seen as the closed system that can be built from all the given scenarios. Its environment does not have any affect on any environment of the given relational scenarios.

Example 3.1. Suppose that we have $\text{sc}_1 \cdot \text{sc}_2 \cdot (1_{\text{SC}} + \text{sc}_3)$ as a term specifying a family, where sc_i , for $i \in 1 \dots 3$ are formal scenarios. The members have in common $\text{sc}_1 \cdot \text{sc}_2$. The two members can be extracted by algebraic manipulations as follows:

$$\begin{aligned} & \text{sc}_1 \cdot \text{sc}_2 \cdot (1_{\text{SC}} + \text{sc}_3) \\ = & \quad \langle \text{Distributivity of } \cdot \text{ over } + \quad \& \quad 1_{\text{SC}} \text{ is the identity for } \cdot \rangle \\ & \text{sc}_1 \cdot \text{sc}_2 + \text{sc}_1 \cdot \text{sc}_2 \cdot \text{sc}_3 \end{aligned}$$

To obtain the specification of the family where $\text{sc}_i = (R_{e_i}, R_{s_i})$, we proceed as follows:

$$\begin{aligned} & \text{sc}_1 \cdot \text{sc}_2 \cdot (1_{\text{SC}} + \text{sc}_3) \\ = & \quad \langle \text{In this case } 1_{\text{SC}} \stackrel{\text{def}}{=} (\sqcap_{i \in 1..3} R_{e_i}, \sqcup_{i \in 1..3} R_{s_i}) \quad \& \quad \text{Definition of } \text{sc}_i \rangle \\ & (R_{e_1}, R_{s_1}) \odot (R_{e_2}, R_{s_2}) \odot ((\sqcap_{i \in 1..3} R_{e_i}, \sqcup_{i \in 1..3} R_{s_i}) \oplus (R_{e_3}, R_{s_3})) \\ = & \quad \langle \text{Definition of } \oplus \rangle \\ & (R_{e_1}, R_{s_1}) \odot (R_{e_2}, R_{s_2}) \odot (\sqcap_{i \in 1..3} R_{e_i} \sqcap R_{e_3}, \sqcup_{i \in 1..3} R_{s_i} \sqcup R_{s_3}) \\ & \quad \langle \text{Definition of } \odot \text{ (twice)} \rangle \\ & (R_{e_1} \sqcup R_{e_2} \sqcup (\sqcap_{i \in 1..3} R_{e_i} \sqcap R_{e_3}), R_{s_1} \sqcap R_{s_2} \sqcap (\sqcup_{i \in 1..3} R_{s_i} \sqcup R_{s_3})) \\ & \quad \langle \text{Idempotence of } \sqcap \text{ and } \sqcup \rangle \\ & (R_{e_1} \sqcup R_{e_2} \sqcup (R_{e_1} \sqcap R_{e_2} \sqcap R_{e_3}), R_{s_1} \sqcap R_{s_2} \sqcap (R_{s_1} \sqcup R_{s_2} \sqcup R_{s_3})) \end{aligned}$$

Hence, knowing R_{e_i} and R_{s_i} for $i \in 1 \dots 3$, we can compute the relation of the environment (i.e., $R_{e_1} \sqcup R_{e_2} \sqcup (R_{e_1} \sqcap R_{e_2} \sqcap R_{e_3})$) and the relation of the system (i.e., $R_{s_1} \sqcap R_{s_2} \sqcap (R_{s_1} \sqcup R_{s_2} \sqcup R_{s_3})$) of this family.

The two following definitions follow from their corresponding in feature algebra.

Definition 3.2. An element (Q_e, Q_s) is said to be a *genuine product*, if $(Q_e, Q_s) \neq (1_{C_{\mathcal{E}}}, 0_{C_S})$ and

$$\begin{aligned} & \forall ((R_e, R_s) \mid R_e \in C_{\mathcal{E}} \wedge R_s \in C_S : \\ & \quad (R_e, R_s) \sqsubseteq_{sc} (Q_e, Q_s) \implies (R_e, R_s) = (1_{C_{\mathcal{E}}}, 0_{C_S}) \vee (R_e, R_s) = (Q_e, Q_s)) \\ & \wedge \quad \forall ((R_e, R_s), (S_e, S_s) \mid R_e, S_e \in C_{\mathcal{E}} \wedge R_s, S_s \in C_S : \\ & \quad (Q_e, Q_s) \sqsubseteq_{sc} ((R_e, R_s) \oplus (S_e, S_s)) \\ & \quad \implies ((Q_e, Q_s) \sqsubseteq_{sc} (R_e, R_s) \vee (Q_e, Q_s) \sqsubseteq_{sc} (S_e, S_s))) . \end{aligned}$$

Therefore, a product cannot contain alternative scenarios. It is solely formed by the $\odot$ -integration of mandatory scenarios.

