Methods, Techniques and Tools for Product Line Model Verification

Raul Mazo, Camille Salinesi

Université Panthéon Sorbonne, Paris 1

Centre de Recherche en Informatique CRI

Internal report

Methods, Techniques and Tools for Product Line Model Verification

Raúl MAZO

Camille SALINESI

Abstract

Requirements for a product line have thus to be expressed in terms of features shared by all members of the product line, known as commonality, and distinct features of individual members, known as variability. Identifying and representing variability is an important aspect of product line devel opment. In order to be able to model and manage common and variable features, they have to be documented in a variability model.

Feature diagram (FD) is a notation that is currently used to express variability models. Feature diagrams model the variability of features at a relatively high level of granularity. Their main purposes are (i) to capture feature commonalities and variabilities, (ii) to represent dependencies between features, and (iii) to determine combinations of features that are allowed and disallowed in the product line model (PLM). All the above can present multiple problems in the models of product lines, problems that, from an industrial point of view, are highly expensive. Just like Pohl and other authors, we have not found in literature a method covering up the different criteria to be verified on a PLM. In the same way, we have found a lack of criteria unification with regard to the characteristics that must be verified, and a lack of language unification used in the rigid processes of verification found in literature. «To our knowledge, specialised techniques for software product line inspections, reviews, or walkthroughs have not be en proposed» [Polh *et al.* 05]. On the other hand, consistency checking of the requirement specification in domain engineering is still an open issue [Lauenroth, Pohl 07].

Motivated by these lacks, we suggest a PLM verification process focused in correctness evaluation on these types of models. We firstly do a bibliographic search that permit us make an inventory of some techniques. We then go on to formalisation work of each criterion, particularly those for model verification, with propositional logic. Next, we have do integration work through MAP formalism, in order to propose a PLM correctness verification process that can be carried out in different ways. We have validated this approach through a real case study and implementation of the proposed MAP process model in a computational tool.

Contents

Part IV Verification of Product Line Models

Part V Multi-method of Verification

Part VI Case Study and Tool Support

Part I

Background

1 Introduction

The use of computers and software products has enormously increased in the last years. In order to obtain high-quality products along with higher productivity, it is required to carefully analyze, model, specify and manage system requirements. And this tendency is extended to industry fields like automobiles and electronic device production. Requirements engineering is introduced to address such issues early in the development process. A well-established requirement engineering process ensures that product requirements are properly elicited, analyzed, documented, verified and managed.
Several other attempts have been made to increase the productivity and quality of goods. A very promising approach is the reuse or the production guided by product line practices. The main goal is to develop a model that represents the family of products, which is then customized to configure individual products. A product family is a collection of similar products with requirements that are common across the family and others features or requirements that are unique to individual products. In this approach it is possible to reuse product components and apply variability with decreased costs and time. Therefore well-established requirements engineering process, in order to produce the product line model and well-established model verification and validation process are essential for any product line practice. One error in the product line model entails a problem in each product of the family. For this reason, an appropriate process of verification and validation (V&V), centred in the product line model and not only in each particular configuration, is necessary to guarantee the success in this paradigm of production.

We have found in our literature search, that there is no standard theory on V&V applied to product line models. Neither is there a standard "box of tools" from which tools are taken in a natural order to verify a product line model or a particular configuration model; see [Davis 92] and [Landry and Oral 93]. Neither is there a list of criteria, expressed in a common language, which must be applied on models through V&V process. In our research work, we have made a list from dispersed criteria or invariants to be evaluated on a product line model or on a particular product model. Also, we have unified the language for these set of criteria desired to be verified in a product line model. Thus, our objective is to propose a Verification process in order to improve the weakness identified above.

1.1 Requirements Engineering

Requirements engineering (RE), in software engineering, is a term used to describe all the tasks that are into the instigation, scoping and definition of a new or modified computer system. Requirements

engineering is an important part of the software engineering process; whereby business analysts or software developers identify the necessities or the requirements of a customer; having identified these requirements they are then in a position to find a solution.

RE provides the global context for our work. As every research domain, there are many definitions about what requirements engineering (RE) is, Nuseibeh and Easterbrook, in their 2000 ICSE roadmap paper for RE [Nuseibeh, Easterbrook 00] introduce RE as follows: "The success' main measure of a software system is the degree to which it meets the purpose for what it was intended. Broadly speaking, software systems requirements engineering is the process of discovering this purpose, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation."

Earlier it was considered that RE was relatively easy and corresponding to introduce the software development process. However, it became clear soon, that RE is a very important and problematic stage. As many failures at system developing have been caused by mistakes admitted during the RE and it was too expensive or even impossible to correct them and to satisfy clients' requirements in the given time. Now, in some development models, RE process must get involved during all software's life cycle.

At RE stage it is possible to construct a stable model of the future system on the basis of requirements prior to the beginning of designing and development process to prevent failures in the future and to formalise what the future system must be. Complete requirements represent a declarative description of the future system. That's why Software Engineering (SE) researchers emphasized on the fact that requirements describe what is to be done, but not how they are implemented [Cockburn 00].

1.1.1 Requirements engineering process

In this work we assume a classical vision of RE, in which the process starts with stakeholder identification. RE then goes on with requirements acquisition or elicitation. Then we move on to requirements analysis and concept formation. Once a set of consistent and a set of relatively complete requirements have been gathered and analysed, proper requirements specification or requirements facet modelling can take place. It is sure that during requirement modelling we find that requirements verification is needed. At the end of requirements modelling we have to perform a requirement validation, which serves to make sure that the requirement development phase has achieved the right

requirements. A final stage of requirements management or requirements satisfiability and feasibility is needed to complete a full and proper requirements development process. It is a little description of each RE process's main activities:

a. Stakeholders identification

At the very outset of a development project identify all possible and potential requirement stakeholders. In order to face this phase, we propose some recommendations to consider. It is better to include a large number of elicited requirements than to exclude some of them which might cause trouble later on, and even more, when such requirements rightfully intervene. Be prepared, throughout a project, to revise the list of requirement stakeholders. At the very outset of a development project define, together with designated requirement stakeholders, their role, their rights and duties, etc. Through the requirement stakeholders' identification is necessary revising the roles of stakeholders.

b. Requirements acquisition or elicitation

Requirement elicitation is the first stage in considering the problem that software system should be able to solve. This process is carried out once the context definition and the software's goal have been established. The task of requirements elicitation is to establish boundaries and requirements for the software system. It is carried out by interaction with stakeholders and detailed studying of corresponding knowledge domains. Requirement elicitation has always been a human activity. There are several techniques and methods of requirement elicitation. The most widespread techniques are interviews, scenarios, prototypes, facilitated meetings, observation, etc.

At this stage the relationship between developers of the system and the customer should be established.

The requirement elicitation is variously termed "requirements' capture", "requirements' discovery" and "requirements' acquisition".

Having been gathered during this stage, requirements may be checked for quality using different methods and tools [Lami 05], [Hooks 93], [Firesmith 03]. The quality of requirements is an important feature for the following stages. Several rules for writing quality requirements are discussed in [Wiegers 99] and some examples are given.

c. Requirements analysis and concept formation

After some requirements were discovered they must be analysed to:

- Necessity (need for the requirement);

- detect and resolve drawbacks in them (for example, consistency, conflicts, ambiguity situations, completeness, etc);
- improve their quality;
- they must be structured and refined;
- discover the bounds and properties of the system;
- discover how system will interact with the environment;
- another necessary analysis.

After this stage the requirements have to be described clear enough to enable their specification, verification and validation.

For providing more convenient analysis procedures, requirements may be structured by different characteristics. For example, whether the requirement is functional or non-functional:

Functional requirements describe the functions that the software is to execute (formatting some text, modulating a signal). They are sometimes known as *capabilities*.

Non-functional requirements are the ones that act to constraint the solution. Non-functional requirements are sometimes known as constraints or quality requirements. They can be further classified according to whether they are performance requirements, maintainability requirements, safety requirements, reliability requirements, or one of other types of software requirements.

One of the most common procedures in requirements analysis is negotiation. It may be used to resolve problems with requirements where conflicts happen between two stakeholders requiring mutually incompatible features (conflicting requirements). Requirements that seem problematic are discussed and the stakeholders involved present their views about the requirements. After negotiation the requirements are prioritised and a compromise set of requirements may be agreed upon. Generally, this will involve making changes to some of the requirements, what also may cause new problems appearing.

Other techniques used for requirements analysis are: requirements classification, conceptual modelling, requirements' negotiation, prioritization, architectural design and requirements allocation.

d. Requirements specification or requirements modelling

First of all, let's start with the definition of Software Requirements Specifications (SRS) process. There will probably be different definitions more or less completed.

The definitions in the work are commonly based on an excellent source for definitions in Software Engineering discipline - the IEEE Computer Society [IEEE 98], [IEEE 04]. Here, the SRS process is defined as "a process result of which are unambiguous and complete specification documents".
It is accepted to consider the SRS is a complete description of the system behaviour to be developed. It includes a set of use cases that describe all of the interactions that users will have with the software, numerical values, limits and measurable attributes which may be checked on the working system.
Briefly, SRS is a document (paper or electronic), which defines (specifies) the Software System.

e. Requirements verification

Requirement verification is the process in which some requirements specifications (RS) are being analysed in order to find out whether what is being described satisfied certain properties. Some of these properties are consistency between the RS model elements, correctness, validity or satisfiability of the RS model constraints, suitability and usability of each RS model element and richness of the model.

f. Requirements validation

Requirement validation is the process, and the resulting documents, in which some requirement specification models are being inspected by both requirement stakeholders and requirement engineers, and in which, whatever is being prescribed, is being validated with reference to the elicitation report and with respect to whatever the requirement stakeholders might now realise about their expectations. In order to achieve this validation, some properties, like *correctness, unambiguousness, completeness, stability, verifiability, modifiability and traceability,* must be considered.

g. Requirement management or requirement satisfiability and feasibility

Requirement management is a relatively new branch in RE process. It is the activity concerned with the effective control of information related to system requirements. Requirement management process is carried out together with other engineering processes. The beginning of this process should be planned at the same time that the process of initial requirement elicitation starts. Directly

requirement management process should begin right after the draft version of the requirements' specification is ready.

1.2 Product Lines Engineering

When we talk about Product Lines Engineering [Pohl *et al.* 05] we might think in the way that goods have being produced and all the different changes experienced throughout. Formerly goods were handcrafted for individual customers. Gradually, the number of people who could afford to buy various kinds of products increased. Pioneer with the invention of the *production line* in the domain of automobiles, Ford starts the production for a mass market much more cheaply than individual product creation on a handcrafted basis. However, the production line reduced the possibilities for diversification and that is why Ford was only able to produce black cars.

Roughly, both types of products, individual and mass produced can be identified in the software domain as well: they are denoted as individual software and standard software. Generally, each of these types of products has its drawbacks. Individual software products are rather expensive, while standard software products lack sufficient diversification.

Example from the Camera World

In 1987, Fuji released the Quicksnap, the first single-use camera. It caught Kodak by surprise: Kodak had no such product in a market that grew from then on by 50% annually, from 3 million in 1988 to 43 million in 1994. However, Kodak won back market share and in 1994, it had conquered 70% of the US market. How did Kodak achieve it? First, a series of clearly distinguishable, different camera models was built based on a common platform. Between April 1989 and July 1990, Kodak reconstructed its standard model and created three additional models, all with common components and the same manufacturing process. Thus, Kodak could develop the cameras faster and with lower costs. The different models appealed to different customer groups. Kodak soon had twice as many models as Fuji, conquered shelf space in the shops and finally won significant market share this way (for details see [Clark and Wheelwright 1995]).

1.2.1 Importance of PLE

As we have already discussed, the main goal that product line engineering pursues is to provide customised standard products at reasonable costs. In this section, we briefly outline the key features and motivations for developing goods under the product line engineering paradigm.

Reduction of Development Costs

One of the most relevant purposes of an engineer is to create solutions that provide human benefices and economical profits. An essential reason for introducing product line engineering is the reduction of costs. When artefacts from the platform are reused in several different kinds of systems, and a standard work is implemented; this implies a cost reduction for each system. Before the artefacts can be reused, strategies investments and even a detailed planning are necessary for creating them. This means that the company has to make an up-front investment to create the platform before it can reduce the costs per product by reusing platform artefacts. Figure 1-3-1 shows the *accumulated costs* needed to develop *n* different systems. The solid line sketches the costs of developing the systems independently, while the dashed line shows the costs for product line engineering. In the case of some few systems, the costs for product line engineering are relatively high, whereas they are significantly lower for larger quantities (one of the main microeconomics theories). The location at which both curves intersect marks the *break-even point*. At this point, the costs are the same for developing the systems separately as for developing them by product line engineering. Empirical investigations revealed that, for software, the break-even point is already reached around three systems.
A similar figure is shown in [Weiss and Lai 1999], where the break-even point is located between three and four systems. The precise location of the break-even point depends on various characteristics of the organisation and the market it has envisaged: the customer base, the expertise, and the range and kinds of products. The strategy that is used to initiate a product line also influences the break-even point significantly [McGregor et al. 2002].

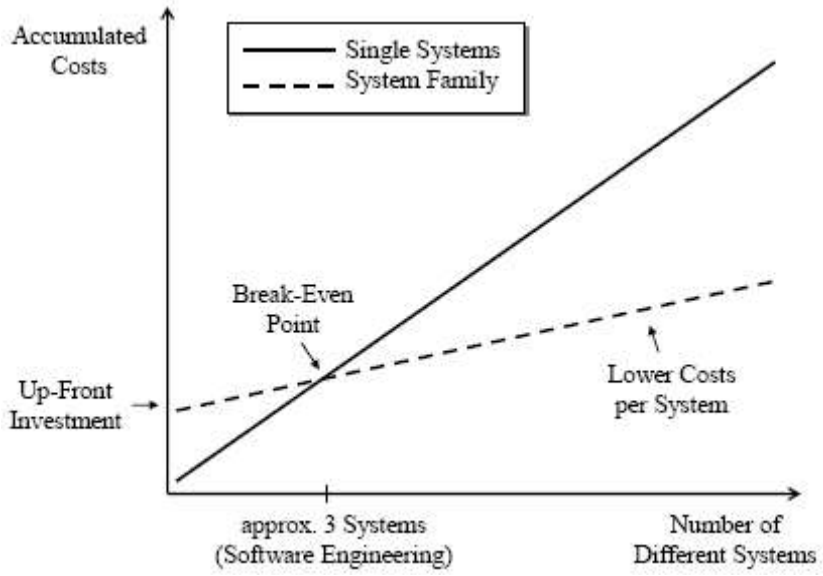


Figure 1-2-1: Costs for developing n kinds of systems as single systems compared to product line engineering

Enhancement of Quality

The artefacts in the platform must be reviewed and tested in many products and different processes. They have to prove their proper and correct functioning in more than one kind of product. The extensive quality assurance implies detecting faults, failures and improper work methods to correct them, thereby increasing the quality and reliability of all products.

Reduction of Time to Market

Often, a very critical success factor for a product is not only the shelf life but when you begin with a project is time to launch it in the market. For single-product development, we assume it is roughly constant, mostly comprising the time to develop the product. For product line engineering, the time to market indeed is initially higher, as the common artefacts have to be planned and built first. Yet, after having passed this hurdle, the time to market is considerably shortened as many artefacts can be reused for each new product (see Figure 1-3-2).

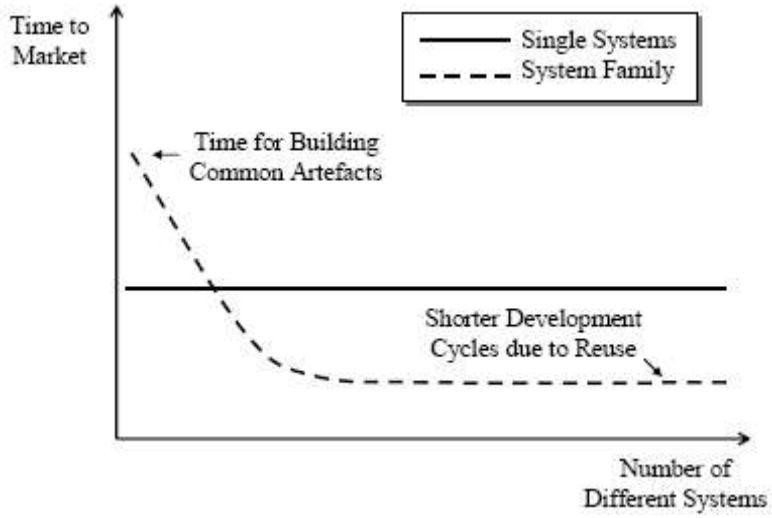


Figure 1-2-2: Time to market with and without product line engineering

1.2.2 The PLE process

As shown in Figure 1-2-3, the PLE process is split along this line into two sub processes: domain engineering and application engineering [Pohl *et al.* 05].

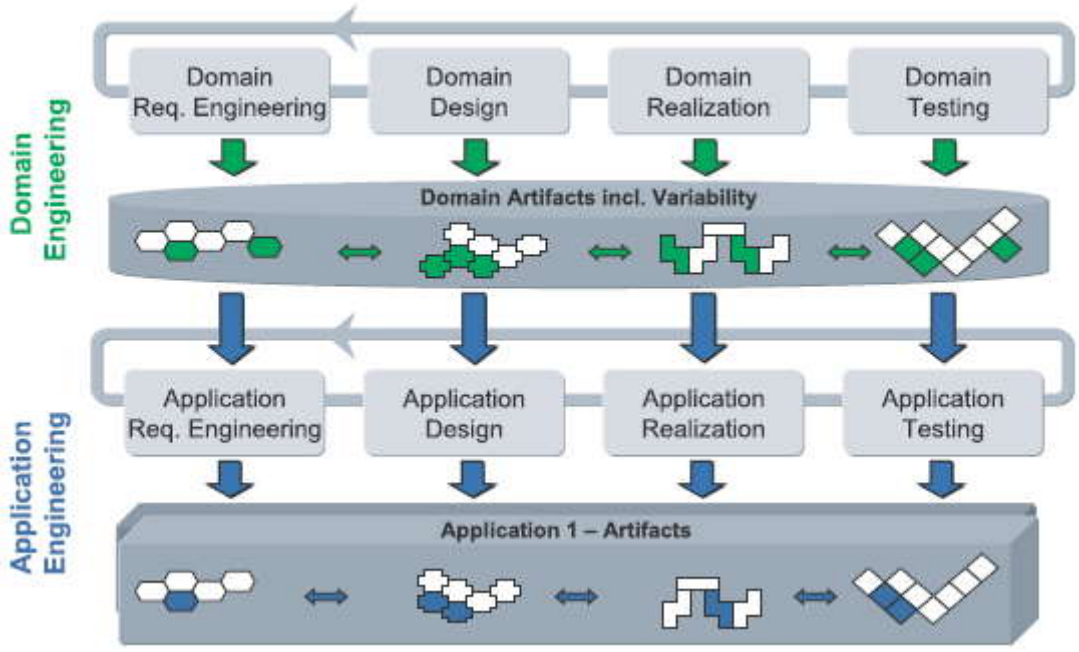


Figure 1-2-3: Schema of the PLE process [Pohl, Metzger 06].

Domain Engineering

The principle of PLE is to exploit common elements of a number of different systems by developing them as one single core while still allowing differences between these systems. Commonalities, i.e. functions or properties that systems of the future product line have in common [Coplien *et al.* 98], and differences (generally called variabilities [van Gurp *et al.* 01]), however, have to be defined into the PL model during this sub-process.
The domain engineering process is the process in which the scope is decided. Based on this scope, the commonalities and differences of systems in the PL are defined.

Application Engineering

This process is based on the Domain Engineering once commonalities have been exploited and implemented as reusable artefacts and variability have been defined as variation points. The application engineering process now exploits this variability to apply it at the correct moment of the process.
Each variation point is analysed and one of its variants chosen (we can say that the variation point is bound). Once all variation points are bound to variants, a particular system of the PL is initiated. The reusable artefacts will be then assembled and after successful integration tests, the development of the new products is finished.

1.2.3 Commonality and Variability

To facilitate mass customisation, the artefacts used in different products have to be sufficiently adaptable and flexible to fit into the different systems produced in the product line. This means that throughout the development process we have to identify and describe where the products of the product line may differ in terms of features that they provide, the requirements they fulfil, or even in terms of the underlying architecture, etc. Thus we have to provide flexibility in all those artefacts to support mass customisation.
A very well known case is that of the different cars of the same product line which may have different windshield wipers and washers. Ingeneers design cars in a way that allows a common approach to support the different motors for these different windshield wipers/washers, their different sizes, etc. Such flexibility comes with a set of constraints. If you drive a convertible, you do not want a rear window washer splashing water onto the seats! Therefore, the selection of a convertible car means the flexibility

that the availability of the windshield wipers and washers is restricted, so that the rear window washer is disabled when the car roof is open.
This flexibility is a precondition for mass customisation; it also means that we can predefine what possible realisations shall be developed (there are only a certain number of windshield wiper configurations conceivable). In addition, it means that we define exactly the places wh ere the products can differ so that they can have as much in common as possible. The flexibility described here is called "variability" in the product line context. This variability is the basis for mass customisation.

1.3 Product Line Models

Product line models or product line diagrams are commonly used to define the valid combinations of elements in a product line. Not all elements are compatible.
There are two types of requirements specifications [Faulk 01] in product line engineering: The product line (or domain) requirements specification and the product (or application) requirements specification. The Product Line Requirements Specification (PLRS) or Product Line Model (PLM) as we will refer to this concept in the rest of document is developed during domain engineering. It contains all the common and variable requirements of all products of the product line. Requirements specification of a particular product of the product line is commonly named Product Requirements Specification (PRS) or Product Line Configuration (PLC), we will use the latest in the rest of document. The PRS for a particular product is derived from the PML [Pohl *et al.* 05], [Bühne *et al.* 06].
When deriving a PLC, all common requirements defined in the PLM become part of each PLC. The variable requirements can be added to a PLC by selecting variants from the variability model and thereby adding the variable requirements related to the selected variants. We can not only make a derivation, we also can create an independent configuration, and then contrasting it with its product line model. Another existing relation between both can be to try to deduce PLM from a set of PLCs.

Product line model notations

The Feature Diagram (FD) notation used in Figure 1-4-1 is built based on the FORE formalism. FORE (Family Oriented Requirements Engineering) [Streitferdt 03] is a method proposed by Riebisch and his work group at Alexandria project [Alexandria]. We have chosen this formalism because it offers all of the modeling facilities of previous notations to its launching. Besides, it includes some of the characteristics and construction rules particularly important at the moment of implementing the method proposed in this work. For example, FORE introduce the use of cardinalities [Riebisch *et al.* 02], enriching feature

diagrams with UML cardinalities. Also, FORE proposes that all variant features grouped with relations of cardinalities be optional features.

Some of the properties that characterise FORE notation are:

- A feature diagram is a Directed Acyclic Graph (DAG)
- A feature is a node of this graph
- Relationships between features are represented by links. There are two types of relationship, variant dependency and transverse dependency.
- Variant dependencies can be of kind mandatory or optional. The sense of relations is determined by a white or black circle at the end of the line. Black circle represents a mandatory relationship between father and child features, that is, if father is chosen, then child feature must be selected too. White circle represents an optional relationship between father and child features, which is, if father is chosen, child can or can not be selected.
- Transverse dependency can be of two kinds: an exclude relation or a require relation. An exclude relation is represented by a two headed arrow ($\leftrightarrow$) and a require relation is represented by one headed arrow ($\rightarrow$). The direction of relations is determined by an arrow at the end of a dotted line.
- Optional relations with the same father can be grouped in a set. A relation can be member of one and only one set.
- A set have a cardinality that indicates the minimal and maximal number of features than it is possible to choice. The set of possible values is: 0.1, 1, 0.N, 1.N, N, p, 0.p, 1.p, p.N, m..p, 0..* and 1..*.
- Graphically, the set of features grouped by a transverse dependency relationship is represented by a line or arch comprising all the implicated relations and a couple of symbols as shown in the previous numeral.
- Grey rectangles are destined to represent features that have as function to facilitate the structure of the model [Streitferdt 03].

Although this notation seems adequate to construct models of product lines by means of features and that eliminates most of the ambiguities, the notation FORE does not, nor its predecessor FODA, establish utilization rules to guide the engineer in the modelling process. It does not establish either any rule permitting to identify a big of ambiguities that exit even in this type of models. Those ambiguities will be treated with detail in chapters III, IV and VI .

Another notation is VFD (varied feature diagrams) introduced by Schobbens et al. in [Schobbens *et al.* 06]. This one is just another notation among several other new or adapted FD notations that have been proposed since the original introduction of FDs as part of the FODA method [Kang *et al.* 90]. Other extensions of FODA are the following: FORM, an extension of FODA [Kang *et al.* 98]. FeatuRSEB [Griss *et al.* 98], an integration of FODA and the Reuse-Driven Software Engineering Business (RSEB) [Jacobson *et al.* 97] are two more propositions based on features. Van Gurp et al. extend FeatuRSEB to include binding times [Van Gurp *et al.* 01]. Riebisch et al. replaced operator nodes by more general cardinalities [Riebisch *et al.* 02]. Batory introduces propositional constraints defined among features [Batory 05]. Czarnecki suggest cardinalities and provide a formalisation for these diagrams [Czarnecki *et al.* 05]. PLUSS [Eriksson *et al.* 05] is another extension of FeatuRSEB. All the above mentioned notations are only a subset of all existing FD notations.
At the moment, there is no unified and universally accepted notation.
However, most of the notations cited above can be defined using a general parameterised FD definition, as proposed in section 1.3.2.

1.3.1 Feature Diagrams

Feature diagrams (FDs) are used to model and manage variability of a PL. FDs capture commonalities and variabilities by structuring the represented information in the form of a feature tree, or a directed acyclic graph (DAG). The tree or the DAG represents the decomposition of one feature into an arbitrary number of sub-features. The root feature is called concept and represents the system itself.

A FD is a hierarchical decomposition of the system into features and subfeatures. This hierarchy can be either a tree or DAG, which allows a feature to have more than one parent node. This structure is supported by FORE formalism, in opposition to FODA notation that is limited to trees. Most authors who extended the tree structure to a DAG argued for more expressiveness.

1.3.2 Formal semantic

In literature we have found several formal definitions of feature models [Czarnecki, Pietroszek 06], [Metzger *et al.* 07]. These formal definitions have not adapted to our verification process based in evaluations of logical rules in order to check both static and dynamic properties of product line models and not only product line configurations.

For us, a Feature Diagram *d* (based in the FD metamodel that we will present in next sections) is a eight-tuple (*F, r, VD, AVD, TD, ATD, A, C*) where:

F is the set of features

$r \in F$ is the root of *d*, *r* is unique $|r| = 1$

$VD \subseteq F \text{ X } F$ is the set of variant dependencies edges,

AVD is an application of *VD* set at $\{\circ, \bullet\}$, $\circ$ representing optional dependency and $\bullet$ representing mandatory dependency. $(f, f') \in VD$ will rather be noted $f \circ f'$ for optional relations and $f \bullet f'$ for mandatory relations, where *f* is the child and *f'* is the father

$TD \subseteq F \text{ X } F$ is the set of transverse dependency edges

ATD is an application of *TD* set at $\{\rightarrow, \leftrightarrow\}$, $\rightarrow$ representing require dependency and $\leftrightarrow$ representing exclude dependency. $(f, f') \in TD$ will rather be noted $f \rightarrow f'$ for include relations and $f \leftrightarrow f'$ for exclude relations

$A \subseteq AVD^n$ is a finite set of arcs, each couple of variant dependencies participating in A must be an optional dependency.

C is an application of arcs set at $(\mathrm{N \text{ X } N} \cup \{*\})$ that define the cardinality of A

For the variant dependency $v = (f, f')$, with f, f' in *F*, the node *f* is so-called father of *v* and the node *f'* is so-called child of *v*.

The ATD application is defined by TD :

$TD \Rightarrow (\rightarrow \oplus \leftrightarrow)(f_i, f_j) \quad i \neq j$

Examples :

$td_1 \in TD \Rightarrow (f_4 \rightarrow f_2)$

$td_2 \in TD \Rightarrow (f_4 \leftrightarrow f_6)$

The cardinality application is defined by C :

$A \Rightarrow \left[\mathrm{N \text{ X } N} \cup \{*\}\right] \; AVD^n$

Example :

$a_1 \in A \Rightarrow [0,2] \; (f_2, f_3)$

Example 1-3-1:

The follow is the graphic representation of Feature Diagram $d = (F, r, VD, AVD, TD, ATD, A, C)$ defined by:

$F = \{ f1, f2, f3, f4, F5, F6 \};$

$r = \{ f1 \};$

$VD = \{ (f2 \circ f1), (f3 \circ f1), (f4 \bullet f1), (f5 \circ f2), (f6 \circ f2) \};$

$TD = \{ (f4 \rightarrow f2), (f4 \leftrightarrow f6) \};$

$A = \{ [0,2] (f2,f3) \},$

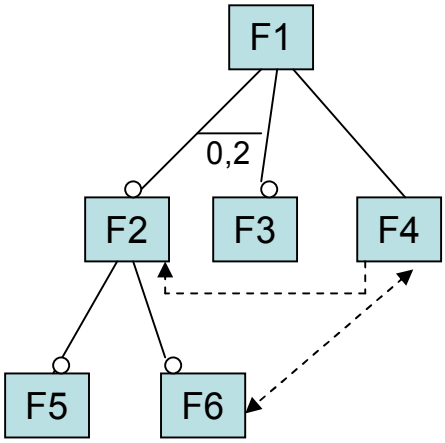


Figure 1-3-1: example of product line model in FORE notation.

1.4 Automated Analysis of Feature Models

The automated analyses of FMs is usually performed in two steps: i) The FM is translated into a certain logic representation ii) Off-the-shelf solvers are used to extract information from the result of the previous translation such as the number of possible products of the feature model, all the products following a criterion, finding the minimum cost configuration, etc [Benavides *et al.* 06].

According to [Benavides *et al.* 07], the current implementation of the framework integrates three of the most commonly used logic representations proposed for the automated analyses of feature models: CSP, SAT and BDD. A complete performance test of solvers dealing with such representations and details about the translation of an FM into a CSP, SAT and BDD were introduced in [Benavides *et al.* 06b], [Benavides, Ruiz 05].

Three of the most commonly used logic representations proposed for the automated analyses of feature models: CSP, SAT and BDD.

Constraint Satisfaction Problem (CSP)

A *Constraint Satisfaction Problem* (CSP) [Tsang 95] consists on a set of finite domains variables, and a set of constraints restricting the values of the variables. Constraint Programming can be defined as the set of techniques such as algorithms or heuristics that deal with CSPs and the purpose is to find combinations of values in which all constraints are satisfied based on a common objective. The main ideas concerning the use of constraint programming on FM analysis were stated in [Benavides *et al.* 05].

Constraint Programming is the most flexible proposal. It can be used to perform the most of the operations currently identified on feature models [Benavides *et al.* 06a]. However, constraint programming solvers reveal a weak time performance when executing certain operations on medium and large size feature models calculating the number of possible combinations of features due most of the time to a high number of variables [Benavides *et al.* 06c].

Boolean Satisfiability Problem (SAT)

A propositional formula is an expression consisting on a set of boolean variables (literals) connected by logic operators ($\neg$,$\wedge$,v, $\Rightarrow$, $\Leftrightarrow$). The *propositional satisfiability problem* (SAT) [Cook 71] consists on deciding whether a given propositional formula is satisfiable, i.e., a logical values can be assigned to its variables in a way that makes the formula true. The basic concepts about the using of SAT in the automated analysis of FMs were introduced in [Batory 05].

The performance results of SAT solvers are slightly better than the results of CSPs. however this approach is not so powerful [Benavides *et al.* 06c]. The best of our knowledge, it is that there is not any approach in which feature models attributes can be translated to SAT in order to perform operations as maximizing or minimizing attribute values.

