VARIABILITY MODELING
IN
SOFTWARE PRODUCT LINES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

BARIŞ CAN CENGİZ KAŞIKÇI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2009

Approval of the thesis:

VARIABILITY MODELING IN SOFTWARE PRODUCT LINES

submitted by **BARIŞ CAN CENGİZ KAŞIKÇI** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. İsmet Erkmen _____
Head of Department, **Electrical and Electronics Engineering**

Prof. Dr. Semih Bilgen _____
Supervisor, **Electrical and Electronics Engineering Dept.**

Examining Committee Members

Prof Dr. Hasan Cengiz Güran (METU, EE) _____

Prof Dr. Semih Bilgen (METU, EE) _____

Assoc. Prof Dr. Ali Doğru (METU, CENG) _____

Asst. Prof. Dr. Cüneyt F. Bazlamaçci (METU, EE) _____

Tolga İpek, M.Sc. (ASELSAN A.S.) _____

Date: 07.09.2009

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: Barış Can Cengiz Kaşıkçı

Signature:

ABSTRACT

VARIABILITY MODELING IN SOFTWARE PRODUCT LINES

Kaşıkçı, Barış Can Cengiz

M.S., Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Semih Bilgen

September 2009, 134 pages

Software product lines provide enhanced means for systematic reuse when constructing systems within a particular domain. In order to achieve this, systems in a product line are expected to have a significant amount of commonality. Variability is what distinguishes these systems from one another and is spread across various product line artifacts. This thesis focuses on modeling and managing product line variability. The concept of concerns is proposed as a means of variability modeling. Another proposal is related to the use of context free grammars to represent product line variability and to guarantee that any application derived according to the variability framework thus defined will be a valid one. This approach is evaluated for an example domain, in the light of novel evaluation criteria that are also introduced in the scope of this thesis.

Keywords: Software Product Lines, Variability Modeling, Software Reuse, Software Product Line Metrics, Traceability, Concerns.

ÖZ

YAZILIM ÜRÜN HATLARINDA DEĞİŞKENLİK MODELLEME

Kaşıkçı, Barış Can Cengiz

Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Semih Bilgen

Eylül 2009, 134 sayfa

Yazılım ürün hatları belirli bir alana özgü sistemlerin geliştirilmesi aşamasında yeniden kullanım olanaklarını artırmak için gelişkin araçlar sunmaktadırlar. Bunun sağlanması için ürün hattına ait sistemlerin önemli ölçüde ortaklık bilgisine sahip olması beklenmektedir. Bununla birlikte, çeşitli ürün hattı varlıklarına yayılmış olan değişkenlik bilgisi, bu sistemlerin birbirlerinden ayırt edilmelerini sağlamaktadır. Bu tez ürün hattındaki değişkenliğin yönetilmesine ve modellenmesine odaklanmaktadır. Değişkenliğin modellenmesi için kaygı kavramı ortaya atılmıştır. Bir başka öneri de kaygıya dayalı değişkenlik modellerinin gösteriminde bağlamdan bağımsız gramerlerin kullanılması ve bu yolla geçersiz uygulamaların geliştirilmesinin engellenmesidir. Bu yaklaşım örnek bir alan üzerinde, gene bu tez kapsamında ortaya atılan özgün değerlendirme ölçütleri kullanılarak değerlendirilmiştir.

Anahtar Kelimeler: Yazılım Ürün Hatları, Değişkenlik Modelleme, Yazılımın Yeniden Kullanımı, Yazılım Ürün Hattı Metrikleri, İzlenebilirlik, Kaygılar.

In memory of

Tuna

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to:

Prof. Dr. Semih Bilgen for his criticism, comments and for his very responsible sense of supervision. Also, I would like to thank him for pushing me into further studies on the subject.

My mother, Yıldız Kaşıkçı and my father Barış Kaşıkçı for their everlasting support and love.

Executives and colleagues in the Software Engineering Department of ASELSAN Inc. and the Corporate Technology Department of Siemens AG for guiding and supporting me in this thesis.

TÜBİTAK, for their continuous support throughout my academic life.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

BNF	Backus-Naur Form
CFG	Context Free Grammar
CFL	Context Free Language
DARTS	Design Approach for Real Time Systems
DSL	Domain Specific Language
EDF	Earliest Deadline First
FODA	Feature Oriented Domain Analysis
FOPLE	Feature Oriented Product Line Engineering
FORM	Feature Oriented Reuse Method
MPP	Marketing and Product Plan
OO	Object Oriented
OOP	Object Oriented Programming
OS	Operating System
RA	Reference Architecture
OOAD	Object Oriented Analysis and Design
PL	Product Line
PR	Permissibility Ratio
PULSE	Product Line Software Engineering Method
RM	Rate Monotonic
RPR	Relative Permissibility Ratio
RR	Round Robin
SPL	Software Product Line(s)
SPLE	Software Product Line Engineering
SSF	System Specification File
STL	Standard Template Library
VP	Variation Point

CHAPTER 1

INTRODUCTION

1.1 Software Reuse, Product Lines and Variability

Efficient reuse in software has always been subject to abundant interest. The primary aim in providing reusable software artifacts is obviously for reusing them in products with significant commonality and possibly in families of products. The common term when referring to the collection of such family of systems is software product lines (SPL).

As explained in [11], a slight distinction exists between SPL and software families. The term software family is related to commonality of the asset base shared by the constituents of the product family. On the other hand the term “product line” is rather market-related. It describes a set of products sold on the market. Thus, it is common that a product line (PL) encompasses more than one product family, in accordance with market needs. However both product lines and product families demand a significant amount of asset commonality. The rest of the text uses terms product line (PL) and software product line interchangeably, since in this scope and domain, they are identical.

Typical software system development processes focus on the delivery of a single product to a single customer, which means that such processes do not consider much the issue of reproducing reusable artifacts. Among others, processes centered on Object Oriented Analysis and Design (OOAD) are very popular (Object Oriented Role Analysis Method [75], Rational Unified Process [76], Object Modeling Technique [77]…).

It is a very common criticism from the viewpoint of PL centered approaches that, the classical OOAD based approaches don't provide enough guidance in developing reusable artifacts [69]. Furthermore these classical OOAD methodologies are far from being lightweight ones. They have artifacts related to requirements, model, code and many others. Certainly, PL approaches bring in some additional overhead and artifact support compared to the ones aimed at delivering a single product. One of the many challenges in dealing with PL approaches is to manage the additional overhead in a systematic manner.

A central point of interest other than the commonality in PL's is variability. Arguably it is the true essence of a PL. Variability is the ensemble of properties of products which make them different from others. Carefully examining the variability and systematically relating it to the PL process artifacts, is a fundamental responsibility of a successful PL.

However, as stated in [12], there is a clear problem of variability tracking when current PL based approaches are considered. The variability information is presented through so called features – see section 2.2.6– and most apparently documented in feature diagrams as initially explained in [14] as an AND/OR hierarchy of features. This method of presenting the variability information introduces yet another level of abstraction in addition to the classical abstractions of software artifacts (requirements, use cases, architecture, design and source code). However it should not be forgotten that the variability information is present across all artifacts and they are all conceptually linked to each other. The need of a mechanism relating all the variability information spread through several artifacts is required for the tracking and retrieval of that information. This need corresponds to the understanding of variability traceability within this work. Achieving this traceability is one of the primary goals of this thesis and foundations to realize this goal are explained.

In light of the issues that are just presented, some questions arise: How to completely and conveniently model and express PL variability? Is it possible to come up with an approach for formally and powerfully expressing variability across multiple PL artifacts? What are the benefits of such an approach in terms of traceability? How would an existing product line approach benefit from the enhanced traceability support for its variability information? Does such an approach add an important value to the conventional PL modeling techniques? Answers to these questions are sought throughout this work.

1.2 **Purpose of the Study**

In this thesis, the concept of *concerns* is introduced as a means of modeling variability of PL artifacts and establishing traceability among these artifacts. Although further discussed in Chapter 3**,** a few words are necessary about *concerns* to clarify the concept and to better grasp the purpose of this study.

Concerns are an extension of decisions. Decisions were discussed originally in [20] as an instrument for better variability modeling, supporting the traceability of variability among PL assets that lie in different levels of abstraction of software artifacts. They are proposed in the scope of this thesis and their usefulness for representing variability in a PL and system specification is discussed.

Concerns provide more detailed traceability information than decisions do. They separate the variability information into two complementary categories, namely that of horizontal and vertical variability, thus providing more complete view into the assets that make up the PL. Also, ultimately they are expected to initiate the formation of a domain specific language (DSL) for specifying new products with greater ease and further accuracy than decisions would do.

That being said, this study intends to introduce *concerns* as a more general replacement for the decisions mentioned in [12]. It is argued that any approach aimed towards the formation of a PL –or more specifically towards managing variability across a PL– will benefit from the usefulness of *concerns*. Among others, most expected yields of this approach are, more realistic set of possible products due to the fact of many artifacts' interactions' being revealed; reduced feature interaction problems; further ease of specification and composition of new products due to span of *concerns* over all artifacts.

Another proposal presented in this thesis is the representation of PL variability using context free grammars (CFG). In this way, validity of systems developed in the PL, represented as strings generated via the CFG, may be effectively controlled.

1.3 **Outline**

The rest of the thesis is organized in the following way:

Chapter 2 presents a literature review performed in order to grasp notions related to PL, reuse, aspects, embedded systems, frameworks, reference architectures (RA) and other relevant topics. Embedded systems and some other related literature terms are introduced because the experimental work in this thesis is related with this field. Other than that, the ideas presented in this work are applicable to any major software development sub-domain (web applications, user interface design, network technologies).

Chapter 3 introduces *concerns* as a means of tracking software PL variability. The discussion builds over decisions and extends them to introduce *concerns*. Use of CFG's for representing PL variability using *concerns* is discussed.

In Chapter 4, experimental work involving the study of real time scheduling algorithms (RTSA) domain is presented, first, supported by a classical PL approach, namely Feature Oriented Reuse Method (FORM) and then, with the same approach supported via *concerns*. Reasons for selecting this particular domain and PL approach are also indicated.

In Chapter 5 two novel evaluation criteria for PL variability modeling effectiveness are proposed. The success of the PL approach using *concerns* is compared to the bare PL approach using these evaluation criteria.

Finally, Chapter 6 concludes the subject by reviewing the contributions, achievements and shortcomings of the work. There is also an investigation for possible future research and application in this area.

CHAPTER 2

LITERATURE SURVEY

2.1 Introduction

The following section presents a survey of topics that are relevant to the subject of this study. Among others, software reuse, software product lines and decisions are discussed by emphasizing on the interrelations of these subjects, whenever necessary and possible. Real time embedded systems modeling approaches are also briefly reviewed because of the pertinence of the subject in the context of the experimental application of the proposed variability modeling approach.

The idea relayed by decisions and *concerns* and the purpose of this study are tried to be linked with each possible item that is presented in the literature survey.

Aspects and aspect oriented programming find their place in the following discussion because they have conceptual resemblance with the concept of *concerns* and decisions that are central to this study. They complement each other in many ways. This resemblance is further clarified in the survey.

2.2 **Literature**

2.2.1 Software Reuse

The root of the discussion for this work lies in efforts for establishing a convenient and efficient reuse environment. Software reuse is defined as the usage of previously acquired software artifacts or knowledge in order to construct new ones [63].

The first solid reuse idea was presented in [64], where components have been proposed as a means of achieving so called "software industrialization". It has been argued that, in order to achieve useful reuse, software components that are classified in terms of precision, robustness, performance and other parameters are needed to be constructed.

The evolution of the understanding of software reuse is particularly interesting for the sake of this work since it brings the discussion to PL's and proactive ways of treating reuse, which are central ideas that this work is based on. Traditional software reuse can be classified into two major categories as code reuse and conceptual reuse.

Code reuse can mean reusing bare code or organizing it in the form of libraries or more elaborately as frameworks. Bare code reuse generally relies on the knowledge of the software people so it is expected to be inefficient in terms of staff utilization. Furthermore it can be risky for its adopters, considering possible organizational restructurings. Using libraries to achieve reuse is a more structured approach compared to using bare code, yet libraries allow a small amount of extendibility. Also applications built on top of these libraries tend to be highly dependent on them. Frameworks described in section 2.2.14 are more sophisticated and they have relations with SPL's.

Conceptual reuse means reusing the abstract knowledge about a problem and its solution. This kind of reuse dates back to the introduction of software design patterns [65]. Software design patterns denote the ensemble of problem, forces, solution and consequences to recurring

software problems. They are not reused as code; rather they are needed to be integrated to designs according to the needs.

A more abstract version of conceptual reuse is that of architectural design patterns [66]. They are the highest level patterns and they define the fundamental application structure. Generally such architecture is composed of several views and is considered to be a vital part of a software system and therefore needs consideration in early phases of system development.

Ideas of design patterns and ultimately architectural reuse were influential for the foundations of Software Product Lines (SPL) and reference architectures which are further discussed in sections 2.2.2 and 2.2.13 respectively.

In short, a product line (PL) aims to achieve what is called proactive reuse. Reuse idea is meant to be integrated to processes and tools. Furthermore the goal is to reuse several aspects of software such as requirements, test cases, components, features, scenarios and not only code or patterns as in the classical approaches.

2.2.2 Software Product Line Engineering (SPLE)

The main driving force for constructing software PL's is to satisfy varying customer needs. The fundamental question to answer is how to produce more and more personalized products at a lower cost with greater ease. The reuse scope in SPL's is broad. It can include requirement, feature, component, test-case and architecture reuse. The essence of a solid PL lies in the thorough exploration of commonalities and variability for the considered family of products. A sound amount of commonality is essential for establishing the base of a successful PL. SPL commonality manifests itself in the form of common modules, components, classes in PL as described in [9]. As indicated in [8] and restated in many other resources [17], [14] and [29]; Software Product Line Engineering (SPLE) framework has two major engineering processes:

1. Domain engineering, where commonality and variability for the PL are explored. A reference architecture is established at this point. In brief, essential domain engineering activities are:

- Domain Analysis: Definition of commonality and variability for the product family. The variability must be explicitly documented. This point is critical for establishing the traceability of variability.
- Domain Design: Establishing a reference architecture.
- Domain Realization: Detailed design involved in constructing reusable components.
- Domain Testing: Quality assurance purposes, at PL level.

It is important to note that these activities are not sequential. Moreover, sometimes they are not present and sometimes they are more elaborate or further decomposed into sub-steps in several PL approaches[1].

2. Application engineering, where the actual product is built by reusing the domain assets and focusing on the PL variability. Variability is resolved in domain engineering artifacts to lead to concrete products.

It can be fairly stated that domain engineering is the principal theme around which all PL approaches revolve. Application engineering is expected to be rather trivial once the domain engineering internals are solid and the mapping methods from the domain to the particular application are well described. Traditionally, domain engineering stands for SPLE. Although the first explicitly named SPL methodology was PuLSE [70], other methodologies which were essentially presented as domain engineering approaches (e.g. Feature Oriented Reuse Method (FORM)), cover the essential core items of SPLE. For such historical reasons, the terms domain engineering and SPLE could be used interchangeably throughout the text.

It is possible to observe previously mentioned PL phases in Figure 2-1, this figure is rather a mixture of FORM Engineering processes in [27] and the SPLE framework presented in [8].

[1] In the scope of this work, domain testing is not considered.

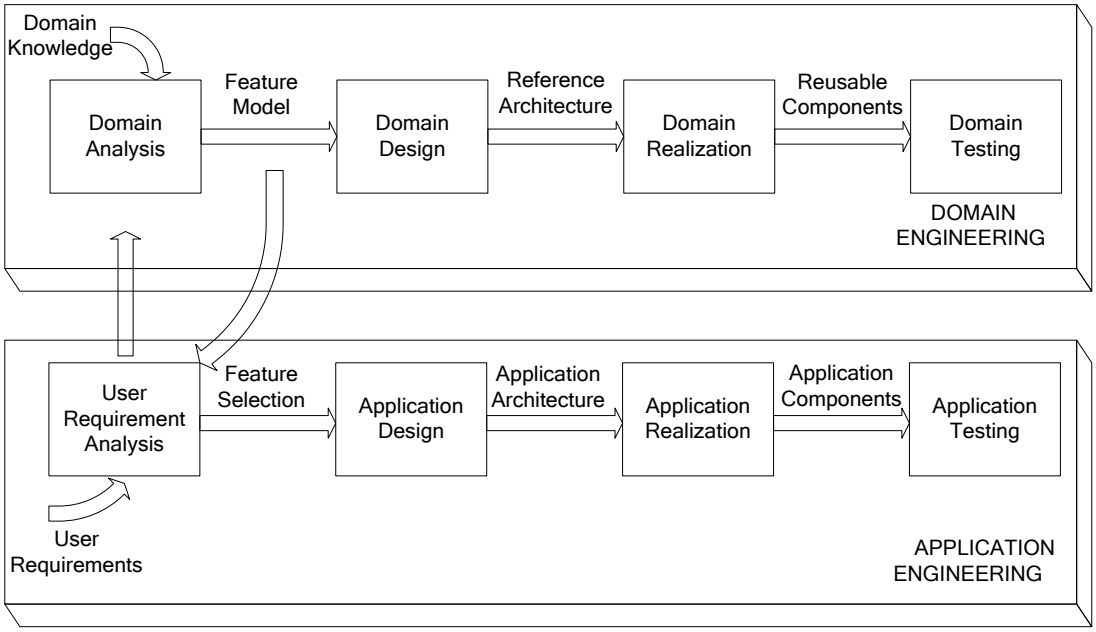

Figure 2-1 Software Product Line Engineering Processes (Adapted from [34] p. 7 and [27] p. 8)

The items depicted in Figure 2-1 are encountered almost in every description of SPLE processes. The primary operation is to identify domain requirements in order to fully understand the domain (domain analysis). Then a methodology to indicate how to build related systems by expressing which items will be present considering the needs of the particular domain needs to be established (domain design). Then the bits and pieces to build those systems must be considered (domain implementation). Finally testing at a PL scope is required (domain testing).

The arrow from the application engineering process to the domain engineering process in Figure 2-1 indicates that domain engineering is open to any evolutionary feedback from application engineering. Also note that the arrow from the feature model to user requirement analysis indicates the usage of the feature model in requirements specification.

Obviously SPLE is not the best solution for any software problem. There are cases when it is more feasible to proceed without SPLE support. However any software process adapted at an institution, which has the potential to develop a family of software, is expected to benefit from domain engineering –thus SPLE–.

Several SPLE methods are available which fit to the previously outlined process template. In the following discussion, some of these will be briefly reviewed. Particular attention to FORM and Feature Oriented Domain Analysis (FODA) [14] is paid, for the former is extended and used as the example method to demonstrate the usefulness of *concerns* in variability traceability management and the latter is the parent method of the former.

A chronological layout of Domain Engineering and SPLE methodologies is presented in Figure 2-2 to give a temporal evolution idea. A selection of the SPLE methods is discussed later in the literature survey.

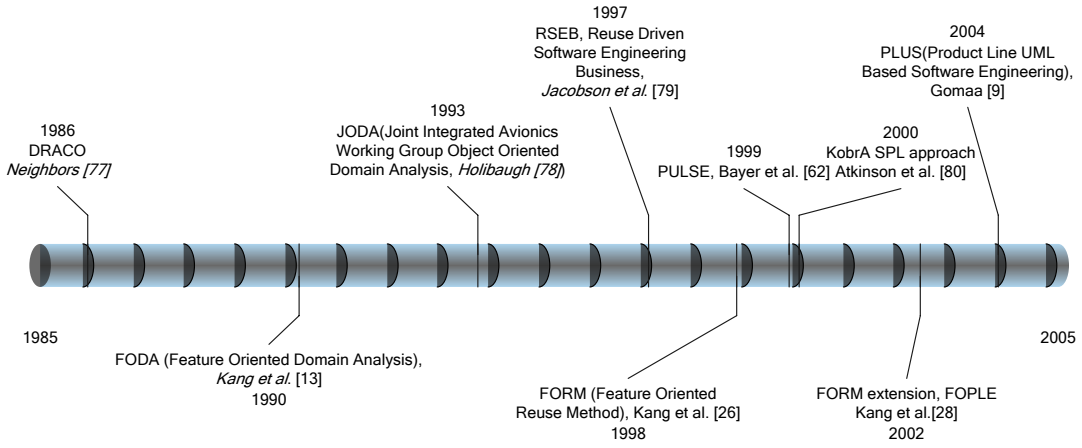

Figure 2-2 Domain and Software Product Line Engineering Timeline

2.2.3 Variability Tracking

As indicated in the section on SPLE, variability –and therefore its management and tracking– is a fundamental element of a successful PL. Traditionally, PL variability is defined as the ensemble of differences in between systems that belong to a PL in terms of features and requirements [55]. The item that varies in a PL is termed as the variation point (VP) whereas the actual instance of this variation –the actual value that this variation takes- is called the variant [56].

As it is just indicated, traditionally, SPL approaches such as FODA or FORM [27] attribute this variability to diversity of features of the systems that make up the PL and try to express individual systems in terms of these features. These approaches use features to parameterize their artifacts.

However, as indicated in [9], PL variability actually has different characteristics, when requirements and design are considered separately. Techniques developed for modeling such variability are feature modeling for the former and parameterization techniques for the latter.

Furthermore, in [56], it has been stated that the variability in a PL requires explicit decisions from product management, software architects, developers and maintenance staff. This ultimately brings in the need of variability management at a broader scope that covers several artifacts. It has also been indicated that the complexity of variability increases as the refinement level of PL artifacts increase along the so called "variability pyramid". The layers on the mentioned variability pyramid are features, requirements, design, realization and test in the order of increasing refinement. Therefore, the importance of consistent and correct tracking of variability across various levels of refinement is emphasized. In order to achieve this, particular attention must be paid to conditional implications of multi layer variability in a PL –such as the inclusion of a particular feature because some other feature has been selected– as well as to the methods of representing the variability across different artifacts that form the PL.

However, as evaluated in [12], modeling PL variations with the aid of features and feature models lack consistency and scalability and most importantly traceability. This problem has also been investigated by the work in [57] and it has been stated that feature models capture both

commonality and variability information which complicates them. Furthermore the inability of tracking variability information consistently across many PL artifacts with mere feature usage has been outlined.

Such limitations of feature models in tracking variability ultimately led to the introduction of several approaches such as Orthogonal Variability Model [56] and decision based modeling [12]. These approaches both consider variability separately from all the artifacts and relate this information to those artifacts to be able to track it at an independent level. The concept of *concerns* takes decisions - section 2.2.4- as primary foundation and extends it in several ways as described in Chapter 3.

2.2.4 Decision (Decision Point)

The concept of decisions is crucial for modeling the variability and establishing its traceability to relevant PL artifacts. Decisions are presented in [20] as being modeling concepts that extend variation points. They constrain other variation points –which is an attribute of variation points as described in some other PL methods– and they provide a question which must be answered at the application engineering time in order to resolve the variability associated with the decision. Furthermore, a decision model is defined as a model that captures and tracks all variation points of all assets in a PL. By all assets, a range from product definitions to implementation and to user manuals is implied. Figure 2-3 shows a metamodel of a decision.

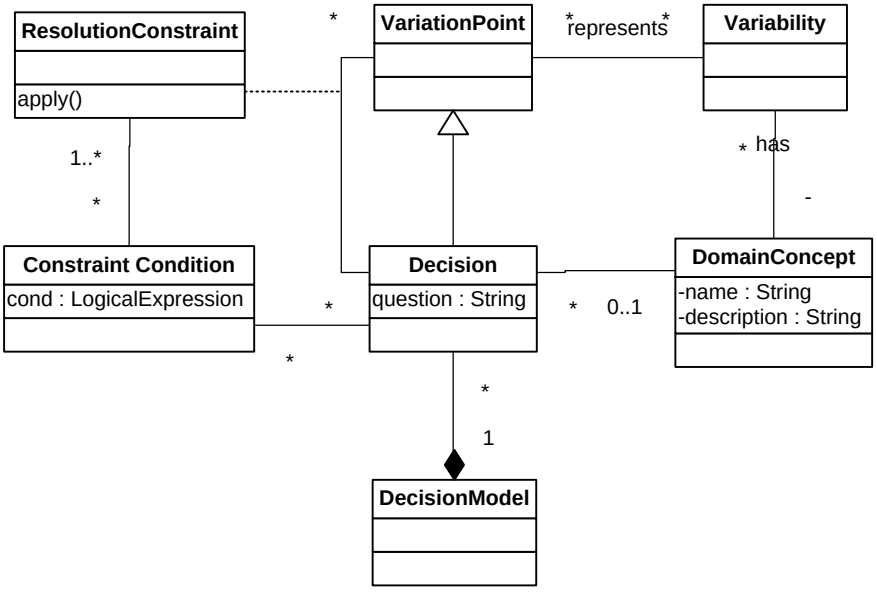

Figure 2-3 Decision metamodel (adapted from [20], p. 124)

Decision metamodel is quite readable. A decision model is composed of several decisions and possibly their interdependencies. As pointed out before, decisions are generalizations of variation points. Variation points are represented via variability. Domain concepts have several variability items and are obviously associated with some decisions. The discussion of decisions further introduces simple decisions and dependencies (decisions, whose resolutions affect or depend on the resolution of other decisions). The constraint that ties a decision to another decision is called a resolution constraint. In the decision model this is how dependency between decisions is expressed. A constraint condition is used to represent the logical expression that relates several decisions –e.g. as alternatives–. Hence, resolution constraints make up constraint conditions.

The question associated with the resolution of a simple decision is a simple "yes or no" question. In the case of a dependency, a decision constrains several variation points and therefore a simple

question is not able to express the variability relations that need to be resolved at application engineering time and more elaborate expressions are used.

Although it is true that feature models represent a "decision space" for software development as indicated in [27], decisions –also *concerns*– are more successful in doing so because they track variability further into artifacts due to their multilayer-spanning nature.

Concerns are introduced, because it is believed that the variability information within a particular abstraction level of an asset cannot be accurately handled by using decisions. This is simply because the resolution of variability[1] at a particular asset abstraction level may not be unique, or may require the resolution of several decisions within that level. This is one of the points that are addressed by *concerns*.

Also it should be noted that decisions are good candidates for forming domain specific languages. Potential candidates could range from easy to use, check-boxed decision based user interfaces to more sophisticated generative languages with decisions –or *concerns*– as their foundation. The reason for this is that it is convenient for one to specify products in terms of the vocabulary and artifacts that are derived from that domain.

2.2.5 Domain and Domain Specific Language

The dictionary description of the term domain is "a sphere of knowledge, influence, or activity" [31].

