Integrating Feature Modeling with the RSEB

Proceedings of Fifth International Conference on Software Reuse, Victoria, B.C., 1998

Martin L. Griss
Laboratory Scientist
Hewlett-Packard Company,
Laboratories, Palo Alto, CA
griss@hpl.hp.com

John Favaro
Senior Software Engineering Consultant
Intecs Sistemi, Pisa - Italy
favaro@pisa.intecs.it

Massimo d'Alessandro
CASE Methodologist
Intecs Sistemi, Pisa - Italy
dalex@pisa.intecs.it

Abstract

We have integrated the feature modeling of Feature-Oriented Domain Analysis (FODA) into the processes and workproducts of the Reuse-Driven Software Engineering Business (RSEB). The RSEB is a use-case driven systematic reuse process: architecture and reusable subsystems are first described by use cases and then transformed into object models that are traceable to these use cases. Variability in the RSEB is captured by structuring use case and object models using explicit variation points and variants. Traditional domain engineering steps have been distributed into the steps of the architectural and component system development methods of the RSEB. But the RSEB prescribes no explicit models of the essential features that characterize the different versions. Building on our experience in applying FODA and RSEB to the telecom domain, we have added explicit domain engineering steps and an explicit feature model to the RSEB to support domain engineering and component reuse. These additions provide an effective reuse-oriented model as a 'catalog' capability to link use cases, variation points, reusable components and configured applications.

Keywords: *reuse, domain engineering, use cases, FODA, FODAcom, telecommunications, architecture, UML*

1. Introduction

It is well-known in the reuse community that in order to achieve high-levels of reuse, one must adopt a product-line perspective. That is, one must develop an architecture and a set of reusable assets well suited to express and efficiently implement the different members of a family of related applications. To do this, we perform a reuse-oriented analysis of the "domain" in which these applications fall, and then carefully design and construct the domain architecture and reusable assets. This latter process is called domain engineering. Since 1988, working individually and together at Hewlett-Packard and at Intecs Sistemi, we have been developing a systematic approach to domain-specific, object-oriented software reuse. At Hewlett-Packard, we have applied domain engineering techniques, OO methods, and organizational design to our corporate reuse programs and to customer reuse efforts. In 1997 we published a book on the "Reuse-Driven Software Engineering Business (RSEB) with partners from Rational Software Corporation, describing our systematic approach [1]. At Intecs Sistemi we have been involved for many years in the development of OO and reuse design tools [2] for industrial and aerospace applications [3]. Over the past two years we have been involved in a customization of the Feature-Oriented Domain Analysis (FODA)[4] method for applications in the Italian telecom arena. Most recently, we have combined our joint experience, integrating FODA-derived ideas into the RSEB.

In this paper we will first briefly review the concepts of Domain Engineering, the RSEB, and FODA, and explain how FODA and parts of the RSEB were applied to the telecom sector in the FODAcom project. We then describe our latest contribution, the adaptation of the feature model from FODA to provide a another model for the RSEB. This model provides a high-level view of the domain architecture and reusable assets to provide the reusers and the domain engineer a more concise and expressive picture of the way to build systems in the application family.

2. Background

2.1. Domain Engineering

Domain engineering (DE) is a key process needed for the systematic design of an architecture and set of reusable assets (components and other workproducts) that can be used to construct a family of related applications or subsystems. It is a systematic process that incorporates business criteria and produces some supporting rationale, models and architectures that enable better decisions to be made, recorded and revisited for further revision, and for process improvement based on learning. Domain engineering is a key part of preparing a reliable architecture and components for reuse[5, 6].

Domain engineering starts with *Domain Analysis* (DA), primarily a systematic analysis of *commonality* and *variability* in a family of systems across a domain: example systems, user needs, domain expertise and technology trends are analyzed to identify and characterize elements that are common to all family members, and also to deal with elements that vary between family members. The subsequent *application engineering* process is driven by both the resulting components and the models and documentation developed during DA. Most domain analysis methods use a specific model to play a unifying role for asset reuse across a family of systems. In contrast, most ordinary OO methods do not explicitly support this family-oriented representation of commonality and variability in either notation or method.

A domain model is a high-level description of the application family; it provides a framework for describing the essential characteristics, and deciding which application features are appropriate for certain user or technical requirements. A domain architecture is much more precise in showing the structure of a typical (or all) applications in the domain; it defines subsystems, and connections between them, identifies key mechanisms and services.

Domain engineering involves specific requirements analysis, architecture, component development and packaging steps, summarized in Table 1. FODA and RSEB will be described in the following sections. Steps 1-3 are referred to as *domain analysis*, and steps 4-7 as *asset* or *component engineering*. Several different domain engineering methods have been developed [5]. They vary in how they identify the domain effectively, and make maximum use of available domain, architecture and systems expertise. Some methods focus on how to select existing exemplars for detailed analysis, while others focus on how to collect, represent and cluster sets of so-called *features*. Different applications in the same application family or problem domain, are often compared or distinguished based on their “features.” When developing an architecture and component systems for reuse, it is important to understand which features are provided by which parts of the set of reusable assets, how features relate, and which features are mandatory or optional for different applications.

Yet there has always been a certain degree of disagreement among the various methods regarding the exact nature of “features” and their role in domain engineering [5]. Our work has made a contribution to a more precise understanding of feature modeling and its relationship to recent requirements capture methods, particularly use case modeling.