Binary Decision Diagrams (BDD)

A *Binary Decision Diagram* (BDD) [Bryant 86] is a data structure used to represent a boolean function. A BDD is a rooted, directed, acyclic graph composed by a group of decision nodes and two terminal nodes called 0-terminal and 1-terminal. Each node in the graph represents a variable in a Boolean function and has two children nodes representing an assignment of the variable to 0 and 1. All paths from the root to the 1-terminal represents the variable assignments for which the represented Boolean

function is true meanwhile all paths to the 0-terminal represents the variable assignments for which the represented Boolean function is false.

Although the size of BDDs can be reduced according to some established rules, the weakness of this kind of representation is the size of the data structure which may vary between a linear to an exponential range depending on the variable ordering [Bryant 86]. Calculating the best variable ordering is an NP-hard problem. However, the memory problem is clearly compensated by the time performance results offered by BDD solvers. While CSP and SAT solver are incapable of finding the total number of solutions of medium and large size feature models in a reasonable time, BDD solvers can work it out in an insignificant amount of time, so it justifies its usage at least on counting operations.

1.5 Conclusion

The objective of this first chapter was to survey different RE and product line relevant definitions to our work. This includes not only notations found in literature, but also owned definitions necessaries for our approach such as a formal semantic of a PLM. These definitions will be used in the rest of this document. In this chapter we also highlight the importance of product line engineering for the industry and its relation with RE.

The next chapter presents the problematic situation that has motivated this research work and the way that we have used to tackle it.

Part II

Research Presentation

2. Research problem, methodology and justification.

2.1 Research problem

We have observed that the few solutions developed over the past ten years have not been integrated into a coherent and flexible process of V&V. The penury of methods for the formal verification of conceptual models and the urgency of proposing well adapted approaches are recognized by the scientific community [Wang et al. 05], [Polh et al. 05], [Lauenroth, Pohl 07]. Likewise, the lack of tools for the industry was cited in the Fourth Product Line Engineering Workshop acts [Bass et al. 99], in 2001 by Zave [Zave, 01] and again recently by Padmanabhan and Lutz [Padmanabhan, Lutz 05]. The recent award of Turing Prize '07 by Joseph Sifakis for his work in model checking confirms the importance of this topic, and the importance to continue its exploration.

2.2 Research methodology

The methodology followed in this investigation can be summarized in following steps:

1. Bibliographic search that permit us make an inventory of some techniques
2. Formalisation work of each criterion, particularly those for PLM verification, with propositional and first order logic notations [Bradley, Manna 07].
3. Integration work through MAP formalism, in order to propose a PLM correctness verification process that can be carried out of different ways.
4. Approach validation through a real case study and implementation of the proposed MAP process model in a computational tool.

3.3 Justification

Graphical representation of PLMs can be directly represented in logical constraints that can be evaluated by means of SAT solvers or constraint scheduling. MAP is a formalism that allows representing multi-process. That is, a MAP representation is not a sequence predefined of tasks. It is rather a set of tasks that may be organized in a different way, in terms of situation and intention of the person using the map.

We have used propositional logic and first order logic in order to represent the set of invariant or logic rules that must respect a PLM. Propositional logic and first order logic are also known as propositional calculus and predicate calculus, respectively, because they are calculi for reasoning about propositions ("the sky is blue", "this comment references itself") and predicates ("x is blue", "y references z"), respectively. Propositions are either true or false, while predicates evaluate to true or false depending on the values given to their parameters (x, y, and z). And to represent the process we have used MAP formalism. Loosely speaking, a Map [Rolland *et al.* 99] is a navigational structure in the form of a graph where nodes are intentions and edges are strategies. It is possible to follow different strategies for each couple of target/source intentions, thus dynamically determining different solution paths between start and end. So a Map is a modelling formalism that permits to represent several processes on the same design.

Part III

State of the art on V&V in RE

3. State-of-the-art on Validation and Verification in Requirements Engineering

3.1 Definition of Verification

Formal verification is the process of checking whether a design satisfies some requirements (called properties or invariants in this document).
Verification work typically proceeds as follows [Bjorner 06]: "Desired properties of the requirement's model, properties that do not transpire immediately from the proofs by symbolic testing or formal proofs, or model checking, is, or are, performed in order to check that the desired property(ies) holds of the requirements model".

3.2 Definition of Validation

By requirements validation we shall understand a process with some resulting documents in which some requirements' prescriptive artefacts (documents, models, etc.) are being inspected by both requirements' stakeholders and requirements' engineers. This includes the pointing in, the pointing out, if necessary, of inconsistencies, incompletenesses, conflicts and errors of prescription that may change the elicitation report.

According to [Bjorner 06], in order to perform domain validation, the validators need the following (input) documents: (i) the list of domain stakeholders; (ii) the domain acquisition documents: questionnaire, and the collection of indexed description units; (iii) the rough-sketch, terminology, narrative, and possibly the formalisation documents that constitute the domain description proper; and (iv) the domain analysis and concept formation documents. That is, the validators need access to basically all documents produced in the domain modelling effort. In order to complete domain validation, the validators produce the following output documents: (i) A possibly updated domain stakeholder document; (ii) Possible updated domain acquisition documents, (iii) Possible updated rough sketches, terminology, narrative and the formalisation documents; and (iv) A domain validation report. We now cover some aspects of the necessarily informal validation process.

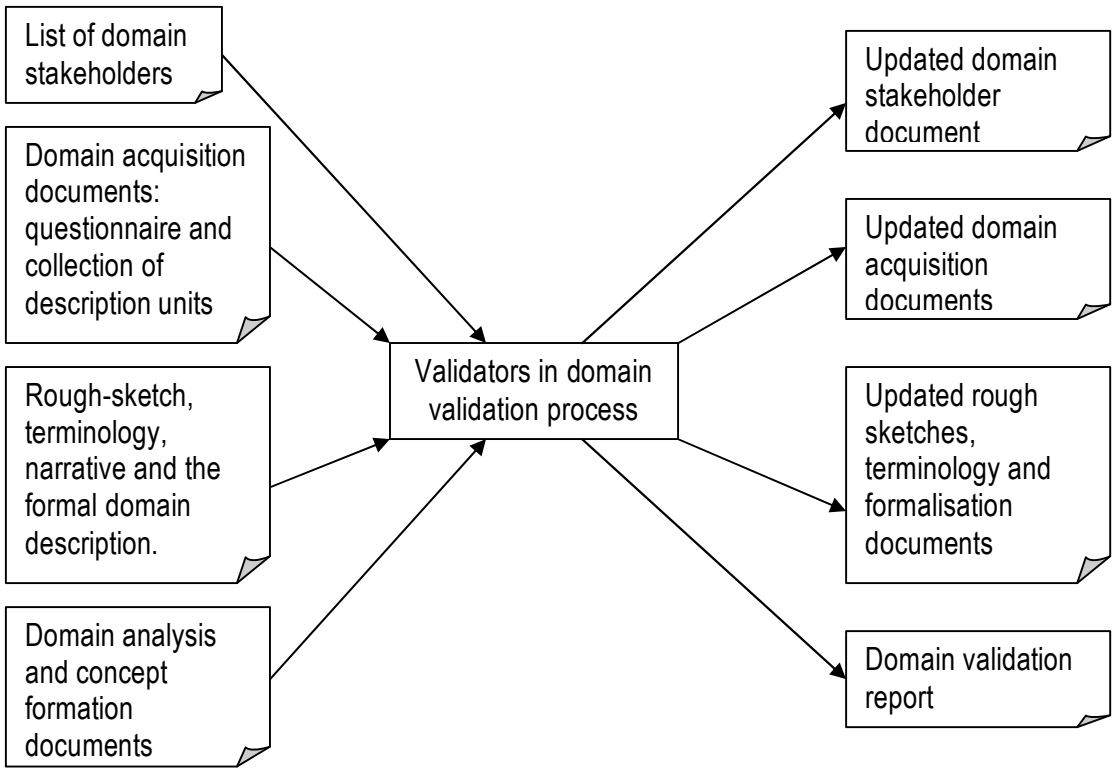


Figure 3-2-1: Domain validation input and output documents

Validation works typically proceeds as follows [Bjorner 06]: requirements engineers, requirements stakeholders and review, line by line, the domain model, holding it up against the previously elicited requirement prescriptions' units, while then noting down any discrepancies.
In doing requirements' validation, requirements' stakeholders usually read the informal, yet precise and detailed narrative prescriptions. No assumption is made as to their ability to read formalisations. On the contrary, it is assumed that they cannot read formal specifications.

3.3 Verification vs. Validation

The following authors give their point of view concerning the difference between verification and validation:

According to Boehm, in verification we examine whether our requirements model is logic or according to such requirements engineers want it to be, "Verification gets the requirements model right" [Boehm 81]. In validation we examine the requirements' model to make sure we are modelling what the

requirements' stakeholders think that the domain is, "Validation gets the right requirements model". Usually verification precedes validation.

According to Kuloor and Eberlein, "Requirements are verified to check their completeness, precision and suitability in the requirements verification stage [Kuloor, Eberlein 02]. Formal reviews, prototyping and requirements testing are some of the techniques used for requirements verification. A product family has more than one product and most of the requirements are common across the family".

Ponsard *et al.*, suggests that "verification is about making sure the system is correct, especially with respect to formal semantics of the goal model. Validation is about making sure the system being built is the system the user is expecting" [Ponsard *et al.* 05].

Verification:

A set of goals G1, ... , Gn refines

a goal G in a domain theory D if

the following conditions holds:

$$Completeness: G_1, G_2, ..., G_n, D \models G$$

$$Minimality: \wedge_{j \neq i} G_j, D \models \neq G \ \forall i \in \{1,2,...,n\}$$

$$Consistency: G_1, G_2, ..., G_n, D \models \neq false$$

In [Probert *et al.* 2003] authors claim that to verify a model is to make sure that it is created correctly, which means there is no defect or error present. Examples include deadlocks, live-locks, and implicit (missing) definitions in the model that are introduced via the design process itself.

"To validate a model is to make sure we create the right model, which means the model created has to match the requirements. Simulation involves making an executable program based on a system specification and running this executable program to understand and debug the behaviour of the system specification".

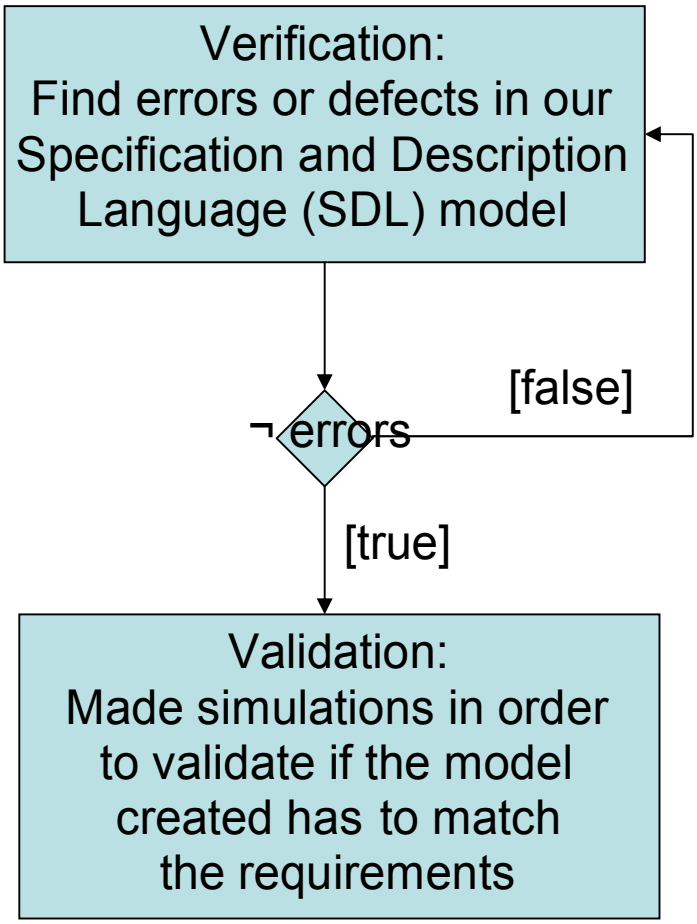


Figure 3-3-1: Relationship between verification and validation proposed by Probert *et al.*

According to [Bahill, Henderson 2005], verifying requirements is the process aiming to proving that each requirement has been satisfied. Verification can be done by logical argument, inspection, modelling, simulation, analysis, expert review, test or demonstration. Bahill and Henderson also define validating requirements as the process to ensure that the set of requirements is correct, complete, and consistent; a model can be created to satisfy the requirements; a real-world solution can be built and tested to prove that it satisfies the requirements.

Otherwise, Easterbrook holds that the terms Verification and Validation are commonly used in software engineering to mean two different types of analysis [Easterbrook 96]. The usual definitions are:
Verification: Are we building the system correctly?
Validation: Are we building the correct system?

Requirements' verification and validation ends up with a report which either accepts the requirements' model, or points out needs to correct the elicitation report, the requirements analysis and concept formation report, and the requirements model.

Thus requirements verification and validation can be expected to be an interactive process alternating with further requirements elicitation report work, possibly with further requirements analysis and concept formation work; and ending with further requirements verification and validation work.

In the following section we will split the characteristics found in the literature which we consider relevant to evaluate in an RS document. Two groups will be created, based on the criterion described above verification / validation. Some characteristics should be verified and validated to ensure a complete evaluation.

In the rest of the document we will assume verification and validation as two complementary processes. We will consider verification as the first step of the validation process. In this sense, the fundamental strategy of verification is to identify and to reduce errors in a model, that is, verification deals with mathematics. And validation addresses the question of the fidelity of the model to specific conditions of the real world, that is, validation deals with requirements.

3.4 Desirable characteristics to verify

In general, the most important elements to respect in order to guarantee an acceptable requirement specification quality level are defined in [Lami 2007] and can be resumed in the next fourth tables:

Indicator	Description
Vagueness	When parts of the sentence are inherently vague (e.g., contain words with non -unique quantifiable meanings). For example: easy, strong, good, bad, useful, significant, adequate and recent.
Subjectivity	When sentences contain words used to express personal opinions or feelings. For example: similar, similarly, having in mind, take into account and as [adjective] as possible
Optionality	When the sentence contains an optional part (a part that can or cannot be considered). For example: possibly, eventually, in case of, if possible, if appropriate, if needed
Implicity	When the subject or object of a sentence is generically expressed by demonstrative

	adjectives (e.g., this, these, that, those) or pronouns (e.g., it, they). Implicit adjective (e.g., previous, next, following, last), or preposition (e.g., above, below).
Weakness	When a sentence contains a weak verb. For example: could, might and may.

Table 3-4-1: Ambiguity Indicators for RE's verification and validation

Indicator	**Description**
Multiplicity	When a sentence has more than one main verb or more than one subject.
Readability	The readability of sentences is measured by the Coleman-Liau Formula of readability. The reference value of this formula for an easy-to-read technical document is 10. If the value is greater than 15, the document is difficult to read.

Table 3-4-2: Understandability Indicators for RE's verification and validation

Indicator	**Description**
Under-specification	When a sentence contains a word identifying a class of objects without a modifier specifying an instance of this class.

Table 3-4-3: Completion Indicators for RE's verification and validation

Indicator	**Description**
Vagueness	See Table 3-4-1
Subjectivity	See Table 3-4-1
Optionality	See Table 3-4-1
Implicity	See Table 3-4-1
Weakness	See Table 3-4-1
Under-specification	The use of words that need to be instantiated (i.e., access [write, remote, authorized access]
Multiplicity	The use of multiple subjects, objects, or verbs, which suggests there are actually multiple requirements.

Table 3-4-4: Expressiveness Defect Indicators for RE's verification and validation

Numerous definitions exist in literature for requirements' quality concepts likely to be verified, as a result of environmental specialization, variety of purpose, granularity level, etc. The following are some of the characteristics that must be verified in an RS document or model.

3.4.1 Completeness

A requirement model specification is completed if full labels and references to all figures, tables, diagrams, definitions of all terms and units of measure are included in the RS document. According to [Bjorner 06] a requirements' model is completed if no holes can be pointed out, that is, everything needed to be prescribed has been prescribed. Completeness is thus relative. It is only written what "needs" be described, not what "can" be described.

According to [Zowghi, Gervasi 03], a measure for the "degree of completeness" of a set S subject to $R \cup D$, we consider the ratio between the size of a maximal subset of S that is entailed by $R \cup D$ (maximal entailed subset, or *mes*) and the size of the whole set S, i.e.:

$$\delta_{compl}(R, D, S) = \frac{|mes(R \cup D, S)|}{|S|}$$

This measure, too, has value 1 when completeness holds, and assumes progressively lower values, down to 1, for decreasing completeness.

3.4.2 Consistency of the requirements model

Zowghi and Gervasi claim that consistency requires the inexistency of two or more requirements in a specification contradicting each other [Zowghi, Gervasi 03].
As a measure for the "degree of consistency" we consider the ratio between the size of a maximal consistent subset (*mcs*) of $R \cup D$ and the size of the whole set, i.e.:

$$\delta_{cons}(R, D) = \frac{|msc(R \cup D)|}{|R \cup D|}$$

For consistent R and D, δ_{cons} (R;D) = 1, whereas the measure tends to 0 the inconsistency degree increases.

A wide variety of possible causes of inconsistency in software development have been identified in the research literature. For example, in [Nuseibeh 96], the author views inconsistency as it arises between the views of multiple stakeholders in software development. In addition, Easterbrook [Easterbrook *et al.* 95] regard an inconsistency similarly as any situation in which two parts of a specification do not obey some relationship that should be hold between them. Something similar proposes Bjorner [Bjorner 06], who claims that inconsistency of a requirements prescription is referred to some pairs (or more) of text where one text prescribes one (set of) property (properties), while another text (of the pair or more) prescribes (prescribe) an "opposite" property (set of properties), th at is: Property P and Property not P.

In [Lamsweerde *et al.* 98] it is possible to find most current techniques for inconsistency handling in the current literature consider binary-relation conflicts only.

There are potentially many ways to resolve inconsistencies [Balzer 91]. Consistency in requirements models thus implies a lack of contradiction within the presented information. Both a direct refutation of previously stated requirement and an indirect denial of this description can constitute contradicti ons within the requirements' model. Direct refutation represents statements within the model that are incompatible with each other. The truth of the first statement of a requirement directly negates the truth of the second statement. Moreover, information within the model can be refuted in an indirect manner. A given set of facts could establish a potential situation that, given the proper set of circumstances, would contradict other facts within the model. In practice whether a statement is an implicit con sequence is a matter of degree. Therefore, establishing consistency within a requirements' model is primarily a semantic task.

3.4.3 Correctness

Correctness in specification requirements is verified by checking both, consistency and completeness [Zowghi, Gervasi 03].

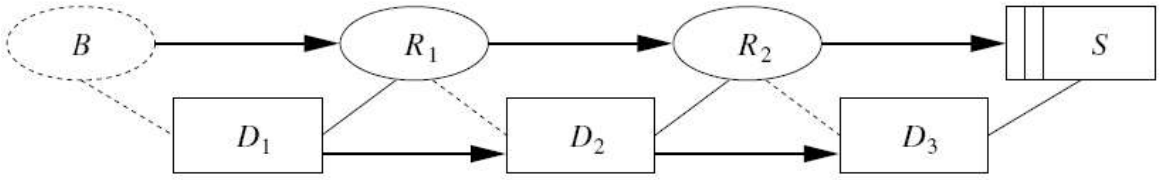


Figure 3-4-1: Relationship between Specification, Domain and Requirement in an evolutionary framework.

In figure 3-4-1, arrows represent evolution steps between successive versions of requirements and domain descriptions.

Several revisions of the requirements are considered, each one serving the role of a specification with respect to the previous one. This situation may be found in practice when we consider the common case of a product family undergoing several release cycles, but also, at a finer grain, inside a single release cycle.

Monotonic domain refinement: if we are performing an evolution step, from Ri and Di to the subsequent versions Ri+1 and Di+1, and we are only adding new information about the domain, i.e. Di+1 j= Di. Then,

$$\underbrace{(R_{i+1} \cup D_{i+1}) \mid \neq \perp}_{consistency} \wedge \underbrace{(R_{i+1} \cup D_{i+1}) \mid = R_i}_{completeness\ w.r.t.Ri} \Rightarrow (R_{i+1} \cup D_{i+1}) \mid = R_i \cup D_i$$

That is, if we can prove that:

Consistency: our new requirements are consistent with the domain (i.e., they are not asking for something that is impossible in the real world), and that *Completeness w.r.t. Ri*: the new requirements and domain description, together, do not contradict the previous requirements, then (Ri+1 U Di+1) |= Ri U Di holds.

Monotonic requirements refinement: if we are performing an evolution step, from Ri and Di to the subsequent versions Ri+1 and Di+1, and we are only adding new requirements, i.e. Ri+1 |= Ri. Then,

$$\underbrace{(R_{i+1} \cup D_{i+1}) \mid \neq \perp}_{consistency} \wedge \underbrace{(R_{i+1} \cup D_{i+1} \mid = D_i)}_{completeness\ w.r.t.Di} \Rightarrow (R_{i+1} \cup D_{i+1}) \mid = R_i \cup D_i$$

In other words, if we can prove that:

Consistency: our new requirements are consistent with the domain described so far, and that *Completeness w.r.t. Di*: the new requirements and domain description, together, do not contradict the previous domain description, then (Ri+1 U Di+1) |= Ri U Di holds.

Where Ri: requirement specification i, Di: domain specification i.

3.4.4 Satisfiability, faithful or constraint consistency

According to [Zhang *et al.* 04], in a product line modelling process, satisfiability ensures that there is no inconsistency in tailoring and binding actions at the moment of deriving a particular product line configuration form the product line model. If this property is not satisfied, constraints on features or those tailoring and binding actions should be reconsidered to eliminate inconsistencies. This definition is not implemented by means of a computational tool and the level of operationalisation and formalisation is not developed. Zhang *et al.* propose a logical formula in order to automate the satisfiability verification process, but their implementation into a computing application is not yet done.

3.4.5 Suitability of each RS model element

Within the framework of software development, ISO/IEC 2001 defines suitability as the presence and appropriateness of a set of functions for specified tasks. And in the context of product line modelling, [Zhang *et al.* 04] claim that suitability ensures that every feature without being selected has the possibility of being removed at any time. If this property is not satisfied, it means that there is one or more features that will not have the chances to be removed at some moments of its life cycle. That is to say, these features actually have been bound. Zhang *et al.* say that possible causes may be that the operators have ignored the binding of these features, or have done some improper tailoring or binding actions, or some constraints themselves are wrong. Zhang *et al.* propose a logical formula in order to automate the suitability verification process, but their implementation into a computing application is not yet done.

3.4.6 Usability of each RS model element

According to [Zhang *et al.* 04] usability ensures that every feature in the product line model has the possibility of being added at any time. They hold also that if the property is not satisfied, it means that there is one or more features that will not have the chances to be bound after the current binding time. That is to say, these features actually have been removed from the feature model. The possible causes may be that the operators have ignored the tailoring actions on these features, or have done some improper tailoring or binding actions, or some constraints themselves are wrong. They propose to eliminate these causes by putting these features to removed feature sets, or by undoing some actions at the current binding time, or by revising constraints on features. Zhang *et al.* propose a logical formula in

order to automate the usability verification process, but their implementation into a computing application is not yet done.

3.4.7 Verifiability of the RS model

In [IEEE 98] and [IEEE 04] we find that an RS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost effective process in which a person or machine can check that the software product meets the requirement. In general any ambiguous requirement is not verifiable.

According to Bjorner, this criterion relates to the implementation stage. For a requirement to have been met by an implementation means that it can be proven or tested [Bjorner 06]. Some requirements can not be so tested, at least not objectively and not quantifiable. In the case of product line, we think that all models can be verified and this premise has been the one that has motivated this research work.

3.5 Desirable characteristics to validate

There are numerous definitions in the literature for requirements quality concepts to validate, as a result of environmental specialization, variety of purpose, granularity level, etc. The following are some of the desirable characteristics that must be validated in a RS document or model.

3.5.1 Completeness

According to Boehm, to be considered complete, the requirements' document must respect three fundamental characteristics [Boehm 84]:

a) To include all significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces. In particular any external requirements imposed by a system specification should be acknowledged and treated.

b) To include definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid input values.

c) To include full labels and references to all figures, tables, and diagrams in the RS and definition of all terms and units of measure.

The first two properties imply a closure of the existing information and are typically referred to as internal completeness. The third property, however, concerns the external completeness of the document. External completeness ensures that all of the information required for problem definition is found within the specification. This definition for external completeness shows why it is impossible to define and measure absolute completeness of specification. The only truly complete specification of something would be defined if no external elements intervene in the aforementioned specification. A compromising position would be to determine whether a specification is sufficiently complete. Decision on what is sufficient completeness would have to be defined with respect to the type of system being implemented. For example, in safety-critical systems sufficient completeness may be defined with respect to system safety design constraints as well as requirements derived from hazard analysis [Leveson 00]. Clearly one of the available techniques that could assist in the determination of external completeness of the specification is domain modeling.

Jaffe *et al* have developed a set of formal criteria to identify missing, incorrect, and ambiguous requirements for process-control systems [Jaffe *et al.* 91]. This work has been continued by Leveson in the design of formal specification languages [Leveson 00]. Throughout the development of various formal specification languages, Leveson has found that propositional logic notation does not scale well to complex expressions in terms of readability and to overcome this concern, she has developed a tabular representation of disjunctive normal form called AND/OR tables. In the same way, in goal oriented methods such as I* [Mylopoulos *et al.* 99] and AGORA [Kaiya 02], goal refinement and elaborations are used in the form of AND/OR graph to validate requirements specification for completeness.

In addition, Letier and Lamsweerde suggest that goals must be made explicit in the requirements engineering process because goals drive the elaboration of requirements to support them and that they provide a criterion for measuring requirements completeness [Letier, Lamsweerde 02].

3.5.2 Consistency of the requirements

In [IEEE 98] and [IEEE 04] reports, their authors hold that if an RS does not agree with some higher-level document, such as a system requirements specification, then it is not correct.

In the following list they present some types of requirements inconsistence:

a) The specified characteristics of real-world objects may conflict.

b) There may be logical or temporal conflict between two specified actions.

c) Two or more requirements may describe the same real-world object but use different terms for that object. The use of standard terminology and definitions promotes consistency.

But they do not propose any method or technique that permits to identify and rectify these types of inconsistencies with the stakeholders or other artefact help.

3.5.3 Correctness

In [IEEE 98] and [IEEE 04] reports, we find that if an RS does not agree with some higher-level document, such as a system requirements specification, then it is not correct. So that, an RS is correct if, and only if, every requirement stated therein is one that the software shall meet.
Our literature research has showed that as there is no tool or procedure that ensures correctness. The RS should be compared with any applicable superior specification, such as a system requirements' specification, with other project documentation, and with other applicable standards, to ensure that it agrees. Alternatively the customer or user can determine if the RS correctly reflects the actual needs. Traceability makes this procedure easier and less prone to error. Completeness is a relative property and may be determined only in relation to an external reference.
According to [Zowghi, Gervasi 03], [IEEE 98] and [IEEE 04], correctness of a requirements' specification describes the correspondence of that specification with the real needs of the intended users much the same way that correctness of a piece of software refers to the agreement of the software part with its specification. In the real-world RE is an evolutionary and incremental process and hence the inconsistency analysis which is part of this process must also be performed in an evolutionary and incremental manner. This means that consistency checking is part of the construction of the requirements specifications and should be performed in parallel.
In the same way, Bjorner claims that a requirements' model is correct (validated) when it has been thoroughly validated with respect to all requirements stakeholders, and that the client has accepted the final document [Bjorner 06].

3.5.4 Importance and/or stability

According to [IEEE 98] and [IEEE 04], an RS is ranked for importance and/or stability if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement. Typically, all of the requirements that relate to a software product are not equally important. Some requirements may be essential, especially for life-critical applications, while others may be desirable.
Bjorner proposes a simple ranking of individual requirements into [Bjorner 06]:
a) Essential requirements (must be implemented);

b) Worthwhile requirement (would be very nice if implemented) ; and
c) Optional requirements (implement if not too costly), and with this ranking being stable during the requirements engineering phase.
This characteristic is implemented in the majority of RE applications in the market. In these tools, ranked is typically made by engineers and the rank possibilities are not so complicated, per contra, is a process well defined and largely implemented in order to classify requirements by characteristics mentioned above.

3.5.5 Modifiability

According to [IEEE 98] and [IEEE 04] an RS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style. Modifiability generally requires an RS to:
a) Have a coherent and easy-to-use organization with a table of contents, an index, and explicit cross referencing;
b) Not be redundant (i.e., the same requirement should not appear in more than one place in the RS);
c) Express each requirement separately, rather than intermixed with other requirements.

It is claimed that requirements change all the time. Whenever such changes actually do happen, one needs to modify the existing requirements' model [Bjorner 06]. To do so, it is of paramount importance that the requirements' prescription documents follow a hopefully existing domain description document. This makes it easier to find, we claim, where changes to the requirements' prescription model need be made, and, given that the final criterion is met, to trace repercussions of the change.

3.5.6 Satisfability

Bjorner claims that a requirements' model is satisfactory if it satisfies the following criteria [Bjorner 06]: correctness (validated by stakeholders), unanbiguity, completeness, consistency, stability, verifiability, modifiability, traceability and faithfulness. Therefore, satisfaction property can not be validated directly, that is, it is necessary to validate the previous RE quality properties in order to achieve satisfiability validation. These set of properties are discussed in this section one by one.

3.5.7 Traceability

In [IEEE 98] and [IEEE 04] reports, an RS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.

On the other hand, Bjorner holds that a requirements model is traceable if every requirement Rs, is annotated with its origin (whom, when, where), and that the reason (rationale) for the requirements is well-documented [Bjorner 06]. Furthermore, traceability means that one can simply find all those other requirements Rs1, Rs2, …, Rsn, on which the meaning of the given requirement depends, that is, Rs relies on Rs1, Rs2, …, Rsn, or whose meaning depends on the given must therefore be provided by suitable requirements documentation tools.

Traceability is widely explored and implanted in almost all modern software systems. And dependence between requirements is each day more and more an automatic activity.

3.5.8 Unambiguousness

One more time in [IEEE 98] and [IEEE 04], an RS is unambiguous if, and only if, every requirement stated therein has only one interpretation. As a minimum, this requires that each characteristic of the final product be described using a single unique term. In cases where a term used in a particular context could have multiple meanings, the term should be included in a glossary where its meaning is made more specific.

Lami claims that a requirement specification is not ambiguous, when it is [Lami 07]:

a) Not vague: When parts of the sentence are not inherently vague (e.g., do not contain words with non-unique quantifiable meanings. Like: easy, strong, good, bad, useful, significant, adequate and recent).

b) Not subjective: When sentences contain words used to express personal opinions or feelings (e.g., similar, similarly, having in mind, take into account, as [adjective] as possible, etc).

c) Not optional: When the sentence do not contains an optional part (i.e., a part that can or cannot be considered. Like: possibly, eventually, in case of, if possible, if appropriate, if needed, etc).

c) Explicit: When the subject or object of a sentence is explicitly expressed (e.g., use of demonstrative adjectives like: this, these, that, those or pronouns like: it, they. Implicit adjectives like: previous, next, following, last or prepositions like: above, below).

d) Not weak: When a sentence do not contains a weak verb (i.e., could, might, may).

According to [Bjorner 06] a requirements model is unambiguous if it does not have inconsistencies, no vaguenesses and no double meaning remain in the final requirements model. In other words: it is precise.

For us, a product line model is unambiguous if it has not inconsistencies in his variability and transversal relationships, and also if it does not admit multiple interpretations, like for example to have a mandatory and optional feature at the same time or a feature that is excluded and also required by one or two mandatory features.