Definition 3.3. A relational scenario (Q_e, Q_s) is said to be *atomic* if it is a product and

$$\begin{aligned} & \forall((R_e, R_s) \mid R_e \in C_{\mathcal{E}} \wedge R_s \in C_{\mathcal{S}} : \\ & \quad (R_e, R_s) \div (Q_e, Q_s) \implies (R_e, R_s) = (1_{C_{\mathcal{E}}}, 0_{C_{\mathcal{S}}}) \vee (R_e, R_s) = (Q_e, Q_s)) \\ & \wedge \forall((R_e, R_s), (S_e, S_s) \mid R_e, S_e \in C_{\mathcal{E}} \wedge R_s, S_s \in C_{\mathcal{S}} : \\ & \quad (Q_e, Q_s) \div (R_e, R_s) \odot (S_e, S_s) \\ & \quad \implies ((Q_e, Q_s) \div (R_e, R_s) \vee (Q_e, Q_s) \div (S_e, S_s)) , \end{aligned}$$

where the divisibility relation $\div$ is given by $(Q_e, Q_s) \div (R_e, R_s) \stackrel{\text{def}}{\iff} \exists((S_e, S_s) \mid S_e \in C_{\mathcal{E}} \wedge S_s \in C_{\mathcal{S}} : (Q_e, Q_s) = (R_e, R_s) \odot (S_e, S_s))$.

Definition 3.3 allows to verify, when we are given a set of scenarios, whether there are some that can be obtained from a $\odot$ -integration of the other given scenarios. In other terms, it allows to verify redundancy in requirements. In the model-transformation presented in Figure 1, the set of initial scenario-models can be and should be kept without redundancy.

The model-transformation that we propose is illustrated in Figure 1. It allows a synchronization and update of the various derived requirements models each time there is a change in the initial models. The change can be a partial modification in an initial model or the removal or addition of an initial scenario model (discussed in Subsection 2.3). The derived models should be computation independent that give an abstract view of the system. A derived model does not show details of the structure of systems. In [1, page 3], we find that “a model is a formal specification of the function, structure and/or behavior of a system”. We adopt this understanding of a model, which is also consistent with Parnas’ perception of a model. Indeed, in [23], he indicates that a model is a simplified version of a product that has some, but not all, the properties of the real product and that are easier to study than the actual product.

4 Illustrative Example

The following example is adapted from a *Human Resource System Requirements Specification* that can be found in [18]. The reader should notice that some collections of features are essential, while other are not. For instance, “Internet/intranet enabled ESS systems” set of features is essential to all member of the family; we point to it by placing “Essential” beside it.