2.2. FODA

Feature Oriented Domain Analysis (FODA) was developed at the Software Engineering Institute [4]. Its thorough description of the Domain Analysis process made it become one of the most popular methods in the 1990s. In particular, FODA made a significant contribution to the current popularity of *model-driven* analysis: the use of several complementary views of a domain to convey complete information about that domain.

TABLE I. FODA and RSEB Domain engineering steps

1. Domain identification and scoping - most applications consist of several recognizable or distinct subsystems or sub-problems, only some of which can be economically reusable. Thus it is important to decide which parts are worth further treatment. In FODA, a *context model* is constructed. In the RSEB, scoping closely related to decomposition into component systems, and the high-level use case model (use cases and actors) that describe the system architecture and context.

2. Selection and analysis of examples, needs and trends - there is a delicate balance between reactive and proactive reuse. A set of reusable components must anticipate future needs. Since this is difficult to do reliably, and it is more expensive to build reusable software than regular software, the reuse process must find the essential commonality and variability, and prioritize which parts should be processed for reuse. Most approaches selecting key examples, and extract their essential feature sets. The examples chosen relate to domain scope and assessment of user needs, market, technology and business trends.

3. Identification, factoring and clustering of feature sets - using analysis models, tables and/or graphical means, features that appear together (AND) or are variants to be selected from (OR, XOR) are structured into a decision framework. Domain terminology is accumulated. In FODA, an *information model* describes the domain entities and the structural relationships among them, whereas in the RSEB a use case and analysis object model is built. In FODA, a *behavioral model* describes the dynamic relationships among the entities of the domain, corresponding roughly to RSEB sequence and interaction diagrams. The FODA *functional model*—related to the RSEB object models—describes the data flows among domain entities. The FODA *feature model*, ties all of these models together by structuring and relating feature sets.

4. Development of ‘domain’ or ‘generic’ model and architecture - from these feature sets, a domain model summarizing the essential characteristics of all/any application in the domain. Also developed is a robust domain architecture **relating** core mechanisms, features, subsystems and variants, using some architecture description. Architecture shapes the resulting applications or subsystems, defines core services, specifies interfaces precisely, and serves as a reference model and functional blueprint. FODA domain architecture is closely related to the overall RSEB layered architecture.

5. Representation of usable commonality and variability - generalized subsystems, modules and functions are identified, and related to each other as generalizations, specializations or alternatives.

6. Exploitation of commonality and variability - notations and mechanisms are used to specify several (classes of) generic or parameterized workproducts, some of which will become reusable.

7. Implementation, certification, and packaging of reusable components - the ‘most important’ subset of the candidate reusable assets and workproducts are implemented and released as certified, reusable components, under configuration management. Assets use a variety of technologies, ranging from components to generators, languages and kits. Other reusable assets will be implemented as needed.

Explicit feature modeling is key in FODA, and is the basis for much subsequent work.. FODA work at the SEI has expanded into Model-Based Software Engineering, positioning FODA as an integral part of domain-engineering leading to application engineering [7]. The kinds of features that can be considered has been enriched, explicitly distinguishing those related to user visible functionality, from those related to implementation issues.

2.3. RSEB

The Reuse-Driven Software Engineering Business (RSEB)[1] is a systematic, model-driven approach to large-scale software reuse. RSEB is based on Jacobson's OO

Software Engineering[8] and OO Business Engineering[9], applied to an organization engaged in building sets of related applications from sets of reusable components. Explicit use case models are central to all steps that define architecture, subsystems and reusable objects.

We use the Unified Modeling Language (UML)[10], with extensions to model and specify application systems, reusable component systems and layered architectures, and to express system variability in terms of variation points and attached variants. RSEB defines several model-driven software development processes: Architecture Family Engineering (develops a layered architecture), Component System Engineering (develops systems of reusable components) and Application System Engineering (develops selected applications). These processes optimize for robustness and reuse. Engineers and managers play specific roles, such as Architect, Use-case Engineer, Component System Engineer, Configuration Manager, etc.

Both RSEB and Hewlett-Packard's domain analysis method [11] derive considerable material from FODA, and from Organizational Domain Modeling(ODM) [12]. ODM highlights how and why "exemplars" (example systems that characterize important aspects of the domain) are chosen and analyzed to meet technical and stakeholder needs. It also includes a significant emphasis on feature analysis.

2.4. FODAcom

Telecom Italia launched the FODAcom project in the mid-1990s under the leadership of Telesoft S.p.A. to create a customized version of FODA for applications in the telecom domain. The choice of FODA as a point of departure comes as no surprise. Telecom organizations, with their heavily "feature-oriented" services, have always had a natural affinity for FODA. In fact, much of the work on FODA since its definition has been carried on by telecom organizations, for instance, by Kang with the Korean telecom authority, and by Bell Northern Research in Canada [13] and NORTEL in the United States [14, 15].

Intecs was tasked with the methodological lead in the FODAcom project. Contacts with Hewlett-Packard led to early introduction of several key concepts from the RSEB into FODAcom, which soon focused on using primarily the feature model as a concise synthesis of the variability and commonality represented in the other RSEB models, especially the use case model. This early experience led to a synthesis of ideas, expressed in the remainder of this paper.

3. Featuring RSEB

FODA is quite compatible with the RSEB: it is a model-driven approach, with domain information captured in several different models reflecting different points of view of the domain. Hence the combination of aspects of FODA and RSEB are generally quite complementary.

