Integration of Quality Attributes in Software Product Line Development.

Tesis de Máster en Ingeniería del Software,

Métodos Formales y Sistemas de Información

(MISMFSI)

Grupo de Ingeniería del Software y Sistemas de Información (ISSI)

Departamento de Sistemas Informáticos y Computación (DSIC)

Universidad Politécnica de Valencia (UPV)

Septiembre 2011

Javier González Huerta

Directores:

Dr. Silvia Mara Abrahão Gonzales

Dr Emilio Insfran Pelozo

Integration of Quality Attributes in Software Product Line Development

Trabajo Final de Máster presentado para cumplir con los requisitos finales para la obtención del título de Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información, de la Universidad Politécnica de Valencia, 2011.

Agradecimientos.

A Silvia y Emilio por su esfuerzo, por su constancia por su apoyo y por estar ahí siempre que les necesito.

A Michel por su acogida, sus consejos, su actitud crítica, sus ánimos, por estar siempre dispuesto y sobre todo por haberme enseñado a ver las cosas de otro modo.

A Rosa por sus comprensión, por su paciencia, por sus ánimos, por apoyarme siempre en los buenos y no tan buenos momentos y por ayudarme en la ausencia.

A Isidro por sus siempre sabios consejos y críticas.

A Adri, Sonia, Alex, Juan Antonio, Abel y David, por su ayuda, por estar siempre ahí, por ser tan magníficos compañeros.

A Werner, Hafeez, Bilal, Dave, Christoph, Luuk, y todo el equipo del LIACS por haberme hecho sentir uno más..

A Ramin por su paciencia y sus consejos que han dado como fruto parte de este trabajo.

Resumen

En los últimos años se han propuesto diferentes aproximaciones para el desarrollo de sistemas complejos. Algunos esfuerzos intentan aplicar la aproximación de Líneas de Producto Software tratando de sacar partido de la reutilización masiva para producir sistemas software que comparten un conjunto común de características. En general, el aseguramiento de la calidad del producto es una actividad crucial para el éxito de la industria del software, pero es, si cabe, más importante cuando se trata del desarrollo de líneas de producto software, dado que la reutilización masiva de activos software hace que los atributos de calidad (propiedades físicas o abstractas de un artefacto software) de los activos software impacten en la calidad de todos los productos de una línea de producto.

Sin embargo, a pesar de la importancia que la calidad tiene en el desarrollo de líneas de producto de software, la mayoría de las metodologías aplicadas en su desarrollo se centran únicamente en la gestión de la variabilidad en líneas de producto, sin dar soporte a los requerimientos no funcionales que el producto debe cumplir. El principal objetivo de este trabajo fin de master es la introducción de atributos de calidad en fases tempranas de los procesos de desarrollo de líneas de producto software, mediante la definición de un multi-modelo que represente las distintas vistas de un sistema software (funcionalidad, variabilidad y calidad) e integre la calidad como una vista más del sistema para describir la extensión de la línea de producto software y de un plan de producción que introduzca los atributos de calidad como un factor de decisión a la hora de seleccionar entre distintas alternativas de diseño.

Nuestra propuesta ha sido definida siguiendo el paradigma de Desarrollo Dirigido por Modelos. Por lo tanto, todos los artefactos definidos cuentan con sus correspondientes metamodelos y todos los procesos descritos se apoyan en transformaciones de modelos. Por ultimo, para ilustrar la factibilidad de la propuesta, hemos integrado la vista de calidad en un ejemplo de línea de productos en el contexto de los sistemas de seguridad critica en el dominio automovilístico.

Resum

Als últims anys s'han proposat diferents aproximacions per al desenvolupament de sistemes complexos. Alguns esforços han intentat aplicar la aproximació de Línies de Producte de Programari tractant de treure partit de la reutilització massiva per a produir sistemes de programari que comparteixen un conjunt comú de característiques. En general, l'assegurament de la qualitat de producte es una activitat crucial per a l'èxit de la industria de la programaria, però es encara mes important quan es tracta del desenvolupament de línies de producte de programari, atès que la reutilització massiva d'actius, el que fa que els atributs de qualitat (propietats físiques o abstractes d'una artefacte de programari que es poden mesurar) dels actius impacten en la qualitat de tots els productes de la línia de producte.

No obstant, a pesar de la importància que la qualitat te en el desenvolupament de les línies de producte de programari, la majoria de les metodologies aplicades en el seu desenvolupament es centren únicament en la gestió de la variabilitat en la línia de productes, sense donar suport als requeriments no funcionals que el producte te que acomplir. El principal objectiu d'aquest treball fi de màster es la introducció de la vista dels atributs de qualitat des de les primeres fases del desenvolupament de línies de producte de programari, mitjançant la definició d'un multi-model que representi les distintes vistes d'un sistema de programari (funcionalitat, variabilitat i qualitat) i que integri la qualitat de la línia de productes de programari i d'un pla de producció que introdueixi els atributes de qualitat com un factor de decisió a l'hora de seleccionar entre les distintes alternatives de disseny.

La nostra proposta ha estat definida seguint el paradigma de desenvolupament dirigit per models. Per la qual cosa, tots els artefactes definits compten amb el seu corresponent metamodel y tots els processos descrits es recolzen en transformacions de models. Per últim, per tal d'il·lustrar la factibilitat de la proposta, hem integrat la vista de qualitat en un eixample de línia de productes en el context dels sistemes de seguretat critica en el domini automobilístic.

Abstract

Different approaches for building modern software systems in complex and open environments have been proposed in the last few years. Some efforts try to apply Software Product Line (SPL) approach to take advantage of the massive reuse for producing software systems that share a common set of features. In general quality assurance is a crucial activity for success in software industry, but it is even more important when talking about Software Product Lines since the massive reuse of assets makes the quality attributes (a measurable physical or abstract property of an software artifact) of the assets impact over the quality of the whole set of products within the product line.

However, despite the importance that quality has in software product line development, most of the methodologies being applied in Software Product Line Development focus only on managing the commonalities and variability within the product line and not giving support to the non-functional requirements that the products must fit. The main goal of this master final work is to introduce quality attributes in early stages of software product line development processes by means of the definition of a multi-model which represents the different views of a software system (functionality, variability and quality) and integrates quality as an additional view for describing the extension of the software product line, and a production plan that introduces the quality attributes as a decision factor during product configuration and when selecting among design alternatives.

Our approach has been defined following the Model- Driven Software Development paradigm. Therefore all the software artifacts defined had its correspondent metamodels and the processes defined rely on automated model transformations. Finally in order to illustrate the feasibility of the approach we have integrated the quality view in an SPL example in the context of safety critical embedded systems on the automotive domain.

Contents

List of figures

List of Tables

Acronyms

AADL	Architecture Analysis and Design Language
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDSD	Model Driven Software Development
OCL	Object Constraint Language
OMG	Object Management Group
QVT	Query View Transformation
SPL	Software Product Line
SPLE	Software Product Line Engineering
SQuaRE	Software product Quality Requirements and Evaluation
SysML	Systems Modeling Language
UML	Unified Modeling Language

UNIVERSIDAD POLITECNICA DE VALENCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Integration of Quality Attributes in Software Product Line Development

Tesis de Máster en Ingeniería del Software,

Métodos Formales y Sistemas de Información

(MISMFSI)

Grupo de Ingeniería del Software y Sistemas de Información (ISSI)

Departamento de Sistemas Informáticos y Computación (DSIC)

Universidad Politécnica de Valencia (UPV)

Septiembre 2011

Javier González Huerta

Directores:

Dr. Silvia Mara Abrahão Gonzales

Dr Emilio Insfran Pelozo

Integration of Quality Attributes in Software Product Line Development

Trabajo Final de Máster presentado para cumplir con los requisitos finales para la obtención del título de Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información, de la Universidad Politécnica de Valencia, 2011.

Agradecimientos.

A Silvia y Emilio por su esfuerzo, por su constancia por su apoyo y por estar ahí siempre que les necesito.

A Michel por su acogida, sus consejos, su actitud critica, sus ánimos, por estar siempre dispuesto y sobre todo por haberme enseñado a ver las cosas de otro modo.

A Rosa por sus comprensión, por su paciencia, por sus ánimos, por apoyarme siempre en los buenos y no tan buenos momentos y por ayudarme en la ausencia.

A Isidro por sus siempre sabios consejos y criticas

A Adri, Sonia, Alex, Juan Antonio, Abel y David, por su ayuda, por estar siempre ahí, por ser tan magníficos compañeros.

A Werner, Hafeez, Bilal, Dave, Christoph, Luuk, y todo el equipo del LIACS por haberme hecho sentir uno mas.

A Ramin por su paciencia y sus consejos que han dado como fruto parte de este trabajo.

Resumen

En los últimos años se han propuesto diferentes aproximaciones para el desarrollo de sistemas complejos. Algunos esfuerzos intentan aplicar la aproximación de Líneas de Producto Software tratando de sacar partido de la reutilización masiva para producir sistemas software que comparten un conjunto común de características. En general, el aseguramiento de la calidad del producto es una actividad crucial para el éxito de la industria del software, pero es, si cabe, más importante cuando se trata del desarrollo de líneas de producto software, dado que la reutilización masiva de activos software hace que los atributos de calidad (propiedades físicas o abstractas de un artefacto software) de los activos software impacten en la calidad de todos los productos de una línea de producto.

Sin embargo, a pesar de la importancia que la calidad tiene en el desarrollo de líneas de producto de software, la mayoría de las metodologías aplicadas en su desarrollo se centran únicamente en la gestión de la variabilidad en líneas de producto, sin dar soporte a los requerimientos no funcionales que el producto debe cumplir. El principal objetivo de este trabajo fin de master es la introducción de atributos de calidad en fases tempranas de los procesos de desarrollo de líneas de producto software, mediante la definición de un multi-modelo que represente las distintas vistas de un sistema software (funcionalidad, variabilidad, calidad) e integre la calidad como una vista más del sistema para describir la extensión de la línea de producto software y de un plan de producción que que introduzca los atributos de calidad como un factor de decisión a la hora de seleccionar entre distintas alternativas de diseño.

Nuestra propuesta ha sido definida siguiendo el paradigma de Desarrollo Dirigido por Modelos. Por lo tanto, todos los artefactos definidos cuentan con sus correspondientes metamodelos y todos los procesos descritos se apoyan en transformaciones de modelos. Por ultimo, para ilustrar la factibilidad de la propuesta, hemos integrado la vista de calidad en un ejemplo de línea de productos en el contexto de los sistemas de seguridad critica en el dominio automovilístico.

Resum

Als últims anys s'han proposat diferents aproximacions per al desenvolupament de sistemes complexos. Alguns esforços han intentat aplicar la aproximació de Línies de Producte de Programari tractant de treure partit de la reutilització massiva per a produir sistemes de programari que comparteixen un conjunt comú de característiques. En general, l'assegurament de la qualitat de producte es una activitat crucial per a l'èxit de la industria de la programaria, però es encara mes important quan es tracta del desenvolupament de línies de producte de programari, atès que la reutilització intensiva d'actius, el que fa que els atributs de qualitat (propietats físiques o abstractes d'una entitat que es poden mesurar) dels actius seran transmeses a tot l'abast de la línia de producte.

No obstant, a pesar de la importància que la qualitat te en el desenvolupament de les línies de producte de programari, la majoria de les metodologies aplicades en el seu desenvolupament es centren únicament en la gestió de la variabilitat en la línia de productes, sense donar suport als requeriments no funcionals que el producte te que acomplir. La principal meta d'aquest treball fi de màster es s la introducció de la vista dels atributs de qualitat des de les primeres fases del desenvolupament de línies de producte de programari, mitjançant la definició d'un pla de producció que per un costat integri la qualitat como una vista addicional del sistema per a descriure la extensió de la línia de producte, y per l'altre que introdueixi els atributes de qualitat com un factor de decisió a l'hora de seleccionar entre les distintes alternatives de disseny.

La nostra proposta ha estat definida seguint el paradigma de desenvolupament dirigit per models. Per la qual cosa, tots els artefactes definits compten amb el seu corresponent metamodel y tots els processos descrits es recolzen en transformacions de models. Per últim, per tal d'il·lustrar la factibilitat de la proposta, hem integrat la vista de qualitat en un eixample de línia de productes en el context dels sistemes de seguretat critica en el domini automobilístic.

Abstract

Different approaches for building modern software systems in complex and open environments have been proposed in the last few years. Some efforts try to apply Software Product Line (SPL) approach to take advantage of the massive reuse for producing software systems that share a common set of features. In general quality assurance is a crucial activity for success in software industry, but it is even more important when talking about Software Product Lines since the intensive reuse of assets makes the quality attributes (a measurable physical or abstract property of an entity) of the assets to be transmitted to the whole SPL scope.

However, despite the importance that quality has in software product line development, most of the methodologies being applied in Software Product Line Development focus only on managing the commonalities and variability within the product line and not giving support to the non-functional requirements that the products must fit. The main goal of this master final work is to introduce quality attributes in early stages of software product line development processes by means of the definition of a production plan that, on one hand, integrates quality as an additional view for describing the extension of the software product line and, on the other hand introduces the quality attributes as a decision factor during product configuration and when selecting among design alternatives.

Our approach has been defined following the Model- Driven Software Development paradigm. Therefore all the software artifacts defined had its correspondent metamodels and the processes defined rely on automated model transformations. Finally in order to illustrate the feasibility of the approach we have integrated the quality view in an SPL example in the context of safety critical embedded systems on the automotive domain.

Contents

List of figures

List of Tables

Acronyms

AADL	Architecture Analysis and Design Language
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDSD	Model Driven Software Development
OCL	Object Constraint Language
OMG	Object Management Group
QVT	Query View Transformation
SPL	Software Product Line
SPLE	Software Product Line Engineering
SQuaRE	Software product Quality Requirements and Evaluation
SysML	Systems Modeling Language
UML	Unified Modeling Language

Chapter 1. Introduction

Different approaches for building modern software systems in complex and open environments have been proposed in the last few years. Some efforts try to apply the Software Product Line (SPL) approach to take advantage of the massive reuse for producing software systems that share a common set of features. However, most of the existing methodologies focus only on managing the commonalities and variability within the product and they do not give the appropriate support to the non-functional requirements that the products must meet.

As a means for improving the quality of current practices in the development of Software Product Lines, the main goal of this Master's Thesis is to introduce quality attributes in early stages of Software Product Line Development by means of the definition of a multi-model and a production plan, conducted by this multi-model [9], that together integrates quality as another view of the system and applies the quality attributes as the decision factor during the product configuration and when selecting among design alternatives.

1.1. Motivation

The challenge in today's software engineering is to deliver high-quality software on-time and within budget to the customers [56]. This challenge has been addressed from many different approaches. In recent years, Software Product Lines (SPL) has emerged as a promising approach to improve software development productivity in IT industry.

Software Product Line Engineering (SPLE) [14], [59] is a development paradigm for producing a set of software-intensive systems sharing a common set of managed features that are developed from a common set of core assets in a prescribe way. These core assets are going to be reused systematically in the product development when configuring specific products. The SPLE comprises two main development phases: *core asset development* that deals with the development of the different software artifacts that will be used for production of products and *product development* that reuses these core asses to configure the multiple products that constitute the software product line. The application of this strategy improves productivity, time to market for individual products, decreases the cost and labor needs and increase quality of individual products.

It is worth noting that the application of SPL quite often relies on Model-Driven Software Development (MDSD), at least partially, for raising the level of abstraction and representing different models, i.e. the variability models, that are present in almost all SPL development process. MDSD is a proposal to maximize productivity, enhancing aspects such as software reusability, interoperability and improved adaptation of technological change. One of the most popular approaches is the Model Driven Architecture (MDA) [51], which is based on the set of standards proposed by the Object Management Group (OMG). MDA advocates taking models as the main artifacts of software development and deriving the product as a series of model transformations.

Software Quality is the capability of the software product to satisfy stated and implied needs (requirements) when used under specific conditions [26]. The assessment of quality in software products is also a reflection of diverse points of view and it is addressed in different phases of software development product life cycle. Quality assurance is a crucial activity for success in the software industry, but it is even more important in Software Product Lines in SPL because an error in the common architecture or in core assets may be propagated to the final products..

However, despite the importance that quality has in software product line development, research works dealing with SPL usually focuses mainly on managing a single view of the system: the variability view. Managing the variability of a SPL implies, on one hand managing the domain features expressed by means of a feature model and, on the other hand, that this variability must be supported by core assets. However, the success of a software product relies not only in the fulfillment of its functional requirements but also the quality attributes (a measurable physical or abstract property of an entity) that the product must fulfill. Describing a SPL only with a variability view (even when extended with non-functional attributes) does not reflect the real extension of the product line.

The adoption of the Software Product Line approach must focus not only on managing the variability but also on expressing, on one hand, how the quality attributes of the core assets impact over the quality of the SPL, and on the other hand, how the selection of features or core assets impact the quality of the products during product configuration. If we can establish and express in a model the relationships between features and/or core assets and quality attributes, these quality attributes can become a decision factor when selecting among design alternatives. This selection can be made automatically by using model transformation techniques.

Although addressing the quality attributes when assembling a product is a way of introducing the quality perspective on the product development phase, our goal is to introduce these concepts at early stages of the product line development process. During a model transformation, which may occur in any stage of the product line development process, usually we can identify alternative transformations (entities in the source model that have more than one representation in the target model) that produce alternative models that have the same functionality but are different with regard to its quality attributes. In those cases, the transformation designers need to identify possible alternative transformations and choose those alternatives that produce the target model with the desired quality attributes. This can be improved by defining model transformation processes where the quality attributes are the decision factor among those alternatives, obtaining target models that satisfy the quality attributes selected by the software engineer.

This work has been defined taking into account the needs of Software Product Line Development in the domain of Safety-Critical Embedded Systems. In this domain, some quality attributes must be ensured in the final product and for achieving this, both the variability and the functional requirements must be related with quality attributes to ensure that every possible product derived from the product line fulfills its quality requirements. However, the concepts and solutions described in this approach can also be applied to any software product line context regardless to the domain, since the Safety-Critical Embedded Systems is one of the most restrictive domains when talking about the quality of the software products.

1.2. Objectives

The main objective of this Master's final work is to introduce quality attributes in early stages of the software product line development processes by means of the definition of a *multi-model* and a *production plan* that integrates the quality attributes as an additional view for describing the extension of the software product line. This approach requires the definition of additional artifacts (a multi-model) that allows us to express the relationships and constraints among the different system views and guide the development of software products.

A multi-model is a collection of models supporting different views of a system, characterized by the existence of relationships among elements of their correspondent metamodels [9]. The multi-model will comprise three main views of the SPL:

- The Functional View for expressing the base architecture of the SPL as well as the different core assets (or components) that are going to be "plugged" into the base architecture. This view should be considered in the different stages of the software product line production process by means of different architectural models (e.g., modular, component-connector).
- The Variability View for expressing the features (user-visible aspects or characteristics of a system) that are common and variable within the system.
- The Quality View where the different quality attributes for the whole software product line and the individual software products as well as the relationships among them are expressed in a hierarchical decomposition.

The production plan describes how the products are produced from the core assets. Core assets should each have an attached process that defines how it will be used in product development. The production plan is essentially a set of these attached processes with the necessary glue. It describes the overall scheme for how these individual processes can be fit together to build a product. It is, in effect, the reuser's guide to product development within the product line [Clements].

In our approach, the production plan provides support for the use of the multi-model in the development of specific products. It also provides support for the definition of model transformation processes where different alternative transformations appear. The quality attributes that the product must fulfill will be used as the decision factor for selecting among alternatives in those processes.

In order to reach the main objective, the following sub-objectives have been formulated:

1. Study the approaches that try to cover the problems that arise when developing SPLs in the domain of safety-critical embedded systems, paying special attention to approaches trying to apply multi-view or multi-modeling techniques, or that which introduces quality in model transformation processes.
2. Study the different standards that can be used for expressing each one of the different system views.
3. Define the system views and identify the relationships among the elements of the different system views.
4. Define a multi-model that is able to express those relationships. Populate the multi-model with a set of quality attributes that has been identified as relevant for the domain of safety-critical embedded systems.
5. Analyze the alternative transformations, which can appear in the PIM-To-PIM, and PIM-To-PSM model transformations that occur within of the two phases of the SPL development and its possible impact on the quality attributes.
6. Define a set of artifacts (mainly composed of model and metamodels) and a process, which allows the definition, and execution of model transformations in which the selection of alternative model transformations is done based on quality attributes.
7. Define how to integrate the quality view in the different activities of a production plan. Establish which are the activities that affect a single core asset and which ones are transversal to the core assets (affect the different core assets or should be taken into account after the product has been configured).
8. Illustrate the feasibility of the approach in case study in the domain of safety-critical systems for the automotive industry.

1.3. Research Environment

This research has been developed within the Software Engineering and Information Systems (ISSI) research group and contributes to the following research projects:

- CALIMO Project: Integration of quality in Model Driven Software Development (January 2009-Enero 2010). Founded by Generalitat Valenciana, Conselleria de Educación - GV/2009/103.
- MULTIPLE Project: Multi-modeling Approach for Quality-Aware Software Product Lines (October 2009-September 2013). Founded by: Ministerio de Ciencia e Innovación - TIN2009-13838.
- TwinTIDE Project: Towards the Integration of Transectorial IT Design and Evaluation (November 2009-November 2013). Founded by: European Union COST action IC0904.

This research has been granted with a "*Becas de Movilidad para Alumnos de Master*" (Master's Mobility Scholarship) allowing the Master's candidate to develop part of the research in the Leiden Institute of Advanced Computer Science (LIACS) at the Leiden University, The Netherlands. The scholarship was funded by the Ministry of Science and Innovation.

1.4. Document Outline

The remainder of this Master's Thesis is organized in the following chapters:

Chapter 2 describes the state-of-the-art about the research topics related to this work: multi-modeling in the context of SPL, quality assurance in SPL, quality in model transformations, and quality in model-driven software development.

Chapter 3 describes the main concepts related to Software Product Lines and Software Product Lines Engineering.

Chapter 4 provides an overview about the model-driven software development context, focusing on MDA, the OMG proposed architecture, and other related standards. Finally, it introduces the technological space related to this work: the Eclipse environment [67], Eclipse Modeling Framework and QVT.

Chapter 5 describes our multi-model for integrating the quality view in software product line development. The chapter describes the multi-model structure, its views and the relationships that have been identified.

Chapter 6 describes our approach for dealing with alternative model transformations. This includes the description of an architecture for supporting quality-driven model transformation and a section that describes how those transformations should be defined.

Chapter 7 describes a production plan that gives support to both the core asset development, and the product development phases.

Chapter 8 describes a case study in the automotive domain (a Car Control System). The case study is intended to illustrate the feasibility of our proposal and was developed by applying the process presented in the previous chapters.

Finally, Chapter 9 concludes with the main contributions of the work. It also introduces future research work and presents the related publications.

Chapter 2. Related Work

This chapter describes related research addressing the integration of quality as an additional view in both the SPL development and model transformation processes.

In Section 0 we analyze and discuss the different works in the field of applying multi-modeling techniques for introducing the quality perspective in software product line development.

In Section 2.2 we analyze and discuss the research works addressing the introduction of quality attributes as a decision factor in model transformation processes.

2.1. Multi-modeling Techniques for Introducing Non Functional Requirements in SPL Development

In this section we analyze some proposals that try to introduce the quality view in software product line development.

Zhang et al. (2003) [68] define an approach based on the Bayesian Belief Network (BBN) to quality prediction and assessment for a software product line. The BBN is used to model the impact of design decisions on system quality attributes. The FODA-based feature model is used to capture functional requirements and the BBN model to capture the impact of functional variants on quality attributes. The BBN method requires building another directed graph whose nodes are features and NFR. The direction of an edge is always from a feature to one or more NFR to denote the influence of a feature upon a quality attribute. Based on expert knowledge, each QA node in the graph will be assigned a numerical value, which represents the conditional probability of that node, given the realization of the parent feature nodes.

González Baixauli et al (2004) [27] propose a Goal-based model and the use of goal-oriented analysis in software product lines. Goal-oriented requirement engineering is an approach that deals with quality attributes or non-functional requirements in single systems. Two sub-models are proposed: A functional goal model and a softgoal model. Quality attributes are represented as soft-goals and the operation of those quality attributes is encoded in the functional goal sub-model as tasks. Priorities are given to each softgoal on a percentile scale to perform the analysis. And correlations are used to represent the links among functional goals and softgoals. Correlation links have different influence labels (--,-,?,+,++). Those qualitative labels are converted to quantitative values: one value for satisfiability and another for deniability.

Benavides et al (2005) [6] define a feature model's extension with so-called extra-functional features representing non-functional features. Proposes a notation that extends feature models with attributes, characteristics of a feature that can be measured such as availability, cost, latency, bandwidth and relations among attributes. Every feature may have one or more attribute relations taking a range of values in either discrete or continuous domains. It also provides automatic reasoning on those extended feature models using CSP (Constraint Satisfaction Problems).

Jarzabek et al (2006) [38] propose the Feature-Softgoal Interdependency Graph (F-SIG), which provides a framework to record design rationale by expressing dependencies between variable features and quality attributes. They propose a new graph: F-SIG, a union of a feature model and a SIG (Softgoal interdependency graph). In F-SIG explicit and implicit contributions from features to quality attributes are modeled. To express the degree of influence, correlations may also have a label (break: --, hurt: -, unknown: ?, Help: +, Make: ++).

Etxebarria et al (2008) [20] proposes an approach extending the features model consisting on a features model that merges functional requirements and quality attributes, and allows expressing variability of quality attributes (define quality attributes that can be mandatory for each product of the product line, the levels that those quality attributes can achieve and the impact among functional variants and quality attributes).

Raaitikainen et al (2008) [60] define the Svamp approach to model functional and non-functional variability at the architectural level of the SPL. This is achieved by defining a multi-model with three main views: a Kumbaug model to represent the functional and structural variability in the architecture; a quality model to specify the quality attributes and a quality variability view for expressing the variability in these quality attributes.

Tawhid and Petriu (2011) [66] define an approach that introduce a multi-model that includes the quality perspective to a multi-view UML model representing the core family assets of a SPL, called the SPL model. This quality perspective is achieved by annotating the UML using the MARTE profile. They apply ATL transformations to obtain first the UML model of a specific product with the MARTE annotations. A second transformation generates the performance model for analyzing its performance.

In Summary, The majority of the approaches consider only the variability view as the way of expressing the functionality of an SPL, establishing relationships between quality and variability view. In those approaches the quality is not addressed at core asset so we cannot define relationships between quality attributes of the core assets and quality attributes of the product. Only the approach Tawhid and Petriu (2011) considers the relationships with core assets, however only allows the definition of quality attributes through MARTE annotations, which means that only simple relationships and with a reduced set of performance attributes can be addressed. Only the work from Benavides et al (2005) has the purpose of describing the product line for analyzing and also for providing guidance during product configuration, obtaining the product that meet some constraints. The FAMA tool based in this approach is quite useful for analyzing feature models (i.e. if a given features model annotated with quality attributes and constraint has valid configurations etc.)

2.2. Quality in model transformation Processes

In this section we analyze some proposals that deal with the quality of model transformations from the perspective of a quality attribute

Zou and Kontogiannis (2003) [69] proposed a quality-driven reengineering framework for object-oriented migration. Analysis tools, transformation rules, and non-functional requirements for the target migration systems characterize this framework. During the migration process, the source-code transformation rules are associated with quality features of the target system (i.e., coupling and cohesion). This approach was applied to transform a set of GNU AVL libraries into an UML class diagram.

Röttger and Zschaler (2004) [61] proposed an approach for refining non-functional requirements based on the definition of context models and their transformations. This approach has been defined in a software development process that separates the roles of the measurement designer and the application designer. It is the measurement designer's responsibility to specify measurements, context models and transformations among these models. Then, the application designer can apply the transformations when developing a system. They defined a XML-based language for the specification of transformations between abstract and concrete context models. The transformations used the response time quality attribute

Merilinna (2005) [49] proposed a tool for quality-driven model transformations for software architectures. Two types of quality attributes are considered: attributes related to software execution (e.g., performance, availability, reliability) and attributes related to software evolution (e.g., maintenance, modifiability, reusability). The transformations are described according to MDA and a proprietary transformation rule language. The approach only considers horizontal transformations (PIM-to-PIM transformations).

Kurtev (2005) [43] proposed a formal technique for the definition of transformation spaces that support the analysis of alternative transformations for a given source model. This technique provides operations for the selection and reduction of transformation spaces based on certain desirable quality properties of the resulting target model. Specifically, this approach deals with the adaptability of model transformations. To generate the transformation space, the process takes a source model and its metamodel, the target metamodel, and the quality properties as input. The proposal has been applied to a set of transformations to obtain XML schemas from UML class diagrams.

Markovic and Baar (2005) [44] defined a set of transformation rules for the refactoring of UML class diagrams. The rules have been defined using the Query/View/Transformation (QVT) standard of OMG (OMG, 2005). The refactoring is applied to UML class diagrams containing annotated OCL constraints that are preserved when the transformations are applied. Therefore, the syntactical correctness of the target model is preserved.

Similar to this proposal, Ivkovic and Kontogiannis (2006) [37] presented an approach for the refactoring of software architectures using model transformations and semantic annotations. In this approach, the architectural view of a software system is represented as a UML profile with its corresponding stereotypes. Then, the instantiated architectural models are annotated using elements of the refactoring context, including soft goals, metrics, and constraints. Finally, the actions that are most advisable for a refactoring context are applied after being selected from a set of possible refactorings. The proposal has been applied to a case study to demonstrate that the refactoring transformations improve the maintenance, performance and the security of a software system.

Kerhervé et al. (2006) [40] proposed a general framework for quality-driven delivery of distributed multimedia systems. The framework focuses on Quality of Services (QoS) information modeling and transformations. The transformations between models express the relationships among the concepts of the different quality information models. These relationships are defined in quality dimensions and are used to transform instances of a source model to a target model. Different types of transformations are applied to different layers and services: vertical transformations are applied to transform information between the different layers (user, service, system, and resource), and horizontal transformation is applied to interchange information between services of the same layer.

Maswar et al. (2007) [45] define an approach to refactoring software architectures using model transformations. The approach is based on applying architectural patterns to refine architecture elements such that the overall system quality properties (e.g. reliability, performance, etc.) are improved while the system's external behavior is preserved. The process consist on measure the levels of some quality attributes, and if those levels are below the requirements, then apply a transformation and reevaluate. This process continues until the levels meet the requirements.

Abrahão et al. (2008) [1] had proposed an approach to drive the model transformations based on quality attributes. This approach has been empirically validated in a specific set of transformation rules in order to obtain UML Class diagrams starting from a requirements model. The main objective of the experiments was obtaining empirical evidence about the appropriate rule selection and how this selection can improve the understandability of the obtained UML Class Diagrams. The transformation rules where defined using MOMENT platform. In this work, the authors present a generic architecture for quality-driven model transformations. Using a controlled experiment empirically validates that alternative transformations can affect to the quality of the output artifacts and uses the information gathered during these controlled experiments as additional inputs of the transformation process. This information will feed the transformation process with the criteria to choose the alternative transformation that maximize the selected quality attribute.

Finally Drago et al (2011) [19] presents and approach for searching the solution space in model transformation processes, where design alternatives exist, trying to find solution that satisfies some quality attributes. The work is an extends a previous work, in this case, the authors had extend QVT-Operational adding the variation point (alternative transformations exist) and variant (one alternative transformation) concepts and the ability of evaluating quality attributes through the operationalization of metrics. The output of the transformation process is a set of models that fits the quality attributes, and the user must decide which one satisfies his needs.

In summary, some proposals focus on defining horizontal transformations for model refactoring (Merilinna 2005) (Markovic & Baar 2005) (Ivkovic & Kontogiannis 2006) (Maswar et al. 2007). Other proposals are aimed at providing vertical transformations for model refinement (Rottger & Zschaler, 2004), synthesis (Kerhervé et al., 2006) (Kurtev, 2005), or reverse engineering (Zou &Kontogiannis, 2003). Of these studies, only the one by Kurtev (2005) presents a more systematic approach for selecting alternative transformations according to a given quality attribute.

All these approaches propose quality criteria that can be used to drive the transformations, but very few of these approaches (Kurtev, 2005) (Markovic & Baar, 2005) (Maswar, 2007) (Drago 2011) illustrate them by means of practical examples. With the exception of (Markovic, Baar ,2005), (Kurtev, 2005) and (Maswar 2007) the transformations are poorly defined. Therefore, more systematic approaches to ensure quality in MDA processes are needed. Another weakness of these proposals is that they are not empirically validated. The practical applicability of model transformations is reported based on the intuition of the researcher.

2.3. Discussion

From a general interpretation of the state of the art of the interrelated field of software product line development, quality assurance and evaluation and MDSD, the conclusions that influence our work are the following:

- Most of the works trying to introduce quality in SPL development make a partial coverage of the development process focusing only in the description of the impact that the features have over quality attributes. Those approaches do not address the impact that the quality of core assets has over the final product.

- In general the works dealing with quality attributes in SPL have the purpose of analyzing the properties of the products rather than trying to offer a integrated solution for obtaining the final products, or at least models that describe the final product and that are the input of other model transformation or code generation processes.
- Works covering the introduction of quality attributes as decision factor when selecting among alternatives very often are quite specific, focusing only in a concrete transformation or trying to improve one quality attribute.

The global state of the art in the field motivates our work to propose an approach introducing the quality as an additional active view in SPL. In addition the revision of the different research works has guided us regarding the concepts and notations to be used for modeling that view and for introducing the quality attributes as decision factor for selecting among design alternatives in SPL development.

Chapter 3. Software Product Lines Engineering

The production of quality software, on time, and within budget, remains an open problem of Software Engineering that has been addressed from different approaches. An industrial approach to this problem is to use Software Product Lines (SPL). Software product lines engineering (SPLE) is a software engineering paradigm institutionalizing reuse throughout the software lifecycle.