3.5.9 Understandability

In the context of natural language analysis for RE, Lami holds that a requirement specification is understandable when a sentence does not have more than one main verb or more than one subject (e.g., the use of multiple subjects, objects, or verbs, which suggests there are actually multiple requirements) and when each sentence is readable [Lami 07]. He also holds that readability of sentences is measured by the Coleman-Liau Formula of readability. The reference value of this formula for an easy-to-read technical document is 10. If the value is greater than 15, the document is difficult to read.

This characteristic will be not considered to be evaluated on product line model because the context of application is rather applied on requirement textual description.

3.5.10 Verifiability with reference to stakeholders and elicitation report

For us, a RS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there are some finite cost-effective process in which a person or machine can check that the software product meets the requirement. In general any ambiguous requirement is not verifiable.

3.6 Verification and validation techniques

According to [Mader *et al.* 07], formal verification is a strong means to obtain the necessary certainty. Its success lies in the ongoing development of sophisticated algorithms, data structures and tools; the size

of (automatically) verifiable problems is increasing; new classes of verifiable problems are being found; and also hardware progress allows dealing with larger and larger systems. Many cases have demonstrated the usefulness or potential usefulness of computer aided verification.
At the same time, formal verification has a number of dangers. It is very reductionistic in the sense that it provides certainty about a model of only one fragment of reality.

According to [Kuloor, Eberlein 02], formal reviews, prototyping and requirements testing are some of the techniques used for requirements verification. Within the context of formal verification, product line engineering presents a big and actual issue for the industry. A product family has more than one product and most of the requirements are common across the family. Any defect or misunderstanding in such requirements would affect the entire family. There are requirements that are specific to each product. Hence requirements verification is very important for product lines. Requirements verification can be conducted in a similar way as they are conducted in single product development. But in the same manner, we have not clear if all the verification techniques used for a single product can also be used for product lines. And in the same way that [Lauenroth, Pohl 07], we consider that it is an open issue in RE research.

Techniques for verification:

Testing

By requirements resting we shall understand that a requirements' prescription is provided with set values for all relevant arguments (the set data), with the prescription then being evaluated for those arguments. A way of performing a requirements' test is by a systematic search for a counter-example to claim (of proof) of correctness. Testing has been and is still today a heuristic-based science. An important element in performing testing is formal text analysis. If requirements prescription parts have been formalised, then theory-based testing technologies have or can be developed and can be used for testing.

Formal proofs

By a formal proof we shall understand a given requirements' prescription, a statement (a theorem) to be proved, and the proof that the requirements' prescription satisfies the statement: this proof refers to a proof system for the language in which the requirements' prescription is expressed (axioms and inference rules), and is otherwise a sequence, composed from steps, where each step in the sequence

is like a theorem (a lemma), a statement, and where pairs of steps in the proof sequence are related by the axioms and the inference rules.

Model checking

By model checking we shall understand a method for formally verifying usual concurrent systems, frequently extremely large, have been reduced to manageable finite state systems. Requirements' prescriptions about such finite state systems are typically expressed as temporal logic formulas. Efficient symbolic algorithms are used to traverse the model defined by the system and check if the requirements prescription holds or not. Three most important techniques for model checking are CSP, SAT and BDD described in section 1.4.

Techniques for validation:

Requirements Reviews

Requirements Reviews are techniques that check system's requirements for completeness, relevance and precision [Sommerville, Kotonya 98]. They can be either formal or informal. Formal reviews include a group session to verify the requirements. Informal reviews involve a discussion between the requirements engineer and the customer. Since there are several products involved Formal Reviews have to be conducted to verify product line requirements. The review team should include domain experts, requirements engineers, customers and stakeholders. Review can be conducted as a group meeting facilitated by the requirements engineer. At first, the product family in general including all the common and variable requirements is considered for review. All the comments must be documented. Any changes required must be recorded. Next, requirements for each product in the family are reviewed. The reviewing process should also ensure that the family and its member requirements are properly mapped. Informal reviews can be conducted between the requirements' engineer and customers to validate the product specific requirements.

Prototyping

Prototyping is a technique during which products are partially implemented in order to learn more about certain problems or to demonstrate that certain features are working as intended [Sommerville, Kotonya 98]. Requirements for a product family can be verified by developing prototypes. In this case the prototype representing the common features can be reused for all the members across the family. For example, a prototype developed to illustrate the user interface facility of a system can be reused for all products.

Requirements Testing

The process of testing the product against each requirement is called Requirements Testing [Macaulay 96]. Product family requirements can also be verified by defining test cases for each requirement. While defining the test cases it is possible to unearth some of the defects in the early stage of the development. In the case of product line development, test cases must be defined for both common and variable requirements.

3.7 Conclusion

The objective of this third chapter was to survey the different definitions of verification and validation in a RE context, to compare them and listing the characteristics that must be verified and validated in a requirement specification artefact. This chapter includes not only methods and notations used in V&V process approaches, but also explains the three most popular techniques for verifying and validating a requirement specification document.

The next Chapter focuses on verification of a particular requirement specification artefact. In the subsequent chapters we will centred our effort in product line models verification process definition and implementation.

Part IV

Verification of Product Line Models

4. Verification of Product Line Models

4.1 Methods proposed in literature

Formal verification is a strong means to obtain the necessary certainty. Its success lies in the ongoing development of sophisticated algorithms, data structures and tools. The size of automatically verifiable problems is increasing; new classes of verifiable problems are being found. Hardware progress also allows us, or at least tries, to automate the verification of this kind of problems. Each day we find out more and more tools or potential tools of computer assisted verification.

At the same time, formal verification fails on its reductionistic view of problems in the sense that it provides certainty about a model of only a fragment of reality. We know that verification process for a real-world system cannot consist of formal techniques only, but also requires interaction with stakeholders and its environment in order to validate the system from a more real point of view. In this work we will focus on the verification issue. We will deal with validation feature in future reports.

In this section we will work on verification process found in literature. We argue that the importance of positive results from formal verification depends on the quality of the verification process and techniques used. Next we present twelve most relevant methods found in literature.

1. A method for formal verification of a feature model is proposed by [Wang *et al.* 05]. They present an approach to modeling and verifying feature diagrams using Semantic Web ontologies. In the proposed method, the feature configuration is constructed as a separate ontology and the reasoning engine is invoked to check its consistency. The configuration is valid if the ontology is checked to be consistent with respect to the feature diagram ontology. They use Protégé-OWL and RACER to detect the inconsistence in a particular feature configuration, looking for errors in relations between features and inconsistent constraints between features.
   In the following diagram, Wang *et al* show how RACER detect constraint inconsistency between a set of value constraints of a particular configuration “E” and the product line model constraints.

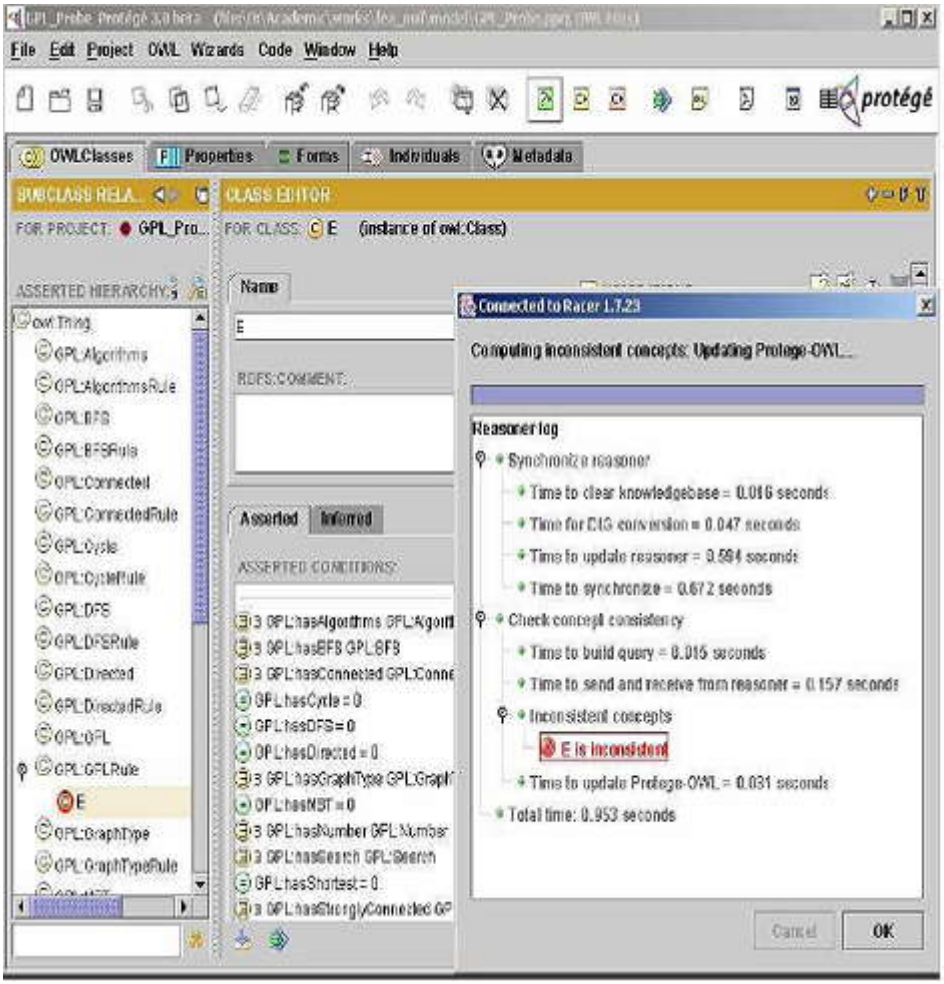


Figure 4-1-1: RACER detect an inconsistency, taken from [Wang *et al.* 05]

2. A propositional logic-based method for verification of feature models at different binding times (construct-time, reuse-time, compile-time, install-time, load-time) is proposed by [Zhang *et al.* 04]. In this method, the constraints in a feature model are formalised in a set of logical sentences. Each binding time, after an undecided feature is bound or removed, the truth value of this feature will become the logical constant True or False respectively. With this conversion the constraint satisfiability problem is transformed in satisfaction problems in the mathematical logic. So, verification problems such as the detection of inconsistent constraints or the detection of the conflicting or unnecessary binding resolution can be automatically revealed. Zhang *et al* propose three properties SUS (i.e. Satisfability, Usability and Suitability) to verify feature models.

   We have modified the original formula proposed by Zhang *et al* in order to permit feature model verification independently of the binding time concept. Thus, it is proposed to check:

   Liveliness (Usability): the usability ensures that every feature in Undefined Feature Set has the possibility of being bound or added in the future model.

$$\forall \text{ Feature } f, (\text{VariantDependency, TransverseDependency}) C \subseteq PLM,$$
$$\exists\ I_k \in \{PLC_1, ..., PLC_i\}\ (1 \le k \le i), I_k \mid = (\cap_{i=1..n} C_i) \cap f$$

Utility (Suitability): the suitability ensures that every feature in Feature Set has the possibility of being removed from the feature model.

$$\forall \text{ Feature } f, (\text{VariantDependency, TransverseDependency}) C \subseteq PLM,$$
$$\exists\ I_k \in \{PLC_1, ..., PLC_i\}\ (1 \le k \le i), I_k \mid = (\cap_{i=1..n} C_i) \cap (\neg f)$$

Satisfiability: the satisfiability ensures that there is no inconsistency in constraints. If this property is not satisfied, constraints on features should be reconsidered to eliminate inconsistencies.

$$\exists \text{ Feature } f, (\text{VariantDependency, TransverseDependency}) C \subseteq PLM, f \mid = \cap_{i=1..n} C_i$$

Zhang *et al* propose five main steps in the process of feature model verification. The firs step is feature model construction. In this step, features, refinement relations between features and constraints on features associated with a product-line are systematically identified. The second step is to formalise constraints on features into logic sentences. The third step is to compute atomic-set contained in a feature model, and use these atomic-sets to simplify constraints by replacing features involved in constraints with their corresponding atomic-sets. After the third step, operators can apply the SUS criteria to verify constraints on features, and further take binding resolutions at each binding-time (the fifth step) and repeatedly apply the SUS criteria to verify these resolutions. The fourth step can be automated by using model checkers, such as SMV. The approach can be summarized with the next process diagram:

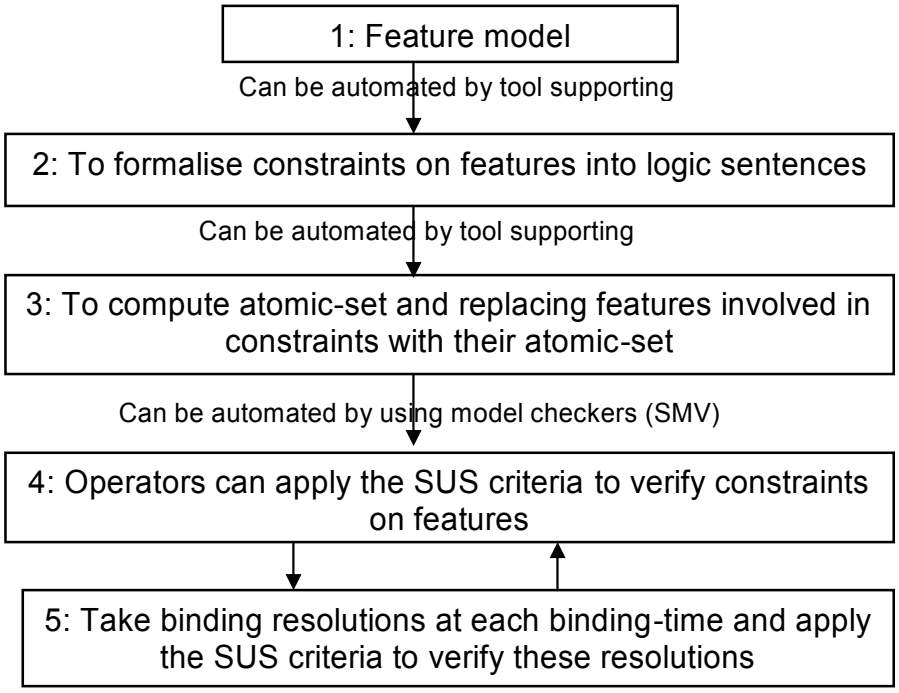


Figure 4-1-2: process of feature model verification proposed by Zhang *et al*

3. Another verification procedure pretends to check feature-based model templates against Object-Constraint Language (OCL) well-formedness rules. [Czarnecki, Pietroszek 06] present an automated verification procedure for ensuring that no ill-structured template instance will be generated from a correct feature-based configuration. This method use a feature-based model template composed of a feature model and an annotated model expressed in some general modeling language such as UML or a domain-specific modelling language. The approach allows expressing the desired well-formedness constraints in OCL with respect to the metamodel of the target modelling language. Next, this set of constraints is transformed into a model in the chosen modelling language. So, a feature-based template is correct if, and only if, every correct configuration results in a correct template instance. Though, in this approach it is easy to forget a necessary constraint in the feature model or an annotation in the annotated model. Their approach can be applied to a domain requirement specification. However, Czarnecki and Pietroszek only deal with static properties of the UML.

   The follow diagram, taken from [Czarnecki, Pietroszek 06], shows the context used in the verification procedure.

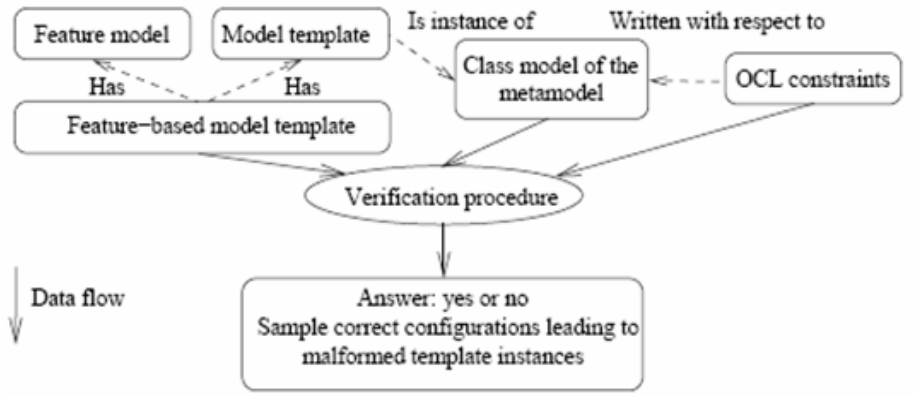


Figure 4-1-3: Context of the Czarnecki and Pietroszek's verification procedure

It is clear that in the following figure, the class diagram, that has been generated from the PL configuration, is not well formed because of the dangling association "MultipleClassification | !Categories"

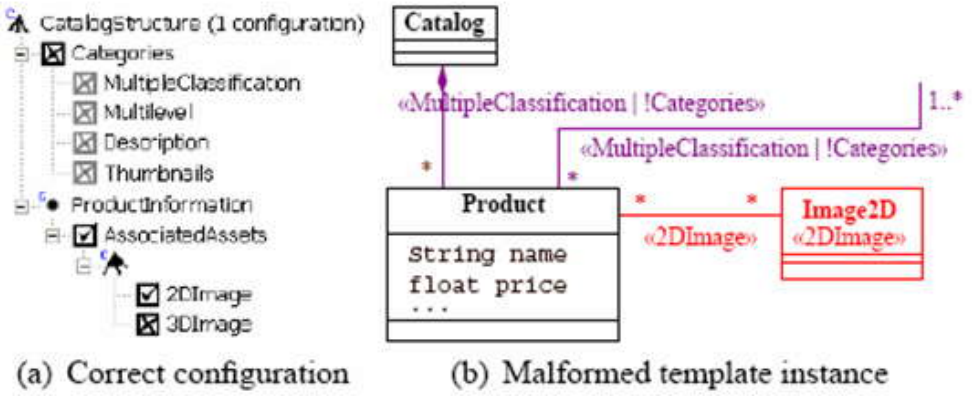


Figure 4-1-4: Example of a UML class model template, taken from [Czarnecki, Pietroszek 06]

So, the key idea of a feature-based model template is that, given a particular feature configuration, an instance of the template can be automatically created by removing the model elements which presence conditions are evaluated FALSE.

4. An initial research path to address volatility analysis in software PL engineering issues, based on aspect-oriented and model-driven techniques is presented in [Alférez *et al*. 07]. This approach motivates the use of feature models in conjunction with use case and activity diagrams, so that every requirement is ideally related to only a feature. But in this verification approach, oriented to satisfaction of requirements, it is possible that a use case be related with several features.
   Figure 4-1-5 describe the process by means of which the generation of the use case model related to a SPL configuration and the generation of activity diagrams related to a SPL configuration is

done. The mapping between features, use cases, activity steps, is defined by creating links between those elements in their respective meta-models.

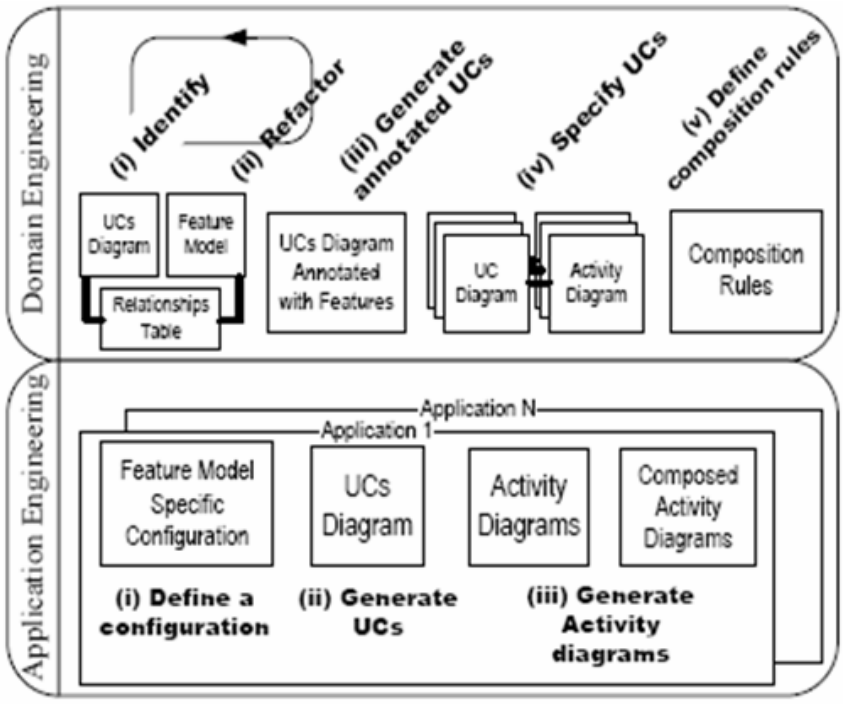


Figure 4-1-5: an overview of the process, taken from [Alférez *et al.* 07]

5. Padmanabhan and Lutz propose a tool-supported verification of product line requirements [Padmanabhan, Lutz 05]. They use logic to define constraint rules for the consistency of requirements and support consistency checks of the PRS. Their approach is implemented in a requirement engineering tool called DECIMAL (DECIsion Modeling AppLication). DECIMAL is an interactive, GUI-driven requirements verification tool that automatically checks for completeness and consistency between a new product and the product line to which it belongs. DECIMAL also performs range and type correctness checking to verify that the values of variabilities selected for the new member fall in the range and are of the same data type as specified for the product line. The tool does not provide run-time checks and neither check to see if the constraints themselves are consistent. They do not support consistency checks of the DRS.

   In Figure 4-1-6 a SQL query looking for inconsistencies in the constraints and a SQL query looking for product-line member that did not satisfy a particular commonality are shown. The utilized method consist in automatically checks that dependency relationships among variabilities are maintained in the new system.

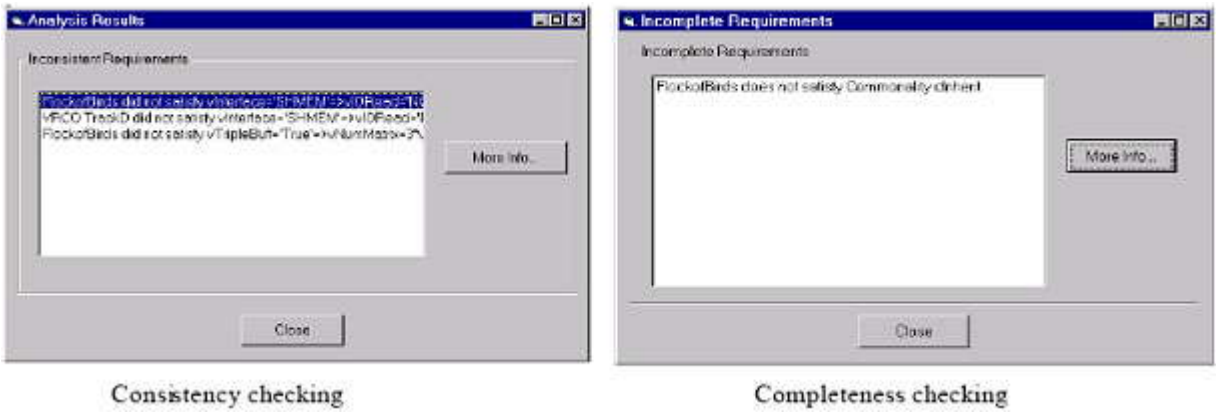
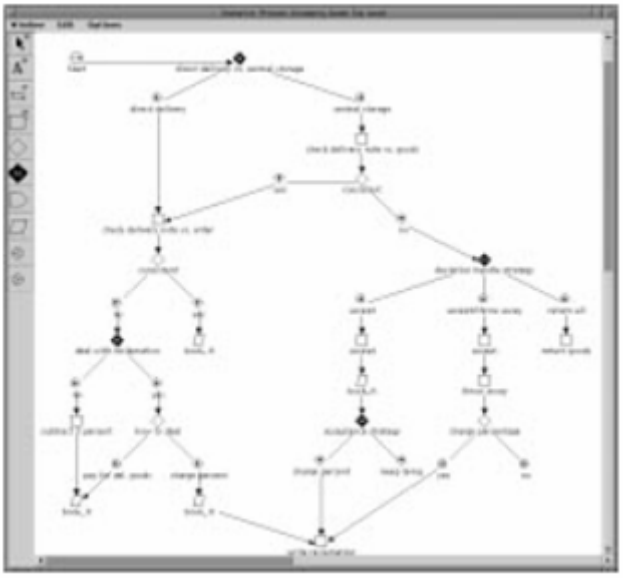


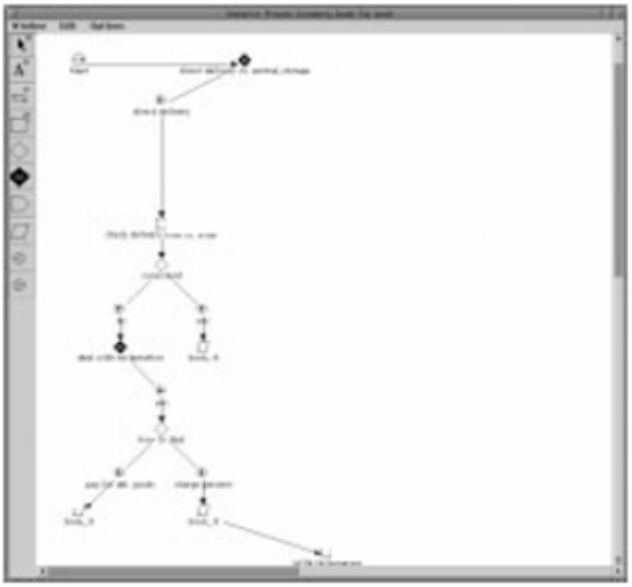
Figure 4-1-6: SQL queries for consistency and completeness verification in DECIMAL

6. Product Line Software Engineering (PuLSE) is a customizable product line practice proposed in [Bayer *et al.* 99]**.** The PuLSE process has three basic elements: Deployment Phases, Technical Components and Support Components. Deployment phases are logical steps in the PuLSE practice that describe the activities needed to define and develop a family of products. PuLSE technical components include the technical expertise required to carry out various PuLSE activities. PuLSE support components provide the guidelines required to solve any non-technical issue such as organization issue, project entry points and process evolution. On the other hand PuLSE provides very little information about requirements specification, verification and traceability. Moreover PuLSE permits to carry out a consistency verification procedure over one member of the product line with respect to PL model. Besides, we have found evolutions of PuLSE, such as KobrA that is an object-oriented customization of PuLSE. However, KobrA focuses more on the design and implementation of a domain framework.

   The follow figures show the generic storyboard of a PL and one particular storyboard's configuration.



Example - Generic Storyboard



Example - Instantiated Storyboard

Figures 4-1-7 and 4-1-8: Examples taken from [Bayer *et al.* 99].

7. Pure:variants is a tool developed by 'Software Acumen' enterprise as a Eclipse plug-in [PureVariants], [Spinczyk, Beuche 04]. In the context of pure::variants, model verification is the process of checking the validity of feature family, and variant description models. Two kinds of model validation are supported, i.e. validating the XML structure of models using a corresponding XML Schema in order to check if the XML structure of a pure::variants model is correct. And performing a configurable set of checks using the model check framework in order to allow the validation of models using a configurable and extensible set of rules (called "model checks"). There are no restrictions on the complexity of model checks.

As feature selections are made, pure::variants checks their validity and, if necessary, automatically resolves dependency conflicts or highlights conflicts if they cannot be resolved automatically. Once a valid selection has been made, an evaluation of the Family models, containing component definitions, is performed.

This evaluation process produces an abstract (XML) description of the variant in terms of software components (components, modules, files etc.). This description is used to control a transformation process that in-turn generates the finished product variant (source code and other artefacts).

The next figure show a computer's product line model, in which several particular configurations can be derived and verified like we have described above.

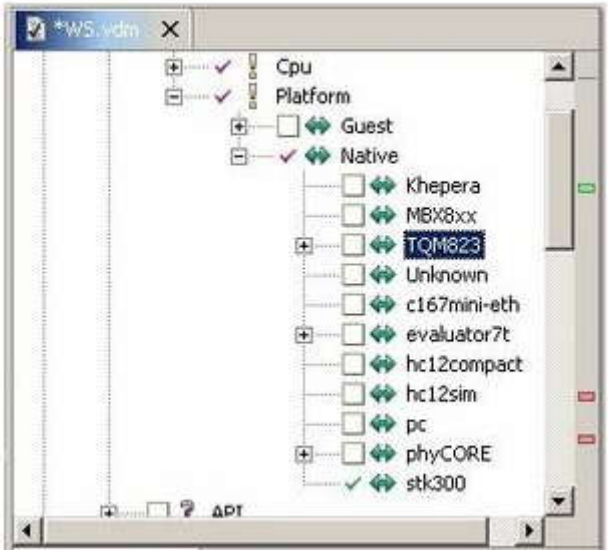


Figure 4-1-9: Example of computer's product line model, taken from [PureVariants]

8. FeaturePlugin is an Eclipse plug-in proposed in [Antkiewicz, Czarnecki 04]. Verification process in FeaturePlugin is yet under construction and refinement because, by the moment, the tool only supports static verifications of the models. FeaturePlugin also support additional constraints, i.e., those that cannot be expressed as feature or group cardinalities. Common examples are implies and excludes constraints. In general, additional constraints in cardinality based feature models require tree-oriented navigation and query facilities, iteration mechanisms or quantifiers, and ways of counting feature clones in the scope of a given feature within a configuration. Furthermore, logic, arithmetic, set, and string operators on feature attributes and feature sets are desirable. Such constraints can be adequately expressed using XPath 2.0.

   FeaturePlugin can check the additional constraints for a given configuration. For example, the configuration in Figure 4-1-10 satisfies all the constraints from Figure 4-1-11. Figure 4-1-11 shows three examples. The first constraint is an example of a local constraint requiring that the attribute of

InDays is positive. The second constraint involves several features and states that selecting FraudDetection implies that CreditCard and/or DebitCard are selected. The third constraint existentially quantifies over feature clones and states that at least one custom shipping method should have a rate of more than 0.

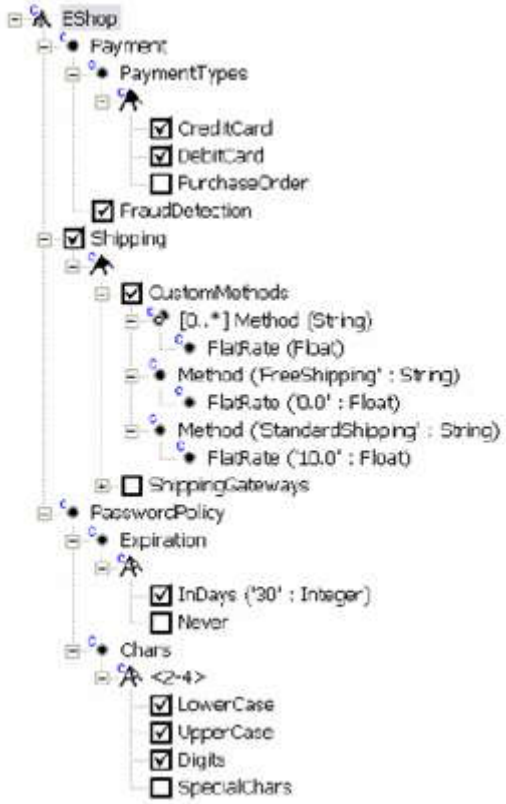


Figure 4-1-10: configuration of EShop, taken from [Antkiewicz, Czarnecki 04].

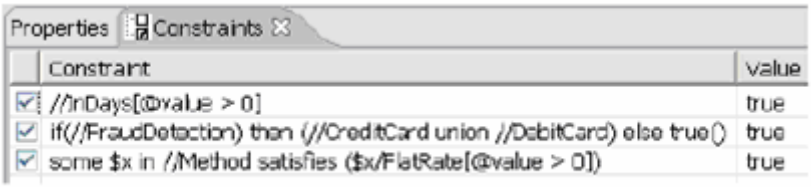


Figure 4-1-11: constraints evaluated from on the configuration from figure #-1, taken from [Antkiewicz, Czarnecki 04].

9. FAMA is a framework for the automated analysis of feature models proposed by [Benavides *et al.* 07]. The automated analyses of FMs is usually performed in two steps:
   i) The FM is translated into a certain logic representation

ii) Off-the-shelf solvers are used to extract information from the result of the previous translation such as the number of possible products of the feature model, all the products following a criteria, finding the minimum cost configuration, etc [Benavides *et al.* 06].