Experience with FODAcom exposed the incompleteness of the RSEB with respect to domain analysis. In the RSEB, we distributed DA-*like* activities across the various processes that develop the system architecture and create component systems. The RSEB method uses features informally, suggesting that different features characterize the different systems that can be constructed from the reusable component systems. Features are related to use cases or parts of use cases, and lightly related to implementation details or operating constraints. Unlike domain engineering methods such as FODA, RSEB provides no explicit feature models or steps that construct and transform such feature models.

In the following sections, we describe in more detail the primary contribution of this paper: integrating the FODA-like feature model with the use-case driven RSEB to produce ***FeatuRSEB*** – the "featured RSEB".

3.1. Features vs. Use Cases

The original FODA purpose for feature analysis was: "… to capture in a model the end-user's (and customer's) understanding of the general capabilities of applications in a domain." This is sounds like *use case modeling.* When FODA first appeared in 1990, use cases were not yet as widely accepted for OO requirements analysis or as central to OO software engineering as they are today. It is now clear that feature analysis anticipated much of modern thinking on requirements capture techniques. But, this also leads naturally to the question: What is the relationship between feature modeling and use case modeling?

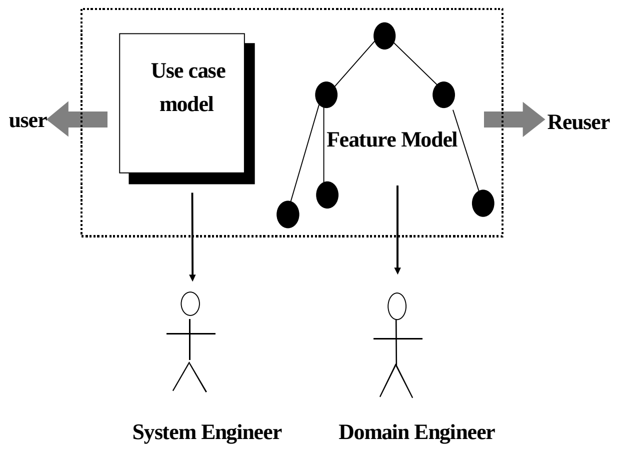

Figure 1 Different unifying models for different audiences.

As shown in Figure 1, the use case model and the feature model serve different purposes. The use case model is *user* oriented; the feature model is *reuser* oriented. The use case model provides the "what" of the domain: a complete description of what systems in the domain do. The feature model provides the "which" of the domain: which functionality can be selected when engineering new systems in the domain.

Use case modeling relieves feature modeling of the "double duty" it was performing in the original FODA definition. Use cases now gather and describe user requirements on system functionality—under the control of the system engineer—and they do a better job of it. The feature model can focus on what it does best: *organize the results of commonality and variability analysis in preparation for domain engineering* and *reuse access.* This is the province of the domain engineer.

In the telecom sector, lack of explicit feature representation can be especially problematic, even in the presence of use case modeling. First, use case models do not explicitly reveal many of the implementation or technical features prevalent in telecom systems, such as" switch types." Second, telecom services can require very large numbers of use cases for their descriptions; and when the use cases are parameterized with RSEB variation points describing many extensions, alternatives, and options, the domain engineer can easily lose his way when architecting new systems. Reusers can easily get confused about which features and use cases to use for which application systems. Here the feature model can play a central organizing role, as discussed in the following section.

3.2. The "+1" Model of Domain Engineering

In the RSEB, we subscribe to the "4+1 View" approach to architecture popularized by Rational methodologist Philippe Kruchten [16]: develop several models (analysis, design, etc.) to represent important architectural views, plus one model that ties them all together. In the RSEB this unifying role is played by the use case model. RSEB doesn't consider features to be a central concern, nor does it contain an explicit DA process. Thus there is no single model in the RSEB that becomes the reference (that is, the "+1" model) for directly recording information about features and their commonality and variability in a domain.

FODAcom partially updated FODA by providing side-by-side use case models and explicit feature models. In FeatuRSEB we go further by integrating explicit domain analysis and RSEB elements to work together effectively. The feature model is no longer "side-by-side" with only the use case model, but with *all* of the other models. The feature model takes "center stage" and assumes the role as the "+1" model for domain engineering. The feature model provides an abstract and concise syntax for expressing commonality and variability in the domain. In the following subsections we explore the implications of this central, unifying role of the feature model in domain engineering.

3.2.1. *Less is More: the added value of Domain Analysis*

Simos [12] and Arango [5] especially have pointed out that domain analysis and domain engineering involve a combination of *both* analysis and synthesis. Even though the domain analysis process starts with the analysis of existing systems and other available information, its purpose is not just to merge and re-organize everything into a unified model from which the different application systems can be instantiated. In fact, the domain model and the domain architecture serve two purposes. First, they describe in a systematic way how existing systems in the domain are actually built, by introducing terminology and structure that summarize the common and different characteristics of existing systems. Second, and more important, they determine how "good" systems should be built in the future. This represents their true added value.

In several cases, the "domain" as such does not fully exist as such before the analysis is performed. Thus when a exemplar system considered in the analysis was built, some domain characteristics may have been ignored, leading to solutions that could be recognized as obsolete or replaceable given the broader view of the problem delivered by the domain analysis. This means that at the end of the analysis, when the domain is clearly defined, an exemplar system initially taken as a reference for the domain definition, may itself not be fully compliant with the domain definition.

During analysis of several exemplars, we might discover that some systems provide certain features and others do not. For example, some telecom switches might offer the call waiting service, and others not. But on closer examination and reflection, we realize that all (or a large subset of) future systems should include some functionality or feature. For example, we might move an optional feature, or little used feature, to become a mandatory feature. Likewise, the analysis of existing systems might initially suggest that two features are mutually exclusive, but on reflection, it makes sense to design systems in which they could both be selected. For example, some phone systems provide both pulse or tone dialing at the flick of a switch. This synthesis involves innovation, and may require a (significant) change in architecture, mechanisms, or corresponding implementation.