Prior to being considered in the scope of PL's, domains were considered as being the real world counterparts of software entities and they were modeled in an artifact called domain model in the scope of object oriented (OO) analysis. A domain model, as explained in [19], describes all the

[1] The expression "resolution of variability" means selecting the appropriate variability items at application engineering stage to come up with the actual product from the variability model of the PL.

domain object types and the relationships among their instances, which collectively describe the domain space. In OO context, a domain model is a representation which is human readable and which, when read, makes sense in describing the solution to the problem in question. Domain model is a pictorial representation of the problem in terms of the conceptual objects (not necessarily objects as in object oriented programming (OOP)) that constitute the problem domain. Although they are not meant to be software objects, they have considerable influence over actual software objects' derivations.

The second interpretation of the term domain is central for the discussion of PL's. Domain, in the context of SPL defines the scope of systems that are included in the PL. According to [17], the term domain, when considered in the scope of SPLE, also encompasses the knowledge to build software systems in that particular area. Therefore, better understanding of a domain that makes up a family of software systems, is expected to increase the chances of reuse for that particular family.

According to [18] , domains are classified into the following complementary classes:

1. Vertical domains span entirety of systems. As noted in [17], the application of domain engineering to a vertical domain is likely to result in reusable software in the form of domain specific framework –or at least initially a reference architecture–.
2. Horizontal domains occupy defined parts of a system. Therefore if domain engineering is applied to a horizontal domain, it will most probably yield reusable components.

The distinction between vertical and horizontal domains can be observed in Figure 2-4. In this picture the stripped parts represent the domain scope whereas the solid parts are system scopes. It can be seen that in the case of vertical domain, domain scope is identical to system scope.

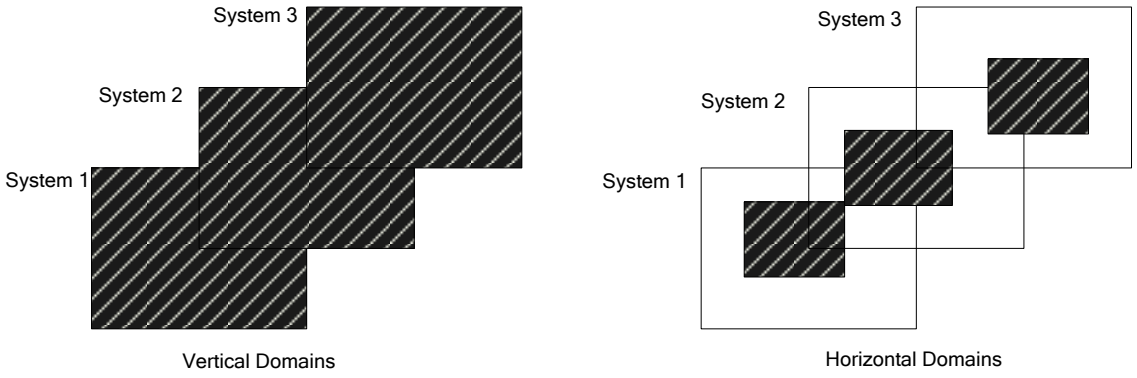

Figure 2-4 Vertical and Horizontal Domains (adapted from [17] p. 42)

A horizontal domain could further be decomposed as:

1. Encapsulated domains which are localized properly by maintaining their integrity inside the systems
2. Diffused domains which are made up of dispersed constituents.

The distinction between an encapsulated and diffused domain is depicted in Figure 2-5.

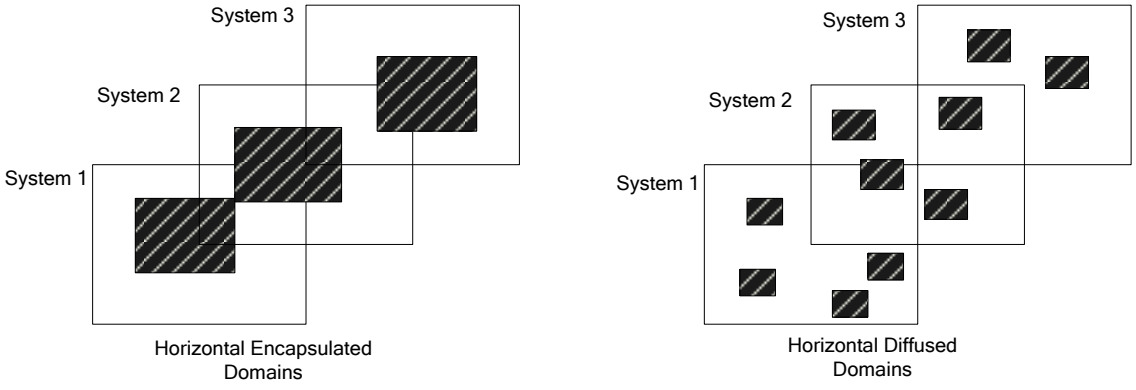

Figure 2-5 Horizontal encapsulated and diffused domains (adapted from [17] p. 42)

An essential activity in SPLE is to determine the scope of the domain for the PL. Several domain analysis methods have a dedicated process component for the identification of the domain scope such as FODA [14]. FODA defines this particular process component as context analysis and provides several context diagram-alike output artifacts in order to depict the scope of the domain.

A topic related with domain and SPL's is Domain specific languages (DSL). As the name indicates a DSL is a particular language aimed at satisfying the needs of a certain domain. As stated in [53] a DSL brings enhanced expressiveness to the domain it represents. Some famous DSL examples are Microsoft's Excel Macros [51], Linux Kernel Configuration interface (*menuconfig)* [50] and HTML. Also as indicated in [52] many other examples are available such as PIC, SCATTER, YACC, CHEM, LEX, and Make.

DSL's are usually considered as being declarative languages and their usage generally leads to the generation of applications. The final applications can be complete binary entities. For instance, Linux kernel configuration tool will allow the Linux kernel code to be generated and therefore the binary file associated with it. Another possibility is the generation of partial skeletal code structure that needs further refinements. As an example, a unified modeling language (UML) profile that allows a particular domain to be modeled can be considered. In such a profile, in order to finalize the application, further coding is necessary.

2.2.6 Feature

A feature is defined as a prominent part or characteristic of an entity [31]. The idea of using features as a means of specifying system characteristics is a central one for the purposes of SPLE methods. The first to do so was FODA [14], which uses features mainly for requirements engineering. On the other hand, FORM [50] bases the entire lifecycle including component parameterization and realization on features.

As indicated, in FODA, features are considered to be somewhat more end-user visible capabilities of applications. FORM, being a successor to FODA adopts the same definition and

elaborates slightly indicating features are unique functional abstractions that are used to communicate with the users, which should be implemented, tested, delivered and maintained.

In Product Line UML based Software Engineering (PLUS) [9], a feature is considered to be a requirement or a characteristic that distinguishes the members of a PL, thus clarifying the commonalities and variability among them. It is expressed clearly in PLUS, and it is a known fact from the FORM and FODA methodologies' perspective that features are used to track variability and establish commonality across constituents of a PL.

There are several clusters of features identified by different methodologies. Based on [14], these can be listed as:

1. Services
2. Operations (user interactions)
3. Presentation (what part(s) of the system exposed to users)
4. Performance
5. Hardware platform
6. Cost
7. Quality

The idea of using features for system specification is a natural one. As an application domain matures, the developers, analysts and users tend to describe relevant systems in terms of well known terminology. For example if we consider the domain of operating systems scheduling algorithms, a partial system specification could be "priority based preemptive scheduling with round robin support".

In FODA and FORM, the fundamental shortcoming related to tracking variability by features is that such an approach merely considers users' views of variability. Thus it won't help in tracking some fundamental aspects of system properties deep into other PL asset levels. Although a layering of features is present both in FODA and FORM, this is a layering for the user's point of view of systems in the PL and thus still does not cover PL artifacts completely.

As it has been the case for the work in [17], feature definition and understanding of Organization Domain Modeling (ODM) is more suitable for the transition of one's state of mind from features to decisions –and ultimately to *concerns*- for tracking and modeling PL variability. In ODM, features are again considered to be uniquely identifiable properties of a system yet from the point of view of any stakeholder. What are relevant to a particular stakeholder may be just requirements, architecture, models or code and not only features. This is particularly true for domains where features –deemed as user visible capabilities– are not well suited for modeling variability. One such large domain is embedded systems software in general where user involvement is minimal and the parties involved into interactions with systems are other components, sub-systems or systems.

2.2.7 Feature Modeling

Feature modeling intends to demonstrate the common and variant features and the interrelationships in between these features, possibly on a convenient model. This concept has been introduced in [14]. For the purposes of this work, feature modeling techniques of FODA – and therefore FORM- are relevant, therefore in the following discussion a summary of these techniques is presented as a feature modeling example. For a further discussion of various feature modeling techniques, a comprehensive collection can be found in [49].

For feature modeling, FODA is mainly focused on services provided by the applications and the operating environment that those applications run on. Features are organized in a diagram called the feature model. Features are classified in three major categories, namely mandatory, optional and alternative (feature sets among which only one can be selected) features, along with compositional rules such as "mutual exclusion" and "requirement" in between them. An example FORM feature model could be seen in Figure 4-4[1].

[1] Mutual exclusion and requirement relations are shown using UML stereotype notation in feature diagrams in this work without using a specific diagrammatic line style as in FORM.

FODA features are also categorized into three main classes in terms of their “bind time”. (By “bind time” the actual instantiation of the feature model for resolving the variability to represent a real product is meant). These are:

1. Compile time features: features fixed in the packaging of the software.
2. Load time features: features selected or defined at the beginning of the execution of applications.
3. Run-time features: features that could be changed at operation time.

Also, FODA and later on FORM classify features into four other main categories classified according to levels of abstraction. These are operating environments, capabilities, domain technology and implementation techniques.

Then, the task is to find features matching these categories; classify them and validate the entire feature model against several applications that fit into the domain. At this point an input from domain experts is necessary.

2.2.8 FODA (Feature Oriented Domain Analysis)

As the name of the methodology implies, FODA is a domain analysis method with feature analysis as its prime driver. It dates back to times when the term “software product lines” were not common and it was launched as being a domain engineering methodology. FODA arose out of the need to discover commonality across several related applications, which forms the foundations of a solid PL. FODA, has been described in [14] and the following discussion constitutes a brief summary of the original text.

Three principal activities (process components or sub processes) are defined in FODA as being context analysis, domain modeling and architecture modeling. These components as well as their resulting products are illustrated in Figure 2-6.

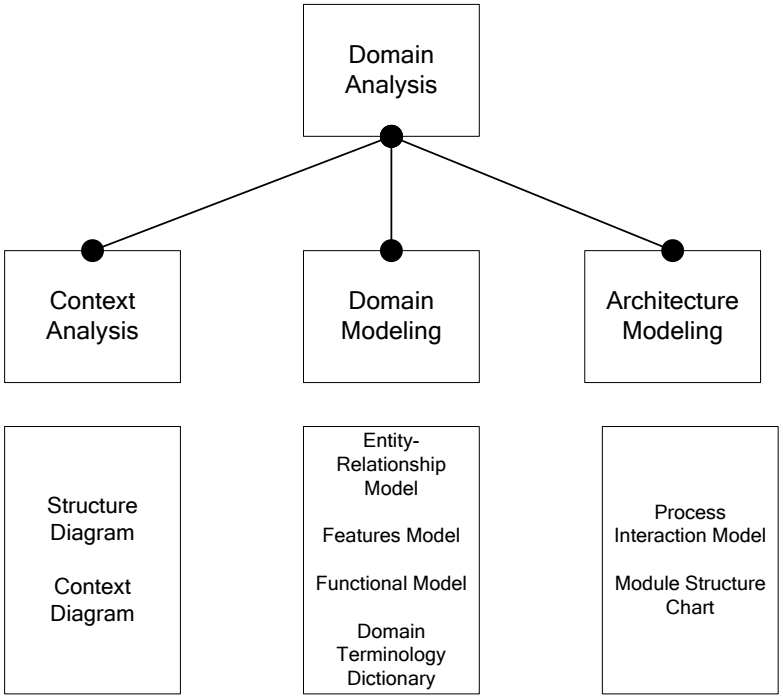

Figure 2-6 Domain analysis in FODA (adapted from [14] p. 7)

During context analysis, the primary objective is to determine the scope of the domain. This includes many things such as project constraints, identification of stakeholders and most importantly determining the scope of the product. This sub process' outputs are structure diagrams, context models and block diagrams.

After the scope of the domain has been determined, in the domain modeling phase, commonality and variability for the systems that make up the domain are explored. Domain modeling phase is further broken down into 3 modeling activities. These are feature analysis and modeling, entity-relationship modeling and functional analysis.

FODA feature modeling has been described as part of feature modeling in section 2.2.7 which treats the subject individually.

Second task in domain modeling is that of entity relationship modeling. Entity relationship models usually consists of domain objects (entities) and their relationships such as “is a” and “consists of” and their primary goal is to represent the domain information in a more readable form. Entity-relationship diagrams are inspiring for the actual system structures.

Final task for domain modeling is functional analysis of the domain. This task is necessary in order to specify functions and behaviors that must be fulfilled by the system. Functions are represented via activity diagrams and behaviors via statecharts. Feature modeling and entity relationship modeling has lots of inputs for functional decomposition. FODA also demands that features be integrated into the functional model. In this way traceability between the functional specification and feature model could be established. Also an analyst dealing with the specification of functions needs to understand the consequences of his/her decisions in constructing an actual product out of the functional model. The same argument is valid for the feature model too. For this purpose, FODA provides decisions –not the decisions in section 2.2.4– in order to make clear for the analyst the various implications of actual calls.

It should be noted that by integrating features into functional model, variability information present in features are somehow inserted into the model too. However it should also be understood that the level of variability introduced into functional specification is limited with that of features.

The final activity in FODA is architecture modeling. This activity aims to produce a reference architecture as an artifact from which it would be possible to bring the design to a more clear and detailed level and ultimately construct the components. FODA aims to construct a four tier architecture.

In this layered structure, FODA focuses primarily on the top two levels. The highest level, namely that of domain architecture, focuses on processes at the domain scope, their concurrent behavior and interconnections among them. The level below that is the domain utilities layer and it shows the decomposition of functions and objects residing in modules and the module interactions. Elements of the common utilities layer are common across domains and those of systems layer are lower level facilities such as the ones provided by the operating system.

FODA suggests embedding feature-related information into the architecture model. Also, FODA acknowledges the fact that many of the design decisions are left out until implementation and that the architectural modeling phase of the methodology needs to gather these in the form of issues –standing partially for decision points described in 2.2.4– and decisions – standing for actual instances of decision points with resolved variability– in the model.

FODA uses Design Approach for Real Time Systems (DARTS) method described in [28] to construct mainly the domain architecture and domain utilities layer of its architectural model. However this method is not applied in the experimental work and for further details one can refer to [14] and [28].

Another important thing that is mentioned in the feasibility study is that the domain analysis efforts' outputs should be verified against at least one product that has been left out during the analysis yet still believed to belong to the domain. Such an effort is necessary to demonstrate the applicability of the analysis to products that are likely to fall into the same domain.

2.2.9 FORM (Feature Oriented Reuse Method)

FORM makes the distinction of domain engineering and application engineering clear as it is depicted in the general template for SPLE methodologies in Figure 2-1. Actually FORM is proposed as an extension to FODA. Being feature oriented, FORM tries to capture the commonalities and variability among several systems in a domain in terms of features just as FODA does and from that information it tries to build domain architectures and components. FORM has been initially introduced in [27] and the following is primarily a summary of the original text.

The construction of a feature model is essential in FORM as it is in FODA. Previous methods mostly focused on the usage of features for the requirements engineering phase and had little or no emphasis on software design and implementation. Also, FORM explicitly mentions the application engineering process that represents the process flow related to individual applications. Therefore what FORM tries to achieve is to map the feature based analysis into

modules that are identified by those features which have the capability to represent each product in the PL.

FORM identifies several so-called engineering principles to construct architectures and components. These principles are an improvement that FORM brings over FODA. Briefly summarized, they are:

- Classification of reusable artifacts in different abstraction layers. These are subsystem models, process models, module models, module specifications and implementations.
- Parameterization of artifacts using features. Using this method, features are embedded into components and remain untouched until instantiation. Therefore it is possible to relate feature selections with the component instantiations. Here it should be noted that FORM acknowledges that this parameterization could be achieved up to only a certain degree. The limitation is due that of features and their limited power in expressing variability.
- FORM advises the usage of a layered architectural framework. This layering is compliant with the layering of features as described in FODA. The basic layering of the architecture in FORM is:
    - Subsystem
    - Process
    - Module

It should be noted that parameterization of artifacts is a central idea in all PL centered approaches. The parameterization technique should be appropriately selected according to the needs of the particular domain that is under consideration.

Regarding the process aspects of FORM, as in all PL approaches, two principal engineering activities are present. These are domain and application engineering. The flow is practically the same as that in Figure 2-1 with several details and omissions inherent to the methodology itself.

FORM mentions nothing related to the testing of a family of products neither in the domain nor in the application engineering phases. This could be considered as a lack in a complete PL approach but this is another debate. This information is provided so that the Figure 2-1 could be

regarded with a filtered view, for it represents a more general picture. At the time of the writing, testing in PL's is another hot topic of discussion which is out of the scope of this work.

FORM elaborates on the original model specified by FODA although the main activities are the same. FORM considers context analysis and feature modeling as a whole to contribute to the "feature space" whereas architectural modeling is considered as belonging to the "artifact space".

Concerning first of the feature space related activities, namely context analysis, the receipt for its success is indicated as choosing a domain with high amount of commonality. The question is, how to be sure about the amount of commonality in a domain? FORM considers domains with available standards and well established histories as wise candidates for successful PL's.

Following the domain context determination, the essential activity of a PL, namely that of feature analysis should be performed. This is described in section 2.2.7 on feature modeling which describes this process for FODA which is ultimately adopted by FORM.

Feature classification of FORM helps in identifying the systems from different points of view. Capabilities could be thought of as the services provided by the systems, whether they are functional, non-functional or performance related. The operating environment features are pretty clear from their name, they are related to the platform and operating systems related aspects of systems. Domain technologies and implementation techniques are more low level and their respective feature classes are apparent from their names.

Moving on to the artifact space, domain engineering has the principal responsibility of clearly establishing a reference architecture. To better grasp FORM's layered architectural approach, the reference architecture spot of Figure 2-1 needs elaboration. This elaboration can be seen in Figure 2-7. (For more detailed discussion on reference architecture concept, refer to Section 2.2.13)

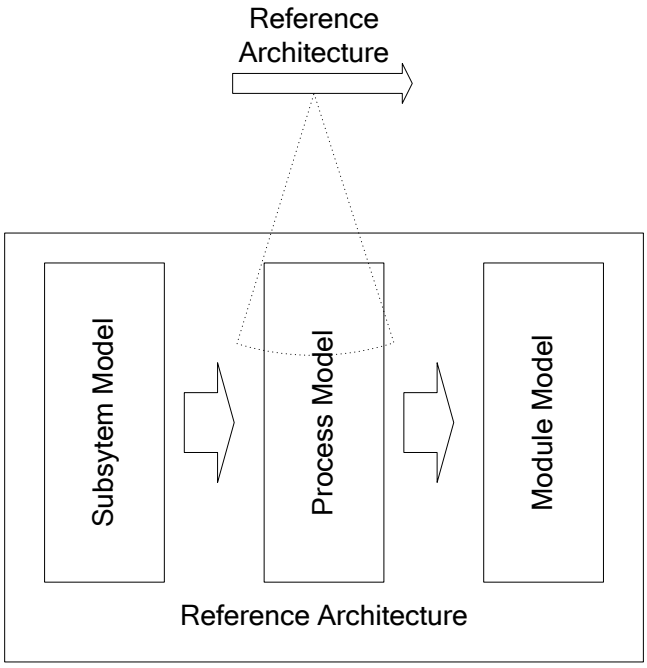

Figure 2-7 Detailed Reference Architecture in FORM (adapted from [27], p. 8)

As it is the case with all PL approaches, one of the most crucial activities is to map features to the architectural components. In doing so, FORM adopts two major philosophies:

1. Differentiate between functional and non-functional features. The former will be used to identify the components and the latter to classify them and determine types of connectors in-between them. This phase makes use of the previously mentioned engineering principles of FORM.
2. Use the feature hierarchy to map features into FORM architectural layering. Briefly each level deal with the following:
    a. For the subsystem model, fundamental decomposition of the system functionality must be considered. For this model, prime input is the capability related features.

b. Process model is related with the internals of a subsystem. Obvious from its name, this model aims to achieve a decomposition of features with regard to factors such as transiency, residency, periodicity, performance among others.
c. It is particularly important to note that, module models are affected by the feature models of all levels. As a matter of fact FORM underlines certain similarity between the feature decomposition hierarchy and the decomposition within a module. Module structure's feature parameterization could be done via popular parameterization techniques such as generalization or template instantiation.

For what concerns application engineering, user requirements related to a specific application must be identified and features from the feature analysis should be matched with these requirements. The feature model - user requirement matching mechanism should check any discrepancies that might arise due to restrictions in the feature model and guide the users. A more sound idea about FORM application engineering can be obtained from the experimental work in section 4.4.

2.2.10 Feature Oriented Product Line Engineering (FOPLE)

FOPLE is an extension to FORM which itself is an extension to FODA. Actually the work in [29] introducing FOPLE, does not represent major novelty over FORM, rather it simply polishes the method to fit better with the contemporary ones. It introduces the fancy term "product line engineering" in FORM by taking into account marketing and product planning aspects of a PL. It was mentioned in the introduction (Chapter 1) that the term software product lines is mostly related to market aspects. PuLSE made this term more popular, and FORM, which already was a SPLE methodology, officially acquired that title.

As a part of FOPLE, Marketing and Product Plan (MPP) is introduced. MPP is described as the initial step in gathering PL assets. As the name indicates a marketing plan is an artifact that states clearly the potential possibilities for selling the product. This includes identifying the potential user profile, clarifying major high level features, quality issues and legal constraints among others. MPP requires feedback from PL requirements analysis. Also MPP fixes some

quality attributes that should be present in all aspects that make up the system. Therefore, it is of fundamental importance from a user's point of view.

The product plan phase of MPP is related to more specific product features that are suitable for the previously established marketing plan. Feature analysis is also a key factor in achieving this. It helps developing products that are scalable, so that they can be related easily with many users' requirements.

2.2.11 PLUS

PLUS is one of the most recent and comprehensive PL engineering approaches that takes advantage of UML constructs. It has been described in [9] and the following discussion briefly summarizes the methodology. PLUS methodology is covered in the literature survey because it emphasizes traceability of feature variations throughout the PL process which pertains to the goals of this work.

In PLUS, one can identify five major phases for constructing a PL. These are requirements modeling, analysis modeling, design modeling, component implementation and testing. As with all PL approaches, this flow manifests itself in two lanes, namely that of domain and application engineering.

The key novelty about PLUS is that it uses UML constructs such as use cases, class diagrams, collaboration diagrams and many others to represent PL concepts such as PL requirements, feature diagrams and object interactions among PL objects. PLUS introduces several stereotypes –such as optional or mandatory– in order to express relationships that are common among PL artifacts.

In the requirements model, use cases and features are related to provide traceability. Furthermore, for each feature that is identified in the requirements modeling phase, traceability must be established between it and the software object that implements this feature in the analysis modeling phase. Hence, PLUS applies the idea of variability tracking in several PL

artifacts such as requirements and design elements where features are mainly used to explore the variability.

2.2.12 Component

2.2.12.1 **Component Definition**

Although the literature on the definition of a component is very rich, there is wide agreement on some aspects of it. As cited in [7], a component is a unit satisfying predefined needs through its interfaces. Interface definitions are mostly related to the functional expressiveness of a component.

As defined in [22], the interface of a component can be defined as the description of its access point. Interfaces of a component seem to be compliant with the interface definition in object oriented world such as with Java interfaces and with abstract classes in C++ (ones providing at least a single pure virtual method) in that they do not provide any implementation related aspects and allow interface extendibility by remaining backward compatible.

Usually a proper component possesses distinct interfaces for providing multiple unrelated services or at least this is preferable for the clearer description of the interfaces. Here by a service, a logical grouping of provided/required functionality is meant. According to the definition of multiple interfaces it can be suggested that the components have test interface, configuration interface, event triggering interface and so on depending on the needs.
As explained previously, the mere specification of an interface is depicting the functional properties of components but usually lack in expressing the extra-functional ones. These properties are quality attributes such as accuracy, availability, latency and security as indicated in [24]. To overcome these limitations, component specification by contracts is an alternative [25], however interface based description is sufficient for the purposes of this work.

2.2.12.2 Component Oriented Software Engineering and Architectural Mismatch

The ultimate goal of component oriented approaches is to build systems out of components that are readily available from different vendors, a task often called “component wiring”.

Although plausible, this idea is idealistic in the fashion that it is presented currently, because this approach seems to underestimate fundamental concerns such as system architecture and performance. In fact it is a very painful task to try building components that could suit the needs of any application. This situation of architectural component mismatch is clearly stated in [10]. The authors have faced several harsh difficulties such as abundant code, poor performance and need of fundamental modification in external packages due to the fact that some components deemed off-the-shelf, required serious updates to suit their needs. Furthermore it has been stated that the mismatch could nearly always be attributed to architectural roots.

This generally happens because components built to suit the needs of a large span of developers usually contain excessive and unnecessary infrastructural support or worse, they may not even contain the mandatory infrastructure for the application in question. Moreover control structure of components –event loops and main thread of control- could be a significant spot of mismatch. Also the assumptions related with the representation and traffic of data in between components is open for inconsistencies. A similar problem is emphasized in [22], that component based systems have problems of sensitivity to change and this situation usually manifests itself in the form of unexpected failure occurrences with the introduction of application level changes to constituent components of the system.

At this point, the central question to ask is that whether component definitions really need to be uniform and have a lot in common or does it make more sense to specialize their definitions according to a particular domain’s needs? Answer to this question is sought in section 2.2.12.3.

2.2.12.3 Components and Software Product Lines

Even within a defined software family domain, there are debates on the definition of a component. As indicated in [1], several consider a component as a process in the embedded systems domain. On the other hand, some assume that compositions of components are components themselves. These last two arguments are not likely to fit together due to possible different priority levels that the processes may possess and combination problems associated with it.

As a further example, a common clustering criterion for components in embedded systems – note that this could still be achieved using interfaces in a single component – is to separate data and control related entities. This kind of separation is a bit more high level than that of separate interfaces for logically separate functionality. FORM [29] adopts as an engineering principle the allocation of control functionality and data processing functionality to different components. Furthermore in [30], it has been indicated that components that take care of control or data manipulation should be placed in two separate *planes* namely that of data and control *plane*. Such separation has been required due to the fundamentally separate issues that these *planes* address. It has been stated that the primary aim of a control *plane* element is to assure correctness whereas for the data *plane*, efficiency is the most important concern.

Resolution of the conflict related to component definition is one of the early steps to be taken, if end products of SPL efforts are expected to be true reusable components. The current presence of such discrepancies between component specifications is promising for the increasing success of PL approaches. Many developers face the mentioned integration problems in their everyday work. This is because usually different product domains demand different architectural structures.