The current implementation of FAMA integrates three of the most promising logic representations proposed in the area of the automated analysis of feature models: CSP, SAT and BDD, but more solvers can be added if needed. The implementation is based on an Eclipse plug – in and uses XML to represent FMs so it can interoperate with other tools that support it.

The operations fully supported by FAMA and showed in Figure 4-1-12 are:

• Finding out if an FM is valid, i.e. there is a product satisfying all the constraints. This verification is showed in Figure 4-1-13 for a particular configuration.

• Finding the total number of possible products of an FM (number of products).

• List all the possible products of a feature model (list of products).

• Calculate the commonality of a feature, i.e. the number of products where a feature appears in.

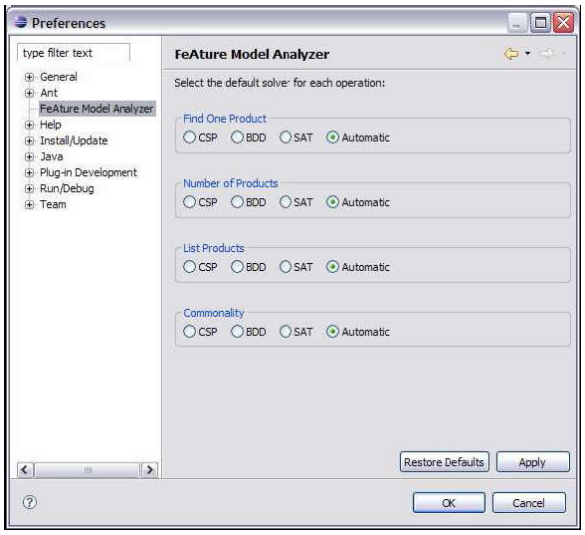


Figure 4-1-12: operations and selection of solvers in FAMA (taken from [Benavides *et al.* 07])

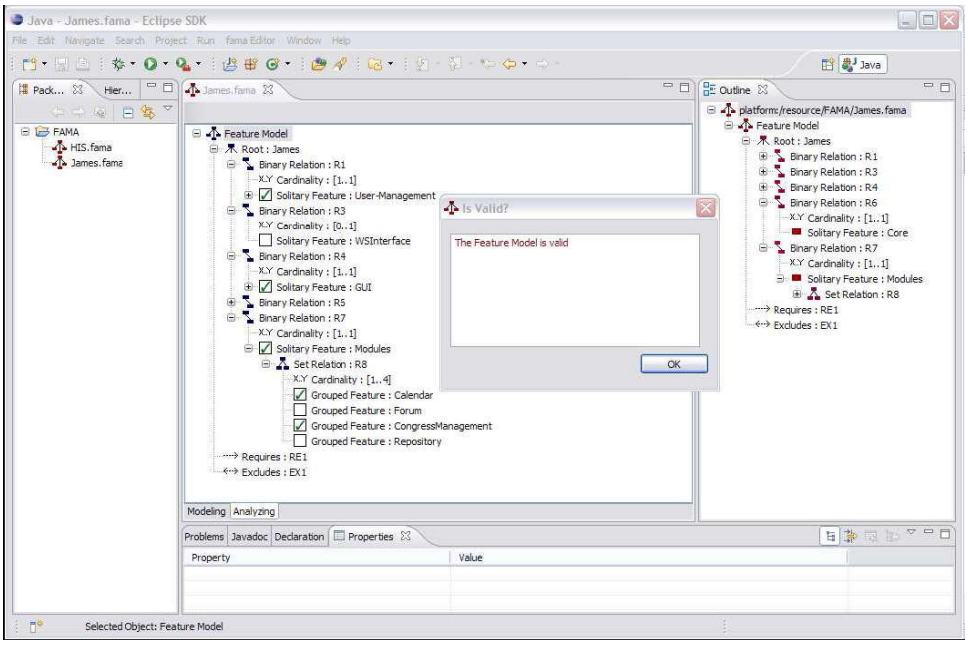


Figure 4-1-13: verification of constrains satisfiability for a particular configuration (taken from [Benavides *et al.* 07]).

10. Another method for product line models verification is proposed in [Batory, Thaker 06]. Batory and Thaker focus on the safe composition of products by ensuring that there is no undefined element (e.g. classes, method) referred to in a composed program implementation. Again, they deal with static properties of the product line model and product line configurations and do not support a consistency check of behavioral properties. They propose some logical expressions dealing with refinement constraint, parent constraints and reference constraints. Like for example:

    If features X, Y, and Z, are refined by feature F:

    $F \Rightarrow X \vee Y \vee Z$

    Let PLf be the propositional formula of product line PL:

    $(PLf \wedge F \wedge \neg X \wedge \neg Y \wedge \neg Z) = false$

11. Gomaa and Shin suggest a multiple-view approach for modelling variability in software product lines and verify properties like consistency of relations between objects and traceability assurance [Gomaa, Shin 04]. They extend UML (Unified Modeling Language) notations with variation points and define a central feature notation. All variabilities are then linked together in a multiple-view meta-model. In addition to assuring traceability, the meta-model can be used in order to check whether all relations between objects are correct, like for instance checking that each class corresponds to a feature. This can be achieved thanks to definition of rules at meta-level and checking that multiple-view model must follow the rules defined in the multiple-view metamodel.

The next figure (Figure 4-1-14) shows the method proposed by Gomaa and Shin:

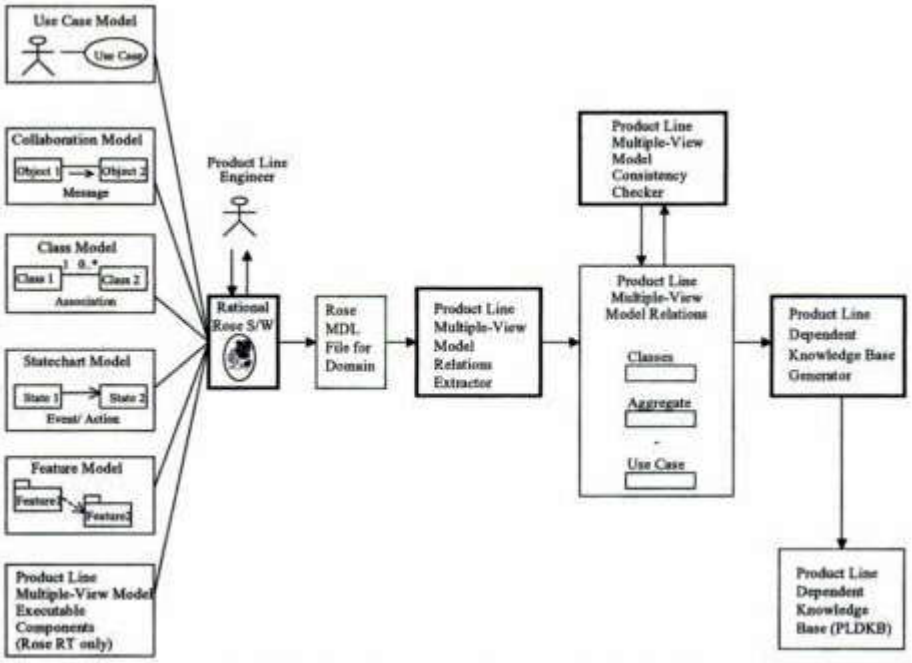


Figure 4-1-14: multiple-view approach for modelling variability and consistency checking (taken from [Gomaa, Shin 04]).

12. Classen proposes a problem-oriented modelling and verification method of software product lines [Classen 07]. He suggests to use the problem frames approach as a complement to feature diagrams; (i) because it is based on the well established requirements engineering framework mentioned before, thus enforcing the distinction of the three main descriptions, and (ii) because it emphasizes and provides appropriate means for modelling the physical context. According to Classen, a PLM is consistent if the set of specification S and the set of domain description (W) satisfy the set of requirements R (expressed in a PLM). That is (S, W) |= R

    The idea behind the feature interaction detection approach is essentially to verify the first proof obligation of the requirements engineering framework by Zave and Jackson [Zave, Jackson 97], S,W |= R, for all valid configurations of a PLM. A given configuration can be checked for interactions by verifying its proof obligation, which constituents can be found in the feature descriptions. If the verification of the proof obligation is to be automated, all descriptions have to be expressed in a formalism that allows for automated reasoning. Classen's approach uses the event calculus (EC) [Shanahan 99] for that purpose, because it is intuitive and well suited for commonsense descriptions such as those found in the domain properties for instance. The whole process can then

be mostly automated: (i) valid configurations of a product line are derived from the feature diagram and (ii) each configuration is verified using the Decreasoner EC implementation.

The first proof obligation of this reference model serves as correctness proof for a feature or a system. The three constituent descriptions of the proof obligation (specifications S, domain descriptions W and the requirement R) are coherently modelled (see Figure 4-1-15) using Jackson's Problem Frames approach [Jackson 01], and formally expressed using the EC [Mueller 06], [Shanahan 99]. This in turn allows to automate the verification of the first proof obligation through automated model checking. Finally, the variability of the PL is modelled using FDs [Schobbens *et al.* 06]. Feature interaction detection is then done by verifying the proof obligation for each possible product of the product line, as defined in the PLM.

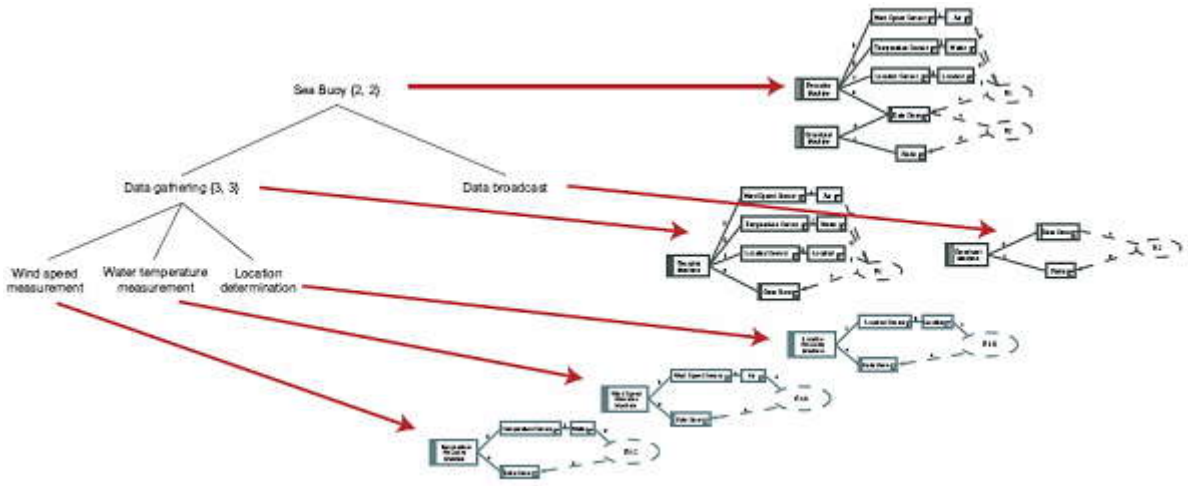


Figure 4-1-15: mapping a FD to several problem diagrams (taken from [Classen 07]).

4.2 PLM Meta-Model

Variability can be defined either as an integral part of development artefacts or in a separate variability model. Many research contributions have suggested the integration of variability in traditional software development diagrams or models such as use case models, feature models, message sequence diagrams, and class diagrams. Kang *et al* and Fey *et al* use feature models to represent variability [Kang *et al.* 02], [Fey *et al.* 02]. Bühne *et al*, Halmans and Pohl, von der Maßen and Lichter introduce variability in use case models [Bühne *et al.* 03], [Halmans and Pohl 03], [V.d. Maßen, Lichter 02]. Bosch et al. and Svahnberg *et al* deal with variability in implementation structures [Bosch *et al.* 02], [Svahnberg *et al.* 01] and nearly Pohl *et al* propose an orthogonal variability model as an artefact that "relates the

variability defined to other software development models such as feature models, use case models, design models, component models, and test models" [Pohl *et al.* 05].

We will work with FORE approach [Streitferdt 03]. We propose the metamodel depicted in Figure 4-2-1 for our tool- supported verification method.

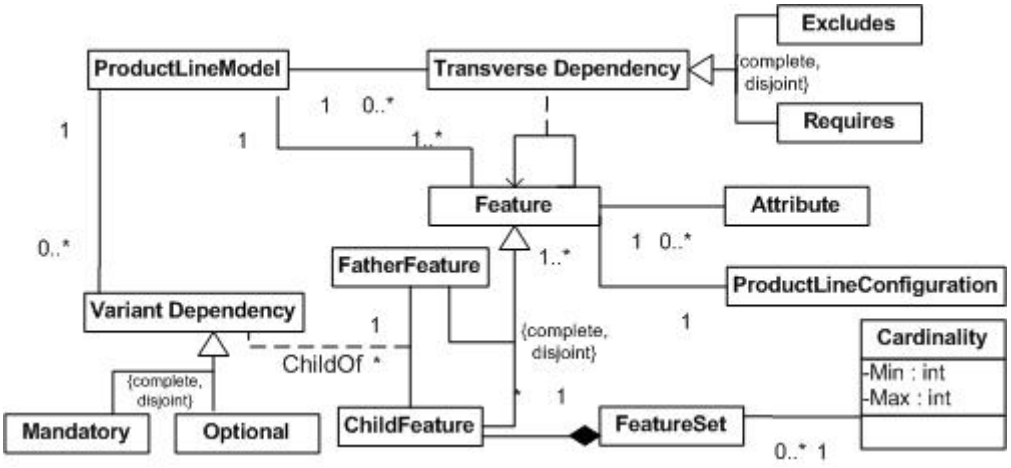


Figure 4-2-1: Proposed metamodel for feature product line models.

The two central elements of the product line metamodel are Product Line Model (PLM) and Product Line Configuration (PLC). A PLM is composed of three elements, the "Feature", "Variant Dependency" and "Transverse Dependency". And a PLC is composed of "Features".

The Feature class is specialised in two classes "FatherFeature" and "ChildFeature". This specialisation is complete and disjoint. Consequently, every Feature is either of the class "FatherFeature" or "ChildFeature". One or several "ChildFeature" are associated by the relation "ChildOf" with a "FatherFeature" class.

Each model element depicted in Figure 4-2-1 has at least one attribute, all grouped in the class "Attribute", i.e. identifier and textual annotation that allow us, for instance, to record the particularities of each element. For the sake of simplicity, the attributes are not shown in the metamodel.

Variability Dependency

A "VariantDependency" is the association class between "FatherFeature" and "ChildFeature" classes. The "VariantDependency" class is specialised in "Mandatory" and "Optional" as a complete and disjoint heritage relationship. The multiplicities of the association enforce that a "FatherFeature" must be associated with at least one "ChildFeature" and each "ChildFeature" must be associated with one "FatherFeature".

The optional variability dependency states that a variant child related to the variation father can be a part of a particular product line configuration but does not need to be a part of it. The set of child features related to parent features by means of optional dependencies is called the variability of a feature product line model. The mandatory variability dependency implies that child feature must be part of a particular configuration if the father feature is chosen. This does not imply that mandatory features have to be included in all configurations of the software product line. A mandatory feature is only part of an application if the related feature father is part of it. The set of child features related to parent features through mandatory dependencies is caller the commonality of the feature product line model.

Transversal Dependency

A "TransverseDependency" is the association class between two features that are not related by father-child relationship. We specialise the "TransverseDependency" relationship class into an "Excludes" and a "Requires" relationship. The specialisation is defined as complete and disjoint. The multiplicities of the association enforce that each feature can be associated through a transverse dependency with one or several variants.

The "Excludes" transverse dependency states that a feature *A* can constraint other feature *B* in the sense that if *A* exists in a particular product line configuration, then feature *B* can not be chosen in the same configuration. The "Requires" transverse dependency stated that a feature *A* can constraint other feature *B* in the sense that if feature *A* is chosen, then feature *B* must be chosen in the same product line configuration.

Alternative Choice

A variability model must offer the facility to define the minimum and the maximum number of features to be selected from a given group of variants. Consequently, we define a modelling element that allows us to group optional features and to define multiplicities for each group. For each "FeatureSet" we have a "Cardinality" with two attributes, "Min" and "Max". They are needed to specify the range for the permissible numbers of variants to be selected from the group. Additionally, the "Cardinality" class has the constraint that the "ChildFeature" that are part of a group must refer to the same "FatherFeature".

4.3. {Characteristics to verify in PL models} + {techniques}* + {lessons}*

In the literature, we have found some characteristics that we can considerate in a process of product line model verification. These characteristics are defined and discussed in this section; also we go to

present the techniques used to their implementation and some lessons learned from this literature analysis to our product line model verification approach.

4.3.1. Consistency

In the literature we found several definitions of consistency in PLMs. According to [Lauenroth, Pohl 07] the PLC of a product line is consistent, if a viable implementation exists. A necessary criterion for consistency is the absence of contradictions in the specification.
Requirements in the PLM can contradict each other. This implies that a PLM must, however, not be consistent. For example, Savolainen and Kuusela argue that inconsistencies (i.e. contradictions) between product line requirements can be intentionally introduced: "… within the product family there are also intentional inconsistencies" [Savolainen, Kuusela 01]. For example, it is possible that one product of a product line *P* has to fulfil the requirement *A*, whereas another product of the product line *P* has to fulfil the requirement ¬*A*. Consequently, a PLM may contain contradicting requirements.
In contrast, a PLC must be consistent. However, from contradicting requirements in the PLM we cannot necessarily deduce that any derived PLC is inconsistent. A set of contradicting requirements in the PLM will imply an inconsistency in a PLC only if the set of contradicting requirements of the PLM can be part of the same PLC. Since the variability model determines if a set of contradicting requirements may become part of the same PLC, the definition of consistency of a PLM must take the product line variability into account.
Laurent and Pohl also claim that the PLM of a product line is consistent, if all PLCs of all possible products (determined by the variability model of the PLM) are consistent. The computational cost for automate these definition is too high, for this reason Laurent and Phol's last definition of consistency is lacking from a pragmatic point of view, and there are not known tools that implement it.

In addition, in [Batory,Thaker 06] authors deal with static properties of the PLC and do not support a consistency check of behavioral properties. Some years back, [Bayer *et al.* 99] had said than a PLC is consistent if satisfied the set of PLM commonalities. And as complement to the previous definition, [Padmanabhan, Lutz 05], a PLC is valid if there are not inconsistencies by reference to PLM's constraints.

On the other hand, for [Wang *et al.* 05], a PLC is valid if its ontology is checked to be consistent with the PLM's ontology. According to [Classen 07], a PLM is consistent if the set of specification and the set of domain description satisfy the set of requirements (or the PLM).

We can say that a PLM is consistent if and only if it satisfied:

1. The right dependency relationships among the features (constraints, cardinality-range, types, etc);
2. A particular Product Line Configuration construction respect all the commonalities;
3. It is possible to derive at least one particular configuration from the product line model.

Techniques proposed for consistency verification:

In [Wang *et al.* 05], the authors present an approach to modeling and verifying feature diagrams using Semantic Web Ontologies. The technique proposed shows how to check PLC's ontology with respect to the PLM's ontology in order to verify consistency. This technique needs to transform the PL feature model and each PLC feature model into PLM's and PLC's ontology respectively and to compare each one with the PLM's ontology. This comparison is made by means of a reasoning engine that is invoked to check its consistency. The configuration is valid if the ontology is checked to be consistent with respect to the feature diagram ontology. Wang *et al's* approach use Protégé-OWL and RACER to successfully detect the inconsistence in a PLC.

Another technique is proposed by [Bayer *et al.* 99]. This technique pretends to check that each commonality in the PLM must be present in each PLC, if this correspondence relation is not satisfied for PLC *k*, then PLC *k* is not consistent. We can represent it by the following logical constraint:

$$\forall Commonality \quad c \in PLM, \exists \quad feature \quad f \in PLC.\ f = c$$

Padmanabhan and Lutz propose to make SQL queries looking for inconsistencies in the constraints and SQL queries looking for PLCs that did not satisfy a particular commonality [Padmanabhan, Lutz 05]. This technique addresses the issue of requirements verification for product lines through a requirements engineering tool called DECIMAL (DECIsion Modeling AppLication). DECIMAL is an interactive, GUI-driven requirements verification tool that automatically checks for completeness and consistency between a new product and the product line to which it belongs and automates completeness, consistency, range, and type checks to verify that the values of variabilities selected for the new member fall in the range and are of the same data type as specified for the product line. On the other hand, DECIMAL still does not check to see if the constraints themselves are consistent.

Lauenroth and Pohl propose a consistency checking technique for dynamic properties of PLMs based on model checking techniques [Lauenroth, Pohl 07]. In this technique, the goal is to check the variability model, the PLM, and their interrelation to determine if the definitions in these artefacts allow the derivation of an inconsistent PLC. They use model checking as a proven technique for performing consistency checks of behavioral specifications in single system engineering, through adaptation and application of existing model checking algorithms.

The central idea of this approach is as follows:

For a given invariant i, an algorithm searches for a valid path from the start state to a state that violates i.

A path is considered as valid if a PLC can be derived that contains this path and the considered invariant. If the derivation of a PLC with such a path is possible, the PLM is inconsistent. The identification of such paths requires the following steps:

1) Capture the behaviour of the product line in a single automaton.

2) Search for states violating an invariant in the variable global system automaton.

3) Search for valid paths from the start state to the states causing the violations.

This technique can be resumed by means of the follow logical expression and algorithm for PLM inconsistency verification:

$$\forall i \in \text{invaSet}, (\exists \quad path \quad P \in PLC \mid \neq i \wedge \exists \ (P,i) \subset PLM) \mid = \perp$$

Algorithm for PLM inconsistency verification:

For **invariant i**:

Search in **PLC** a valid **path P** from initial state to a state that violates i.

If a **PLC** exists with **P**, **PLM** is inconsistent

In [Batory, Thaker 06], the authors propose a technique for support the automatic creation of a product line software implementation based on feature models. They focus on the safe composition of products by ensuring that there is no undefined element (e.g. classes, method) referred to in a composed program implementation. Again, Batory and Thaker deal with static properties of the product line and do not support a consistency check of behavioral properties. For example, they proposed verification logical formulas like Refinement Constraint: If features X, Y, and Z, are refined by feature F:

$F \Rightarrow X \vee Y \vee Z$

Lessons learned from consistency verification:

Numerous research contributions for checking consistency in single system engineering have been proposed [Heitmeyer *et al.* 96], [Hunter, Nuseibeh 98], [Huzar *et al.* 05]. Whereas those approaches support consistency checking during the engineering of a single software system, their use for checking consistency in product line requirements specification is not suitable since those approaches do not provide adequate support to handle the variability in product line requirements specifications.

Existing approaches for consistency checking of requirements specifications in product line engineering focus on the requirements specification derived in application engineering for a particular product [Fantechi *et al.* 04], [Padmanabhan, Lutz 05]. Consistency checking of requirements specification is thus only performed in application engineering and not in domain engineering. In application engineering, a correction of the inconsistency quite likely influences not only the product currently derived, but also the domain artifacts as well as products previously derived from the product line.

We consider the approaches presented in [Wang *et al.* 05], [Bayer *et al.* 99], [Padmanabhan, Lutz 05], [Batory, Thaker 06] and [Lauenroth, Pohl 07] are important for our work. All these approaches are important for consistency analysis in product line implementation.

4.3.2. Correctness or Satisfiability of Constraints

In the literature we found several definitions of correctness in PL models. According to [Bjorner 06] a model of requirements is correct when it has been thoroughly validated with respect to all requirements of stakeholders, and that the client has accepted the final document. For Czarnecki and Pietroszek, a model is well-formed or correct if it conforms to the metamodel, i.e., it satisfies the multiplicities and the Object-Constraint Language (OCL) constraints of the metamodel [Czarnecki, Pietroszek 06]. Alternatively, in [Wang *et al.* 05] authors say that a particular PL configuration is valid if its ontology is checked to be consistent with the PL model ontology.

On the other hand, Zhang, Zhao and Mei hold that constraint satisfiability property ensures that there is no inconsistency in tailoring of features [Zhang *et al.* 04]. The verification of feature models is converted into satisfaction problems in the logic. Therefore, verification problems such as the detection of inconsistent constraints or the detection of the conflicting or unnecessary binding resolution can be automatically revealed.

In the literature, we have also found that Batory and Thaker work in the satisfiability of constraints issue from the point of view of the safe composition of products by ensuring that there is no undefined element (e.g. classes, method) referred to in a composed program implementation [Batory, Thaker 06].

Techniques proposed for correctness or satisfiability of constraints verification:

Czarnecki and Pietroszek propose an automatic verification procedure which can establish that no ill-formed template instances will be produced given a correct configuration of the template's feature model [Czarnecki, Pietroszek 06]. According with their approach, it is necessary to express the desired well-formedness constraints in the OCL with respect to the metamodel of the target modelling language of the template instances. This key capability is achieved through a new semantics of OCL for templates. The semantics maps OCL constraints to propositional formulas, which are then fed into a SAT solver.

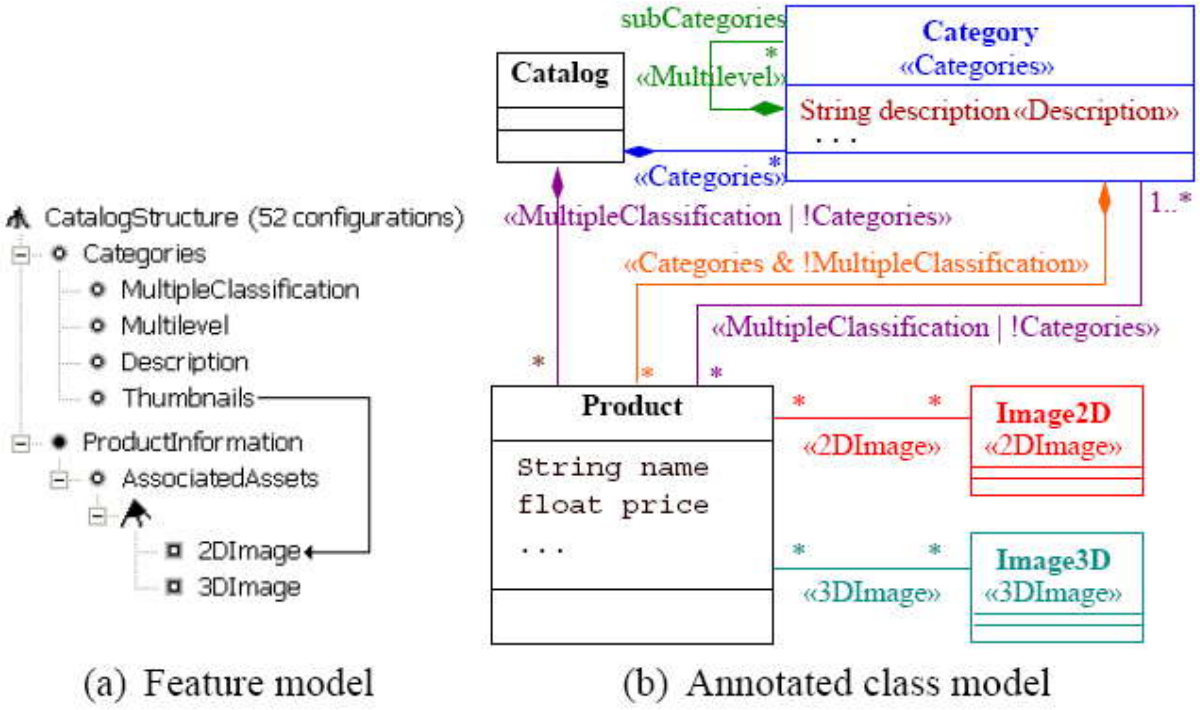


Figure 4-3-1: Example of a UML class model template, taken from [Czarnecki, Pietroszek 06]

The feature model in Figure 4-3-1 (a) denotes 52 correct configurations. A sample configuration is shown in the same figure using a so-called *check-box rendering* of a configuration tool [Czarnecki, Kim 05]. In this rendering, optional features are shown as check boxes. The boxes of selected features are checked. The boxes of eliminated features are crossed. Figure 4-3-1 (b) shows an example of a UML class model template, which is a UML class model annotated with presence conditions.

The key idea of a feature-based model template is that, given a particular feature configuration, an instance of the template can be automatically created by removing the model elements which presence conditions are evaluated FALSE.

Feature	Abbr.	Feature	Abbr.
CatalogStructure	cs	Thumbnails	tn
Categories	ct	ProductInformation	pi
MultipleClassification	mc	AssociatedAssets	aa
Multilevel	ml	2DImage	i2
Description	ds	3DImage	i3

Table 4-3-1: Abbreviations of feature names from Figure 4-3-1 (a), taken from [Czarnecki, Pietroszek 06]

For the feature model in Figure 4-3-1 (a) and assuming the abbreviations in Table 4-3-1, the formula to verify q*FM* is as follows:

q*FM* =

root: cs^

parent-child: (ct ∘ cs) ^ (mc ∘ ct) ^ (ml ∘ ct) ^ (ds ∘ ct) ^ (tn ∘ ct) ^ (pi • cs) ^ (aa ∘ pi) ^ (i2 ∘ aa) ^ (i3 ∘ aa) ^

group: (aa $\Rightarrow$ $choice_{1,2}$(i2, i3)) ^

transverse: (tn $\rightarrow$ i2)

where

$choice_{1,2}$(i2, i3) = i2 ^ ¬i3 v ¬i2 ^ i3 v i2 ^ i3 = i2 v i3

Let us consider the following example, taken from [Czarnecki, Pietroszek 06], in which the template in Figure 4-3-1 has an annotation error leading to a dangling association for any configuration with Categories which are False. The resulting malformed instance for one such correct configuration is shown in Figure 4-3-2. The annotation error in Figure 4-3-1 (b) can be corrected by changing the annotation of the non-aggregate association between Category and Product from MultipleClassification | !Categories to just MultipleClassification.

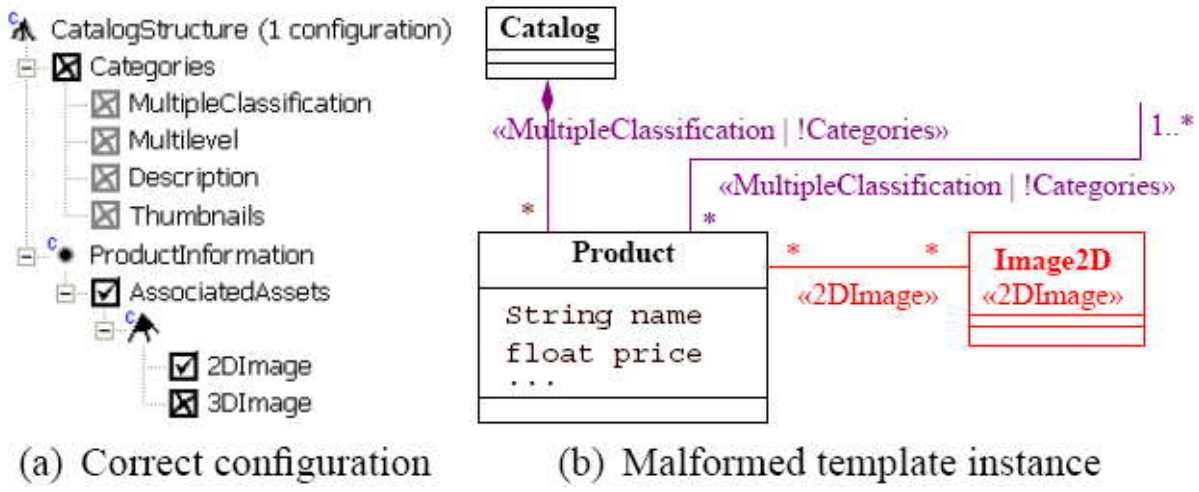


Figure 4-3-2: Sample configuration leading to a dangling association , taken from [Czarnecki, Pietroszek 06]

[Zhang *et al.* 04] propose a method for product line model's validity verification based in logic. In this approach, Ci represent a particular constraint i.

$$\exists Feature \;\; I \in FeatureSet \;,\;\; I \;\Big| = \cap_{i=I..n} C_i$$

The follow table presents a list of constraints and his logical representation that it is possible to find in a product line model:

f1 require f2	$f1 \rightarrow f2$
f1 exclude f2	$\neg(f1 \cap f2)$
Mutex-Group(f1,f2,…fn)	$\forall fj, fk \in \{ f1,f2,…fn \}, j \neq k, \neg(fj \cap fk)$
None-Group(f1,f2,…fn)	*True*
All-Group(f1,f2,…fn)	$(\cap 1 \leq j \leq n\ fj) \cup (\cap 1 \leq j \leq n\ \neg fj)$
single-binding(f1,f2,…fn)	$\exists\, 1 \leq i \leq n\ (\cap 1 \leq j \leq n, j \neq i\ \neg fj) \cap fi$
multiple-binding(f1,f2,…fn)	$\cup i=1,…,n\ fi$
all-binding(f1,f2,…fn)	$\cap i=1,…,n\ fi$
single-bound(f1,…fn) require multiple-bound(g1,…gn)	*single-bound(f1,…fn)* $\rightarrow$ *multiple-bound(g1,…gn)*

Table 4-3-2: formalisation of typical constraints. Taken from [Zhang *et al.* 04]

The idea in this technique is to apply the set of typical constraint to product line model and to evaluate it like a classical logical formula that can be true or false. The process is not automatic and the details of implementation are not showed.