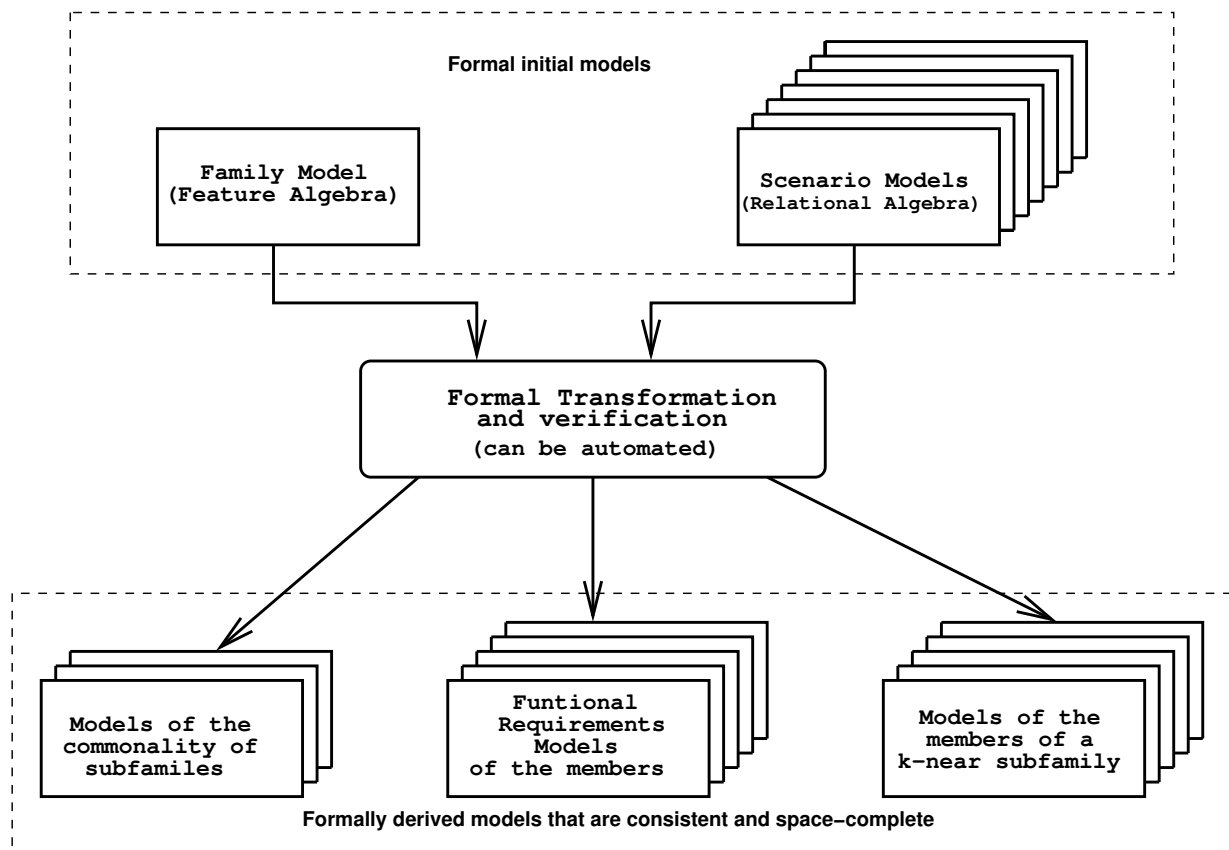


Figure 1: A Formal Transformation of Initial Requirements Models (a combination of family-oriented and model-driven approaches to software production)

4.1 Informal Specification of a Family

Employee Self Service (ESS)

Internet/intranet enabled ESS systems (Essential)

Scenario.Personal.Info:

Employees view and maintain their own information, real time, 24/7, from any location world wide.

Scenario.Personal.Info.Flexibility:

Flexibility as to which tasks may be delegated to managers and which tasks to employees.

ESS - basic personal tasks (Essential)**Scenario_Basic_Personal_Tasks:**

Routine HR administration tasks such as changing name, address, home/work telephone number, emergency contacts, next of kin, marriage, births or adoption details.

Scenario_Update_Personal_Info:

Common information uploaded to existing databases eg telephone directories and organisation charts.

Scenario_Employment_History:

View employees own employment history e.g., promotions, dates, appraisal reviews, training records.

Scenario_HR_Policies:

View conditions of employment, HR policies, procedures, information packs, benefits details, guidance templates for users changing personal information.

ESS - time and attendance tasks (Essential)**Scenario_Time_Management:**

Enter time sheets, plus with workflow, automatically submit request to manager for approval and once approved, automatically posted to payroll.

Scenario_Absence_Management:

Manage absences from work

Scenario_Absence_Information:

Enter absence details. Input sickness details. Submit time taken off for any other reason e.g., unpaid leave, jury duty.

Scenario_Absence_Calendar:

View absence calendar.

Scenario_Holiday_Entitlement: View holiday entitlement and days taken.

Scenario_Holiday_Administration:

Enter holiday requests, plus with workflow enabled, automatically submitting to manager for approval.

ESS - expenses tasks**Scenario_Expenses_Tasks:**

Enter expenses, plus with workflow, automatically submit request to manager for approval and once approved, automatically post for payment.

ESS - payroll and benefits tasks

Scenario_Payroll_Administration:

Routine payroll administration tasks such as changing bank accounts (in which to receive net pay). Employee access to own pay and benefits information, current transactions, and history from a single page.