All these difficulties point out that, reusable components may make more sense when they are considered within a PL. Discrepancies just mentioned within the embedded systems domain further reveal that the size and scope of the domain is also important for accurate and valid component specification. The domain of embedded systems software therefore could be considered as being too broad.

Therefore, it can be suggested that a component definition specifically targeted towards a particular PL's needs is expected to be more resilient and reliable. The size and the completeness of the services provided by components could render them unusable in platforms restricted in terms of resources. This brings in a tradeoff between reusability and usability. That being said, if the ultimate goal in developing components is reusability without any consideration of usability, there may be inter-violations of system and/or performance requirements in-between components developed with that goal.

Mismatch between components of the same product family is less expected than that between ones with little functional intersection. Furthermore although it cannot be claimed that components in the same product family are usually constructed with the same architectural approach, it is believed that they should be. This can only be achieved by imposing several restrictions related to the component internals and interactions via the reference architecture of a PL. The purpose of such architecture is made clearer in section 2.2.13, but briefly it can be considered as a first step architecture to initiate product construction and a set of rules and practices that facilitate the integration efforts for components that make up the systems in the PL.

Although not much attention has been paid to component construction in PL's as part of this work, the component understanding previously mentioned reflects an important part of the general understanding of a PL approach. This is because variability modeling techniques described in this thesis are expected to be applied for constructing components that are developed with the mentioned philosophy in mind.

2.2.13 Reference Architecture

The term reference architecture has several different interpretations depending on the context. In [60], reference architectures are considered in different contexts such as that of customer, technical architecture and business architecture.

As an example of the non-technical view of reference architectures, the description in [4] suggests that a reference architecture is a consistent set of best practices to be used by all the

teams in a organization. In this respect, creation of reference architecture is considered as an organizational issue; in the sense that the structure, content and management of reference architecture should be based on the organization's own needs. Actually this view of reference architecture is representing the general idea of reference architecture, without paying particular attention to PL's.

The definition in terms of PL's is to be considered mainly under technical reference architecture. In the technical side, reference architecture is related to a high level architecture. However, as indicated in [5], reference architecture should not be confused with architectural style. Architectural styles are well known recurring patterns related with high level conceptual reuse, which, when applied lead to known results. The well known examples are layered architecture, pipe and filter, client-server and so on. So unlike an architectural style which consists of some rules of thumb which when followed, allows arriving to the system architecture, the reference architecture is already intrinsically present in the system architecture, so it is an architecture but a generic one. Notable technical reference architecture examples are ADAGE of IBM [62] from avionics domain and KOALA project from Phillips [61] from consumer electronics.

With all types of reference architecture, assets from prior efforts in real projects are considered as beneficial when shaping the reference architecture. Furthermore, reference architecture is practically always considered as a living thing and to keep it up to date is essential for its success.

2.2.14 Framework

A framework is a structure that imposes several rules and restrictions upon its constituents [1]. These are meant to be present for the betterment of the systems built using those frameworks. As an example, the constraint that serial channel communication items are byte streams enhances the reusability of the protocol among parties that want to use it to exchange data. This definition is compliant with the definition of reference architecture –mostly technical reference architecture-which itself imposes several rules of composition, structure and communication.

In the scope of PL's, frameworks can be considered as the extended and enhanced versions of reference architectures, providing tools and facilities to construct systems as scoped by the reference architecture. Ultimately, PL frameworks should supply the tools and mechanisms to their users to build systems within the PL scope. In the meantime such frameworks should assure that during the composition and creation of these new systems, fidelity to the reference architecture is kept. However in addition to being based on the reference architecture, a framework needs to provide full support for expressing variability of the domain it represents.

Up until now several very successful frameworks have been developed, but it is disputable whether they are fitting the framework definition that has just been provided for SPLE. As an example Rhapsody[1] is an integrated development environment (IDE) that supports Model Driven Development (MDD) through the use of UML and has an extensive object oriented framework which helps to build platform independent systems, which later can be specialized into platform specific ones via the tool's built in support.

In order to benefit from such a framework, its domain scope should be carefully selected. Since frameworks are assumed to be useful composition environments for new systems, it is believed that the ultimate and most productive outcome of domain engineering methods is domain specific frameworks. A similar view has been stated in [26], that a framework is an incomplete template for a system within a specific domain.

In section 2.2.12 which is related to components, the difficulty of wiring components was mentioned and the primary reason for such a mismatch was indicated to be architectural mismatch. It has been indicated that restriction of the components to a certain domain could help in overcoming such difficulty. This is further eased if the components' presences are managed via the framework that they are embodied in.

As a matter of fact, several framework structures have been suggested to handle such a problem. One such infrastructure to overcome architectural mismatches is to opt for a framework that is

[1] Rhapsody is a registered trademark of IBM Rational.

comparable to a circuit board with empty slots into which components could be inserted to create a working product instance as described in [6]. This view of component oriented frameworks is presented in Figure 2-8. In a circuit board, among other facilities that are provided, the rack has fundamental infrastructural support to supply power to boards and provide the inter-board communication infrastructure.

Consistent with this analogy; the components have no existence without the depicted variant free framework skeleton. It is this variant free skeleton that assures proper communication among components and it is what gives the vitality to components. This architectural structure for the framework is initially suggested in a work from Philips [32].

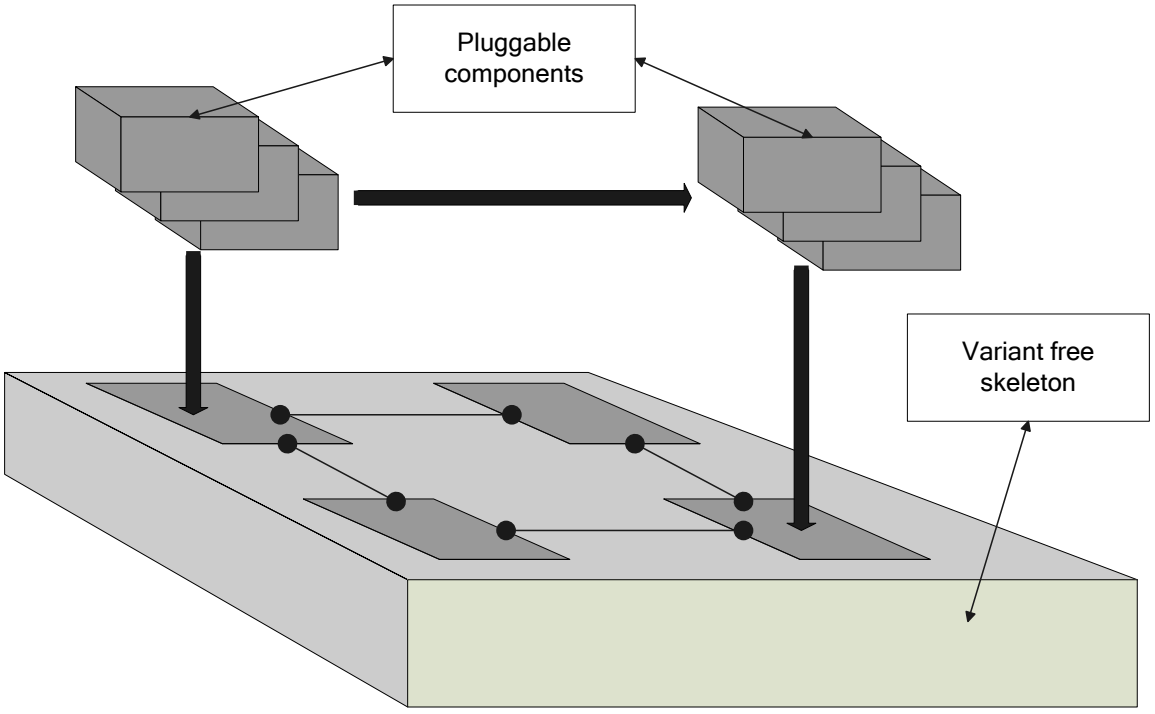

Figure 2-8 Component Frameworks with plug-ins (Adapted from [6] p.218)

In Figure 2-8, it can be seen that there is no independent deployment of ground skeleton components as well as pluggable components. Furthermore the reuse strategy in such frameworks is composition based and composition requirements are specified in detail via interfaces. Obviously composition has the advantage of run-time interchangeability over inheritance in OO reuse. Pluggable components introduce variability into the overall application and we suggest that all of the –common and variable– components are compliant in terms of architectural criterion.

2.2.15 Aspects and Aspect Oriented Programming

In the context of aspect oriented programming (AOP), an aspect is a cross-cutting concern that is spread through many modules in a software system and that cannot be properly separated away from those modules. The effort to integrate aspects –such as error handing, logging, performance criteria- to a code usually results in what is called a *tangled code* [15].

The idea of aspects grew out of the limitations of programming paradigms that were used prior to the paradigm of AOP. The main logic behind procedural and object oriented programming (PP, OOP) paradigms is to divide the concerns making up a software system so that they are well encapsulated; as subroutines in the former approach and as classes in the latter. But some obvious concerns like the ones mentioned before could not be encapsulated properly inside modules. This lack of expression power gave birth to the concept of aspects.

In [15] aspects are defined as properties of a system that cannot be cleanly localized in a general procedure as with OOP and PP, that are not parts of the functional decomposition of a system. The aspect definition has an important relation with software reuse. Other than several inefficient and awkward ways, the usual means by which cross-cutting concerns are incorporated into systems built without direct aspect support is by introducing that proper cross-cutting issue into each and every encapsulated module that requires it. Then a possible change about the issue for a brand new system will require a series of changes in every encapsulated element. This is a poor practice in terms of reuse; actually very few items, if any could be reused in the original design at all.

Hence, proper identification and incorporation of aspects into systems designed without considering aspects will have a positive effect on reuse. The controversy is that aspects require several language support mechanisms in order to be used, which is not always possible.

Identifying these cross cutting concerns and ascertaining them as dispersed information is a fundamental way of reasoning about any kind of modeling scheme. The initial intent was aimed at easing some odd parts related to procedural and object oriented modeling. This intent has been another source of inspiration for a different modeling approach for PL's in this work.

When regarded from a PL perspective, several important aspects are present in the artifacts that constitute the PL itself. Of central importance and consideration is variability. Variability can be considered as an aspect that is dispersed through PL artifacts, which when not managed effectively, results in "tangled" PL models and assets.

The aim in using *concerns* is to reduce this tangling of variability information, representing it neatly for the user of the model. Moreover, usage of *concerns* aims to bring significant ease and efficiency in resolving the variability to build real systems by considering variability as an artifact that is present in all of the PL assets.

A distinction must be made between the aspect language in which the aspects are specified and a component language in which the hierarchically –or functionally– decomposed units are specified. Keeping this distinction in mind, several important terms need to be defined in order to proceed with the discussion of AOP.

Join Point: A join point is either a built in construct or is an element specified in any possible manner of the component language as defined in [15]. It is the exact point where the cross-cutting concern cuts the component language. In other words it is the point where the aspect should be present.

Point Cut: According to [16], point cuts are a means to refer to the join points. They can be considered as a collection of join points since they specify that collection.

Advice: Advices are method-like constructs that are used to define the additional behavior at join points.

In terms of the AOP jargon, the triplet ensemble of join point, pointcut and advice form an aspect. The information model of an aspect as it is originally described can be seen pictured in Figure 2-9.

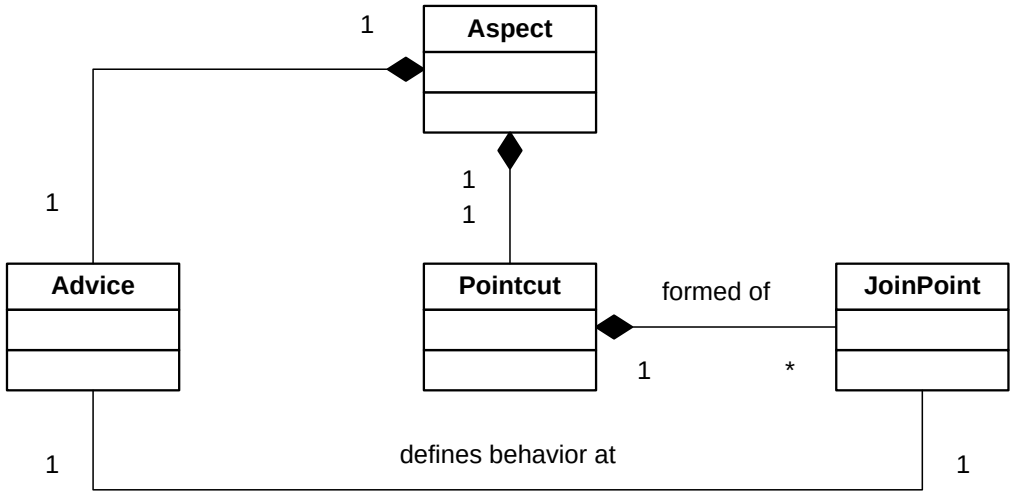

Figure 2-9 Aspect metamodel

Although original ideas were presented back in 1997, the consequent research on aspect oriented programming mainly focuses on how to integrate the aspects into an existing language in many ways. The literature is very populated with such work (Java, C, C++ …).

However, from the AOP point of view, a joint point model can be conceptually interpreted in a different way than it is interpreted for general purpose programming languages.

With all the debate going on about AOP and its applicability, the foundations it provides could be used for novel models resembling join point models. This is achievable when we think in line with the consideration of variability as a different aspect. The question is: What if we have a central aspect called variability, our join points are carefully and hierarchically distributed throughout all –or some- of our PL artifacts and finally our pointcuts are specified with the aid of a higher level, DSL? The essential argument is that variability traceability is expected to be easier to achieve with such an approach.

2.2.16 Real Time Embedded Software

Experimental work performed within the scope of this study is based on real time scheduling algorithms. Real time scheduling algorithms are primarily used in real time operating systems (RTOS). A vast amount of real time embedded systems software run on RTOS's. Scheduling of processes in an RTOS is a fundamentally critical and is one of the core elements for RTOS's making them clearly different than a regular operating system (OS). This difference of scheduling mechanisms has its roots in the difference between embedded application internals and other applications. Understanding embedded systems' properties is helpful for grasping the objectives of real time scheduling algorithms (RTSA).

In very general terms, an embedded system is a computer system designed for a specific dedicated purpose. As specified in [2], to regard embedded software just as software on small computers is inappropriate, since embedded systems exhibit more fundamental differences compared to desktop computing.

The essential difference is that embedded systems need an interaction with the physical world and this interaction requires a fairly sound understanding of the physical world itself. This gives rise to the fact that embedded software needs to be more dependable and it should be able to sustain its integrity regardless of the possible unexpected external behavior. It is expected that embedded systems operate for very long periods of time, with -usually- clear timing requirements and responsiveness.

Timeliness is a particularly important concern in real time systems. A real time system is one which, as the name implies needs to fulfill some real time constraints, such as operational deadlines. Real time systems are classified in two major groups, namely hard and soft real time systems. As stated in [3], in the case of hard real time systems, a missed deadline constitutes an erroneous computation and system failure. Thus late data is useless and wrong. On the contrary, soft real time systems are characterized by time constraints which can be missed depending on the particular circumstance or by small deviations or completely in some cases. Obviously, real time scheduling mechanisms that such real time software is operating against are critically important for the sake of such systems.

In addition to timeliness, different classes of embedded software have different performance, reliability, availability, quality of service (QoS) etc. requirements.

CHAPTER 3

USING CONCERNS FOR TRACING VARIABILITY IN SOFTWARE PRODUCT LINES

3.1 Introduction

This chapter introduces the concept of *concerns*. The reasons for the necessity of *concerns* are outlined. Since the concept of *concerns* is derived from decisions, the relation of *concerns* with decisions and their differences are discussed.

3.2 *Concerns*

The concept of *concerns* that is introduced in the scope of this thesis is closely related to the concept of decisions. *Concerns* intend to be a powerful way of tracing variability across all assets that make up a product line (PL). A *concern* is an ensemble of variants organized in vertical and horizontal hierarchies in order to represent the variability of a PL. A *concern* model of a PL is a model which represents the PL via the usage of *concerns*. It is a snapshot of variability for the artifacts of a PL.

During the course of this study, it has been observed that some features' selection criteria are deeply affected by some underlying technical and non-technical issues which are not directly obvious from the mere feature specification. The roots of such constraints lie deeper in the abstraction hierarchy that makes up the systems in the PL. Decisions try to track this variability to some extent.

Concerns address several open points in decisions. As mentioned in section 2.2.4, decisions do not provide a separate mechanism for addressing the variability information spread inside a particular asset abstraction layer. When using *concerns*, an explicit distinction must be made between variability that is present in each asset abstraction layer and the variability spread within a single layer, as vertical and horizontal variability respectively to overcome this limitation. Vertical variability idea is present in decisions, but as an example of horizontal variability, one can consider the variability existing within possible choices of a communication protocol - typically this would be in the software design and/or source code level- for a particular variability item that is resolved at a higher abstraction level.

Concerns are introduced as covering variability in four major vertical artifact spaces. In other words variation points are considered at four different levels of abstraction. These are product features, requirements, software design and finally source code. It is possible to make a removal from or add an extension to these layers as needed (A candidate could be test artifacts that have been deliberately left outside of the discussion). In addition, any necessary amount of horizontal variation may be represented in any particular vertical variation level. To better visualize the situation, consider the Algorithm *concern* from real time scheduling algorithm (RTSA) domain in Figure 3-1. The *concern* is located to the left side of the figure enclosed in a box named Algorithm. Actual variability items in the PL artifacts are on the right side of the picture.

Note that inside a *concern*, actual variability items are referred to by their symbolic names. The numbering scheme is up to the modeler; the only thing that is imposed is that all variation points need to be represented by a different substitute name. However if the suggested convention is followed, it is expected to ease modeling task For convenient reference, variation points at different layers of a *concern* are referred to by a numbering scheme which is prefixed with the uppercase version of the initial letter of the name of the related layer. These initials are P, R, D, S standing for product features, requirements, software design and source code respectively. Therefore if one wishes to refer to a source code variation point it could be of the form “S 1”. Note that it is not mandatory to have one level numbering, one could have chosen to enumerate the variations to end up with two or more levels of numbering such as “S 1.2”.

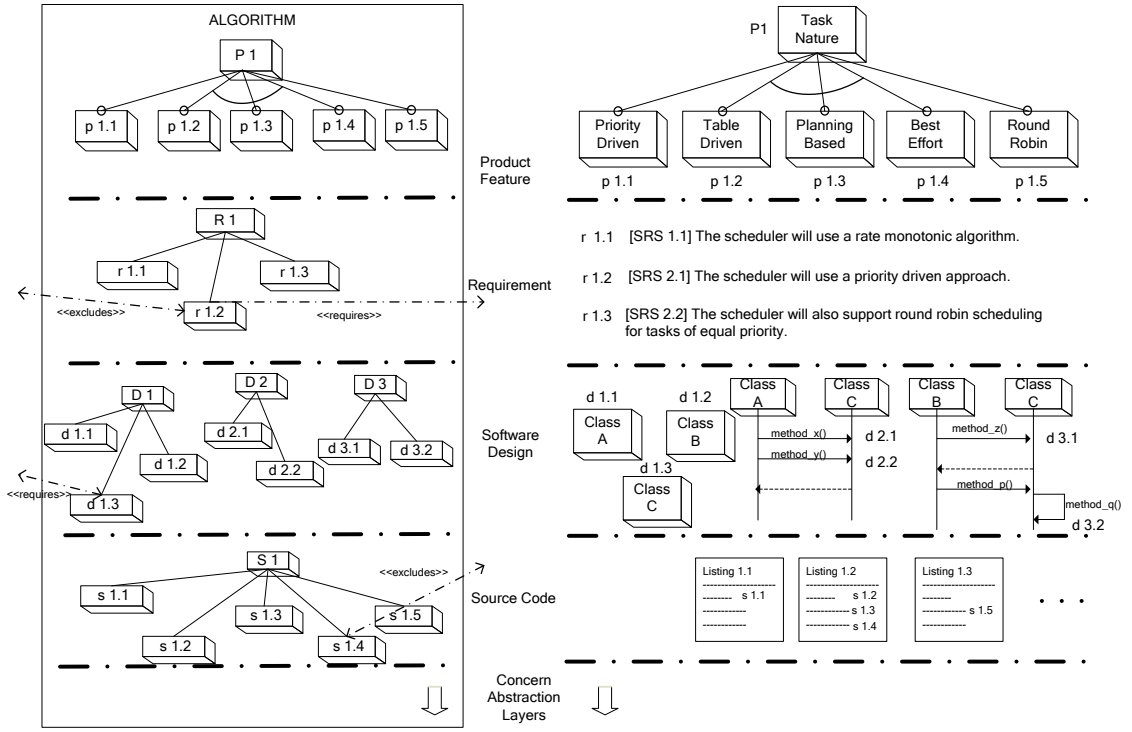

Figure 3-1 Example *Concern* "Algorithm" and corresponding actual variability

When it comes to variants related to these variation points, their representation is initiated by the lowercase letter of the related *concern* abstraction layer. Numbering starts one level indented with respect to the related variation point numbering. For instance, if a variation point named "S 1" has 3 possible variants they will be numbered as "s 1.1", "s 1.2", "s 1.3".

The purpose of this uppercase and lowercase differentiation will make more sense in the formal treatment of *concerns* in section 3.3. Furthermore the naming convention forbids a number as a prefix to any uppercase letter. This reason also becomes clear in section 3.3.

In order to establish traceability between actual variability and the *concern* representation, one should observe traceability symbols in the actual artifacts. In this proper case, actual artifacts have variants marked with the related symbol notation on all artifacts. Feature diagrams, as an

exception, have also the variation point traceability symbols since variation points are inherent in feature diagrams. For other artifacts, it is obvious to derive variation points from variants.

The relationships between variability items within a *concern* are represented by the same mechanism used for features of Feature Oriented Reuse Method (FORM) as described in sections 2.2.9 and 4.4.2.2. Note the "requires" and "excludes" relationships that diverge out of the *concern*. Their presence knowledge comes from the interrelationships of the artifacts in the domain and at the other end of this relationship a variant -which is possibly from a different abstraction layer- in a different *concern* should be expected. Note that in this way, variation points and/or variants from different abstraction layers are related with each other. By indicating these relationships, *concerns* aim to find a means to lift such information buried deep down into non-user visible parts of a system to a manageable level, discover its possible effects on features –such as inclusion and exclusion of several more features– that were not spotted at first sight with mere feature analysis. This ultimately makes it possible for the end user to compose more reasonable systems.

Requires and excludes relationships are shown for relations between different *concerns*. Following question needs further clarification: Isn't it possible to have inclusion and mutual exclusion relationships within a single *concern*? This is very likely to happen. As an example consider R 1.2 and R 1.3 in Figure 3-1. These requirements come from the same requirements specification document namely from that for System I in APPENDIX C. These two requirements come hand in hand from the requirements specification so it is obvious that they mutually require each other. This could have been depicted as a "requires" relationship between the two variants which would crowd the picture. Also the handling of these kinds of relationships brings additional complexity. Therefore even if such relations exist within a single *concern*, the tendency is not to show them on the pictorial *concern* model.

3.3 *Concerns* to Context Free Languages

The essential question to answer concerning variability modeling is "how to represent conveniently all the rules that govern the relations between variation points and variants?" A suitable candidate to define these rules appears to be context free grammars (CFG). The idea of

representing feature diagrams via grammars has been suggested in [71]; however, instead of using CFG's, special production rules and notation have been preferred. Also [72] introduces an iterative grammar using structures as one or more (+) or zero or more (*) and special notations to denote optional features. Furthermore it has been pointed out in [72] context free languages (CFL) are generally inadequate for the complete modeling of variability. This criticism will be treated later on.

A CFG is more formally defined in [67] but in short, where a capital letter denotes a non-terminal (say A) and where lower case denotes a terminal and/or nonterminal string (say α), if all the production rules of a grammar are of the form $A \rightarrow \alpha$, then this defines a CFG.

Remembering the notation presented in section 3.2, variation points correspond to non-terminals and variants correspond to terminals of the CFL defined by the CFG of transformation rules. Thus any string of non-terminals is obliged to comply with the rules of the related CFG in order to represent a valid system in the PL.

The question is why it is important to represent variation relation in a formal way. The reason is that it is straightforward to obtain a parser from a CFG that can validate whether a given string of terminals represent a valid item –a valid string within the CFL– in the PL.

However, the production rules are not very straightforward at first glance. How to denote optional, mandatory and alternative variants as production rules? What happens for relations in between different layers of *concern* abstraction? How to represent exclusion and inclusion? All these questions need to be answered to clarify production rule representations for *concerns*. In order to consider possible variability relationships, consider Figure 3-2.

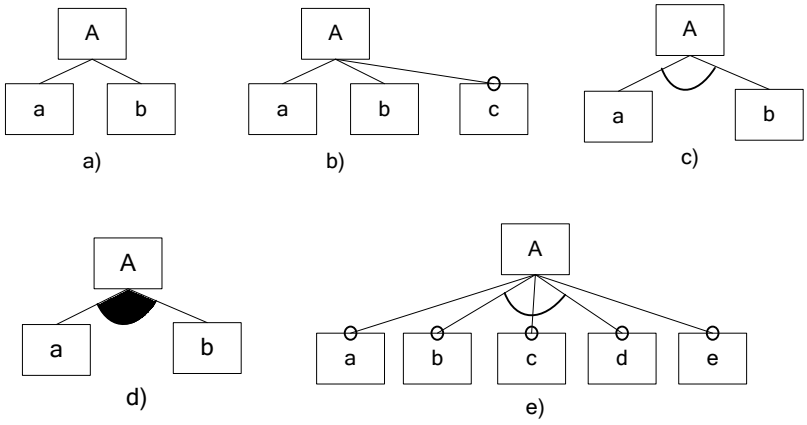

Figure 3-2 Possible Variation point and Variant Relations

The situation in a) denotes no variability at all. It is plain commonality and the rules are as in Listing 3-1. Although not helpful for variability modeling purposes, this is mentioned for completeness.

$$A \rightarrow a\ b$$

Listing 3-1 Commonality

Listing 3-2 corresponds to b) in Figure 3-2 and introduces an optional variant to the relations in a. Note that Λ is the null string. If the transformation $A2 \rightarrow c \mid \Lambda$ is chosen in the favor of Λ then the optional variant c is not included in the string that describes a potential system in the language. Also note that intermediate non-terminals A1 and A2 are introduced –this is a single indentation level as there is a single number after the original non-terminal– which does not appear on Figure 3-2 just to clearly represent transformation rules. This technique is commonly used in a CFG.

$$A \rightarrow a\ A1$$
$$A1 \rightarrow b\ A2$$
$$A2 \rightarrow c \mid \Lambda$$

Listing 3-2 Optional Variability

In Listing 3-3, the case c) of Figure 3-2 is expressed. This is the case of alternative variants. The item d) in Figure 3-2 represents a "logical A or B" relationship is which is expressed as in Listing 3-4.

$$A \rightarrow a \mid b$$

Listing 3-3 Alternative Variability

$$A \rightarrow a \mid b \mid a\ b$$

Listing 3-4 Logical "Or" Relationship

A slightly complex example is depicted in e) of Figure 3-2. The fundamental pieces of this variation have been described previously and it leads to the CFG snippet in Listing 3-5.

$$A \rightarrow A1\ A2 \mid \Lambda$$
$$A1 \rightarrow b \mid c \mid d \mid \Lambda$$
$$A2 \rightarrow a \mid e \mid a\ e \mid \Lambda$$

Listing 3-5 Mixed Variation with Alternative and Optional Variants

Since *concerns* are organized in several abstraction layers, it is required to convey a particular string of terminals to the next abstraction layer for complete specification of systems. As an example, consider the hypothetical partial *concern* in Figure 3-3.