On the other hand, Batory and Thaker show how, from a PLM described in terms of logical expressions, many properties of safe composition can be evaluated by AHEAD product lines tool using feature models and SAT solvers [Batory, Thaker 06]. Their technique considers verifications like this:
Refinement Constraint: If features X, Y, and Z, are refined by feature F, then
$F \Rightarrow X \vee Y \vee Z$
Let PLf be the propositional formula of product line PL:
$(PLf \wedge F \wedge \neg X \wedge \neg Y \wedge \neg Z) = false$

Also, in the Batory and Thaker's technique, the propositional formula of a grammar is considered as the conjunction of the formulas for each production, each cross-tree constraint, and the formula that selects the root feature (i.e., all products have the root feature). Thus, all constraints except ordering constraints of a feature model can be mapped to a propositional formula. This relationship of feature models and propositional formulas is essential to make a safe derivation of a particular product. The following example evidences the use of grammars like descriptors of PLMs.

// grammar of our automotive product line
Car : [Cruise] Transmission Engine+ Body ;
Transmission : Automatic | Manual ;
Engine : Electric | Gasoline ;
// cross-tree constraints
Cruise ⇒ Automatic ;

Lessons learned from correctness or satisfiability of constraints verification:

In the approach proposed by Czarnecki and Pietroszek in [Czarnecki, Pietroszek 06], creating and evolving model templates there has been an error-prone process because, for example, it is easy to forget a required constraint in the feature model or to overlook an annotation in the annotated model. While particular instances of the template that are being currently used may be correct, instantiating the

template for other configurations, which we would expect to be correct, could lead to incorrect template instances.

In this approach the verification process is applied at product line configurations and not at product line model directly. Otherwise, the verification process only makes verification of structural properties of the particular configurations and does not consider dynamic properties of the product line model. On the other side, verifications rules are not generalized mathematically by any configuration.

Batory and Thaker deal with static properties of the product line and do not support a consistency check of behavioral properties [Batory, Thaker 06]. Although the work that they have developed is supported by a computational tool (AHEAD), their work is oriented to safe composition of PLCs and not to verify the PLM itself.

4.3.3. Validity or richness

According to [Mannion 02], a valid product line model is one in which it is possible to select at least one set of single requirements from the model that satisfies the relationship between the requirements in the model. An invalid product line model is one from which it is not possible to make such a selection .

In the same way, in [Benavides *et al.* 07] we found that a feature model is valid if it exists a product satisfying all the constraints.

Techniques proposed for validity:

Mannion proposes to use propositional connectives for modelling variability and dependency between requirements, and so a logical expression can be developed for the model. And then to check satisfaction of this logical expression on each particular configuration that can be derived from PLM. This approach can be used to validate the model as a whole. The problem of this technique is the combinatory explosion generated in the evaluation process.

Benavides propose to use an algorithm to evaluate following logical expressions in order to search a PLC that satisfies all set of PLM constraints and so verifies the validity of these PLM:

$$PLM_k \models \text{Constraint Set} \wedge$$
$$\exists PLC \models \text{Constraint Set} \in PLM_k$$

And in particular :

$$\forall \text{ Constraint } C_i \in PLM, \bigwedge_{i=1}^{n} C_i \neq \bot$$

Differently to Mannion, it is not necessary to evaluate all possible particular configurations, because Benavides' validity definition deals with only one configuration that respects the set of PLM's constraints.

Lessons learned from validity:

Definitions of validity are not standardised in literature and neither techniques to verify it. The maturity level of Benavides' definition made possible to implement it in a PLM verification computational tool.

4.3.4. Suitability or utility

According to [Zhang *et al.* 04], the suitability ensures that every feature in the set of features has the possibility of being removed. If this property is not satisfied, it means that there is one or more features that will not have the chances to be removed after the current binding time. That is to say, these features actually have been bound. The possible causes may be that the operators have ignored the binding of these features, or have done some improper tailoring or binding actions, or some constraints themselves are wrong.

Techniques proposed for Suitability verification:

According to [Zhang *et al.* 04], the set of constraints in a feature model are formalised in a set of logical sentences. Then at each binding time, after an undecided feature is bound or removed, the truth value of this feature will become the logical constant True or False respectively. Thus, the verification of feature models is converted into satisfaction problems in the mathematical logic. Therefore, properties such as feature possibility of being removed from the model, can be automatically revealed through evaluation of the following formula:

$$\forall \text{ Feature } f, (\text{VariantDependency}, \text{TransverseDependency}) C \subseteq PLM,$$
$$\exists\ I_k \in \{PLC_1, ..., PLC_i\}\ (1 \le k \le i), I_k \models (\cap_{i=1..n} C_i) \cap (\neg f)$$

Lessons learned from Suitability verification:

We have adapted Zhang *et* al's proposition to be able to apply it on our meta-model. Thanks to the logical expression proposed by them, it is possible to automatize the suitability verification process in the tool that we will present in the chapter V.

4.3.5. Usability, liveliness or decidability

According to [Zhang *et al.* 04], the usability ensures that every feature in the set of susceptible features to be chosen has the possibility of being bound in the future. If this property is not satisfied, it means that there is one or more features that will not have the chances to be used in a PLC. That is to say, these features actually have been removed from the feature model. The possible causes may be that the operators have ignored the tailoring actions on these features, or have done some improper tailoring or binding actions, or some constrains themselves are wrong. These causes can be eliminated by putting these features to set of features not susceptible to be chosen, or by undoing some actions at the current binding time, or by revising constraints on features. They propose evaluation of follow formula in order to evaluate usability property:

$$\forall \text{ Feature } f, (\text{VariantDependency}, \text{TransverseDependency}) C \subseteq PLM,$$
$$\exists\ I_k \in \{PLC_1, ..., PLC_i\}\ (1 \le k \le i), I_k \big| = \left(\cap_{i=1..n} C_i\right) \cap f$$

Techniques proposed for Usability verification:

We have found Zhang *et al*'s technique [Zhang *et al.* 04] based on logical evaluation of constraints. In this technique, properties such as feature possibility of being selected from the model, can be automatically revealed through evaluation of the following formula:

$$\forall \text{ Feature } f, (\text{VariantDependency}, \text{TransverseDependency}) C \subseteq PLM,$$
$$\exists\ I_k \in \{PLC_1, ..., PLC_i\}\ (1 \le k \le i), I_k \big| = \left(\cap_{i=1..n} C_i\right) \cap f$$

Lessons learned from Usability verification:

We have adapted Zhang *et* al's proposition to be able to apply it on our meta-model. Thanks to the logical expression proposed by them, it is possible to automatize the usability verification process in the tool that we will present in the chapter V.

4.3.6 Verifiability

According to [Mannion, Camara 03] a verifiable single system (PLC) is one for which its product line logical expression evaluates to TRUE. Otherwise the single system is unverifiable.

Techniques proposed for verifiability:

According to [Mannion, Camara 03], a product line model can be represented using propositional logic. By considering each requirement as an atom and each relationship between requirements as a logical expression, a logical expression for the product line model can be developed. A selected combination or requirements draw from the product line model can then be tested using this expression. In order to illustrate this technique, let us consider the following "mobile phone product line example":

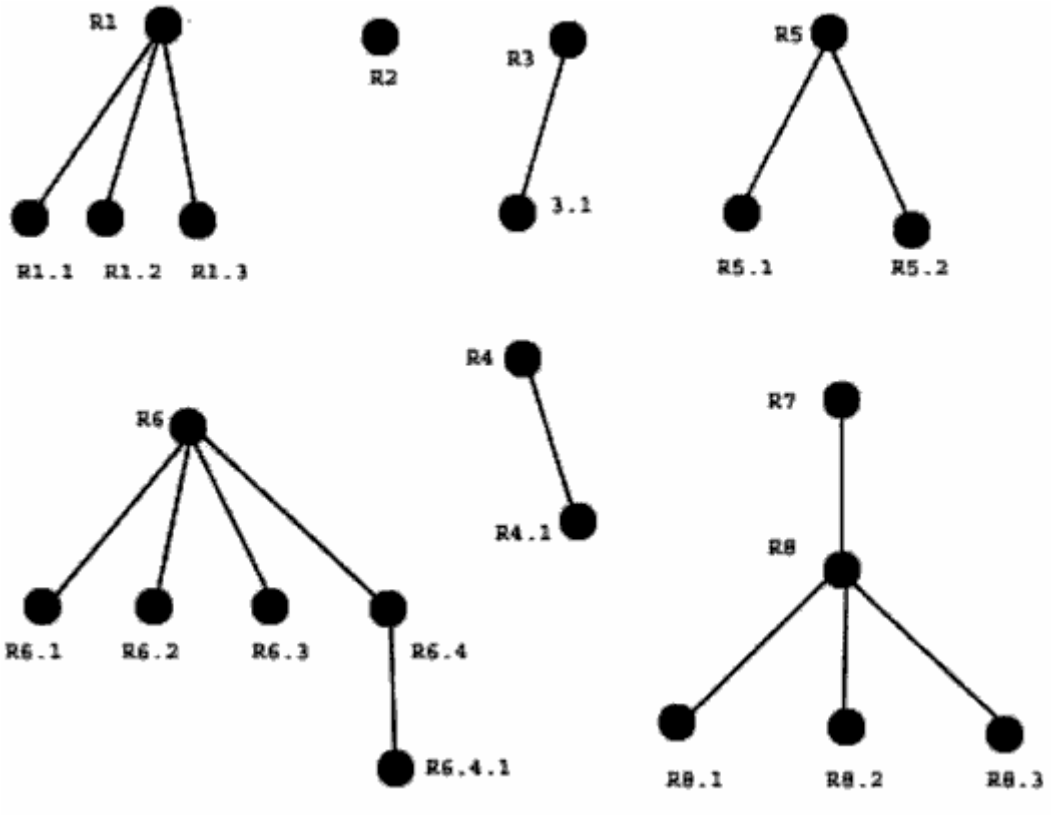


Figure 4-3-3: mobile phone product line model, taken from [Mannion, Camara 03]

The logical expression for the mobile phone product requirements shown in Figure 4-3-3 are:

((R1 ^ (R1.1 ^ R1.2 ^ R1.3))^ (G1)

(R2) (G2)

(R3 ^ R3.1) ^ (G3)

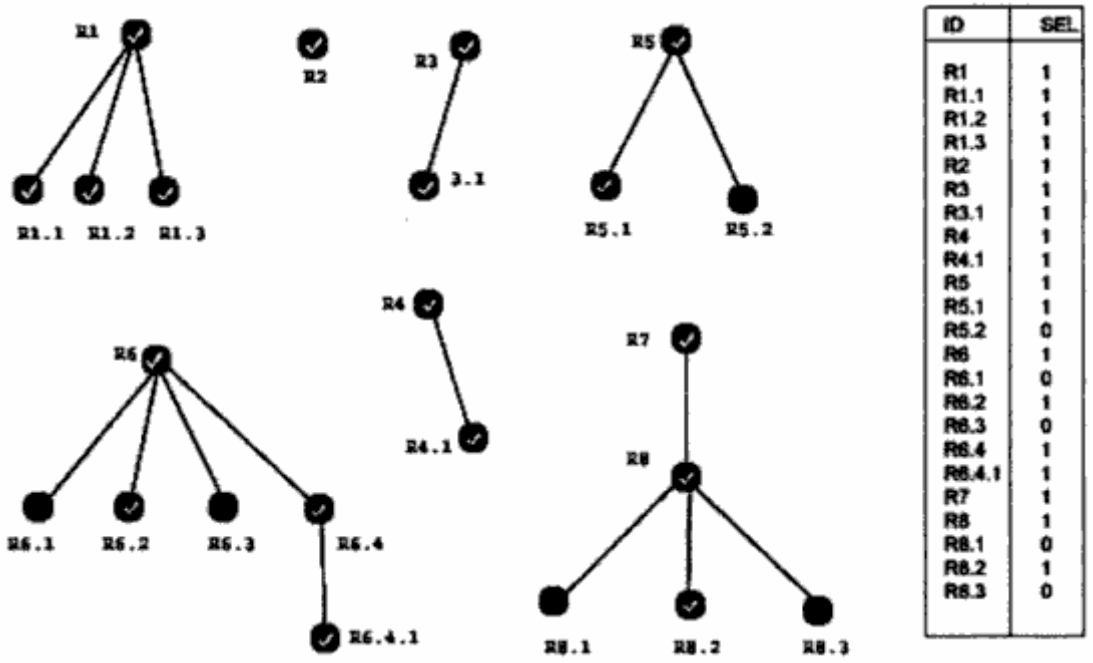
(R4 ^ R4.1) ^ (G4)

(R5 ^ (R5.1 ⊕ R5.2)) ^ (G5)

(R6 ^ (R6.1 v R6.2 v R6.3 v (R6.4 ^ R6.4.1))) ^ (G6)

(R7 ⇔ (R8 ^ (R8.1 v R8.2 v R8.3))) (G7)

It is possible to instantiate the requirements in the logical expression to TRUE or FALSE depending on whether they appear in the single system or not. A verifiable single system is one for which the product line logical expression evaluates to TRUE. Otherwise this PLC is unverifiable.



ID	SEL
R1	1
R1.1	1
R1.2	1
R1.3	1
R2	1
R3	1
R3.1	1
R4	1
R4.1	1
R5	1
R5.1	1
R5.2	0
R6	1
R6.1	0
R6.2	1
R6.3	0
R6.4	1
R6.4.1	1
R7	1
R8	1
R8.1	0
R8.2	1
R8.3	0

Figure 4-3-4: requirement selection and equivalent boolean vector representation.

During the construction of a single system, TRUE (T) is assigned to those requirements that are selected and FALSE (F) is assigned to those not selected.

Suppose some requirements are freely selected (indicated by √ in Figure 4-3-4) from the mobile phone product line model. The product line logical expression becomes:

((T^(T^T^T^T))^ (G1)

(T) (G2)

(T^(T^T)) (G3)

(T^T) (G4)

(T^(T ⊕ F))^ (G5)

(T^(FvTvTv(T^T)))^ (G6)

(T ⇔ (T^(FvTvF))) (G7)

(G1), (G2), (G3), (G4), (G5), (G6) and (G7) each one evaluate to TRUE. Hence G1^G2^G3^G4^G5^G6^G7 evaluates to TRUE.

But this technique considers only structural verification of PLM through evaluation of PLCs transformed into logical expressions, which is very expensive from a computational point of v iew.

Lessons learned from verifiability:

Mannion and Camara's approach is easy to automatize, but only consider evaluation of verifiability on PLCs. For the moment, we have not found a technique that permits to evaluate the verifiability's aspects in PLMs, we will take this lack as an open issue in our future research work.

4.4. General lessons

General lessons can be resumed in Table 4-4-1. This table permits to present what criterion is being verified using some technique and how verification process is being achieved.

Technique / **Criterion**	Matching	Queries	Model checking	Manual testing	Counter - example testing
Consistency	Semantic Web Ontologies comparison. PLC's ontologies against PLM ontology [Wang *et al.* 05]	Make SQL queries looking for inconsistencies in the constraints and SQL queries looking for PLCs that did not satisfy a	For a given invariant i, an algorithm searches for a valid path from the start state to a state that violates i [Lauenroth, Pohl 07] Safe composition of		

	Each commonality in the PLM must be presented in each PLC [Bayer *et al.* 99]	particular commonality [Padmanabhan, Lutz 05]	products by ensuring that there is no undefined element [Batory and Thaker 06]		
Correctness or Satisfiability of Constraints			OCL constraints are transformed to propositional formulas, which are then fed into a SAT solver [Czarnecki, Pietroszek 06] From a PLM described in terms of logical expressions, many properties of safe composition can be evaluated by AHEAD product lines tool. Safe composition is made by means of classical grammar derivations [Batory, Thaker 06]	For each feature, to apply the set of PLM's typical constraint [Zhang *et al.* 04]	
Validity or richness			To use an algorithm looking for a particular		To develop a logical expression for

			configuration that satisfied all set of PLM constraints [Benavides *et al.* 07]		the PLM and to check its satisfaction on each particular configuration [Mannion 02].
Suitability or utility					Feature possibility of be removed from the model, can be automatically revealed through evaluation of a logical expression [Zhang *et al.* 04]
Usability , liveliness or decidability			Feature possibility of being used or decidability to use it in a PLC, can be automatically revealed through evaluation of a logical expression [Zhang *et al.* 04]		
Verifiability			A PLC is verifiable if its product line logical expression evaluates to TRUE.		

			An algorithm is proposed to do this evaluation [Mannion, Camara 03].		

Table 4-4-1: Approaches presented to evaluate critters of PLM verification using certain techniques

4.5. Conclusion

The first part of this chapter was dedicated to survey the current state of the art of research in verification process. Next, we have presented our PLM meta-model in order to define the set of concepts and their inter-relation. These concepts and constraints had been used to formalize each characteristic to be verified in a PL or a PLC model. For each characteristic, we have presented the set of techniques used for its verification and some lessons that we will take into account in the definition and improvement stages of our approach.

The next Chapter presents MAP and NATURE formalisms. Next, we use these formalisms in order to define our multi-process model and explaining systematically each of his sections.

Part V

Verification Multi-method

5 The Approach

Concisely, we propose three tasks to achieve in order to verify a PLM. In our approach, the firs step is feature model construction supported by a computational tool. In thi s step, industrial product line models can be constructed according to FORE formalism [Streitferdt 03]. The second step is to formalize constraints on features into first order logic sentences. For the second step we propose a first set of criteria that imperatively have to be respected by all feature product line model. In the third step w e check a PLC (and so on for each PLC), and their interrelation with PLM to determine if these PLC satisfied PLM's structure. With the aim of modeling PLM's process of ve rification, we have decided to use the Map formalism, this formalism and the process proposed by us, will be dealt at following sections.

5.1 Context and MAP formalism

We use the Map formalism proposed in [Rolland et al. 99] and [Benjamen 99] to express the process model of our approach. Map provides a representation system allowing us to combine multiple ways of working into one complex process model. It is based on a nondeterministic ordering of two fundamental concepts, *intentions* and *strategies.*

An intention represents a goal that can be achieved by the performance of the process. It refers to a task (activity) that is a part of the process and is expressed at the intentional level. A strategy represents the manner in which the intention can be achieved. Therefore, the map is a directed labelled graph with notes representing intentions and labelled edges expressing strategies. As shown in Figure 5-1-1, a map consists of a number of *sections* each of which is a triplet <*Ii, Ij,* Sij>. The directed nature of the map identifies which intention can be done after a given one. A map includes two specific intentions, *start* and *stop*, to begin and to end the process respectively. There are several paths from *start* to *stop* in the map for the reason that several different strategies can be proposed to achieve the intentions.

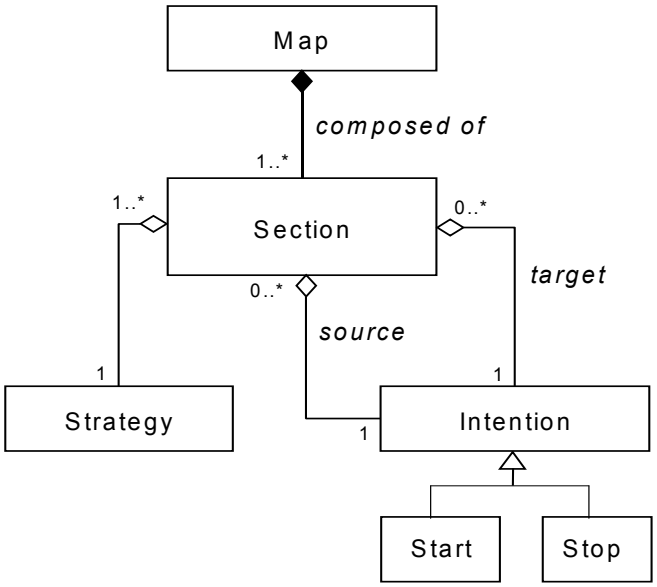


Figure 5-1-1: The process map meta-model

According to the Figure 5-1-1, a process map is composed of a collection of *sections* (at least one). Any section belongs to a single map. Each section has one source intention, one target intention, and is related to a single strategy.

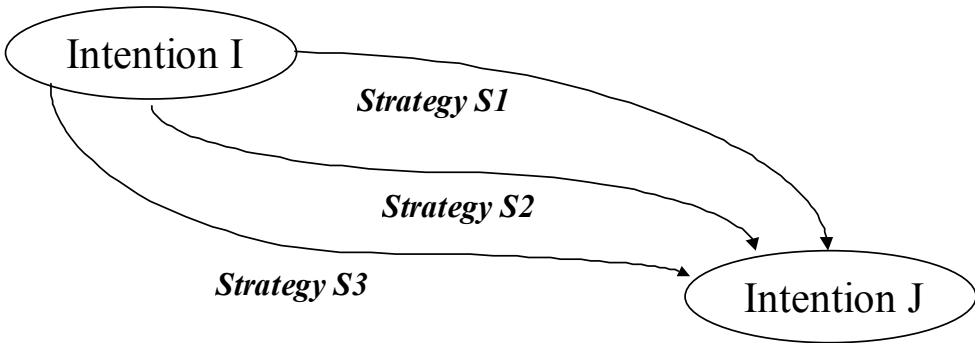


Figure 5-1-2: The concept of section

As we can see in figure 5-1-2, a *strategy* is an approach, a manner to achieve an intention. The strategy, as part of the triplet <*Ii,Ij*,Sij> characterizes the flow from *Ii* to *Ij* and the way *Ij* can be achieved. The specific manner in which an intention can be achieved is captured in a section of the map whereas the various sections having the same intention *Ii* as a source and *Ij* as target show the different strategies that can be adopted for achieving *Ij* when coming from *Ii*. Similarly, there can be different sections having *Ii* as source and *Ij1, Ij2, ...Ijn* as targets. This shows the different intentions that can be achieved after the achievement of *Ii*.

5.2 MAP model of our approach

Map is a formalism that permits to draw multi-process models in an easy and intuitive manner. So, Map permits integrating techniques that have been developed in an independent way, to combine them in a coherent process. The next process model is an integration of different process model adapted from literature. That is, a multi-process model designed by the Map formalism. Broadly speaking, properties of a PLM that must be considered in a verification process can be categorized into structural and semantic properties. Properties that deal with structural correctness can be verified before starting verification process of semantic properties. In order to do these two verification processes there is no single strategy to achieve it. Some of the most popular strategies permitting automatic verification of a conceptual model are formal proofs, model checking (at the same time composed of: constraint satisfaction problem CSP, boolean satisfaction problem SAT and binary decision diagram BDD), matching with another model and searching of a counter example. On the other hand, in order to verify certain properties of a PLM, some PLCs must be considered. By this reason, verification of PLCs is included into multi-method presented in Figure 5-2-1.

Table 4-4-1 (<criterion, technique>) has permitted us defining the different strategies that can be used to achieve a determined intention in a process of product lines models verification. In this work we only consider correctness verification of product line models and validity of product models. By this reason the process model in Figure 5-2-1 deals with static and semantic correctness of PLMs and validity of PLCs models. Others properties that have been analysed in precedent chapters will be integrated at this process model in feature works.

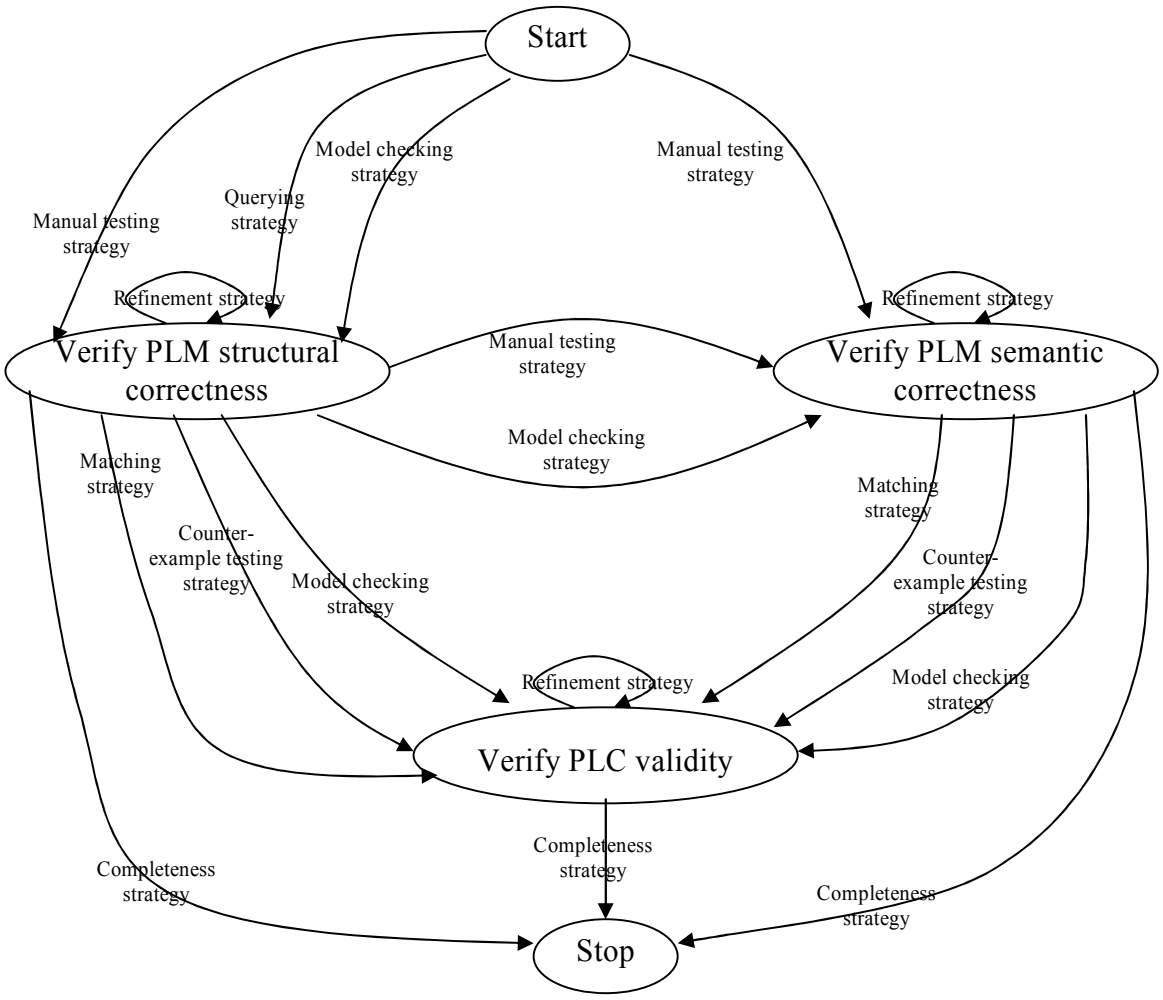


Figure 5-2-1: Process model of our approach using the Map formalism

5.3 Context models of the MAP

The oriented-contextual models define the process by combining observable situations with a set of specific intentions. The work to accomplish is described in the process as being dependent on both situation and intention. Central concept in context-oriented models: couple <situation, intention>

Using the concept of a decision allows to apply the properties of decision-oriented models previously seen. The decisions are applied to situations where the process is now in order to transform this situation into a new desired one. Strong relationship between situation and decision helps focus the guidance, track and explanations on the specific situations of the process .

The *context-driven* process model is based on the NATURE process modeling formalism [Jarke *et al.* 99], [Rolland *et al.* 96]. According to this formalism, a process model can be expressed as a hierarchy of

contexts. A context is viewed as a couple <*situation, intention*>. The *situation* represents the part of the product undergoing the process, and the *intention* reflects the goal to be achieved in this situation. NATURE proposes three types of guidelines, namely choice, plan and executable. The body of a choice guideline offers different alternative ways for achieving the process intention. Arguments are provided to help in the selection of the most appropriate alternative. A plan guideline can be looked as dealing with a macro issue which is decomposed into sub-issues, each of which corresponds to a sub-decision. An executable guideline corresponds to an operationalizable intention that is directly applicable through a set of activities. The body of an executable guideline proposes a set of activities to be performed for achieving its process intention.

For instance, in CG1 below:
<*PLM, execute invariants*> is a choice guideline that includes: <*PLM, execute a constraint satisfaction problem analysis*>, <*PLM, execute a boolean satisfaction problem analysis*> and <*PLM, execute a binary decision diagram analysis*>.

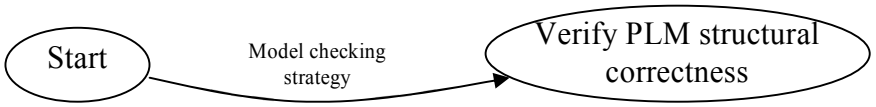
At the same time <*PLM, execute a boolean satisfaction problem analysis*> is a plan guideline, marked by the symbol '@' in order to reuse it in other place of the tree structure .