This chapter describes the Software Product Line Engineering paradigm, its main activities, the inputs and outputs of each activity, production plans and variation mechanisms.

The chapter is organized as follows: In Section 3.1 we introduce the historical perspective of the Product Line paradigm. In Section 3.2 we describe the Software Product Line Engineering paradigm, its main activity and its variation mechanisms for SPL development.

3.1. Historical Perspective

Until the days of industrial revolution the production of goods was based on the craftsmanship and produced for individual customers. This production method increases dramatically the cost of the final product due to the time consumed and the specialization needed in each process. After the first industrial revolution, with the introduction of production machinery and the batch production, the production of series of products became more and more cheap. By and by the number of people who could afford those products increased, and with this, the number of goods produced.

The solution to this problem was the *assembly line* introduced by Eli Whitney, and its adoption as *production line* by Ford Motor Company between 1908 and 1915 for its new Ford Model T. These new production techniques enabled mass production much more cheaply than individual handcrafted product creation. However the production line reduced the customization possibilities. People were content with standardized mass products for a while – but not all people want the same kind of product for any purpose. If we return to the automotive industry, it's easy to realize that different customers need different cars that fit their needs (familiar, sportive, and so on). The market needs introduced the need for individual customized products. This was the beginning of *mass customization*. Mass customization is the large-scale production of goods tailored to individual customers' needs. This can be achieved in many different ways, for example by reusing parts by combination, in order to obtain different products in close accordance with customers' wishes. Customization under the customer perspective means the ability of having individualized products. Customization under the company perspective means higher technological investments, which leads to higher prices for the individual products and/or to lower profit margins for the company. Both effects are undesirable. The solution adopted in the automotive industry was the introduction of *common platforms* for their different types of cars by planning beforehand which parts will be used in different car types. The idea is to have an skeleton where later more parts will be added, obtaining different configurations of products.

Software Industry suffered of the same problems than any other production industry. It requires the ability of producing highly customized products, which cannot be created using mass production. The combination of mass customization and a common platform allows us to reuse a common base of technology and, at the same time, to bring out products in close accordance with customers' wishes. The systematic combination of mass customization and the use of a common platform for the development of software-intensive systems and software products is the key of the software development paradigm software product line engineering [56].

3.2. Software Product Line Engineering

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. Software product lines engineering (SPLE) is a software engineering paradigm institutionalizing reuse throughout the software lifecycle.

Each product is formed by taking applicable components from the base of common assets, tailoring them as necessary through preplanned variation mechanisms such as parameterization or inheritance, adding new components that may be necessary and assembling the collection according to the rules of a common, product line wide architecture. Building a new product (system) becomes more a matter of assembly or generation than one of creation: the predominant activity is integration rather than programming. For each software product line there is a predefined guide or plan that specifies the exact product-building approach.

3.2.1. SPL Engineering Essential activities:

The development of Product Lines involves three main activities: Core Asset Development, Product Development and Management. Figure 1 shows the schemas of the 3 main activities of the SPLE, the three activities are linked together, in perpetual motion; they can occur in any order and are highly iterative. Core assets and product development have a strong connection since the development of new products can refresh the core assets repository and new core assets or modifications in a given asset may allow the SPL the production of new products. The diagram is neutral in regard to which activity takes part first. It depends on the nature of the organization and the production strategy. In some cases, the starting point is a set of products, mined for generic assets which are migrated into the product line core asset inventory. In other cases the core assets may be developed or procured for later use in the product development activity, as will be stated in section 3.2.1.1.

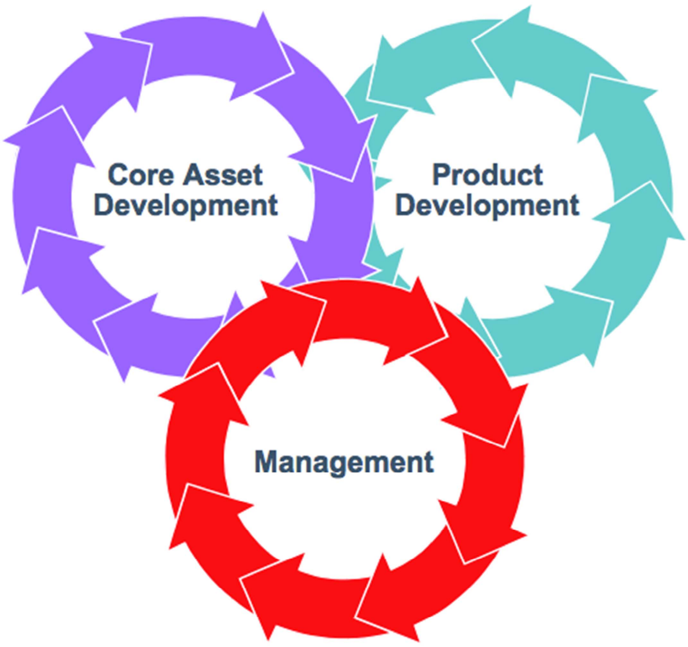


Figure 1. Main activities in SPL Engineering

3.2.1.1. Core Asset Development:

The goal of the core asset development activity is to establish a production capability for products. There are three main outcomes of the core asset development activity. Figure 2 shows the inputs and outputs of the core asset development activity and its main schema.

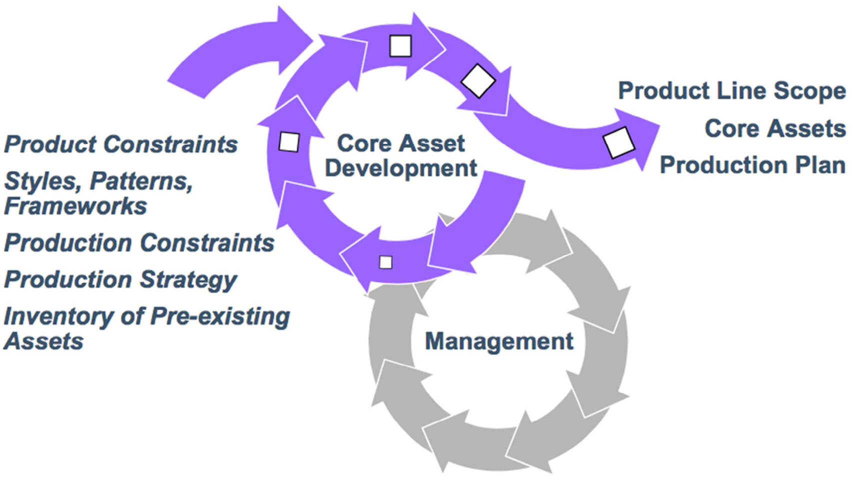


Figure 2. Core Asset Development

- **Product line scope:** The product line scope is a description of the products that will constitute the product line or that the product line is capable of including. At its simplest, scope may consist of an enumerated list of product names. Typically, this description is cast in terms of the things that the products all have in common and the ways in which they vary from one to another. These might include features or operations they provide, performance or other quality attributes they exhibit, platforms on which they run and so on. The scope of the product line may vary as the market evolves or changes or as the organizations plans change. Evolving the scope is the starting point of evolving the product line itself.
- **Core assets:** Core assets are the basis for production of products in the product line. Those assets may include a base architecture that the products in the product line will share as well as software components that are developed for systematic reuse across the product line. Test plans, test cases and all manner of design documentation, Requirements specification and domain models, are considered also as core assets under the SPL perspective. Commercial off-the-self (COTS) components, if adopted, also constitute core assets. Each core asset has associated with it an *attached process* describing the way it will be used in the development of products.

  Among those core assets, the architecture warrants special treatment. Product Line Architecture is a software architecture that will satisfy the need of the product line in general and the individual products in particular by explicitly admitting a set of variation points required to support the spectrum of products within the scope. The product line architecture plays a special role among the other core assets. It specifies the structure of the products in the product lines and provides interface specifications for the components that will be in the asset base. Producing a product line architecture requires the product line scope; a knowledge of relevant styles, patterns and frameworks; and any available inventory of pre-existing assets.

- **Production plan:** a production plan describes how a product is derived from the core assets. The production plan orchestrates the core assets' attached processes for deriving a product. It describes the overall scheme for how these processes can be fitted together to build a product. In fact is the guideline which defines the way of reusing to product development within the product line. The production plan depends on the nature that the core assets can be tailored when deriving a product. There different variation mechanisms that allow core assets to be parameterized will be discussed in section 3.2.2. Figure 3 shows an schema of the attached processes associated to each asset and its combination in the production plan for a specific product.

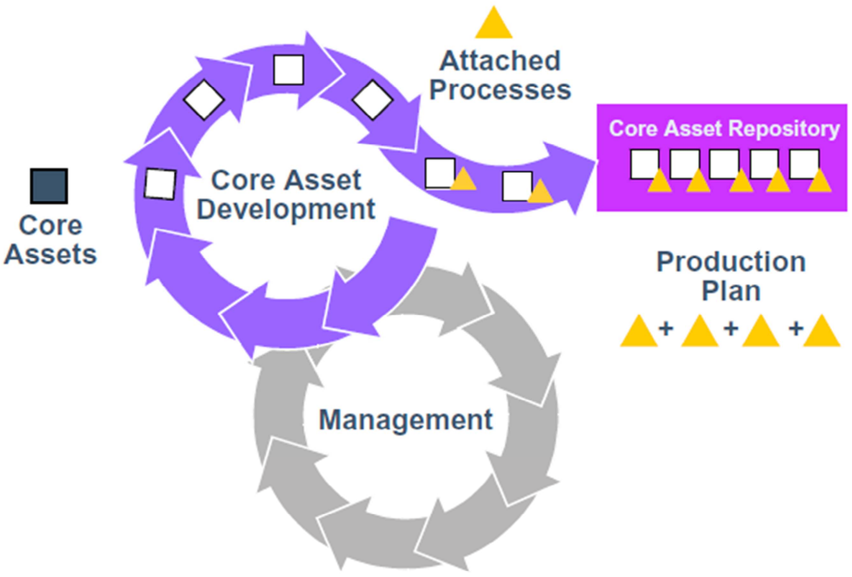


Figure 3. Attached processes and production plan

The main inputs for the core asset development are five:

- **Product constraints:** Commonalities and variation that constitute the product line, behavioral feature, future improvements and technology forecasts, standards to accomplish, performance limits, external systems to interface, physical constraints, quality requirements, and so on.

- **Styles and patterns:** An architectural pattern in software, also known as an architectural style, is analogous to an architectural style in buildings, such as Romanic, Gothic or Greek-Revival [4]. It consists on a set of key features and rules for combining them. Architectures are often built by applying patterns and styles that solve specific problems [63]. An architectural style defines a family of s uch systems in terms of a pattern or structural organization. Moreover, each architectural style has its own properties; including how well suited each style is for achieving specific quality attributes. Design patterns play a similar role at design level, with finer granularity [14]. An architectural pattern expresses a fundamental structural organization schema for software systems. It describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. Patterns are more detailed or less abstract than styles [11].
- **Production Constraints:** Standards that must accomplish, infrastructures on which the products must be built, time to market requirements and so on.
- **Production strategy:** the production strategy specifies how to build the core assets. There are two major strategies: building assets and combining them for deriving products (top-down) or starting with a set of products and generalizing their components to produce the assets (bottom-up). The production strategy will constrain the architecture and the product line itself, how it evolves and grows.
- **Inventory of preexisting assets:** The organization can utilize (reuse) existing libraries, frameworks, algorithms, tools and component. Technical management processes, funding models, training resources can also be adapted for the product line. The inventory includes all existing assets. Then its appropriateness must be analyzed in order to decide whether or not can be reused.

3.2.1.2. Product Development

The product development activity depends on the three outputs of the core asset development activity: the product line scope, the core assets, and the production plan (plus the requirements for individual products).

A software product line is a set of related products but how they are built can vary greatly depending on the assets, the production plan and the organizational context. Figure 4 shows the schema of the product development activity, the different SPL inputs and the output, a specific derived product.

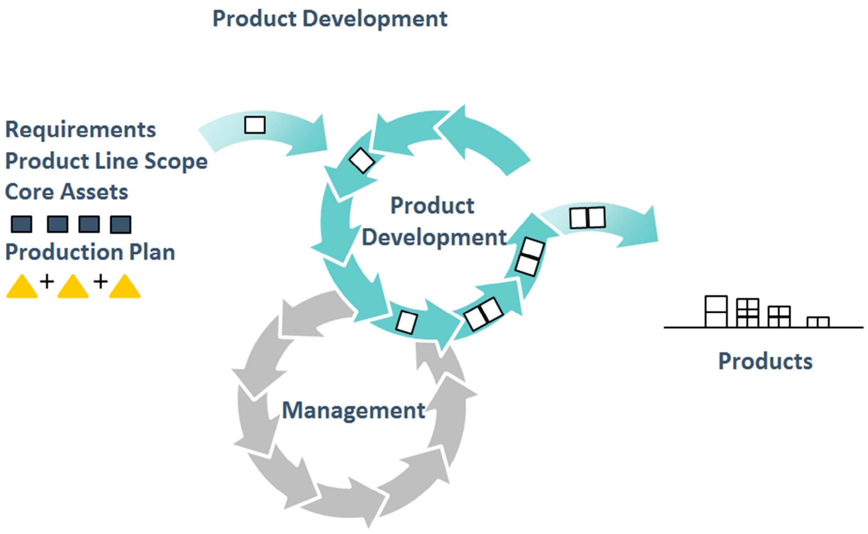


Figure 4. Product Development

3.2.1.3. Management

The different activities must be given resources, coordinated and supervised. Management activity can be divided into technical (project) and organizational levels. Technical management takes care of the core asset development and the product development activities by ensuring that the groups that build core assets and the groups that deploy the products are following the right procedures and track the process in the right way. Organizational management takes care of the resource allocation in the different activities. Organizational management also orchestrates the technical activities and iterations between essential activities of core asset development and product development. Organization management mitigates the riks that threaten the success of the product line.

Both technical and organizational management also contribute to the core asset base by making available for reuse those management artifacts used in developing products in the product line.

3.2.2. Variation Mechanisms

As stated before one of the key concepts in SLPE is variability. The variability in a SPL is not restricted to core asset combination; there are different mechanisms for achieving this variability. A. Nolan enumerates a list of variability mechanisms used in Rolls-Royce for supporting variability in Safety Critical Embedded Systems [57]:

- **Calibrated (Constant Data):** Component's behavior is altered by manipulating "constant" data values to fine-tune the performance of the component to match the domain. This is NOT used to switch in/out functionality, only to shape the component's algorithmic response.
- **Plug Replaced:** Multiple components are produced to the same interface specification to allow large-scale functional replacement.
- **Auto-coded:** Component has a defined interface but its behavior is specified by the project using a graphical programing language, and the component implementation is auto generated.
- **Composed:** Component's behavior can be varied by the inclusion/exclusion of code fragments and operations. The PL asset contains all permissible variations and mechanisms for a project to select and compose the required behavior.
- **Generated (PL Generator):** The PL assets are a restricted definition language (Domain Specific Language) and a code generator which can produce components for a specific purpose. Projects define the required behavior in the restricted language and the component is auto-generated. This is a more specific and restricted for of the "Auto-code" shown above – here the languages are typically mucho more specialized and defined by the PL organization rather than being a *Commercial off-the-shelf* (COTS) auto-code product.

Chapter 4. Model Driven Engineering and Technological spaces

Modeling is not a new idea in software development, it is present in software development processes since many years, and it is used to document software's inner structure. Those models were seen as a merely documents that must be fulfilled during the project lifecycle or as reverse engineering tools for source code visualization. Sometimes models are informal, meaning that they cannot be machine-processed. Programmers use them as guidelines and specifications, but not as something that directly contributes to production. Consequently many view them as peripheral to the production process. Model Driven Engineering is a discipline that relies on models not only as documentation artifacts but also as conductors of the software development process. The effort resides on the modeling rather than on programing, raising the level of abstraction.

This chapter describes all the concepts, standards and tools that are directly related to the work proposed along the work. Specifically, we present the Model Driven Engineering and Model Driven Architecture paradigms, and the technology related to the work.

The chapter is organized as follows: In Section 4.1 we describe the Model Driven Engineering discipline. In Section 4.2 we describe Model Driven Architecture standard. Finally, in Section 4.3 we describe the technological spaces and tools directly related to the work proposed along the work.

4.1. Model Driven Engineering

Model Driven Engineering (MDE) is a discipline of Software Engineering that relies on models as first class artifacts and that aims to develop, maintain and evolve software by means of model transformations. The development process under MDE is called Model Driven Software Development (MDSD).

MDE offers a more effective approach; models are active parts of the software development process. Models are abstract and formal at the same time. Abstractness does not stand for vagueness here, but for compactness and a reduction to essence. MDE models have the exact meaning of program code, in the sense that the bulk of the final implementation, not just class and method skeletons can be generated from them [64]. In this case models are no longer only documentation but parts of the software, constituting a decisive factor in increasing both the speed and quality of software development.

A Model-Driven approach requires languages for model specification, transformation definition and metamodel description. MDE proposes the use of model transformations in order to transform one model into another, and also to produce the final product.

MDE is embraced by various organizations and companies including OMG, IBM and Microsoft.

There are five things an MDE supporting infrastructure must define:

- Concepts available for creating models and clear rules which govern their use.
- The notation to use in depicting models.
- It has to be clear, how the model's elements represent real-world elements and software artifacts.
- Concepts to facilitate dynamic user extensions to model concepts, model notation, and the models created from them.
- Concepts to facilitate the interchange of models concepts and notation, and the models created from concepts to facilitate user-defined mappings from models to other artifacts.

4.2. Model Driven Architecture

Model Driven Architecture (MDA), as stated before, is included in the MDE definition. The MDA standard from the Object Management Group (OMG) is just a specific incarnation of the Model Driven Engineering.

MDA is a young standard established by the OMG. The OMG was founded in 1989 and is an open consortium currently of about 800 companies worldwide. The OMG creates manufacturer-independent specifications to improve the interoperability and portability of software systems.

MDA is about using modeling languages as programming languages rather than merely as design languages. Programming with modeling languages can improve the quality and the speed of software development.

The objective of MDA is to decouple the way that application systems are defined from the technology they run on [47].

The MDA starts with the well-known and long established idea of separating the operational specification of a system from the details of the way that system uses the capabilities of its platform.

Software-platform independence is analogous to hardware-platform independence. A hardware-platform independent language, such as C or Java, enables the writing of a specification that can execute on a variety of hardware platforms with no change. In the same way a software-platform-independent language enables the writing of a specification that can execute on a variety of software platforms, or software architecture designs, with no change. So, a software-platform-independent specification could be mapped to a multiprocessor / multitasking CORBA environment, or a client-server relational database environment, with no change to the model.

In general, the data organization and processing implied by a conceptual model may not be the same as the organization of the data and processing in implementation. If we consider two concepts, those of "customer" and "account," modeling them as classes using the UML suggests that the software solution should be expressed in terms of software classes named Customer and Account. However, there are many possible software designs that can meet these requirements, many of which are not even object-oriented. Between concept and implementation, an attribute may become a reference; a class may be divided into sets of object instances according to some sorting criteria; classes may be merged or split; state-charts may be flattened, merged, or separated; and so on. A modeling language that enables such mappings is software-platform independent.

Raising the level of abstraction changes the platform on which each layer of abstractions depends. Model-based development relies on the construction of models that are independent of their software platforms, which include client-server relational database environments, and the very structure of the final code [48].

MDA provides an approach for, and enables tools to be provided for:

- specifying a system independently of the platform that supports it,
- specifying platforms,
- choosing a particular platform for the system,
- And finally transforming the system specification into one for a particular platform.

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns.

The MDA development life cycle, does not look very different from the traditional life cycle. The same phases are identified. One of the major differences lies on the nature of the artifacts that are created during the development process. The artifacts are formal models, i.e., models that can be understood by computers [40].

As conclusion MDA is an approach to system development, which increases the power of models in that work. It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification.

4.2.1. MDA Viewpoints

A *viewpoint* on a system is a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system. The Model-Driven Architecture specifies three viewpoints on a system, a computation independent viewpoint, a platform independent viewpoint and a platform specific viewpoint [51].

4.2.1.1. Computation Independent Viewpoint

The computation independent viewpoint focuses on the environment of the system, and the requirements for the system; the details of the structure and processing are hidden or as yet undetermined.

4.2.1.2. Platform Independent Viewpoint

The platform independent viewpoint focuses on the operation of a system while hiding the details necessary for a particular platform. A platform independent view shows that part of the complete specification that does not change from one platform to another. A platform independent view may use a general purpose modeling language, or a language specific to the area in which the system will be used.

4.2.1.3. Platform Specific Viewpoint

The platform specific viewpoint combines the platform independent viewpoint with an additional focus on the detail of the use of a specific platform by a system.

4.2.2. Models in MDA

The first class artifacts in Model Driven Engineering are models. The first thing to do is to define what a model is. A model is a simplification (or an abstract description) of a part of the world named system, built with an intended goal in mind. A model should be easier to use and understand than the original system, should be able to answer questions about the system. The answers provided by the model should be exactly the same as those given by the system itself [62], [7]. Models may consist of a set of elements with a graphical and/or textual representation. While this serves as a starting point, Kleppe et al. [40] gives a definition even more directed to MDSD A model is a description of a (part of) systems written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer.

The idea of MDA is creating different models of a system at different levels of abstraction. Each model represents a given aspect of the system.

According to those definitions, source code is a model too. Source code is a simplified representation of the lower-machine structures and operations that are required to automate the tasks in the real world. Moreover, correct source code is a very useful model since it tells the machine what actions need to be taken to maintain the system's goal.

4.2.3. Kind of models:

MDA standard [51] defines four kinds of models into the lifecycle of Model Driven Development:

4.2.3.1. Computation Independent Model (CIM)

A computation independent model is a view of a system from the computation independent viewpoint, which it focuses on the environment and the requirements of the system; structural or processing details of the system are hidden or as yet undetermined. A CIM does not show details of the structure of systems. A CIM is sometimes called a domain model and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification [51]. Focuses on the Environment of the system and the requirements the user has on the system; the description provides what the system is expected to do.

4.2.3.2. Platform Independent Model (PIM)

The platform independent model focuses on the operation of a system while hiding the details necessary for a particular platform. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type. It is also a representation of business functionality and behavior, undistorted by technology details. It shows the part of the complete specification that does not change from one platform to another.

The objective is to postpone in the development process the creation of models that take into account technological aspects of a platform as much as possible. The main advantage is to be able to react efficiently and with low costs to technology changes.

4.2.3.3. Platform Specific Model

A Platform Specific Model Is a combination of a PIM with additional details of the system's specific platform.

4.2.3.4. Platform Model

Finally platform models are the representation of technical concepts of platform's parts, the services provided by that platform, and for further use in PSM's, concepts which models the use of the platform by the applications.

4.2.4. Metamodels

The word “meta” is Greek and means “above”, therefore the term metamodel can be interpreted as a model describing another model. To understand the term metamodel a simple analogy to languages is drawn. A language consists of words whose combination is constraint by a grammar. If a sentence in a language is seen as one possible model, the definition of its structure, the grammar, can be seen as its metamodel. Earlier it was said that in MDSD a metamodel defines how a model can lookalike, this can be more precisely formulated as: a metamodel defines the constructs and rules usable to create a class of models. This is consistent with the following definitions:

- A metamodel is a model of a set of models [51].
- A metamodel is a model that defines the language for expressing a model [20].

A metamodel is a specification model that describes its models in a certain modeling language. A metamodel says what can be expressed in a valid model of the modeling language.

The interpretation of a metamodel is a mapping of elements of the metamodel to elements of the modeling language. The truth-value of statements in the metamodel can be determined for any model expressed in the modeling language. Since the metamodel is the models specification, a model in the modeling language is valid only if none of these statements are false. Precise metamodels are a prerequisite for performing automated model Transformation and for defining accurate models.

Since a metamodel is also a model, could be expressed in some modeling language. A metamodel for a modeling language could use the same modeling language. The statements in the metamodel are expressed in the same language as is being described by the metamodel. This is called reflexive metamodel.

4.2.5. Meta-Object Facility (MOF)

Meta-Object Facility (MOF) is a standard proposed and defined by OMG [52]for supporting MDA. This standard proposes four levels meta-modeling architecture with four meta-layers as shown in Figure 5. Each meta-layer is the metamodel of the constructs in the layers above. Layers are described as follows:

1. M3 meta-metamodel layer: In this layer resides the meta-metamodel, a language for defining level M2 metamodels. In OMG standard, MOF is the language defined in this layer.
2. M2 Metamodel layer: M2 Metamodels are used for describing M1 Models. OMG’s UML Metamodel will describe UML constructs.
3. M1 Model layer: At these level models is where we define models. A model is an instance of a M2 metamodel. I.e., if in M2 layer

resides UML metamodel in M1 we could have one of its models: Class Diagram, Activity Diagram, Sequence Diagram and so on.

4. M0 Instance layer: in this layer objects of the real world are defined.

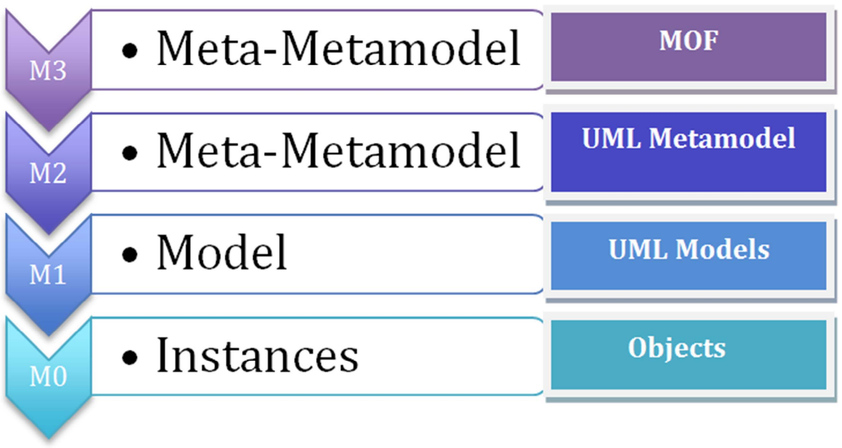


Figure 5. MOF layered architecture

MOF is closed metamodeling architecture. This means that M3 level could be defined with instances of M3 elements. This implies that whit MOF we can define MOF.

In addition MOF provides concepts to define a language:

- Classes, which model MOF metaobjects.
- DataTypes (property), which model needed descriptive data (i.e., primitive types).
- Associations (property), which model binary relationships between metaobjects.
- Packages, which modularize the models.

4.2.6. Model Transformations

Model Transformation: is the process of converting one model to another model of the same system [51]. One model may be transformed to several alternative models that can maintain the semantics but with different syntax. The mappings and relations are defined as specializations of transformations. Figure 6 shows the two different levels in which the transformation is defined (Metamodel level) and executed (Model level).

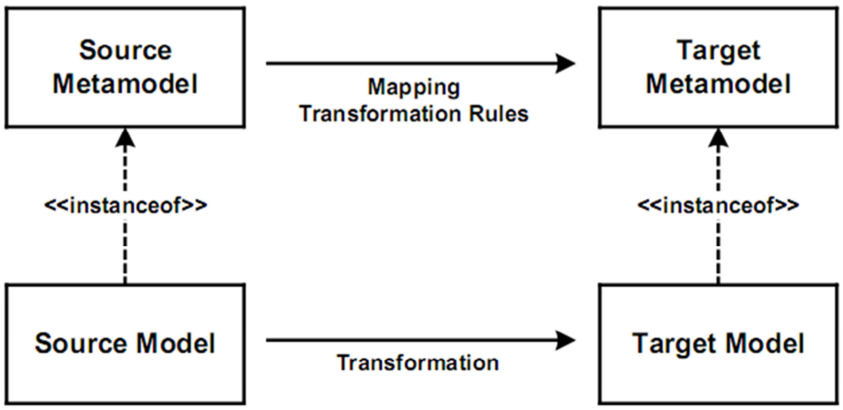


Figure 6. Model Transformation Definition

A transformation definition consists of a collection of transformation rules which are unambiguous specifications of the way that (a part of) one model can be used to create a part of another model. The transformations are defined in terms of the metamodels involved in the transformation process. Transformation, transformation definition and transformation rule can now be defined as follows [40]:

- "A transformation is the automatic generation of a target model from a source model, according to a transformation definition".
- "A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language".
- "A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language" .

The most important characteristic of a transformation it is the fact that a model transformation should maintain the meaning between the source and the target model. At this point it must be said that the meaning of the model can only be preserved as it can be expressed in both source and target model. Part of information should be lost if target language is less expressive than source language.

A mapping is defined as a unidirectional transformation in contrast to a relation that defines a bi-directional transformation.

4.2.7. QVT (Query/View/Transformation)

QVT is a Model Transformation standard [53] defined by the Object Management Group and related to the MOF architecture.

The QVT specification has a hybrid declarative/imperative nature, with the declarative part being split into two-level architecture. The two-level architecture of the declarative part forms the framework for the execution semantics of the imperative part.

4.2.7.1. QVT Two Level Declarative Architecture

The declarative parts of this specification are structured into a two-layer architecture. The layers are:

- A user-friendly Relations metamodel and language that supports complex object pattern matching and object template creation. Traces between model elements involved in a transformation are created implicitly.
- A Core metamodel and language defined using minimal extensions to EMOF and OCL. All trace classes are explicitly defined as MOF models, and trace instance creation and deletion is defined in the same way as the creation and deletion of any other object.

Figure 7 shows the relationships between the QVT metamodels that configure the QVT Architecture.

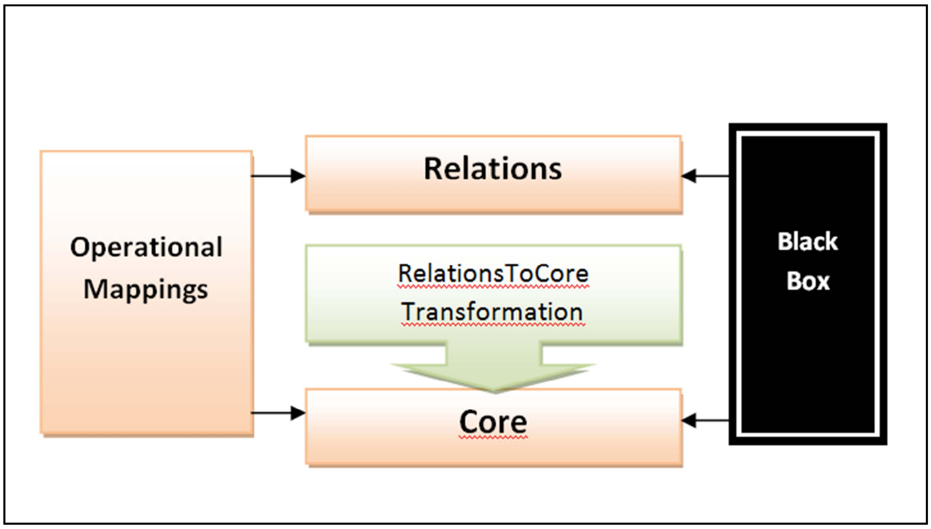


Figure 7. Relationships between QVT Metamodels

4.2.7.2. QVT Relations

The Relations language supports complex object pattern matching, and implicitly creates trace classes and their instances to record what occurred during a transformation execution. Relations can assert that other relations also hold between particular model elements matched by their patterns. The semantics of Relations are defined in a combination of English and first order predicate logic, as well as by a standard transformation for any Relations model to trace models and a Core model with equivalent semantics. It can be used purely as a formal semantics for Relations, or as a way of translating a Relations model to a Core model for execution on an engine implementing the Core semantics.

4.2.7.3. QVT Core

The Core language is a small model/language that only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models. It treats all of the model elements of source, target, and trace models symmetrically. It is equally powerful to the Relations language, and because of its relative simplicity, its semantics can be defined more simply, although transformation descriptions described using the Core are therefore more verbose. In addition, the trace models must be explicitly defined, and are not deduced from the transformation description, as is the case with Relations. The core model may be implemented directly, or simply used as a reference for the semantics of Relations, which are mapped to the Core, using the transformation language itself.

4.2.7.4. QVT Imperative Implementations

In addition to the declarative Relations and Core Languages that embody the same semantics at two different levels of abstraction, there are two mechanisms for invoking imperative implementations of transformations from Relations or Core: one standard language, Operational Mappings, as well as non-standard Black-box MOF Operation implementations. Each relation defines a class that will be instantiated to trace between model elements being transformed, and it has a one-to-one mapping to an Operation signature that the Operational Mapping or Black-box implements.

This language is specified as a standard way of providing imperative implementations, which populate the same trace models as the Relations Language. It provides OCL extensions with side effects that allow a more procedural style, and a concrete syntax that looks familiar to imperative programmers.

4.2.7.5. Operational Mappings

Operational Mappings Language can be used to implement one or more Relations from a Relations specification when it is difficult to provide a purely declarative specification of how a Relation is to be populated. Mappings Operations invoking other Mappings Operations always involves a Relation for the purposes of creating a trace between model elements, but this can be implicit, and an entire transformation can be written in this language in the imperative style. A transformation entirely written using Mapping Operations is called an operational transformation.

4.2.7.6. The QVT Relations Language

In the relations language, a transformation between candidate models is specified as a set of relations that must hold for the transformation to be successful. A candidate model is any model that conforms to a model type, which is a specification of what kind of model elements any conforming model can have, similar to a variable type specifying what kind of values a conforming variable can have in a program. Candidate models are named, and the types of elements they can contain are restricted to those within a set of referenced packages.