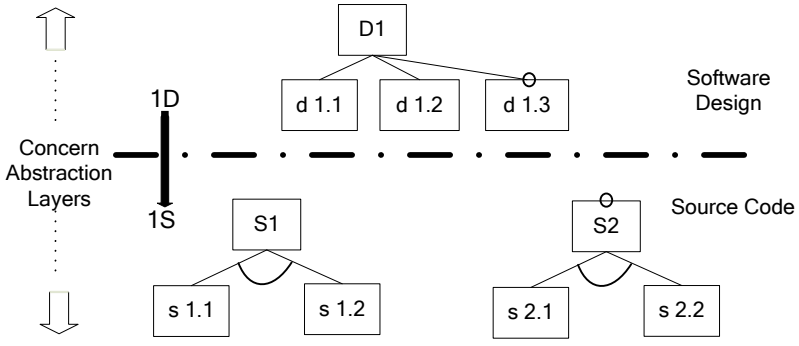

Figure 3-3 Partial *concern* abstraction layers

The forbidden notation of inserting a number prefix in front of an upper case non-terminal in section 3.2 makes use in defining the transition. The Listing 3-6 describes this transition along with production rules for the partial *concern* in Figure 3-3. Note that this sequence of rules defines the most general possible transitions from one layer to the other (The transition is from D1 to S1 or S2 or both or none of them). The arrow from 1D to 1S denotes the general nature of this transition. In a particular case this transition could be specialized.

$$D1 \rightarrow d1.1\ d1.2\ D1.1\ 1D$$
$$D1.1 \rightarrow d1.3 \mid \Lambda$$
$$1D \rightarrow 1S \mid \Lambda$$
$$1S \rightarrow S1 \mid S2 \mid S1\ S2 \mid \Lambda$$
$$S1 \rightarrow s1.1 \mid s\ 1.2$$
$$S2 \rightarrow s2.1 \mid s2.2 \mid \Lambda$$

Listing 3-6 Grammar for *Concern* Layer Transition

Note that when the non terminal D1.1 is used in the production rule (D1.1 → d1.3 | Λ), the convention stated in section 3.2, that the terminals remain a single numbering level indented below the non-terminal they were produced from was not followed. This is not important as this non-terminal was merely used to ease the expression of an intermediate production step. It should also be stated why there is the production rule 1D → Λ. If this production rule is applied, this results in a partial system specification where variability is not resolved within all abstraction layers that the *concern* represents –in this particular example, unresolved variability would be in the source code. This special production rule can be particularly important in the case where variability is not modeled in all *concern* abstraction layers due to resource constraints. This feature is referred to as partial applicability of a *concern* model.

Furthermore, it is likely to expect multiple occurrences of a single variation point throughout the *concern* model. This is not a problem; production rules follow the same rules. As an example consider the hypothetical partial *concern* model in Figure 3-4.

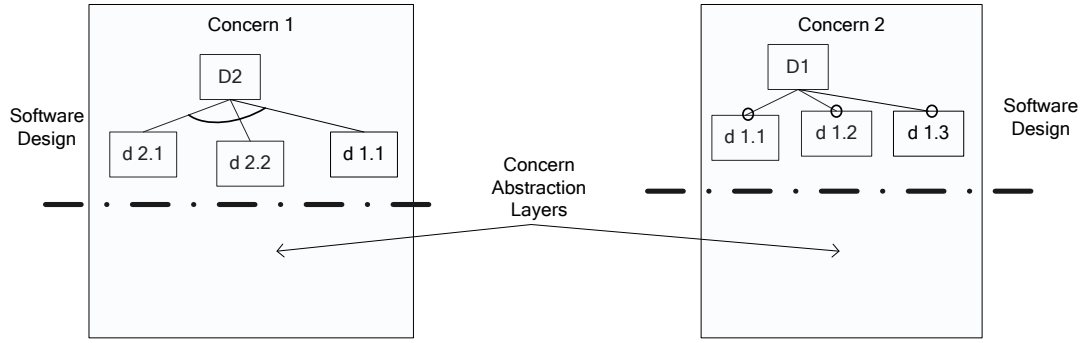

Figure 3-4 Multiple Occurrence of a Variation Point in the *Concern* Model

The grammar defining this partial *concern* model is listed in Listing 3-7.

$$D1 \rightarrow d1.1 \mid d\ 1.2 \mid d1.3 \mid d1.1\ d1.2 \mid d1.1\ d1.3 \mid d1.2\ d1.3 \mid d1.1\ d1.2\ d1.3 \mid \Lambda$$
$$D2 \rightarrow d2.1 \mid d2.2 \mid d1.1$$

Listing 3-7 Grammar for Multiple Occurrence of a Variant

However this situation may cause the following dilemma. Suppose the person modeling the system decides by barely considering D1, that d1.1 should be included in the system. However, if the very same person also requires that either d2.1 or d2.2 be included in the system, then since d1.1 is an alternative to these; such a selection would not be possible. The modeler should take care when considering variation points with multiple occurrences or, the parser representing the grammar will and should warrant such an illegal selection.

The last remaining relation is that of inclusion and exclusion. Treatment of exclusion and inclusion is however tricky and therefore not managed via a context free grammar. The first reason for this is that exclusion relationships that exist between variability items suggest that a particular production rule which is able to govern the exclusion relationship as if it were a valid production, needs to be excluded from the grammar. Consider the “excludes” relation between d 1.1 and s 1.2 represented in Figure 3-5.

.

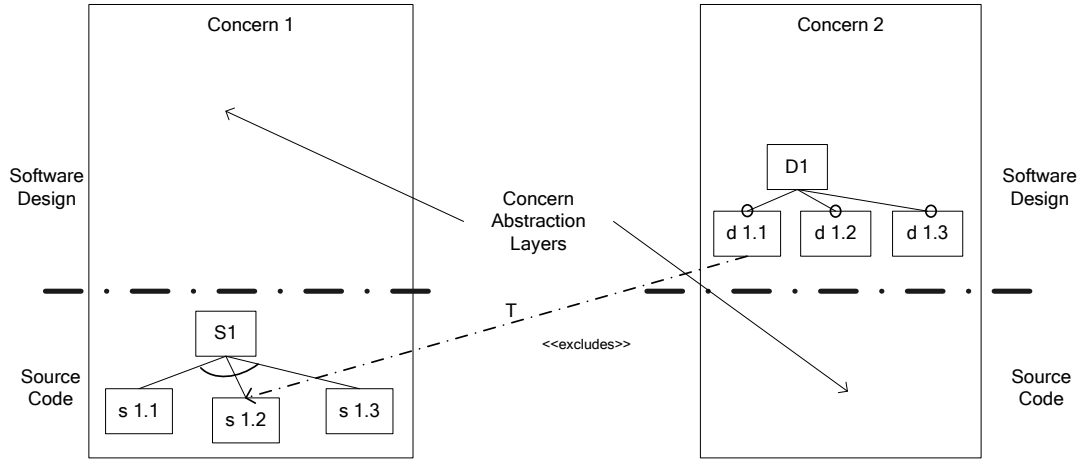

Figure 3-5 *Concern* Interrelationships

A non-terminal "T" has been introduced in order to represent the production rule which can be seen in Listing 3-8. The last line of the listing indicates a transition via the usage of the non-terminal T. This transition includes the string composed of the variant d.1.1 in the language generated by the respective grammar.

$$D1 \rightarrow d1.1\ T \mid d1.3\ T \mid d1.1\ d1.2\ T \mid d1.1\ d1.3\ T \mid d1.2\ d1.3\ T \mid d1.1\ d1.2\ d1.3\ T \mid \Lambda$$
$$T \rightarrow s1.2$$

Listing 3-8 Production Rules Defining Complement of Excludes Relationship

However, the actual grammar that is desired needs to exclude this particular rule. In order to demonstrate this with a CFG, any other valid string production rule needs to be explicitly stated. This is because in a CFG the productions that are not allowed are simply not included in the production rules. The valid production rules for this fairly simple example with partial *concerns* are as in Listing 3-9.

D1 → d1.1 T1 | d 1.2 T2 | d1.3 T2 | d1.1 d1.2 T1 | d1.1 d1.3 T1 | d1.2 d1.3 T2 |
d1.1 d1.2 d1.3 T1 | Λ

T1 → s1.1 | s1.3 | Λ

T2 → s1.1 | s1.2 | s1.3 | Λ

Listing 3-9 Production Rules Defining Excludes Relationship

Note that T1 and T2 are used to relate d1.1, d1.2, d1.3 to the variants in Concern 1 in Figure 3-5. Notice how complicated the rules can become and anticipate the volume of all the rules that need to be included to represent all possible production rules just in order to denote a single exclusion rule for a larger model. This effort makes representation of exclusion relationships with a CFG undesirable.

There is another inconvenience that also applies to exclusion relationships. There appears to be a clear problem of keeping the coherence of the relationships in the model. To picture the situation, consider Figure 3-5, Listing 3-8 and Listing 3-9 again. Imagine that the excludes relationship is modified somehow so that d 1.1 now excludes another terminal. This means the diagrammatic representation of the excludes relation in Figure 3-5 and all the production rules Listing 3-8 Listing 3-9 need to be changed. This is a serious overhead.

The third inconvenience is related with both requires and excludes relationships. It has been seen that in order to express inclusion and exclusion relationships, as it can be seen in Listing 3-8 through Listing 3-9 additional non-terminals need to be introduced which brings significant overhead and complicates the overall grammar and consequently the language.

As indicated before, [72] advocated that a CFG grammar has problems in adequately modeling variability and to justify this claim proposes several relations that are difficult to express via a CFG such as “A includes B or C or D”. This difficulty is recognized in the present study as well, *concern* based modeling and such complex relations are treated together with the just mentioned exclusion and inclusion relations apart from the CFG based modeling. In theory, a Turing machine can handle these relationships. In application, this can be represented by a postprocessor program that checks the strings that are specified using *concerns.* A string that defines a potential system is first passed through the parser generated using the rules of the CFG

and then passed to the post processor checking the simple/complex requires excludes relationships. The motivation here is that individual pieces of this two step approach are both strong and simple in what they do. The infrastructural application model for using *concerns* in system specification can be seen in Figure 3-6.

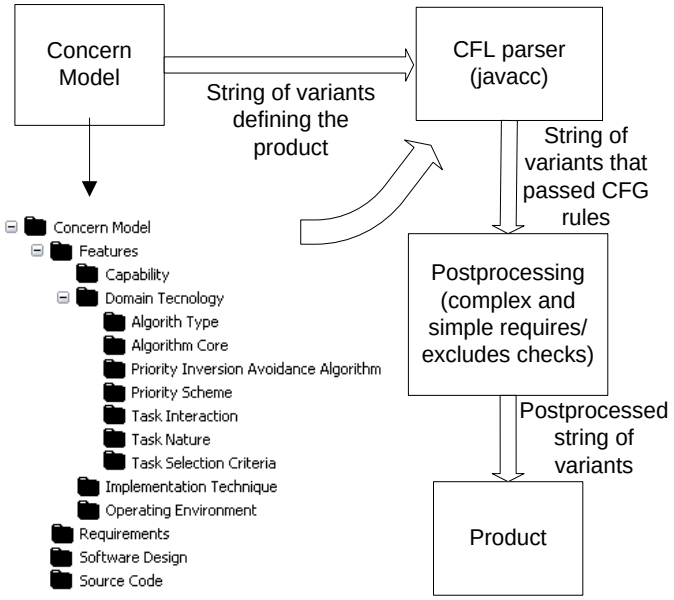

Figure 3-6 Using *Concerns* for System Specification

The tree-like representation of the *concern* model can be considered as a domain specific language (DSL) that is driven by the variability information of the PL. The preferred parser generator for this work is *javacc* [73]. If a string of variants can pass without any violation through the CFL parser and the postprocessor, this means that it defines a proper system in the PL.

3.4 **Benefits of *Concerns***

One of the primary topics that *concerns* address is that of realizable system composition using available assets. In this respect, the bare usage of features may be insufficient because of inadequate variability modeling and tracing. As a general difficulty of PL's, it has been indicated in [29] that selection of meaningful feature combinations leading to reasonable products for all combinations is particularly difficult. This is because although only features are used for system specification, feature modeling alone is insufficient for revealing correct feature relationships accurately. Feature relationships have dynamics that have foundations buried in several artifacts of a PL which could be elaborately modeled using *concerns*.

A second important point that is addressed by *concerns* –as a direct consequence of the first point that has been just mentioned– is the feature interaction problem. Feature interaction problem, as the name suggests, occurs when the mutual presence of more than one feature of a product is somehow impeding the overall system operation. This topic is widely covered in telecommunication systems to a point where specific workshops have been held –first called the feature interaction workshop (FIW) and later the International Conference on Feature Interactions (ICFI) - [68]. The extent to which this problem can have an impact varies based on the criticality of the system. Feature interaction problem is expected to increase with the increasing variability since the chances of possible feature interactions are increased. *Concerns*, by managing variability more accurately and relating various product artifacts aim to reduce this feature clash.

The real outcomes of the benefits that are explored in this section are sought later in Chapter 5. For this purpose two evaluation criteria are defined and the advantages of *concerns* are assessed.

Although not assessed by any evaluation criterion in Chapter 5, it is helpful to mention a supplementary benefit of *concerns*. Since *concerns* are expected to establish traceability of variability among all SPL assets, they remain in contact with every aspect of the PL at any abstraction level. Since when using *concerns*, any particular application is created by the resolution of the variability at several asset abstraction layers in the *concern* model, any feedback from the user side to the actual application is traceable to all related assets in all abstraction levels in the domain model if the appropriate links exist. It should be noted that

without *concerns*, a feedback from the user side would remain as a feedback targeted solely to an asset at a single particular level. Although that could be helpful, such a feedback would usually affect only the asset collection at the particular spot that has been aimed and its impact to other abstraction layers would be concealed.

CHAPTER 4

DOMAIN ENGINEERING OF REAL TIME SCHEDULING ALGORITHMS

4.1 Introduction

In the first part of this chapter the example domain of real time scheduling algorithms (RTSA) in operating systems (OS) will be briefly introduced along with the reasons for the selection of this domain.

In the second part of this chapter the reason why feature oriented reuse method (FORM) has been chosen as the domain engineering approach for modeling efforts is discussed. Then the example domain of RTSA will be modeled using a version of FORM that is explained together with the modeling.

Following that, *concerns* are added to the modeling efforts of FORM to enhance variability tracking capabilities, as presented in Chapter 3, and the same domain of RTSA is modeled with this modified version of FORM.

4.2 Real Time Scheduling Algorithms

The most important thing before moving on to software product line engineering's (SPLE) domain and application engineering phases, is to consider what kind of a product could be a

financial success in that particular technology domain. This need is obvious from the definition of PL's as being a market centered concept. In Feature Oriented Product Line Engineering (FOPLE) this part is referred to as the Marketing and Product Plan (MPP).
Classification of horizontal (encapsulated and diffused) and vertical domains were explained in section 2.2.5. Furthermore, as indicated in [17], analysis of vertical domains is targeted towards constructing domain specific frameworks and that of horizontal domains towards reusable components. For a thesis effort aiming to demonstrate feasibility of a novel approach, analyzing a vertical domain is a colossal effort. Moreover since this work does not consider MPP critical for the points that are wished to be made, the potentials of salability of the products of RTSA domain are not explored.

It is indicated in [27] that a small domain with a high level of commonality is appropriate for an initial effort in establishing a PL. Moreover it has been stated in Feature Oriented Domain Analysis (FODA) [14] that the target domain need not be at a particular abstraction level, it can well be part of another broader domain.

The primary goal of this work is to demonstrate usefulness of *concerns* in capturing and tracing variability and the ease that they bring in system specification in a PL. For the just mentioned constraints of size and resource and for the relation of the domain to the author's personal expertise, the domain of RTSA is an ideal fit.

4.3 FORM as a Domain engineering approach

As it is mentioned on several occasions, FORM has been selected as the approach to adopt and modify for the experimental work. Also as indicated previously, FORM is based on FODA and although it is placed among the domain engineering approaches, it has been revised in [29] to become a full SPLE methodology named FOPLE.

This work is not paying extensive attention to the MPP phase of FOPLE as this phase is non-crucial for proving the actual point of the contribution that is aimed to be made. Therefore FORM has been mentioned as the principal approach that has been used rather than FOPLE.

Then why is FORM chosen as the experimental approach? The reason will be more apparent when the example domain is investigated, but in short it is the appropriate approach to model the chosen domain of RTSA.

It has been indicated in [21] that, different products in different scopes need different domain engineering approaches. For instance, a software product line (SPL) approach supported with use case modeling is introduced in [33]. Such an approach is more helpful in the context of a systems involving abundant user interaction. Use case modeling is least helpful when user involvement is minimal, such as in low level embedded systems software.

Is FORM a one to one match for operating on such low level software domain? Perhaps this is not the case either, however; as it will be seen, during the modeling of the RTSA domain, FORM is both trimmed and extended to fit the needs of this domain wherever necessary. But one fundamental reason for selecting FORM as the PL approach to model RTSA domain is its maturity and completeness. For instance, compared to FODA, FORM focuses more completely on domain modeling –constructing a reference architecture– and on domain design –construction of reusable components.

4.4 **Modeling the RTSA Domain with FORM**

FORM, in its complete specification has far too many items to consider when the RTSA domain is considered. Such items are just not necessary for the purpose of this particular domain. Furthermore, FORM does not dictate a particular way to specify the reusable components that are constructed in the domain engineering efforts, rather it suggests several engineering principles and design guidelines to help achieving this task. Due to this fact, a discussion related with the tailoring of FORM is presented before modeling RTSA domain using it. The philosophy of adapting the FORM method to tailor it for the domain of RTSA is compliant with the approach that suggests having a proper SPLE method per domain [17].

In the following discussion, the RTSA domain is modeled with a tailored version of FORM approach. Due to the experimental nature of this work, justification of models and the overall engineering effort is performed by relying on domain expertise rather than relying on the support provided via a third party expert. Furthermore, this work emphasizes domain engineering sub-process of FORM since that is the part related with the expected effects of *concerns* in the scope of this work.

4.4.1 Domain Scoping

A textual description of the domain is a first step in determining the context of the domain. Several assumptions are necessary to further narrow down the domain to be considered in this study. The true classification of the example domain is RTSA algorithms for OS designed to run on uniprocessor, non-distributed hardware platforms. This distinction is visible in the feature analysis part of this domain engineering effort in section 4.4.2.2.

There are two primary reasons for this reduction of domain scope:

1. Scheduling problems are among the most difficult computational problems. In the broadest sense, they are classified as NP hard problems. That is, non-deterministic Turing machines that solve the considered problem in polynomial time may not exist. Domain engineering over a larger class of scheduling algorithms is beyond the scope and resources of this study.
2. The primary aim of this study is to demonstrate the usefulness of *concerns* in capturing variability information spread across many artifacts of a PL and using this information to support decision making and ease system construction. Therefore any domain of reasonable size is acceptable which serves the previously mentioned purpose.

The structure of the RTSA domain is depicted in Figure 4-1. This model is obtained by considering several operating system architectures such as that of Windows NT [36], Sun Solaris [35], UNIX [37] , QNX Neutrino [38] and it shows where the scheduler resides in the domain of operating systems.

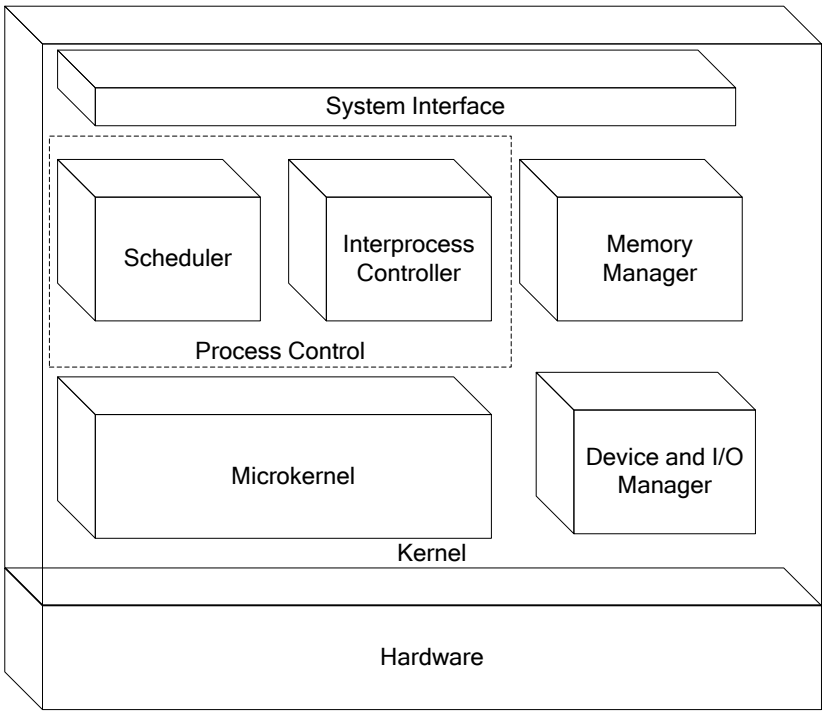

Figure 4-1 Structure diagram for RTSA domain

Here it should be noted that the scheduler block stands for the RTSA domain and other blocks for their respective domains. Such an abstraction is necessary to more accurately represent the entities in the structure diagram. This is because in the actual implementation, it is not the RTSA domain that interacts with the other entities in the structure diagram; rather it is the scheduler component itself. However scheduler component of an OS provides a little more than the scheduling of tasks. Among other things, it interacts with the memory manager and makes some hardware specific calls. Nevertheless, the principle goal of a scheduler is to determine the schedule for a set of tasks it is responsible of. Therefore in text we refer to the scheduler

component as if referring merely to the component that realizes the scheduling algorithms of real time systems, whereas this is not entirely true.

Obviously the OS's used for constructing the structure diagram in Figure 4-1 have several derivatives and versions but the general idea beyond their structure is considered to derive a context model for the RTSA domain.

A strategy could have been to place the scheduler inside a logical unit called process controller which can include several other modules responsible of interprocess communication and memory management (see the rectangle with dashed line boundaries in Figure 4-1). This would have reflected the case for some OS's and not the others. What is common to all of them is that in any case the blocks responsible from scheduling are in interaction with other major parts of the OS. Therefore the structure model encompasses fundamental OS building blocks.

The context diagram showing the interaction of these major OS parts with the scheduler can be observed in Figure 4-2. This architecture represents mostly the Linux kernel architecture and interdependencies of its subsystems, as explained in [39] as well as those of μCOS-II [40] and BeRTOS [41] (Note that Linux can be operated as both real time and non-real time whereas the other two are both pure real time kernels). The flows imply both data and control interactions; however this distinction is not made explicit for the purpose of the context diagram which is a broad picture of interactions.

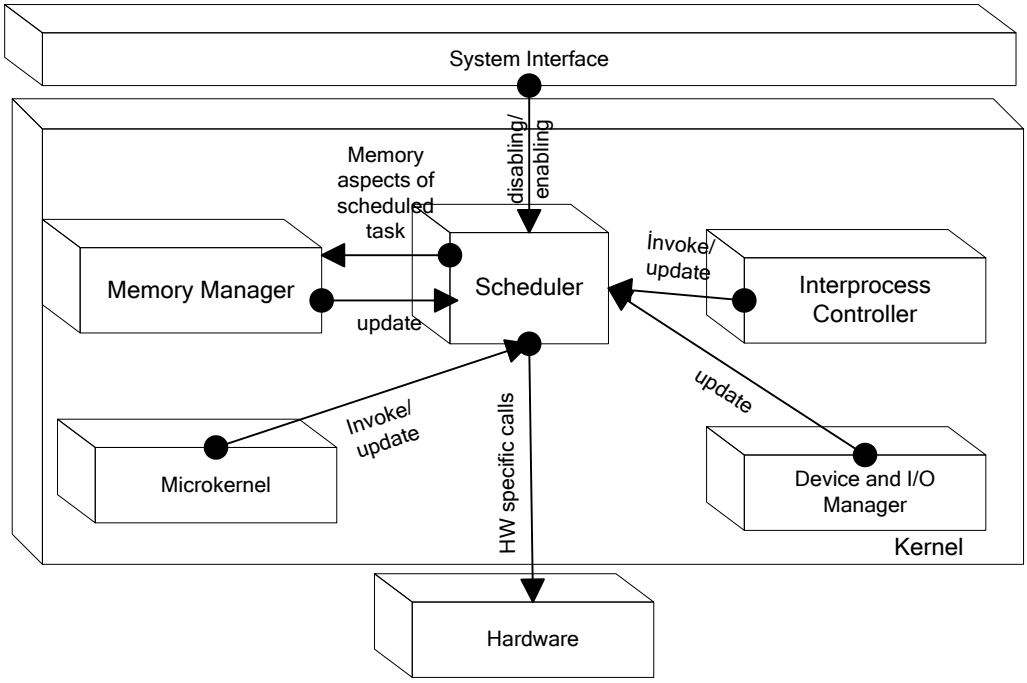

Figure 4-2 Context diagram for RTSA domain

Several points that are expressed through the aid of the context diagram can be further detailed as:

1. Schedulers are classified as being part of the OS kernel; therefore they have relatively less user interaction compared with more user related structures such as the OS shell. However there is usually still some user interaction via the system call interface that is aimed towards enabling and disabling scheduling in order to provide mutual exclusion for shared variables and any global data. This fact can be observed on the context diagram.
2. Practically all subsystems inside the kernel require scheduler functionality –i.e. basically running the scheduling algorithm and modifying/updating several data structures-. This fact is visible in the Kernel scope in Figure 4-2.

3. Hardware specific calls may be necessary in order to disable/enable interrupts, check register statuses, suspend/resume tasks and perform context switch.

4.4.2 Domain Analysis

4.4.2.1 **Entity Relationship Modeling**

Rather than any strict regulation about entity relationship modeling, a casual approach to draw the diagram is preferred.