Part of the reason for having an explicit feature model in addition to a use case model is to make such choices and opportunities much clearer, and easier to analyze. The feature model in FeatuRSEB provides the central repository for organizing the results of this creative process. There are several kinds of features (mandatory, optional, variant) that can be used to describe the essence of the domain, as well as some of its major variations:

- *Mandatory* features correspond to a core capabilities embodying the main domain characteristics at the problem level, and constitute the "infrastructure" of the domain—the first step toward a domain architecture.
- *Optional* features correspond to the identification of some capabilities or characteristics which may be unnecessary in some systems of the domain. A feature which does not appear in all exemplar systems considered in the domain analysis does not necessarily become optional. Optional features indicate secondary properties of the domain, in contrast with the primary properties associated with mandatory features.
- *Variant* features correspond to alternative ways to configure a mandatory or an optional feature. These are not simply a list of all alternative ways to do something as found in the systems under examination, but rather a selection of them, or better, a *revision* of them.

Thus the domain analysis process should carefully "critique" the exemplar systems (and other assets), since a broader point of view usually leads to the identification of inadequate or incomplete solutions. Special attention should be paid to discriminate between "optional" and "missing," and between "alternative" and "obsolete/inadequate" solutions. This admonition is certainly true at requirement and specification level but is even more effective at the architectural level. In fact is quite possible that the domain architecture which results from the domain engineering phase will not be simply a template from which all the other architectures and details of the individual pre-existing systems (the exemplars) can be formally derived. Due to the creative process of domain analysis, some particular solutions are made obsolete and a more advanced, integrated and flexible architecture should result, allowing reuse of domain specific component systems. All of this is recorded in the feature model.

3.2.2. A Catalog and Roadmap

The feature model is used as a *catalog* of feature commonality and variability. It needs to be as concise as possible. The feature model contains a well-organized collection of items that have been *selected* according to a process that includes deciding what should or should not become a visible feature of systems in the domain. This is similar to the more familiar systems engineering *data dictionary*. Distilled from many sources, it is an organized, selective, and normalized catalog of terms that becomes the reference for terminology used by project developers. In a domain engineering context, the data dictionary corresponds closely to the domain terminology dictionary. Whereas the use case model has to cover all of the requirements of individual systems in the domain, the feature model need not (and *should not*) include all possible features, but only those which the domain analyst deems important. Thus, paradoxically, the added value of a feature identification process isn't its completeness with respect to the space of all possibilities, but rather its "incompleteness"—it synthesizes only the essential items relative to the domain objectives.Now consider how the reuser approaches this catalog of features when building a new system in the domain. He has a complete specification of the domain given by the RSEB use case model of the domain (using *uses* and *extends* relationships and variation points). The feature model provides a "configuration roadmap" through the use case model, guiding through an understanding of what can be combined, selected, and customized in his system. This schematic roadmap also expresses semantic constraints that can't be found in any of the other domain models—for example, "this operating system can't be selected together with that hardware." And the feature model references characteristics that are hard to express directly as simple use cases, such as performance constraints, etc.

3.3 Notation for Features

In this section we consider notational aspects while developing a more concrete example. Not all features are automatically related to complete (or even partial) use cases, and therefore traditional use case notation is not immediately suitable as a feature notation. A much better starting point is the notation of *variation points* that was introduced in RSEB. Variation points reflect parameterization in the domain in an abstract manner, and appear in—and are traceable across—several RSEB models, not just the use case model. The following preliminary feature model for telephone service provision is inspired by work at MCI[17].

An easy-to-use feature model is particularly effective for "rapid creation of new services," leading to decreased cycle time in creating and deploying such systems. There is a group of basic features that the user sees as capabilities, but not all of these features would appear in use cases; some come from detailed implementation and configuration choices in deployable telecom systems; others from domain experts and others from analysis of typical switch and network configurations. While some features can be associated with use cases and variations, others must be associated with types of actors, or details such as included or omitted subsystems, parameters in objects, choice of target platform, etc. These only appear in later design or implementation models.

In the telecom domain, quite a few of the characterizing features are already known by domain experts without detailed analysis of use cases. These can be used to build an initial feature model. Or, some of them might be known to exist *a priori*, such as knowing that a switch type is a PBX switch. Likewise, some common and variant services described by use cases are also known early. Thus a rudimentary requirements model or initial use case model might exist even before serious domain engineering has begun. Both of the models provide concepts and terminology for describing, analyzing and structuring the other sources of information as domain engineering proceeds.

The feature model is represented in UML as a linked set of feature elements containing data describing attributes of the features, such as name, kind, etc. These feature elements are linked together by a set of relationships (typically UML *dependencies* or *refinements*), used to build up trees or networks of features. Some of the feature elements may also have relationships (typically a *trace*) to elements in other models, such as use cases, variation points or objects. We find it useful to display the feature model at several levels of detail (or views): one view is the simple feature tree or graph, showing feature names, major relationships, and a few attributes, as illustrated in Figure 2. This view may itself be filtered to show more or less detail. Thus, we have embraced the multiple-view approach for working with the feature model itself.