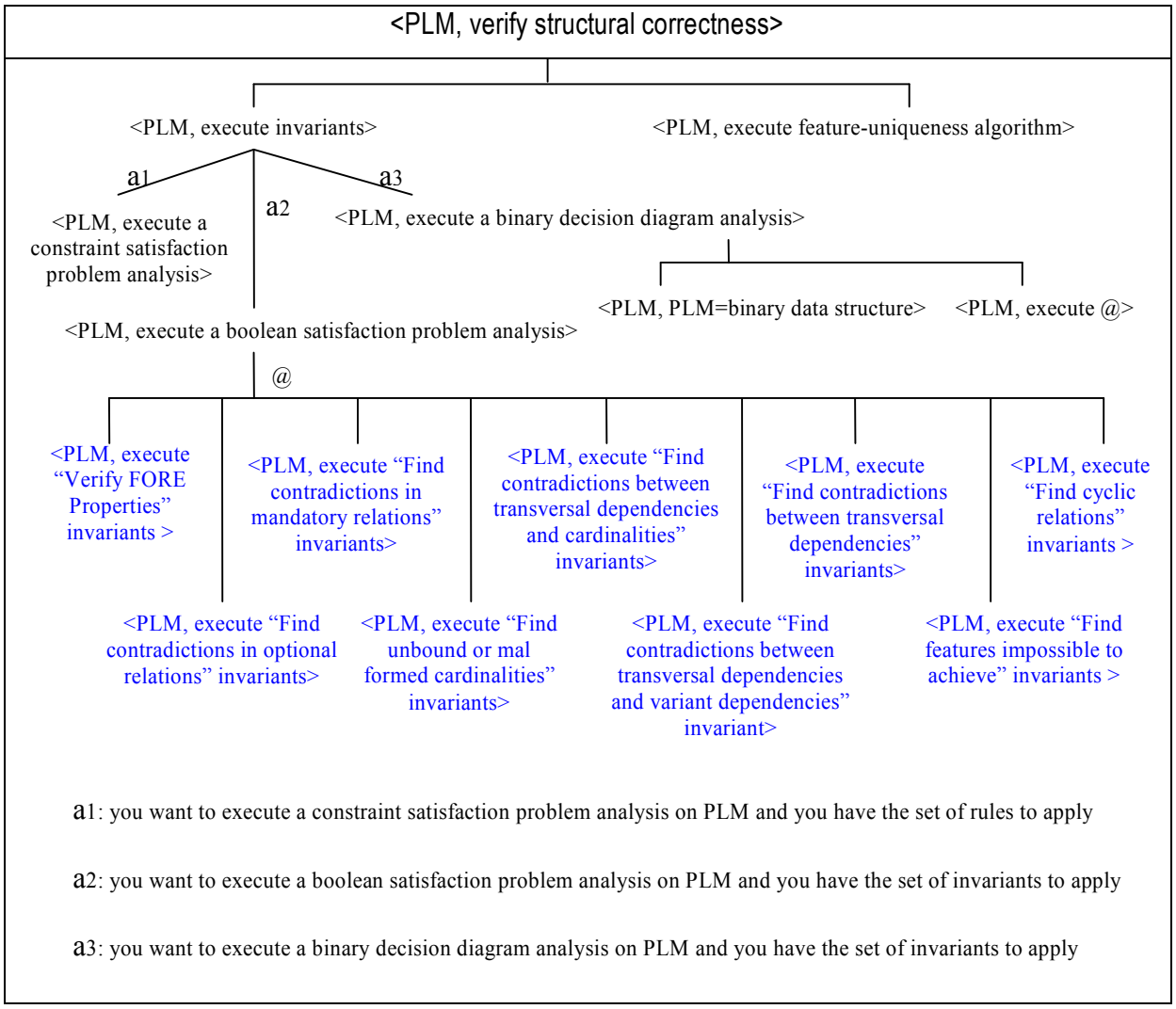
And in the same way, <*PLM, PLM = binary data structure*> and <*PLM, execute require-relation invariant*> are examples of executable guidelines, because they are directly operationalized. <PLM, execute "Find contradictions in mandatory relations" invariants> is a guideline that evaluates the following logical formulae or invariant:

$$\exists\{(feature\ P, feature\ C)_i, feature\ K\} \subseteq PLM.\ FatherFeature(P) \wedge ChildFeature(C)$$
$$\wedge (C\ childOf\ P) \wedge (C\ Mandatory\ P) \models \wedge_i (C \bullet P) \wedge \neg(K \leftrightarrow C)$$

In this section we present the context guidelines for sections developed in our approach and supported by the computational tool described in chapter V.

CG1: Section <*Start, verify PLM structural correctness, Model checking strategy*>





Next we present the list of invariants that must be evaluated in each executable guideline proposed in CG1 with blue colour.

< PLM, execute "Verify FORE Properties" invariants >

It is necessary to execute:

1. Root unicity invariant :

$$\exists featuref \in PLM.\ f = root \wedge |f| = 1$$

2. Child – Father unicity invariant:

$$\forall (feature\ P_i, feature\ C) \in PLM.\ FatherFeature(P_i) \wedge ChildFeature(C)$$
$$\wedge (C\ childOf\ P_i) \wedge ((C\ Mandatory\ P_i) \oplus (C\ Optionally\ P_i)) \Rightarrow |(C \bullet P_i) \oplus (C \circ P_i)| = 1$$

3. Cardinality relation invariant:

$$\forall Cardinality\ D_a(m,n),(fatherFeature\ P, childFeature\{C_1,...C_k\})_a \ \Big| =$$

$$\left\{\underbrace{C_i,...,C_j}_{m}\right\} \vee \left\{\underbrace{C_i,...,C_{j+1}}_{m+1}\right\} \vee ... \vee \left\{\underbrace{C_i,...,C_{k-1}}_{n-1}\right\} \vee \left\{\underbrace{C_i,...,C_k}_{n}\right\},$$

$$m \in \mathrm{N} \cup \{0\} \wedge n \in \mathrm{N} \cup \{*\} \wedge 0 \le m \wedge (m \le i \le n = * \le k)$$

4. Invariant evaluating optionality of relations intervening in a cardinality :

$$\forall Cardinality\ C(m,n),(fatherFeature\ P, childFeature\{C_1,...C_k\}) \in C \ \Big| =$$

$$Optional(C_1),...,Optional(C_k), m \in \mathrm{Z} \wedge n \in \mathrm{Z} \cup \{*\} \wedge 0 \le m \wedge (m \le n = * \le k)$$

<PLM, execute “Find contradictions in optional relations” invariants>

It is necessary to execute:

1. Optionally relationship constraint invariant:

$$\exists\{(feature\ P, feature\ C)_i, feature\ K\} \subseteq PLM.\ FatherFeature(P) \wedge ChildFeature(C)$$

$$\wedge (C\ childOf\ P) \wedge (C\ Optionally\ P) \ \Big| = \vee_i (C \circ P) \wedge \neg (K \ \rightarrow \ C)$$

2. Invariant to validate if a cardinality is right or not:

$$\exists\{parentFeature(f_a), featureSet(childFeature(f_i,...,f_j)), cardinality(m..n)\} \subseteq PLM$$

$$m < n,\ \ m,n \in [i,j] \wedge (\oplus_{x=1}^{n-m} (\wedge_1^{m+x-1} (f_i \wedge \neg\{f_{i+1}...,f_j\} \vee f_k \wedge \neg\{f_i,...,f_{k-1},f_{k+1},...,f_j\}$$

$$\vee f_j \wedge \neg\{f_i,...,f_{j-1}\})) \wedge (count_{z=i}^{j}(f_z optional) \le n < j) \ \Big| = true$$

<PLM, execute “Find contradictions in mandatory relations” invariants>

It is necessary to execute:

1. Mandatory relationship constraint invariant:

$$\exists\{(feature\ P, feature\ C)_i, feature\ K\} \subseteq PLM.\ FatherFeature(P) \wedge ChildFeature(C)$$

$$\wedge (C\ childOf\ P) \wedge (C\ Mandatory\ P) \ \Big| = \wedge_i (C \bullet P) \wedge \neg (K \ \leftrightarrow \ C)$$

<PLM, execute “Find unbound or badly formed cardinalities” invariants>

It is necessary to execute:

1. Invariant to validate if a cardinality is right or not:

$$\exists\{parentFeature(f_a), featureSet(childFeature(f_i,...,f_j)), cardinality(m..n)\} \subseteq PLM$$

$$m < n,\ \ m,n \in [i,j] \wedge (\oplus_{x=1}^{n-m} (\wedge_1^{m+x-1} (f_i \wedge \neg\{f_{i+1}...,f_j\} \vee f_k \wedge \neg\{f_i,...,f_{k-1},f_{k+1},...,f_j\}$$

$$\vee f_j \wedge \neg\{f_i,...,f_{j-1}\})) \wedge (count_{z=i}^{j}(f_z optionaly) \le n < j) \ \Big| = true$$

2. Invariant to evaluate cardinalities whit value 0,0:

$$\forall\{parentFeature(f_a), featureSet(childFeature(f_i,...,f_j)), cardinality(m..n)\} \subseteq PLM$$
$$(f_i\ childOf\ f_a),...,(f_j\ childOf\ f_a) \mid \neq$$
$$(m = n = 0) \vee (n < m) \vee (m < 0) \vee (\neg f_i \wedge ... \wedge \neg f_j \wedge (m \neq 0)) \mid = false$$

<PLM, execute "Find contradictions between transversal dependencies and cardinalities" invariants>

It is necessary to execute:

1. An algorithm considering interactions between transverse dependencies and cardinalities , this is not still available.

<PLM, execute "Find contradictions between transversal dependencies and variant dependencies" invariant>

It is necessary to execute:

1. Require-child invariant:

$$\exists features\ (fm_i, fm_k)_{i \neq k} \in PLM, fm_i \neg childOf\ fm_k \mid = fm_i \rightarrow fm_k$$

2. Exclude definition and invariant:

$$ExcludeDefinition : \forall features(fm_i, fm_k) \in PLM.\ exclude(fm_i, fm_k) \Rightarrow (fm_i \rightarrow \neg fm_k)$$
$$\forall features(f_i, f_j, f_k)_{i \neq j} \in PLM.\ (f_j\ childOf\ f_i) \wedge (f_j\ mandatory\ f_i) \mid = \neg\exists(f_k \leftrightarrow f_j)$$

3. Require definition and invariant:

$$\mathrm{Re}quireDefinition : \forall features(fm_i, fm_k) \in PLM.\ require(fm_i, fm_k) \Rightarrow (fm_i \rightarrow fm_k)$$
$$\forall features(f_i, f_j, f_k)_{i \neq j} \in PLM.\ (f_j\ childOf\ f_i) \wedge (f_j\ optionally\ f_i) \mid = \neg\exists(f_k \rightarrow f_j)$$

<PLM, execute "Find contradictions between transversal dependen cies" invariants>

It is necessary to execute:

1. Require-exclude definition:

$$\mathrm{Re}quireDefinition : \exists features\ (fm_i, fm_k)_{i \neq k} \in PLM, fm_i \neg childOf\ fm_k \mid = fm_i \rightarrow fm_k$$

2. Transversal dependencies contradiction invariant:

$$\forall features(fm_i, fm_k) \in PLM.\ require(fm_i, fm_k) \Rightarrow (fm_i \rightarrow fm_k)$$
$$\forall features(fm_i, fm_k) \in PLM.\ exclude(fm_i, fm_k) \Rightarrow (fm_i \rightarrow \neg fm_k)$$
$$\forall features(f_j, f_k)_{j \neq k} \in PLM \wedge (f_k\ requires\ f_j) \mid =$$
$$\neg\exists((f_k \leftrightarrow f_j) \vee (f_j \leftrightarrow f_k) \vee (f_k \leftrightarrow ancestorOf(f_j)))$$

<PLM, execute "Find features impossible to achieve" invariants >

It is necessary to execute:

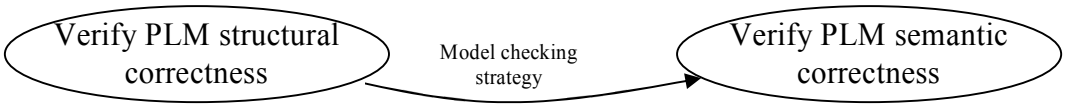
1. Require-exclude definition:

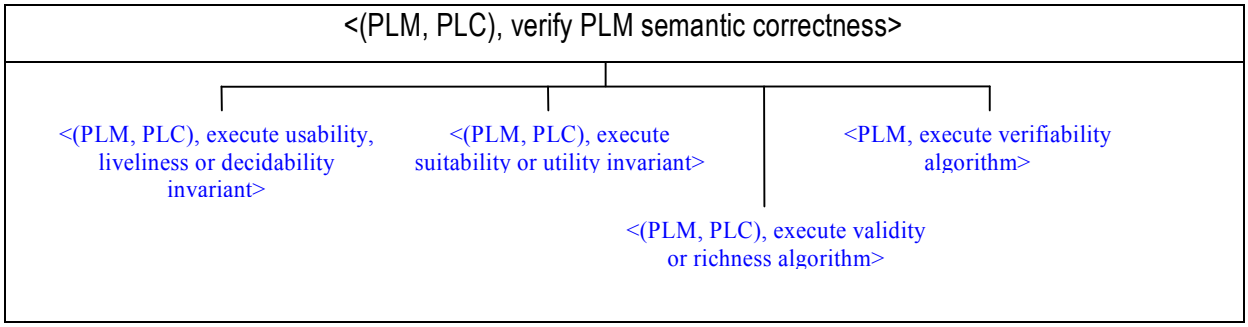
$$\mathrm{Re}\,quireDefinition: \exists features\ (fm_i, fm_k)_{i\neq k} \in PLM, fm_i \neg childOf\ fm_k \ \big| = fm_i \rightarrow fm_k$$

2. Transversal dependencies contradiction invariant:

$$\forall features(fm_i, fm_k) \in PLM.\ require(fm_i, fm_k) \Rightarrow (fm_i \rightarrow fm_k)$$
$$\forall features(fm_i, fm_k) \in PLM.\ exclude(fm_i, fm_k) \Rightarrow (fm_i \rightarrow \neg fm_k)$$
$$\forall features(f_j, f_k)_{j\neq k} \in PLM \wedge (f_k\ requires\ f_j)\ \big| =$$
$$\neg\exists((f_k \leftrightarrow f_j) \vee (f_j \leftrightarrow f_k) \vee (f_k \leftrightarrow ancestorOf(f_j)))$$

CG2: Section <*Start, verify PLM structural correctness, Model checking strategy*>





<(PLM, PLC), execute usability, liveliness or decidability invariant>

It is necessary to execute:

1. Liveliness, usability or decidability invariant:

$$\forall \text{ Feature } f, (\text{VariantDependency}, \text{TransverseDependency}) C \subseteq PLM,$$
$$\exists\ I_k \in \{PLC_1, ..., PLC_i\}\ (1 \leq k \leq i), I_k \big| = (\cap_{i=1..n}\ C_i) \cap f$$

<(PLM, PLC), execute suitability or utility invariant>

It is necessary to execute:

1. Utility or suitability invariant

$\forall$ Feature f, (VariantDependency, TransverseDependency) $C \subseteq PLM$,
$\exists\ I_k \in \{PLC_1,...,PLC_i\}\ (1 \le k \le i), I_k \models (\cap_{i=1..n}\ C_i) \cap (\neg f)$

<(PLM, PLC), execute validity or richness algorithm>

It is necessary to execute:

1. An algorithm that permits to evaluate:

   $PLM_k \models$ Constraint Set $\wedge$
   $\exists PLC \models$ Constraint Set $\in PLM_k$

   And in particular :

   $\forall$ Constraint $C_i \in PLM, \bigwedge_{i=1}^{n} C_i \not\models \perp$

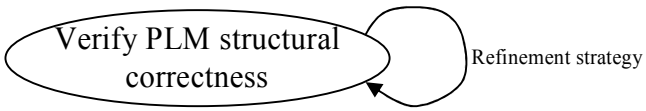
<(PLM, PLC), execute verifiability algorithm>

It is necessary to execute:

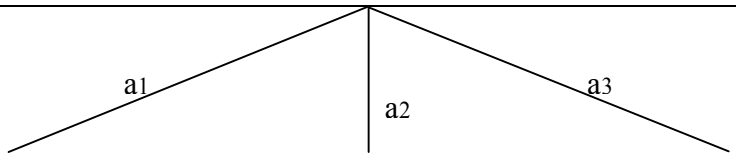
1. An algorithm that permits to evaluate:

   $\forall$ logicalExpression $G = SubG_i, SubG_j.\ SubG_i \wedge SubG_j = true$

CG3: Section <*verify PLM structural correctness, verify PLM structural correctness, Refinement strategy*>



<PLM, verify PLM structural correctness>



<PLM, implement manual testing strategy>

<PLM, implement formal proofs strategy>

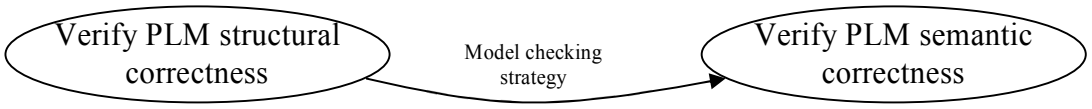
<PLM, implement model checking strategy: CG1>

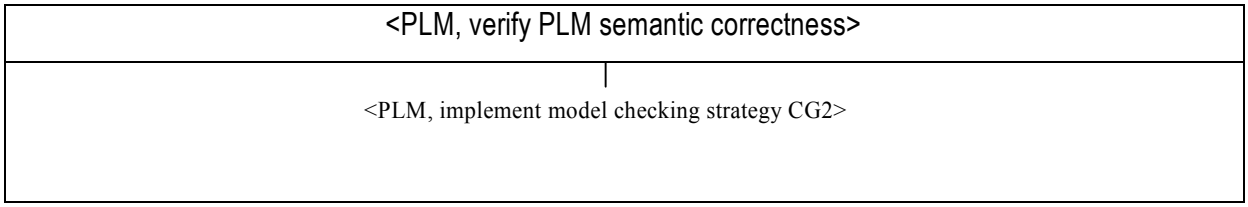
a1: you want to refine the PLM structural correctness verification process by means of "manual testing" strategy

a2: you want to refine the PLM structural correctness verification process by means of "formal proofs" strategy

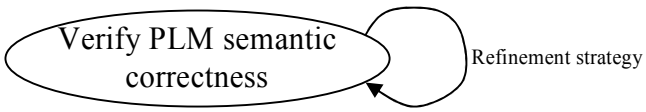
a3: you want to refine the PLM structural correctness verification process by means of "model checking" strategy

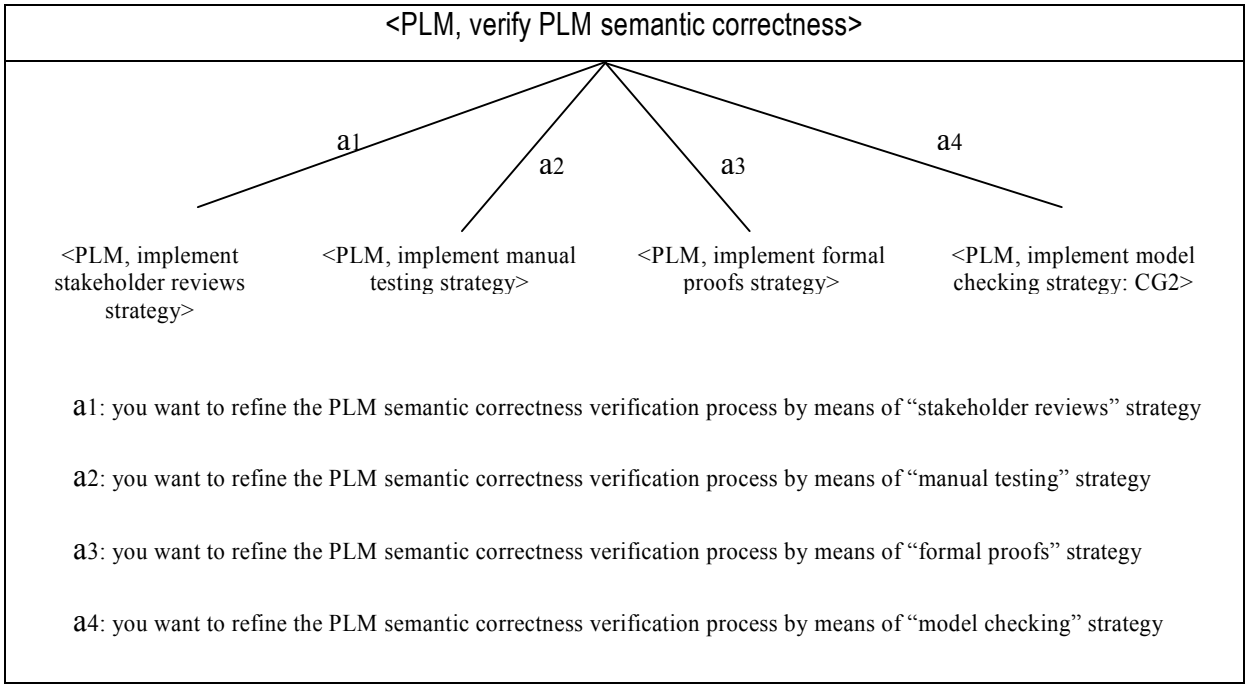
CG4: Section <*verify PLM structural correctness, verify PLM semantic correctness, model checking strategy*>





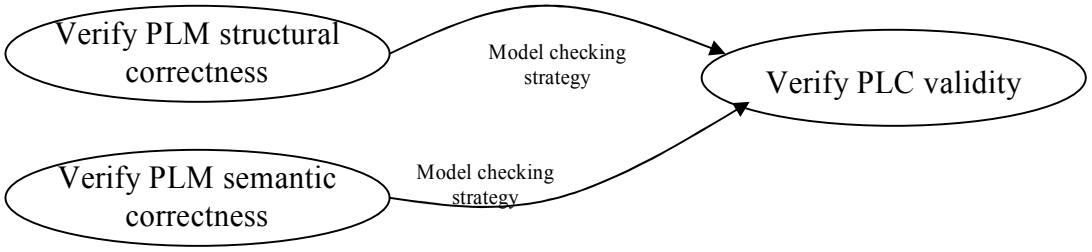
CG5: Section <*verify PLM semantic correctness, verify PLM semantic correctness, refinement strategy*>





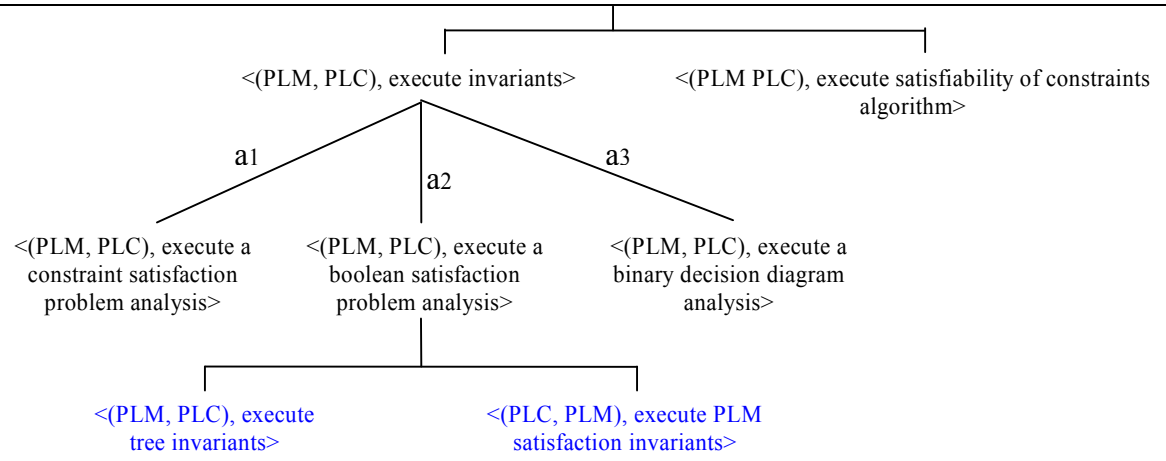
CG6: Section <*verify PLM structural correctness, verify PLC validity, model checking strategy*>

CG7: Section <*verify PLM semantic correctness, verify PLC validity, model checking strategy*>



<(PLM, PLC, PLM structure & PLM semantic = "correct"), verify PLC validity>

<(PLM, PLC, PLM structure & PLM semantic = "correct"), verify PLC validity>



a1: you want to find values or states in which all constraints are satisfied

a2: you want to evaluate logical expressions

a3: you want to evaluate a binary data structure representing a boolean function

<(PLM, PLC), execute tree invariants>

It is necessary to execute:

1. Root unicity invariant

$$\forall PLC, \exists feature \quad f \in (PLM \cap PLC).\ f = root \wedge |f| = 1$$

2. Child-parent unicity invariant

$$\forall (feature\ P_i, feature\ C) \in PLM.\ FatherFeature(P_i) \wedge ChildFeature(C)$$
$$\wedge (C\ childOf\ P_i) \wedge ((C\ Mandatory\ P_i) \oplus (C\ optional\ P_i)) \Rightarrow |(C \bullet P_i) \oplus (C \circ P_i)| = 1$$

<(PLC, PLM), execute PLM satisfaction invariants>

It is necessary to execute:

1. Satisfaction of PLM's mandatory relationships invariant

$$\exists\{(feature\ P, feature\ C)_i, feature\ K\} \subseteq PLM \cap PLC.\ FatherFeature(P)$$
$$\wedge ChildFeature(C) \wedge (C\ childOf\ P) \wedge (C\ Mandatory\ P) \mid = \wedge_i (C \bullet P) \wedge \neg (K \leftrightarrow C)$$

2. Satisfaction of PLM's optionally relationships invariant

$$\exists\{(feature\ P, feature\ C)_i, feature\ K\} \subseteq PLM \cap PLC.\ FatherFeature(P)$$
$$\wedge ChildFeature(C) \wedge (C\ childOf\ P) \wedge (C\ Optionally\ P) \mid = \vee_i (C \circ P) \wedge \neg (K \rightarrow C)$$

3. Satisfaction of PLM's require relationships invariant

$$\forall features\ (fc_i, fc_k) \in PLM, fc_i \in PLC.\ require(fc_i, fc_k) \Rightarrow (fc_i \in PLC \wedge fc_k \in PLC)$$

4. Satisfaction of PLM's exclude relationships invariant

$$\forall features\ (fc_i, fc_k) \in PLM, fc_i \in PLC.\ exclude(fc_i, fc_k) \Rightarrow (fc_i \in PLC \wedge fc_k \notin PLC)$$
$$\forall features\ (fc_i, fc_j, fc_k) \in PLM, (fc_i, fc_j) \in PLC.$$
$$exclude(fc_i, fc_j) \wedge (fc_k\ childOf\ fc_j) \Rightarrow (fc_i \in PLC \wedge (fc_j, fc_k) \notin PLC)$$

5. Algorithm to verify satisfiability of PLM's constraints

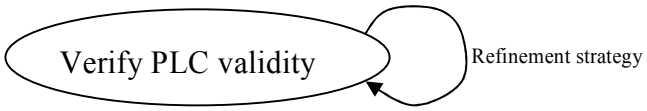
$$\forall i \in \text{invariantSet}, (\exists\ path\ P \in PLC \mid \neq i \wedge \exists (P, i) \subset PLM) \mid = \perp$$

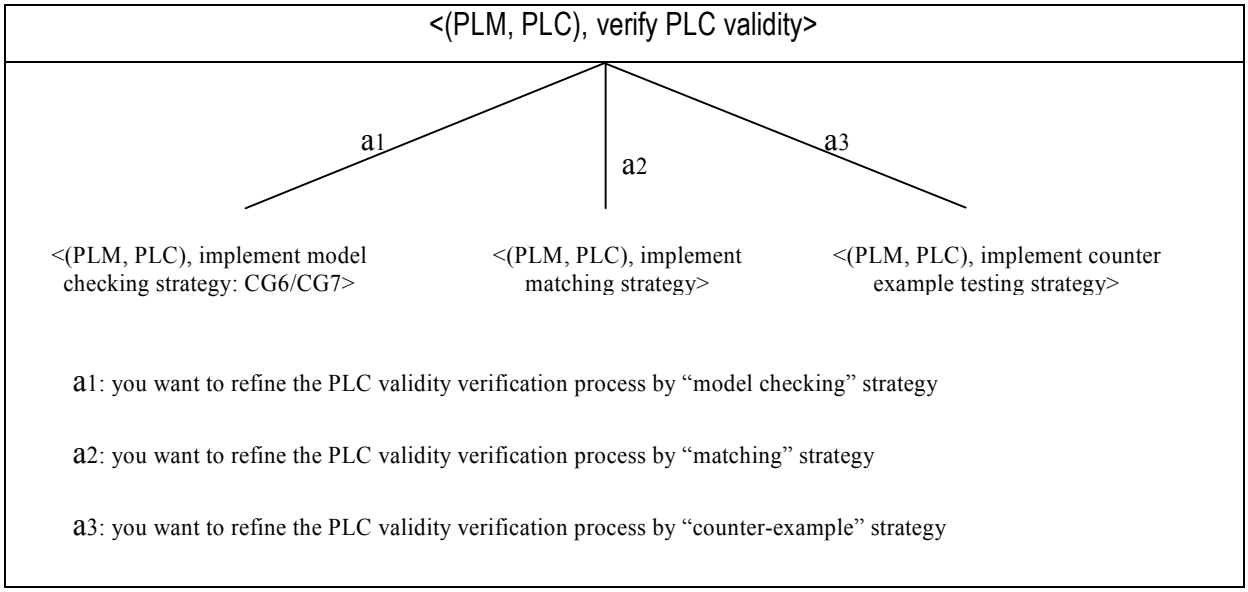
For **invariant i**:

Search in **PLC** a valid **path P** from an initial state into a one that violates i.

If a **PLC** exists with **P**, **PLM** is inconsistent

CG8: Section <*verify PLC validity, verify PLC validity, Refinement strategy*>





5.4 Discussion

Now that the approach has been presented, this section will discuss several of its contributions and limitations.

There are two aspects that must be taken into account in a verification process of PLM's correctness, the structural and the semantic correctness of the model. In the verification process of PLM's structural correctness, we have included the list of the notation FORE's characteristics, that they are largely treated in literature. From the point of view of the structural correctness, we have enriched the range of properties to verify in a PLM and we have proposed the corresponding logic invariants to evaluate each one. We have included not only the properties of a model designed with the notation FORE, but also another one like: correctness in optional relations, correctness in mandatory relations, correctness in cardinalities, correctness in transversal dependencies and variant dependencies interactions, correctness in transversal dependencies interactions, every feature must be possible to achieve and no cyclic relations are permitted. And from the point of view of the semantic correctness, we have unified dispersed criteria found in literature and have select some of the most important characteristics to be

verified in a PLM. With regard to the unification of criteria, we claim that the terms *Usability, liveliness and decidability* refer to the same concept; we have established also equivalency of concepts among suitability and utility; as well as for the terms *validity* and *richness*. With respect to verification, some logic invariants to evaluate usability, suitability, validity and verifiability properties have been proposed. The different paths to execute each one of the previous logic invariants on a PLM are defined in the multi-process model in Figure 5-2-1.

For the verification of each of correctness' characteristics, we have used propositional logic and first-order logic for writing out every invariant to verify. In order to evaluate a propositional logic formula, we have used in some cases satisfiability criterion and validity criterion in others. A formula F is satisfiable if there is an interpretation K such that K |= F. A formula F is valid if for all interpretations K, K |= F. Determining satisfiability and validity of formulae are important tasks in logic. Satisfiability and validity are dual concepts, and switching from one to the other is easy. F is valid if ¬F is unsatisfiable. For example, suppose that F is valid; then for any interpretation K, K |= F. By the semantics of negation, K |≠ ¬F, so ¬F is unsatisfiable. Conversely, suppose that ¬F is unsatisfiable. For any interpretation K, K |≠ ¬F, so that K |= F by the semantics of negation. Thus, F is valid. Because of this duality between satisfiability and validity, we are free to focus on either one or the other in the evaluation of a logical expression, depending on which is more convenient for the particular case.

As for limitations, formulas or invariants proposed can not be directly used by any available SAT tools. Although the aforementioned invariants have been used in our PLMV&V tool, its syntax is not enough independent of implementation as for being used by some SAT solver standard. This limitation will be treated in future research.

5.5 Conclusion

In the present chapter, we suggested a specific approach for the verification of product line models. Benefits of this approach to PLM verification are (i) its foundations in the well accepted requirements engineering framework, which allows the approach to be very general; (ii) this approach not only gathers the proposals of verification found in literature, but proposing another innovative rules and standardizes them through a same language and a same multi-model of verification; (iii) this is an approach not only focusing on single-system models (PLC models like the most of the literature do), but extended to the evaluation of PLMs.

Based on the general approach, we validate the approach making use of a case study and automate it through a computational tool called PLMV&V. These matters will be further explored in the next chapter, where we present our validation with a real industrial case and our tool implementation of the approach.

Part VI

Case Study and Tool Support

6 Case Study: Stago's Product Line Model

6.1 Introduction

Diagnostica Stago, Inc. is one of the most important providers of hemostasis products in the world. This society offers a set of hemostasis instrumentation and optimized reagent kits for research as well as for routine analysis. Diagnostica Stago, Inc. is a French industry with a staff close to 1500. Diagnostica Stago devotes its research and innovative skills to the development of increasingly effective medical diagnostic products and instrumentation.