The entity relationship diagram in Figure 4-3 shows major common parts that make up a scheduler component in a real time kernel. The construction of such a diagram is mainly based on the real time scheduler classification efforts in [42], [43], [45], [46] as well as on the review of several open source software –only μCOS-II [40] is closed source, yet its source is usable for educational purposes-. The reviewed scheduler codes are from:

1. μCOS-II [40]
2. POLIS[1] [44]
3. S.Ha.R.K[2] [54]
4. Stream[3] [58]
5. OS/161[4] [59]

---

[1] POLIS is in fact a framework for realizing hardware/software codesign. However it involves a tiny operating system with various real time scheduler supports.

[2] S.Ha.R.K is a configurable kernel that can be soft/hard/non real time. Thus it supports various schedulers. [54]

[3] Stream is the stream data manager that handles stream data management over media such as sensor networks or telecommunication systems. This system has built in scheduling facility. [58]

[4] OS/161 is an operating system built for educational purposes that includes a standalone kernel. [59]

It should be noted that this selection has been made to cover a sufficient amount of scheduler codes– both from OS's and also from other systems that require scheduling-. This particular selection represents a set that has been readily available and more importantly consisting of terse scheduler implementations. Figure 4-3 presents the entity-relationship model of the RTSA domain.

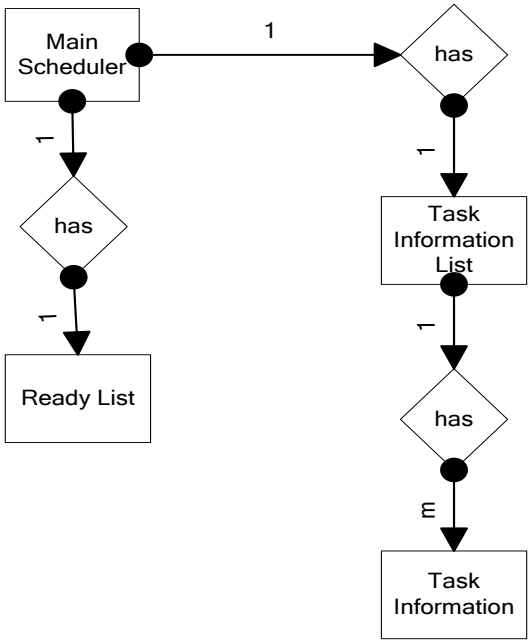

Figure 4-3 Entity Relationship Diagram for RTSA domain

The attributes and constraints on these attributes are as follows (Note that some attributes and constraints are only present under the presence of some features. This fact is expressed by indicating the associated feature in between square brackets next to the relevant item):

1. Main Scheduler attributes:
    - **Main scheduler operations:**
        - **schedule**
        - **isFeasible [Feasibility Calculation]**
        - **updateTaskInfoList [Modification]**
        - **init**
        - **deinit [Deinitialize]**
        - **start**
        - **stop [Disable]**
    - **Internals:**
        - **Task Information List**

2. Task Information List attributes:
    - **Task Information List operations:**
        - **sort**
        - **add [Add to /Remove from Ready List]**
        - **remove [Add to /Remove from Ready List]**
    - **Internals:**
        - **Task Information pointers**

3. Task Information attributes
    - **Internals**
        - **Task ID**
        - **Period [Periodic]**
        - **Deadline**
        - **Computation Time**
        - **Release Time**

4. Ready List attributes
    - **Internals**
        - **Task ID**

For the implementation concerning the scheduler component, global variables are not preferred. Whenever possible, interface functions[5] should be preferred to access data within components and sub-components.

4.4.2.2 **Feature Modeling**

Feature diagram of FORM is slightly extended to increase its modeling power. The first extension is that, inclusion relationships may or may not be directional –in the sense that x requires y can mean they both require each other or only x requires y and vice versa. Such a distinction is not available in FORM. This fact is illustrated using directed lines for feature interrelationships on the feature diagram. In the complete feature relations section in APPENDIX B, these relations are already assumed as being single way so if they are bidirectional this is understandable by examining both ends of the respective relations.

The four-level feature diagram can be seen in Figure 4-4 through Figure 4-7. Note that these figures show the features clustered into respective FORM groups. The symbolic names that appear adjacent to features and feature groups are for *concern* modeling purposes as described in section 3.2. Feature exploration and classification is based on the same resources and references mentioned in section 4.4.2.1. It should be noted that feature diagrams involve only variability information. This is generally not the case since feature modeling is necessary also in exploiting commonality and also constructing a reference architecture (RA). The reason for not presenting commonality on the feature diagrams is because the central focus of this work is variability.

[5] In case C++ is the implementation language, interface functions are true interfaces that the scheduler component implements, however in the case C is the implementation language, interface functions are meant to be functions used to access data that is internal to the scheduler.

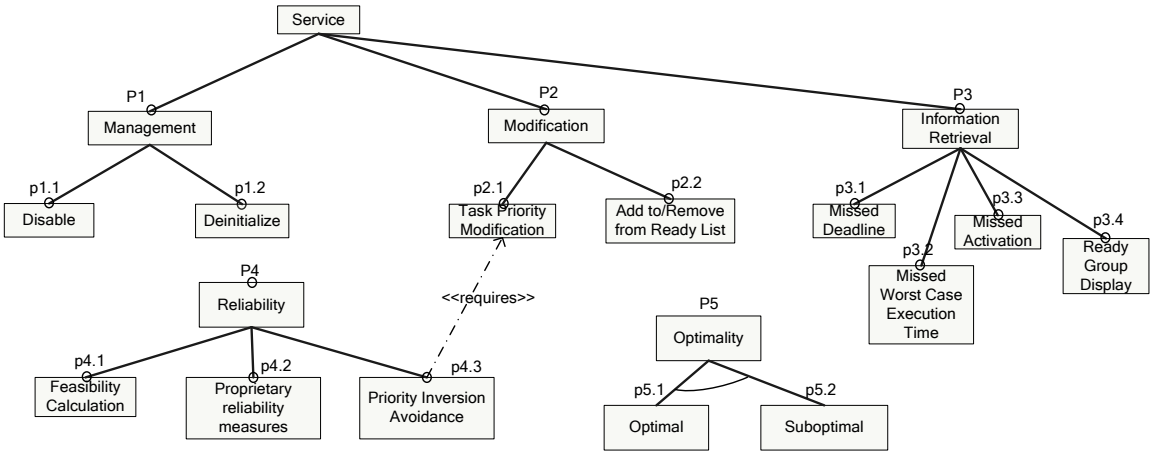

Figure 4-4 Feature diagram for RTSA domain-I Capability Layer

Note that, in the following discussions clarifying various feature diagrams, the feature variation points and feature variants are indicated inside curly brackets and square brackets, respectively. Also note that feature groups stand for the variation points and features for the variants. Regarding several feature variants, issues and decisions section in APPENDIX A provides further explanations. Please note that, when referring to issues and decisions, the term *decision* is not used in the same meaning as in [12] and in section 2.2.4, but in a strictly FORM-defined manner. That is, a decision is a particular way of resolving an issue related with the selection of features from a feature group.

The capabilities layer represents the different functional and non-functional aspects of the RTSA domain. In this case the functional aspects are gathered under the top level {Service} feature group. Generally, a real time kernel provides basic management facility over the scheduler. There should at least be a means to initialize and get the scheduler running. Since these are considered as commonality spots, they are not represented in the feature diagram in Figure 4-4 Stopping the scheduler [Disable] and deinitializing it [Deinitialize] are included optionally.

Depending on the nature of the algorithm used, the scheduler may need to allow other subsystems and certainly the kernel to make some modifications {Modification} on its data fields. These modifications are particularly made on the tasks and the task list that the scheduler maintains.

{Reliability} is another feature group –non-functional one– that plays an important role for critical real time applications. Calculating feasibility is an integral part of several scheduling algorithms such as rate monotonic scheduling (RMS) and earliest deadline first (EDF) [Feasibility calculation]. Absolute feasibility calculation requires a lot of information about the tasks to be scheduled which may not be available most of the time. For priority based preemptive algorithms, in systems that take into account dependent tasks, avoiding priority inversion can be of great importance [Priority Inversion Avoidance]. Finally several proprietary measures can be taken to increase the reliability of task scheduling [Proprietary Reliability Measures].

{Optimality} is another non-functional criterion that affects the users of the scheduler. This feature is closely related with the [Feasibility Calculation] feature grouped under {Reliability}. A feasibility calculation is expected to lead some sort of optimality. In this context by optimality absolute optimality is implied. Feasibility calculation can provide optimality to some extent but real optimality comes only with very detailed knowledge about the tasks to be scheduled and is referred to as theoretical optimality [46]. Optimal scheduling algorithms such as RMS, EDF, Least Laxity First (LLF) and Deadline Monotonic Scheduling (DMS) are optimal under some harsh and arguably unrealistic assumptions. As an example, in the case of RMS negligible preemption costs, no task interdependency, presence of solely periodic tasks are assumed [42].

As it is specified in FODA [14], depending on the particular application area, one can choose to selectively apply feature modeling at each particular level in the four level feature hierarchy that was presented (This should not be confused with the four level hierarchy of *concerns*, these are the feature layers in FODA and FORM). If the information obtained from one of the levels in the hierarchy is not applicable to the domain in question, than feature modeling at that level can be omitted partially or completely. For the RTSA domain, all the feature layers are of equal importance except the operating environment layer. This is because a good scheduling algorithm

software should work in a wide range of hardware and software platforms. Thus it is expected that such a system supports practically any possible combination that can be suggested in the feature model. There are however several operating environment layer features, presented for model completeness. The feature model for the operating environment layer can be seen in Figure 4-5.

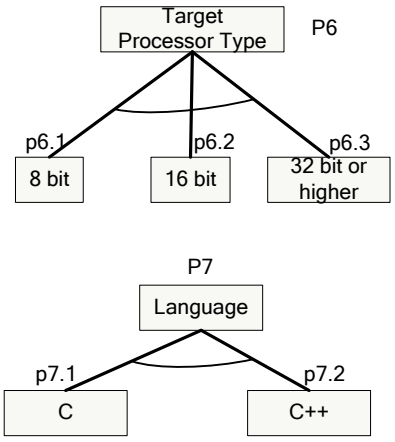

Figure 4-5 Feature diagram for RTSA domain-II Operating Environment Layer

The simple nature of operating environment layer is possible if hardware specific code is omitted from the scheduler and portable code is written. As indicated in domain scoping in section 4.4.1, although the component associated with the RTSA domain is selected to be the scheduler, scheduler component includes more information than the RTSA domain covers. In fact, the algorithmic part of the scheduler component has practically no hardware relation other than data types that are restricted by the particular hardware environment due to differences of particular processor architectures. This distinction is made clear in the processor type selection {Target Processor Type}.

The implementation language selection at the operating environment layer {Language} is restricted to either C or C++. One can argue that the correct layer to place such a feature classification is the implementation technique layer. However domain experience shows that the selection of the implementation language has fundamental effects on practically all layers in the feature model. Therefore it has been decided that the most appropriate layer to place this feature variation point is the operating environment layer.

In the domain technology layer in Figure 4-6, there are two main clusters of scheduling algorithms {Algorithm Type}. These are either online or offline algorithms. Offline algorithms require minimal run time computation and they need prior information about a task; in contrast, dynamic algorithms perform scheduling related operations during run-time and therefore are more demanding in terms of processor needs [43]. Sometimes in the literature, this classification is made by using static for the term offline and dynamic for online [42]. The distinction is that an offline algorithm has stricter constraints compared to a static one. An offline algorithm is considered as performing every scheduling related computation in the pre run time phase. This distinction is omitted in this work and static-dynamic classification is only used for priorities.

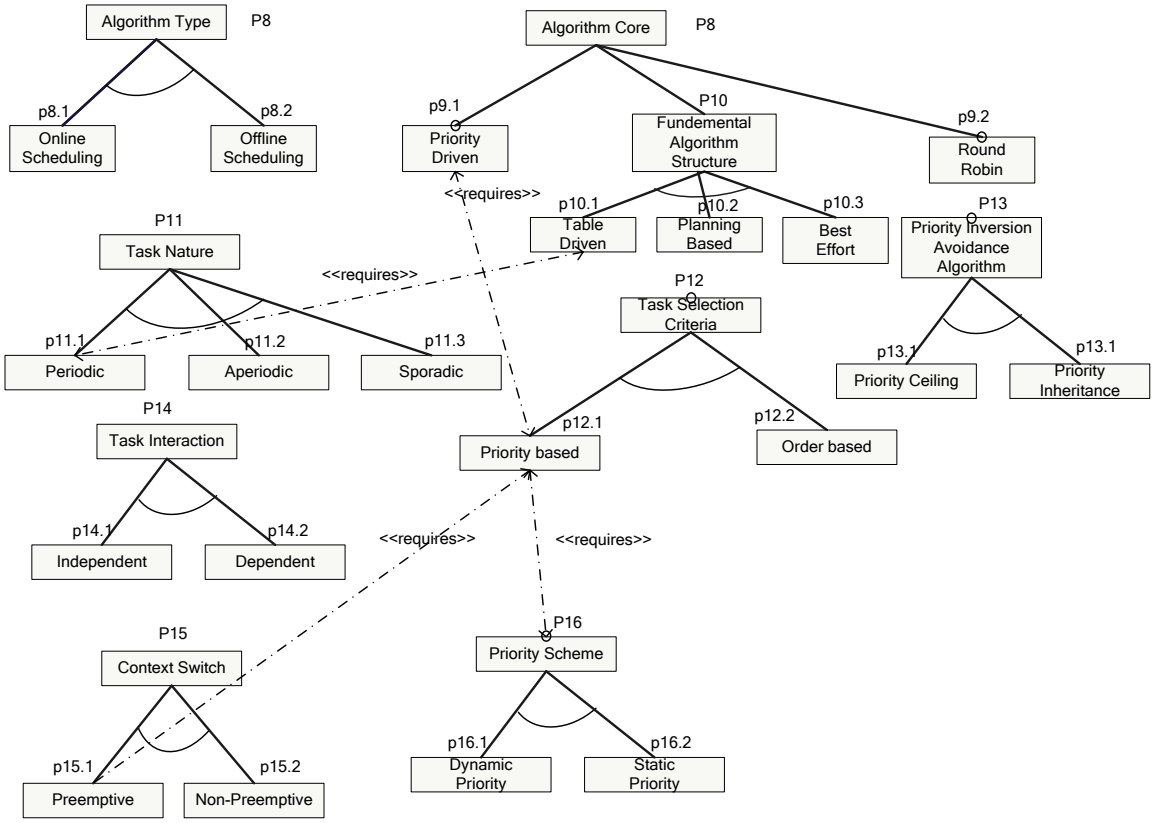

Figure 4-6 Feature diagram for RTSA domain-III Domain Technology Layer

The {Algorithm Core} feature group is of central importance for all feature layers and for the entire domain analysis. The actual value of this feature group determines the main operation principle of the scheduler. The features in this feature group and its child feature group – {Fundamental Algorithm Structure}– for the RTSA domain are not restricted to the ones on Figure 4-6, however a restriction has to be made to represent the most common ones. The reason for having a sub feature group under {Algorithm Core} is that priority driven and round robin algorithms can both be intermixed with the other fundamental algorithms [Fundamental Algorithm Structure].

{Task nature} is another feature group that classifies tasks in terms of temporal matters. Periodic tasks are obvious from their name. Aperiodic tasks are tasks with unknown activation instants and they possibly occur at an unbounded and unknown rate. Sporadic tasks are like aperiodic tasks but they occur at a bounded rate [42]. Periodic tasks are important as they are the main utensil of fundamental scheduling algorithms.

{Task selection criteria} is introduced after noticing the distinction that is present in [44] for task selection. Some algorithms explicitly select tasks based on a priority value whereas others put them in a particular order based on criteria such as laxity, rate or deadline. Although one can argue that such an ordering boils down to the same point as prioritization, a distinction makes the actual purpose clearer.

{Priority Inversion Avoidance Algorithm} feature group is particularly important when tasks share resources and they have the ability to block each other. A real time system that supports task interactions by using semaphores, mutual exclusion mechanisms, signals and events is more realistic and frequently encountered in real life. However the priority inversion problem occurs in some cases and [Priority Ceiling] and [Priority Inheritance] are two major techniques to overcome this problem. For further details on the subject, [47] provides detailed treatment.

{Task Interaction} is a feature group that describes task interrelationships. Tasks can be dependent to each other - [Dependent] - meaning that they share global variables, a shared printer or any resource and they are synchronized using rendezvous, semaphores or mutexes. This is the usual case in a large real time system and most of the algorithms overlook this fact and develop their models according to more hypothetical foundations. It has been shown in [48] that scheduling algorithms considering task dependencies –specifically those using semaphores– are NP-Hard; therefore one cannot be sure of finding polynomial-time solutions to such problems.

{Context Switch} is the feature group that represents the policy that the tasks base their preemption strategies upon. A preemptive strategy allows the exclusion of a lower priority task in favor of a high priority task whenever the higher priority task is ready to run [Preemptive]. A

non preemptive scheduler provides cooperative multitasking, a multitasking scheme used in early computer systems [Non-Preemptive].

{Priority Scheme} is a feature group related to the determination method of a task's priority. It should not be confused with the {Algorithm Type} feature group. If it is possible to change the priority of a particular scheduling algorithm at runtime then this feature has the value [Dynamic Priority] and vice versa.

Implementation technique layer, as the name implies clarifies issues that pertain to the implementation strategies applied in constructing real time schedulers. The related feature diagram can be seen in Figure 4-7.

Here, clarification is necessary, which also points to some limitations of FORM. One should not confuse implementation techniques with the actual implementation. An implementation technique is a methodical element leading to different implementation strategies – such as choosing a particular sorting algorithm over another one. Therefore, inside the actual implementation, more intricate details reside that could be related to other domain constituents in a way that is unpredicted from bare investigation of the implementation technique. FORM, by not capturing such details, could be accused of missing some important domain information.

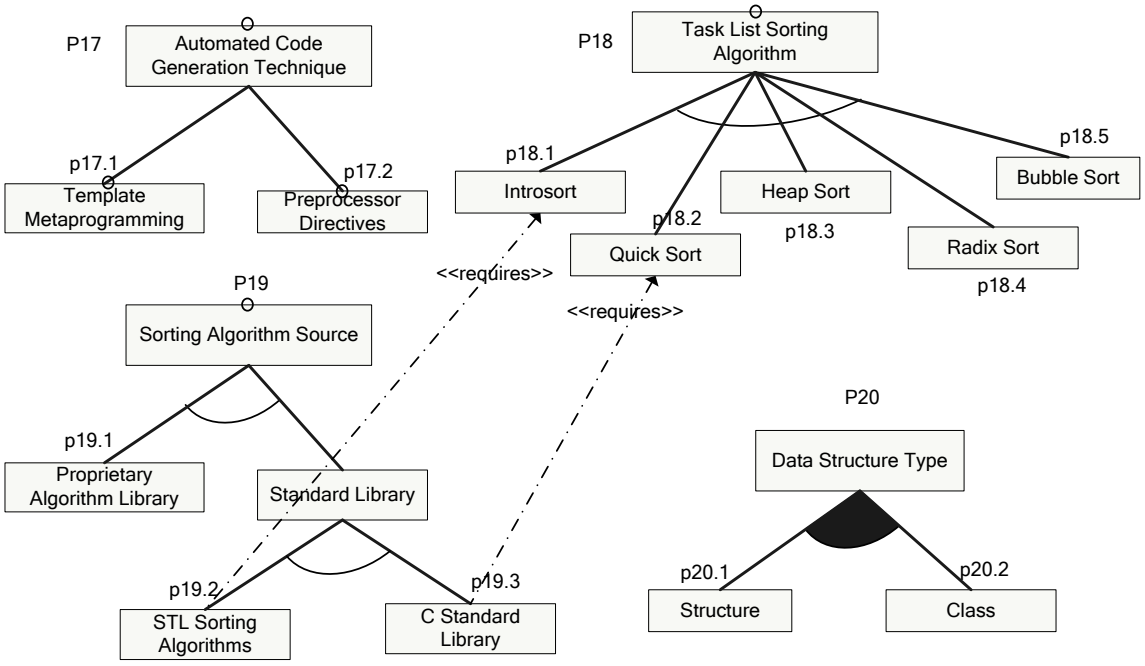

Figure 4-7 Feature diagram for RTSA domain-IV Implementation Technique Layer

{Automated Code Generation Technique} feature group refers to the method used for software generation using the model information provided by FORM. The choice [Template Metaprogramming] is only possible if the implementation language is C++. It is a special technique used in the context of generative programming. {Preprocessor Directives} is one of the classical approaches for parameterization in C and C++. {Task Sorting Algorithm} is a collection of different sorting algorithms.

{Sorting Algorithm Source} can be a private library [Proprietary Algorithm Library] or a standard library such as Standard Template Library (STL) [STL Sorting Algorithms] or the standard C library [C Standard Library]. Obviously, [Proprietary Algorithm Library] allows a wider range of {Task Sorting Algorithm} selections whereas in the case of other two, the choice is already made by the actual library implementations.

{Data Structure Type} is a feature group that specifies the particular data representation method that is used for representing relevant items in the domain. Either one of the feature variants can be selected or the data could be represented via a mixture of these two possible alternatives.

Since it is difficult to represent all feature dependencies inside a particular feature diagram layer as well as in between different feature model layers, a textual summary is necessary. This can be found in APPENDIX B in a casual format.

4.4.2.3 Reference Architecture

The usual three tiered layering in the reference architecture of FORM is not suitable for RTSA domain. Although the domain is clearly expressed in context analysis phase, it is worth mentioning at this point that RTSA for distributed systems is out of the scope of this study. Therefore a subsystem model is unnecessary.

Furthermore, schedulers are very demanding in terms of performance; therefore it is usually not preferable to allocate several processes -tasks- to accomplish this job. Some OS's prefer to use a single high priority task to handle scheduling whereas others have different strategies. For such reasons, it is usually not preferred to have many interacting processes in OS scheduler architecture. Therefore no process model is necessary in the reference architecture either. Other than that classical FORM guidelines are usable for constructing the reference architecture.

4.4.2.4 Functional Modeling

Functional modeling comprises functional and behavioral analysis. The behavioral part of the functional model has less emphasis since the domain is rather of algorithmic nature. Therefore the functional side is more relevant. The behavioral model represents some states that the scheduler goes through during its operation and can be seen in Figure 4-8. Notice the parameterizations with features in square brackets.

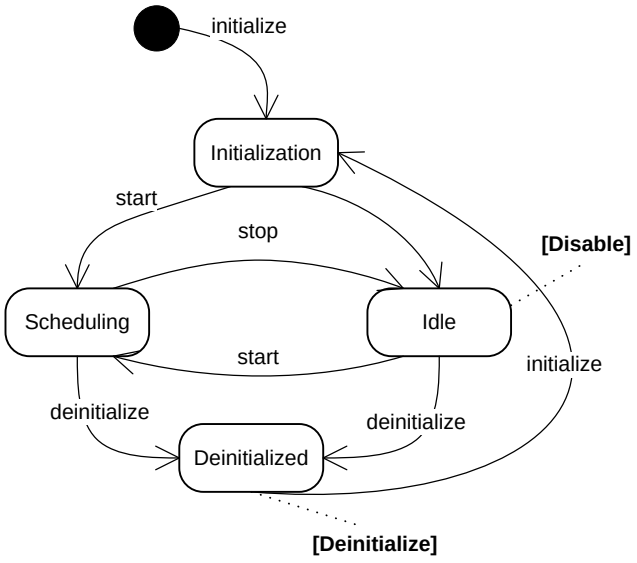

Figure 4-8 Statechart for the behavioral model of RTSA domain

The functional model emphasizes on the scheduling operation itself. Unlike the original way of specifying the functional model by using *activitycharts* of Statemate[6], classical UML activity diagrams are used. The functional model can be seen in Figure 4-9. Again, features that are directly associated with the action states are indicated on the diagram in square brackets.

[6] Statemate is currently a trademark of IBM Rational and of i-Logix Inc back at the time of the introduction of FODA.

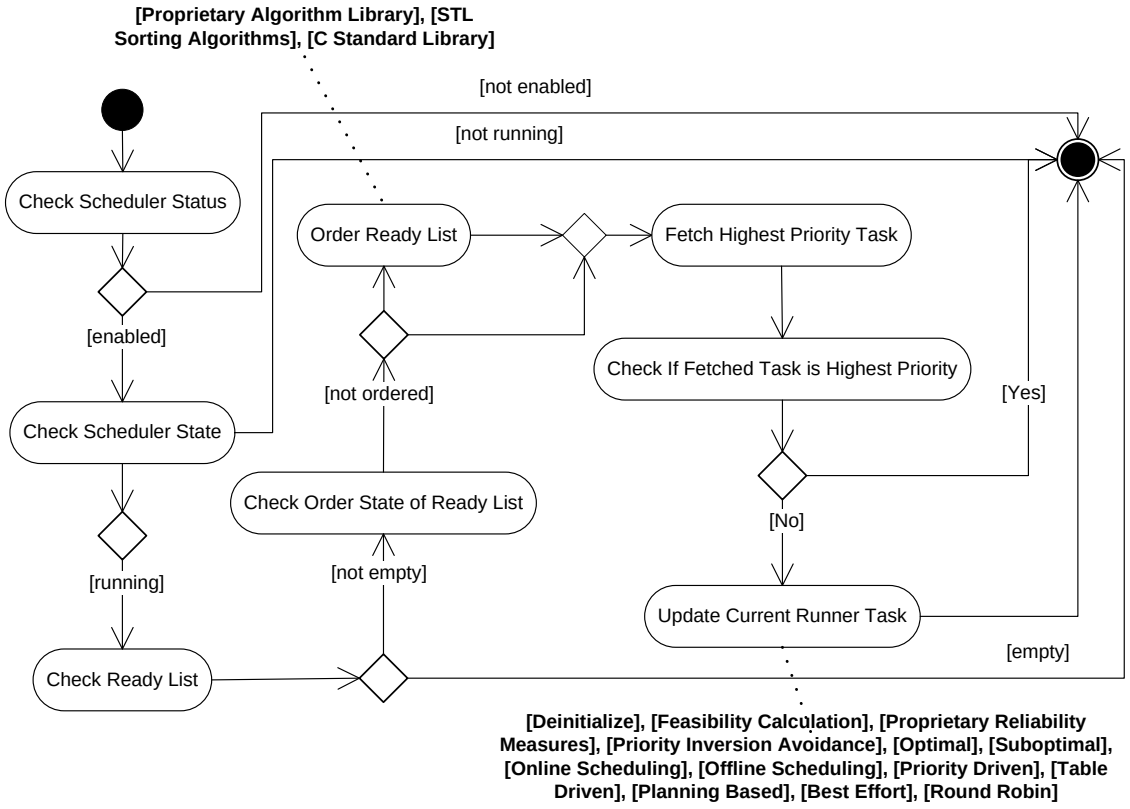

Figure 4-9 Activity Diagram for the functional model of the RTSA domain

4.4.2.5 Domain Design and Reusable Components

Although implementation of the systems belonging to the RTSA domain is not directly considered in this work, the approach to follow is indicated for completeness. For modeling the RTSA domain, it is convenient to be more restrictive in terms of what concerns the components and their structure. FORM advocates the usage of one the following reuse strategies when constructing modules in its module model [27]:

1. Selection of existing components to build up the model.
2. Template instantiation by providing external parameters.
3. Selecting and completing a skeleton code.