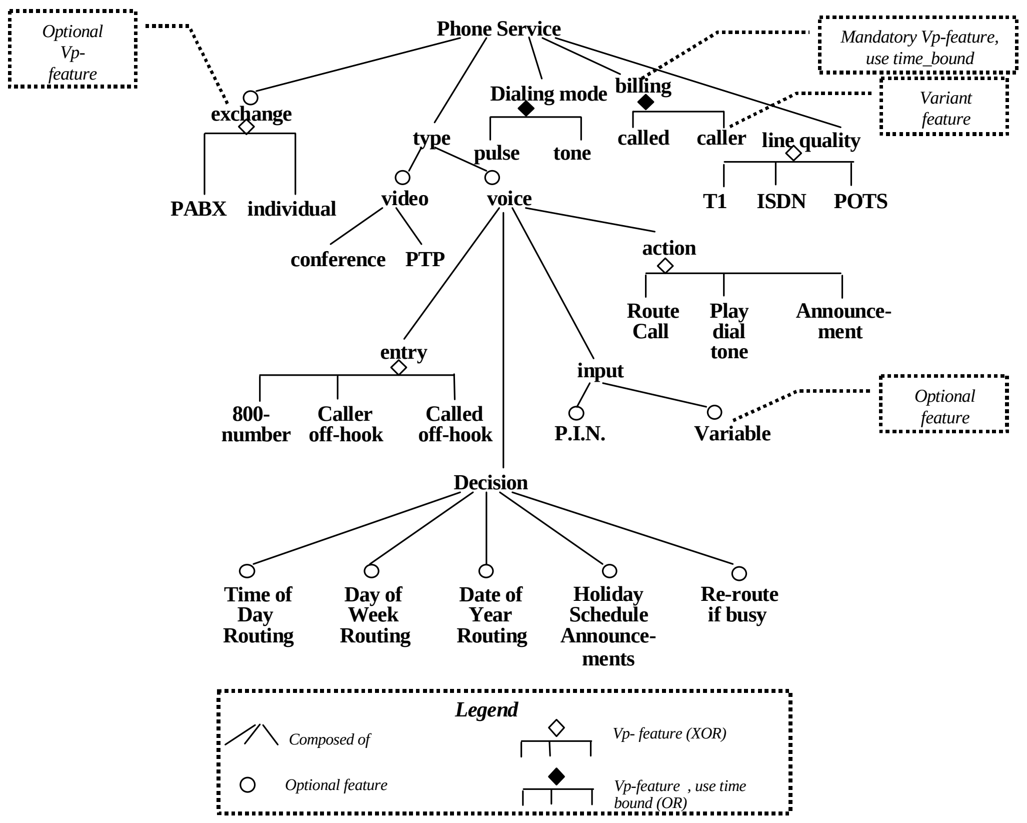

Figure 2 A high-level view of the feature model for Rapid Telephone Service Creation.

The set of features shown in Figure 2 can be specified and structured using the notation summarized in the legend:

- the *composed_of* relationship. A feature can be modeled as composed of several sub-features, following a decomposition/ aggregation abstraction mechanism. In the example, the feature "Phone Service" is composed of "exchange", "type", "billing", "line quality" and "dialing mode." The relationship is represented by a line from the super-features to each of its components.
- the *existence attribute*. A feature can be mandatory or optional. A mandatory feature must be selected in all the configurations of the feature model, while an optional feature may be disregarded in some configurations. An optional feature is represented with a circle above the feature name. A mandatory feature composed exclusively of optional features requires at least one of them to be selected in all the configurations.
- the *alternative* relationship: *variation* and *variant features*. A feature can act as a variation point (called *variation point feature or vp-feature*) in the model, while other features play the role of its possible variants (called *variant features*). In the example, the feature "exchange" is a vp-feature with "PBX" and "individual" as variants. From an inheritance point of view "exchange" is a more abstract feature with "PBX" and "individual" as two possible refinements. A feature which defines a variation point is represented with a diamond under its name. A line is drawn to each available variant from the diamond. In the textual notation a particular variant ***v*** of a variation point ***vp*** is ***vp.v*** (e.g. "*exchange.PBX*" or "*exchange.individual*").
- the *binding time* attribute of *vp-feature*s. Vp-features can be bound at *reuse time*, i.e. when the reuser accessess the domain infrastructure to configure reusable

assets for his development. In such a case, from the reuser point of view, vp-features are an XORed disjunction of their variants, since only one of them can be selected. Instead, these features may be bound at *use time*, i.e. included together in the system even though it allows a run time or load time binding [4]. In this case, from a reuser point of view, vp-features act as an ORed disjunction of their variants, allowing the reuser to select one or more. The binding time attribute is shown by the color of the diamond of the feature: white means reuse time, black means use time. In the example the features dialing mode and billing are use time bound vp-features.

- the *requires* and *mutual_exclusion* constraints. These rules define the semantic constraints on optional and variant features, to give the model consistency. It is possible to specify which features must be selected together with a designated one. In the example the feature "P.I.N" requires that the feature "dialing_mode.tone" must be selected as well. It is also possible to specify the incompatibility in the choice of two features. In the previous example the selection of the feature video implies that the "line_quality.POTS" cannot be selected. These constraints can be expressed as separate rules with respect to the diagram, directly in the feature element using the *requiresFeatures* or *excludesFeatures* attributes, or as UML constraints attached the feature elements or relationships.