In order to illustrate the definitions we will an example extracted from the QVT specification (OMG, 2008) (OMG, 2008) (OMG, 2008) (OMG, 2008) which shows the relations between elements fron the “UML” domain and “RDBMS” domain.

transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS)

In this declaration named “umlRdbms,” there are two typed candidate models: “uml” and “rdbms.” The package named “uml” and the package named “rdbms” declare the SimpleUML and SimpleRDBMS package as its metamodels, respectively. A transformation can be invoked either to check two models for consistency or to modify one model to enforce consistency.

4.2.7.6.1. Transformation Execution direction

A transformation invoked for enforcement is executed in a particular direction by selecting one of the candidate models as the target. The target model may be empty, or may contain existing model elements to be related by the transformation. The execution of the transformation proceeds by first checking whether the relations hold, and for relations for which the check fails, attempting to make the relations hold by creating, deleting, or modifying only the target model, thus enforcing the relationship.

4.2.7.6.2. Relation

Relations in a transformation declare constraints that must be satisfied by the elements of the candidate models. A relation, defined by two or more **domains** and a pair of **when** and **where** predicates, specifies a relationship that must hold between the elements of the candidate models.

4.2.7.6.3. Domain

A domain is a distinguished typed variable that can be matched in a model of a given model type. A domain has a pattern, which can be viewed as a graph of object nodes, their properties and association links originating from an instance of the domain's type.

Alternatively a pattern can be viewed as a set of variables, and a set of constraints that model elements bound to those variables must satisfy to qualify as a valid binding of the pattern. A domain pattern can be considered a template for objects and their properties that must be located, modified, or created in a candidate model to satisfy the relation.

4.2.7.6.4. When and Where clauses

A relation also can be constrained by two sets of predicates, a when clause and a where clause, as shown in the example relation ClassToTable below. The when clause specifies the conditions under which the relationship needs to hold, so the relation ClassToTable needs to hold only when the PackageToSchema relation holds between the package containing the class and the schema containing the table. The where clause specifies the condition that must be satisfied by all model elements participating in the relation, and it may constrain any of the variables in the relation and its domains. Hence, whenever the ClassToTable relation holds, the relation AttributeToColumn must also hold Implementations.

```
relation ClassToTable  /* map each persistent class to a table */
  {
      domain uml c:Class {
        namespace = p:Package {},
        kind='Persistent',
        name=cn
    }
    domain rdbms t:Table {
        schema = s:Schema {},
         name=cn,
         column = cl:Column {
             name=cn+'_tid',
             type='NUMBER'},
         primaryKey = k:PrimaryKey {
              name=cn+'_pk',
              column=cl}
    }
    when {
      PackageToSchema(p, s);
    }
    where {
      AttributeToColumn(c, t);
    }
  }
```

The when and where clauses may contain any arbitrary OCL expressions in addition to the relation invocation expressions. Relation invocations allow complex relations to be composed from simpler relations.

4.2.7.6.5. Top-Level Relations

A transformation contains two kinds of relations: top-level and non-top-level. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level relations are required to hold only when they are invoked directly or transitively from the where clause of another relation.

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS)
{
    top relation PackageToSchema {…}
    top relation ClassToTable {…}
    relation AttributeToColumn {…}
}
```

A top-level relation has the keyword top to distinguish it syntactically. In the example above, PackageToSchema and ClassToTable are top level relations, whereas AttributeToColumn is a non-top-level relation.

4.2.7.6.6. Check and Enforce

Whether or not the relationship may be enforced is determined by the target domain, which may be marked as checkonly or enforced. When a transformation is enforced in the direction of a checkonly domain, it is simply checked to see if there exists a valid match in the relevant model that satisfies the relationship. When a transformation executes in the direction of the model of an enforced domain, if checking fails, the target model is modified so as to satisfy the relationship, i.e., a check-before-enforce semantics.

In the example below, the domain for the “uml” model is marked checkonly and the domain for the rdbms model is marked enforce.

```
relation PackageToSchema  /* map each package to a schema */
  {
    checkonly domain uml p:Package {name=pn}
    enforce domain rdbms s:Schema {name=pn}
  }
```

If we are executing in the direction of uml and there exists a schema in rdbms for which we do not have a corresponding package with same name in uml, it is simply reported as an inconsistency - a package is not created because the “uml” model is not enforced, it is only checked.

However, if we are executing the transformation umlRdbms in the direction of rdbms, then for each package in the uml model the relation first checks if there exists a schema with same name in the rdbms model, and if it does not, a new schema is created in that model with the given name. To consider a variation of the above scenario, if we execute in the direction of rdbms and there is not a corresponding package with the same name in uml, then that schema will be deleted from the rdbms model, thus enforcing consistency in the enforce domain.

These rules apply depending on the target domain only. In this execution scenario, schema deletion will be the outcome even if the uml domain is marked as enforced, because the transformation is being executed in the direction of rdbms, and object creation, modification, and deletion can only take place in the target model for the current execution.

4.2.7.7. QVT Graphical Syntax

Diagrammatic notations have been a key factor in the success of UML, allowing users to specify abstractions of underlying systems in a natural and intuitive way. Therefore this specification contains a diagrammatic syntax to complement the textual syntax of Section 7.13.1. There are two ways in which the diagrammatic syntax is used, as a way of:

- Representing transformations in standard UML class diagrams,
- Representing transformations, domains, and patterns in a new diagram form: transformation diagrams.

The syntax is consistent between its two uses, the first usage representing a subset of the second. A visual notation is suggested to specify transformations. A relationship relates two or more patterns. Each pattern is a collection of objects, links, and values. The structure of a pattern, as specified by objects and links between them, can be expressed using UML object diagrams. Using object diagrams with some extensions to specify patterns within a relation specification is suggested. The notation is introduced through some examples followed by detailed syntax and semantics. [53]. Figure 8 specifies a relation, UML2Rel from UML classes and attributes to relational tables and columns. A new symbol is introduced to represent a transformation. The specifications “uml1:UML” and “r1:RDBMS” on each limb of the transformation specifies that this is a relationship between two typed candidate models “uml1” and “r1” with packages “UML” and “RDBMS” as their respective meta models. The “C” under each limb of the relation symbol specifies that both domains involved in this relation are checkonly.

```
relation UML2Rel {
 checkonly domain uml1 c:Class
{
name = n,
attribute = a:Attribute{name = an}
}
 checkonly domain r1 t:Table
{
name = n,
column = col:Column{name = an}
}
}
```

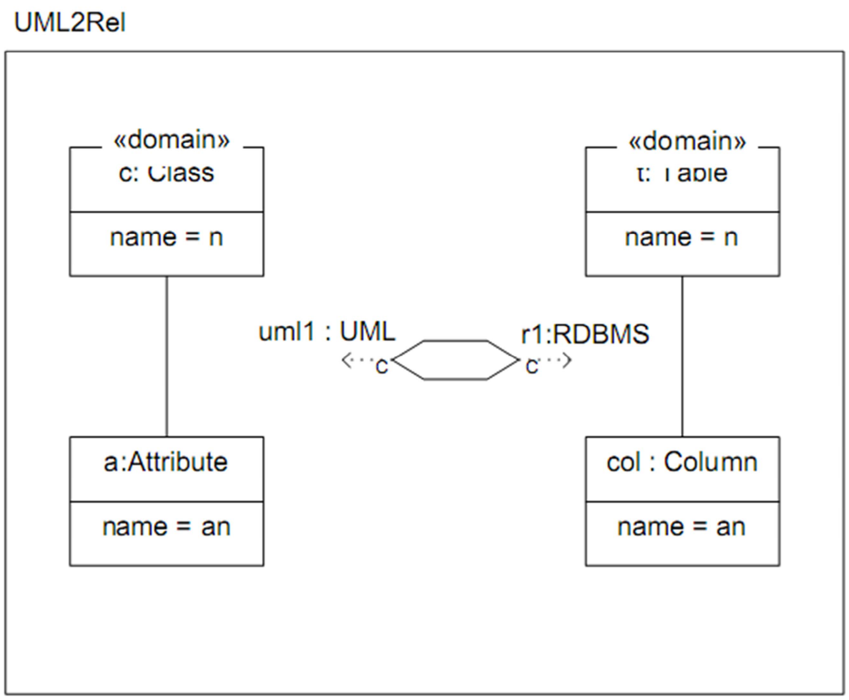


Figure 8. QVT Graphical notation for UML2Rel Class to Relational Table Relation

Table 1 gives a brief description of the visual notation elements.

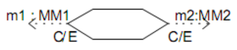
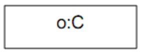
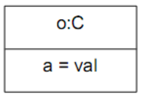
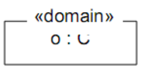
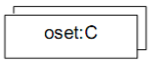
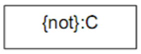
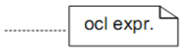
Notation	Description
	A relation between models m1 having MM1 as meta-model and m2 having MM2 as meta-model. The label C/E indicates whether the domain in that direction is checkable or enforceable.
	An object template having type *C* and referred to by the free variable *o*.
	An object template having type *C* and a constraint that the property *a* should take the value *val*. *val* can be an arbitrary ocl expression.
	The domain in a relation.
	*oset* is an object template that matches a set of objects of type *C*.
	A *not* template that matches only when there is no object of type *C* satisfying the constraints associated with it.
	A constraint that may be attached to either a domain or an object template.

Table 1. QVT Diagramatic elements

4.2.7.8. Object Constraint Language (OCL)

Object Constraint Language (OCL) [55]is a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. Note that when the OCL expressions are evaluated, they do not have side effects; i.e. their evaluation cannot alter the state of the corresponding executing system.

OCL expressions can be used to specify operations / actions that, when executed, do alter the state of the system. UML modelers can use OCL to specify application-specific constraints in their models. UML modelers can also use OCL to specify queries on the UML model, which are completely programming language independent.

OCL not only can be applied in UML models but also it can be applied into UML or MOF metamodels since are expressed in UML or a subset of UML. Hereby OCL can be used to restrict metamodel semantics, for example by means of stereotypes or DSL's.

In MDA context, OCL can be used in three ways:

- Precise modeling in MOF M1 level.
- Definition of modeling languages.
- Definition of transformations.

4.3. Technological Spaces

In this section we describe the technological areas related to the work.

4.3.1. Eclipse

The Eclipse community [66]is an open source community whose projects are focused on creating an open development platform comprised of extensible frameworks, tools for creating, deploying and managing software across the lifecycle.

One of his most important projects is Eclipse. Eclipse is an open source project, robust, with multiple features, commercial-quality industry platform for the development of highly integrated tools. Integrated Development Environment (IDE) uses different modules to enhance its functionality; this is an advantage over other monolithic environments where functionality is not configurable.

In short, the nature of this tool makes it an open, extensible IDE for multiple purposes. This work will be of particular interest the Eclipse Modeling Framework (EMF), a framework for managing models and code generation from models described in XMI. EMF is described in detail in the next subsection.

4.3.2. EMF

The EMF project is a modeling framework and code generation facility for building tools and other applications based on structured data models. It starts with a model specification described in XMI. EMF provides tools to produce a set of Java classes for the model. You can generate a set of adapter classes that enable views and edition commands based on the model.

Models can be specified using annotated Java, XML documents, or modeling tools like Rational Rose through which can be imported to EMF. The most important thing is that EMF provides the foundation for establishing interoperability with other tools and applications based on EMF. With regard to the relationship of EMF to OMG and MOF, EMF started out as an implementation of the MOF specification matured from the experience gained in the development of tools by the developers of Eclipse. EMF can be seen as an efficient implementation of joint use of the MOF API. However, to avoid confusion, the EMF metamodel based on the MOF core is called Ecore [66].

4.3.3. Medini QVT

Medini QVT [33] is a QVT transformation engine which implements the OMG's QVT relations specification in a powerful QVT engine. This standard is designed for model-to-model transformations to allow fast development, maintenance and customization of process specific transformation rules.

Highlights among the supported features are:

- Execution of QVT transformations shown the textual concrete syntax of the relations language.
- Editor with code assistant.
- Integrated debugging which allows to run the relations step by step.
- Trace management enabling incremental updates during transformations.
- Key concept enabling incremental updates as well as the transition from manual modeling to automatic transformations.
- Bidirectional transformations.
- Multi-model transformations, multiple source or/and target models.

There are two ways of working with Medini QVT a standalone version, an Eclipse-like standalone executable or an Eclipse Plug-in. In the standalone version EMF editors and the transformation engine are available for defining transformations, designing models and metamodels. The Eclipse Plug-in integrates QVT relations Engine in the Eclipse IDE. Once the plug-in has been activated the QVT transformations are available as any other Eclipse functionality.

The selection of Mediny QVT against other transformation engines for this project is based on:

- Multi-model transformations support.
- Complaint with OMG's QVT Relations standard.

- Bidirectional transformations.
- Eclipse Integration.
- EMF compatible.

Chapter 5. A Multi-model for Software Product Line Development

Through the years it had been demonstrated that variability management is inherent to product line engineering and must be taking into account during software development. However, the variability information must be related with the functional and quality requirements and thus be verified to ensure that every possible product derived from a product line fulfills its functional and quality requirements.

The main objective of this work is to introduce quality attributes in early stages of software product line development. As a means for achieving this goal we need to define the artifacts for collecting the quality requirements and use them during the core asset and product development activities.

This chapter presents a multi-model that integrates the quality view together with the variability and functional views and allows describing the software product line extension.

The chapter is organized as follows: In Section 5.1 we explain the role of the multi-model in the software product line development process. In Section 5.2 we describe the structure of the multi-model, how it has been defined as well as the different views that comprises the multi-model. In Section 5.3 we describe the, the different variability modeling techniques the variability view, followed by the structure of the variability metamodel. In Section 5.4 we describe the quality view, including the structure of the quality metamodel. In Section 5.5 we describe the functional view, the different architectural description languages and an excerpt of the AADL modeling language. Finally, in Section 5.6 we describe the relationships among the different system views and how they have been defined in the context of our multi-model.

5.1. The Multi-Model in the Development Process

Quality attributes, such as response time, accuracy, security, reliability, are properties that affect the system as a whole. Most approaches deal with quality attributes separately from the functional requirements of a system. This means that integration is difficult to achieve and usually is accomplished only at the later stages of the software development process.

Furthermore, software product line engineering approaches emphasizes more on specifying functional aspects of a product line and its variability than in ensuring the quality attributes that the product line and/or the products derived from it must fulfill [18]. Research in the field of software product lines has primarily been focused on analysis, design, and implementation and very few studies have addressed the quality assurance problems and challenges that arise in a reuse context [42]. The selection of different features and the instantiation of functional components in the product development phase will impact on the quality of the product. If we can establish the relationships between features and/or functional components and quality attributes then the quality attributes can be another factor to decide whether or not to select one feature/functional component.

Therefore, what we propose is a multi-model to capture the different system views (functionality, features, quality...) and the relationships among them (e.g., features with different qualities). This will allow us to identify and specify quality attributes that crosscut functional requirements at an early stage of the product line development process. This also allows the parameterization of the production process by means of the multi-model proposed.

The multi-model plays a pivotal role in the SPL's production plan, since it will be used as the main artifact for selecting among the features and the core assets that will be deployed in a specific product configuration. Therefore, the multi-model is going to be used during the two main phases of the SPL development, see Figure 9:

- In the **core asset development phase** by expressing the impact that the features and the functional components have over the quality attributes and the relationships among features and functional components, describing the whole extension of the product line.
- In the **product development phase**, as the artifact that guides the product configuration, allowing with the selection of features and functional components based on the quality attributes that a specific product must fulfill.

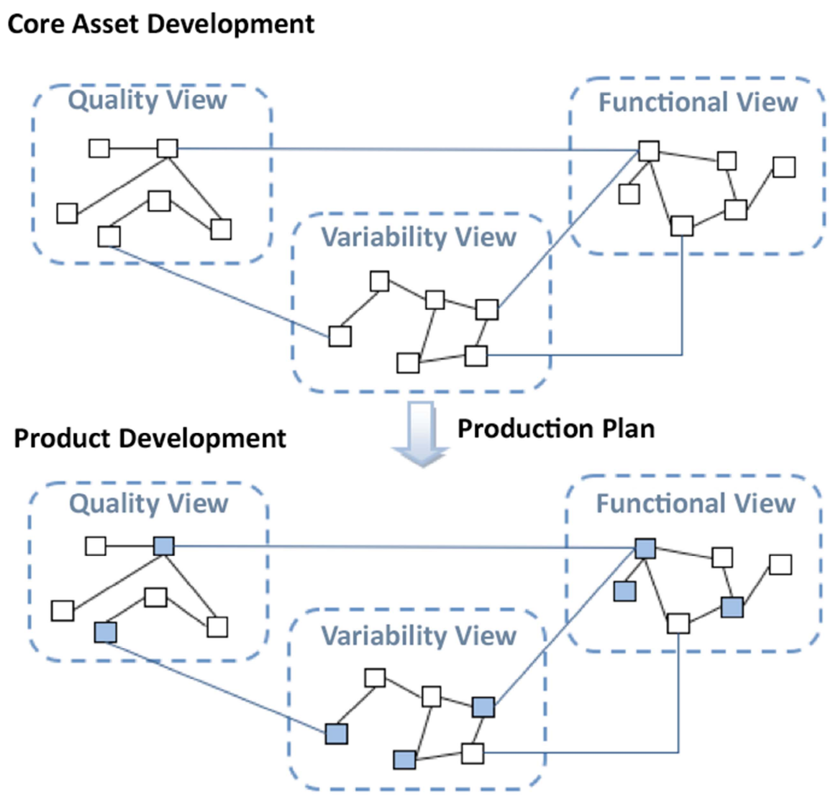


Figure 9. The Multi-model in the SPL development process

At core asset development level, variability management as the basis for LPS development, implies on one hand, to handle the common and optional features of a specific domain, expressed in a features model, and on the other hand, that this variability must be supported by software assets. The specification of variability and functionality (Functional Model) can be managed in separated and independent models. In addition, in order to assure the quality of the product this also should be included in the development process, through a quality model. In this way, we consider (at least) three system views expressed in a multi-model: the system's variability view, the system's functional view and the system's quality view

At the application level, the multi-model represents the mandatory and the selected features from the Features Model together with the elements of the Functional Model and the Quality Model that are affected by these features.

This leads to the parameterization of the production process by means of a multi-model, which is able to capture the different views of the product and the relationships among them. In this scenario we should considered the *intra-model* consistence problems (i.e. the consistence of the variability model) and *inter-model* consistence problems (i.e. the consistence of the relation between the Features Model and the Quality Model) in a more wide and realistic context.

Furthermore, the quality assurance and evaluation processes are, in general, executed independently to the development process. However, following the multi-modeling approach, the quality view participates in the software development process, becoming an active artifact throughout the life cycle. The quality view interacts with the other system views and its relationships drive the production plan of a final product.

5.2. Multi-model Structure

The multi-model structure is established by the definition of what a multi-model is: "a collection of models supporting different views of a system with relationships among them". This leads to a solution where the metamodels that give support to the different views of the system remain unchanged (are exactly the original metamodels). In addition we need to define a new metamodel that gives support to the multi-model containing, basically, the relationships among the meta-classes of the different metamodels defined as external resources. The use of the proxy pattern [23] allows us to define new meta-classes in the multi-model that inherit from the original meta-classes and that can be extended for holding the additional information needed for expressing the impacts and the relationships among the views.

"Design patterns describe problems that occur over and over again in our environment, and then describe the core of the solution to that problem, in such a way that this solution can be used in different scenarios, without ever doing it in the same way twice. Design patterns make it easier to reuse successful design and architectures and help designers choosing alternatives that make a system reusable and avoid alternatives that compromise reusability" [23].

"The Proxy Pattern provides a surrogate or placeholder for another object to control access to it". Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer (i.e. a smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed) [23].

Figure 10 shows the application of the proxy pattern in an excerpt of the multi-model. The meta-class *ComponentType* shown in Figure 10.1 is defined in one of the system views' metamodels (the functional view). This class has its proxy-pattern *EComponentType*[1] meta-class in the multi-model's metamodel (Figure 10.2). The original meta-classes remain unchanged, as it was required by the multi-model definition, and we can add the additional structures for

[1] By convention each proxy meta-class has been named as *E+OriginalClassName.*

supporting the relationships among the elements in the different views as will be described in section 5.6.

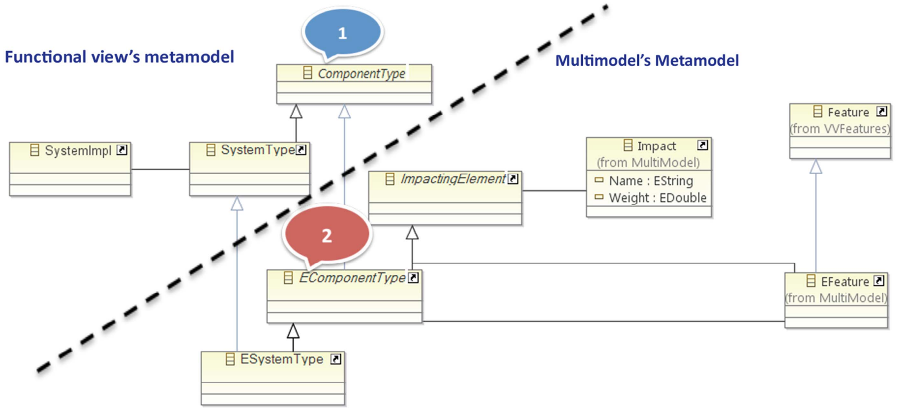


Figure 10. Application of the Proxy Pattern in the Multi-model

Once we have defined what is a multi-model and how it should be defined, we need to define its different views. The multi-model will comprise three views of the SPL:

- The **Functional View** for expressing the base architecture of the SPL as well as the different assets (or components) that are going to be "plugged" into the base architecture.
- The **Variability View** expressing the features (user-visible aspects or characteristics of a system) that are common and variable within the system.
- The **Quality View** of the SPL is where the different quality attributes and the relationships among them are expressed in a hierarchical decomposition. As a starting point this model will contain only the most relevant quality attributes for the safety-critical embedded systems domain.

5.3. Variability View

One of the key aspects that characterize SPLE against other software reuse techniques is how it describes and manages variability. The majority of the approaches addressing this problem are based in Feature-Oriented Domain Analysis (FODA) [38], which is a method (and also a notation) for performing a domain analysis. The feature-oriented concept is based on the emphasis placed by the method on identifying those features a user commonly expects in a domain. In the literature we can find a wide set of definitions for the feature concept within a Software Product Line's context [12]:

- *"A Feature is a logical unit of behavior specified by a set of functional and non-functional requirements" Bosch* [10].
- *"Features are user-visible aspects or characteristics of a system that are organized into a tree of and/or nodes to identify the commonalities and variabilities within the system" Clements et al.* [14].

The latter definition contains an explicit reference to feature models, which is the most common way to express the variability view of a system. Feature Modeling is a technique, which uses a specific visual notation to characterize the variability of product lines by means of diagrams.

5.3.1. Feature models

The purpose of a feature model is to describe the "requirements space" of known solutions of a given problem. The model should encompass as many valid solutions (configurations) as is feasible, to include the fullest range of features and feature values. A feature model as defined in FODA methodology is composed by:

- A feature diagram: a graphical and/or hierarchy of features.
- Composition rules: mutual dependency (Requires) and mutual exclusion (mutex-with) relationships.
- Issues and decisions: record of trade-offs, rationales and justifications.
- System feature catalogue: record of existing system features.

Figure 11 shows a feature diagram using the FODA notation. It is a tree of different features. A small circle above the feature name designates optional features. Alternative features are shown in the diagram as being children of the same parent feature with an arc joining all the options, meaning that one and only one of those features can be chosen. The remaining features with no special notation are all mandatory. The lines drawn between a child and a parent mean that the child feature needs the parent to be present. If an optional parent feature were not marked as valid all its children would be unreachable for that specific configuration, as happens in the case of *tiledColumns* or *tiledArbitrary* when *overlappedLayout* is marked as valid.

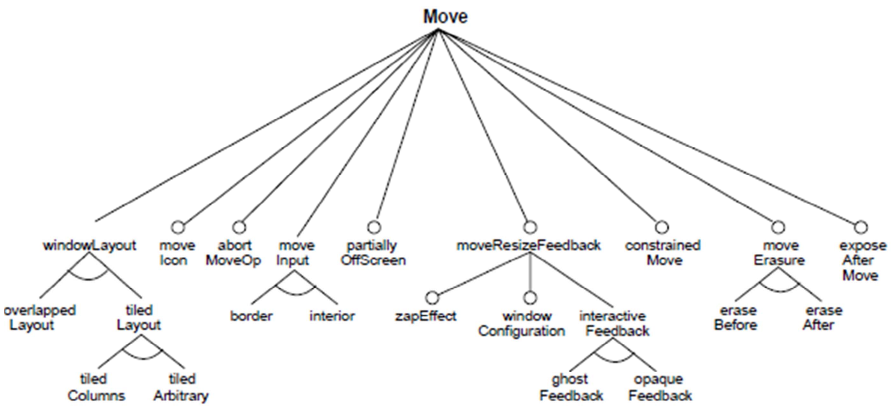


Figure 11. Features Model following the FODA notation

A specific implementation of a solution may be through an instantiation (configuration) of the feature model. A configuration of a feature model is usually defined as the set of features that are selected from a feature model without violating any of the constraints defined on it, but it can also be defined as a valid set of instances of a feature mode, i.e., the relationship between a feature model and a configuration is comparable to the relationship between a class and an object. Figure 12 shows two different valid configurations of the features model shown in Figure 11. The selected optional and alternative features are highlighted in the diagram with boxes. For example, notice that the feature *partiallyOffScreenWindows* (abbreviated on the diagram) is present in *X10/uwm*, but not present in *SunView*. Thus, when a *SunView* window is moved so that its border touches the edge of the screen, the window will stop moving in that direction. In *X10/uwm* the window will continue to move, disappearing off the screen, until the cursor hits the screen edge and stops the window from moving completely off.

Composition rules are the tool for expressing the relationships among features, and are a type of constraint on the use of a feature. In the FODA notation composition rules have two forms:

- A feature *requires* the existence of another feature (they are interdependent). Its textual representation is <feature1> “*requires”* <feature2>.
- A feature is *mutually exclusive* with another (they cannot coexist). Its textual representation is <feature1> “*mutex-with”* <feature2>.

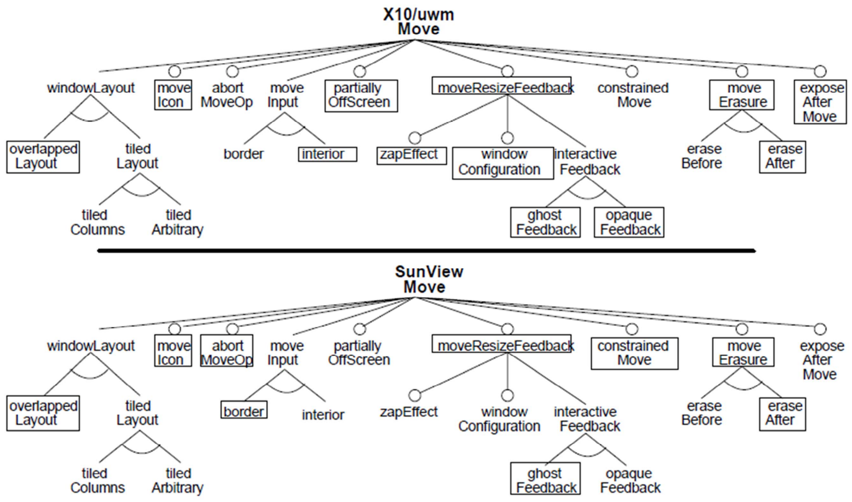


Figure 12. Comparison of two valid configurations of a features model

5.3.2. Cardinality Based Feature Models

Cardinality based feature models [16] integrate the extensions proposed to the original FODA notation [39]. A cardinality based feature model is, in essence, a hierarchy of features. The main difference with the FODA proposal resides in the fact that each feature has associated a cardinality that specifies the number of *clones* of the feature that are allowed in a specific configuration. Cloning features is useful when defining multiple copies of a part of the system that can be replied and configured in different ways.

Features can be organized in *feature groups* with its own *group cardinality*. This cardinality restricts the minimum and maximum number of group members that can be selected in a configuration. Finally an *attribute type* can be specified for a given feature. Thus, a primitive value for this feature can be defined during configuration.

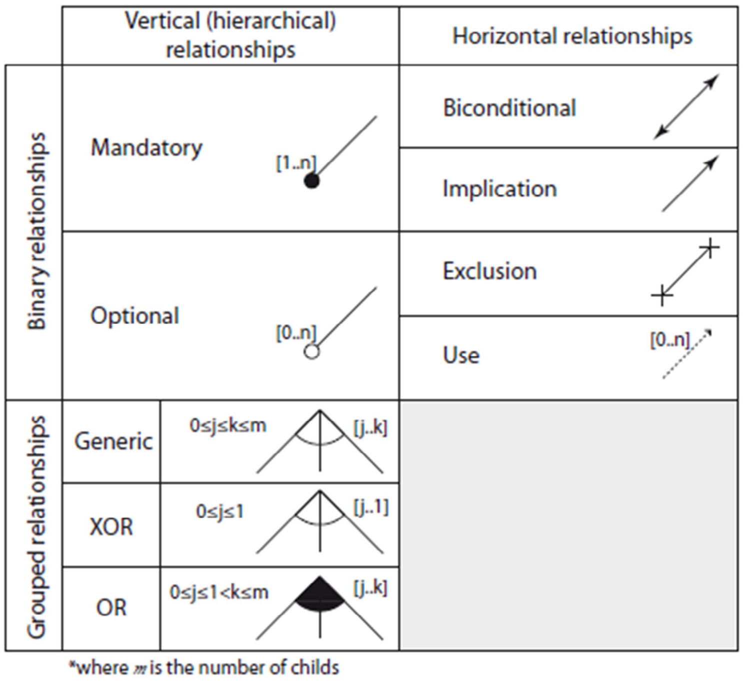
The variability view of the multi-model is going to be defined using a variant of this cardinality-based feature model defined in [26]. This variant allows representing explicitly the relationships between features. Thus, its metamodel represents in a uniform way the hierarchical relationships and the constraints between features. In addition it allows adding OCL-like constraints and, based on these constraints, to check whether the instances are valid or not. The approach also allows representing a Feature Model as a Class Model, by means of model transformations. Table 2 shows the types of relationships the feature metamodel contains. Relationships are classified in two orthogonal groups:

- ***Vertical vs. horizontal relationships:*** Vertical relationships define the hierarchical structure of a feature model and horizontal relationships define dependencies and restrictions between features.

- ***Binary vs. grouped relationships:*** Binary relationships define relationships between two single features. In turn, grouped relationships are a set of relationships between a single feature and a group of childs.

Given this classification, the following relationships exist:

- **Binary and vertical relationships:** these relationships define structural relationships between two single features. In this approach, they represent a *has_a* relationship between a parent and a child feature. They can be mandatory or optional depending on the lower bound value. The upper bound (n) can be on both cases 1 or greater than 1, and indicates how many instances of the child feature will be allowed.
- **Grouped and vertical relationships:** grouped and vertical relationships are a set of binary relationships where the child features share an *is_a* connotation with respect to their parent feature. A group can have an upper and a lower bound. These bounds specify the minimum and the maximum number of features that can be instantiated (regardless of the total number of instances).
- **Binary and horizontal relationships:** these relationships are specified between two features and do not express any hierarchical information. They can express constraints (conditional, implications and exclusion) or dependencies (use). The first group applies to the whole set of instances of the involved features. However, the second one allows us to define the following dependencies at instance level:
- Implication (A -> B): if an instance of feature A exists, at least an instance of feature B must exist too.
- Co-implication (A <-> B): if an instance of feature A exists, at least an instance of feature B must exist too and vice versa.
- Exclusion (A<-> B): if an instance of feature A exists, cannot exist any instance of feature B and vice versa.
- Use (A →B): this relationship will be defined at configuration level, and it will specify that an specific instance of feature A will be related to one (or more) specific instances of feature B as defined by its upper bound (n).



	Vertical (hierarchical) relationships			Horizontal relationships
Binary relationships	Mandatory		[1..n]	Biconditional
				Implication
	Optional		[0..n]	Exclusion
				Use [0..n]
Grouped relationships	Generic	0≤j≤k≤m	[j..k]	
	XOR	0≤j≤1	[j..1]	
	OR	0≤j≤1<k≤m	[j..k]	

*where *m* is the number of childs

Table 2. Symbols used in cardinality-based feature modeling [26]

Figure 13 shows an example of a feature model created using the editor described in [26], illustrating an excerpt of a Car Control System. This control system has different features as *ABS, TractionControl, StabilityControl* and so on. Some of those features are linked by requires relationships; *TractionControl* requires the sensors of the *ABS* system or the GPS Navigation feature requires the feature *MultimediaAudioSystem* to be selected. There is a feature group below the car definition, with a multiplicity ranging from 1 to 7, due to the presence of a mandatory feature (Car Audio) that should be selected in each car configuration. Finally there is a group of optional features below the *CarAudio* feature, meaning that one and only one type of *CarAudio* should be selected in each car configuration.

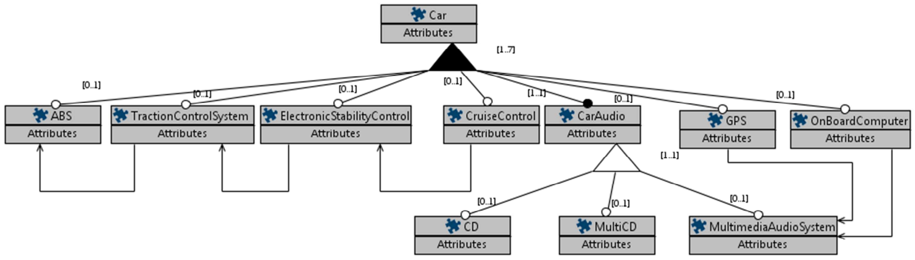


Figure 13. Example of Cardinality Based Features Model

Figure 14 shows an excerpt the structure of the cardinality based features model metamodel. The *FeatureModel* contains *features*; a *feature* can be constrained, and can contain *groups* of child-features. The Feature model contains also, as top level entities, the *relationships* among features and groups. There are two main types of relationships, the structural relationships for expressing the parent-child relationships and the binary and horizontal relationships: uses, excludes, implies and biconditional (co-implication).

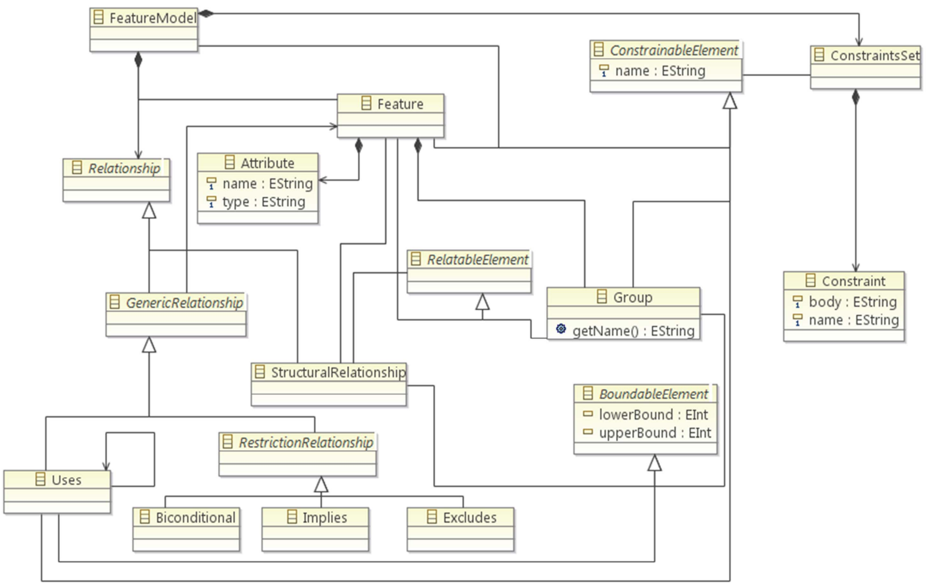


Figure 14. Cardinality-Based Features Model Metamodel

5.4. Quality View

Software Quality is the capability of the software product to satisfy stated and implied needs (requirements) when used under specific conditions [36]. The ISO/IEC FCD 9126-1 [36] defines a quality attribute, , as *“a measurable physical or abstract property of an entity”*. This definition has been redefined and extended in ISO2500-SQuaRE [35] as the *“inherent property or characteristic of an entity that can be distinguished quantitatively or qualitatively by human or automated means”.*

The evaluation of software products, in order to satisfy software quality needs, is one of the most important activities in the software development life cycle. Software product quality can be evaluated by measuring internal software quality (typically static measures of intermediate products), or by measuring external software quality (typically by measuring the behavior of the code when executed), or by measuring software quality in use. The objective is to ensure product have the required effect in a particular context of use as shown Figure 15.

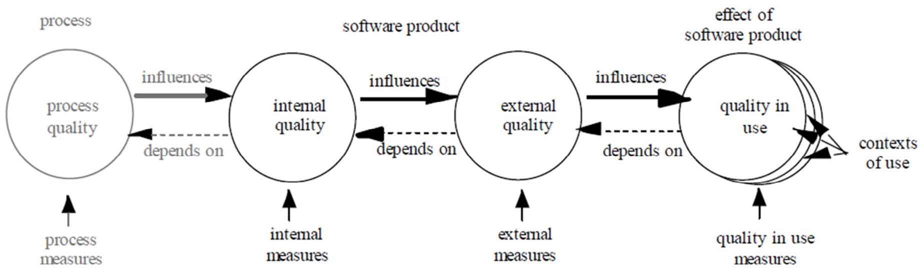


Figure 15. Quality in the life cycle

5.4.1. Quality Models

The quality attributes of a software product are usually difficult to be checked. This is partly due to their nature, but there are other reasons that contribute to this, namely the lack of structured and widespread descriptions. To solve this problem, several approaches that define quality models have been proposed the last years.

A quality model is a "*set of characteristics and relationships among them, which provides a framework for specifying quality requirements and evaluating quality*" [27]. The quality view of a system can be specified through a quality model.

Early quality models that still serve, as reference models are the models proposed by Boehm [8] McCall [46], the ISO 9126 [36] and the latest ISO25000-SQuaRE [35] standard for software quality, which contains a quality model that is based on the McCall's model. All these models decompose quality in characteristics and subcharacteristics that are inherent to the product. For instance, the SQuaRE standard establishes criteria for the specification of software product quality requirements, their measures and evaluation. The standard is defined with the intent of allowing customers and designers defining the quality attributes (which are not defined in the standard) assuming that they are domain specific. The two-part quality model addresses the problem of aligning the customer-defined quality attributes and the quality attributes of the development process. Figure 16 shows the SQuaRE decomposition of software product quality in characteristics and subcharacteristics.

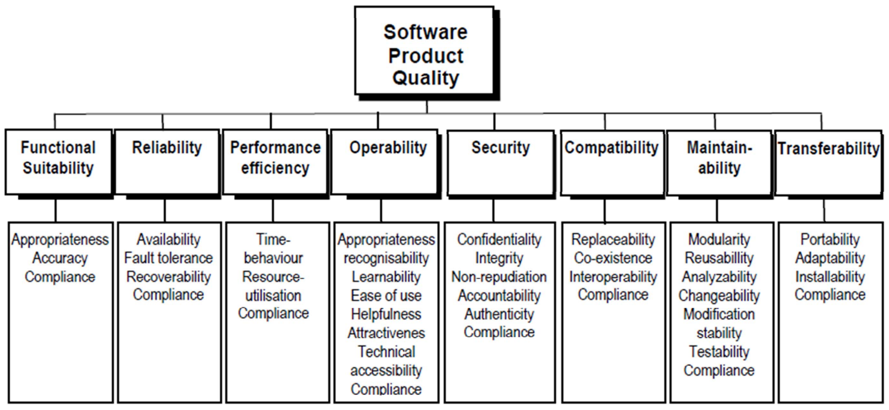

Software Product Quality
Functional Suitability
Reliability
Performance efficiency
Operability
Security
Compatibility
Maintain-ability
Transferability
Appropriateness Accuracy Compliance
Availability Fault tolerance Recoverability Compliance
Time-behaviour Resource-utilisation Compliance
Appropriateness recognisability Learnability Ease of use Helpfulness Attractivenes Technical accessibility Compliance
Confidentiality Integrity Non-repudiation Accountability Authenticity Compliance
Replaceability Co-existence Interoperability Compliance
Modularity Reusability Analyzability Changeability Modification stability Testability Compliance
Portability Adaptability Installability Compliance

Figure 16. SQuaRE Software Product Quality Decomposition

5.4.2. Quality Models for Software Product Line Development

There are approaches defining quality models specifically for software product line development, such as the model proposed by Montagud [50].This quality model, compliant to the SQuaRE standard, defines the hierarchical decomposition of quality in characteristics, subcharacteristics and quality attributes specifically for software product lines.

The model allows, the definition of relationships among the quality attributes (in terms of relative and quantifiable impacts).

The model also allows the definition and operationalization of metrics which can then be associated to different artifacts at different phases of the product line life cycle (core asset or product development).

This quality model can be used to evaluate quality at three different levels:

- **Level 1:** Core asset quality evaluation within the core asset development activity.
- **Level 2:** Selection of core assets that satisfy some quality attributes that a particular software product must fulfill within the product development activity.
- **Level 3:** Quality evaluation of the SPL, giving feedback to the multi-model, in order to validate the relationships defined among the different views of the multi-model.

Figure 17 shows an excerpt with the structure of the Montagud's quality metamodel. It contains only the relevant classes and relationships that play a role in the multi-model. The I*sAssociatedWith* relationship allows the quality characteristics and attributes to be related with others, expressing dependencies among them. The *Perspective* attribute in the different meta-classes allows the definition of quality characteristics, subcharacteristics and attributes for the different quality perspectives (internal and external quality, quality in use and so on). The *AttributeImpact* meta-class allows the definition of positive or negative impacts among quality attributes. The Attribute meta-class includes two attributes *Phase* and *Artifact* that allows defining whether an attribute covers core assets or product quality and whether is applied to features, core assets or final product. A more detailed version of the quality model for software product lines can be found in [50].

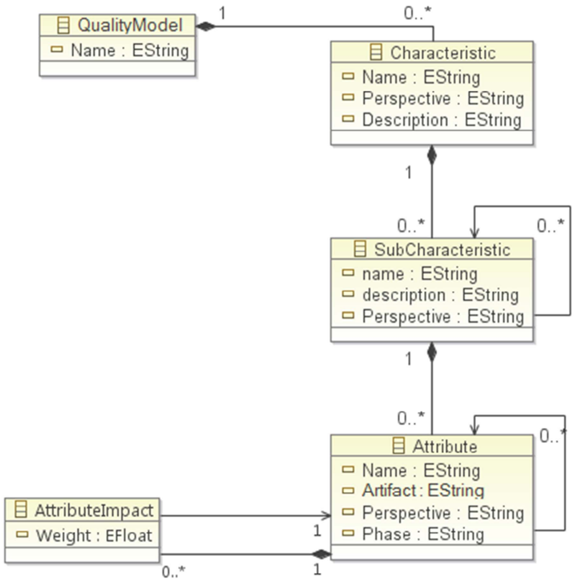


Figure 17. Quality View Metamodel

5.5. Functional View

Functional components under the SPL approach can be seen as the system behavior, which satisfies the requirements of the different features. Those components can be described by using architectural models. Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their compositions, and constraints on these patterns [63]. A more precise definition can be found in [3] *“software architecture of a program or computing system is the structure or structures of the system, which comprises software elements, the externally visible properties of those elements, and the relationships among them”.*

5.5.1. Architectural Description Languages

Since software architecture is often described by using different views, the component and connector view, module view, the deployment view and so on, there are different modeling solutions or standards that can be used as Architectural Description Languages. Clements et al. propose [13] the use of UML, SysML and AADL as architectural description languages.

The Unified Modeling Language (UML) is a standardized, general purpose, visual language for modeling software designs. It was originally created to merge notations for object-oriented modeling, but now UML has grown to become the de-facto standard for representing software designs. UML can be used to describe some information found in software architectures as component-and-connector view, allocation views, behavior documentation and interfaces. However, object oriented abstractions are not always the best tool for describing software architectures. UML has no notation for a layer, context diagram, or rich connector. Many changes were introduced in revision 2.0 of UML and some of them were motivated by the need of improving architecture abstractions. Language elements as connectors and ports were introduced to address some problems. Other elements were enriched to improve their suitability; for example, UML components now share many features with classes, such as the ability to add interfaces and behavioral descriptions. The result is that today's 2.x versions of UML are better adapted to documenting architectures, but there are some gaps between UML and architectural abstractions, particularly when defining Component and Connector views.

The System Modeling Language (SysML) language [54], although it is not intended to be a dedicated architecture description language, it provides enough constructs to meet many of the needs of a systems engineer. The engineer can represent, by means of the different abstraction the language provides, the topology of the hardware and allocate software units into those hardware units. It is also possible to represent the various architecture views needed to document software architectures.

SysML is a general-purpose systems-modeling language intended to support a broad range of analysis and design activities for system-engineering applications. SysML is a standard maintained by the Object Management Group (OMG) in collaboration with the International Council on Systems Engineering (INCOSE). SysML started as a UML profile (in the same way that MARTE for real-time systems). Being a UML profile means that SysML reuses much of UML, but it also provides the extensions needed to cover the demands of systems engineers.

The Architecture Analysis and Design Language AADL [22] is an aerospace, automotive and medical device standard developed by SAE.

The AADL standard defines a textual and graphical language to represent the runtime architecture of software system as a component-based model in terms of distinct components and their interactions. AADL is extensible through user-defined properties and sublanguage. It includes abstractions of software, computational hardware, and system components for doing the following:

- Specifying and analyzing real-time embedded and high dependability systems, complex systems of systems, and specialized performance capability systems ,
- Mapping software abstractions onto computational hardware elements.

The AADL is especially effective for model-based analysis and specification of complex real-time embedded systems.

We have decided to use AADL for supporting the functional view of the multi-model since it is specifically designed for architecture modeling. In addition there are some research efforts to optimize embedded systems architectures by using model transformation as input of the description of the architecture in AADL. Our intention, as mentioned in the future works section, is to use architecture optimizers to analyze the output of the production plan, in order to obtain not only an architecture that meets the required quality attributes but also the best architecture in terms of performance, cost, memory consumption and so on. The weak point is that AADL is not a OMG standard and it is out of the MDA approach, but since the AADL is becoming a de-facto standard in the software architecture field and since it is fully supported by the technological space (i.e., the AADL metamodel is defined in Eclipse EMF), this language is, under our point of view, the best choice for expressing the functional view of the proposed multi-model. An overview of the AADL components, component types, component implementations, packages, property sets and annexes are provided below.

5.5.2. The AADL Language

This subsection describes in detail the different components that integrate an AADL specification, its purpose and structure.

5.5.2.1. AADL Components

Components form the central modeling vocabulary for the AADL. Components are assigned a unique identity (name) and are declared as a type and *implementation* within a particular component category. A component category defines the runtime essence of a component. There are three distinct sets of component categories:

Software Components:

- *Thread:* basic unit of concurrent execution.
  - *Process:* a protected address space. A process must contain at least one thread.

- *Thread Group:* a compositional unit for organizing threads, declares the features and required subcomponent access through which threads contained in a thread group can interact with components declared outside the thread group.
- *Data:* data types and static data in source text.
- *Subprogram:* callable sequentially executable code.

Hardware Components

- *Processor:* an abstraction of the hardware and software that is responsible for scheduling and executing threads. For their execution, threads will be bound to a processor that supports the dispatch protocol required by the thread. A processor component must contain at least one memory component (the application software executes in the attached memory) or provide access to memories via a bus**.**
- *Memory:* components that store data and code. Memory can represent any randomly accessible storage (e.g., random access memory [RAM], read-only memory [ROM]) or more complex (random or sequential) storage devices (e.g., mag- netic disk, optical disk, and tape). Memories have properties such as size (e.g., 8 bits), proto- col (e.g., read_only, write_only, read_write), and amount (e.g., max_word_count). Subprograms, data, and processes are bound to memory components for access by processors when executing threads.
- *Device:* components that interface with and represent the external environment. A device can represent single-function components (e.g., sensors) as well as more complicated components (e.g., global positioning system [GPS] units, camcorders). In reality, the more complicated de- vices would have internal processors, memory, and so forth that are not explicitly modeled.
- *Bus:* components that provide access among execution platform components as processors, memories, and devices. The **bus** component represents a communication channel, typically hardware coupled with a communication protocol.

Composite

- *System:* a composite of software, execution platform, or system components. Supports the hierarchical grouping of both software and hardware components. A system can be used to organize processes, execution platform components, or the combina- tion of both. A system can also contain lower level system instances. It is generally used early in the modeling process as a generic modeling component.

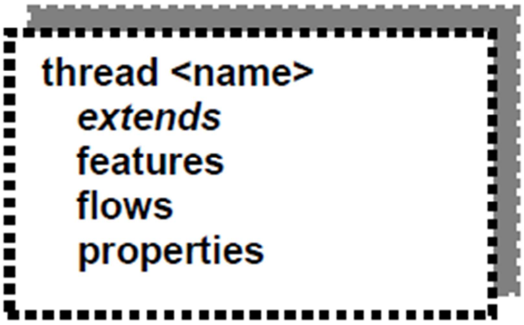
AADL components interact exclusively through defined interfaces. A component interface consists of directional flow through:

- *Data ports* for unqueued state data.
- *Event Data Ports* for queued message data.
- *Event Ports* for asynchronous events.
- Synchronous subprogram calls.
- Explicit access to data components.

5.5.2.2. Component Types

An AADL component type declaration defines, by means of subclauses, the component's externally visible characteristics. The type declaration specifies four subclasses defining those characteristics. Figure 18 shows the four subclauses of a thread type declaration.

- ***Extends*** clause enables the component type to build upon the definition of another existing component of the same type. The component defined as extension of another inherits the characteristics of the original type.
- ***Features*** subclause defines the interfaces of the component, including the input and accesses required by the component, and all the output and items the component provides.
- ***Flows*** subclause defines specifications of logical flows through the component from incoming interaction points to outgoing interactions points (These flows can be used to specify end-to-end flows without having to expose or have available any *implementation* detail of the component. Flows can trace data, control, or mixed flow by connecting event and data ports).
- ***Properties*** subclause specifies the properties of the component that apply to all instances of this component unless overwritten in implementations or extensions.

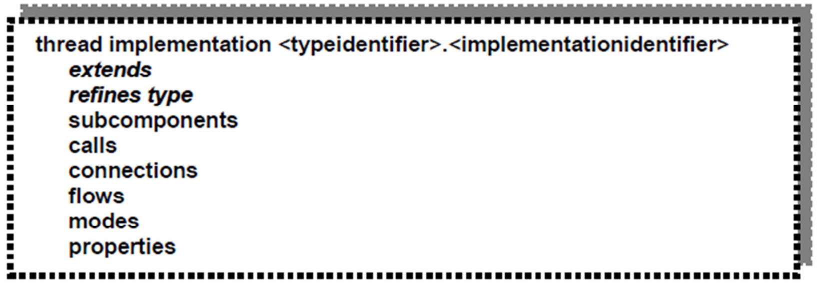