Since the scheduler needs to maintain close interactions with system properties, it should offer its services and demand its requests across a clean and handy interface. For this reason a proper architecture inside the module model is necessary.

FORM advices to use the concepts “modularity”, “information hiding” and “data abstraction” which are almost universal. Module construction should be carried out by clearly building interfaces primarily based on the capability layer features. Particular attention should be paid to separate data and control flows.

It is important into note that at this stage, the structure and interfaces of the components that constitute the architecture are laid out. A layered architecture that corresponds to the feature hierarchy is generally preferred.

For the particular domain of RTSA, the language of choice for implementation is either C or C++ depending on the domain’s requirements. Interfaces at module’s boundaries should be realized using clearly defined functions on a per file basis in the former case and as pure virtual functions in latter. Language choice reflects the demanding nature of the domain in terms of performance.

4.5 Modeling Real Time Scheduling Algorithms Domain with FORM and *Concerns*

In this section, variability tracking capabilities of FORM are challenged. Regarding other aspects of domain engineering for the RTSA PL, FORM is assumed to cover the ground. *Concerns* have as primary target to more thoroughly model the variability that is covered otherwise via feature diagrams in FORM. In order to achieve this, *concerns* need to model variability at a larger scope than features do.

Regarding the scope of this work, there is a limit of actual real scheduling strategies to be considered when *concern* modeling is in question. The *concern* model presented in section 3.2 represents a four level hierarchy for the *concerns*. Regarding size and resource constraints, only

the top 3 levels of the *concern* model artifacts are considered. Furthermore the *concern* model itself is not constructed covering all these three layers, but the assets that relate to the *concern* model at different abstraction layers are indicated in the various artifacts in APPENDIX C and APPENDIX D for investigating their effects which will be discussed in Chapter 5.

Due to the same resource restriction reasons, only three actual algorithms are considered when constructing PL artifacts for *concern* layers below the product feature layer. These are:

1. Rate monotonic (RM) scheduling.
2. Earliest deadline first (EDF) scheduling.
3. Priority based preemptive scheduling with round robin (RR) support for equal priorities.

For ease of reference, these systems are referred to as system I, system II and system III in the respective order that they are presented throughout the rest of the text. The requirements and software design elements associated with these systems can be found in APPENDIX C and APPENDIX D respectively.

The principle piece of work done in this section has been to construct the application model as described in section 3.3. When doing this, only the features are considered in constructing the context free grammar (CFG) both because of resource restrictions and also because features are the common means for system specification both when using or not using *concerns* for variability management. This is achieved by using the partial applicability property of the *concern* model. However, the outcome of considering three layers of *concerns* on system specification is reflected in the application model by considering the effects of concerns on the postprocessor in Figure 3-6. This is necessary to be able to contrast the advantages of both approaches.

The CFG representing the features layer of the *concern* model can be seen Table 4-1. The left side column stands for the variation point for which the grammar rule is defined. The actual feature group and feature counterparts of the symbolic names can be seen in figures from Figure 4-4 through Figure 4-7.

Table 4-1 CFG for RTSA domain, features layer

Variation Point	CFG Production Rule
Management (P1)	P1 → p1.1 | p1.2 | p1.1p1.2 | Λ
Modification (P2)	P2 → p2.1 | p2.2 | p2.1p2.2 | Λ
Information Retrieval (P3)	P3 → p3.1| p3.2 | p3.3 | p3.4 | p3.1p3.2 | p3.1p3.3 | p3.1p3.4 | p3.2p3.3 | p3.2p3.4 | p3.3p3.4 | p3.1p3.1p3.3 | p3.1p3.2p.3.4 | p3.2p3.3p3.4| p3.1p3.3| p.34 | p3.1p3.2p3.3p3.4 | Λ
Reliability (P4)	P4 → p4.1 | p4.2 | p4.3 | p4.1p4.2 | p4.1p4.3 | p4.2p4.3 | p4.1p4.2p4.3 | Λ
Optimality (P5)	P5 → p5.1 | p5.2
Target Processor Type (P6)	P6 → p6.1 | p6.2 | p6.3
Language (P7)	P7 → p7.1 | p7.2
Algorithm Type (P8)	P8 → p8.1 | p8.2
Algorithm Core (P9)	P9 → p9.1 | p9.2 | p9.1p9.2 | Λ
Fundamental Algorithm Structure (P10)	P10 → p10.1 | p10.2 | p10.3
Task Nature (P11)	P11 → p11.1 | p11.2 | p11.3
Task Selection Criteria (P12)	P12 → p12.1 | p12.2 | Λ
Priority Inversion Avoidance Algorithm (P13)	P13 → p13.1 | p13.2 | Λ
Task Interaction (P14)	P14 → p14.1 | p14.2
Context Switch (P15)	P15 → p15.1 | p15.2
Priority Scheme (P16)	P16 → p16.1 | p16.2 | Λ
Automated Code Generation Technique (P17)	P17 → p17.1| p17.2 | p17.1p17.2 | Λ
Task List Sorting Algorithm (P18)	P18 → p18.1 | p18.2 | p18.3 | p18.4 | p18.5 | Λ
Sorting Algorithm Source (P19)	P19 → p19.1| p19.2 | p19.3 | Λ
Data Structure Type (P20)	P20 → p20.1 | p20.2

As indicated before, the application model uses *javacc* to construct the parser from the rules in Table 4-1. *Javacc* is first responsible of making the lexical analysis of the input stream and then it parses recognized tokens to see whether the rules of the grammar are followed. Since the composition of variants express candidate systems in the PL, each variant should be treated as a token by *javacc*. Sample token declaration for the {Management} variation point can be seen in Listing 4-1. As an example, symbolic name for the variant string p1.1 is P1_1.

```
TOKEN: { < P1_1: "p1.1" > }
TOKEN: { < P1_2: "p1.2" > }
TOKEN: { < P1_1P1_2: "p1.1p1.2" > }
```

Listing 4-1 Token Declaration for Management Variation Point

There is an important remark regarding the nature of tokens. Token declarations are made in such a way that all possible variant selection combinations are uniquely represented by a single token. This can be seen in Listing 4-1, as the selection of both p1.1 and p1.2 is represented by the string p1.1p1.2 and the symbolic name P1_1P1_2. Also note that, in the following text, tokens derived from the same variation point are referred to as token groups.

Obviously lexicographer logic is directly implemented by *javacc* once the tokens are defined. However parsing logic is more complex and must be separately implemented. Normally parsers are specified in the Backus-Naur Form (BNF). However if such a specification is used, it is difficult to implement it so that during parsing, the arrival order of the tokens don't matter. Such a requirement is intrinsic to the *concern* application model. The appearance sequence of variants is not important for system specification purposes, their bare presence is important. In order to handle this nature of the *concern* application model, slightly more powerful means for parser specification is used rather than BNF scripts. It is possible to invoke straight java code from within *javacc* scripts by using the keyword JAVACODE. The main parser logic is implemented using this strategy.

A snippet from the method implementing this logic can be seen in Listing 4-2. The method will implement the parser logic and return to its caller method 3 possible flags indicating the status of the parsing operation. A return value of 1 indicates successful operation whereas 2 indicates a failure case in which a redundant double token is present in the system specification file (SSF). Finally a return value of 3 still indicates a failure case where end of the SSF has been reached, yet not all mandatory variations have been resolved.

Line 5 normally initializes flags that retain the presence information of all tokens, in this case for simplicity only one flag initialization is provided. Line 9 retrieves the first token from the input SSF and line 10 saves it for later use. At line 11 the loop that implements the logic checking token presences starts. As it can be seen from line 13, a switch statement over the token kinds is in place. The case statements are grouped regarding token groups as can be seen in lines 15 through 18. This makes sense since only a single selection can be made from a particular token group. Then, for each token group, a similar logic is applied. The check in line 18 is made to assure that a particular token does not occur more than once in system specification. If this is not the case, the flag indicating the presence of that token is set to true. Similar logic is applied for each token group.

```
1- // 0 : continue 1: success  , 2: duplicate token 3:not all
2- //mandatory items are present but EOF reached
3- int retVal = 0;
4- // initialize all token checker flags
5- boolean p1RetVal = false; // here single one present for
6- //example
7-
8- Token tok;
9- tok = getNextToken(); // retrieve the first token
10- saveToken(tok);
11- while(0 == retVal)
12- {
13- switch(tok.kind)
14- {
15- case P1_1:
16- case P1_2:
17- case P1_1P1_2:
18-if(p1RetVal)
19- retVal = 2;
20- else
21- p1RetVal = true;
22- break;
```

```
23- case P2_1:
24- //do similar check for all other variants…
25- }
26-//check that all mandatory items are selected, end of   27-
//tokens in the input file is reached and no double      28-
//entries exist
29-if
30- (
31- (true == (p5RetVal && p6RetVal && p7RetVal && p8RetVal &&
32- p10RetVal && p11RetVal && p14RetVal && p15RetVal &&    33-
p20RetVal)) &&
34- (tok.kind == EOF) &&
35-(retVal != 2)
36-)
37- retVal = 1;
38- //end of file reached but not all mandatory items are   39-
//selected
40- else if (tok.kind == EOF)
41- retVal = 3;
42-else if(0 == retVal)
43- {
44- tok = getNextToken();
45- saveToken(tok);
46-}
47- }
48- if(retVal == 1)
49- System.out.println("Parsing Successful!, System valid    50-
according to CFG rules");
51- else if(retVal == 2)
52- System.out.println("Parsing Failed, Redundant double entry
53- exists");
54- else if(retVal == 3)
55-System.out.println("Parsing Failed, All mandatory        56-
variability points must be resolved");
57- return retVal;
```

Listing 4-2 Parser Code for RTSA Domain, Snippet from Method *tryAllRules*

After each token group is processed, there are three options to check. The first option –on lines 29 through 37- checks that all the mandatory item flags are set to true, end of the file has been reached and no duplicate items have been processed from the SSF. This indicates a successful SSF in terms of the CFG rules. The second option –on lines 40-41– is that end of file has been reached but not all mandatory items have been encountered in the SSF. The last possible option

–on lines 42 to 46– indicate that processing is successful until now and therefore retrieves the next token from the SSF to continue back again from line 11. When an option different from 0 is obtained, looping ends and the parsing result is displayed as can be seen on lines 48 to 57.

Usage of *javacc* is handy for processing an input text stream from file or from the standard input. The post processor part of the *concern* application model is constructed using a logic that assures the mutual presence or absence of particular variants. However token parsing facilities of *javacc* are used. This is achieved by saving the valid tokens that are present in the system specification string during the lexical analysis and parsing phase of the *concern* application model for later use in the postprocessor as previously explained. The postprocessor reflects requirement and exclusion relations simply as can be seen in Listing 4-3. The switch statement on line 3 is over the token kinds. For each token there is a treatment for requires and excludes relationships. If any of these treatments end up with a failure *concernCheck* method returns false otherwise if these checks succeed the method returns true by default indicating a successful check.

```
1- private static boolean concernCheck(int tokKind)
2-{
3-switch(tokKind)
4-{
5- //previous cases for each token before
6-
7- case P4_3:
8- //Requires
9-if(!tokens[P2_1]  ||  !tokens[P2_1P2_2]  ||  !tokens[P8_2]  10-
|| !tokens[P16_1])
11-return false;
12-//Excludes
13-if(tokens[P10_1] || tokens[P8_1] || tokens[P16_2])
14-return false;
15-break;
16-
17-//following cases for each token after
18- }
19- return true;
20- }
```

Listing 4-3 Postprocessor Code for RTSA Domain, Snippet from Method *concernCheck*

Checks for requires statements make sure that, for the particular token considered within the switch clause, all of the items in the if clause are present in SSF or otherwise the check fails. On the contrary, for an excludes relationship, check is made to assure that none of the elements in the if clause are present in the SSF, or otherwise the check fails.

The logic in these code snippets forms the backbone of the *concern* application model. Postprocessor checks in the *concern* application model reflect feature relations derived from the consideration of the requirements and software design artifacts in APPENDIX C and APPENDIX D respectively can be seen in APPENDIX E (which would also be present in the *concern* application model for a complete system specification). Note that these relations have a direct affect on the postprocessing in *concern* application model and therefore dramatically change accuracy of system specification as validated in the next chapter. The counting of valid systems in terms of the *concern* application model rules is straightforward by simply incrementing a counter variable when both of the just mentioned checks succeed. Note that the post processor rules can be modified to reflect just the rules derived from feature analysis or the rules derived from feature analysis combined with *concern* modeling.

CHAPTER 5

EVALUATION OF CONCERNS

5.1 Introduction

This chapter discusses the benefits of *concerns*. In the first part of this chapter, two evaluation criteria are introduced that assess *concerns*' benefits which may be actually used to evaluate any PL approach. In the second part, these criteria are used to determine whether or not the benefits mentioned in section 3.4 are congruous in the light of the experimental work in Chapter 4.

5.2 Evaluation Criteria for Product Lines

It is worth explaining why this section has "evaluation criteria" in its title instead of "metrics". The reason is that, no matter the usefulness of the criteria presented in this section is obvious, the criteria are subjective in the sense that they are not as concrete as some metrics in the software community such as lines of code, number of methods, number of classes, etc…

The actual topic that could be discussed is whether objective criteria are really always useful in order to assess concepts that are themselves not very objective in the first place. However such a discussion is out of the scope of this work and it has been preferred to call the following two assessment entities as evaluation criteria rather than metrics.

Concerns' primary aim is to track variability effectively. As a secondary impact this is expected to bring an enhancement to system specification using domain engineering artifacts. A

successful domain engineering effort is expected to yield the users of that effort to a point where it is possible to derive numerous applications that are valuable and useful from the domain engineering products.

The first criterion that is proposed is the permissibility ratio. Whether it is any bare PL approach or an approach that has been supported with the usage of *concerns*, there is a way of specifying – possibly via merely using features in the former and via using *concerns* in the latter– actual systems in the application engineering phase. Therefore, systems are specified using the variability infrastructure that they are represented with. These specifications are checked versus the rules that are present amongst the artifacts of the product line (PL) in order to determine whether they are valid or not. The permissibility ratio is defined as the ratio of systems that comply with the imposed rules of the PL to the total possible system combinations as in Equation 5-1.

$$PR = Na / Nt$$

Equation 5-1 Permissibility Ratio

The symbols in Equation 5-1 stand for:

PR: The permissibility ratio (PR)
Na: Number of acceptable systems considering domain expert views and customers.
Nt: Number of total systems that can be generated using PL artifacts, taking into account constraints and relationships in between the artifacts.

The value of this ratio is ideally one when the relations and constraints among a PL define systems that are all valid from the point of view of actual customers –or domain experts–. As it will be demonstrated in the next section, such an ideal ratio is far from being achievable for a PL of reasonable magnitude.

Although very instructive, this ratio is difficult to obtain. The difficulty lies in validating any possible number of combinations –Nt– to obtain all the acceptable systems –Na–. Such an attempt demands a significant validation task for a number of items that is proportional to the square with the amount of variability expression items used in system specification.

Furthermore it is not straightforward to find a common ground for comparison between several PL approaches modeling variability in a different manner. Consider the example case where a PL is modeled using Feature Oriented Reuse Method (FORM) and also FORM with *concerns*. For the simplicity of the following discussion, let us call the former as model 1 and the latter as model 2.

The variability model of model 2 is expected to have fairly more variation points –and thus variants– compared to the feature model of model 1. Therefore both of the figures Na and Nt are expected to be higher for model 2 compared to model 1. However, if PR is calculated using the values of model 2, its relative value with respect to the ratio obtained for model 1 is unpredictable (in the sense that such a ratio could equally be smaller or greater than the ratio obtained using model 1). This is simply because it is not possible to determine the amount by which the model 2 differs from model 1 in terms of number of variability items.

If the variability items that are used in specifying systems are the same for different PL modeling approaches then there is an easier way of comparison. This means that if – for the sake of comparison – only the features layer of *concerns* are used in system specification from model 2 versus features themselves from model 1, the amount of permitted systems are expected to be the same. In other words Na for both models will be the same under the assumption that same set of criteria is used by the customer and/or PL experts in validating possible systems that can be derived from the models. Consider the PR's for model 1 and model 2 in Equation 5-2. For the above mentioned case, since Na_1 is equal to Na_2, the expression in Equation 5-3, namely the relative permissibility ratio (RPR) is obtained.

$$PR_1 = {Na_1}/{Nt_1} \quad PR_2 = {Na_2}/{Nt_2}$$

Equation 5-2 Permissibility Ratio for Two Models

$$RPR_{1,2} = \frac{PR_1}{PR_2} = \frac{Nt_2}{Nt_1}$$

Equation 5-3 Relative Permissibility Ratio for Two Variability Models

The RPR in Equation 5-3 is helpful in comparing the relative values of artifact numbers that can be generated taking into account constraints of the variability model. The closer this ratio is to zero, the more accurately does the variability modeling of model 2 represents actual realizable systems.

The second criterion is somewhat related to the first one. Inaccurate variability modeling is expected to increase the possibility of encountering feature interaction problems as described in section 3.4. Since it is expected that *concerns* reveal a higher number of feature interactions compared to a case lacking their support, feature interaction problems are also expected to diminish. In variability models, feature interaction problems are identified and avoided by *excludes* relationships. Therefore if for the same set of variation points and variants, if some approach is able to identify a larger number of exclusion relationships, that particular approach is more preferable in avoiding feature clashes.

An increase amount in terms of a percentage is helpful in demonstrating the effect of a particular variability modeling approach in avoiding feature interaction problems. The relative increase in the percentage for avoiding feature interaction problems – the variable A representing avoidance – is defined as in Equation 5-4.

$$A_{1,2} = \frac{En_2}{En_1} \times 100$$

Equation 5-4 Relative Feature Interaction Problem Avoidance Percentage

The fields in Equation 5-4 can further be clarified as:

$A_{1,2}$: Relative feature interaction avoidance percentage for variability models 1 and 2.
En_1: Exclusion relationship number for model 1.
En_2: Exclusion relationship number for model 2.

The value of $A_{1,2}$ is expected to be greater than hundred if model 2 is more effective in discovering exclusion relationships in the variability model and vice versa. Needless to say, for an approach supported with *concerns*, the exclusion relationships to be considered are the ones

among the features in the layer “product features” of *concerns*. Introduction of *concerns* are expected increase the exclusion relationships among features and therefore cause an increase in feature avoidance.

5.3 Evaluating *Concerns*’ Impact on Domain Engineering

In this section, the evaluation criteria described in section 5.2 are used to compare variability modeling approaches presented in sections 4.4 and 4.5. This script is presented in APPENDIX F. The computation of the items related to the evaluation criteria has been carried out using the infrastructure explained in section 4.5 and the results in APPENDIX E derived using the artifacts in APPENDIX C and APPENDIX D. The Nt_1, Nt_2 and corresponding $RPR_{1,2}$ values are calculated as 18911232, 2285568, 0.12 respectively. The number of *excludes* relations for model-1 and model 2 are 63 and 96 respectively, so $A_{1,2}$ turns out to be %52.4.

These results clearly indicate that even the partial application of *concerns* is valuable in constructing a more realistic system specification infrastructure. The value of $RPR_{1,2}$ indicates that model-2 describes actual realizable system combinations about 9 times more accurately compared to model-1. The number of excludes in model-2 are about %50 higher compared to the original case, resulting in the reduction of feature interaction problems.

CHAPTER 6

CONCLUSION

Reuse is one of the fundamental issues that software engineering methodologies try to address. Different strategies and reuse levels have been suggested for different kind of reuse problems. The problem that has been addressed in this work is reuse within a software product line. Approaches related with software product lines try to address the problem of systematically producing reusable artifacts for a family of systems rather than for a particular system as conventional software development methods do.

Satisfying the needs of a product line (PL) approach is generally significantly harder compared to single system development. This is because software analysis, design and development artifacts such as features, requirements, architecture and source code are considered for several systems rather than for a particular system. However the expected yield of such approaches is to obtain significantly reusable artifacts. When reuse is mentioned in the scope of PL's, a proactive sense of reuse for all PL-related artifacts is implied.

Obviously, for a set of systems to constitute a PL, there must be a significant amount of commonality in between them. Without enough commonality, the artifacts of the PL will tend to be far from being reusable and therefore violate a fundamental PL property. Necessary commonality must be at the initial phases. PL approaches tend to handle this during a context determination phase.

On top of commonality, variability information of a PL is essential to be able to construct new systems with similar properties but individualized according to customer needs. Variability in a PL is present in all the artifacts that make up the PL. However, traditional approaches tend to

track variability by merely using features. This results in incomplete variability information and therefore leads to several shortcomings during system specification.

To overcome this problem, *concerns* have been proposed in this thesis as a means to model and track variability in a PL. *Concerns* aim to be simple enough to enable straightforward modeling, however powerful enough to formally specify variability elements. *Concerns* cover the multidimensional nature of variability items by modeling items in a multilayer variability model. *Concerns* model basic variation point-variant relationships via the usage of a context free grammar and use a post processor checker to cover the remaining kinds of variant relationships such as simple or complex requires/excludes relationships. Two separate considerations for variability modeling enable the usage of a context free grammar for the first kind of relations, which are both simple and powerful due their formality. The postprocessor phase covers the second kind of relationships; is simple on its own and achievable using a simple rule generator program. This split modeling technique is essential for the simplicity and formality of the approach. Furthermore this formal foundation is what brings the traceability to the approach. That is if for some reason, an error has been made in the system modeling phase which violates constraints and requirements of the variability information inside the concern model, this will be detected by the context free language (CFL) parser of the application model.

As an additional property, *concern* models contain information that can be selectively applied because not all the variability modeling artifacts of all layers are needed to be used. This brings scalability to the approach in terms of resource utilization since it may not be possible to model variability in all abstraction layers due to resource constraints.

Concerns are a means to track and model PL variability and therefore they are meant to be used with an existing PL approach by replacing its intrinsic variability modeling technique. In the particular context of this work *concerns* are used to replace the traditional feature-oriented variability modeling strategy of feature oriented reuse method (FORM).

As a means to assess the benefits of using *concerns* as a replacement of variability modeling of a particular PL approach, novel evaluation criteria are introduced and advantages of *concerns* are assessed following the modeling of an example domain with FORM and with FORM and *concerns* in the light of these criteria. Furthermore, selective applicability of *concerns* is used

due to resource constraints. Tools for both supporting the *concern* application model presented in section 3.3 and measurement have been developed. *Concerns* are shown to significantly ameliorate variability modeling considering the evaluation criteria. The expectation is that these values will change further in the favor of *concern* based modeling when variability modeling is performed in all asset abstraction layers. This must be considered as a natural next step after the present study.

The following step to take after this work is to begin the construction of a tool core that realizes the underlying formal foundation of the suggested approach. Then a wrapper environment that suits the needs of a particular organization can be developed on top of this core foundation. This wrapper can possibly be a visual environment to express various PL artifacts. In this way the modeler would be isolated from the intricate details of formal modeling and could then focus on the actual task that he/she is involved with.

Concerns provide a sound amount of configuration knowledge over all domain artifacts, therefore they may be considered as an important foundation for domain specific languages. Furthermore, the ultimate objective of using variability information spread into various PL artifacts is considered to be able to use them to actually create systems rather than only specify them. For the very same reason *concerns* in the context of reflective and generative programming deserve to be explored as future work as well as their relations and inputs to DSL's.

Also, a promising study subject is the measurement of the effort required to apply the suggested variability approach that uses concerns. Such a study is necessary to see whether it is worth investing in building a concern model to express the variability information in a PL.

REFERENCES

[1]E. Lee, "What's Ahead for Embedded Software?" *Computer*, vol. 33, 2000, pp. 18-26

[2]E. A. Lee, "Embedded Software". *ADVANCES IN COMPUTERS*, vol. 56, Academic Press, London, 2002.

[3]B. P. Douglass, *Real Time UML*: *Advances in the UML for Real-Time Systems (3rd Edition)*, Addison-Wesley Professional, 2004

[4]Reed Paul, "Reference Architecture: The Best of Best Practices", 15 September 2002, Available at: http://www.ibm.com/developerworks/rational/library/2774.html, last access date: 02 August 2009

[5]R. Jigorea, S. Manolache, P. Eles and Z. Peng, "Modeling of Real Time Embedded Systems in an Object Oriented Design Environment with UML", *Object Oriented Real- Time Distributed Computing, 2000. (ISORC 2000) Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2000, pp. 210-213.

[6]Z. Kızıltan, T. Jonsson; B. Hnich, "On the definition of Concepts in Component based Software Development"

[7]I. Crnkovic, M. Larsson, *Building Reliable Component-Based Software Systems*, Artech House Publishers, 2002.

[8]K. Pohl, G. Böckle and F.J.V.D Linden, *Software Product Line Engineering: Foundations, Principals and Techniques*, Springer, 2005.

[9]H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures, Addison-Wesley Professional, 2004

[10]Garlan, D., R. Allen, and J. Ockerbloom, "Architectural Mismatch: why reuse is so hard", Software, IEEE, Vol. 12, 1995, pp. 17.26.

[11]W. James, Investment analysis of Software Product Lines, Technical Report CMU/SEI-96-TR-010 ESC-TR-96-010 , Software Engineering Institute, Carnegie Mellon University, November 1996

[12]K. Berg, J. Bishop and D. Muthig, "Tracing software product line variability:from problem to solution space, Proceedings of the 2005 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in

[13]developing countries, White River, South Africa: South African Institute of Computer Scientists and Information Technologists, 2005, pp. 182-191.

[14]K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson; *Feature Oriented Domain Analysis (FODA) Feasibility Study*, Software Engineering Institute, Carnegie-Mellon University, 1990.

[15]G. Kiczales, J. Lamping , A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier and J. Irwin, "Aspect-Oriented Programming", ECOOP'97 – *Object Oriented Programming*, 1997, p.220.

[16]G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten,, J. Palm and W. G. Griswold, "An Overview of AspectJ", *Proceedings of the 15th European Conference on Object-Oriented Programming*, Springer-Verlag, 2001, pp.327-353.

[17]K. Czarnecki, "Generative Programming Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models", Ph.D. dissertation, Technical University of Ilmenau, October 1998

[18]M. A. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang, "Organization Domain Modeling, (ODM) Guidebook, Version 2.0. Informal Technical Report for STARS, STARS-VCA025/001/00, June 14, 1996,

[19]T. Howard, SmallTalk Developers Guide to VisualWorks, The Cambridge University Press, 1995.

[20]D. Muthig, C. Atkinson, "Model-Driven Product Line Architectures", Proceedings of the Second International Conference on Software Product Lines, Springer-Verlag, 2002, pp.110-129.