Through Stago's product lines model we will explain the use of multi-process verification model described in Chapter V. On the aforementioned PLM, we will apply the logic invariants explained in Chapter V with the aim of identifying the different errors that the model may have. Because of reasons of space and simplicity, some details about implementation of each logical formula will not be showed. We will have to restrict our analysis in order to summarize how each logical expression allows the identification of errors in a PLM. Errors identified by an algorithm or by a logical expression are highlighted in red in the respective model. The aforementioned formulas will be applied by means of traversing algorithms on DAGs and tree automatons than have been implemented in PLMV&V, that is the computational tool for verification of product lines models described in the second part of this chapter.

6.2 Stago's Product Line Model

Figure 6-2-1 shows the Stago's product line model. We have introduced some typical errors in the model in order to detect them through applications of invariants proposed in our PLM and PLC verification method. The Stago's product line model modified is showed in figure 6 -2-2.

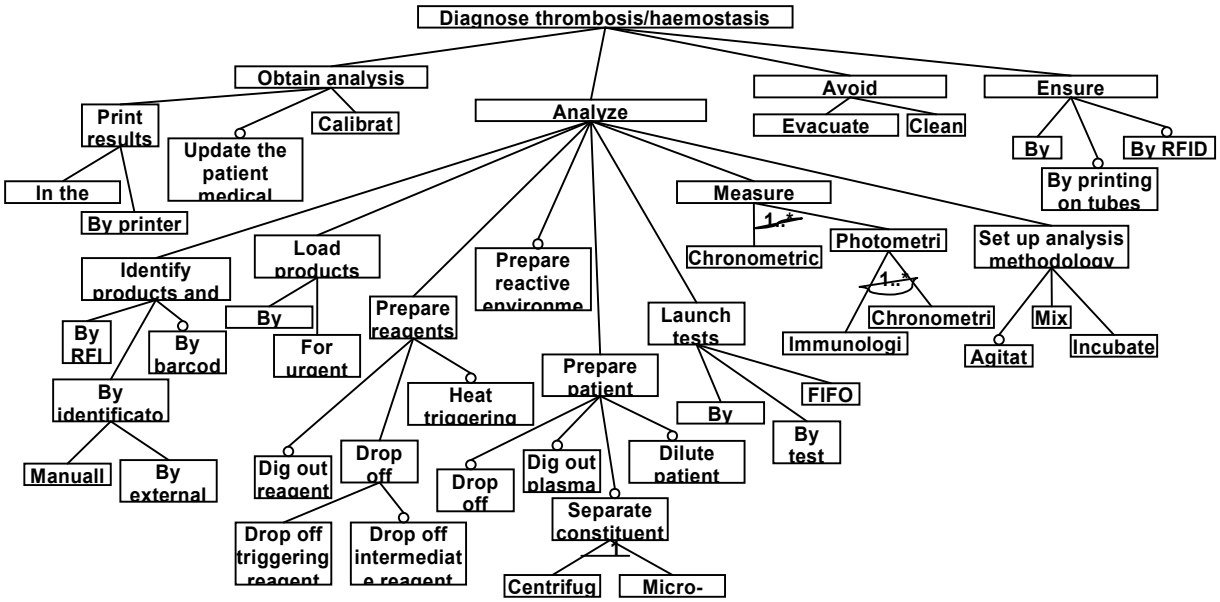


Figure 6-2-1: Stago's product line model

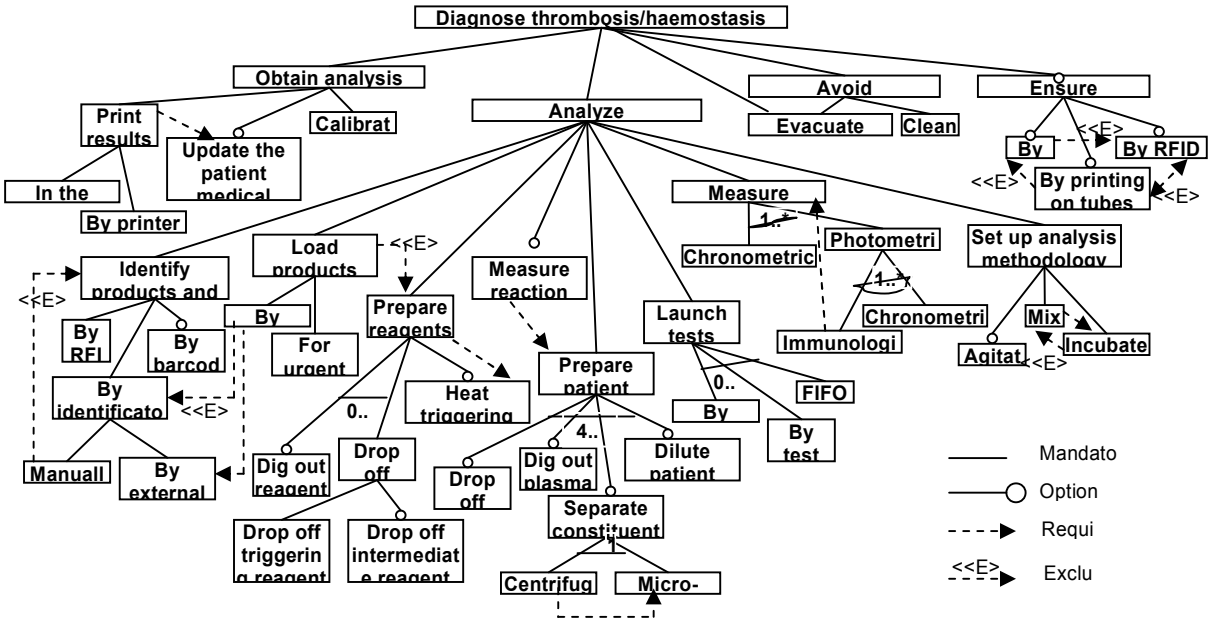


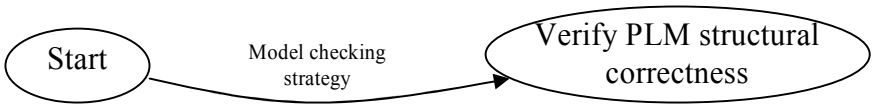
Figure 6-2-2: Stago's product line model modified

6.3 Stago's Product Line Model Verification

The types of errors identified in the PLM corresponding to Figure 6-2-2 and the invariant logical expressions to identified its, are presented next. This case study deals with model checking strategies to achieve three intentions proposed in our multi-method approach.

In next paragraphs, we will apply the verification process, particularly the intention "Verify PLM structural correctness", to PLM represented in Figure 6-2-2. That is, in a more graphic way, we will execute the following part of the model from the defined process in Figure 5 -2-1.

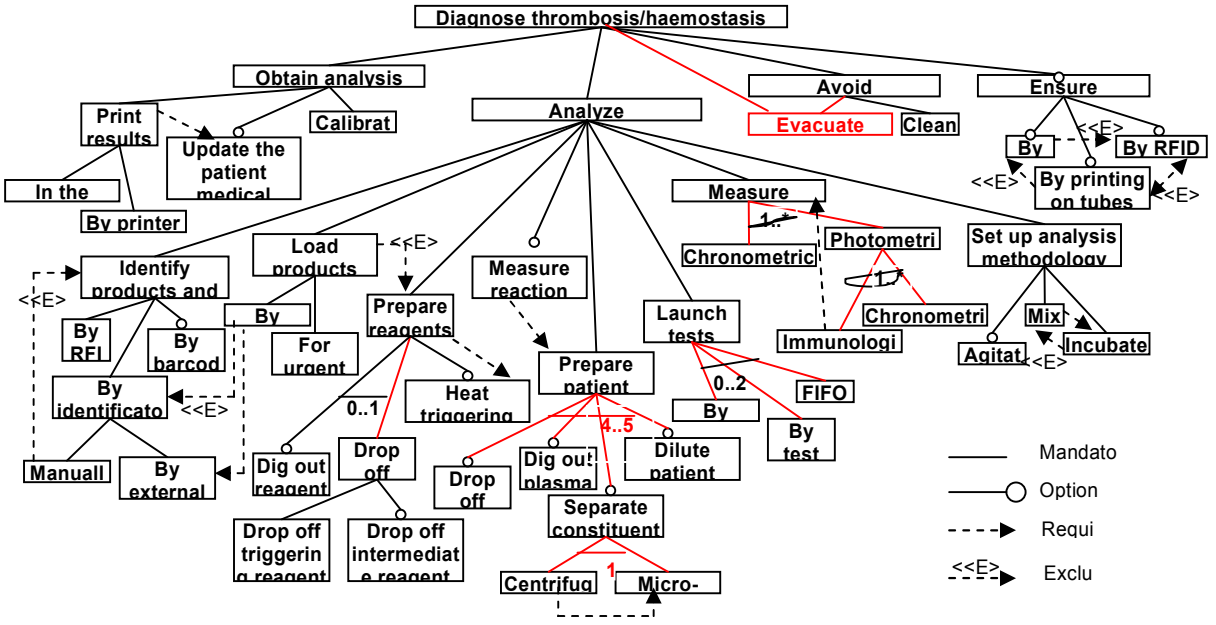
6.3.1 Section <*Start, verify PLM structural correctness, Model checking strategy*>



And then, we will apply in a systematic way the instructions of the direct sequency, selection and execution proposed for this section of the process model.

Execution directives of this MAP section are presented in blue.

6.3.1.1 < PLM, execute "Verify FORE Properties" invariants >



It is necessary to execute:

1. Root unicity invariant :

$$\exists feature \quad f \in PLM.\ f = root \wedge |f| = 1$$

With application of this invariant we have not found any problem because this PLM have only one root element.

2. Child – Father unicity invariant:

$$\forall(feature\ P_i, feature\ C) \in PLM.\ FatherFeature(P_i) \wedge ChildFeature(C)$$
$$\wedge (C\ childOf\ P_i) \wedge ((C\ Mandatory\ P_i) \oplus (C\ optional\ P_i)) \Rightarrow \left|(C \bullet P_i) \oplus (C \circ P_i)\right| = 1$$

In this equation, we take each child feature, and for one of them, we evaluate if the number of variability relations between this characteristic "C" and a potential group of parents "Pi", a group of at least one element, it is equal to one.

If it is not equal to one, the logical expression is evaluated "False", and an error type "cycle" is identified between the characteristic C and its parents group Pi.

In the model, we can see the characteristic "Evacuate" is related with "Diagnose thrombosis/haemostasis", and at the same time it is related with "Avoid", it means

$$\left|(C \bullet P_i) \oplus (C \circ P_i)\right| = 2$$

3. Cardinality relation invariant:

$$\forall Cardinality\ D_a(m,n), (FatherFeature(P), ChildFeature\{C_1,...C_k\})_a\ \Big| =$$
$$\left\{\underbrace{C_i,...,C_j}_{m}\right\} \vee \left\{\underbrace{C_i,...,C_{j+1}}_{m+1}\right\} \vee ... \vee \left\{\underbrace{C_i,...,C_{k-1}}_{n-1}\right\} \vee \left\{\underbrace{C_i,...,C_k}_{n}\right\},$$
$$m \in \mathrm{N} \cup \{0\} \wedge n \in \mathrm{N} \cup \{*\} \wedge 0 \le m \wedge (m \le i \le n = * \le k)$$

This invariant proposes that features group {C1,…,Ck} that make part of a cardinality D (m, n), must be children of the same father 'P'. The set of features grouped in cardinality relations must be comprised among cardinality values ('m' is the inferior value and 'n' the superior value). The value of 'm' must belong to set of natural numbers joined with '0' and the value of 'n' must belong to set of natural numbers joined with the symbol '*'. Besides, value of 'n' must not be inferior than value of 'm', neither superior to number features ('k') than intervene in cardinality relation 'Da'.

In the previous model, i = 1 and j = 4, they correspond to the counters of each relation grouped by the cardinality, in which m = 4 and n = 5. In this logical expression it is not accomplished that 'n' should be lower than 'j' (n < j), therefo re the expression is evaluated FALSE.

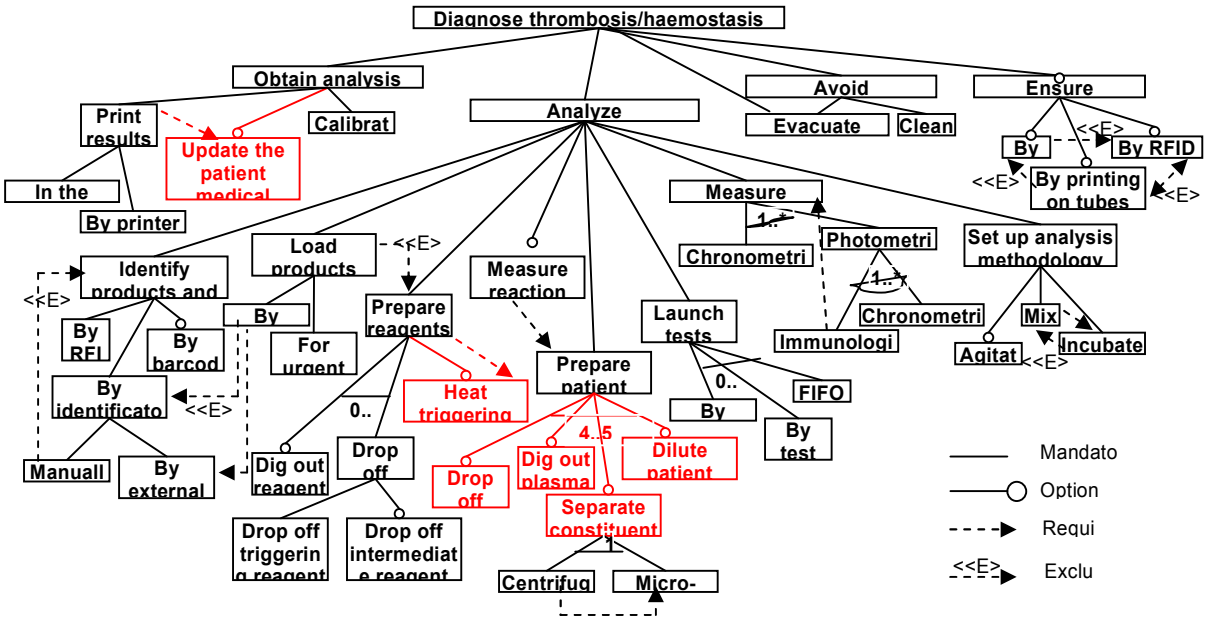
Cardinality must have two values ('m' and 'n'). In the previous PLM it is a cardinality relation with only one value. This logical expression permits 'm' be equal to 'n', therefore the correct way to write the cardinality that presents a problem, would be [1,1].

4. Invariant evaluating optionality of relations intervening in a cardinality:

$$\forall Cardinality\ C(m,n),(FatherFeature(P),ChildFeature\{C_1,...C_k\}) \in C \ \Big| = Optional(C_1),...,Optional(C_k), m \in Z \wedge n \in Z \cup \{*\} \wedge 0 \le m \wedge (m \le n = * \le k)$$

All relations joined by a cardinality relation must be of type "Optional". In the previous PLM we highlight in red all relations that are not of type "Optional" and are members of a cardinality relationship. At each case, invariant evaluating optionality of relations intervening in a cardinality is evaluated FALSE.

6.3.1.2 <PLM, execute "Find contradictions in optional relations" invariants>



These errors are detected executing optionally relationship constraint invariant:

$$\exists\{(feature\ P, feature\ C)_i, feature\ K\} \subseteq PLM.\ FatherFeature(P) \wedge ChildFeature(C) \wedge (C\ childOf\ P) \wedge (C\ Optionally\ P) \ \Big| = \vee_i (C \circ P) \wedge \neg(K \to C)$$

And invariant to validate if a cardinality is right or not:

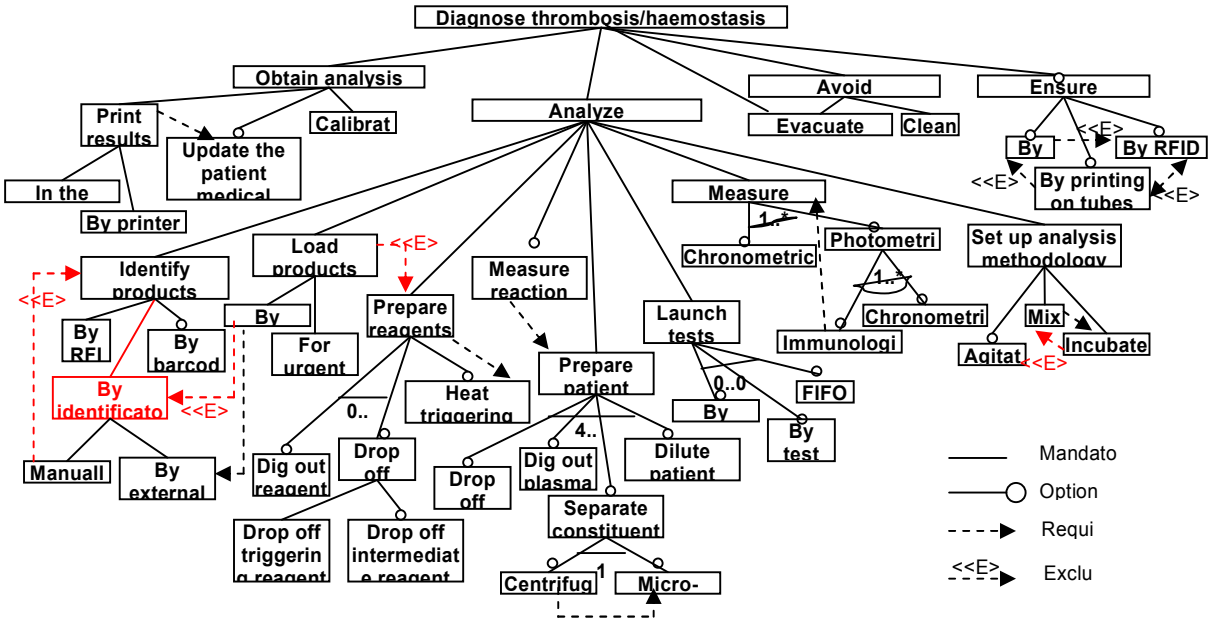
$$\exists\{parentFeature(f_a), featureSet(childFeature(f_i,...,f_j)), cardinality(m..n)\} \subseteq PLM$$
$$m < n,\ \ m,n \in [i,j] \wedge (\oplus_{x=1}^{n-m}(\wedge_1^{m+x-1}(f_i \wedge \neg\{f_{i+1}...,f_j\} \vee f_k \wedge \neg\{f_i,...,f_{k-1},f_{k+1},...,f_j\} \vee f_j \wedge \neg\{f_i,...,f_{j-1}\})) \wedge (count_{z=i}^{j}(f_z optional) \le n < j) \ \Big| = true$$

The first equation pretends to identify a couple of features ‘P’ and ‘C’ connected through an optional relation, where ‘P’ is the father feature and the ‘C’ is the child. Next, we identified a feature ‘K’ among the set of PLM’s features, in such a way that ‘K’ is connected with ‘C’ by means of a required relationship. If ‘K’ exists, the logical expression is evaluated FALSE. This equation is evaluated FALSE in the two cases in red (model immediately above), because the equation’s right side does not come true. In one of those cases, feature “Update the medical patient” (‘C’) is an optional child of “Obtain Analysis” (‘P’) and exists a feature “Print results” (‘K’) that is connected with “Update the patient medical” (‘C’) by means of a required relation.

With the second equation, a third error of the model is detected, just like the two previous ones concerning the contradictions in optional relations. In this logical expression, the idea is to search for each group of features joined by cardinality relations (f i,,…,fj) and sharing the same father (fa). Also, it is evaluated that cardinality limits are right. In our example, we see that i = 1 and j = 4, corresponding to each counters of relations grouped by the cardinality relation, besides m = 4 and n = 5. Logical expression evaluate that n should be lower than j, that is n < j, therefore the expression is evaluated FALSE.

Notice than in the logical expression to evaluate cardinality relations we do not take into account mandatory relations, because FORE notation requires that all features intervening in cardinality relations must be of an optional type.

6.3.1.3 <PLM, execute "Find contradictions in mandatory relations" invariants>

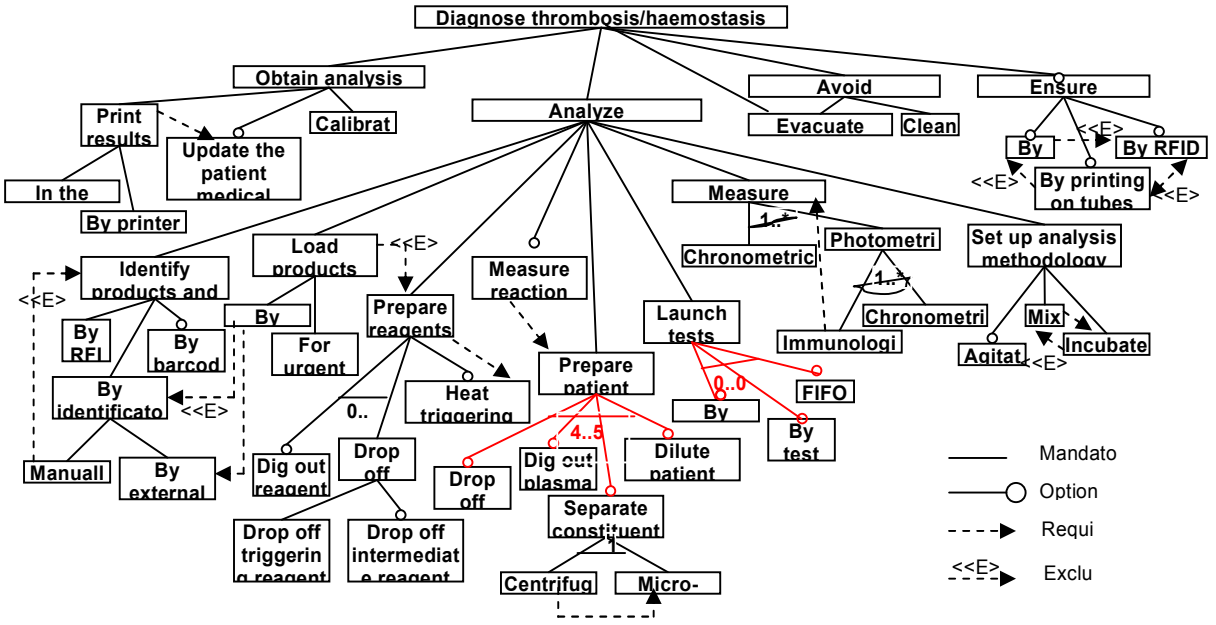


These errors can be detected by executing mandatory relationship constraint invariant:

$$\exists\{(feature\ P, feature\ C)_i, feature\ K\} \subseteq PLM.\ FatherFeature(P) \wedge ChildFeature(C)$$
$$\wedge (C\ childOf\ P) \wedge (C\ Mandatory\ P) \mid = \wedge_i (C \bullet P) \wedge \neg (K \leftrightarrow C)$$

With this equation it is possible to detect errors in mandatory relations. Features like "Identify products", "By identificator", "Prepare reagents" and "Mix" are being excluded by other features that could be mandatory, or not in the model. In the first logical expression, we search pairs of features joined by a mandatory relation from 'P' to 'C' and for each found pair, we search for one exclusion relationship from any feature of the model ('K') towards 'C'. If the feature 'K' exists, then second part of the logical expression is evaluated FALSE.

6.3.1.4 <PLM, execute "Find unbound or mal formed cardinalities" invariants>



This error can be identified by executing invariant to validate if a cardinality is right or not:

$$\exists\{parentFeature(f_a), featureSet(childFeature(f_i,...,f_j)), cardinality(m..n)\} \subseteq PLM$$

$$m < n,\ \ m,n \in [i,j] \wedge (\oplus_{x=1}^{n-m}(\wedge_{1}^{m+x-1}(f_i \wedge \neg\{f_{i+1}...,f_j\} \vee f_k \wedge \neg\{f_i,...,f_{k-1},f_{k+1},...,f_j\}$$

$$\vee f_j \wedge \neg\{f_i,...,f_{j-1}\})) \wedge (count_{z=i}^{j}(f_z optionaly) \leq n < j) \ \ | = true$$

And invariant to evaluate cardinalities whit value 0,0:

$$\forall\{parentFeature(f_a), featureSet(childFeature(f_i,...,f_j)), cardinality(m..n)\} \subseteq PLM$$

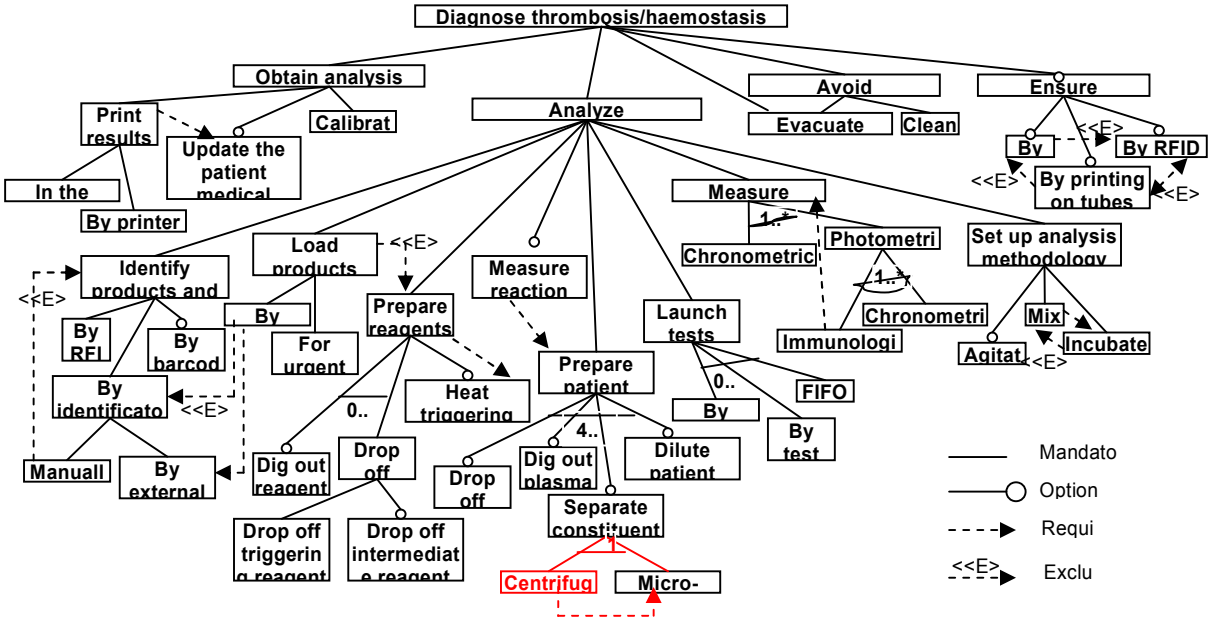
$$(f_i\ childOf\ f_a),...,(f_j\ childOf\ f_a) \ \ | \neq$$

$$(m = n = 0) \vee (n < m) \vee (m < 0) \vee (\neg f_i \wedge ... \wedge \neg f_j \wedge (m \neq 0)) \ \ | = false$$

With first equation it is possible to detect errors on the model concerning the overflowing of superior value in cardinality relationship. In the model, we have i=1 and j=4, that correspond to counters of the whole relations grouped by cardinality, and also m=4 and n=5. Logical expression evaluation does not fulfil n must be lower than j, that is n < j, therefore entire logical expression is evaluated FALSE.

With second logical expression, it is possible to evaluate cardinality's relations and to recognize erroneous cardinalities like those where both boundaries have value 0 (0 ,0) or those where the superior limit is lower than the inferior one or those where inferior limit is a negative number, or those where any feature is selected when inferior limit is different to zero. In our case study, second logical expression has been useful in order to detect the cardinality's error (0,0).

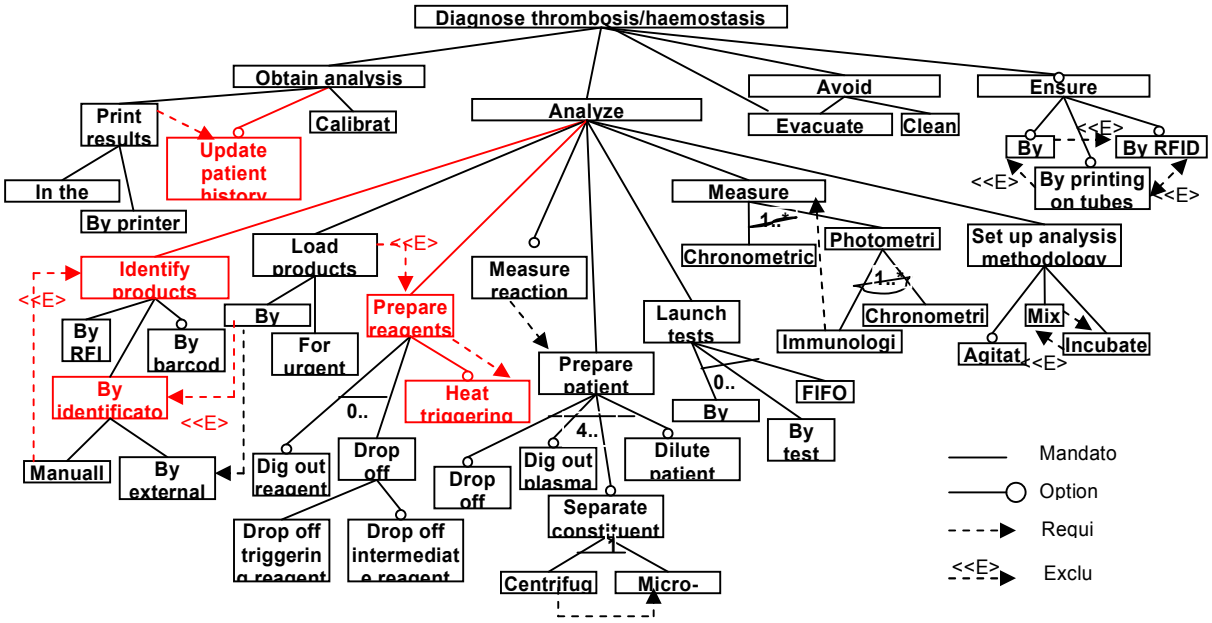
6.3.1.5 <PLM, execute "Find contradictions between transversal dependencies and cardinalities" invariants>



We have not developed yet an invariant to deal with contradictions between transversal dependency and cardinality.

In this case, it is necessary an algorithm considering interactions between transverse dependencies and cardinalities.

6.3.1.6 <PLM, execute "Find contradictions between transversal dependencies and variant dependencies" invariant>



These errors can be detected by intersecting include and exclude invariants. In fourth errors stressed on the model a feature child that can be "optional" or "mandatory" is at the same time required and excluded, or optional and required. Interpreting this result we can deduce that it is an error.