```
thread <name>
   extends
   features
   flows
   properties
```

Figure 18. Type Declaration Subclauses

5.5.2.3. Component Implementations

A *component implementation* specifies the internal structure in terms of *subcomponents*, *interactions* (*calls* and *connections*) among the features of those subcomponents, flows across a sequence of subcomponents, modes that represent operational states and properties of a component type defined previously. Figure 19 shows the different subclauses of a component implementation clause. The *subcomponents*, *connections* and *calls* clauses allow the definition of a component implementation as a collection of subcomponents and their interactions. *Flows* subclause represents the flow implementation defined in the component type declaration. *Modes* clause represent alternative operational modes that may manifest themselves as alternate configurations of subcomponents, calls sequences, connections, flows sequences, and properties. *Properties* define intrinsic characteristics of a component. There are predefined properties for each component implementation, and the user can define additional ones by using the language extension capabilities.



```
thread implementation <typeidentifier>.<implementationidentifier>
    extends
    refines type
    subcomponents
    calls
    connections
    flows
    modes
    properties
```

Figure 19. Type Implementation Subclauses

In addition AADL allows the declaration of multiple implementations of a component type, allowing behavioral variants with the same external interface.It also allows extending and refining previously declared component implementations through the *extends* or *refines* subclauses. Extended implementations inherit all the characteristics of the original component implementation and all of its predecessors. Refinement allows partially specified component implementations (templates) to be completed as opposed to the extension mechanism that allows a component implementation to be expressed as a variation of a common component description through additions. Extended implementations can add *property* values to the features of its component type by means of the *refines type* subclause.

The *subcomponent* subclause allows the component decomposition tithing component implementation declaration clause. The subcomponent represents the decomposition element and the classifier represents a choice in the family of implementations of that subcomponent (that had been previously defined as component type and component implementation). A component instance is created by instantiating a component implementation and each of its subcomponents recursively.

5.5.2.4. Packages, Property Sets and Annexes

AADL packages allow the declaration of collections of components to be organized in separate units with their own namespaces. Packages support the independent development of large-scale systems providing independent namespaces for each group of subsystem elements.

AADL *property set* is a named grouping of *property* declarations that define new properties and property types that can be included in a specification. These properties are referenced using the property set name and can be associated with components and other modeling elements (e.g., ports or connections) within a system specification. Their declaration and use become part of the specification.

An *annex* enables a user to extend the AADL language, allowing the incorporation of specialized notations within a standard AADL model.

5.5.2.5. The AADL Textual and Graphical Specifications

This section shows an excerpt of the AADL textual and graphical specification. Table 1 shows the AADL specification as a human readable collection of textual declarations that comply with the AADL standard.

Declaration	Description
Component Type: System, process, thread, thread group data, subprogram, processor, device, memory, and bus	The component type declaration establishes the identity (component category and name) and defines the features, flows, and properties of a component type. A component type declaration may also declare the type as an extension of another type (extends).
Component Implementation: System, process, thread, thread group data, subprogram, processor, device, memory, and bus	The component implementation declaration establishes the identity (component category, type, and name) and defines the refinements (refines type sub-clause), subcomponents, calls, connections, flows, modes, and properties of a component implementation. The identity must include a declared component type consistent with the component category. The component implementation declaration may also declare the implementation as an extension of another implementation (extends sub-clause).
Port Group Type	Port group type declarations establish the identity (name) and define the features and properties of a grouping of ports and/or port groups. Within the features declaration, a port group may be defined as the inverse of another port group. A port group type declaration may also declare the port group as an extension of another port group type (extends).
Package	The package declaration establishes the identity (name) of a collection of AADL declarations, groups those declarations into private and public sections, and declares properties associated with a package. Packages are used to logically organize AADL declarations. AADL component type, implementation, and port group declarations placed in AADL packages can be referenced by declarations in other packages.
Property Set	Property set declarations introduce additional properties, property types, and property constants that are not included as pre-declared AADL properties.8 Each property set has a unique global name and provides a unique namespace for the items declared in it. In other words, properties and property types declared in a property set are referenced by property set name and item name.
Annex Library	Annex library declarations establish the identity (name) and define the contents of a set of reusable declarations that are not part of the standard AADL language. Annex declarations are used to extend AADL's core modeling and analysis capabilities.

Table 3. AADL Textual notation

Figure 20 shows AADL graphical notation. AADL graphical notation facilitates a clear visual presentation of a system's structural hierarchy and communication topology and provides a foundation for distinct architecture perspectives.

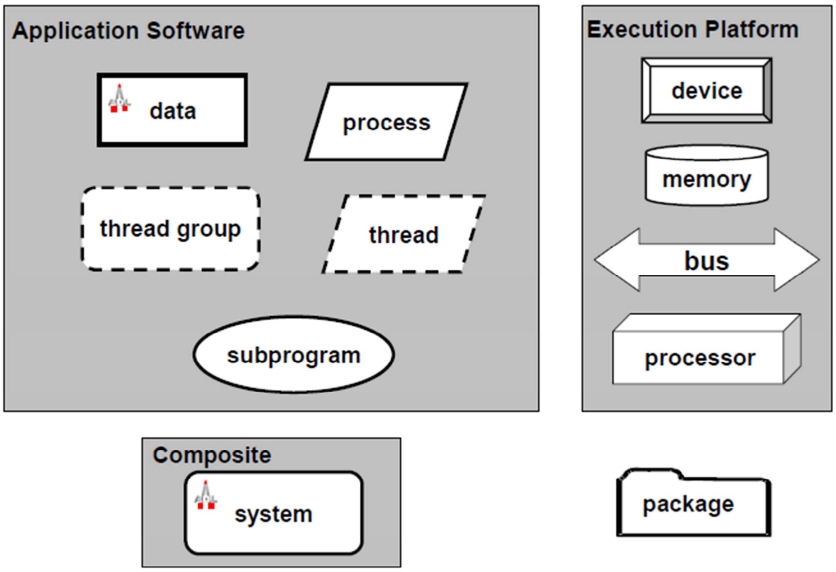


Figure 20. AADL Graphical Notation [22]

5.5.2.6. The AADL Metamodel

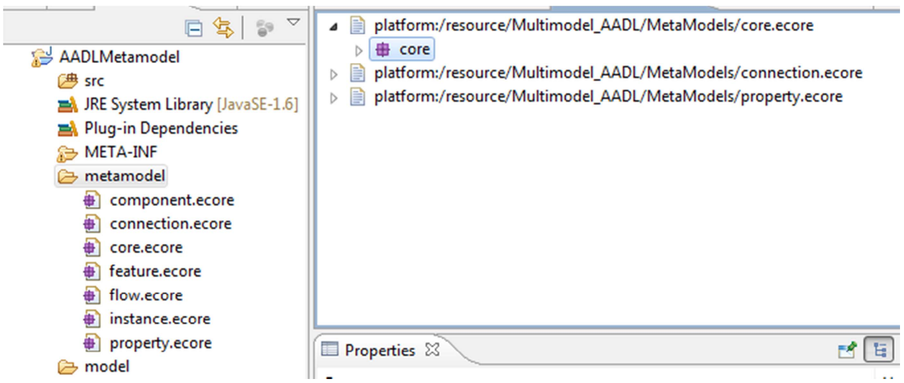
AADL Metamodel is defined as a multi-model itself. The representation of the AADL metamodel as a single unit would make it difficult to understand and maintain. The modular metamodel approach allows metamodels for such sublanguages to be defined separately in Ecore and added to the core AADL metamodel.

The metamodel for the core AADL is divided into six metamodel packages:

- Core: defines the concepts of component type, implementation, and subcomponent as abstractions, as well as packages and modes.
- Component: defines the concrete classes for the different categories of components, including the constraints on their subcomponents.
- Feature: defines the features of component types.
- Connection: defines the connections between component features.

- Flow: defines flow related elements of the AADL.
- Property: defines the elements for associating property values and for introducing new property types and properties via property sets.

Figure 21 shows the multi-model structure with the seven metamodels on the right side and the external packages (connection and property) loaded on the core metamodel on the left side.


AADLMetamodel
src
JRE System Library [JavaSE-1.6]
Plug-in Dependencies
META-INF
metamodel
component.ecore
connection.ecore
core.ecore
feature.ecore
flow.ecore
instance.ecore
property.ecore
model

platform:/resource/Multimodel_AADL/MetaModels/core.ecore
core
platform:/resource/Multimodel_AADL/MetaModels/connection.ecore
platform:/resource/Multimodel_AADL/MetaModels/property.ecore

Figure 21. Multi-Package Metamodel Structure

5.6. Relationships among Views and Metamodel Definition

Variability management involves the manipulation of domain features, represented as feature models, and the support of such variability in the so-called product line core assets. Specifications of both system variability and functionality can be dealt with models that are independent from each other. In addition, quality requirements should also be included in the product line development process by means of a quality model. Therefore, we will consider (at least) three different views of software systems: System Variability View (SVV), System Functional View (SFV), and the System Quality View (SQV).

The SVV is based on cardinality-based feature models. The SFV is considered in the different stages of the production process of a SPL by means of different architectural models (e.g., component-connector). Finally, the SQV is defined over the aforementioned views defining relationships that guide the production plan towards a final product. These different system models correspond to different system views that constitute a multi-model that represents a SPL by considering the relationships among views.

However, in order to define production plans to guide the development of specific software products using the multi-model, we should establish relationships among the elements of the different system views. This will allow us to formally analyze properties over the system as a whole (i.e, the consistency of the multi-model).

In section 5.3 we have defined a feature as "*user-visible aspects or characteristics of a system*" and in section 5.5 we have defined functional component as the"*system behavior, which satisfies the requirements of the different features*". According to these definitions we must consider, at least, the following relationships:

- A functional component can be also combined with others in order to fulfil those requirements (more than one functional component to fulfil the requirements of one feature).
- A functional component can fulfil the requirements of more than one feature.

In addition, if we focus on the quality attributes, the features selected in a product configuration will impact directly over the quality attributes of the product. The mapping between features and functional component will impact also in the quality of the final product. .

This relationship is analyzed, from the asset point of view by Nolan et al. in [58], concluding that the selection of core assets with different levels of quality (maturity and variability in this case) increases the cost of the core asset

We have defined the relationships with features and quality attributes. However, the functional view, as mentioned in section 5.5, is modeled by using the AADL modeling language. In this language there are different meta-classes that could be considered as functional components (system, component, thread, process and so on), depending on the level of detail that needs to be achieved.

To cope with this problem we have defined the *ImpactingElement* meta-class that allow, by defining inheriting meta-classes, to have different meta-classes that can impact over the quality attributes.

The *Impact* meta-class expresses the relationship between the *ImpactingElement* and the quality attributes. This relationship is associated with the *EAtribute* meta-class, defined by applying the proxy pattern over the *Attribute* meta-class from the quality metamodel. The *impact* meta-class includes the *Weight* attribute that allows to expressing the relative importance of this impact over the quality attribute. The impacts could be defined as the relationship of more than one *ImpactingElement* over a quality attribute, meaning that the combination of some elements may have impact (negative or positively) over the quality attributes. This allows us to express more complex relationships that occur when dealing with the selection of features or components. In general the quality of the final product can be more (or less) than the addition of the quality of the parts. The composability of quality attributes had been studied from the perspective of Component Based Software Engineering perspective [15] but the ideas can be applied also to SPL development.

The *EComponentType* meta-class contains the relationship between features and functional components. There are different meta-classes that inherit from this meta-class, such as *ESystemType*, *EProcessType* and *EThreadType*. All those three classes have also been defined also as subclasses of the homonymous *ComponentType* subclasses, by applying the proxy pattern over the AADL's metamodel meta-classes. Therefore a feature can be associated with an *ESystemType,* an *EProcessType* or an *EThreadType,* as shown in Figure 22.

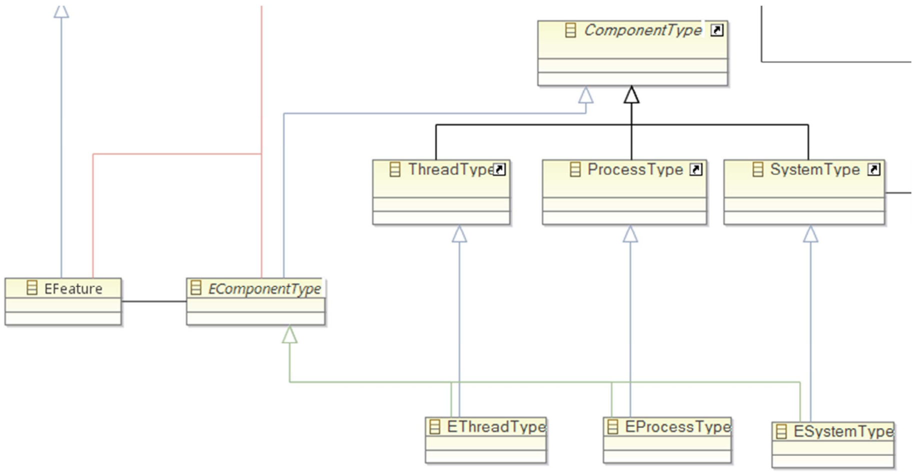


Figure 22. EComponent Type Subclases (Excerpt of the Multi-Model's Metamodel)

Figure 23 shows the relationships among the different meta-classes, that have an inheritance relationship with the *ImpactingElement* meta-class. It also shows the relationships between the *ImpactingElement* and the *Impact* meta-classes.

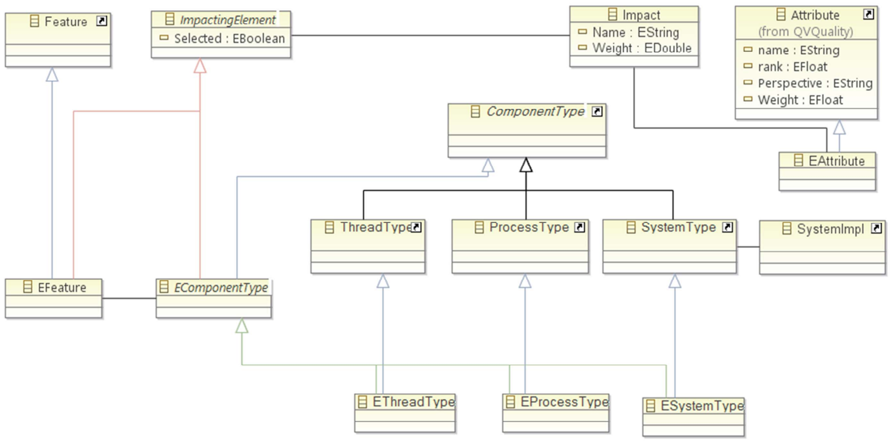


Figure 23. Impacting Elements Sub-classes (Excerpt of the Multi-Model's Metamodel)

The structure of the multi-model, with the proxy meta-classes and the definition of the *ImpactingElement* allows adding new kinds of entities to the meta-classes that already have an impact over the quality attributes. If in the future, we discover that other entities have an impact on the quality attributes, this impact can be defined only by defining an inheritance relationship with the *ImpactingElement* meta-class. By the same way, if in the future other entities are discovered to have relationships with features, this relationship can be established by only defining an inheritance relationship with the *EComponentType* meta-class.

Chapter 6. Quality-Driven Model Transformations and SPL

The existence of several system models or views with relations between them suggests the parameterization of the production process by means of the multi-model proposed. However, in order to do this, we should define production plans that are able to derive a specific product by using the multi-model as input. In this work, the production plan of a software product line will be realized by means of model transformations.

In a Model Driven Software Development (MDSD) context, model transformations become a key aspect in the development process. This means that part of the development effort is relocated to the transformation design and implementation. The design of model transformations needs to deal with not only with functional concerns, i.e., how to move from a source model to a target model, but also with nonfunctional concerns, which are the desired quality attributes of the target model. In order to do this, the transformation designers need to identify possible alternative transformations and choose those alternatives that produce the target model with the desired quality attributes.

The chapter is organized as follows: In Section 6.1 we introduce the concept the alternative model transformations and its purpose in product line development processes. In Section 6.2 we present our architecture for supporting quality-driven model transformation processes dealing with alternative transformations that impact over the quality attributes of the target models. Finally in Section 6.3, we describe the schema that must be followed for defining quality-driven model transformations.

6.1. Alternative Model Transformations

In current model transformation design practices, the required quality attributes of the target model are hard-coded in the transformation definition even though this information is not directly related to the functional concern of the transformation. Hard-coding refers to the software development practice of embedding input or configuration data directly into the source code of a program or specification, rather than of obtaining the required information or data from external sources.

In a MDD approach, a software system is developed by refining models. This refinement is implemented as transformations over models. Given a source model there may be several ways to transform this source model into target models.

Alternative target models may exist when a structural pattern in the source model in the source model can be transformed into different structural patterns in the target model by the application of different transformation rules. Those alternative target models may have the same functionality but may differ in their quality attributes.

In a software product line development context there are at least two main scenarios where alternative model transformations may exist:

- Different functional components or assets, which, having the same structure, interface and behavior are different with regard to quality attributes. It is quite common, If we consider the embedded systems context, to have various subsystems (hardware or software), that develop exactly the same functionality but with different attributes as cost, time response, power consumption and so on depending on the component vendor or technology.
- The improvement of the quality of models (or software artifacts) during the development process, both in core asset and in product development activities. This can be applied as endogenous, horizontal transformations (the source and target models conform to the same metamodel in the same level of abstraction), such as optimization of architectural models by applying architectural patterns for obtaining architectures that meets some quality attributes, similar to the technique applied by Maswar et al in [45]. It also can be applied to exogenous vertical transformations (source and target models that conform to different metamodels across levels of abstraction as PIM to PSM transformations), where is common to have design alternatives, as the examples shown in [28], [29], [30], [34],

6.2. A Quality-Driven Model Transformations Architecture.

This section presents an overview of our architecture to support quality-driven model-transformations. A key aspect of this architecture is the use of two models as active artifacts to drive the selection and execution of model transformations: a quality model to represent quality attributes, and a transformation model to represent the relationship among quality attributes and the alternative transformations.

Unlike other transformation processes, which only use source models as input to apply the transformations, our approach proposes the use of two additional models: a quality model to represent quality attributes, and a transformations model to represent the relationship among quality attributes and the alternative transformations in a particular domain.

The architecture divides the transformation process into two phases: Rules Analysis and Transformation. In the Rules Analysis phase, the software engineer performs the Alternatives Identification activity to identify the alternative transformations in a particular domain. The domain expert also performs the Trade-Off Analysis activity among quality attributes and alternative model transformations. This trade-off analysis is performed by means of the Analytic Hierarchy Process (AHP) [3].

The AHP is a decision-making technique that is widely used to resolve conflicts in which it is necessary to address multi-criteria comparisons. In our approach, we use the two main phases of the AHP technique: i) Value Score Computation, and ii) Normalization. In the first phase of the AHP, the domain expert performs two pair wise comparisons, weighting the relative importance of the different quality attributes identified by comparing every pair of quality attributes using the AHP weighting scale [3] and obtaining a W_z value. The domain expert then determines how each design alternative relatively supports the relevant quality attributes. For every quality attribute, we compare all the alternatives in a pair-wise fashion, thus obtaining the values of Siz which allows us to then compute the value score for each possible alternative transformation i for the structural pattern j using the following formula:

$$V_{i,j} = \sum_{z=1}^{k} W_z S_{iz}$$

Finally, in the second phase of the AHP, the value scores obtained in the previous phase are normalized. The value scores from different decisions must be compared, and they are then scaled relatively.

Once the trade-off analysis has been completed, the software engineer performs the Rules Selection activity. This activity uses the results from the trade-off analysis and the quality attributes selected by the software engineer to generate the Active Rules Model, which contains the selected transformations from the set of alternative transformations that produces a target model that best fits the desired quality attributes.

In the second phase of our architecture (Transformation), the Quality-Driven Model Transformation activity is performed by using both the definition of non-alternative transformations and the definition of the selected transformations from the Active Rules Model. Quality Driven Model Transformation activity uses the Source Model and the Actives Rules Model as input to generate the Target Model. Every time a structural pattern with alternative transformations is found in the source model the rules associated to this structural pattern in the Active Rules Model is applied. Since only those alternative transformations which best support the desired quality attributes are executed, we ensure that the produced target model satisfies the desired quality attributes. Figure 24 shows schema the two phases, the different activities, transformation processes and artifacts of the architecture.

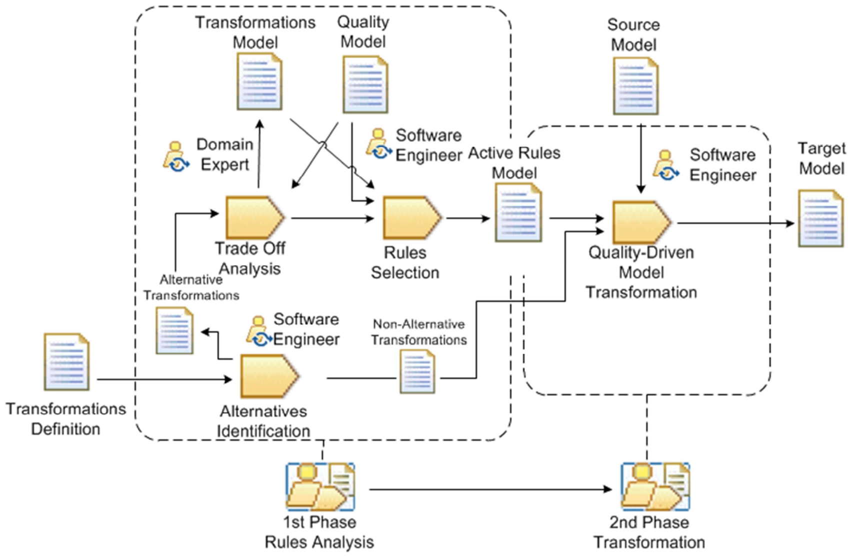


Figure 24. Phases and artifacts of the quality-driven model transformation architecture

The next sections will introduce the artifacts of our architecture, describing their metamodels.

6.2.1. Transformation Model

The transformations model represents the relationships among alternative transformations and quality attributes (trade-off analysis results) in a given domain. The most relevant classes of the corresponding metamodel are described in Figure 25.

The Characteristic, Subcharacteristic, and Attribute classes are used to represent the domain-specific subset of the quality model from Section 3.1. The domain expert can also define new associations among subcharacteristics and among attributes.

The Attribute class has an attribute for representing the AHP ranking of the quality attributes obtained from the trade-off analysis. The impacts between quality attributes and the relative importance of these impacts (weight) are expressed by means of the Impact class. The Transformations Model also contains a StructuralPattern class to represent the structural patterns in the source model that will be transformed into elements of the target model. A structural pattern is a subset of elements in the source model that has a transformation definition. A transformation definition is defined by the Rule class. In this transformations model we include only those structural patterns that have alternative transformations (those structural patterns that have no alternatives are not related to quality attributes nor are they weighted since they must always be executed). A structural pattern is defined by an id, name, and description. The different alternative transformations for a structural pattern are represented by the association between the Structural Pattern class and the Alternative class.

Finally, the transformation of a structural pattern using one alternative or another may have a different effect on the quality attributes of the target model. These effects are quantified by the domain expert during the trade-off analysis and are represented in this metamodel by means of the Weight class, which is associated with the alternative and its respective quality attribute.

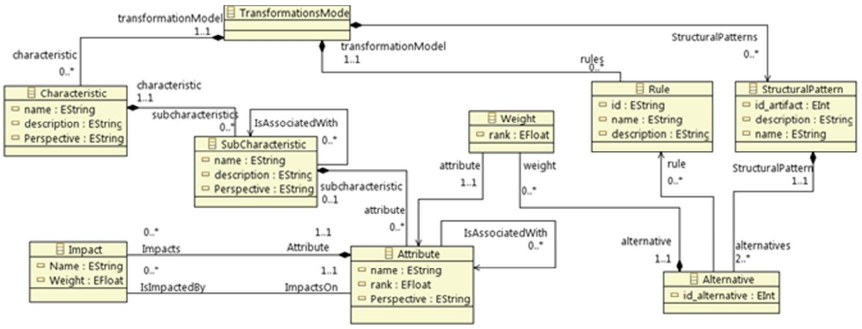


Figure 25. Quality Driven Model Transformations Transformation Metamodel

6.2.2. The Quality Model

As stated in section 5.4.1 a quality model is defined as being the set of characteristics and the relationships between them, which provide a framework for specifying quality requirements and evaluating quality [35]. These kinds of models are often used to represent hierarchies of quality attributes. Top-level attributes represent general quality characteristics (e.g., usability, maintainability) while bottom-level attributes represent measurable properties of software artifacts.

This quality model allows the transformation designer to select the quality attributes to be considered by the transformation process. This quality model its independent from the one defined in section 5.4 and addresses only the quality attributes of the models in the domain of the transformation process. Its corresponding metamodel is composed of four classes (Figure 26): a *QualityModel* class which is the container class, and the *Characteristic, Subcharacteristic* and *Attribute* classes which represent the ISO 25000-SQuaRE [27] hierarchical quality decomposition structure for quality attributes. These classes have three attributes, which represents the *name*, *description* information, and the stakeholder's *perspective*. Different categories of stakeholders have their own quality perspective signifying that the particular set of quality characteristics varies according to the stakeholders' different categories (e.g., maintainability is perceived by the software engineer whereas time behavior is perceived by the end users). The *Attribute* class has an additional attribute, *checked* to represent the selection made by the software engineer. The quality model contains the complete ISO attribute decomposition and the domain expert can choose a subset of those quality attributes that are relevant for a given domain to define the Transformations Model (see Section 6.2.1). We use an extensible and standard-based quality model in order to been capable to adapt this approach to multiple domains.

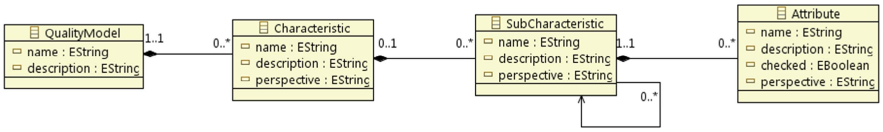


Figure 26. Quality Driven Model Transformations Quality Metamodel

6.2.3. The Active Rules Model

The active rules model represents the information from the transformation rules that are eventually selected from among the alternatives through which to perform the transformation of the source model. This model is generated automatically by the *rules selection* activity in the first phase of the architecture. The corresponding metamodel shown in Figure 27 is composed of three classes, the *ActiveRulesModel* class which is a container class, the *StructuralPattern* class that represents the information concerning the structural patterns, and the *Rule* class that represents the information concerning the selected transformation rules.

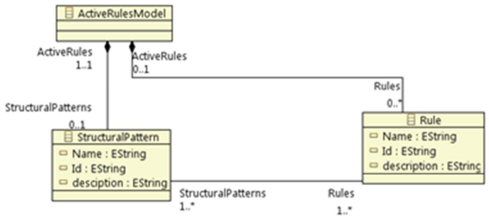


Figure 27. Quality Driven Model Transformations Active-Rules Metamodel

6.3. Transformation Definition

A model transformation definition contains transformation rules. Transformation rules are the units in which transformations are defined. A transformation rule is responsible for transforming a particular selection (structural pattern) of the source model into constructs of the target model. A transformation rule consists of two parts: a left-hand side (LHS) and a right-hand side (RHS). The LHS accesses the source model, whereas the RHS expands in the target model.

A transformation could be declaratively defined as a *relation* among models. Relations in a transformation declare constraints that must be satisfied by the elements of the participating models. A relation is defined by two or more domains, preconditions, and post-conditions that specify a relationship that must be hold between the elements of participating models [53]. The precondition specifies the conditions under which the relationship needs to hold. The post-condition specifies the condition that must be satisfied by all model elements that participate in the relation.

The transformation process is controlled by matches. A match occurs during the application of a transformation when elements from the left-hand and/or right-hand model are identified as meeting the constraints specified by the declaration of a transformation rule. A match triggers the creation (or update) of model elements in the target model, and is driven by the declarative and/or implementation parts of the matched rule [17], [25].

In our approach, a model transformation definition contains two kinds of rules: top-level transformation rules and non top-level transformation rules, similar to the classification performed in the QVT-relations language [53]. Existing alternative rules in the transformation definition will be modified to become non top-level transformation rules, and a new top-level rule will be added for each structural pattern with alternative transformations. The alternative rules will never automatically match; they will be called from the new top-level rule which will perform the transformation based on the information in the active rules model.

6.3.1. Top-level transformation rules

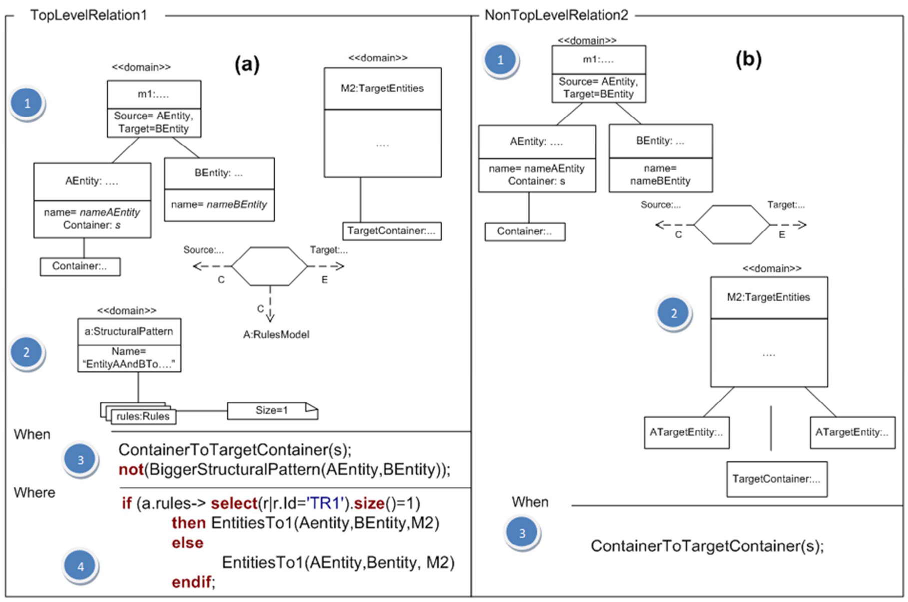
The alternative transformations are dealt with by performing a transformation composition [5] of the alternative transformation rules. Figure 5(a) shows an example of a generic top-level transformation rule for alternative transformations using the graphical QVT-relations notation [53].

The set of all the alternative transformation rules for a given structural pattern are grouped into a new top-level transformation rule. This top-level transformation rule defines the common structural pattern among all the alternatives in the source model (see Figure 28.a.1). The structural pattern in the active rules model defines the association between the name of the alternative and the various non top-level rules that create or update the corresponding particular constructs in the target model (see Figure 28.a.2). The post-condition of the top-level transformation rule deals with the invocation of the corresponding alternatives represented as non top-level rules (see Figure 28.a.4). Preconditions may be needed to describe constraints that must hold in order to execute this transformation rule (see Figure 28.a.3).

6.3.2. Non top-level transformation rules.

The constructs created or updated in the target model are specified by the non top-level transformation rules. These consist of a left-hand side that defines the structural pattern in the source model and a right-hand side that describes the constructs in the target model.

Preconditions may be needed to describe constraints that must hold in order to execute this transformation rule. Figure 28.b.1 shows the structural pattern definition in the source model. Figure 28.b.2 shows the constructs in the target model, and Figure 28.b.3 shows a precondition using the when clause.