Scenario_Benefit_Display: (Alternative to Scenario_Multi_Benefit_Programs)

Employees to view benefits information on summary level pages with quick navigation to more detailed information.

Scenario_Multi_Benefit_Programs: (Alternative to Scenario_Benefit_Display)

Build and manage a comprehensive benefits system tailored to company or individual employee needs, including multiple benefit programmes, flexible spending, automated enrollment.

Table 4.1 summarizes the general structure of the family.

4.2 Formal Specification of a Family

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

$$\begin{aligned}
 \text{ESS} = & \text{Scenario_Personal_Info} \odot \text{Scenario_Personal_Info_Flexibility} \\
 & \odot \text{Scenario_Basic_Personal_Tasks} \odot \text{Scenario_Update_Personal_Info} \\
 & \odot \text{Scenario_Employment_History} \odot \text{Scenario_HR_Policies} \\
 & \odot (1 \oplus \text{Scenario_Time_Management}) \odot (1 \oplus \text{Scenario_Absence_Management}) \\
 & \odot (1 \oplus \text{Scenario_Absence_Information}) \odot (1 \oplus \text{Scenario_Absence_Calender}) \\
 & \odot (1 \oplus \text{Scenario_Holiday_Entitlement}) \odot (1 \oplus \text{Scenario_Holiday_Administration}) \\
 & \odot (1 \oplus \text{Scenario_Expenses_Tasks}) \odot (1 \oplus \text{Scenario_Payroll_Administration}) \\
 & \odot (\text{Scenario_Benefit_Display} \oplus \text{Scenario_Multi_Benefit_Programs})
 \end{aligned}$$

In Appendix B, we provide a structured specification of the family that can be directly loaded to our tool³. Using it, we find that this family contains 432 potential members. Some of them are very likely not going to be genuine products due to possible scenario inconsistency.

Also, each of the above features is formalise into a relational scenario as illustrated in Section 2.2. The above feature algebra term and the set of formal scenarios constitute the initial models.

³The tool and a short description can be found at: http://www.informatik.uni-augsburg.de/lehrestuehle/dbis/pmi/publications/all_pmi_tech-reports/tr-2006-4_hoe_khe_moe/.

Table 1: Employee Self Service (ESS) Software Family

Collection of Features	Features	Ess.1/Opt./Alt.
Internet/intranet enabled ESS systems	1. Scenario.Personal_Info 2. Scenario.Personal_Info_Flexibility	1. Essential 2. Essential
ESS - basic personal tasks	1. Scenario.Basic_Personal_Tasks 2. Scenario.Update_Personal_Info 3. Scenario.Employment_History 4. Scenario.HR_Policies	1. Essential 2. Essential 3. Essential 4. Essential
ESS - time and attendance tasks	1. Scenario.Time_Management 2. Scenario.Absence_Management 3. Scenario.Absence_Information 4. Scenario.Absence_Calender 5. Scenario.Holiday_Entitlement 6. Scenario.Holiday_Administration	1. Optional 2. Optional 3. Optional 4. Optional 5. Optional 6. Optional
ESS - expenses tasks	1. Scenario.Expenses_Tasks	1. Optional
ESS - payroll and benefits tasks	1. Scenario.Payroll_Administration 2. Either Scenario.Benefit_Display or Scenario.Multi_Benefit_Programs	1. Optional 2. Alternatives

4.3 Model Transformations

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

- Identify the product (according to the understanding given by Definition 2.2) that is common to all the members of the family. This product is formed by the $\odot$ -integration of all the common features to all the members. Using the prototype tool described in [8], we get that the following commonality of the family.

Scenario_Basic_Personal_Tasks $\odot$ Scenario_Employment_History
 $\odot$ Scenario_HR_Policies $\odot$ Scenario_Personal_Info
 $\odot$ Scenario_Personal_Info_Flexibility $\odot$ Scenario_Update_Personal_Info

For more details, we refer the reader to Appendix B .

- Substitute each of the scenarios that occur in the above commonality specification by its corresponding relational scenarios to perform, if it is possible, the $\odot$ -integration (i.e., when all the relations of the system of all the scenarios are consistent). We note that $\odot$ -integration is associative and therefore the order under which we integrate the scenario does not matter.

In a same way, we can build the specification of each of the 432 potential members of this example family, when that is possible. Also, the specification of any sub-family can be generated in the same way.