[21]K. Czarnecki, C. H. Kim, "Cardinality-based feature modeling and constraints: A progress Report", 200 University Ave. West Waterloo, ON N2L 3G1, Canada, October 2005.

[22]M. Larsson, "Applying Configuration Management Techniques to Component-Based Systems", 2000.

[23]C. Szyperski, *Component Software—Beyond Object-Oriented Programming*, Addison-Wesley Professional, 1997.

[24]F. Bachman, L. Bass, C. Buhman, F. Long, J. Robert, R. Seacord and K. Wallnau, *Technical Concepts of Component-Based Software Engineering*, Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

[25]B. Meyer, "Applying Design by Contracts", *Computer*, Vol. 25, 1992, pp. 40-51.

[26]I. Jacobson, M. Griss and P. Jonsson, *Software Reuse, Architecture, Process and Organization for Business Success*, Addison-Wesley Professional, 1997.

[27]K. Kang, S. Kim, J. Lee, K. Kim, M. Huh, "FORM: A feature-oriented reuse method with domain-specific reference architectures", *Annals of Software Engineering*, vol. 5, Jan. 1998, pp. 143-168.

[28]H. Gomaa, "A Software Design Method for Real-Time Systems", Commun. ACM vol. 27, 1984, pp. 938-949

[29]Kyo C. Kang, Jaejoon Lee, Patrick Donohoe, "Feature Oriented Product Line Engineering", *IEEE Software*, vol 19, no. 4, pp. 58-65, July/Aug. 2002.

[30]M. Kim, J. Lee, K.C. Kang, Y. Hong and S. Bang, "Re-engineering software architecture of home service robots: a case study", *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, USA: ACM, 2005, pp. 505-513.

[31]Available: http://www.merriam-webster.com/dictionary, last access date: 03 August 2009.

[32]J. G. Wijnstra, "Supporting Diversity with Component Frameworks as Architectural Elements", Proceedings of the 22nd International conference on Software Engineering, Limerick, Ireland: ACM, 2000, pp 51-60.

[33]Magnus Eriksson; An approach to Software Product Line Use case Modeling; Licentiate Thesis 2006, Department of Computing Science, Umea University.

[34]F.J.V.D. Linden, K. Schmid, E. Rommes; *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, Springer-Verlag Berlin Heidelberg 2007.

[35]Available at: http://www.sun.com/bigadmin/features/articles/write_dev_driver.jsp Sun Solaris OS architecture, last access date: 03 August 2009.

[36]Available at: http://technet.microsoft.com/en-us/library/cc768129.aspx Windows NT OS Architecture, last access date: 03 August 2009.

[37]Available at: http://technet.microsoft.com/en-us/library/bb496993.aspx UNIX OS Architecture, last access date: 03 August 2009.

[38]Available at: http://www.ocera.org/archive/deliverables/ms1-month6/WP1/D1.1.html QNX Neutrino Microkernel Architecture, last access date: 03 August 2009.

[39]Available at: http://docs.huihoo.com/linux/kernel/a1/index.html#Toc_3_1, Conceptual Architecture of the Linux Kernel, Ivan Bowman, January 1998, last access date: 03 August 2009.

[40]J. J. Labrosse, *MicroC OS II, The Real Time Kernel*, CMP Books 2002

[41]Available at: http://www.bertos.org/, last access date: 03 August 2009.

[42]Arezou Mahammadi and Selim G: Akl; Scheduling Algorithms for Real Time Systems; School of Computing Queen's university Kingston Ontario Canada July 2005

[43]N. Audsley, A. Burns; Real Time System Scheduling; Department of Computer Science, University of York, UK

[44]Available at: http://www-cad.eecs.berkeley.edu/~polis/, POLIS, A design environment for control-dominated embedded systems. Version 0.4 User's manual, November 10, 1999, last access date: 03 August 2009

[45]A. Burns, "Scheduling hard real time systems: a review", *Soft. Eng. J.*, vol. 6, 1991, pp. 116-128.

[46]K. Ramamritham and J. Stankovic; "Scheduling Algorithms and Operating Systems Support for Real-Time Systems", *Proceedings of the IEEE*, vol. 82, 1994, pp. 55-67.

[47]B. P. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison-Wesley Professional, 1999

[48]A.K. Mok, FUNDAMENTAL DESIGN PROBLEMS OF DISTRIBUTED SYSTEMS FOR HARD REAL TIME ENVIRONMENTS, Massachusetts Institute of Technology, 1983.

[49]K. Czarnecki, S. Helsen, and U. Eisenecker; "Staged configuration through specialization and multi-level configuration of feature models", *Software Process Improvement and Practice*, vol. 10, no. 2 pp. 143-169, 2005.

[50]Available at: http://www.kernel.org/doc/menuconfig/, last access date: 03 August 2009.

[51]Available at: http://office.microsoft.com/en-us/excel/CH062528391033.aspx last access date: 25.02.2009.

[52]J. Bentley. "Programming pearls: Little languages", Commun. ACM, vol. 29, 1986, pp. 711-721.

[53]M. Mernik, J. Heering and A.M. Sloane, "*When and how to develop domain-specific languages*", *ACM Comput. Surv.*, vol. 37, 2005, pp. 316-344

[54]Available at: http://shark.sssup.it/; Last access date: 03 August 2009

[55]A. Metzger, K. Pohl, P. Heymans, P.Y. Schobbens and G. Saval, "Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis", in *Requirements Engineering Conference*, 2007. *RE'07. 15th IEEE International* 2007, pp. 243-253.

[56]K. Pohl, G. Böckle, and F.J.V.D. Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer, 2005.

[57]A. Metzger and P. Heymans. Comparing feature diagram examples found in the research literature. Technical report TR-2007-01, Univ. of Duisburg-Essen, 2007

[58]Avialable at: http://infolab.stanford.edu/stream/ last access date: 03 August 2009

[59]Available at: http://www.eecs.harvard.edu/~syrah/os161/ last access date: 03 August 2009

[60]Gerrit Muller, A Reference Architecture Primer, Embedded Systems Institute, 2008, Available at: http://www.gaudisite.nl/ReferenceArchitecturePrimerSlides.pdf, last access date: 03 August 2009.

[61]R. van Ommering, F. van der Linden, J. Kramer, J. Magee, "The Koala component model for consumer electronics software" *Computer*, vol. 33, 2000, pp. 78-85.

[62]D. Batory, L. Coglianese, S. Shafer, and W. Tracz; The ADAGE Avionics Reference Architecture; AIAA Computing in Aerospace-10 Conference, San Antonio, March 1995.

[63]W. Frakes and Kyo Kang, "Software Reuse Research: Status and Future" Software Engineering, IEEE Transactions on, vol. 31, 2005, pp. 529-536.

[64]D. McIlroy, "Mass produced software components", in Proceedings of the 1st international Conference on Software Engineering, Garmisch Pattenkirchen, Germany 1968, pp. 88-98

[65]E. Gamma, R. Helm, R. Johnson, J. M. Vlissides; *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994

[66]F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern Oriented Software Architcture Volume 1: A system of patterns*; Wiley 1996.

[67]J. Martin; *Introduction to Languages and the Theory of Computation*; Mc Graw-Hill Science/Engineering/Math, 2003

[68]International conference on feature interactions homepage; Available at: www27.cs.kobe-u.ac.jp/wiki/icfi/*;* last access date: 03 August 2009

[69]K. Czarnecki and U. Eisenecker, *Generative Programming, Methods Tools and Applications*, Addison Wesley Professional, 2000.

[70]J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen and J. DeBaud, "PuLSE: a methodology to Develop software product lines", Proceedings of the 1999 symposium on Software reusability, Los Angeles, California, United States: ACM, 1999, pp. 122-131.

[71]M. de Jong ve J. Visser and S. Impllang, “Grammars as Feature Diagrams”, PROCEEDINGS OF WORKSHOP ON GENERATIVE PROGRAMMING, vol.. 12, 2002, pp. 23-24.

[72]Don Batory, “Feature Models, Grammars and Propositional Formulas”, *Software Product Lines* , 2005, pp. 7-20.

[73]Available at: https://javacc.dev.java.net/, last access date: 03 August 2009

[74]Available at: http://www.ddj.com/web-development/184401948, last access date: 03 August 2009.

[75]T. Reenskaug, Working With Objects: *The Ooram Software Engineering method*, Prentice Hall, 1995.

[76]Available: http://www-01.ibm.com/software/awdtools/rup/ last access date: 02.08.2009

[77]J.R. Rumbaugh, M.R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani, Object-*Oriented Modeling and Design*, Prentice-Hall, 1990.

[78]J.M. Neighbors, “Draco: a method for engineering reusable software systems,” *Software reusability: vol. 1, concepts and models*, ACM, 1989, pp. 295-319.

[79]R. Holibaugh, Joint Integrated Avionics Working Group (JIAWG) Object Oriented Domain Analysis Method (JODA), Version 1.3, Technical Report CMU/SEI-92-SR-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. (April 13th, 2006).

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

[81]C. Atkinson, J. Bayer, D. Muthig, “Component-based Product Line Development: the KobrA approach”, *SOFTWARE PRODUCT LINE CONFERENCE, 2000*, 2000 pp. 289–309.

APPENDIX A: RTSA ISSUES AND DECISIONS

This section presents the issues and decisions raised during the feature analysis activities of RTSA domain.

- **Issue**: Algorithm reliability

Description: The decision of whether a particular scheduling algorithm should provide a mechanism of reliability check.

Raised at: Reliability

Decision: Feasibility Calculation

Note that here another level for online and offline feasibility check could have been introduced, however for matters of simplicity this is avoided.

Description: Provides feasibility calculation support to ensure reliability of an algorithm. In some cases this is determined offline and in others online (Here it should be noted that in literature sometimes online-offline and static-dynamic distinctions are used interchangeably, however this is not absolutely correct as the distinction has been made in this section).

Rationale: Offline feasibility analysis is less costly in terms of CPU resources. They are more predictable as with static algorithms.

Online feasibility analysis goes in line with online scheduling algorithms, so they are more flexible in the respect that they can handle tasks arriving at runtime.

Decision: Proprietary Reliability Measures

Description: As the name suggests this kind of measures are dependent on the particular implementation. This could be totally ad-hoc or systematic. This is mostly relevant when no explicit feasibility check is performed yet the system does its best to assure system reliability. Such measures are related generally with dynamic best effort algorithms.

Rationale: System reliability is increased. Real time behavior is under control and proactively checked.

Decision: Priority Inversion Avoidance

Description: Priority inversion is a problem that occurs when a lower priority task hinders the execution of a higher priority task because of some shared resource. Several techniques exist that can avoid this problem such as priority inheritance and priority ceiling protocols.

Rationale: Helps avoiding priority inversion problem whenever some resources are shared amongst tasks.

- **Issue:** Core of Scheduling Algorithm

Description: This is related with the determination of the principal operation logic behind a scheduling algorithm.

Raised at: Algorithm Core

Decision: Priority Driven

Description: Priority is generally a positive integer that is assigned to a particular task representing its importance and urgency [42]. Even cooperative multitasking where there is no preemption mechanism requires priorities. The concept of a task requires that a priority field is associated with it. Only background/foreground processes as explained

in [40] do not require a priority concept since they do not provide true multitasking.

Rationale: Assigning priorities to tasks is necessary in order to achieve some sort of true multitasking. Task importance is expressed through priority levels.

Decision: Table Driven

Description: Table driven approaches are generally offline scheduling algorithms – also static table driven approaches–. Such an algorithm constructs a table that indicates task scheduling information for duration equal to the least common multiple of all the periods of all tasks [42]. Then this table is used for scheduling information. Note that it is mandatory that tasks are periodic or at least they are modeled as being periodic.

Rationale: Scheduling predictability is highly increased. Run time overheads are significantly reduced. No preemption is necessary, hence this brings an ease of implementation.

Decision: Planning Based

Description: These are mainly considered under online [Algorithm Type] feature and they perform their checks based on several criteria such as resource constraints and worst case execution times [46]. They are dynamic, because they perform feasibility checks during run time and the task that successfully passes this check is guaranteed to be scheduled by the scheduler at runtime [42].

Rationale: They require a small amount of knowledge about the tasks. They are flexible and provide better response to aperiodic and soft real time tasks [42].

Decision: Best Effort

Description: These algorithms try to provide a high amount of benefit to the tasks they schedule. These algorithms do not provide an explicit feasibility check [46], they simply try to determine the best possible scheduling

scheme possible under overload conditions [42], In order to achieve this, they try to optimize a certain energy function that takes into account either the minimum task laxity or the earliest deadline or the highest number of tasks completed under load [46] etc. They are mostly in the form of priority driven preemptive scheduling algorithms with practically no assumptions about tasks.

Rationale: Proves to be successful in overload conditions. Although they are sub-optimal due no feasibility calculation, they are more realistic and cover a wide range of task characteristics spectrum.

- **Issue**: On line and Off-line Scheduling Algorithms

Description: Determination of whether an algorithm will determine the whole schedule before run time or after run time.

Raised at: Algorithm Type

Decision: Online

Description: An online scheduling algorithm is one that runs scheduling algorithms while the system is running and therefore does not require any knowledge about tasks that aren't ready for scheduling [42]. Also if any feasibility check is carried out, this is done at run-time too. They are much more flexible than offline scheduling algorithms which need to generate a whole new schedule every time task characteristics change. Note that online algorithms can be static or dynamic "priority" based.

Rationale: Since no knowledge about non-scheduled tasks is assumed, online algorithms are extremely flexible and adaptive to the changes in the system [42].

Decision: Offline

Description: All of the scheduling related decisions are performed in pre-run time phase. An offline algorithm requires practically any knowledge necessary to construct the schedule prior to system execution. This knowledge could be maximum delay, minimum delay, run time, deadline etc [42]. The actual contents of this knowledge depend on the particular offline algorithm applied. Also, if feasibility –schedulability– analysis is carried out, this is realized at compile time [46]. This kind of scheduling requires that the scheduled tasks be periodic –or modeled as being periodic–.

Rationale: Whenever run time overhead of a scheduling algorithm is desired to be minimized, offline scheduling is appropriate. Furthermore, since tasks are scheduled completely offline, their deadlines will be met under worst case conditions [46], which means they are more predictable.

- **Issue**: Priority assignment mechanism

Description: The determination instant of a task's priority if a priority based scheduling is used.

Raised at: Priority scheme

Decision: Static

Description: Priorities are assigned and do not change during entire system operation.

Rationale: Small amount of on-line resources are utilized [42]. Implementation complexity is significantly reduced.

Decision: Dynamic

Description: Priorities can change during system operation.

Rationale: Enables higher CPU utilization due more flexible scheduling. Algorithms are more flexible. This scheme proves to be powerful for handling soft real time and aperiodic tasks. [42]

- **Issue**: Task switching policy.

Description: The decision whether a task execution is based on priority of tasks that are ready to run at any instant or not.

Raised at: Context Switch

Decision: Preemptive

Description: A preemptive algorithm will stop the execution of a task if some task with higher priority becomes available for running and allocates the CPU to the higher priority task.

Rationale: The mechanism that allows a real-time behavior. High priority tasks are run whenever lower priority tasks can wait, therefore a more responsive system behavior is observed in terms of critical jobs.

Decision: Non-Preemptive

Description: A non-preemptive algorithm is one which lets a particular task run until completion whenever it has occupied the CPU. Multitasking is provided by running task after task. Such a scheme is called cooperative multitasking.

Rationale: The execution of tasks cannot be involuntarily stopped. Such an infrastructure is much easier to realize.

- **Issue**: Task interdependency

Description: The choice of whether tasks will be allowed to communicate with each other via several shared communication media.

Raised at: Task Interaction

Decision: Independent

Description: Independent tasks are ones that do not share data or communicate with each other.

Rationale: They allow much easier treatment of scheduling characteristics. Algorithms based upon the assumption of using independent tasks are reliable and predictable (for example EDF and RMS).

Decision: Dependent

Description: Dependent tasks share data and/or communicate with each other.

Rationale: Real time systems are more accurately modeled using dependent tasks since this is more realistic and is the actual case in general. Systems designed with this strategy are more likely to succeed in a broad range of real time applications.

APPENDIX B: RTSA COMPLETE FEATURE RELATIONS DEDUCED BY APPLYING FORM

It has been observed that capturing all feature relations for the analysis with FORM in 4.4 on feature diagrams is difficult and leads to a cumbersome picture. Therefore on the diagrams only major relations have been indicated. Furthermore several relations exist in between diagrams that are on different FORM feature layers which bring the impossibility of representing them on a diagram. In this section all of the feature relations are detailed for completeness.

To derive these relations, a feature matrix has been constructed and domain expertise has been used to determine all the relationships in between features. It should be noted that these set of relations are not unique and represent the modeling effort of this thesis work.

Feature name: [Task Priority Modification]
Requires: [Dynamic Priority]
Excludes: [Offline Scheduling], [Table Driven], [Static Priority], [Order Based]

Feature name: [Feasibility Calculation]
Requires: -
Excludes: [Best Effort]

Feature name: [Proprietary Reliability Measures]
Requires: -
Excludes: [Best Effort]

Feature name: [Priority Inversion Avoidance]
Requires: [Online], [Dynamic Priority], [Task Priority Modification]
Excludes: [Offline Scheduling], [Table Driven], [Static Priority]

Feature name: [Optimal]
Requires: [Periodic]
Excludes: [Non Optimal], [Planning Based], [Best Effort], [Aperiodic], [Sporadic]

Feature name: [Non Optimal]
Requires: -
Excludes: [Optimal]

Feature name: [8 bit]
Requires: -
Excludes: [16 bit], [32 bit]

Feature name: [16 bit]
Requires: -
Excludes: [8 bit], [32 bit]

Feature name: [32 bit]
Requires: -
Excludes: [8 bit], [16 bit]

Feature name: [C]
Requires: -
Excludes: [Class], [Template Metaprogramming], [STL Sorting Algorithms], [C++]

Feature name: [C++]
Requires: -
Excludes: [C]

Feature name: [Priority Driven]
Requires: [Online], [Dynamic Priority]
Excludes: [Offline], [Static Priority]

Feature name: [Table Driven]

Requires: [Periodic], [Offline]
Excludes: [Task Priority Modification], [Priority Inversion Avoidance], [Planning Based], [Best Effort], [Aperiodic], [Sporadic], [Online], [Dynamic Priority]

Feature name: [Planning Based]
Requires: [Feasibility Calculation], [Online]
Excludes: [Optimal], [Table Driven], [Best Effort], [Offline]

Feature name: [Best Effort]
Requires: [Online]
Excludes: [Optimal], [Table Driven], [Planning Based], [Offline]

Feature name: [Round Robin]
Requires: -
Excludes: [Proprietary Algorithm Library], [STL Sorting Algorithms], [C Standard Library], [Introsort], [Quicksort], [Heapsort], [Radixsort], [Bubblesort]

Feature name: [Periodic]
Requires: -
Excludes: [Aperiodic], [Sporadic]

Feature name: [Aperiodic]
Requires: [Online]
Excludes: [Optimal], [Table Driven], [Periodic], [Sporadic], [Offline]

Feature name: [Sporadic]
Requires: [Online]
Excludes: [Offline], [Periodic], [Aperiodic], [Table Driven], [Optimal]

Feature name: [Priority Based]
Requires: [Priority Driven]
Excludes: [Order Based]

Feature name: [Order Based]
Requires: -
Excludes: [Priority Based]

Feature name: [Offline]
Requires: [Periodic]
Excludes: [Task Priority Modification], [Priority Inversion Avoidance], [Planning Based], [Best Effort], [Aperiodic], [Sporadic], [Online], [Dynamic Priority]

Feature name: [Online]
Requires: -
Excludes: [Table Driven], [Offline]

Feature name: [Dynamic Priority]
Requires: [Task Priority Modification], [Priority Driven], [Online]
Excludes: [Table Driven], [Order Based], [Offline]

Feature name: [Template Metaprogramming]
Requires: -
Excludes: [C]

Feature name: [Preprocessor Directives]
Requires: -
Excludes: -

Feature name: [Proprietary Algorithm Library]
Requires: -
Excludes: [STL Sorting Algorithms], [C Standard Library]

Feature name: [STL Sorting Algorithms]
Requires: [C++]
Excludes: [C], [Proprietary Algorithm Library], [C Standard Library],

Feature name: [C Standard Library]

Requires: -

Excludes: [Proprietary Algorithm Library], [STL Sorting Algorithms]

Feature name: [Introsort]

Requires: -

Excludes: [Quicksort], [Heapsort], [Radixsort], [Bubblesort]

Feature name: [Quicksort]

Requires: -

Excludes: [Introsort], [Heapsort], [Radixsort], [Bubblesort]

Feature name: [Heapsort]

Requires: -

Excludes: [Introsort], [Quicksort], [Radixsort], [Bubblesort]

Feature name: [Radixsort]

Requires: -

Excludes: [Introsort], [Quicksort], [Heapsort], [Bubblesort]

Feature name: [Bubblesort]

Requires: -

Excludes: [Introsort], [Quicksort], [Heapsort], [Radixsort]

Feature name: [Structure]

Requires: -

Excludes: -

Feature name: [Class]

Requires: [C++]

Excludes: [C]

Feature name: [Add/Remove from Ready List]

Requires: -

Excludes: -

Feature name: [Missed Deadline]

Requires: -

Excludes: -

Feature name: [Missed Worst Case Execution Time]

Requires: -

Excludes: -

Feature name: [Missed Activation]

Requires: -

Excludes: -

Feature name: [Ready Group Display]

Requires: -

Excludes: -

Feature name: [Independent]

Requires: -

Excludes: [Dependent]

Feature name: [Dependent]

Requires: -

Excludes: [Independent]

Feature name: [Preemptive]

Requires: -

Excludes: -

Feature name: [Non Preemptive]

Requires: -

Excludes: -

APPENDIX C: RTSA EXAMPLE REQUIREMENTS SPECIFICATIONS

Note that each item in the requirement specification is terminated by a symbolic name between parentheses. These symbolic names are utilized in the *concern* model and provided here for cross reference. Next to several requirement clauses, enumerations are provided to establish a cross reference between the clause and a corresponding derived feature relation listed in APPENDIX E.

Requirements Specification for System I, RM Scheduler

[SRS 1.1]The language of implementation is C regarding performance constraints. **R.18, R.19, R.20**

[SRS 1.2]The scheduler will use a rate monotonic algorithm.

[SRS 1.3]All tasks should be of periodic nature.

[SRS 1.4]There won't be any attempt to convert any aperiodic or sporadic task into a periodic one.

[SRS 1.5]Context switch will be based on preemptive policies. **R.4, R.6, R.11, R.24, R.25**

[SRS 1.6]Priorities –derived from the periods- are statically assigned to the tasks to be scheduled.

[SRS 1.7]Kernel space should have the ability to initialize the scheduler.

[SRS 1.8]Kernel space should have the ability to deinitialize the scheduler.

[SRS 1.9]Kernel space and the user space should have the ability to start scheduling whenever desired.

[SRS 1.10]Kernel space and the user space should have the ability to stop scheduling whenever desired.

[SRS 1.11]The scheduler should have a mechanism to determine whether the predicted schedule is feasible or not. **R.22, R.3**

[SRS 1.12]It is required that the scheduling be optimal. However, absolute optimality is not aimed and optimality definitely requires that there be a feasibility check. However to be closest as possible to being optimal, every assumption in the favor of optimality has to be made. **R.1, R.2, R.4, R.5, R.6, R.12, R.13, R.14, R.15, R.21, R.22, R.23, R.25**

[SRS 1.13]For assuring optimality, any proprietary reliability check measure is despised. Optimality should be based on feasibility calculation that uses related algorithm criteria. **R.3, R.22**

[SRS 1.14]The scheduler will sort its items using C standard library's quick sort function.

[SRS 1.15]The scheduler should allow an interface function that enables adding a new item into the scheduler's ready list.

[SRS 1.16]The scheduler should allow an interface function that enables removing an existing item from the scheduler's ready list.

[SRS 1.17]The scheduler should have a predetermined size for the maximum size of its ready list.

[SRS 1.18]The ordering of the ready list must be re-determined each time a new task is added to the ready list.

[SRS 1.19]The ordering of the ready list must be re-determined each time a task is removed from the ready list.

[SRS 1.20]The scheduler should display the list of ready tasks upon request.

[SRS 1.21]The scheduler should provide and interface to report the erroneous operations when desired.

[SRS 1.22]The scheduler operates inside an OS running on a 32 bit platform. **R.7, R.8, R.9, R.10, R.16, R.17, R.19**

Requirements Specification for System II, EDF Scheduler

[SRS 2.1]The language of implementation is C, regarding performance constraints. **R.18, R.19, R.20**

[SRS 2.2]The scheduler uses earliest deadline first algorithm.

[SRS 2.3]There won't be any attempt to convert any aperiodic or sporadic task into a periodic one.

[SRS 2.4]Context switch will be based on preemptive policies. **R.4, R.6, R.11, R.25, R.24**

[SRS 2.5]Priorities –derived from the deadlines of the tasks- are statically assigned to the tasks to be scheduled.

[SRS 2.6]Kernel space should have the ability to initialize the scheduler.

[SRS 2.7]Kernel subsystem should have the ability to start scheduling whenever desired.

[SRS 2.8]Kernel subsystem should have the ability to stop scheduling whenever desired.

[SRS 2.9]The scheduler should have a mechanism to determine whether the predicted schedule is feasible or not. **R.22, R.3**

[SRS 2.10]It is required that the scheduling be optimal. However, absolute optimality is not aimed and optimality absolutely requires that there be a feasibility check. However to be closest as possible to being optimal, every assumption in the favor of optimality has to be made. **R.1, R.2, R.4, R.5, R.6, R.12, R.13, R.14, R.15, R.21, R.22, R.23, R.25**

[SRS 2.11]The results of the feasibility test should indicate missed deadlines in the case of failure of the feasibility check.

[SRS 2.12]For assuring optimality, any proprietary reliability check measure is despised, optimality should be based on feasibility calculation using related algorithm criteria.

[SRS 2.13]The scheduler should allow an interface function that enables adding a new item into the scheduler's ready list.

[SRS 2.14]The scheduler should allow an interface function that enables removing an existing item from the scheduler's ready list.

[SRS 2.15]The scheduler should have a varying size for the maximum size of its ready list. An interface to manage this size should be present.

[SRS 2.16]The ordering of the ready list must be re-determined each time a new task is added to the ready list.

[SRS 2.17]The ordering of the ready list must be re-determined each time a task is removed from the ready list.

[SRS 2.18]The scheduler will sort items in its ready list using an optimized proprietary sorting algorithm.

[SRS 2.19]The scheduler operates inside an OS running on a 16 bit platform. **R.7, R.8, R.9, R.10, R.16, R.17**

Requirements Specification for System III, Priority Based Preemptive Scheduler with RR support

[SRS 3.1]The scheduler will use a priority driven scheduling policy.