Each feature node in Figure 2 is in fact an iconic view of a more complete feature element, perhaps implemented in UML as a stereotype, *«feature»*, of Class[10], just as we did

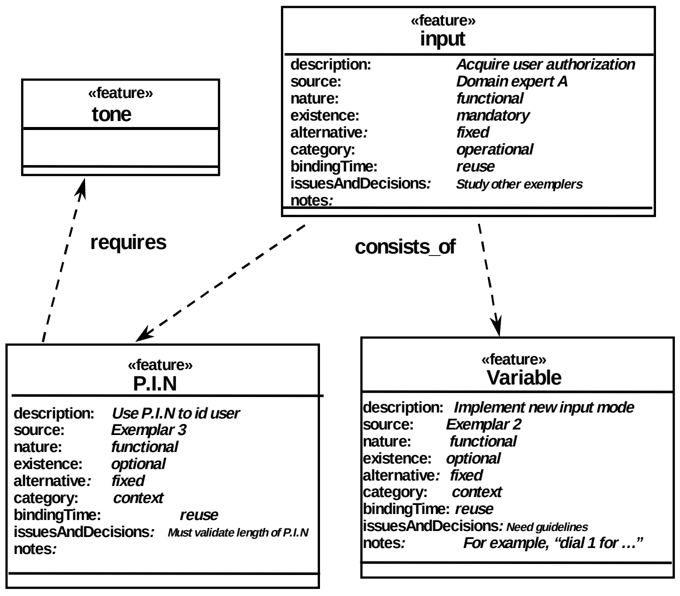

Figure 3 Expanded view of a section of Figure 2, showing more detail.

for the RSEB extensions[1,6]. The collapsed or iconic notation for a feature will be shown just as the name, or name decorated with small circle or diamond as in Figure 2, while the expanded or full view might show as in the section of Figure 2 reproduced in Figure 3. The various attributes record the kind of feature, rationale, and notes, while the relationships show special connections to other features, constraints, and a *«trace»* to other model elements.

Note that the existence attribute and the other relations are orthogonal. This means that you can have a feature composed of mandatory and optional features where an optional (or mandatory) feature can in turn be a vp-feature.

3.3. Process for feature model construction

Model construction in our "featured RSEB" is a concurrent process, just as the RSEB layered architecture, application

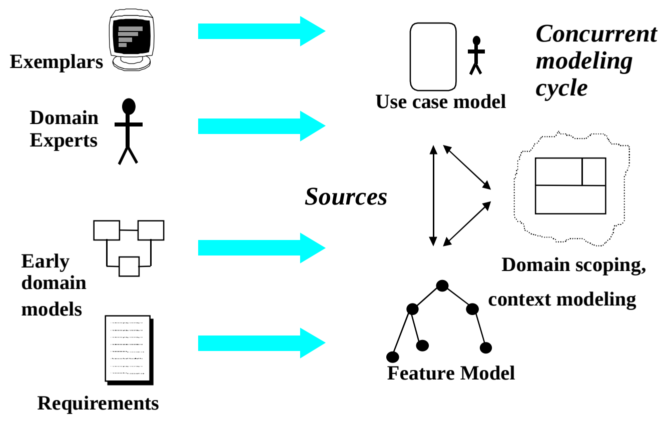

Figure 4 The concurrent model building process.

system and component system engineering processes are concurrent. Figure 4 shows several sources of information, including exemplars, requirements, domain experts, and early domain models, which are fed into a concurrent and continuous cycle of context modeling and scoping; feature modeling; and use case modeling. (The other models are not shown here for simplicity.) Each of the models is gradually built up concurrently with others.

However, in our experience, some software engineers (and managers) are quite uncomfortable with this concurrent aspect. They would much prefer to either think of completely developing the feature model before developing the use case model, or (more effectively) think of a series of releases: first develop increment 1 of the feature model, then increment 1 of the use case model. Then iterate to obtain increment 2, and so on. However, as we illustrate, there is significant cross feeding between feature model analysis and synthesis, and use case development with variation point structuring.

Let us consider now in more detail the process for constructing the feature model in FeatuRSEB, beginning with an analogy to the process for object model construction in OOSE (Object-Oriented Software Engineering) or RSEB. In both OOSE and the RSEB, the essential process is as follows:

1. *Identify Objects*. Given a use case model, apply the *Boundary-Entity-Control* pattern [1] to map each use case onto a collaboration of (analysis) objects. Heuristics aid in choosing names related to actorss and system data.
2. *Homogenize the set of identified objects*. Decide whether similarly named objects are in fact the same object, different instances of the same object, or distinct. Rename or merge as appropriate.
3. *Perform robustness analysis*. Group objects into stable subsystems, possibly re-grouping or merging objects. Apply principles of cohesion and coupling to increase robustness of object clusters.
4. *Complete the set of operations*. Examine the use cases and assign responsibilities and operations to corresponding objects. At the end of the process the use case model can be rewritten in terms of the objects to obtain the analysis use case model.

The next step then maps analysis objects into design objects, adding or modifying object to account for implementation constraints. UML *«trace»* connect the use cases to corresponding analysis objects, and analysis objects to corresponding design objects. Other patterns can be applied also. In the RSEB all the models use various mechanisms to express variability in terms of variation points and attached variants.-In the RSEB process, the starting use case model is a family use case model, representing a set of related use case models by using variation points. This family variability model will be defined during Application Family Engineering when the family is architected. Some variability could also be derived by merging several individual use case models.