5 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 commonality of family or any of its sub-families. The family-model is a feature algebra term. Each of the other initial models is a relational scenario that captures the environment-system interactions. Obtaining the model of a member is done through $\odot$ -integration of all its relational scenarios. Through this integration, inconsistency can be detected when we verify the definiteness of the demonic meet of the relations of the system of the scenarios forming the member. When a concrete model of a family is derived, an additional verification is performed on the consistency of alternative features (relational scenarios). Indeed, $\oplus$ -integration requires that the environments of alternative formal scenarios need to be consistent; otherwise they cannot be taken as alternative. 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 contributions to model driven development of the requirements that we have reported in this report are two-fold. First, we set up a mathematical background for formal derivation of high level requirements to more a concrete level. Second, our approach deals with expected changes through the adoption of a family approach and deals as well with unexpected changes by having a mathematical setting that enhance automation. Indeed, prototype tools exist for the $\odot$ -integration of formal scenario [15] and for the manipulation of feature algebra terms and the verification family-models. A change to a software family touches either the feature algebra term that specifies it or its set of relational 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 derived models are possible but could be not viable due to these constraints.

As a software family develops gradually, some of the scenario initial models cease to be used. These models are called *obsolete requirements* in [24]. These scenario model can be

detected automatically. The feature algebra term can be factorized to obtain the atomic relational scenarios needed, which need to be kept in the set of initial models and all the obsolete scenarios should be removed.

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 Definition 3.3, which enable to verify whether a scenario can be formed by others already existing. The approach enhances consistency verification not only of the system behaviour but also of the system environment. For instance, if two scenarios are alternative, the proposed approach enables to verify whether their respective required environments are consistent or not. The consistency verification can be automated [15], which enhances the scalability of the approach since many of the tasks of the verification are repetitive and require 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 [13] 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 [14]. It constitutes the specification of the tester; the tester needs to act according to the relation of the environment.

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.

Acknowledgments

The report benefited from discussions with Imen Bourguiba. She also has made contributions by pointing to some relevant papers that I used in the review of the literature.

References

- [1] Model Driven Architecture: A Technical Perspective (Document Number ab/2001-02-04). <http://www.omg.org/docs/ab/01-02-04.pdf>, 21 February 2001. (accessed May 04, 2007).
- [2] C. Brink, W. Kahl, and G. Schmidt, editors. *Relational Methods in Computer Science*. Springer, 1997.
- [3] Jules Desharnais, Nadir Belkhiter, Salah Ben Mohamed Sghaier, Fairouz Tchier, Ali Jaoua, Ali Mili, and Nejib Zaguia. Embedding a demonic semilattice in a relation algebra. *Theoretical Computer Science*, 149(2):333–360, 1995. Also: Research Report DIUL-RR-9302, Département d’Informatique, Université Laval, Canada, 1993.
- [4] Jules Desharnais, Ridha Khedri, and Ali Mili. Representation, validation and integration of scenarios using tabular expressions. *Formal Methods in System Design*, page 40, 2005. To appear.
- [5] Stuart R. Faulk. Product-line requirements specification (prs): An approach and case study. In *5th IEEE International Symposium on Requirements Engineering (RE 2001)*, pages 48–55, Toronto, Canada, August 27–31 2001. IEEE Computer Society.
- [6] M. Frappier, A. Mili, and J. Desharnais. Program construction by parts. *Sci. Comput. Programming*, 26(1–3):237–254, May 1996.
- [7] Peter Höfner, Ridha Khedri, and Bernhard Möller. Feature algebra. Technical Report Report 2006-04, Institut für Informatik, Universität Augsburg, February 2006. (accessed November 1, 2006).
- [8] Peter Höfner, Ridha Khedri, and Bernhard Möller. Feature algebras. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science series*, pages 300–315, 14th International Symposium on Formal Methods, McMaster University, Hamilton, Ontario, Canada, August 21 - 27 2006. Springer.
- [9] Jean Bézivin and Michaël Barbero and Frédéric Jouault. On the Applicability Scope of Model Driven Engineering. In João M. Fernandes, Ricardo J. Machado, Ridha Khedri, and Siobhán Clarke, editors, *The Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*, pages 3–7. IEEE Computer Society, 31 March 2007.
- [10] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon Software Engineering Institute, Carnegie Mellon University, 1990.