```
ContainerToTargetContainer(s);
not(BiggerStructuralPattern(AEntity,BEntity));
```

```
if (a.rules-> select(r|r.Id='TR1').size()=1)
        then EntitiesTo1(Aentity,BEntity,M2)
        else
                EntitiesTo1(AEntity,Bentity, M2)
        endif;
```

```
ContainerToTargetContainer(s);
```

Figure 28. A generic top-level and non-top level with the QVT graphical notation

6.3.3. Rule-Conflict Avoidance

When dealing with transformation rules, one of the problems that might arise is the conflict among rules. A conflict appears when an structural pattern in the LHS of a transformation rule overlaps with an structural pattern in the LHS of any other transformation rule.

The application of our two-phase architecture reduces the number of possible conflicts, due to the selection among alternatives performed in the first phase.

We define the granularity of a transformation rule as the size of structural pattern in the LHS of the transformation rule. A transformation rule has higher granularity if its structural pattern covers a larger portion of the source model. The theoretical idea for solving the granularity issue is to define a high-order relationship (hereafter HOR function), which assigns a value to each transformation rule based on the number of entities covered by the structural pattern defined in the domain of the source model. This also reduces the ambiguity of the whole transformation process.

In a Term Rewriting System R, an overlapping can be defined as follows:

Given two rules, *R1* and *R2 ϵ R, l1→r1* and *l2→r2* they overlap if there exists almost one non-variable instance *u* of *l1* (or *l2*) such as *l1/u* (alternatively *l2/u*) unifies with l2 (alternatively with *l1*) [18].

If we consider the transformation rules as a term rewriting system, then we can reduce the possible ambiguity among rules by incorporating a high-order relationship. Using the example described in Figure 5(a) as TopRelation1 we could have another rule (TopRelation2) defined on a structural pattern that includes the structural pattern of the TopRelation1.

$$L_{\text{TopRelation1}} \subset L_{TopRelation2} \rightarrow HOR(L_{\text{TopRelation1}}) \leq HOR(L_{\text{TopRelation2}})$$

Therefore, when an aEntity and bEntity in the source domain of TopRelation1 are found in the source model (see Figure 28.a.1)), and if the entities defined in the structural pattern of the TopRelation2 are also found in the source model, only TopRelation2 will be executed, and TopRelation1 will fail due to the precondition shown in Figure 28.a.3. This precondition is defined by means of a query. A query is an expression that is evaluated over a model. The result returned by a query is one or more instances of types defined in the model, or defined by the query language. Executing a query can be considered as a specific transformation task, since it only returns elements from the model or entities defined by the query language.

This query evaluation verifies the presence of the whole LHS of another rule that has a larger value in the HOR, and this query is therefore evaluated as true, causing the precondition of TopRelation1 to be evaluated as false, and that it cannot consequently be executed.

Chapter 7. A Quality-Driven Production Plan for SPL

In this chapter we describe our proposal for defining a production plan where the quality attributes play the role of being the decision factor during the product configuration process and for selecting among different design alternatives.

The chapter is organized as follows: In Section 7.1 we describe how to define the production plan during in the core asset development phase, how to define the multi-model, how to detect design alternatives and define quality-driven transformation that processes as part of the attached processes associated to the core assets. In Section 7.2 we describe how the production plan is applied in the product development phase driving the product configuration and the different model transformation processes that are part of this phase.

7.1. Production Plan Definition

There are two pivotal activities that should be performed in the core asset development activity for defining the quality-driven production plan:

- Multi-model definition: The domain experts and engineers must fulfill the different views of the multi-model with the variability, functional and quality information.
- Definition of Quality-Driven Model Transformations: In software product line development there are different scenarios where design alternatives appear impacting over the quality of products or software artifacts. Those scenarios require the definition of quality-driven model transformation, to enable the customers; designers or domain experts obtaining the products or software artifacts that best fit their quality attributes.

Some of the quality-driven model transformations processes to be defined can be integrated on the attached processes associated of the assets where the design alternatives appear as shown in Figure 29.a, and other will be transversal to the core assets (it may affect to different core assets, may substitute a core asset by its equivalents or may update the whole SPL architecture i.e. by applying architectural patterns) as shown in Figure 29.b.

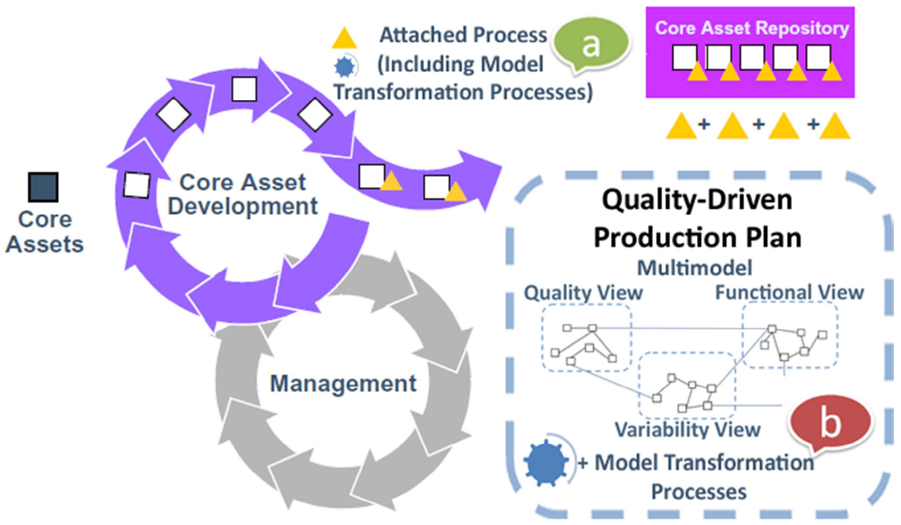


Figure 29. Integration of the Quality-Driven Model Transformations in the Core Asset Development Activity

7.1.1. Multi-model Definition

The starting point is the definition of the multi-model for reflecting the extension of the product line. Starting with an existing core-asset repository, a variability model expressed by using a cardinality-based features model and a functional model expressed by using an AADL specification, the variability and the functional views can be populated automatically by means of two model transformations that transform:

- The functional view of the software product line, expressed by means of an AADL specification into the functional view of the multi-model (see Figure 30.a). This transformation projects only the meta-classes that had been defined as subclasses of the meta-class *EComponentType*: *ESystemType*, *EProcessType* and *EThreadType*, since these are the meta-class that have the capability of impacting over the quality attributes and of being associated with the features of the variability view. This transformation also projects all the AADL features, AADL flows, the AADL hardware components that allow the deployment of those systems and its implementations.
- The Cardinality-Based Features Model into the variability view of the multi-model (see Figure 30.b).

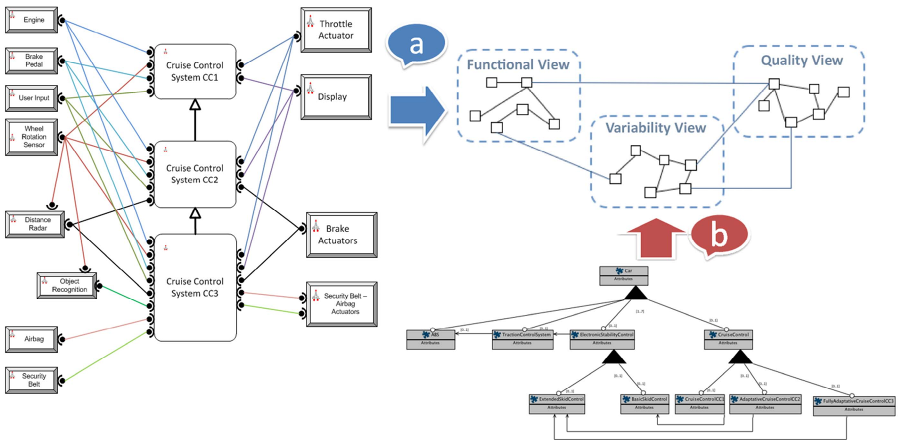


Figure 30. Transformation for Populating the Views

The next step is the quality view definition. The domain experts should define the quality view of the multi-model containing the quality decomposition in characteristics, sub-characteristics and the quality attributes identified as relevant in the domain.

The next step is the definition of the relationships among the variability and the functional view, performed by the domain experts and software engineers. This relationship expresses the combination of functional components for fulfilling the requirements of a feature or how a functional component can fulfill part of the requirements of more than one feature.

The next step is to associate the relative impacts that the different elements of the variability and functional view have over the quality attributes. This information comes from the knowledge of the domain expert and the software engineers, from empirical evidence obtained by experiments, or from the operationalization of metrics applied over the software artifacts. These impacts are expressed using the *Weight* attribute of the *Impact* class.

7.1.2. Definition of Quality-Driven Model Transformations

There are many scenarios in SPL development that may require the definition of quality-driven model transformation processes due to the presence of design alternatives which impact over the quality attributes of the target models of these processes (the product or the software artifacts generated).

There were two main groups of quality driven model transformations as stated in Section 6.1:

- Case1. Different functional components with the same interface, structure and behavior but different with regard to quality attributes: In this case the quality-driven model transformation cannot be defined as a part of the attached processes associated to each core asset, since it is transversal to many core assets, and should be executed once the blocks (or core assets) had been selected from the repository and had been deployed on the base architecture.

- o The domain expert must perform the following activities:
- o Identify the equivalent blocks.
- o Identify the impact of the selection the different functional components over the quality attributes.
- o Build the transformation model. In the transformation model a block with equivalences is created as an instance of the *StructuralPattern* meta-class. Each alternative is then defined as instances of the *Alternative* meta-class and the rule/s substituting a block by its alternatives are defined as instances of the *Rule* meta-class. Finally each *Rule* is associated with its *alternative*.
- o The result off the trade-off analysis is used to fulfill the weights of the different alternatives, representing the relative impact that each alternative has over the quality attributes.
- o The transformation engineer should adapt the transformation rules (or define it from the scratch) following the schema defined in section 6.3.
- o When executing the transformation, the software engineer selects the quality attributes in the quality model defined for this transformation process as stated in section 6.2.2, executes the transformation process and obtains a new architectural model with the functional components (or assets) that best fit the quality attributes selected.

- Case 2. The improvement of the quality of models (or software artifacts) during the development process, both in core asset and in product development activities: In this case, depending on the nature of the transformation process, the quality-driven model transformation can be defined as part of the attached process associated to the core asset, or if it is transversal to different core assets then it should be executed after the selection of core assets as in case 1.
  - o The domain expert must:
  - o Identify structural patterns in the source models with alternative transformations.
  - o Identify the impact of the different alternative transformations over the quality attributes.

- Build the transformation model. Each structural pattern with alternative transformations is created as an instance of the *StructuralPattern* meta-class, the alternative transformations for each structural pattern are created as instances of the *Alternative* meta-class, and all the alternative rules are created as instances of the *Rule* class. In addition, each rule is associated with the corresponding alternative transformation.
- The result off the trade-off analysis is used to fulfill the weights of the different alternatives, representing the relative impact that each alternative has over the quality attributes.
- The transformation engineer should adapt the transformation rules (or define it from the scratch) following the schema defined in section 6.3.
- When executing the transformation, the software engineer selects the quality attributes in the quality model defined for this transformation process as stated in section 6.2.2, executes the transformation process and obtains a model that best fit the quality attributes selected.

7.2. The Production Plan in the Product Development Activity

The production plan defined in this approach has two main phases, where the model transformation processes play an important role. In the first phase, a transformation process performs the product configuration, which takes as input the multi-model and assists the product engineer selecting the features and assets that best fit the quality attributes chosen in the multi-model 's quality view. Figure 31 illustrates the relationships among the quality-driven model transformations, the core assets and the attached processes.

In the second phase, after obtaining a valid product configuration a model transformation process generates the architectural model of the product. This architectural model can be optimized by transformation processes defined for substituting the equivalent functional components or assets or by the quality-driven model transformation processes defined as part of the attached processes associated to the different core assets.

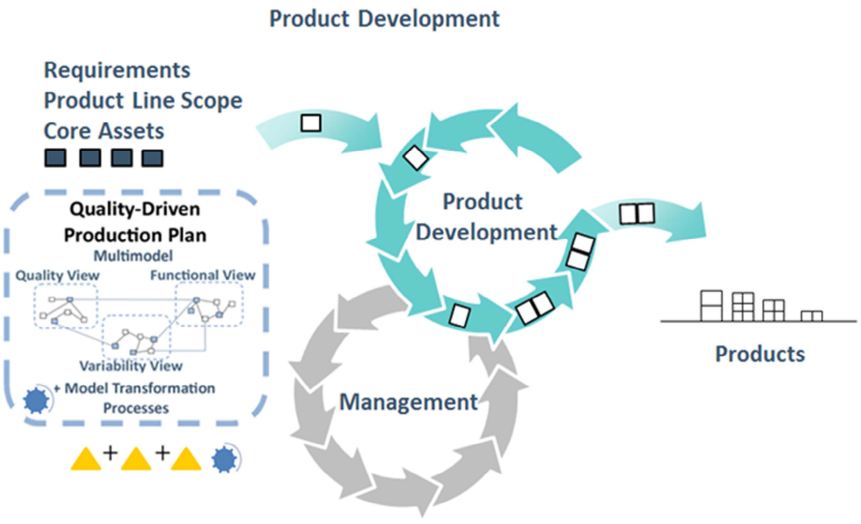


Figure 31. The Production-Plan in the Product Development

7.2.1. Product Configuration

The features and components that best fits the quality attributes chosen by product engineer are selected by using the AHP [3] trade-off analysis process, automated by an endogenous (the source and target models conforms to the same metamodel) model transformation process.

The product engineer selects the quality attributes by introducing weights on the Attribute meta-class. Those weights mean the relative importance that each quality attribute has in this product configuration. The higher a weight is, the more important this quality attribute is, under product engineer point of view.

The transformation process takes as input the multi-model, containing the selections of the product engineer over the quality view and modifies the multi-model, marking as selected the features and functional components that best fit those quality attributes as shown in Figure 32.

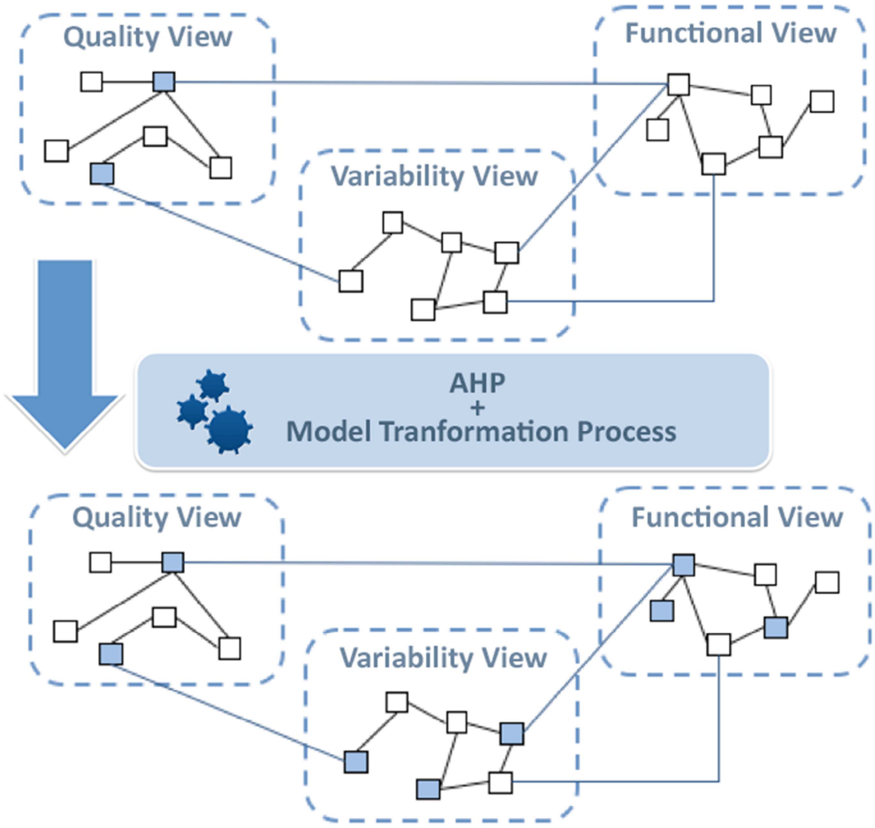


Figure 32. Quality-Driven Product Configuration

Depending on the nature of the Product Line this process can drive the whole selection of features and functional components. However there are different factors that affect to the completeness and satisfiability of this selection:

- The density of the multi-model: we can consider the density of the multi-model as the number of relationships that contains, first and most important, between quality attributes and features and second, between features and functional components. As more features in the variability view have impact over the quality attributes as more complete the process of selection will be.
- Invalid multi-model configurations: In the same way than the problems with invalid configurations in feature modeling, the impacts between the views can express product configurations that cannot be satisfied. For example because of impacts between qualities attributes. The quality view of the multi-model allows the definition of positive or negative impacts among quality attributes

(represented by the meta-class *AttributeImpact*), if two attributes, which have negative impacts between them, are selected to be maximized, then that configuration can hardly be satisfied.

When the trade-off can only make a partial selection of the features and functional components, product engineer should complete the product configuration manually. This process is performed by selecting, directly over the multi-model, the features (and/or components) that complete the functional and non-functional requirements of the product being configured.

The next step to be performed is to analyze if the selections made over the multi-model are a valid product configuration (syntactic) satisfying all the constraints expressed in the multi-model. This analysis can be performed by using model-checking tools as the one presented by Garcia Galan et al in [24], which allows verifying whether or not a feature model has valid product configurations by using constraint programing techniques.

Once the model checker confirms that the selections over the multi-model express a valid configuration, it is time for the product engineer to validate, under a semantic point of view, whether or not the multi-model contains a product configuration that meets all the functional and non-functional requirements, especially when the product configuration has been made fully automatically.

7.2.2. Architecture Generation and Optimization

After obtaining a valid configuration under the syntactic and the semantic point of view the next step is to generate, by means of a model transformation, the architectural models of that configuration. Figure 33 illustrates the profile of this transformation process, which takes as inputs the multi-model and the complete AADL specification containing architectural model of the software product line, and generates as output an AADL specification containing with the architectural model of that product configuration.

The purpose of merging the functional view of the multi-model and the complete architectural view reflects the need of completing the partial projection of the architectural model that the multi-model contains. As described in section 5.6 the relevant meta-classes for defining the multi-model's functional view were the *ESystemType*, the *EProcessType*, *EThreadType*. For completeness the functional view of the multi-model contains also the implementations, the hardware components where those components can be deployed, the AADL features and the AADL flows associated to those three types of software components. Therefore there are parts of an AADL specification that are out of the multi-model functional view, and this merge operation allows the specification to be completed. The output of this transformation process contains the architectural specification, in AADL of the product configured in the previous steps.

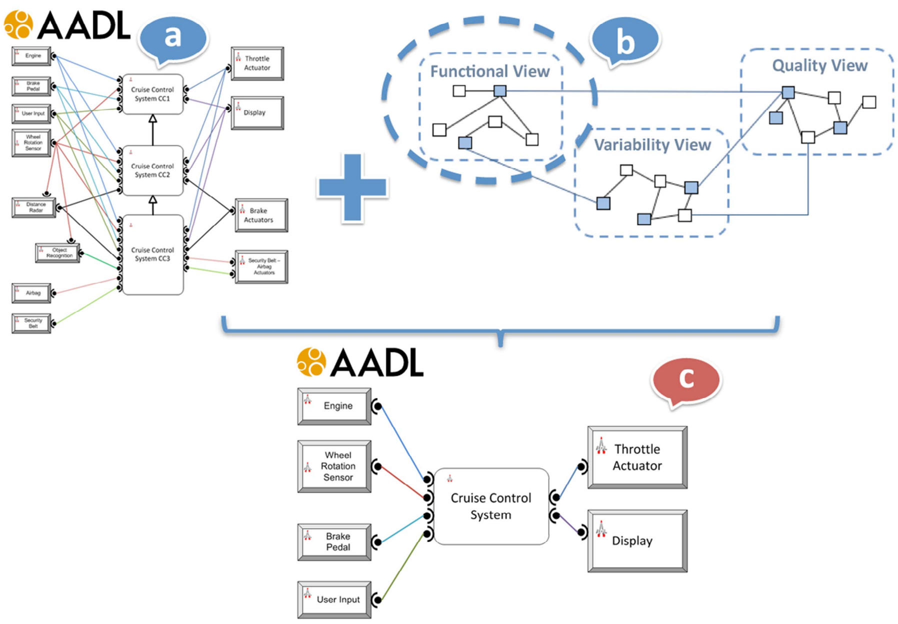


Figure 33. Generation of the architectural view of a product

The final step of the production plan, under the architectural point of view is the architectural improvement and optimization based on some quality attributes. The AADL specification containing the architectural models of that configuration is now the input for the quality-driven model transformation processes, which substitute some functional components by its equivalents, or applies architectural patterns or optimizes the deployment of the software components into the hardware components of the architecture.

Chapter 8. Case Study

In this chapter we illustrate the feasibility of the approach through a case study in the context of the Safety Critical Embedded Systems, from the automotive industry. The automotive industry is one of the domains where the product lines paradigm and in particular, the software product lines, is becoming more and more extended. It sounds natural to configure a car by selecting different features, as are the engine configuration, the onboard and safety equipment, the color and so on. This customization capability impacts directly over the complexity of the software that controls the car. Nowadays, when the number of software elements and computational nodes is rising exponentially, this customization capability tends to make the problem of building the control system a problem that without applying the software product line cannot be solved in an economical way. The case study describes a Vehicle Control System with different features, based in the example shown in [32] and extended applying the variation points described in [56].

The chapter is organized as follows: In Section 8.1 we introduce the Vehicle Control System this case study focuses on, describing the variability and functional views. In Section 8.2 we describe the steps followed to describe the multi-model for this case study. In Section 8.3 we illustrate how the selection of different quality attributes impacts on the selections of the different features and assets. Finally in Section 8.4 we enumerate the conclusions and lessons learned from this case study.

8.1. The Vehicle Control System

We have used the Vehicle Control System to verify the feasibility of our approach in areas as multi-model definition, product configuration and alternative identification. The Vehicle Control System contains different subsystems and features, as are Antilock Braking System (ABS), Traction Control System (TCS), Stability Control System (SCS), Cruise Control System (CC) and the Airbag and Security Belt Controllers. All these systems comprise a set of embedded systems present in modern automobiles. In general, those systems rely on capturing input signals from sensors, making some computation of those inputs based on specific control laws, and sending the outputs, a control value, to a actuator that affect the state of different subsystems of the car (engine, throttle position, brakes, security belts etc.).

The Goal of the *Antilock Braking System (ABS)* is to ensure that maximum braking force is transmitted to all four wheels of the vehicle, even under adverse conditions such as skidding on rain, snow or ice. Antilock brakes work by sensing slippage at the wheels during braking and adjusting brake pressure to ensure maximum contact between the tires and the road. In the most basic version, wheel rotation sensors from all four wheels are used as input and the output is the brake valve on each of the four brake lines.

The Goal of the *Traction Control System (TCS)* is to avoid wheels to slip. TCS deals with the front to back loss of tire to road friction during acceleration. The traction control system uses the data from the rotation sensor of each wheel of the vehicle, compares the rotation data with the speed to detect slipping wheels, and compensates these slipping wheels by reducing the speed of the wheels. This is achieved by applying individual braking force to the slipping wheel or by reducing the power of the engine via the throttle control to ensure maximum contact between the road surface and the tires, even under less-than ideal road conditions, such as ice or snow.

The goal of the *Stability Control System (SCS)* is to keep the vehicle going in the direction in which the driver is steering the car. To achieve this, the stability control system applies the brake to one wheel (or passes the torque to the opposite one) to help steer the car in the correct direction. If poor traction causes the front end of the car to slip sideways when you are going around a corner, the control laws will cause the brakes to be applied on the inside wheels of the corner (and/or apply more proportion of the torque to the outside wheels), causing the car to turn and slow down. If the back end of the car slips sideways, the brake on the wheel that is outside of the corner is applied to bring the car back into line The system works when the car starts to slide on a straight road an when turning corners. The SCS takes information from different sensor and then determines whether the car is a stable or unstable state. By combining the data from wheel rotation sensors, steering angle sensors, yaw sensors that measure the amount of sideways g-force the car is suffering, the central processing unit can actually detect whether or not the car is following the driver intends. The control system then applies the brakes or changes the distribution of torque among the traction wheels to counteract the destabilizing force. In some cases the engine speed may also be reduced. The SCS differs from the TCS in what both systems prevent. A Traction Control System acts on a vehicle’s traction wheels to prevent unwanted wheel spin under acceleration, which is producing slipping effect. The SCS, on the other hand, goes one step further by detecting when a driver has lost some steering control over the car’s trajectory. It then automatically stabilizes the vehicle to help the driver regain control of the vehicle.

The goal of the cruise control system (CC) is to maintain a constant speed as determined by the driver. The system is in effect between some minimum and maximum speeds (e.g., 40 Km/h to 120 MPH). The cruise control system maintains the vehicle speed at the predetermined value (target value) by storing the speed of the wheel rotation when the speed value is set and attempts to keep the throttle actuator at a position to maintain the vehicle speed at the target value. As the road inclination changes, the vehicle speed changes, and the throttle position should change to maintain the vehicle speed. The control system observes the speed difference between the current speed and the target value and either decreases or increases the throttle actuator position to counteract the speed differential. The algorithm to accomplish this is called the control law.

This case study presents three variants of Cruise Control System: The Basic Cruise Control System, an Adaptive Cruise Control System and Fully adaptive Cruise Control System:

- The goal of the Basic Cruise Control is to maintain a constant vehicle velocity as determined by the driver (target speed).
- The goal of the Adaptive Cruise Control System is to extend the Basic Cruise Control and provide a new function of constant-distance cruise control that maintains the distance to a target vehicle traveling in front of the vehicle. In order to ensure constant-distance cruise control, this system includes a radar sensor that yields the following information: the distance from the equipped vehicle to the target vehicle and the relative speed between these two vehicles.
- The goal of the Fully Adaptive Cruise Control System is to extend the Adaptive Cruise Control, by adding an image sensor that captures the behavior of vehicles and objects in front of the system, and extends the effective range of the radar sensor and is capable to stop the vehicle in case of imminent collision.

In all of the above systems, signals regarding overall engine and vehicle state (e.g. engine on/off, brake pedal on/off) are also considered in each control subsystem. These signals are used to ensure proper operation of each subsystem. For example, if the brake pedal is depressed, the cruise control system should disengage (or not become active) and the traction control system should not become active. There are also outputs from each system used as feedback to the driver that each system is on. These outputs can be indicating lights or some form of intelligent operator display (e.g., LCD panel). Moreover some of the systems described above share some of the input and output devices, for example the SCS and the TCS that obtain information coming from the ABS wheel sensors and apply its outputs to the braking systems. All those systems are represented in Figure 34. The elements on the left side represent the different sensors and input components that feed the controllers. The elements on the right side represent the output devices that receive control information, as the brake or throttle actuators, or display devices that show the status information to the driver.

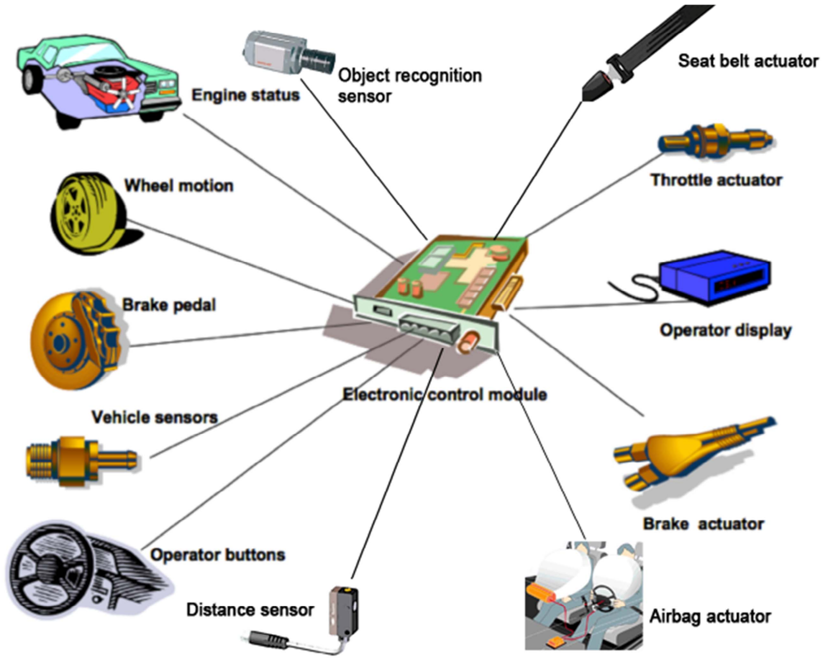


Figure 34. Main Components of the Vehicle-Control System

Figure 35 shows the Cardinality Based Features Model that describes the variability of the Vehicle Control System, showing only the features with regard to the systems and subsystems described in this case study. There are five main features, four of them that are optional and one (the airbag) that is mandatory. There are features that hold *implies* relationships with other features, since they share sensing devices or infrastructures (as in case of TCS to ABS or SCS to TCS).

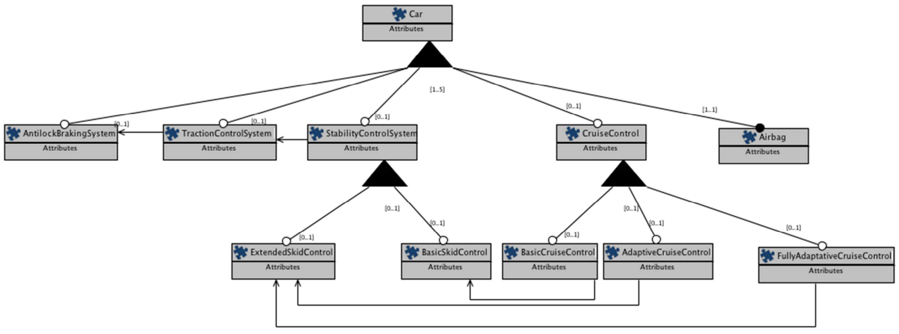


Figure 35. Vehicle Control System Features Diagram

The next subsection describes with more detail the different versions of the Cruise Control System, showing its main features, the devices and systems with which interacts and an excerpt of the implementation diagrams. The whole AADL specification of the Vehicle Control System can be found in Annex I. The Annex II contains the AADL specification of the different versions of the Cruise Control.

8.1.1. Basic Cruise Control

The system acts only between a minimum and maximum speeds (e.g. 40km/h to 120 km/h). The cruise control system maintains the vehicle speed at the target speed by noting the wheel's rotation speed and keeps the throttle actuator position to maintain the vehicle speed to this target value. As the road conditions changes, the vehicle speed is affected, and the throttle position changes to maintain the vehicle speed in accordance with the user target speed value. The cruise control system renders the difference between the current vehicle speed and the target speed and either decreases or increases the throttle actuator position to counteract the speed differential.

The Components that provide inputs signals to the system are:

- **The On/Off switch:** the user can activate or deactivate the system by pressing this switch
- **Engine Status:** any alarm detected in the engine automatically disengages the cruise control.
- **Brake Pedal Status:** If the brake pedal is pressed by the driver, while the cruise is active, then the system is automatically disengaged
- **Resume Button:** The system can be engaged again after being disengaged by the brake pedal through the resume button.
- **Decrease speed and increase speed:** Once the system is working the driver can vary the target speed value by pressing the increase speed or decrease speed buttons.
- **The Set Button:** The driver can establish the target speed by pressing the set button, the system automatically stores the actual speed as the new target speed of the cruise control system.
- **Wheel Rotation Sensor:** The system reads the wheel rotation pulse (one pulse per rotation) for calculating the actual speed of the car.

The system has only one main output[2], the throttle actuator. This version of the cruise control is only capable of controlling the speed by controlling the

[2] The display devices are not shown nor in the diagrams nor in the AADL specification in order to make the example more easy to understand

throttle; it has not control over the braking system. The Figure 36 shows the context diagram of the basic cruise control version, with the related component and the inputs and outputs described above.

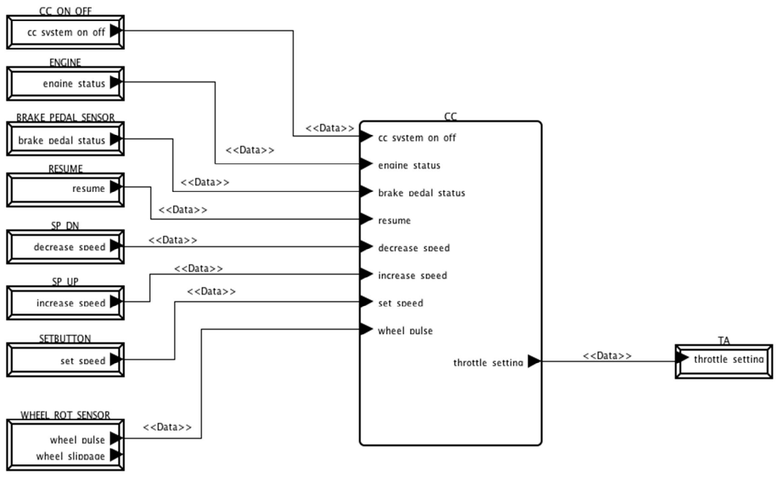


Figure 36. Basic Cruise Control System Context Diagram

The Figure 37 shows the implementation diagram of the basic cruise control. The system has four subcomponents (three processes and one device):

- **In Control (IC):** this process calculates, based on the different status sygnals, the values of two signals: *ok_to_run* that indicates whether or not the system is engaged and the *selected*_speed that contains the target speed value.
- **Calculate Velocity (CV):** this device calculates the value of *instantaneous_velocity*, the current speed of the car, taking as input the wheel rotation pulse.
- **Compute Desired Speed (CDS):** this process renders the difference, as a relative value, between the actual speed (*instantaneous_velocity*) and the target speed and by applying the control law. The value calculated in the previous iteration is used for detecting when the system has achieved the steady state.
- **Compute Throttle Setting (CTS):** This process converts the relative speed into a throttle position (*throttle_setting*) for the throttle actuator.

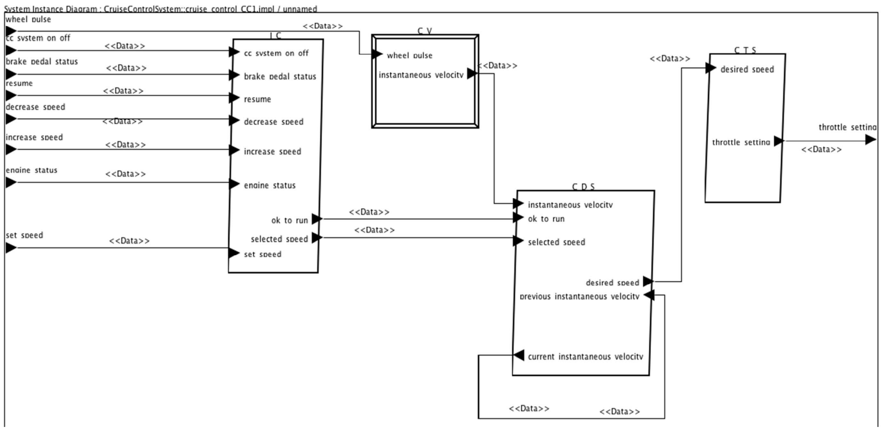


Figure 37. Basic Cruise Control System Implementation Diagram

8.1.2. Adaptive Cruise Control System

The Adaptive Cruise Control System extends the definition of the Basic version by providing a new function of constant-distance cruise control that maintains the distance to a target vehicle traveling in front of the vehicle. In order to ensure constant-distance cruise control, this system includes a radar sensor (distance sensor) that provides the system the distance from the equipped vehicle to the target vehicle and the relative speed between these two vehicles. The speed limits for operation are the same as the basic version (40-120Km/h).

Figure 38 shows the AADL context diagram of the Adaptive version of the cruise control. In this diagram the red boxes show the additional components and signals that this version needs for providing the new functionality. The new components are:

- **Set Distance Button:** allows the driver to fix the distance value he wants to maintain with the target vehicle.
- **Distance Sensor:** Calculates the actual distance between the car equipped with the sensor and the target vehicle and the relative speed between them.
- **Braking actuators:** This version of the cruise control is capable of controlling the speed by controlling the throttle position or by applying the brakes.

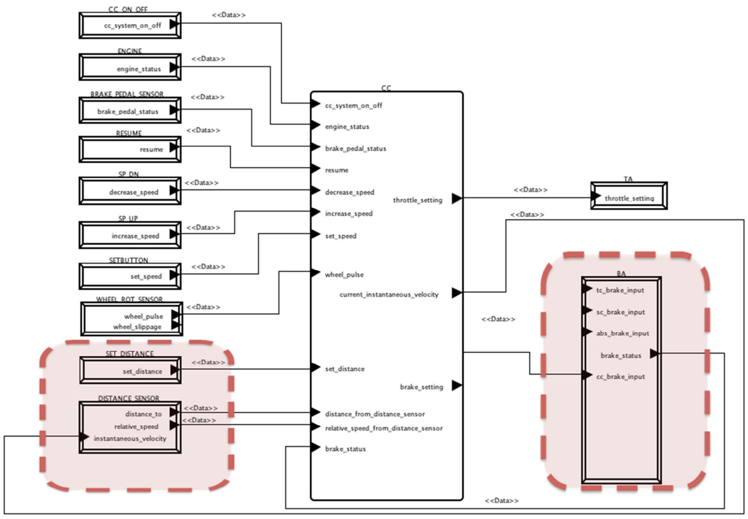


Figure 38. Adaptive Cruise Control System Context Diagram

Figure 39 shows the implementation diagram of the adaptive cruise control. The system has five subcomponents (four processes and one device). In this diagram the red boxes show the implementation changes from the previous version. There is a new component **Compute Braking Setting (CBS)** process converts the relative speed (when it is a negative value) into the braking force value to be sent to the braking actuators. The **In Control (IC)** has been extended into a new version (*In_Control_Ext*) that calculates not only the *ok_to_run* and the value the *selected_speed* but also generates the *selected_distance* signal containing the distance that should be maintained with the target car. The **Calculate Desired Speed (CDS)** it has been also extended for calculating the relative speed based also in the relative speed to the previous car.

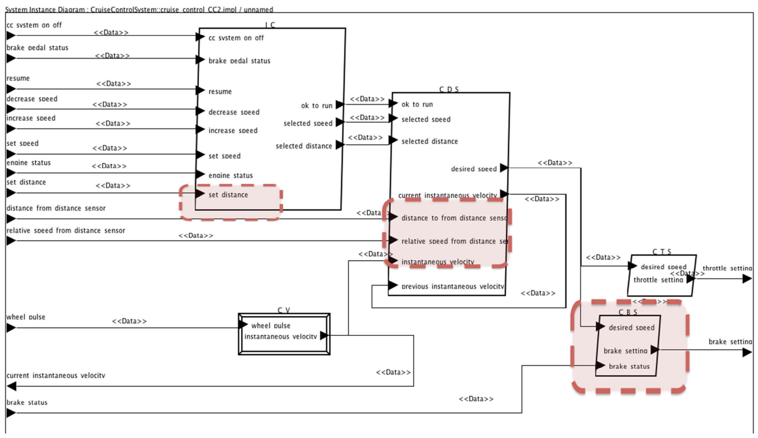


Figure 39. Adaptive Cruise Control System Implementation Diagram

8.1.3. Fully Adaptive Control System

The Fully Adaptive Cruise Control system is a further extended system derived from Adaptive Cruise Control. In addition to the radar sensor, this system includes an image sensor labeled Object Recognition Sensor. The object recognition sensor captures the behavior of vehicles in front of the system-equipped vehicle. It works together with the radar sensor in a compensatory way to expand the effective range of sensing the distance and relative speed. This alleviates the speed limits for operation, such as 0 km/h–140 km/h. This wide-range sensing enables a stop-and-go function by following on the target vehicle and adds the capability of stopping the vehicle in case of imminent collision. When a imminent collision is detected, the system sends signals to the airbag and security belt controllers in order to, first pretense the security belt and after to fire the airbag system.

Figure 40s hows the AADL context diagram of the Fully Adaptive version of the cruise control. In this diagram the red boxes show the additional components and signals that this version needs for providing the new functionality. The new components are:

- **Objet Recognition Sensor:** Extends the distance sensor capabilities, being able to detect obstacles that can cause a collision and expanding the range of sensing distance. Provides the same kind of signals than the distance sensor (distance to previous car and relative speed).
- **Airbag and Security Belt Status:** The status of that airbag and security belt controllers needs to be taken into account since the system now interacts with them.
- **Airbag and Security Belt Actuators (ASA):** As described above, this version of the cruise control is capable of controling the security belt and the airbag.

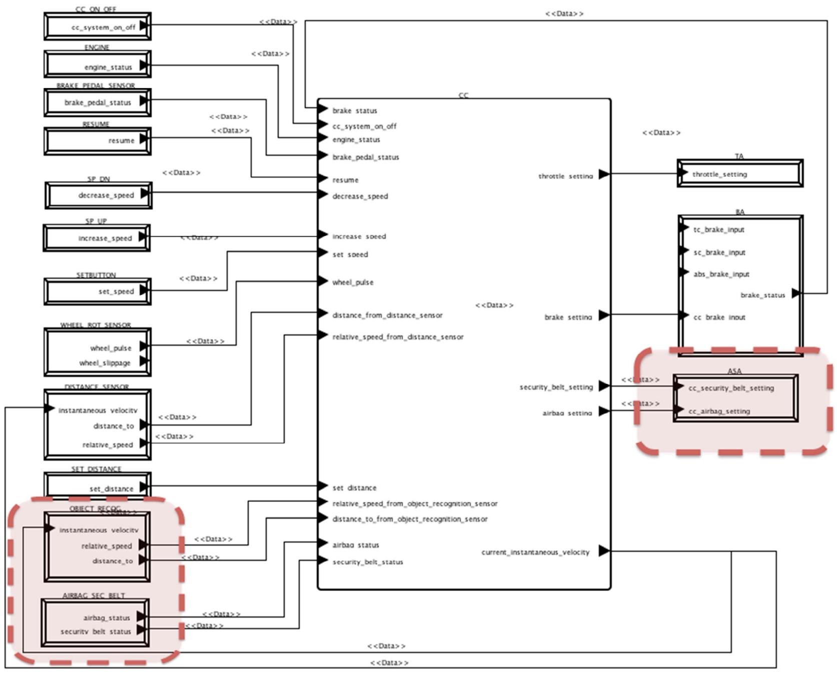


Figure 40. Fully Adaptive Cruise Control System Context Diagram

Figure 41 shows the implementation in AADL of the fully adaptive cruise control. Due to the complexity that the implementation diagram has, we have decided to show only an excerpt of that definition in the AADL language. The system has now two additional component **Compute Airbag Setting (CAS),** and the **Compute Security Belt Setting (CAS),** which is capable to calculate, based on the relative speed and the distance to an object the need of activating the airbag system or the security belt, respectively, before an imminent collision. The **Compute Desired Speed (CDS)** (*compute_desired_speed_CC3*) has been extended into a new version for calculating the relative speed based also in the information of relative speed and distance to the previous car coming from the object recognition sensor.

```
 system implementation cruise_control_CC3.impl
   subcomponents
     I_C: process in_control_ext;
     C_V: device compute_velocity;
     C_D_S: process compute_desired_speed_CC3;
     C_T_S: process compute_throttle_setting;
     C_B_S: process compute_brake_setting;
     C_A_S: process compute_airbag_setting;
     C_S_S: process compute_security_belt_setting;
   connections
     C1: data port cc_system_on_off -> I_C.cc_system_on_off;
     C2: data port brake_pedal_status -> I_C.brake_pedal_status;
     C3: data port engine_status -> I_C.engine_status;
     C4: data port resume -> I_C.resume;
     C5: data port decrease_speed -> I_C.decrease_speed;
     C6: data port increase_speed -> I_C.increase_speed;
     C7: data port set_speed -> I_C.set_speed;
     C8: data port wheel_pulse -> C_V.wheel_pulse;
     C9: data port I_C.ok_to_run -> C_D_S.ok_to_run;
     C10: data port C_V.instantaneous_velocity -> C_D_S.instantaneous_velocity;
     C11: data port C_D_S.current_instantaneous_velocity -> C_D_S.previous_instantaneous_velocity;
     C12: data port C_D_S.desired_speed -> C_T_S.desired_speed;
     C13: data port C_T_S.throttle_setting -> throttle_setting;
     C14: data port distance_from_distance_sensor -> C_D_S.distance_to_from_distance_sensor;
     C15: data port relative_speed_from_distance_sensor -> C_D_S.relative_speed_from_distance_sensor;

             ...

    relative_speed_from_distance_sensor_flow_1: flow path relative_speed_from_distance_sensor -> C15 -> C_D_S.FS5
       -> C19 -> C_B_S.FS1
       -> C20 -> brake_setting;
    set_distance_flow1: flow path set_distance -> C35 -> I_C.FS6
       -> C36 -> C_D_S.FS11
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
end cruise_control_CC3.impl;
```

Figure 41. AADL Implementation of the Fully Adaptive Cruise Control

8.2. Definition of the Production Plan for the Vehicle Control System

This section illustrates the creation of the production plan for the system based on the definition described in the previous section.

8.2.1. Multi-model Definition

As explained in the previous chapter the definition of the production plan starts with de definition of the multi-model. The starting point of this process is the AADL specification of the system (see Annex I and II) the cardinality based features model that is shown in Figure 35. Vehicle Control System Features Diagram. The variability and functional views are going to be populated automatically with two transformation processes that take as input the AADL specification and the features model respectively.

The quality view should be created with the quality characteristics, sub-characteristics and attributes. In this case study we had included some additional quality attributes identified as relevant by domain experts for Safety

Critical Systems context in [58], applicable to core assets and which should be taken into account during product configuration process, as are:

- **Maintainability:** *"The ease with which a product or a core asset can be modified to correct faults, improve performance, or adapt to a changing environment."*
- **Maturity:** *"The degree to which a core asset is free from further modification."*
- **Reusability:** *"The degree to which a core asset can be used in more than one software system, or in building other core assets."*
- **Variability:** "The ability of a system, an asset, or a development environment to support the production of a set of artifacts that differ from each other in a pre-planned fashion".
- **Testability: "**The ease with which a product or a core asset can be made to demonstrate its faults through (typically execution-based) testing."

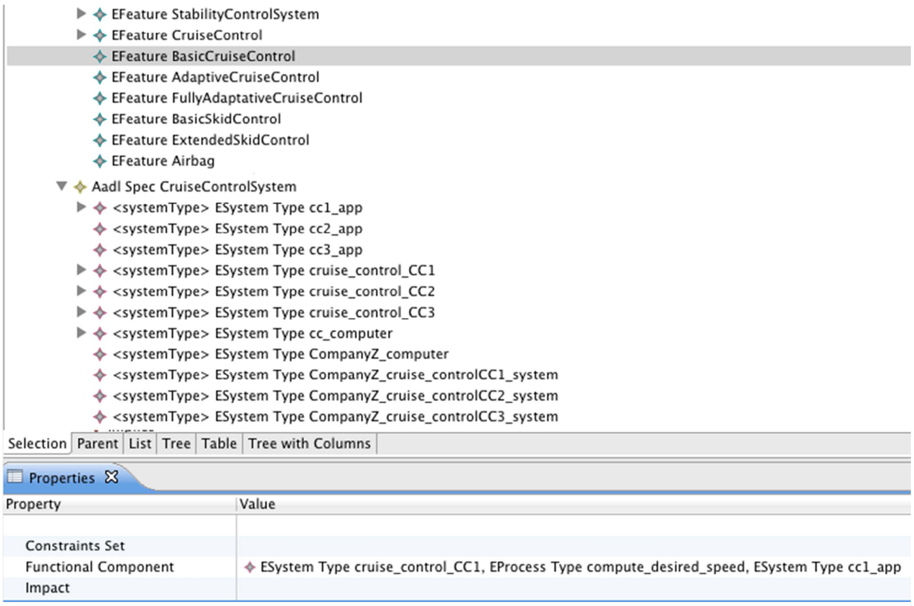
We have also included some quality attributes of the *Quality-In-Use* perspective, as is the *Driving Security*, in order to illustrate how the product configuration can be, from the customer point of view.

The next step is to define the relations among features and functional components. These relations should be defined by associating functional components that fulfill the requirements of the different features. Main components that fulfill the requirements of the feature and/or those that can impact over the quality attributes should be included in these relationships. Depending on the level of detail to be achieved the number and type of the functional components in those relationships may vary. Table 4 shows an excerpt, in tabular representation, of the relationships between the variability and the functional view.

Feature	Functional Components
Basic Cruise Control	Cruise Control CC1
	Cc1_app
	Compute desired speed
	In Control
Adaptive Cruise Control	Cruise Control CC2
	Cc2_app
	Compute desired speed
	In Control Ext
	Set Distance
	Distance Sensor
Adaptive Cruise Control	Cruise Control CC3
	Cc3_app
	Compute desired speed
	In Control Ext
	Set Distance
	Distance Sensor
	Object Recognition Sensor
Airbag	Airbag Controller
ABS	Brake Actuators
	Antilock Brake System
Stability Control	Throttle Setting
	Stability Control System
Traction Control	Throttle Setting
	Traction Control System
...	...

Table 4. Relationships between Functional and Variability Views

Figure 42 shows the Eclipse-based editor implemented for supporting the multi-model definition and the mechanism for defining the relationships among features and functional components. The figure shows that the feature *BasicCruiseControl* is selected, and the properties view on the bottom shows the functional components that had been associated.



EFeature StabilityControlSystem
EFeature CruiseControl
EFeature BasicCruiseControl
EFeature AdaptiveCruiseControl
EFeature FullyAdaptativeCruiseControl
EFeature BasicSkidControl
EFeature ExtendedSkidControl
EFeature Airbag
Aadl Spec CruiseControlSystem
<systemType> ESystem Type cc1_app
<systemType> ESystem Type cc2_app
<systemType> ESystem Type cc3_app
<systemType> ESystem Type cruise_control_CC1
<systemType> ESystem Type cruise_control_CC2
<systemType> ESystem Type cruise_control_CC3
<systemType> ESystem Type cc_computer
<systemType> ESystem Type CompanyZ_computer
<systemType> ESystem Type CompanyZ_cruise_controlCC1_system
<systemType> ESystem Type CompanyZ_cruise_controlCC2_system
<systemType> ESystem Type CompanyZ_cruise_controlCC3_system

Property	Value
Constraints Set	
Functional Component	ESystem Type cruise_control_CC1, EProcess Type compute_desired_speed, ESystem Type cc1_app
Impact	

Figure 42. Definition of the Relationships in the Eclipse-Based Multi-Model Editor

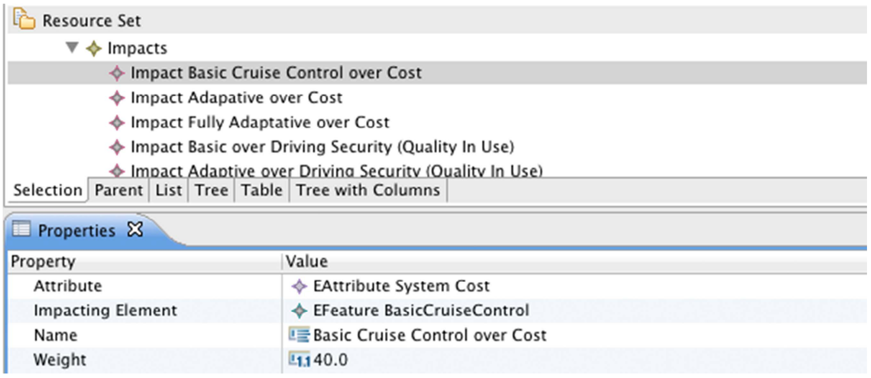
The next task is to associate the impacts that each element has over the quality attributes. The Table 5 shows an excerpt, in a tabular representation, of the main impacts detected. Those impacts must be defined according to the knowledge of domain experts or to the results of empirical evidence obtained experimentally. In this case those values reflect, approximately, what the impacts of the different elements would be.

The *Weight* value expresses how this impacting element increases the quality of the product, for positive quality attributes (as driving security), higher values implies better quality. In the cases of negative attributes that should be minimized (as cost) high levels of the *Weight* attribute mean poor quality. The impacts could be defined as the relationship of more than one *ImpactingElement* over a quality attribute, meaning that the combination of some elements may have impact (negative or positively) over the quality attributes. This allows us to express more complex relationships that occur when dealing with the selection of features or components. In general the quality of the final product can be more (or less) than the addition of the quality of the parts, and this should be modeled and taken into account during the trade off analysis that will perform the product configuration.

Impacting Element	Quality Attribute	Artifact/Phase	Weight
Basic Cruise Control	Cost	Feature/Product Development	40
Adaptive Cruise Control	Cost	Feature/Product Development	30
Fully Adaptive Cruise Control	Cost	Feature/Product Development	20
Traction Control System	Cost	Feature/Product Development	70
Basic Cruise Control	Driving Security (Quality In Use)	Feature/Product Development	50
Adaptive Cruise Control	Driving Security (Quality In Use)	Feature/Product Development	70
Fully Adaptive Cruise Control	Driving Security (Quality In Use)	Feature/Product Development	90
Traction Control System	Driving Security (Quality In Use)	Feature/Product Development	70
Cruise Control CC1	Maturity	Asset/Asset Dev.	95
Cruise Control CC2	Maturity	Asset/Asset Dev.	90
Cruise Control CC3	Maturity	Asset/Asset Dev.	80
Cruise Control CC1	Maintainability	Asset/Asset Dev.	90
Cruise Control CC2	Maintainability	Asset/Asset Dev.	70
Cruise Control CC3	Maintainability	Asset/Asset Dev.	60
...	...	...	...

Table 5. Excerpt of the Main Impacts Detected

Figure 43 shows the impact definition, in the Eclipse-Based editor, of the impacts. The selected item represents the relative impact that the feature *BasicCruiseControl* has over the final cost of the system. Although the *BasicCruiseControl* is the cheapest version of the cruise control, it is expensive enough for having a significant negative impact over the cost (40 value).


Resource Set
Impacts
Impact Basic Cruise Control over Cost
Impact Adapative over Cost
Impact Fully Adaptative over Cost
Impact Basic over Driving Security (Quality In Use)
Impact Adaptive over Driving Security (Quality In Use)

Property	Value
Attribute	EAttribute System Cost
Impacting Element	EFeature BasicCruiseControl
Name	Basic Cruise Control over Cost
Weight	40.0

Figure 43. Definition of impacts in the Multi-Model Editor

Once the multi-model has been defined the next step is to detect the model transformations that have alternative model transformations.

8.2.2. Alternative Identification and Definition of Quality-Driven Model Transformations

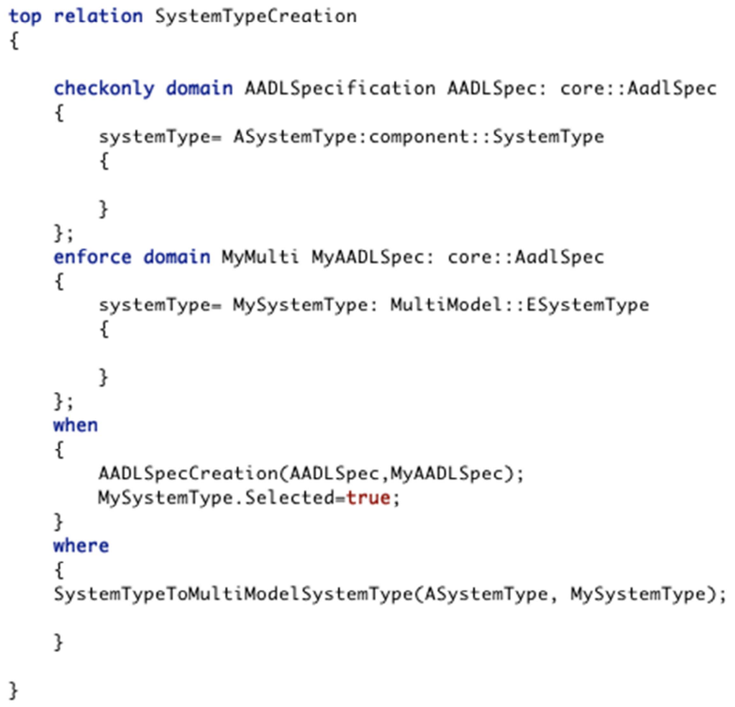
In the scenario being presented, there are not visible alternative transformations. The different variants of the system described are user-visible and then must be classified as features and modeled in the variability view. However in the context of the embedded systems development its quite common to have design decisions that will impact over the quality attributes of the final product. For example equivalent software components that have the same structure, the same interface and exactly the same functionality, but that are different with regard to attributes as cost, memory consumption or CPU usage. The second example of alternative transformations is the Deployment Process of the architecture, where the different software components are assigned to hardware-processing and communication elements [4]. Both those kind of transformation processes can be built by first detecting the alternatives and identifying the impact over the quality attributes as described in Section 6.1, then by instantiating the architecture defined in Section 6.2 and finally by defining the transformation process following the step described in Section 6.3.

8.3. Product Configuration

The Product configuration process takes as input the multi-model and based on the selections made over the quality view (in terms of values introduced in the *Weight* in the *EAttribute* meta-class of the multi-model) chooses the features and functional components that satisfy the quality attributes selected. This process applies a model transformation which implements the trade-off among attributes and then selects the features and assets by changing the value of its *Selected* attribute to true.

In our example if we want to obtain a product where the driving security is relatively more important than the cost then the features selected will include the Fully Adaptive version of the cruise control. However if we want to configure a product in which the cost is the most important quality attribute and the driving security is not important, then the features selected will not include a version of the cruise control. In the case of attributes related to Assets/functional component the selection must be performed first selecting those assets and then including the features that, having a relationship with those assets, satisfy the other quality attributes selected. In our example if we select to obtain a product configuration that include the more mature core assets, then the selection it will include the component *CruiseControlCC1* and then the feature/s with which is associated, in our case the basic version of the cruise control.

Once the product configuration has chosen the features and functional components that best fit the quality attributes selected, we can extract the architectural view of the configured product by applying a model transformation. This transformation is a variation of the model transformation that populates the functional view of the multi-model, taking profit of the directionality of the QVT-R transformations. Figure 44 shows the structure of a transformation rule that extracts the *SystemType* entities (Systems defined in an AADL specification) selected during the product configuration (*Selected* attribute equals to true).