Now consider the process in FeatuRSEB. The first extension to the RSEB process is to make explicit the *set* of use case models used to construct family use case model, each one associated with an exemplar system selected for domain analysis. For our discussion, we refer to the family use case model as the domain use case model, and so on. Then we construct a feature model concurrently with the use case model. Analogous to the process described above for the RSEB, in FeatuRSEB we then perform commonality/variability and added-value analysis/synthesis, first on the use case models and then directly on the feature model in a phase of robustness analysis. The feature model construction process can thus be outlined as follows:

1. Merge individual exemplar use case models into a domain use case model (known as the Family Use case Model in the RSEB), using variation points capture and express the differences. Keep track of the orginating exemplars using *«trace»*
2. Create an initial feature model with *functional features* derived from the domain use case model, (typically using use case names as a starting point for feature names).
3. Create the RSEB analysis object model, augmenting the feature model with *architectural features*. These features relate to system structure and configuration rather than to specific function.
4. Create the RSEB design model, augmenting the feature model with *implementation features*.

Since the second and third steps give the feature model its baseline structure, we describe them now in more detail. These steps are largely based on the abstract concept of RSEB *variation points*. As we will see, it is important to determine what constitutes a good feature. As noted earlier, not everything that *could* be a feature *should* be a feature. Feature descriptions need to be robust and expressive. Features are used primarily to discriminate between *choices*, not to describe functionality in great detail; such detail is left to the use case or object models.

3.3.1. Domain use case model construction

Construct domain actors model. Classify the set of actors over all individual use case models on the basis of their *affinity* (as perceived by domain experts, existing domain descriptions, domain dictionary, etc.). Identify synonyms and homonyms—actors with different names but the same role (e.g. *client* and *customer* become *user*). Factor common responsibilities between similar actors with abstract actors (e.g. *configuration manager* and *webmaster* become an abstract *administrator* actor). Trace each actor back to its originating exemplar system, whereby abstract actors are traced to all originating exemplar systems.

Construct domain use case model. Merge all exemplar use case models, replacing original actors with those from the domain actors model. For abstract actors, extract abstract use cases (with commonalities) from the concrete use cases of their concrete ancestors, using variation points. For all actors, factor out similarities in the use cases in which they participate, through the variation point mechanism. Trace concrete and abstract use cases back to originating use cases in the exemplar models.

Perform robustness analysis on domain use case model. Consider the domain use case model as a specification of a single system and perform a critical review (consistency, redundancy, ambiguities) of the user requirements implied by it, leading to further modifications.

3.3.2. Extracting functional features from the domain use case model

This step assumes that a feature is only traced to one use case (whereas one use case may correspond to many features). This can be seen as a *feedback* constraint on the use case model.

Identify mandatory and optional features. Create a list of top level use cases, ordered according to their frequency of occurrence in exemplar systems. Higher-frequency use cases are more likely to generate *mandatory* features, whereas lower-frequency use cases will generate more *optional* features. Not all use cases will or should generate features—the experience and judgment of the domain analyst is important here. Heuristics assisting feature analysis can be found in the literature. Examples are the *Context-Operational-Representation* classification in FODA [4] and the *concept starter set* of ODM [12].

Decompose features into sub-features. For each feature identified, decompose as appropriate according to the structure (e.g. use and extends relationships) of the related use case in the domain use case model.

Identify *vp-features* and their variants. If the source use case of a feature contains variation points, consider decomposing the feature into as many sub-features as variation points, possibly as *vp-features*. Trace each *vp-feature* back to its source variation point in the case model.

Perform robustness analysis on the feature model. Analyze and restructure the feature model as a whole for consistency and integrity, introducing semantic constraints such as *mutual_exclusion* and *requires* as appropriate. An important restructuring activity is the identification of new feature categories which could cause homogeneous features to be grouped in separate hierarchies [12].

As noted earlier, during the construction of the other models the same scenario may occur. For example during architectural model construction some analysis may identify variations and commonalities, leading to the definition of variation points. Once again, this *must* result in an extension of the feature model (or the realization that the variation point can reference an existing feature). This will create features that relate to system structure and configuration rather than to specific function.

In summary, although several models are constructed during featured RSEB domain analysis, the process is primarily driven by the construction of the feature model *as a result of* the “other” analyses, placing it side-by-side with all the other models. It is in this way that the feature model functions as a “roadmap” to the other models: the reuser first selects features from the catalog, from which he is led to the associated variation points in the detailed models. This roadmap capability allows us to identify further guidelines related to feature definition: optional and vp-features not related to any variation point in any model are suspected of not being good features. Conversely, variation points not related to any feature cannot be exploited effectively by reusers.

3.4. Tool support for FeatuRSEB

In this section we consider the kinds of tools we are considering for the concepts we have presented in this paper.

Tool Support for Feature Analysis. Basic support for the construction and manipulation of the feature model has already been developed for original FODA [18]. The RSEB book describes some tools to support Component System Engineering and process-driven development, and these are also useful in this context. For the featured RSEB we would add multiple view capabilities (e.g. “outline mode”) as well as selective viewing, zooming, suppression of optional nodes, etc. Some of this capability might be obtained directly or with some customization from an existing OOA tool, such as Rational Rose, Platinum Paradigm Plus or the UML-NICE toolset of Intecs, using the UML extension mechanisms.

Repository-based tool support. Tools focused *specifically* on domain analysis and feature modeling (such as the abstract feature tree viewer) might be best developed separately, with access to a common model repository. For example, in our research at HP Laboratories we will explore the UML-based Microsoft Repository. This repository presents itself as an OLE server (ActiveX object), and so can be easily managed from Visual basic, or interfaced to many existing UML tools. The repository meta-model is expressed in UML, and can be extended to include other kinds of elements, such as features. Another approach to tool architecture we might explore would be to use CORBA to integrate tool objects, and take advantage of the IDL interfaces defined for the UML 1.1 tool exchange facility.