- [11] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.
- [12] Ridha Khedri. Sequential scenarios verification and integration using tabular expressions. CRL Report 374, Communications Research Laboratory, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, June 1999.
- [13] Ridha Khedri and Imen Bourguiba. Formal derivation of functional architectural design. In *2nd IEEE International Conference on Software Engineering and Formal Methods*, pages 356–365, Beijing, China, September 26–30 2004. IEEE Computer Society Press.
- [14] Ridha Khedri and Imen Bourguiba. Requirements scenarios based system-testing. In *Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'04)*, pages 252–257, Banff, Alberta, Canada, June 20–24 2004. Knowledge Systems Institute Graduate School. ISBN 1-891706-14-4.
- [15] Ridha Khedri, Rong Wu, and Bahati Sanga. SCENATOR: a prototype tool for requirements inconsistency detection. In Farn Wang and Insup Lee, editors, *Proceedings of the 1st International Workshop on Automated Technology for Verification and Analysis*, pages 75–86, Taiwan, Republic of China, December 10–13 2003. National Taiwan University.
- [16] Jules Desharnais, Marc Frappier, Ridha Khedri, and Ali Mili. Integration of sequential scenarios. *IEEE Transactions on Software Engineering*, 24(9):695–708, September 1998.
- [17] Jules Desharnais, Marc Frappier, Ridha Khedri, and Ali Mili. Integration of sequential scenarios. In M. Jazayeri and H. Schauer, editors, *Lecture Notes in Computer Science 1301*, pages 310–326, Zurich, Suisse, September 1997. Software Engineering-ESEC/FSE'97, Springer-Verlag. (Preliminary version of paper # 16).
- [18] Axia Consulting Limited. Human resource system requirements specification page. http://www.axia-consulting.co.uk/html/software_requirements_specification_example.html, 2007. (accessed May 17, 2007).
- [19] David Marker. *Model Theory: An Introduction*, volume 217 of *Graduate Texts in Mathematics*. Springer, 2002.
- [20] Mikyeong Moon and Keunhyuk Yeom and Heun Seok 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, July 2005.

- [21] A. Mili, J. Desharnais, and F. Mili. Relational heuristics for the design of deterministic programs. *Acta Inform.*, 24(3):239–276, 1987.
- [22] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE2(1):1–9, March 1976.
- [23] David Lorge Parnas. Precise description and specification of software. In V. Stavridou, editor, *Mathematics of Dependable Systems II*, pages 1–14. Clarendon Press, 1997.
- [24] Juha Savolainen, Ian Oliver, Mike Mannion, and Hailang Zuo. Transitioning from product line requirements to product line architecture. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, pages 186–195, Edinburgh, UK, 26–28 July 2005. IEEE Computer Society.
- [25] G. Schmidt and T. Ströhlein. *Relations and Graphs*. EATCS Monographs in Computer Science, Springer-Verlag, 1993.
- [26] Software Engineering Standards Committee of the IEEE Computer Society. IEEE Recommended Practice for Software Requirements Specifications, IEEE Std 830-1998 (revision of IEEE std 830-1993). <http://ieeexplore.ieee.org/iel4/5841/15571/00720574.pdf?arnumber=720574>. (accessed May 23, 2007), 25 June 1998.
- [27] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison Wesley Longman, Inc., 1999.

APPENDIX

A Characteristic Predicate of the Relation corresponding to the Scenario **Scenario_Basic_Personal_Tasks**