```
top relation SystemTypeCreation
{

    checkonly domain AADLSpecification AADLSpec: core::AadlSpec
    {
        systemType= ASystemType:component::SystemType
        {

        }
    };
    enforce domain MyMulti MyAADLSpec: core::AadlSpec
    {
        systemType= MySystemType: MultiModel::ESystemType
        {

        }
    };
    when
    {
        AADLSpecCreation(AADLSpec,MyAADLSpec);
        MySystemType.Selected=true;
    }
    where
    {
    SystemTypeToMultiModelSystemType(ASystemType, MySystemType);

    }

}
```

Figure 44. Example of a Model Transformation Rule for Extracting the Architectural View

Finally the architectural view obtained is the input model for the transformations defined following the schema described in Section 8.2.2. First we should execute the transformation that may substitute the equivalent components and then execute the architectural deployment optimizations.

8.4. Discussion

The Car Control System has allowed us to define the multi-model with the different views and then establish the relationships among them and finally define the impacts over the quality attributes that are relevant in the context of safety critical embedded systems.

We have had the opportunity of study some examples where the selection of different quality attributes influences the selection of features and or core assets during product configuration.

The example had also help us on analyzing the different scenarios where appear design alternatives influencing the quality attributes of the software artifacts.

Finally it had been useful to analyze the appropriateness of transformation processes that populate the functional and feature views and that extract the architectural models of the product once the product configuration concludes.

In summary, the main lesson learned that we can extract from this case study is that the approach allows us to introduce the quality as an additional view in the Software Product Line in order to define the impact that the different design decisions have over the quality attributes and then using those quality attributes as a decision factor in decision-making processes.

Chapter 9. Conclusions and Future work

The main motivation of this research work is the lack of coverage that SPL development gives to the identification of the impacts that the different design decisions have on the product quality attributes. This lead to not taking these impacts into account during the different decision-making processes that occurs during the SPL development process. The majority of the approaches consider only one view of the system (for managing the commonalities and variability of the product line) and the quality attributes that the products must fulfill are not considered during the product configuration and they are not supported by the existing production plans.

9.1. Conclusions

In this Master's Thesis we have achieved the following objectives defined for this research work:

9. Study the approaches that try to cover the problems that arise when developing SPLs in the domain of Safety Critical Embedded Systems, paying special attention to approaches trying to apply multi-view or multi-modeling techniques, or that introduces quality in model transformation processes.
10. Study the different standards that can be used for expressing each one of the different views. Identify the relationships among the elements of the different views. Define a multi-model that is able to express these relationships. Populate the multi-model with a set of quality attributes that has been identified as relevant for the domain of Safety Critical Embedded Systems.
11. Analyze the alternative transformations, which can be present in the PIM-To-PIM, and PIM-To-PSM model transformations that occur within of the two phases of the SPL development and its possible impact over the quality attributes.
12. Define a set of artifacts (mainly composed of model and metamodels) and a process, which allows the definition, and execution of model transformations in which the selection of alternative transformations is make based on quality attributes.
13. Define how to integrate the quality view in the different activities of the production plan. Establish which are the activities that affect to a single asset and which ones are transversal to the core assets

(affect to different assets or should be taken into account after the product has been configured).

14. Illustrate the feasibility of the approach in case study in the domain of safety critical on the automotive industry.

With regard to the first objective, we have analyzed the different approaches trying to apply multi-view or multi-modeling techniques realizing that the majority of the approaches do not integrate the quality view as one of the main views of the system. The approaches trying to introduce quality attributes in model transformation processes covers only a specific set of model transformation or only a reduced set of quality attributes.

With regard to the second objective we had analyzed the different standards for expressing the variability, the functional and the quality views,and we have select AADL for the functional and architectural view, the cardinalty based features models for the variability view and an SQuaRE based quality model for the quality view. We have defined a multi-model that allows us to join the three views and stablish the relationships needed among views. The multi-model has been implemented in Eclipse by using the Eclipse Modelling Framework. In addition we have defined, in QVT-Relations, the transformations that populate the variability and the functional views.

With regard to the third objective, we have analize the different model transformations where alternatives would appear. In SPL development there are two main groups of model transformations where alternative model transformations may exists: different assets, which having the same structure, interface and behavior are different with regard to quality attributes and the improvement of the quality of models (or software artifacts) during the development process, both in core asset and in product development activities. We have also analized which ones could be considered transversal to many core assets and must be executed after the core assets had been deployed over the SPL architecture.

With regard to the fourth objective, we have defined an architecture composed of a set of artifacts and a process to support quality-driven model transformations in automated transformation processes, improving traditional model transformation strategies by ensuring the quality of the software artifacts obtained as a result of applying the proposed process.

With regard to the fifth objective, we have defined the process for integrating the multi-model in the production plan, we have described how to define the quality driven model transformations, we have defined the process that must be followed for performing the product configuration based on the quality attributes that the product must fulfill and we have defined the process for obtaining the software architecture model in AADL containing the assets deployed by the product configuration process. This architectural model can now be obtained by means of the inverse of the QVT transformation defined for populating the model, taking profit of the bidirectional capability of QVT-R transformations. This architectural model can be the input of a quality-driven model transformation that applies architectural patterns to this model trying to improve the quality attributes selected.

With regard to the sixth objective, After studying some case studies in the domain of safety critical embedded systems that deal with automotive examples [31], [32], [65] we have defined a example putting together all the aspects found in the different case studies. Finally, the case study has been modeled entirely in order to allow us to analyze the feasibility of the approach with a complex and complete example.

9.2. Future Works

The work presented constitutes the first steps toward the definition of a fully automated production plan for software product lines regardless to the domain. However, some of the points of our contribution need future works to be completed or improved.

The first line of research is to improve our analysis of the existing approaches by performing a systematic literature review for analyzing the state of the art in the topics that had been informally analyzed in this work.

The second line of research is to improve the production plan. In the short term by implementing the complete the product configuration process that selects the features and the assets based on the quality attributes selected, and the transformation process that extracts the software product being configured by combining the functional view and the original SPL architecture expressed by means of an AADL specification. In our proposal this transformation only is capable to extract the entities contained functional view, but since this view it is only a partial projection, containing only the relevant entities of the architectural view. This partial projection needs to be decorated with the extra information contained in the original specification. In the medium term we plan to extend the definition of the production plan in order to obtain the configuration as a chain of model transformations. This requires a more precise definition of what is a production plan, what is a delta (the increment that each core asset represents) and how the attached processes can be defined in terms of model transformations.

The third line of research deals with the multi-model definition. We need to analyze if the relationships defined are enough for expressing the possible connections among views regardless to the domain. We plan also to analyze if additional views (and the relationships with the existing ones) added to the multi-model will allows us to describe the SPL in a more precise way, i.e., by including a risk view for describing the uncertainty associated to core assets or features of the variability view.

The forth line of research is to integrate our architectural model with other architectural optimization tools, as the one being developed at the LIACS. This tool nowadays allows the optimization of the deployment of software architecture in the context of embedded systems. Integrating both approaches will add flexibility to their tool, allowing the user to configure a product and then to obtain the optimal deployment for this software architecture.

The fourth line of research is to empirically validate our approach through a series of experiments, using both embedded systems development experts and students as subjects for evaluating both the product configuration efficiency and efectiveness and the effectiveness of the quality-driven model transformation architecture. The experiments will provide feedback to the multi-model. Thus, the multimodel can evolve based on the empirical evidence provided by the experiments.

9.3. Related Publications

During the development of the present work, different public the following list gathers these publications:

International Conferences

- **ACM/IEEE 13th International Conference on Model-Driven Engineering Languages and Systems MODELS 2010**

  Emilio Insfran, **Javier Gonzalez-Huerta** and Silvia Abrahão: *Design Guidelines for the Development of Quality-Driven Model Transformations* In: Lecture Notes in Computer Science, 2010, Volume 6395/2010, pp. 288-302

  In this paper we presented a set of guidelines for the development of quality-driven model transformations (Discussed in Section 6.3).

  This conference is listed in the 2010 CORE Conference Ranking (www.core.ed.au) as Type B Conference. MoDELS is the most important international conference in the area of Model Driven Software Development. Only publishes research papers. The language is English.

- **11th International Conference on Product-Focused Software Process Improvement PROFES 2010**

  Javier Gonzalez-Huerta, David Blanes, Emilio Insfran, and Silvia Abrahão: *Towards an Architecture for Ensuring Product Quality in Model-Driven Software Development*

  In this paper we present the architecture for supporting quality-driven model transformations. It defines a set of artifacts and a process for specifying and executing model transformations (Discussed in Section 6.2)

  This conference is listed in the 2010 CORE Conference Ranking (www.core.ed.au) as Type B Conference. Profess is one of the most relevant conferences in the product-focused software and process improvement. Only publishes research papers. The language is English.

Posters on International Events:

- **1st International Master Class on Model-Driven Engineering. Modeling Wizards 2010**

  Javier Gonzalez-Huerta, Emilio Insfran, and Silvia Abrahão: *Definining an Architecture for Quality-Driven Model Transformations*

  In: IRIT/RR-2010-20-FR

  This poster briefly presents our architecture for supporting quality-driven model transformations.

  The Modeling wizard is an autumn school on model driven development. It puts together PhD students with the most relevant researchers on the topic. The school includes theoretical and practical sessions and the presentation of the last research works. It allows the students to present the works they are working with and receive feedback about it.

National Conferences

- **XVI Jornadas de Ingeniería del Software y Bases de Datos JISBD 2011**

  Javier Gonzalez-Huerta, Emilio Insfran, and Silvia Abrahão: *Un enfoque Multi-modelo para la Introducción de Atributos de Calidad en el Desarrollo de Líneas de Producto Software*

  This article proposes a preliminary approach of the multi-model for expressing the relationships among elements of the different views (discussed in Chapter 5).

  JISBD is the main Software Engineering national conference in Spain. Allows researchers to present both regular and emergent works to be discussed with the community. The language is Spanish.

Workshops collocated with national conferences

- **VII Taller en Desarrollo de Software Dirigido por Modelos DSDM2010, junto a las XV Jornadas de Ingeniería del Software y Bases de Datos JISBD 2010**

URL: http://www.sistedes.es/TJISBD/Vol-4/No-2/index.html

Javier González, Emilio Insfrán**,** Silvia Abrahão: *Automatización de la Selección de Transformaciones Alternativas Basada en Atributos de Calidad*

This article presents a practical example of the application of a Quality-Driven Transformation Process (Discussed in Chapter 6).

Although being a national workshop, DSDM had a high-level reviewing process by a high-qualified program committee with reviewers, as are program chairs or members of the program committee of international conferences as FASE or CaiSE, international workshops as VaMOS or MDI Workshop 2010 collocated with MODELS 2010.

References:

[1] Abrahão, S., Insfran, E., Genero, M. Carsí, J. A., Ramos, I. y Piattini, M. Quality-Driven Model Transformations: From Requirements to UML Class Diagrams. In: Model-Driven Software Development: Integrating Quality Assurance, Jörg Rech y Christian Bunse (Eds.), IGI Global, 302-326. (2008).

[2] Ameller, D., Franch, X., Cabot, J.: Dealing with Non-Functional Requirements in Model-Driven Development. In proceedings of 18th Requirements Engineering Conference (RE2010), pp 189 - 198, Sydney, NSW (2010).

[3] Al-Naeem, T., Gorton, I., Ali Babar, M., Rabhi, F., Benatallah, B.: A Quality-Driven Systematic Approach for Architecting Distributed Software Applications. In: Proceedings of International Conference on Software Engineering (ICSE2005), St. Louis pp. 244--253. (2005).

[4] Baas, L., Clements, P., Kazman, R.: Software Architecture in Practice. 2nd edition, Addison Wessley, Reading, Mass, ISBN: 0321154959 (2003).

[5] Belaunde, M.: Transformation Composition in QVT. In: First European Workshop on Composition of Model Transformations (CMT). Bilbao, Spain (2006).

[6] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In O. Pastor and J. F. e Cunha, editors, Advanced Information Systems Engineering, 17th International Conference, CAiSE, Proceedings, volume 3520 of Lecture Notes in Computer Science, pages 491–503. Springer, (2005).

[7] Bézivin, J. and Gerbé, O.: Towards a Precise Definition of the OMG/MDA(TM) Frame-work. In: Proceedings of Automated Software Engineering (ASE'01), San Diego, USA, pp. 26-29, (2001).

[8] Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., Macleod, G. J., Merrit. M. J.: Characteristics of Software Quality, volume 1 of TRW Series of Software Technology. North-Holland Publishing Company, Amsterdam, (1978).

[9] Boronat, A., Knapp, A., Meseguer, J. Wirsing, M.: What is a Multi-Modeling Language? In: LNCS, vol 5486/2009, pp 71—87 Springer, Heidelberg, (2009).

[10] Bosch, J.: Design and use of software architectures: adopting and evolving a product-line approach. ACM Press/Addison-Wesley, New York, USA (2000).

[11] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley, ISBN: 0471958697 (1996).

[12] Classen, A., Heymans, P., Schobbens., P.: What's in a Feature: A RequirementsEngineering Perspective. In Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE2008), volume 4961 of Lecture Notes in Computer Science, pp 16–30. Springer-Verlag, (2008).

[13] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond, Addison Wesley, ISBN: 9780321552687 (2002).

[14] Clements, P. and Northrop, L.: Software Product Lines: Practices and Patterns, Addison-Wesley, Boston, ISBN: 9780201703320 (2007).

[15] Crnkovic, I., Larsson M., Preiss, O.: Concerning Predictability in Dependable Component- Based Systems: Classification of Quality Attributes. Lecture Notes in Computer Science 3549, Springer, pp 257-278. (2005)

[16] Czarnecki, K., Kim, C.H.: Cardinality-based feature modeling and constraints: A progress report, In International Workshop on Software Factories, San Diego, California, (2005).

[17] Czarnecki, K, Helsen, S.: Feature-based Survey of Model Transformation Approaches. In: IBM Systems Journal, Vol. 45(3), pp. 621--645, (2006).

[18] Dershowitz, N., Jouannaud, J. P.: Rewrite Systems. In: Formal models and semantics, Handbook of Theoretical Computer Science, Vol. B, (ed. J. van Leeuwen), Elsevier – The MIT Press, Amsterdam, pp. 243--320 (1990)

[19] Drago, L.M, Ghezzi, C., Mirandola, R.: Towards Quality Driven Exploration of Model Transformation Spaces, 14th International Conference on Model Drive Engineering Languages and Systems (MODELS2010), Part II, LNCS 6395, pp. 288–302, Wellington, New Zealand (2011).

[20] Etxeberria, L., Sagardui, G.: Variability Driven Quality Evaluation in Software Product Lines. In: Proceedings of the 13thInternational Software Product Line Conference (SPLC2008), Limerick, Ireland, pp.243-252 (2008).

[21] Favre, J.M: Towards a Basic Theory to Model Model Driven Engineering. In: Workshop on Software Model Engineering, Lisboa, (2004).

[22] Feiler, P. H., Gluch, D. P., Hudak, J.: The Architecture Analysis & Design Language (AADL): An Introduction (CMU/SEI-2006-TN-011). Software Engineering Institute, Carnegie Mellon University, From http://www.sei.cmu.edu/publications/documents/06.reports/06tn011.html (2006)

[23] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, ISBN: 0201633612 (1995).

[24] García Galán, J., Trinidad, P., Ruiz-Cortés, A.: FaMa Abductive: una herramienta para explicaciones de errores en modelos de características. (In Spanish) En Actas de las XVI Jornadas de Ingeniería del Software y Bases de Datos (JISBD2011), A Coruña (2011).

[25] Gardner, T., Griffin, C., Koehler, J., Hauser, R.: A Review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations Toward the Final Standard. In: OMG Document, Object Management Group, ad/03-08-02, http://www.omg.org/cgi-bin/doc?ad/03-08-02 (2003).

[26] Gómez, A., Ramos, I.: Cardinality-based feature modeling and model-driven engineering: Fitting them together. In Fourth International Workshop on Variability Modeling of Software intensive Systems (VAMOS'10), Linz (2010).

[27] González-Baixauli, B. , do Prado Leite, J. C. S., Mylopoulos, J.: Visual variability analysis for goal models. In 12th IEEE International Conference on Requirements Engineering (RE), pages 198–207. IEEE Computer Society, (2004)

[28] Gonzalez-Huerta, J., Insfran, E., Abrahão, S.: Un enfoque Multi-modelo para la Introducción de Atributos de Calidad en el Desarrollo de Líneas de Producto Software. (In Spanish) En Actas de las XVI Jornadas de Ingeniería del Software y Bases de Datos (JISBD2011), A Coruña, Spain (2011).

[29] Gonzalez-Huerta, J., Blanes, J., Insfran, E., Abrahão, S.: Towards an Architecture for Ensuring Product Quality in Model-Driven Software Development. In: 11th International Conference on Product-Focused Software Process Improvement (PROFES) Limerick, Ireland (2010).

[30] Gonzalez-Huerta, J., Insfran, E., Abrahão, S.: Automatización de la Selección de Transformaciones Alternativas Basada en Atributos de Calidad. En Actas del VII Taller sobre Desarrollo de Software Dirigido por Modelos, pp. 10-18, Valencia, Spain (2010).

[31] Hause, M. C., Thom, F.: An Integrated MDA Approach with SysML and UML. In 13th IEEE International Conference on Engineering of Complex Computer Systems (IECCS2008)., pp 249-254, Belfast (2008).

[32] Hudak, J. and Feiler, P.: Developing AADL Models for Control Systems: A Practitioner’s Guide, Technical Report CMU/SEI-2007-TR-014, Software Engineering Institute, (2007).

[33] Ikv++ technologies AG. ikv++ mediniQVT website. Last accessed on August 2011, from http://projects.ikv.de/qvt (2011).

[34] Insfran, E., Gonzalez-Huerta, J., Abrahão, S.: Design Guidelines for the Development of Quality-Driven Model Transformations. In: Proceedings of 13th International Conference on Model Drive Engineering Languages and Systems (MODELS2010), Part II, LNCS 6395, pp. 288–302, Oslo, Norway (2010).

[35] ISO/IEC 25000:2005. Software Engineering. Software product Quality Requirements and Evaluation (SQuaRE) (2005).

[36] ISO/IEC FCD 9126-1.2. Information Technology - Software product quality - Part 1: Quality model (2001).

[37] I. Ivkovic and K. Kontogiannis, "A Framework for Software Architecture Refactoring Using Model Transformations and Semantic Annotations," in *Proceedings of the Conference on Software Maintenance and Reengineering,* , pp. 135-144 (2006)

[38] Jarzabek, S., Yang, B., Yoeun, S.: Addressing quality attributes in domain analysis for product lines. IEE Proceedings - Software, 153(2):61–73, (2006).

[39] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990).

[40] Kerhervé, B., Nguyen, K. K., Gerbé, O. y Jaumard, B. A. (2006). Framework for Quality-Driven Delivery in Distributed Multimedia Systems. Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006), 195–205. (2006)

[41] Kleppe A., Warmer, J., and Bast, W.: MDA Explained, TheModel Driven Architecture: Practice and Promise, Addison-Wesley, Boston, MA, ISBN: 032119442X, (2003).

[42] Kolb, R., McGregor, J. D., Muthig, D.: Introduction to quality assurance in reuse contexts. In First International Workshop on Quality Assurance in Reuse Contexts (QUARC2004), Boston, USA (2004).

[43] Kurtev, I.: Adaptability of Model Transformations. PhD Thesis. University of Twente, Twente, the Netherlands (2005)

[44] Markovic, S., Baar, T.: Refactoring OCL annotated UML class diagrams. In Briand, L.C., Williams, C., eds.: MoDELS’05: Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica. Volume 3713 of Lecture Notes in Computer Science., Springer-Verlag pp 280–294 (2005).

[45] Maswar, F., Chaudron, M., Radovanovic, I., Bondarev, E.: Improving Architectural Quality Properties through Model Transformations. Software Engineering Research and Practice (SERP2007), Las Vegas, USA, pp.687-693, (2007)

[46] McCall, J. A., Richards, P. K., Walters, G. F.: Factors in Software Quality, volume vol 1-3 of AD/A-049-015/055. Springfield, (1977).

[47] McNeile, A.: MDA: The Vision with the Hole? Last accessed on August 2011, from http://www.metamaxim.com/download/documents/MDAv1.pdf, (2003).

[48] Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, New York. ISBN: 0201788918. (2004).

[49] Merilinna, J.: A Tool for Quality-Driven Architecture Model Transformation. PhD thesis. VTT Technical Research Centre of Finland, Vuorimiehentie, Finland (2005)

[50] Montagud, S.: Un Método para la Evaluación de la Calidad de Líneas de Productos Software basado en SQuaRE. Master's Thesis, (In Spanish), Master en Ingenieria del Software Metodos Formales y Sistemas de Información. Universidad Politécnica de Valencia, Spain (2009).

[51] OMG: MDA Guide Version 1.0.1. Last accessed on August 2011, from http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf (2002).

[52] OMG: Meta Object Facility (MOF) Core Specification 2.4. Last accessed on August 2011, from http://www.omg.org/spec/MOF/2.4/Beta2/PDF/ (2010).

[53] OMG: Meta Object Facility 2.0 Query/View/Transformation Specification. Last accessed on August 2011, from http://www.omg.org/spec/QVT/1.0/PDF (2008).

[54] OMG: SysML 1.2 Specification: http://www.omg.org/spec/SysML/1.2/PDF/

[55] OMG: UML 2.0 Object Constrain Languaje (OCL) Specification. Last accessed on August 2011, from, http://www.omg.org/spec/OCL/2.0/ (2006).

[56] O'Regan, G.: A Practical Approach to Software Quality, Springer-Verlag, New York ISBN 978-0-387-95321-2 (2002).

[57] Nolan, A. J.: Building a Comprehensive Software Product Line Cost Model. In: Proceedings of the 13thInternational Software Product Line Conference (SPLC2009), San Francisco-CA, USA, IEEE Press, (2009).

[58] Nolan, J.A., Abrahão, S., Clements, P., McGregor, J.D., Cohen, S.: Towards the Integration of Quality Attributes into a Software Product Line Cost Model. In 15th Software Product Line Conference, Munich (2011).

[59] Pohl, K., Böckle, G., and Linden, F. J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer- Verlag New York, Inc, ISBN: 10 3-540-24372-0 (2005).

[60] Raatikainen, M., Niemelä, E., Myllärmiemi, V., Männistö, T.: Svamp – An Integrated Approach for Modeling Functional and Quality Variability. 2nd Workshop on Modeling and Analysis of Software-Intensive Systems (VaMOS 2008), Essen, Germany (2008).

[61] Röttger, S., Zschaler, S.: Tool support for refinement of non-functional specifica- tions. Software and Systems Modelling journal (SoSyM) 6(2) (2007)

[62] Seidewitz, E.: What models mean, IEEE Software, September/October 2003, (Vol.20, No. 5), pp. 26-32 (2003).

[63] Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, New Jersey, USA, ISBN: 0131829572 (1996).

[64] Stahl, T. and Völter, M.: Model-Driven Software Development - Technology, Engineering, Management, John Wiley and Sons, Ltd., Chichester, England , ISBN: 0470025700, (2006).

[65] Shiraishi, S.: An AADL-Based Approach to Variability Modeling of Automotive Control Systems. In: Proceedings of 13th International Conference on Model Drive Engineering Languages and Systems (MODELS2010), Part II, LNCS 6394, pp 346-360, Oslo, Norway (2010).

[66] Tawhid, R., Petriu D.C.: Automatic Derivation of a Product Performance Model from a Software Product Line Model. In 15th Software Product Line Conference (SPLC 2011), Munich, Germany (2011)

[67] The Eclipse Foundation. Last accessed on August 2011, from http://www.eclipse.org (2011).

[68] Zhang, H., Jarzabek, S., Yang, B.: Quality Prediction and Assessment for Product Lines. In 15th International Conference on Advanced Information Systems Engineering (CAiSE2003), pp. 681-695, (2003).

[69] Zou, Y., Kontogiannis, K.: Quality Driven Transformation Framework for Object Oriented Migration. 2nd ASERC Workshop on Software Architecture. (2003)

Annex I. Vehicle Control AADL Specification

```
-- AADL model of automobile control systems consisting of: traction
-- control, stability control, cruise control, and antilock brake
-- systems.
-- THIS IS THE 'PORT GROUP' VERSION
data bool_type
end bool_type;

data real_type
end real_type;

-- Device declarations -------------------------------------------
device engine
  features
    engine_signals: port group engine_socket_1;
end engine;

device wheel_rotation_sensor
  features
    wheel_signal: port group wheel_sensors_socket_1;
end wheel_rotation_sensor;

device brake_pedal
  features
    brake_signals: port group brake_sensors_socket_1;
end brake_pedal;

device vehicle_state_sensors
  features
    vehicle_state_signals: port group vehicle_state_sensors_socket_1;
end vehicle_state_sensors;

device user_console
  features
    user_console_outputs: port group user_console_socket_1;
end user_console;

device distance_radar_sensor
  features
    distance_radar_signals: port group distance_radar_socket_1;
    -- As input will use the self_speed of the car for calculating relative
speed with the target vehicle
    self_speed: port group wheel_sensors_plug_1;
end distance_radar_sensor;

device object_recognition_sensor
  features
    object_recognition_sensor_signals: port group
object_recognition_sensor_socket_1;
    -- As input will use the self_speed of the car for calculating relative
speed with the target vehicle
    self_speed: port group wheel_sensors_plug_1;
end object_recognition_sensor;

----- output devices
device throttle_actuator
  features
    tc_throttle_signals: port group tc_throttle_actuator_socket_1;
    cc_throttle_signals: port group cc_throttle_actuator_socket_1;
end throttle_actuator;

device display
  features
```

```
    tc_display_input_signals: port group tc_user_display_socket_1;
    cc_display_input_signals: port group cc_user_display_socket_1;
    sc_display_input_signals: port group sc_user_display_socket_1;
    abs_display_input_signals: port group abs_user_display_socket_1;
end display;

device brake_actuators
  features
    -- device receives braking signals from three systems
    tc_brake_actuator_signals: port group tc_brake_actuator_socket_1;
    sc_brake_actuator_signals: port group sc_brake_actuator_socket_1;
    abs_brake_actuator_signals: port group abs_brake_actuator_socket_1;
end brake_actuators;

-- extension for the brake_actuators in the version 2 and 3
device brake_actuators_ext
  extends brake_actuators
  features
    cc_brake_actuator_signals: port group cc_brake_actuactor_socket_1;
end brake_actuators_ext;

-- port group declarations ----------------------------------
port group engine_socket_1
  features
    engine_status: out data port;
    -- engine is off (0) or on (1)
    engine_temp_1: out data port;
    throttle_position: out data port;
end engine_socket_1;

port group engine_plug_1
  inverse of engine_socket_1
end engine_plug_1;

port group wheel_sensors_socket_1
  features
    wheel_pulse: in data port bool_type;
    wheel_slippage: in data port real_type;
end wheel_sensors_socket_1;

-- assume only one rotation sensor on one wheel....could add one on -- other
wheels for redundancy
port group wheel_sensors_plug_1
  inverse of wheel_sensors_socket_1
end wheel_sensors_plug_1;

port group brake_sensors_socket_1
  features
    brake_status: out data port;
end brake_sensors_socket_1;

port group brake_sensors_plug_1
  inverse of engine_socket_1
end brake_sensors_plug_1;

port group vehicle_state_sensors_socket_1
  features
    steering_wheel_angle: in data port;
    yaw_rate: in data port;
    lateral_acceleration: in data port;
end vehicle_state_sensors_socket_1;

port group vehicle_state_sensors_plug_1
  inverse of vehicle_state_sensors_socket_1
end vehicle_state_sensors_plug_1;

port group user_console_socket_1
  features
    cc_system_on_off: out data port;
    speed_set: out data port;
    resume: out data port;
    cancel: out data port;
    speed_increase: out data port;
    speed_decrease: out data port;
    -- switch for enabling maintain distance
    maintain_distance: out data port;
end user_console_socket_1;

port group user_console_plug_1
  inverse of user_console_socket_1
end user_console_plug_1;