Support for Traceability. All the models in the featured RSEB process are traceable to each other. A tool which allows definition, modification and navigation of the traceability relationships (as hypertext) would provide valuable assistance to the domain engineer. It could show the feature tree represented as a graphical map from which to access in a direct way (e.g. by mouse click) relevant places in the other models. Also, an automatic warning may appear whenever a new variation point is created in a model, reminding and signaling its traceability with a feature. This traceability between features and variation points—a special “feature” of FeatuRSEB—would particularly benefit from tool support.

Strong integration with configuration management. The close relationship between reuse and Configuration Management (CM) was already appreciated by Intecs in the development of the Intecs' ReuseNICE toolset [3]. CM can be used in FeatuRSEB to manage the combination of the features and of the respective variation points in the models.

Strong integration with CM facilities would provide strong support for the “catalog” and “roadmap” characteristics of FeatuRSEB discussed in this paper.

4. Conclusion

By starting from the use case based RSEB models which already incorporate variability and some aspects of domain engineering, we have been able to integrate the RSEB and standard FODA processes and models. UML and RSEB provide a natural base, requiring only a few extensions. UML stereotyping provides an explicit representation for the feature model. The resulting FeatuRSEB feature models are simpler than those in original FODA, focusing on the feature-oriented parts for domain-engineering purposes. These feature models act as a convenient catalogue or index into the commonality and variability present in use case and object models, simplifying the task for the reuser. We have outlined a FeatuRSEB domain analysis process as an extension to the RSEB process, and hinted at tools to support the process of constructing and revising the models.

Not only does this seem like a natural and robust extension to the simplified treatment of features in RSEB, but more importantly, this FeatuRSEB combination has been applied to the telecom business in the earlier form of FODAcom. It provides a robust and easy to use way of characterizing the many hundreds of reuse-oriented features and use case variants.

Next steps involve the exploration of appropriate tool support for our feature-extended RSEB models. One route would be to start from a standard UML tool (e.g. Rational ROSE) and add RSEB and FODA extensions using the UML extension mechanisms. Another route would be to create or modify existing reuse tool-sets (such as Intecs' ReuseNICE or UML-NICE) to add more direct support for UML, FODA, and RSEB.

5. Acknowledgments

We thank Sholom Cohen, Patricia Cornwell, Steven Fraser and Mark Simos for several useful comments on a previous version of this paper. Their suggestions greatly improved the flow and focus of this expanded treatment.

6. References

[1] I Jacobson, M Griss, P Jonsson, Software Reuse: Architecture, Process and Organization for Business Success, Addison-Wesley-Longman, May 1997.

[2] M D’Alessandro, PL Iachini, A Martelli, The Generic Reusable Component: An Approach to Reuse Hierarchical OO Designs, Proc. Second International Workshop on Software Reusability, IEEE Press, 1993, pp. 39-46.

[3] M. D’Alessandro, Reuse and Configuration Management in HOOD: the SCALE experience, DASIA 96—Data Systems in Aerospace, ESA Publications, 20-23 May 96, Rome.

[4] K Kang et al, Feature-Oriented Domain Analysis Feasibility Study, SEI Technical Report CMU/SEI-90-TR-21, November 1990.

[5] G Arango, Domain Analysis Methods, in W. Schãfer et al., Software Reusability, Ellis Horwood, Hemel Hempstead, UK, 1994.

[6] ML Griss, Domain Engineering And Variability In The Reuse-Driven Software Engineering Business, Object Magazine, Dec 1996.

[7] S Cohen, R Krut, S Peterson and J Withey, Models for Domains and Architectures: A Prescription for Systematic Software Reuse, Proceedings of AIAA Computing in Aerospace 10, San Antonio, TX, March, 1995. Pages-125.

[8] I Jacobson et al, Object-oriented software engineering: A use case driven approach, Addison-Wesley, Reading, MASS. 1992.

[9] I Jacobson et al, The Object Advantage - Business Process Reengineering with Object Technology, Addison-Wesley, Menlo Park, CA, 1994.

[10] G Booch, I Jacobson and J Rumbaugh, The Unified Modeling Language, Version 1.1, OMG Submission, Sept 1, 1997.

[11] P Collins-Cornwell, HP Domain Analysis: Producing Useful Models for Reusable Software. Hewlett-Packard Journal, 47(4), August 1996. Pp. 46-55.

[12] STARS. Organization Domain Modeling (ODM) Guidebook, Version 2.0. STARS Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defense Systems, Manassas VA, June 1996.

[13] S Fraser, D Leishman, R McClellan, "Patterns, Teams, Domain Analysis", in SIGSOFT Software Engineering Notes, Special Issue - 1995 Symposium on Software Reusability, 1995.

[14] R. Krut, N. Zalman, Domain Analysis Workshop Report for the Automated Prompt and Response System Domain (CMU/SEI-96-SR-001). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1996.

[15] N S Zalman, Making the Method Fit: An Industrial Experience in Adopting FODA, Proc. Fourth International Conference in Software Reuse, IEEE Press, Los Alamitos, CA, 1996, pp 233-235.

[16] P. Kruchten, The 4+1 View Model of Architecture, IEEE Software, 42-50, November 1995.

[17] B.S. Ku, A Reuse-Driven Approach for Rapid Telephone Service Creation, Proc. Third International Conference in Software Reuse, IEEE Press, 1994.

[18] R. Krut, Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology, CMU/SEI-93-TR11, July 1993.