Scenario_Basic_Personal_Tasks=

$$\begin{aligned}
& \text{MenuOption} = \text{BasicPersonalTskUpdate} \\
& \wedge \left(i = \epsilon \wedge \text{Authentication} = \text{false} \wedge i' \in A^+ \right. \\
& \quad \vee i \in A^+ \wedge (i' = \epsilon \wedge \text{OnlyChge}(i) \vee \text{Authentication}' = \text{true} \wedge i = \epsilon \\
& \quad \quad \wedge \text{OnlyChge}(i, \text{Authentication})) \\
& \quad \vee \text{Authentication} = \text{true} \wedge i = \epsilon \wedge i' \in A^+ \\
& \quad \quad \wedge \text{OnlyChge}(i) \\
& \quad \vee \text{Authentication} = \text{true} \wedge i \in A^+ \wedge i \in \text{SetOfEmployees} \\
& \quad \quad \wedge i' = \epsilon \wedge \text{MenuOption}' = \text{UpdateData} \\
& \quad \quad \wedge \text{OnlyChge}(i, \text{MenuOption}) \\
& \quad \vee \text{Authentication} = \text{true} \wedge i \in A^+ \wedge i \notin \text{SetOfEmployees} \wedge i' = \epsilon \\
& \quad \quad \left. \wedge \text{OnlyChge}(i) \right) \\
& \vee \text{MenuOption} = \text{UpdateData} \wedge i \in A^+ \\
& \quad \wedge \left(\text{Address}' = i \wedge \text{EffectDateNeeded}' = \text{true} \wedge i' = \epsilon \right. \\
& \quad \quad \wedge \text{OnlyChge}(\text{Address}, i, \text{EffectDateNeeded}) \\
& \quad \quad \vee \text{HomeTel}' = i \wedge \text{EffectDateNeeded}' = \text{true} \wedge i' = \epsilon \\
& \quad \quad \quad \wedge \text{OnlyChge}(\text{HomeTel}, i, \text{EffectDateNeeded}) \\
& \quad \quad \vee \text{WorkTel}' = i \wedge \text{EffectDateNeeded}' = \text{true} \wedge i' = \epsilon \\
& \quad \quad \quad \left. \wedge \text{OnlyChge}(\text{WorkTel}, i, \text{EffectDateNeeded}) \right) \\
& \vee \text{MenuOption} = \text{UpdateData} \wedge i = \epsilon \wedge \text{EffectDateNeeded} = \text{true} \wedge i' \in A^+ \\
& \quad \wedge \text{OnlyChge}(i) \\
& \vee \text{MenuOption} = \text{UpdateData} \wedge \text{EffectDateNeeded} = \text{true} \wedge i \in A^+ \\
& \quad \wedge \text{EffectDate}' \geq \text{CurrentDate} \wedge i' = \epsilon \wedge \text{OnlyChge}(\text{EffectDate}, i) \\
& \vee \text{MenuOption} = \text{UpdateData} \wedge i \in A^+ \\
& \quad \wedge \left(\text{Name}' = i \wedge \text{EffectDateNeeded}' = \text{false} \wedge i' = \epsilon \right. \\
& \quad \quad \wedge \text{OnlyChge}(\text{Name}, i, \text{EffectDateNeeded}) \\
& \quad \quad \vee \text{EmrgceCntcts}' = i \wedge \text{EffectDateNeeded}' = \text{false} \wedge i' = \epsilon \\
& \quad \quad \quad \wedge \text{OnlyChge}(\text{EmrgceCntcts}, i, \text{EffectDateNeeded}) \\
& \quad \quad \vee \text{NextKin}' = i \wedge \text{EffectDateNeeded}' = \text{false} \wedge i' = \epsilon \\
& \quad \quad \quad \left. \wedge \text{OnlyChge}(\text{NextKin}, i, \text{EffectDateNeeded}) \right)
\end{aligned}$$

$$\begin{aligned}
& \vee \text{BirthAdoption}' = i \wedge \text{EffectDateNeeded}' = \text{false} \wedge i' = \epsilon \\
& \quad \wedge \text{OnlyChge}(\text{BirthAdoption}, i, \text{EffectDateNeeded}) \\
& \vee \text{MariageDtls}' = i \wedge \text{EffectDateNeeded}' = \text{false} \\
& \quad \wedge \text{MessageToDisplay}' = \text{ChangeMightAffectNextkin} \wedge i' = \epsilon \\
& \quad \wedge \text{OnlyChge}(\text{MariageDtls}, \text{MessageToDisplay}, i, \text{EffectDateNeeded})
\end{aligned}$$