-------- distance radar port
port group distance_radar_socket_1
  features
    distance_to: in data port;
    relative_speed: in data port;
end distance_radar_socket_1;

port group distance_radar_plug_1
  inverse of distance_radar_socket_1
end distance_radar_plug_1;

--------
port group object_recognition_sensor_socket_1
  features
    distance_to: in data port;
    relative_speed: in data port;
end object_recognition_sensor_socket_1;

port group object_recognition_sensor_plug_1
  inverse of object_recognition_sensor_socket_1
end object_recognition_sensor_plug_1;

-- TCS output port groups
port group tc_throttle_actuator_socket_1
  features
    throttle_actuator: out data port;
end tc_throttle_actuator_socket_1;

port group tc_throttle_actuator_plug_1
  inverse of tc_throttle_actuator_socket_1
end tc_throttle_actuator_plug_1;

port group tc_user_display_socket_1
  features
    tc_state: out data port;
end tc_user_display_socket_1;

port group tc_user_display_plug_1
  inverse of tc_user_display_socket_1
end tc_user_display_plug_1;

port group tc_brake_actuator_socket_1
  features
    tc_brake_output: out data port;
end tc_brake_actuator_socket_1;

port group tc_brake_actuator_plug_1
  inverse of tc_brake_actuator_socket_1
end tc_brake_actuator_plug_1;

port group cc_brake_actuactor_socket_1
  features
    cc_brake_output: out data port;
end cc_brake_actuactor_socket_1;

-- introduce to add the braking capability for the CC_Version 2 and 3
port group cc_brake_actuactor_plug_1
  inverse of cc_brake_actuactor_socket_1
```

```
end cc_brake_actuactor_plug_1;

---------------------------------------- -- cc output port groups
port group cc_throttle_actuator_socket_1
  features
    cc_throttle_actuator: out data port;
end cc_throttle_actuator_socket_1;

port group cc_throttle_actuator_plug_1
  inverse of cc_throttle_actuator_socket_1
end cc_throttle_actuator_plug_1;

port group cc_user_display_socket_1
  features
    cc_state: out data port;
end cc_user_display_socket_1;

port group cc_user_display_plug_1
  inverse of cc_user_display_socket_1
end cc_user_display_plug_1;

------------------------------------------------- -- sc output port groups
port group sc_brake_actuator_socket_1
  features
    sc_brake_actuator: out data port;
end sc_brake_actuator_socket_1;

port group sc_brake_actuator_plug_1
  inverse of sc_brake_actuator_socket_1
end sc_brake_actuator_plug_1;

port group sc_user_display_socket_1
  features
    sc_state: out data port;
end sc_user_display_socket_1;

port group sc_user_display_plug_1
  inverse of sc_user_display_socket_1
end sc_user_display_plug_1;

--------------------------------------------- -- abs output port groups
port group abs_brake_actuator_socket_1
  features
    abs_brake_actuator: out data port;
end abs_brake_actuator_socket_1;

port group abs_brake_actuator_plug_1
  inverse of abs_brake_actuator_socket_1
end abs_brake_actuator_plug_1;

port group abs_user_display_socket_1
  features
    abs_state: out data port;
end abs_user_display_socket_1;

port group abs_user_display_plug_1
  inverse of abs_user_display_socket_1
end abs_user_display_plug_1;

------------------------------------------------------------------------------
system traction_control_system
  features
    tcs_wheel_input: port group wheel_sensors_plug_1;
    tcs_engine_input: port group engine_plug_1;
    tcs_user_input: port group user_console_plug_1;
    tcs_throttle_out: port group tc_throttle_actuator_socket_1;
    tcs_display_out: port group tc_user_display_socket_1;
    tcs_brake_out: port group tc_brake_actuator_plug_1;
end traction_control_system;
```

```
system cruise_control_system_CC1
  features
    cc_user_input: port group user_console_plug_1;
    cc_wheel_speed: port group wheel_sensors_plug_1;
    cc_engine_input: port group engine_plug_1;
    cc_brake_status: port group brake_sensors_plug_1;
    cc_throttle_actuator: port group cc_throttle_actuator_plug_1;
    cc_display_out: port group cc_user_display_plug_1;
end cruise_control_system_CC1;

system cruise_control_system_CC2
  extends cruise_control_system_CC1
  features
    cc_distance_radar_setting: port group distance_radar_plug_1;
    cc_brake_actuator: port group cc_brake_actuactor_plug_1;
end cruise_control_system_CC2;

system cruise_control_system_CC3
  extends cruise_control_system_CC2
  features
    cc_object_rectonition_setting: port group object_recognition_sensor_plug_1;
end cruise_control_system_CC3;

system stability_control_system
  features
    sc_user_input: port group user_console_plug_1;
    sc_wheel_speed: port group wheel_sensors_plug_1;
    sc_engine_input: port group engine_plug_1;
    sc_brake_status: port group brake_sensors_plug_1;
    sc_display_out: port group sc_user_display_plug_1;
    sc_brake_output: port group sc_brake_actuator_plug_1;
end stability_control_system;

system antilock_brake_system
  features
    abs_user_input: port group user_console_plug_1;
    abs_wheel_speed: port group wheel_sensors_plug_1;
    abs_engine_input: port group engine_plug_1;
    abs_brake_actuator: port group abs_brake_actuator_plug_1;
    abs_display: port group abs_user_display_plug_1;
end antilock_brake_system;
```

Annex II. Cruise Control AADL Specification

```
data bool_type
end bool_type;

data float_type
end float_type;

-- Devices
device cruise_control_button
  features
    cc_system_on_off: out data port;
end cruise_control_button;

device brake_pedal
  features
    brake_pedal_status: out data port bool_type;
  flows
    Flow1: flow source brake_pedal_status;
end brake_pedal;

device wheel_rotation_sensor
  features
    wheel_pulse: out data port;
    --Traction & Stability Control
    wheel_slippage: out data port;
end wheel_rotation_sensor;

device airbag_and_security_belt_status
  features
    airbag_status: out data port;
    security_belt_status: out data port;
end airbag_and_security_belt_status;

device engine
  features
    engine_status: out data port;
end engine;

device resume_button
  features
    resume: out data port;
end resume_button;

device speed_up_button
  features
    increase_speed: out data port;
end speed_up_button;

device speed_dn_button
  features
    decrease_speed: out data port;
end speed_dn_button;

device set_button
  features
    set_speed: out data port;
end set_button;

device set_distance_button
  features
    set_distance: out data port;
end set_distance_button;
```

```
device operator_panel
  features
    cc_system_on_off: out data port bool_type;
    set_speed: out data port float_type;
    decrease_speed: out data port bool_type;
    increase_speed: out data port bool_type;
    resume: out data port bool_type;
    set_distance: out data port float_type;
end operator_panel;

device sensor_platform
  features
    yaw_sensor_1: out data port float_type;
    yaw_rate_sensor_1: out data port float_type;
    lateral_force_sensor_1: out data port float_type;
end sensor_platform;

device steering_wheel
  features
    steering_angle: out data port float_type;
end steering_wheel;

device distance_radar_sensor
  features
    distance_to: out data port float_type;
    relative_speed: out data port float_type;
    -- The previously computed instantaneous_velocity
    instantaneous_velocity: in data port;
end distance_radar_sensor;

device object_recognition_sensor
  features
    distance_to: out data port float_type;
    relative_speed: out data port float_type;
    -- The previously computed instantaneous_velocity
    instantaneous_velocity: in data port;
end object_recognition_sensor;

-- Device: actuators
device throttle_actuator
  features
    throttle_setting: in data port;
  flows
    Flow1: flow sink throttle_setting;
end throttle_actuator;

device brake_actuators
  features
    -- device receives braking signals from three systems
    tc_brake_input: in data port;
    sc_brake_input: in data port;
    abs_brake_input: in data port;
    brake_status: out data port;
  flows
    flow1: flow source brake_status;
end brake_actuators;

-- extension for the brake_actuators in the version 2 and 3
device brake_actuators_ext
  extends brake_actuators
  features
    cc_brake_input: in data port;
end brake_actuators_ext;

-- device for airbag and security belt actuators, which will be controlled by
version 3
device airbag_and_security_belt_actuators
  features
    cc_security_belt_setting: in data port;
    cc_airbag_setting: in data port;
  flows
    Flow1: flow sink cc_security_belt_setting;
    Flow2: flow sink cc_airbag_setting;
end airbag_and_security_belt_actuators;

------ end of additional component declarations
-- System declarations
-- the cruise control software application is declared (sw + devices,
-- devices will be bound later)
system cc1_app
  features
    device_bus: requires bus access PC104_ISA_16BIT;
end cc1_app;

system cc2_app
  extends cc1_app
end cc2_app;

system cc3_app
  extends cc2_app
end cc3_app;

bus PC104_ISA_16BIT
end PC104_ISA_16BIT;

memory SDRAM
  features
    controller_memory: requires bus access PC104_ISA_16BIT;
end SDRAM;

processor PENTIUM
  features
    controller_cpu: requires bus access PC104_ISA_16BIT;
end PENTIUM;

system cruise_control_CC1
  features
    cc_system_on_off: in data port;
    engine_status: in data port;
    brake_pedal_status: in data port bool_type;
    resume: in data port;
    decrease_speed: in data port;
    increase_speed: in data port;
    set_speed: in data port;
    wheel_pulse: in data port;
    throttle_setting: out data port;
  flows
    brake_flow_1: flow path brake_pedal_status -> throttle_setting;
    decrease_speed_flow1: flow path decrease_speed -> throttle_setting;
    increase_speed_flow1: flow path increase_speed -> throttle_setting;
    set_speed_flow1: flow path set_speed -> throttle_setting;
end cruise_control_CC1;

system cruise_control_CC2
  extends cruise_control_CC1
  features
    distance_from_distance_sensor: in data port float_type;
    relative_speed_from_distance_sensor: in data port float_type;
    brake_status: in data port;
    current_instantaneous_velocity: out data port;
    brake_setting: out data port;
    set_distance: in data port;
  flows
    brake_flow_2: flow path brake_pedal_status -> brake_setting;
    brake_flow_3: flow path brake_status -> brake_setting;
    current_velocity_flow_1: flow path wheel_pulse ->
current_instantaneous_velocity;
    relative_speed_from_distance_sensor_flow_1: flow path
relative_speed_from_distance_sensor -> brake_setting;
```

```
    set_distance_flow1: flow path set_distance -> throttle_setting;
end cruise_control_CC2;

system cruise_control_CC3
  extends cruise_control_CC2
  features
    distance_to_from_object_recognition_sensor: in data port float_type;
    relative_speed_from_object_recognition_sensor: in data port float_type;
    security_belt_status: in data port;
    airbag_status: in data port;
    security_belt_setting: out data port;
    airbag_setting: out data port;
  flows
    airbag_flow1: flow path airbag_status -> airbag_setting;
    security_belt_flow1: flow path security_belt_status ->
security_belt_setting;
end cruise_control_CC3;

device compute_velocity
  features
    wheel_pulse: in data port;
    instantaneous_velocity: out data port;
  flows
    FS1: flow path wheel_pulse -> instantaneous_velocity;
end compute_velocity;

process compute_desired_speed
  features
    ok_to_run: in data port;
    instantaneous_velocity: in data port;
    selected_speed: in data port;
    current_instantaneous_velocity: out data port;
    previous_instantaneous_velocity: in data port;
    desired_speed: out data port;
  flows
    FS1: flow path ok_to_run -> desired_speed;
    FS2: flow path instantaneous_velocity -> desired_speed;
    FS3: flow path selected_speed -> desired_speed;
end compute_desired_speed;

process compute_desired_speed_CC2
  extends compute_desired_speed
  features
    distance_to_from_distance_sensor: in data port float_type;
    relative_speed_from_distance_sensor: in data port float_type;
    selected_distance: in data port float_type;
  flows
    FS4: flow path distance_to_from_distance_sensor -> desired_speed;
    FS5: flow path relative_speed_from_distance_sensor -> desired_speed;
    FS11: flow path selected_distance -> desired_speed;
end compute_desired_speed_CC2;

process compute_desired_speed_CC3
  extends compute_desired_speed_CC2
  features
    distance_to_from_object_recognition_sensor: in data port float_type;
    relative_speed_from_object_recognition_sensor: in data port float_type;
  flows
    FS6: flow path distance_to_from_object_recognition_sensor -> desired_speed;
    FS7: flow path relative_speed_from_object_recognition_sensor ->
desired_speed;
end compute_desired_speed_CC3;

process compute_throttle_setting
  features
    desired_speed: in data port;
    throttle_setting: out data port;
  flows
    FS1: flow path desired_speed -> throttle_setting;
end compute_throttle_setting;
```

```
process compute_brake_setting
  features
    desired_speed: in data port;
    brake_setting: out data port;
    brake_status: in data port;
  flows
    FS1: flow path desired_speed -> brake_setting;
    FS2: flow path brake_status -> brake_setting;
end compute_brake_setting;

process compute_airbag_setting
  features
    desired_speed: in data port;
    instantaneous_velocity: in data port;
    airbag_status: in data port;
    distance_to_from_object_recognition_sensor: in data port float_type;
    relative_speed_from_object_recognition_sensor: in data port float_type;
    airbag_setting: out data port;
  flows
    FS1: flow path desired_speed -> airbag_setting;
    FS2: flow path instantaneous_velocity -> airbag_setting;
    FS3: flow path distance_to_from_object_recognition_sensor -> airbag_setting;
    FS4: flow path relative_speed_from_object_recognition_sensor ->
airbag_setting;
    FS5: flow path airbag_status -> airbag_setting;
end compute_airbag_setting;

process compute_security_belt_setting
  features
    desired_speed: in data port;
    instantaneous_velocity: in data port;
    security_belt_status: in data port;
    distance_to_from_object_recognition_sensor: in data port float_type;
    relative_speed_from_object_recognition_sensor: in data port float_type;
    security_belt_setting: out data port;
  flows
    FS1: flow path desired_speed -> security_belt_setting;
    FS2: flow path instantaneous_velocity -> security_belt_setting;
    FS3: flow path distance_to_from_object_recognition_sensor ->
security_belt_setting;
    FS4: flow path relative_speed_from_object_recognition_sensor ->
security_belt_setting;
    FS5: flow path security_belt_status -> security_belt_setting;
end compute_security_belt_setting;

process in_control
  features
    cc_system_on_off: in data port;
    brake_pedal_status: in data port bool_type;
    resume: in data port;
    decrease_speed: in data port;
    increase_speed: in data port;
    set_speed: in data port;
    engine_status: in data port;
    ok_to_run: out data port;
    selected_speed: out data port;

  flows
    FS1: flow path brake_pedal_status -> ok_to_run;
    FS2: flow path decrease_speed -> selected_speed;
    FS3: flow path increase_speed -> selected_speed;
    FS4: flow path set_speed -> selected_speed;
    FS5: flow path cc_system_on_off -> ok_to_run;
end in_control;

process in_control_ext
  extends in_control
  features
    set_distance: in data port;
    selected_distance: out data port float_type;
flows
    FS6: flow path set_distance -> selected_distance;
end in_control_ext;

system implementation cruise_control_CC1.impl
  subcomponents
    I_C: process in_control;
    C_V: device compute_velocity;
    C_D_S: process compute_desired_speed;
    C_T_S: process compute_throttle_setting;
  connections
    C1: data port cc_system_on_off -> I_C.cc_system_on_off;
    C2: data port brake_pedal_status -> I_C.brake_pedal_status;
    C3: data port engine_status -> I_C.engine_status;
    C4: data port resume -> I_C.resume;
    C5: data port decrease_speed -> I_C.decrease_speed;
    C6: data port increase_speed -> I_C.increase_speed;
    C7: data port set_speed -> I_C.set_speed;
    C8: data port wheel_pulse -> C_V.wheel_pulse;
    C9: data port I_C.ok_to_run -> C_D_S.ok_to_run;
    C10: data port C_V.instantaneous_velocity -> C_D_S.instantaneous_velocity;
    C11: data port C_D_S.current_instantaneous_velocity ->
C_D_S.previous_instantaneous_velocity;
    C12: data port C_D_S.desired_speed -> C_T_S.desired_speed;
    C13: data port C_T_S.throttle_setting -> throttle_setting;
    C14: data port I_C.selected_speed -> C_D_S.selected_speed;
  flows
    brake_flow_1: flow path brake_pedal_status -> C2 -> I_C.FS1
       -> C9 -> C_D_S.FS1
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
    decrease_speed_flow1: flow path decrease_speed -> C5 -> I_C.FS2
       -> C14 -> C_D_S.FS3
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
    set_speed_flow1: flow path set_speed -> C7 -> I_C.FS4
       -> C14 -> C_D_S.FS3
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
    increase_speed_flow1: flow path increase_speed -> C6 -> I_C.FS3
       -> C14 -> C_D_S.FS3
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
end cruise_control_CC1.impl;

system implementation cc1_app.impl
  subcomponents
    CC: system cruise_control_CC1.impl;
    BRAKE_PEDAL_SENSOR: device brake_pedal;
    TA: device throttle_actuator;
    CC_ON_OFF: device cruise_control_button;
    ENGINE: device engine;
    RESUME: device resume_button;
    SP_UP: device speed_up_button;
    SP_DN: device speed_dn_button;
    SETBUTTON: device set_button;
    WHEEL_ROT_SENSOR: device wheel_rotation_sensor;
  connections
    C21: data port CC_ON_OFF.cc_system_on_off -> CC.cc_system_on_off;
    C22: data port BRAKE_PEDAL_SENSOR.brake_pedal_status ->
CC.brake_pedal_status;
    C23: data port ENGINE.engine_status -> CC.engine_status;
    C24: data port RESUME.resume -> CC.resume;
    C25: data port SP_DN.decrease_speed -> CC.decrease_speed;
    C26: data port SP_UP.increase_speed -> CC.increase_speed;
    C27: data port SETBUTTON.set_speed -> CC.set_speed;
    C28: data port WHEEL_ROT_SENSOR.wheel_pulse -> CC.wheel_pulse;
    C29: data port CC.throttle_setting -> TA.throttle_setting;
  flows
```

```
    ETE_F1: end to end flow BRAKE_PEDAL_SENSOR.Flow1 -> C22 -> CC.brake_flow_1
       -> C29 -> TA.Flow1
    ;
end cc1_app.impl;

system implementation cruise_control_CC2.impl
  subcomponents
    I_C: process in_control_ext;
    C_V: device compute_velocity;
    C_D_S: process compute_desired_speed_CC2;
    C_T_S: process compute_throttle_setting;
    C_B_S: process compute_brake_setting;
  connections
    C1: data port cc_system_on_off -> I_C.cc_system_on_off;
    C2: data port brake_pedal_status -> I_C.brake_pedal_status;
    C3: data port engine_status -> I_C.engine_status;
    C4: data port resume -> I_C.resume;
    C5: data port decrease_speed -> I_C.decrease_speed;
    C6: data port increase_speed -> I_C.increase_speed;
    C7: data port set_speed -> I_C.set_speed;
    C8: data port wheel_pulse -> C_V.wheel_pulse;
    C9: data port I_C.ok_to_run -> C_D_S.ok_to_run;
    C10: data port C_V.instantaneous_velocity -> C_D_S.instantaneous_velocity;
    C11: data port C_D_S.current_instantaneous_velocity ->
C_D_S.previous_instantaneous_velocity;
    C12: data port C_D_S.desired_speed -> C_T_S.desired_speed;
    C13: data port C_T_S.throttle_setting -> throttle_setting;
    C14: data port distance_from_distance_sensor ->
C_D_S.distance_to_from_distance_sensor;
    C15: data port relative_speed_from_distance_sensor ->
C_D_S.relative_speed_from_distance_sensor;
    C16: data port C_V.instantaneous_velocity -> current_instantaneous_velocity;
    C17: data port C_D_S.desired_speed -> C_B_S.desired_speed;
    C18: data port C_B_S.brake_setting -> brake_setting;
    C19: data port I_C.selected_speed -> C_D_S.selected_speed;
    C20: data port brake_status -> C_B_S.brake_status;
    C21: data port set_distance -> I_C.set_distance;
    C22: data port I_C.selected_distance -> C_D_S.selected_distance;
  flows
    brake_flow_1: flow path brake_pedal_status -> C2 -> I_C.FS1
       -> C9 -> C_D_S.FS1
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
    brake_flow_2: flow path brake_pedal_status -> C2 -> I_C.FS1
       -> C9 -> C_D_S.FS1
       -> C17 -> C_B_S.FS1
       -> C18 -> brake_setting;
    brake_flow_3: flow path brake_status -> C20 -> C_B_S.FS2
       -> C18 -> brake_setting;
    current_velocity_flow_1: flow path wheel_pulse -> C8 -> C_V.FS1
       -> C16 -> current_instantaneous_velocity;
    decrease_speed_flow1: flow path decrease_speed -> C5 -> I_C.FS2
       -> C19 -> C_D_S.FS3
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
    set_speed_flow1: flow path set_speed -> C7 -> I_C.FS4
       -> C19 -> C_D_S.FS3
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
    increase_speed_flow1: flow path increase_speed -> C6 -> I_C.FS3
       -> C19 -> C_D_S.FS3
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
    relative_speed_from_distance_sensor_flow_1: flow path
relative_speed_from_distance_sensor -> C15 -> C_D_S.FS5
       -> C17 -> C_B_S.FS1
       -> C18 -> brake_setting;
    set_distance_flow1: flow path set_distance -> C21 -> I_C.FS6
       -> C22 -> C_D_S.FS11
       -> C12 -> C_T_S.FS1
```

```
      -> C13 -> throttle_setting;
end cruise_control_CC2.impl;

system implementation cc2_app.impl
  subcomponents
    CC: system cruise_control_CC2.impl;
    BRAKE_PEDAL_SENSOR: device brake_pedal;
    TA: device throttle_actuator;
    BA: device brake_actuators_ext;
    CC_ON_OFF: device cruise_control_button;
    ENGINE: device engine;
    RESUME: device resume_button;
    SP_UP: device speed_up_button;
    SP_DN: device speed_dn_button;
    SETBUTTON: device set_button;
    WHEEL_ROT_SENSOR: device wheel_rotation_sensor;
    DISTANCE_SENSOR: device distance_radar_sensor;
    SET_DISTANCE: device set_distance_button;
  connections
    C21: data port CC_ON_OFF.cc_system_on_off -> CC.cc_system_on_off;
    C22: data port BRAKE_PEDAL_SENSOR.brake_pedal_status ->
CC.brake_pedal_status;
    C23: data port ENGINE.engine_status -> CC.engine_status;
    C24: data port RESUME.resume -> CC.resume;
    C25: data port SP_DN.decrease_speed -> CC.decrease_speed;
    C26: data port SP_UP.increase_speed -> CC.increase_speed;
    C27: data port SETBUTTON.set_speed -> CC.set_speed;
    C28: data port WHEEL_ROT_SENSOR.wheel_pulse -> CC.wheel_pulse;
    C29: data port CC.throttle_setting -> TA.throttle_setting;
    C30: data port DISTANCE_SENSOR.distance_to ->
CC.distance_from_distance_sensor;
    C31: data port DISTANCE_SENSOR.relative_speed ->
CC.relative_speed_from_distance_sensor;
    C32: data port CC.current_instantaneous_velocity ->
DISTANCE_SENSOR.instantaneous_velocity;
    C33: data port CC.brake_setting -> BA.cc_brake_input;
    C34: data port BA.brake_status -> CC.brake_status;
    C35: data port SET_DISTANCE.set_distance -> CC.set_distance;
  flows
    ETE_F1: end to end flow BRAKE_PEDAL_SENSOR.Flow1 -> C22 -> CC.brake_flow_1
      -> C29 -> TA.Flow1
    ;
end cc2_app.impl;

system implementation cruise_control_CC3.impl
  subcomponents
    I_C: process in_control_ext;
    C_V: device compute_velocity;
    C_D_S: process compute_desired_speed_CC3;
    C_T_S: process compute_throttle_setting;
    C_B_S: process compute_brake_setting;
    C_A_S: process compute_airbag_setting;
    C_S_S: process compute_security_belt_setting;
  connections
    C1: data port cc_system_on_off -> I_C.cc_system_on_off;
    C2: data port brake_pedal_status -> I_C.brake_pedal_status;
    C3: data port engine_status -> I_C.engine_status;
    C4: data port resume -> I_C.resume;
    C5: data port decrease_speed -> I_C.decrease_speed;
    C6: data port increase_speed -> I_C.increase_speed;
    C7: data port set_speed -> I_C.set_speed;
    C8: data port wheel_pulse -> C_V.wheel_pulse;
    C9: data port I_C.ok_to_run -> C_D_S.ok_to_run;
    C10: data port C_V.instantaneous_velocity -> C_D_S.instantaneous_velocity;
    C11: data port C_D_S.current_instantaneous_velocity ->
C_D_S.previous_instantaneous_velocity;
    C12: data port C_D_S.desired_speed -> C_T_S.desired_speed;
    C13: data port C_T_S.throttle_setting -> throttle_setting;
    C14: data port distance_from_distance_sensor ->
C_D_S.distance_to_from_distance_sensor;
```

```
    C15: data port relative_speed_from_distance_sensor ->
C_D_S.relative_speed_from_distance_sensor;
    C16: data port C_V.instantaneous_velocity -> current_instantaneous_velocity;
    C17: data port distance_to_from_object_recognition_sensor ->
C_D_S.distance_to_from_object_recognition_sensor;
    C18: data port relative_speed_from_distance_sensor ->
C_D_S.relative_speed_from_object_recognition_sensor;
    C19: data port C_D_S.desired_speed -> C_B_S.desired_speed;
    C20: data port C_B_S.brake_setting -> brake_setting;
    C21: data port C_V.instantaneous_velocity -> C_A_S.instantaneous_velocity;
    C22: data port distance_to_from_object_recognition_sensor ->
C_A_S.distance_to_from_object_recognition_sensor;
    C23: data port relative_speed_from_distance_sensor ->
C_A_S.relative_speed_from_object_recognition_sensor;
    C24: data port C_D_S.desired_speed -> C_A_S.desired_speed;
    C25: data port C_V.instantaneous_velocity -> C_S_S.instantaneous_velocity;
    C26: data port distance_to_from_object_recognition_sensor ->
C_S_S.distance_to_from_object_recognition_sensor;
    C27: data port relative_speed_from_distance_sensor ->
C_S_S.relative_speed_from_object_recognition_sensor;
    C28: data port C_D_S.desired_speed -> C_S_S.desired_speed;
    C29: data port C_A_S.airbag_setting -> airbag_setting;
    C30: data port C_S_S.security_belt_setting -> security_belt_setting;
    C31: data port airbag_status -> C_A_S.airbag_status;
    C32: data port security_belt_status -> C_S_S.security_belt_status;
    C33: data port I_C.selected_speed -> C_D_S.selected_speed;
    C34: data port brake_status -> C_B_S.brake_status;
    C35: data port set_distance -> I_C.set_distance;
    C36: data port I_C.selected_distance -> C_D_S.selected_distance;
  flows
    brake_flow_1: flow path brake_pedal_status -> C2 -> I_C.FS1
       -> C9 -> C_D_S.FS1
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
    brake_flow_2: flow path brake_pedal_status -> C2 -> I_C.FS1
       -> C9 -> C_D_S.FS1
       -> C19 -> C_B_S.FS1
       -> C20 -> brake_setting;
    brake_flow_3: flow path brake_status -> C34 -> C_B_S.FS2
       -> C20 -> brake_setting;
    security_belt_flow1: flow path security_belt_status -> C32 -> C_S_S.FS5
       -> C30 -> security_belt_setting;
    airbag_flow1: flow path airbag_status -> C31 -> C_A_S.FS5
       -> C29 -> airbag_setting;
    current_velocity_flow_1: flow path wheel_pulse -> C8 -> C_V.FS1
       -> C16 -> current_instantaneous_velocity;
    decrease_speed_flow1: flow path decrease_speed -> C5 -> I_C.FS2
       -> C33 -> C_D_S.FS3
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
    set_speed_flow1: flow path set_speed -> C7 -> I_C.FS4
       -> C33 -> C_D_S.FS3
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
    increase_speed_flow1: flow path increase_speed -> C6 -> I_C.FS3
       -> C33 -> C_D_S.FS3
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
    relative_speed_from_distance_sensor_flow_1: flow path
relative_speed_from_distance_sensor -> C15 -> C_D_S.FS5
       -> C19 -> C_B_S.FS1
       -> C20 -> brake_setting;
    set_distance_flow1: flow path set_distance -> C35 -> I_C.FS6
       -> C36 -> C_D_S.FS11
       -> C12 -> C_T_S.FS1
       -> C13 -> throttle_setting;
end cruise_control_CC3.impl;

system implementation cc3_app.impl
  subcomponents
```

```
    CC: system cruise_control_CC3.impl;
    BRAKE_PEDAL_SENSOR: device brake_pedal;
    TA: device throttle_actuator;
    BA: device brake_actuators_ext;
    ASA: device airbag_and_security_belt_actuators;
    CC_ON_OFF: device cruise_control_button;
    ENGINE: device engine;
    RESUME: device resume_button;
    SP_UP: device speed_up_button;
    SP_DN: device speed_dn_button;
    SETBUTTON: device set_button;
    AIRBAG_SEC_BELT: device airbag_and_security_belt_status;
    WHEEL_ROT_SENSOR: device wheel_rotation_sensor;
    DISTANCE_SENSOR: device distance_radar_sensor;
    OBJECT_RECOG: device object_recognition_sensor;
    SET_DISTANCE: device set_distance_button;
  connections
    C11: data port AIRBAG_SEC_BELT.airbag_status -> CC.airbag_status;
    C12: data port AIRBAG_SEC_BELT.security_belt_status ->
CC.security_belt_status;
    C21: data port CC_ON_OFF.cc_system_on_off -> CC.cc_system_on_off;
    C22: data port BRAKE_PEDAL_SENSOR.brake_pedal_status ->
CC.brake_pedal_status;
    C23: data port ENGINE.engine_status -> CC.engine_status;
    C24: data port RESUME.resume -> CC.resume;
    C25: data port SP_DN.decrease_speed -> CC.decrease_speed;
    C26: data port SP_UP.increase_speed -> CC.increase_speed;
    C27: data port SETBUTTON.set_speed -> CC.set_speed;
    C28: data port WHEEL_ROT_SENSOR.wheel_pulse -> CC.wheel_pulse;
    C29: data port CC.throttle_setting -> TA.throttle_setting;
    C30: data port DISTANCE_SENSOR.distance_to ->
CC.distance_from_distance_sensor;
    C31: data port DISTANCE_SENSOR.relative_speed ->
CC.relative_speed_from_distance_sensor;
    C32: data port CC.current_instantaneous_velocity ->
DISTANCE_SENSOR.instantaneous_velocity;
    C33: data port OBJECT_RECOG.distance_to ->
CC.distance_to_from_object_recognition_sensor;
    C34: data port OBJECT_RECOG.relative_speed ->
CC.relative_speed_from_object_recognition_sensor;
    C35: data port CC.brake_setting -> BA.cc_brake_input;
    C36: data port CC.current_instantaneous_velocity ->
OBJECT_RECOG.instantaneous_velocity;
    C37: data port CC.security_belt_setting -> ASA.cc_security_belt_setting;
    C38: data port CC.airbag_setting -> ASA.cc_airbag_setting;
    C39: data port SET_DISTANCE.set_distance -> CC.set_distance;
  flows
    ETE_F1: end to end flow BRAKE_PEDAL_SENSOR.Flow1 -> C22 -> CC.brake_flow_1
       -> C29 -> TA.Flow1
    ;
end cc3_app.impl;

system cc_computer
  features
    device_bus: provides bus access PC104_ISA_16BIT;
end cc_computer;

system CompanyZ_computer
end CompanyZ_computer;

system CompanyZ_cruise_controlCC1_system
end CompanyZ_cruise_controlCC1_system;

system CompanyZ_cruise_controlCC2_system
end CompanyZ_cruise_controlCC2_system;

system CompanyZ_cruise_controlCC3_system
end CompanyZ_cruise_controlCC3_system;

system implementation cc_computer.CompanyZ
```

```
  subcomponents
    CompanyZ_memory: memory SDRAM;
    CompanyZ_bus: bus PC104_ISA_16BIT;
    CompanyZ_processor: processor PENTIUM;
end cc_computer.CompanyZ;

system implementation CompanyZ_cruise_controlCC1_system.impl
  subcomponents
    CompanyZ_computer: system cc_computer.CompanyZ;
    CompanyZ_software: system cc1_app.impl;
  connections
    C1: bus access CompanyZ_computer.device_bus -> CompanyZ_software.device_bus;
end CompanyZ_cruise_controlCC1_system.impl;

system implementation CompanyZ_cruise_controlCC2_system.impl
  subcomponents
    CompanyZ_computer: system cc_computer.CompanyZ;
    CompanyZ_software: system cc2_app.impl;
  connections
    C1: bus access CompanyZ_computer.device_bus -> CompanyZ_software.device_bus;
end CompanyZ_cruise_controlCC2_system.impl;

system implementation CompanyZ_cruise_controlCC3_system.impl
  subcomponents
    CompanyZ_computer: system cc_computer.CompanyZ;
    CompanyZ_software: system cc3_app.impl;
  connections
    C1: bus access CompanyZ_computer.device_bus -> CompanyZ_software.device_bus;
end CompanyZ_cruise_controlCC3_system.impl;
```