[SRS 3.2]The scheduler will also support round robin scheduling for tasks of equal priority.

[SRS 3.3]Context switch will be based on preemptive policies. **R.11, R.24, R.25**

[SRS 3.4]Tasks are generally aperiodic. They can be periodic and sporadic also.

[SRS 3.5]Priorities can be dynamically assigned to the tasks anytime.

[SRS 3.6]The scheduler should have an interface function to change the priorities of tasks.

[SRS 3.7]Kernel space should have the ability to initialize the scheduler.

[SRS 3.8]Kernel subsystem should have the ability to start scheduling whenever desired.

[SRS 3.9]Kernel subsystem should have the ability to stop scheduling whenever desired.

[SRS 3.10]Scheduler is suboptimal since it does not provide any feasibility guarantees. **R1, R2, R4, R.5, R.6, R.12, R.13, R.14, R15, R.21, R.22, R.23**

[SRS 3.11]The implementation language is C++. This is because already sub-optimality is assumed. Therefore, architectural structure and compliance to data-function separation is more important. **R.18, R.19, R.20**

[SRS 3.12]Priority inheritance technique is used to overcome priority inversion problem.

[SRS 3.13]The scheduler will sort items in its ready list using an optimized proprietary sorting algorithm.

[SRS 3.14]The scheduler should allow an interface function that enables adding a new item into the scheduler's ready list.

[SRS 3.15]The scheduler should allow an interface function that enables removing an existing item from the scheduler's ready list.

[SRS 3.16]The scheduler should have a varying size for the maximum size of its ready list. An interface to manage this size should be present.

[SRS 3.17]The ordering of the ready list must be re-determined each time a new task is added to the ready list.

[SRS 3.18]The ordering of the ready list must be re-determined each time a task is removed from the ready list.

[SRS 3.19]The scheduler operates inside an OS running on a 32 bit platform. **R.7, R.8, R.9, R.10, R.16, R.17, R.19**

APPENDIX D: RTSA EXAMPLE SOFTWARE DESIGN ELEMENTS

Note that each item in the software design artifacts possesses a symbolic name in parentheses adjacent to it. These symbolic names are utilized in the *concern* model and provided here for cross reference. Within several design artifacts, enumerations are provided –in bold typeface– to establish a cross reference between the particular artifact and a corresponding derived feature relation listed in APPENDIX E.

Use Case Diagrams and Use Cases

Figure 7-1 displays the combined use case diagram for System I, II and III. The distinction of use cases with respect to individual systems can be made considering the symbolic names on the diagram. The surrounding boundary is the system boundary for the scheduler. Note that although the Scheduling use case is present, its description is not very elaborate. This is because it has a detailed algorithmic nature and therefore is not described very well using a use case. Rather, a sequence diagram is used to represent the scheduling activity.

Note that it is not necessary to separately consider the use cases for each system since they basically state the same kind of flow of action. In the mean time it should be noted that use cases do not provide information that is directly valuable for the *concern* models (Obviously they have links to the model however they do not tend to reveal feature relationships that escaped bare feature modeling). They are present since they ease the construction of more design-related artifacts such as the class diagrams and the sequence diagrams. However, it should be noted that the reason for use cases' non-direct influence is due to the particular nature of the domain which lacks user interactions. Had the domain involved many user interactions, then use cases' inputs' would have been more direct to the *concern* model.

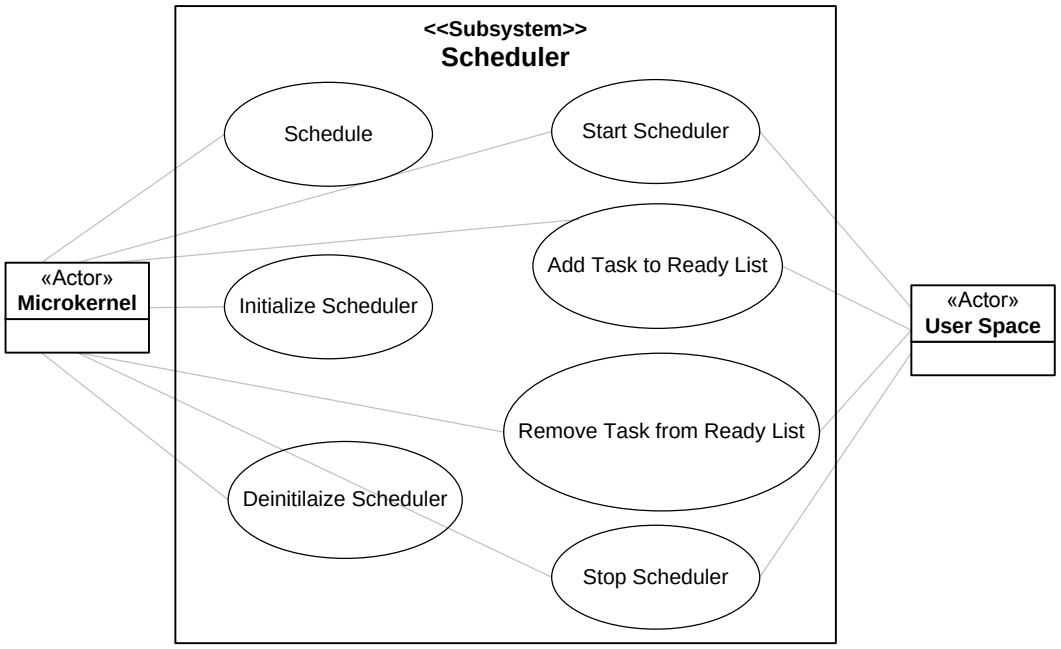

Figure 7-1 Use Case Diagram for System I, II, III

One can see the descriptions of the use cases in Figure 7-1 from Table 7-1 through Table 7-7.

Table 7-1 Use Case Initialize Scheduler

Name	Initialize Scheduler
Identifier	UC 1.1
Description	Initialize the scheduler components by setting up initial values necessary for its proper operation.
Preconditions	None in terms of scheduler, the basic OS initializations must have been done.
Post conditions	The scheduler will be ready to start operation.
Basic Course of Action	1. The use case begins when the Microkernel wants to initialize the Scheduler. 2. The Scheduler initializes all its state information. 3. The Scheduler initializes the ready task list. 4. The Scheduler checks that all initialization is done properly. **[Alt Course A]** 5. The Scheduler terminates the initialization and lets the Microkernel learn the result.
Alternate Courses	**Alternate Course A**: The scheduler initialization checks fail A 1.The scheduler does necessary cleanup, before warning the Microkernel.

Table 7-2 Use Case Deinitialize Scheduler

Name	Deinitialize Scheduler
Identifier	UC 1.2
Description	Deinitialize the scheduler components by cleaning up necessary fields.
Preconditions	The Scheduler must have been initialized.
Post conditions	The Scheduler is inactive and all its fields are cleaned up.
Course of Action	1. The use case begins when the Microkernel wants to deinitialize the Scheduler. 2. The Scheduler resets all its state information. 3. The Scheduler cleans up its ready list. 4. The Scheduler checks that all deinitialization is done properly. **[Alt Course A, Deinitialization fails]** 5. The Scheduler terminates the deinitialization and lets the Microkernel learn the result.
Alternate Courses	**Alternate Course A, Deinitialization fails**: A 1. The scheduler does necessary cleanup, before warning the Microkernel.

Table 7-3 Use Case Start Scheduler

Name	Start Scheduler
Identifier	UC 1.3
Description	Begin scheduling of the ready tasks.
Preconditions	The Scheduler must have been initialized.
Post conditions	The scheduler is operating.
Course of Action	1. The use case begins when Microkernel or the User Space wants to start the scheduler. 2. The Scheduler state is set to running. 3. The task scheduling algorithm is run on the current list of ready tasks. 4. The initial scheduling succeeds. **[Alt Course A, Scheduling fails]** 5. The Scheduler terminates the startup and lets the Microkernel or the User Space learn the result.
Alternate Courses	**Alternate Course A, Scheduling fails:** A 1.The scheduler does necessary cleanup, before warning the Microkernel.

Table 7-4 Use Case Stop Scheduler

Name	Stop Scheduler
Identifier	UC 1.4End the scheduling of tasks.
Description	End the task scheduling.
Preconditions	The scheduler must have been initialized and started.
Post conditions	The scheduler stops operating.
Course of Action	1. The use case begins when Microkernel or the User Space wants to stop scheduling. 2. The scheduler checks that there are tasks present in the ready list waiting for scheduling. **[Alt Course C, No Tasks in the Ready List]** 3. The scheduler state is set to not running. 4. The Scheduler stops running and lets the Microkernel or the User space know the result.

Table 7-5 Use Case Schedule

Name	Schedule
Identifier	UC 1.5
Description	Run the scheduling algorithm.
Preconditions	The scheduler must have been started.
Post conditions	The task scheduled according to the algorithm is running.
Course of Action	1. The use case begins when Microkernel wants to trigger scheduling. This could be when a new task is added to or an existing task is removed from the ready list. 2. The scheduling succeeds on the current set of ready tasks. **[Alt Course A, Scheduling fails]** 3. The current running task is updated. 4. The Scheduler ends scheduling and lets the Microkernel learn the result.
Alternate Courses	**Alternate Course A, Scheduling fails:** Scheduling fails due to some reason. Necessary error code preparation is made. The use case proceeds from step 4 in the main course of action.

Table 7-6 Use Case Add Task to Ready List

Name	Add Task to Ready List
Identifier	UC 1.6
Description	Add a task to the ready list to make it available for scheduling.
Preconditions	The Scheduler must have been started.
Post conditions	Task is added to the ready list. Running task is modified if necessary.
Course of Action	1. The use case begins when Microkernel or the User Space wants to add a new task to the ready list. 2. The Scheduler checks whether there is space in the task list. **[Alt Course A, No space in the ready list]** 3. The scheduler checks whether the task to be added is already in the ready list. **[Alt Course B, Task already in the ready list]** 4. The task is inserted into the ready list. 5. State of the Scheduler is updated according to the newcomer task. 6. Task scheduling algorithm is run on the current list of ready tasks. 7. Scheduling succeeds. **[Alt Course C, Scheduling fails]** 8. The Scheduler terminates the addition of the task and lets the Microkernel or the User Space learn the result.
Alternate Courses	**Alternate Course A, No Space in the Ready List:** A 1.There is not enough space in the task list. Necessary error code preparation is made. The use case proceeds from step 8 in the main course of action. **Alternate Course B, Task already in the ready list:** B 1.No task addition performed. Necessary error code preparation is made. The use case proceeds from step 8 in the main course of action. **Alternate Course C, Scheduling fails:** C 1.Scheduling fails due to some reason. Necessary error code preparation is made. The use case proceeds from step 8 in the main course of action.

Table 7-7 Use Case Remove Task from Ready List

Name	Remove Task from Ready List
Identifier	UC 1.7
Description	Remove a task form the ready list
Preconditions	The scheduler must have been started.
Post conditions	Task is removed from the ready list. The running task is modified if necessary.
Course of Action	1. The use case begins when Microkernel or the User Space wants to remove a new task from the ready list. 2. The scheduler checks whether there are some tasks present in the ready list. **[Alt Course A, No tasks in the ready list]** 3. The scheduler checks whether the task to be removed is present in the ready list. **[Alt Course B, Task not present in the ready list]** 4. The task is removed from the ready list. 5. State of the Scheduler is updated according to the removed task. 6. Task scheduling algorithm is run on the current list of ready tasks. 7. Scheduling succeeds. **[Alt Course C, Scheduling fails]** 8. The Scheduler terminates the removal of the task and lets the Microkernel or the User Space learn the result.
Alternate Courses	**Alt Course A, No tasks in the ready list:** A 1.There are no tasks in the task list. Necessary error code preparation is made. The use case proceeds from step 8 in the main course of action. **Alt Course B, No tasks in the ready list:** B 1.No task removal performed. Necessary error code preparation is made. The use case proceeds from step 8 in the main course of action. **Alt Course C, Scheduling fails:** C 1.Scheduling fails due to some reason. Necessary error code preparation is made. The use case proceeds from step 8 in the main course of action.

Component Diagrams

The only component diagram of the scheduler that realizes the RTSA domain can be seen in Figure 7-2. The reusable component displays several ports related with the services provided by the component and the required and provided interfaces on these ports. Again, as with the use cases, the primary aim in showing the component diagram is to provide a foundation for the lower layers to build upon. Note that ports are used to make a major logical partitioning for the services, which are expressed through several required or provided interfaces, which in turn are represented by collection of methods. Methods associated with the interfaces are shown in Figure 7-3. In both figures, methods and interfaces that are specific to some of the systems of the PL are explicitly indicated.

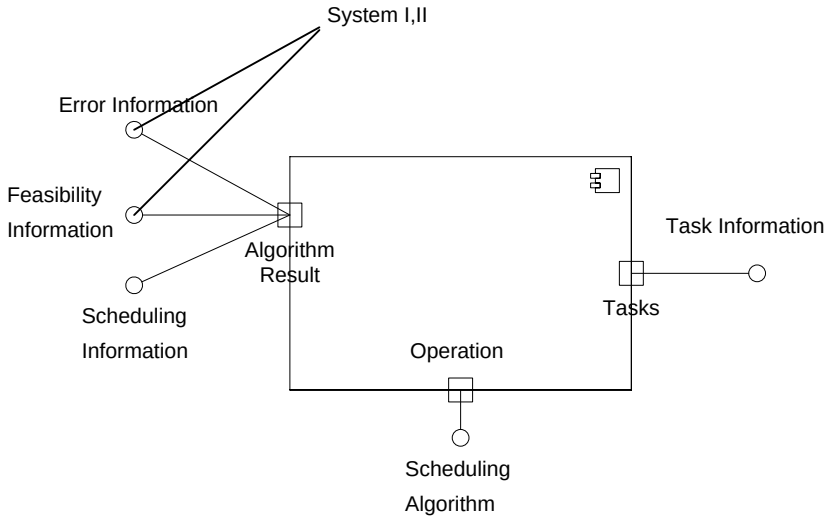

Figure 7-2 Component Diagram for System I, II, III

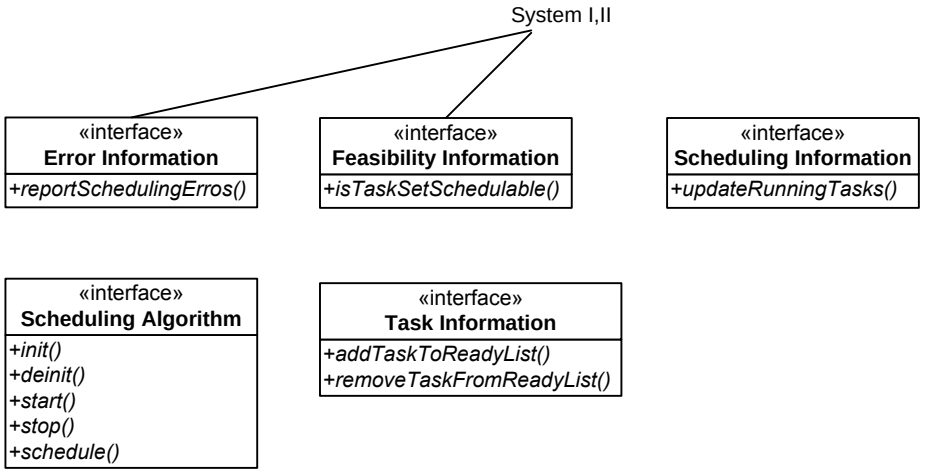

Figure 7-3 Interfaces for the scheduler component and methods

Class Diagrams

Although the realization language for some of the schedulers in this domain analysis is chosen to be C, which is a non-object oriented language, conventional UML class diagrams can still be used to express the structure information of a system realized using C [74]. Evidently, since there are no classes in C, there are no methods in the OO sense. Data groups are implemented using *structs* and the functions that operate on those data are enlisted as if they were methods.

Class diagrams for systems I, II and III are shown on a single diagram in Figure 7-4. Again as with the component diagram, methods and interfaces that are specific to some of the systems of the PL are explicitly indicated. Also note the presence of symbolic names for cross reference purposes to derived feature relations in APPENDIX E.

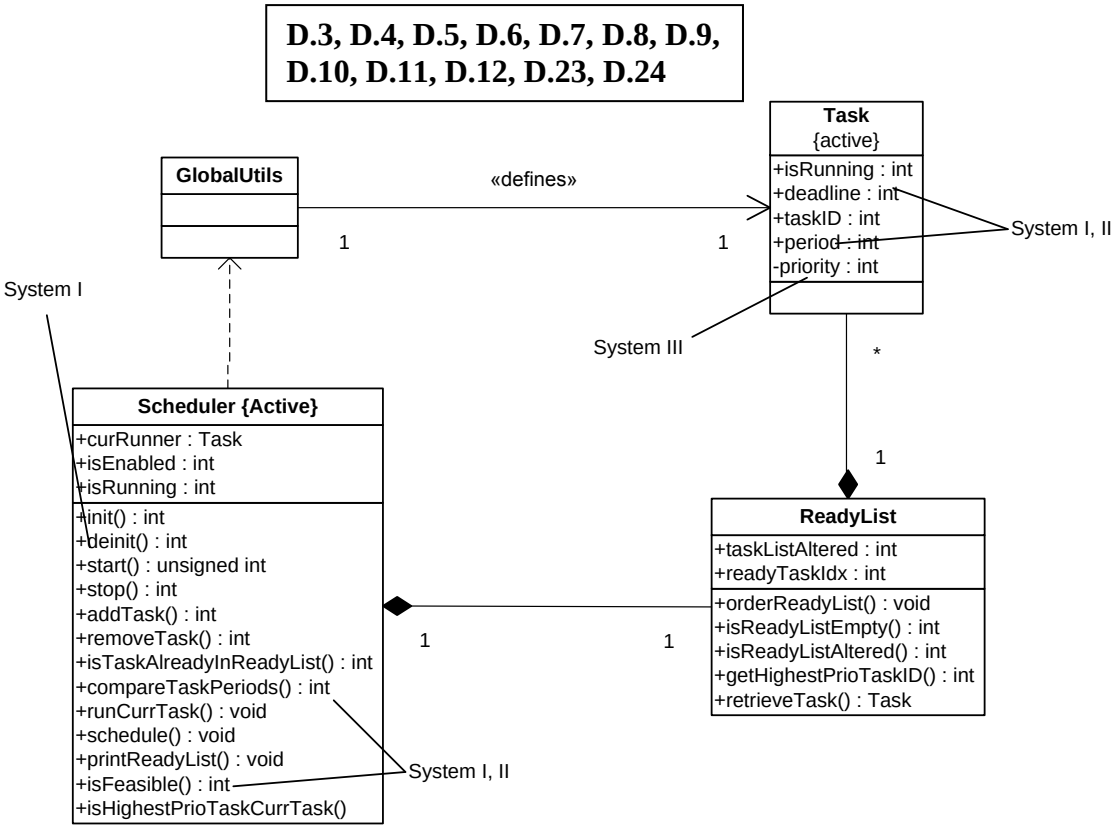

Figure 7-4 Class Diagram for System I, II, III

Sequence Diagrams

The sequence diagram that is preferred at this stage is a service level diagram rather than a system level one. This is because it is considered that system level sequence diagrams are quite sufficiently covered by use cases. For what concerns the service level sequence diagrams, the main focus is on the scheduling operation, since it is its algorithmic nature and specific demands that reveals several hidden feature relationships.

The sequence diagram related with the schedule operation for System I, II and III can be seen in Figure 7-5. The sequence diagram shows a happy path scenario, since, due to the comprehensive nature of this case, most of the unapparent feature relations are unraveled. In this case, the scheduling logic is basically same for all the systems except for the ordering of the ready list,

where the strategy depends on the particular algorithm. Again as with the class diagram, symbolic names are indicated for cross reference purposes with APPENDIX E.

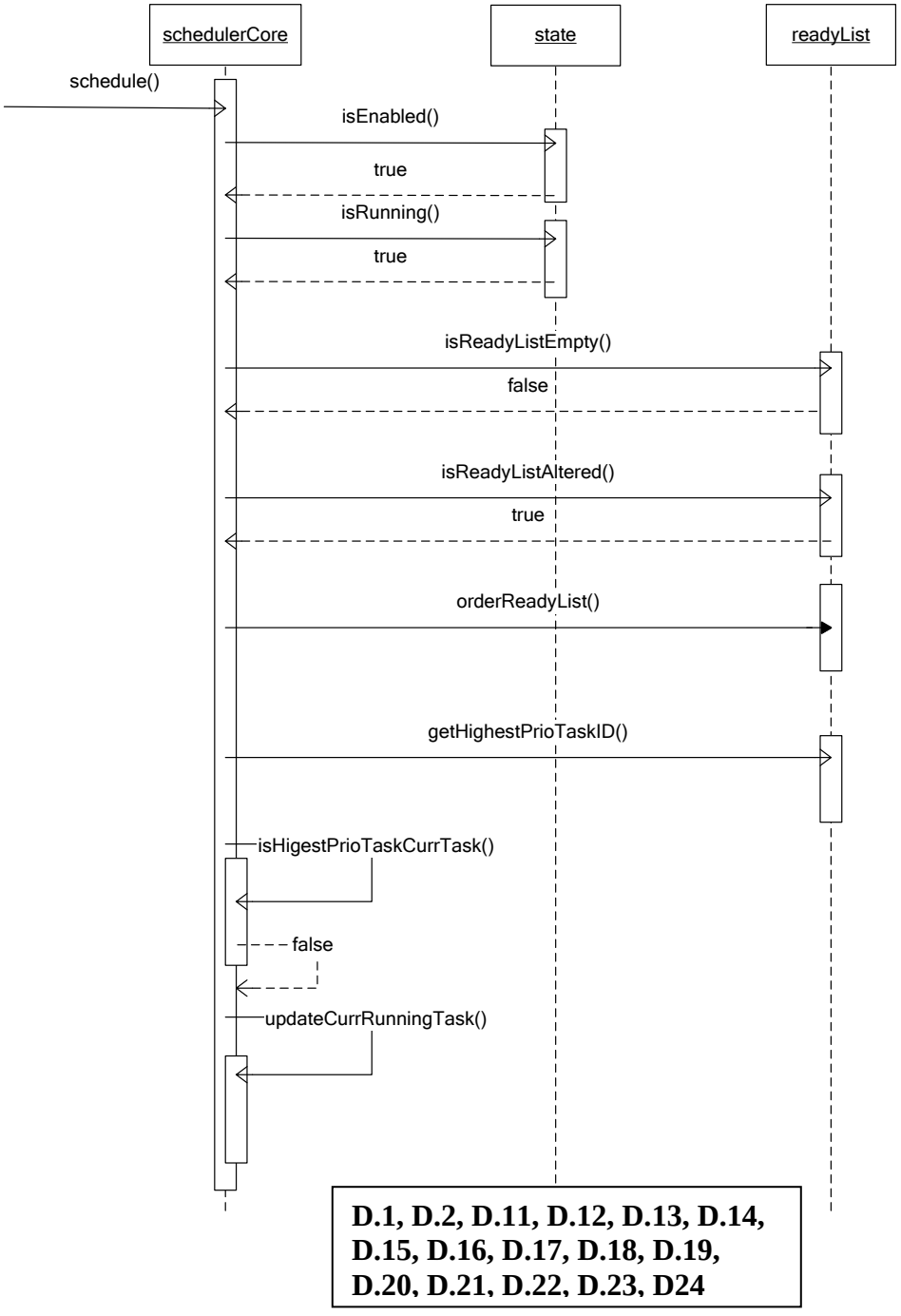

Figure 7-5 Sequence Diagram for System I, II, III

APPENDIX E: FEATURE RELATIONS DERIVED USING THE CONCERN MODEL

Note that the relationships introduced here are in addition to those in APPENDIX B. All these new relations are marked with an enumeration –in bold typeface– to establish a cross reference with the requirement and design artifacts in APPENDIX C and APPENDIX D.

Feature name: [Feasibility Calculation]
Requires: [Independent] **R.1**
Excludes: [Dependent] **R.2**

Feature name: [Optimal]
Requires: [Feasibility Calculation] **R.3**, [Preemptive] **R.4**
Excludes: [Template Metaprogramming] **D.1**, [Preprocessor Directives] **D.2**, [Class] **D.3,** [Dependent] **R.5**, [Non Preemptive] **R.6**.

Feature name: [8 Bit]
Requires: -
Excludes: [C++] **D.4**, [Template Metaprogramming] **D.5**, [STL Sorting Algorithms] **R.7**, [C Standard Library] **R.8**, [Class] **D.6.**

Feature name: [16 Bit]
Requires: -
Excludes: [C++] **D.7**, [Template Metaprogramming] **D.8**, [STL Sorting Algorithms] **R.9**, [C Standard Library] **R.10**, [Class] **D.9.**

Feature name: [C++]

Requires: -

Excludes: [Dependent] **D.10**

Feature name: [Table Driven]

Requires: -

Excludes: [Preemptive] **R.11**

Feature name: [Planning Based]

Requires: [Non-Optimal] **R.12**

Excludes: -

Feature name: [Best Effort]

Requires: [Non-Optimal] **R.13**

Excludes: -

Feature name: [Offline]

Requires: [Independent] **R.14**

Excludes: [Dependent] **R.15**

Feature name: [Template Metaprogramming]

Requires: [Optimal] **D.11**

Excludes: [8 Bit] **R.16,** [16 Bit] **R.17**

Feature name: [Preprocessor Directives]

Requires: [Optimal] **D.12**

Excludes: -

Feature name: [STL Sorting Algorithms]

Requires: [Introsort] **D.13**

Excludes: -

Feature name: [C Standard Library]

Requires: [Quicksort] **D.14**

Excludes: [Introsort] **D.15**, [Heapsort] **D.16**, [Radixsort] **D.17**, [Bubblesort] **D.18**

Feature name: [Introsort]

Requires: -

Excludes: [C Standard Library] **D.19**

Feature name: [Heapsort]

Requires: -

Excludes: [C Standard Library] **D.20**

Feature name: [Radixsort]

Requires: -

Excludes: [C Standard Library] **D.21**

Feature name: [Bubblesort]

Requires: -

Excludes: [C Standard Library] **D.22**

Feature name: [Structure]

Requires: -

Excludes: [Class] **R.18**

Feature name: [Class]

Requires: -

Excludes: [8 Bit] **R.19**, [Structure] **R.20**

Feature name: [Independent]

Requires: -

Excludes: [Optimal] **R.21**

Feature name: [Dependent]

Requires: -

Excludes: [Feasibility Calculation] **R.22**, [Optimal] **R.23**

Feature name: [Preemptive]

Requires: [Priority Driven] **D.23**, [Priority Based] **D.24**

Excludes: [Order Based] **R.24**

Feature name: [Non Preemptive]

Requires: -

Excludes: [Optimal] **R.25**

APPENDIX F: SOURCE CODE OF APPLICATION MODEL FOR RTSA DOMAIN

Please see the enclosed CD.