B Example

```

-----
--ESS systems Example
personalInfo = bf "Scenario_Personal_Info"
personalInfoFlexibility = bf "Scenario_Personal_Info_Flexibility"
basicPersonalTasks = bf "Scenario_Basic_Personal_Tasks"
updatePersonalInfo = bf "Scenario_Update_Personal_Info"
employmentHistory = bf "Scenario_Employment_History"
hr_Policies = bf "Scenario_HR_Policies"
timeManagement = bf "Scenario_Time_Management"
absenceManagement = bf "Scenario_Absence_Management"
absenceInformation = bf "Scenario_Absence_Information"
absenceCalender = bf "Scenario_Absence_Calender"
holidayEntitlement = bf "Scenario_Holiday_Entitlement"
holidayAdministration = bf "Scenario_Holiday_Administration"
expensesTasks = bf "Scenario_Expenses_Tasks"
payrollAdministration = bf "Scenario_Payroll_Administration"
benefitDisplay = bf "Scenario_Benefit_Display"
multiBenefitPrograms = bf "Scenario_Multi_Benefit_Programs"

-- more features
internet_intranet_enabled_ess = personalInfo *. personalInfoFlexibility

ess_basic_basic_personal_tasks = basicPersonalTasks *. updatePersonalInfo
*. employmentHistory *. hr_Policies

ess_time_and_attendance_tasks = opt[timeManagement] *. opt[absenceManagement]
*. opt[absenceInformation *. opt[absenceCalender]]
*. opt[holidayEntitlement *. opt[holidayAdministration]]

ess_payroll_and_benefits_tasks = opt[payrollAdministration]
*. opt[benefitDisplay .+. multiBenefitPrograms]

--whole productline
employee_selft_service = internet_intranet_enabled_ess
*. ess_basic_basic_personal_tasks
*. ess_time_and_attendance_tasks

```

```
.*.  opt[expensesTasks]
.*.  ess_payroll_and_benefits_tasks
```

Using our prototype tool, we can get the list of features that are common to the members of the family. For that we execute the command *common (norm employee_selft_service)*.

```
*Main> common (norm employee_selft_service)
["Scenario_Basic_Personal_Tasks","Scenario_Employment_History","Scenario_HR_Policies"
"Scenario_Personal_Info_Flexibility","Scenario_Update_Personal_Info"]
*Main>
```

We can get the number of products in the the above family. For that we execute the command *length (norm employee_selft_service)*.

```
*Main>
*Main> length (norm employee_selft_service)
432
*Main>
*Main>
```

To list all the members, we execute the command *norm employee_selft_service*. It would take a while to list all the members with their features.

C Mathematical Details

Proposition C.1 ([3]). *Let $(\mathcal{B}_R, \sqsubseteq_r)$ be a complete join semilattice, with greatest element $\emptyset$. Letting $\sqcup$ denote join with respect to $\sqsubseteq_r$, we have*

$$\sqcup_{i \in X} R_i = (\cup_{i \in X} R_i) \cap (\cup_{i \in X} R_i ; \mathbb{L})$$

and

$$(\sqcup_{i \in X} R_i) ; \mathbb{L} = \cap_{i \in X} R_i ; \mathbb{L}$$

where X is a nonempty index set and $\forall (i \mid i \in X : R_i \in \mathcal{B}_R)$.

Proposition C.2 ([3]). *Let $(\mathcal{B}_R, \sqsubseteq_r)$ be a complete join semilattice, and X be an arbitrary index set. For each $i \in X$, let $R_i \in \mathcal{B}_R$. The greatest lower bound of the set of relations $\{R_i \mid i \in X\}$ exists if and only if*

$$\mathbb{L} \subseteq (\cap_{i \in X} R_i \cup \overline{R_i} ; \mathbb{L}).$$

When the above condition is satisfied, the greatest lower bound is

$$\sqcap_{i \in X} R_i = (\cap_{i \in X} R_i \cup \overline{R_i} ; \mathbb{L}) \cap (\cup_{i \in X} R_i ; \mathbb{L}),$$

where $\sqcap$ denotes meet with respect to $\sqsubseteq_r$; furthermore,

$$(\sqcap_{i \in X} R_i) ; \mathbb{L} = \cup_{i \in X} R_i ; \mathbb{L}.$$

Simple cases worth noticing include the following:

Proposition C.3. *When the demonic meet of relations Q are R is defined, we have*

1. $R \sqcap R = R$,
2. $R \sqcap \emptyset = \emptyset \sqcap R = R$,
3. Q and R partial functions $\implies Q \sqcap R = Q \cup R$,
4. $\text{dom}(Q) \cap \text{dom}(R) = \emptyset \implies Q \sqcap R = Q \cup R$,
5. $\text{dom}(Q) = \text{dom}(R) = \text{dom}(Q \cap R) \implies Q \sqcap R = Q \cap R$.