Require-child invariant:

$$\exists features\ (fm_i, fm_k)_{i \neq k} \in PLM, fm_k \neg childOf\ fm_i \ \Big| = fm_i \rightarrow fm_k$$

Exclude definition and invariant:

$$\forall features(fm_i, fm_k) \in PLM.\ exclude(fm_i, fm_k) \Rightarrow (fm_i \rightarrow \neg fm_k)$$

$$\forall features(f_i, f_j, f_k)_{i \neq j} \in PLM.\ (f_j\ childOf\ f_i) \wedge (f_j\ mandatory\ f_i) \ \Big| = \neg\exists(f_k \leftrightarrow f_j)$$

Require definition and invariant:

$$\forall features(fm_i, fm_k) \in PLM.\ require(fm_i, fm_k) \Rightarrow (fm_i \rightarrow fm_k)$$

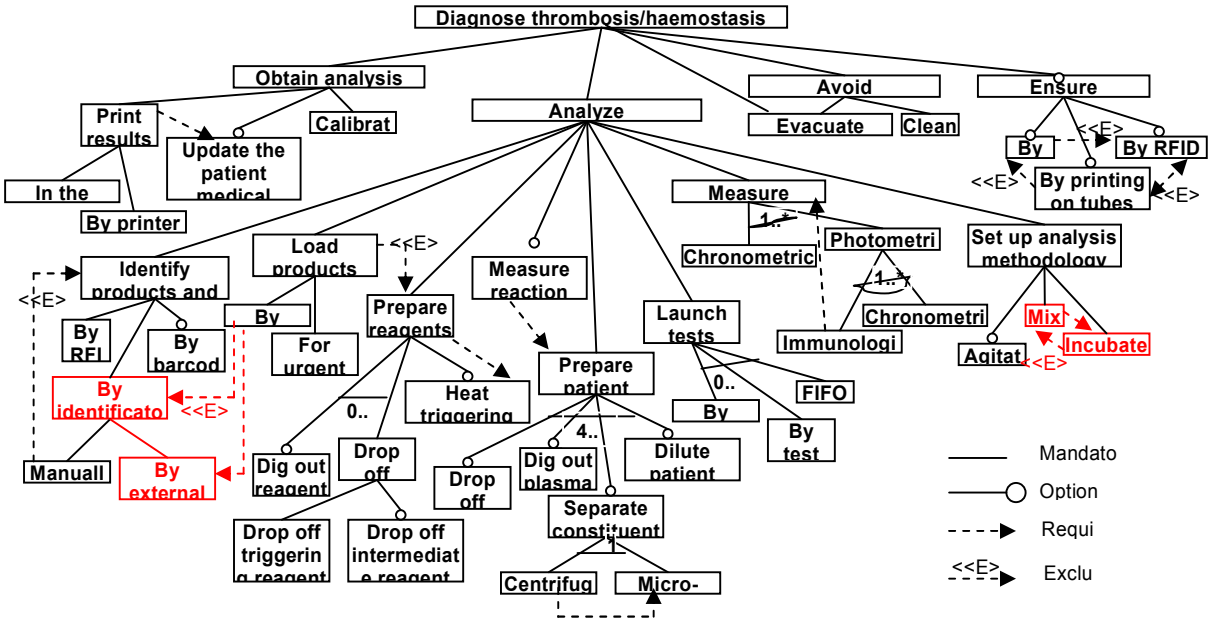
$$\forall features(f_i, f_j, f_k)_{i \neq j} \in PLM.\ (f_j\ childOf\ f_i) \wedge (f_j\ optionally\ f_i) \ \Big| = \neg\exists(f_k \rightarrow f_j)$$

With the first logical expression (require-child invariant), error existing among features "prepare reagents" and "heat triggering" can be detected. The aforementioned features are joined through a transversal relationship, respecting right side restrictions of the logical expression. At the same time, they are joined by means of a "childOf" relationship, and not respecting restrictions of the left side expression. For this reason, the entire expression is evaluated FALSE.

With the logical expression "exclude invariant" it is possible to detect three cases of error, each one related with features "identify products", "by identificator" and "prepare reagements". Since the previous features are all joined with their respective parents through a mandatory relationship, and at the same time each one is being excluded by a third feature, the expression is evaluated FALSE for each of three cases.

The error highlighted in the feature "update patient history" is detected with the logical expression "require invariant" because "update patient history" (fj) and "obtain analysis" (fi) are joined through an optional relationship and, on the same model it exists a relation "require" among features "print result" (fk) and "update patient history" (fj). Given that it exists (fk "require" fj), the logical expression is evaluated FALSE and the error is put in evidence.

6.3.1.7 <PLM, execute "Find contradictions between transversal dependencies" invariants>



These errors can be detected by require-exclude contradiction invariant. In a PLM it is not possible that a feature (k) can require another feature (j) or one of their children and at the same time, this feature k excludes feature j.

Require-exclude definition:

$$\exists features\ (fm_i, fm_k)_{i \neq k} \in PLM, fm_k \neg childOf\ fm_i \ \Big| = fm_i \rightarrow fm_k$$

and contradiction invariant:

$$\forall features(fm_i, fm_k) \in PLM.\ require(fm_i, fm_k) \Rightarrow (fm_i \rightarrow fm_k)$$

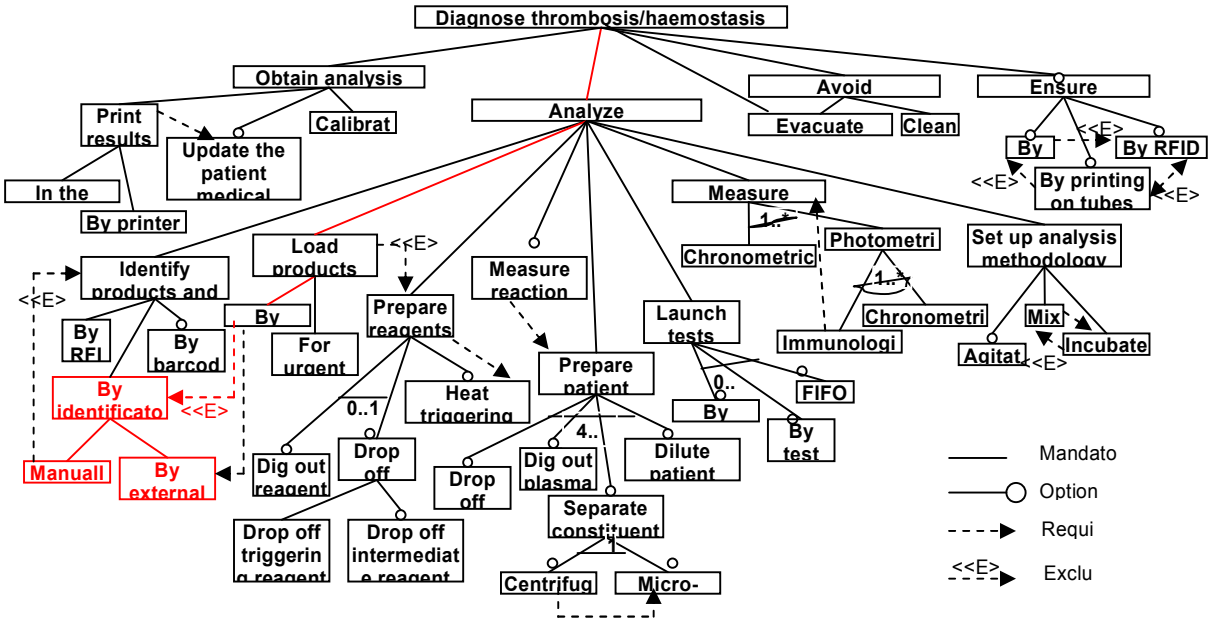
$$\forall features(fm_i, fm_k) \in PLM.\ exclude(fm_i, fm_k) \Rightarrow (fm_i \rightarrow \neg fm_k)$$

$$\forall features(f_j, f_k)_{j \neq k} \in PLM \wedge (f_k\ requires\ f_j) \ \Big| =$$

$$\neg\exists((f_k \leftrightarrow f_j) \vee (f_j \leftrightarrow f_k) \vee (f_k \leftrightarrow ancestorOf(f_j)))$$

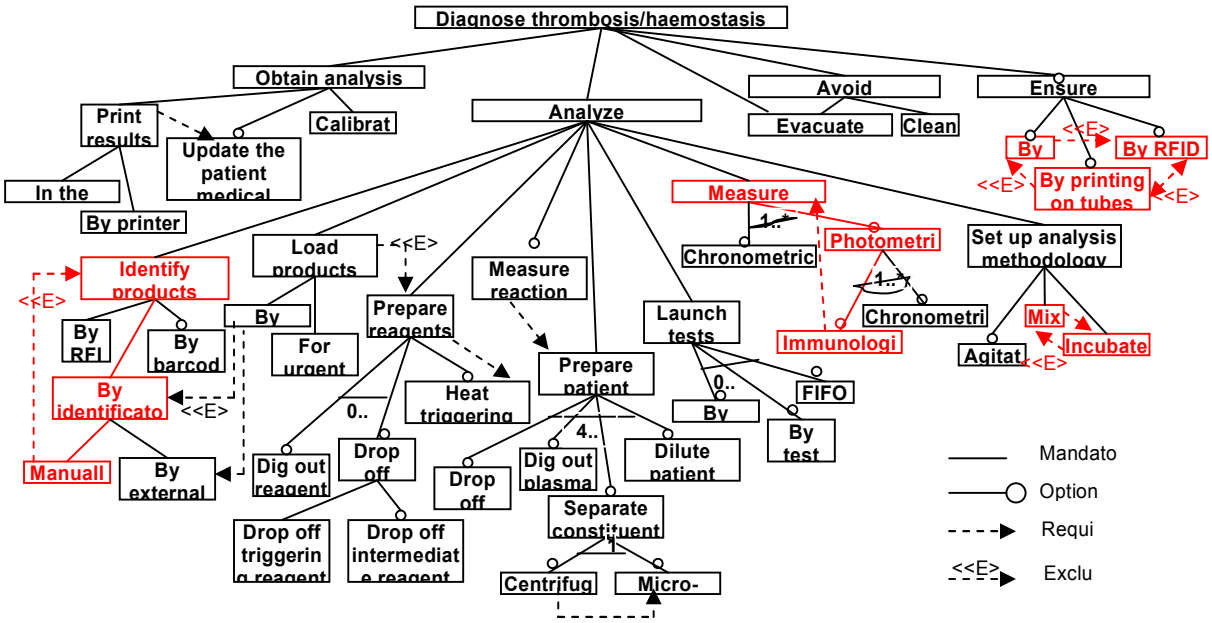
Two errors highlighted in red can be identified by the previous logical expression. In the first case, feature "by" (fk) requires feature "by external" (fj) and at the same time, fk excludes feature "by identificator", which is fj's ancestor. In the second case, feature "min" (fk) requires feature "incubate" (fj) and at the same time, fj excludes fk. In both cases, the logical left side expression is evaluated TRUE and the right side one FALSE, this implies that complete formulae must be evaluated FALSE.

6.3.1.8 <PLM, execute “Find features impossible to achieve” invariants>



An error of impossible features to achieve is not yet automatically identified with an i nvariant. It will be considered in the future work.

6.3.1.9 <PLM, execute “Find cyclic relations” invariants>



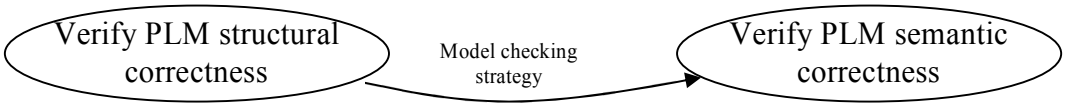
These errors can be detected executing next acyclic invariant:

$$\forall feature\ f \in PLM, \neg\exists\{f_1,..,f_k\} \subseteq PLM.\ TransversalDependency\ or\ VariantDependency$$
$$(f_1, f_k)...TransversalDependency\ or\ VariantDependency(f_k, f_1)\ | = true$$

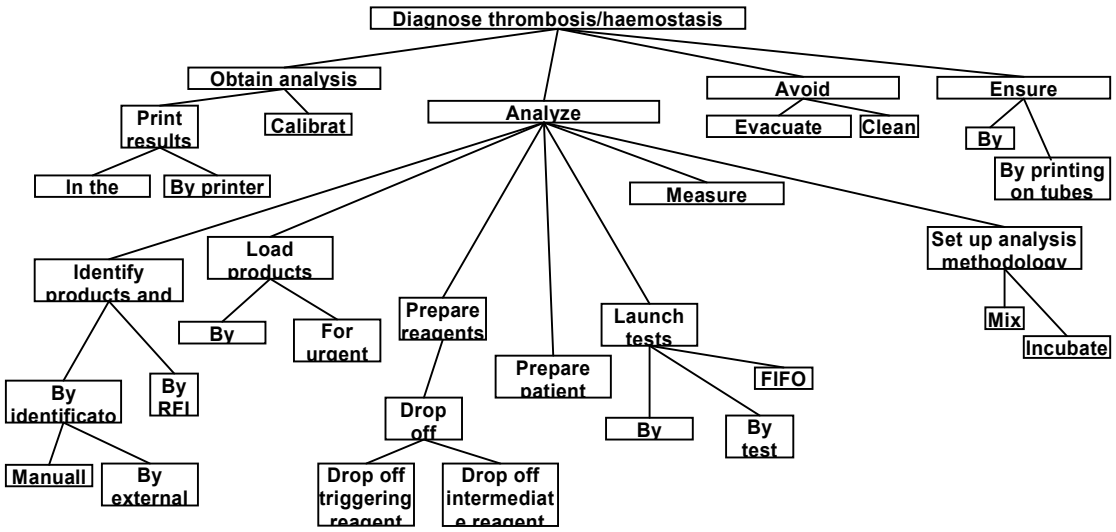
The cycles in red have the particularity to contain at least, one transverse relation (require, exclude). With the previous logic invariant, we look for cycles of features joined through variant or transversal relations. If at least one cycle is detected, right-hand side of the logical expression is evaluated FALSE and consequently, all the logical expression is evaluated FALSE, evidencing the existence of one cycle. One example of cycle detected by the aforementioned invariant is: "Manual" *childOf* "By identificator" *childOf* "Identify products" *excluded_by* "Manual".

Next, we will apply the verification process, particularly the intention "Verify PLM semantic correctness", to PLM represented in Figure 6-2-1. So, in a graphic way, we will execute the following section of the process model defined in the figure 5-2-1.

6.3.2 Section <*Start, verify PLM structural correctness, Model checking strategy*>



6.3.2.1 <(PLM, PLC), execute usability, liveliness or decidability invariant>



It is necessary to execute:

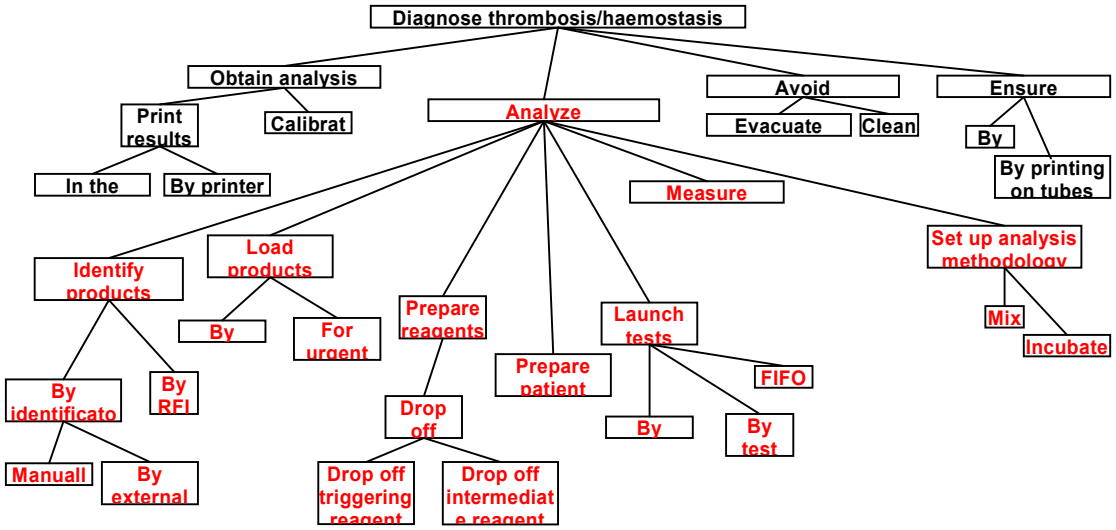
Liveliness, usability or decidability invariant:

$\forall$ Feature f, (VariantDependency, TransverseDependency)$C \subseteq PLM$,

$\exists\ I_k \in \{PLC_1,...,PLC_i\}\ (1 \le k \le i), I_k| = (\cap_{i=1..n}\ C_i) \cap f$

Respecting the set of model's constraints, all features of the PLM described in figure 5-2-1 could be used in order to derivate previous PLC model.

6.3.2.2 <(PLM, PLC), execute suitability or utility invariant>



It is necessary to execute:

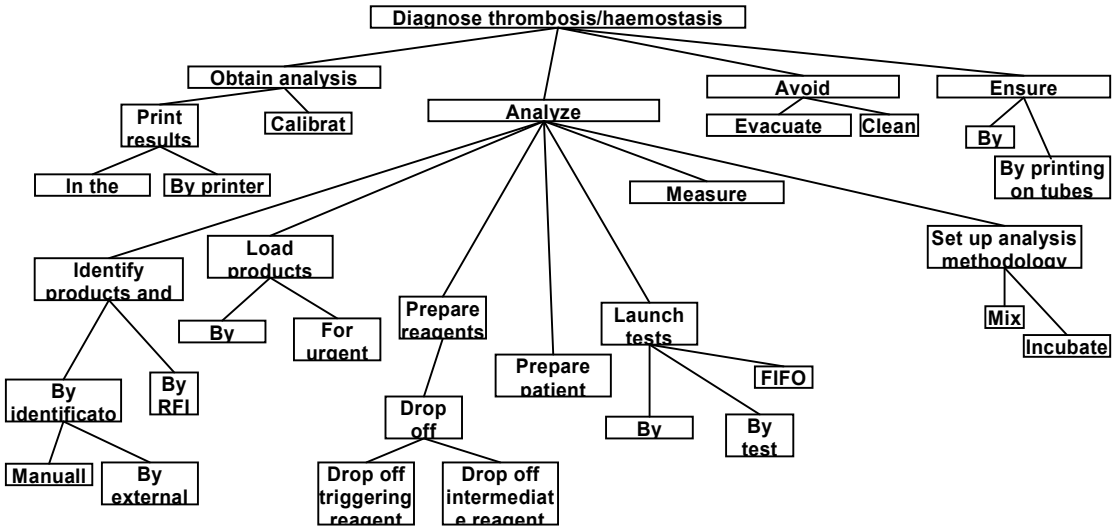
Utility or suitability invariant

$\forall$ Feature f, (VariantDependency, TransverseDependency)$C \subseteq PLM$,

$\exists\ I_k \in \{PLC_1,...,PLC_i\}\ (1 \le k \le i), I_k| = (\cap_{i=1..n}\ C_i) \cap (\neg f)$

Previous PLC model is an example of configuration in which most of the optional features have not been selected from the PLM. It is a problem if we remove some mandatory features from the PLM, because there might be other features depending of the feature eliminated. For instance, if we want to eliminate feature Analyse only, it is not possible to do that because features like Identify products, Load products, Prepare reagents, Prepare patient, Launch tests, Measure, Set up analysis methodology and all their children must disappear as well from the model. We believe that it is necessary to develop a special mechanism to deal with this issue.

6.3.2.3 <(PLM, PLC), execute validity or richness algorithm>



It is necessary to execute:

An algorithm that allows to evaluate:

$$PLM_k \models \text{Constraint Set} \wedge$$
$$\exists PLC \models \text{Constraint Set} \in PLM_k$$

And in particular :

$$\forall \text{ Constraint } C_i \in PLM, \bigwedge_{i=1}^{n} C_i \neq \bot$$

Previous PLC model is a proof that there is a particular configuration that satisfied all PLM set of constraints.

6.3.2.4 <PLM, execute verifiability algorithm>

It is necessary to execute:

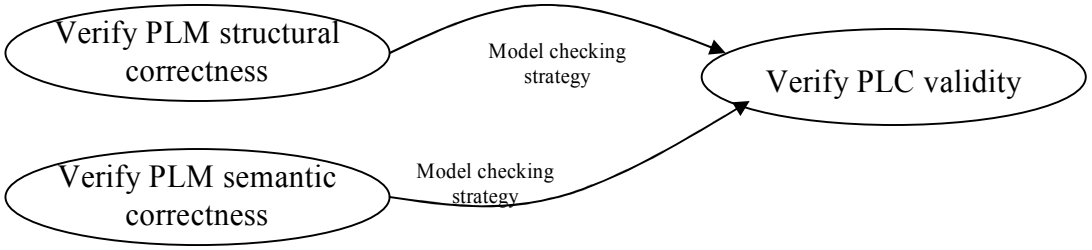
An algorithm that permits to evaluate:

$$\forall \log icalExpression \; G = SubG_i, SubG_j . \; SubG_i \wedge SubG_j \models true$$

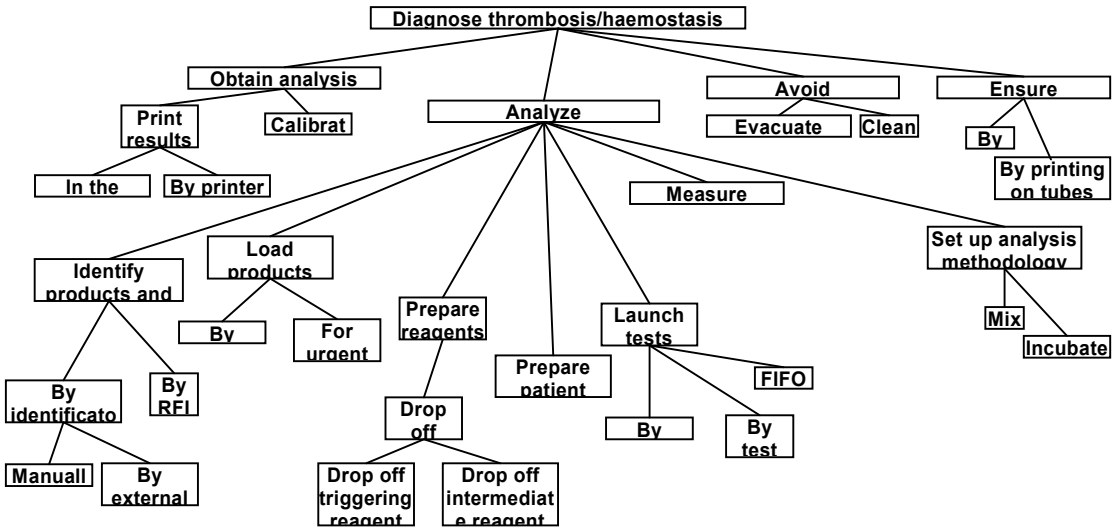
In order to apply previous formulae, we need to divide the product line model into an ordered set of sub sub-models, and to make the same thing with each sub-model and so on, even arriving at the level of leaves. Due to space reasons, this procedure is not going to be applied to the model. Our PLM example is evaluated TRUE after being divided into sub-models and being evaluated of a recursive manner.

Afterwards, we will apply the verification process, particularly the intention "Verify PLC validity", to a PLC model derived from PLM represented in Figure 6-2-1. It means, in a graphic way, we will execute the following two sections of the process model defined in the figure 5 -2-1.

6.3.3 Section <*verify PLM structural correctness, verify PLC validity, model checking strategy*> & Section <*verify PLM semantic correctness, verify PLC validity, model checking strategy*>



6.3.3.1 <(PLM, PLC), execute tree invariants>



It is necessary to execute:

1. Root unicity invariant

$$\forall PLC, \exists feature \quad f \in (PLM \cap PLC).\ f = root \wedge |f| = 1$$

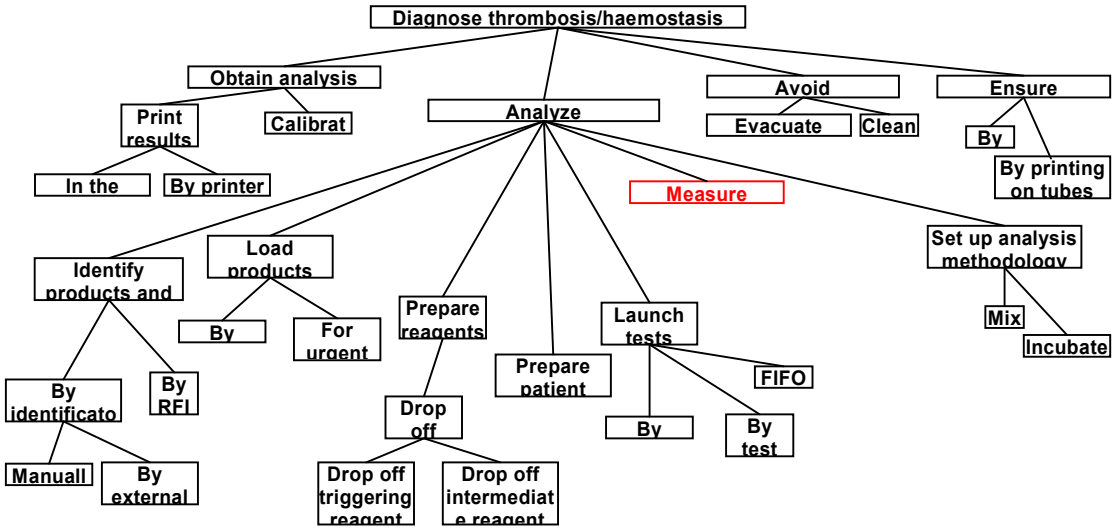
This invariant is evaluated TRUE because previous PLC model has only one root element.

2. Child-parent unicity invariant

$$\forall(feature\ P_i, feature\ C) \in PLM \cap PLC.\ FatherFeature(P_i) \wedge ChildFeature(C)$$
$$\wedge (C\ childOf\ P_i) \wedge ((C\ Mandatory\ P_i) \oplus (C\ Optionally\ P_i)) \Rightarrow \left|(C \bullet P_i) \oplus (C \circ P_i)\right| = 1$$

This invariant is evaluated TRUE because each feature in previous PLC model has only one parent element.

6.3.3.2 <(PLC, PLM), execute PLM satisfaction invariants>



It is necessary to execute:

1. Satisfaction of PLM's mandatory relationships invariant

$$\exists\{(feature\ P, feature\ C)_i, feature\ K\} \subseteq PLM \cap PLC.\ FatherFeature(P)$$
$$\wedge ChildFeature(C) \wedge (C\ childOf\ P) \wedge (C\ Mandatory\ P)\ \left| = \wedge_i (C \bullet P)\right)$$

This PLC model respects all mandatory relationships founded in Stago's PLM.

2. Satisfaction of PLM's optionally relationships invariant

$$\exists\{(feature\ P, feature\ C)_i, feature\ K\} \subseteq PLM \cap PLC.\ FatherFeature(P)$$
$$\wedge ChildFeature(C) \wedge (C\ childOf\ P) \wedge (C\ Optionally\ P)\ \left| = \vee_i (C \circ P)\right.$$

This PLC model includes some features declared as optional in Stago's PLM.

3. Satisfaction of PLM's require relationships invariant

$$\forall features\ (fc_i, fc_k) \in PLM, fc_i \in PLC.\ require(fc_i, fc_k) \Rightarrow (fc_i \in PLC \wedge fc_k \in PLC)$$

The Stago's PLM didn't have transverse relationships, therefore evaluation of this invariant on the intersection among our examples of PLM and PLC model is always TRUE.

4. Satisfaction of PLM's exclude relationships invariant

$$\forall features\ (fc_i, fc_k) \in PLM, fc_i \in PLC.\ exclude(fc_i, fc_k) \Rightarrow (fc_i \in PLC \wedge fc_k \notin PLC)$$

$$\forall features\ (fc_i, fc_j, fc_k) \in PLM, (fc_i, fc_j) \in PLC.$$

$$exclude(fc_i, fc_j) \wedge (fc_k\ childOf\ fc_j) \Rightarrow (fc_i \in PLC \wedge (fc_j, fc_k) \notin PLC)$$

The Stago's PLM didn't have transverse relationships, therefore evaluation of this invariant on the intersection among our examples of PLM and PLC model is always TRUE.

5. Algorithm to verify satisfiability of PLM's constraints

$$\forall i \in \text{invariantSet}, (\exists\ path\ P \in PLC \mid \neq i \wedge \exists (P, i) \subset PLM) \mid = \perp$$

For **invariant i**:

Search in **PLC** a valid **path P** from an initial state into a one that violates i.

If it exists a **PLC** with **P**, **PLM** is inconsistent

This algorithm detects three PLM's paths, one of which must be included in every PLC model, and no one is presented on our particular configuration model. First path is: "Diagnose thrombosis/haemostasis"--"Analyze"--"Measure"--''Chronometric". Second path is: "Diagnose thrombosis/haemostasis"--"Analyze"--"Measure"--"Photometri"--"Immunologi". Third path is: "Diagnose thrombosis/ haemostasis"--"Analyze"--"Measure"--"Photometri"--"Chronometri". Because the choice between "Chronometric" and "Photometri" is determined by a cardinality (1, *) and the choice between Immunology and "Chronometri" is also determined by a variability (1,*). No one of these paths is presented on our PLC example and an error is evidenced by the precedent algorithm.

6.4 Conclusion

The purpose of this chapter was to illustrate and validate the PLM correctness verification approach introduced in Chapter 5. We started with a presentation of the particular PLM used in this case study, followed by a execution of sections of our process Map presented in Figure 5-2-1, corresponding to model checking strategies. Model checking is one strategy that permits to evaluate logical expressions written in propositional logic and first order logic formalisms. With this example we thus showed that the approach is operational and can be applied to real cases.

7 Tool Support

7.1 Presentation

We have developed a computing tool in order to support the verification method of PLM and PLC, presented in chapter IV.

The name of the tool is PLMV&V, is an acronym of Product Line Model Verification and Valida tion.

Characteristics of the application:

In the construction process of PMLVyV tool, we have use d the following tools:

- Microsoft .NET Framework v2.0.50727 (free distribution by Microsoft at http://www.microsoft.com/downloads/Search.aspx?displaylang=en#)
- Microsoft Visual Studio 2005 Professional Edition
- XmlExplorer Controls V1.0.0.0 (free distribution by The Code Project http://www.codeproject.com)

7.2 Architecture

The project is composed of fourth DLL, distributed as is shown in Figure 7-1-1.

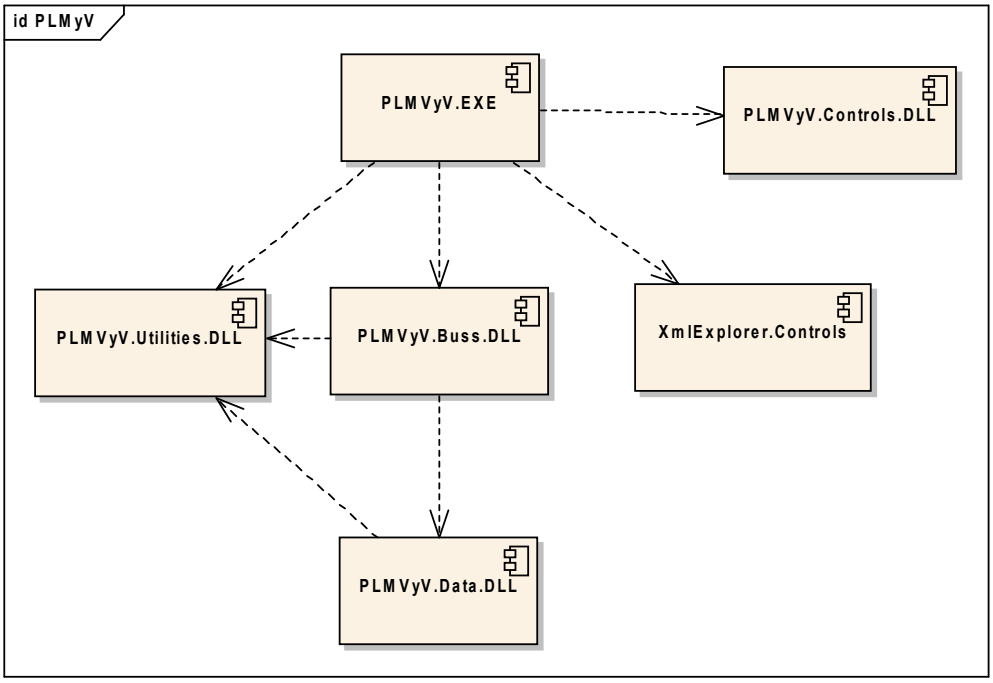


Figure 7-1-1: Architecture of the application PLMV&V.

Application PLMVyV.EXE

Description:

In this application, the user can do design and verification of product line models. It also provides interface facilities for design of PLMs through use cases and description by restriction formalisms. At present, these two formularies only work for writing of text, but we intend to improve its implementation in futures versions.

Figure 7-1-2 shows class diagram of the application PLMVyV.EXE

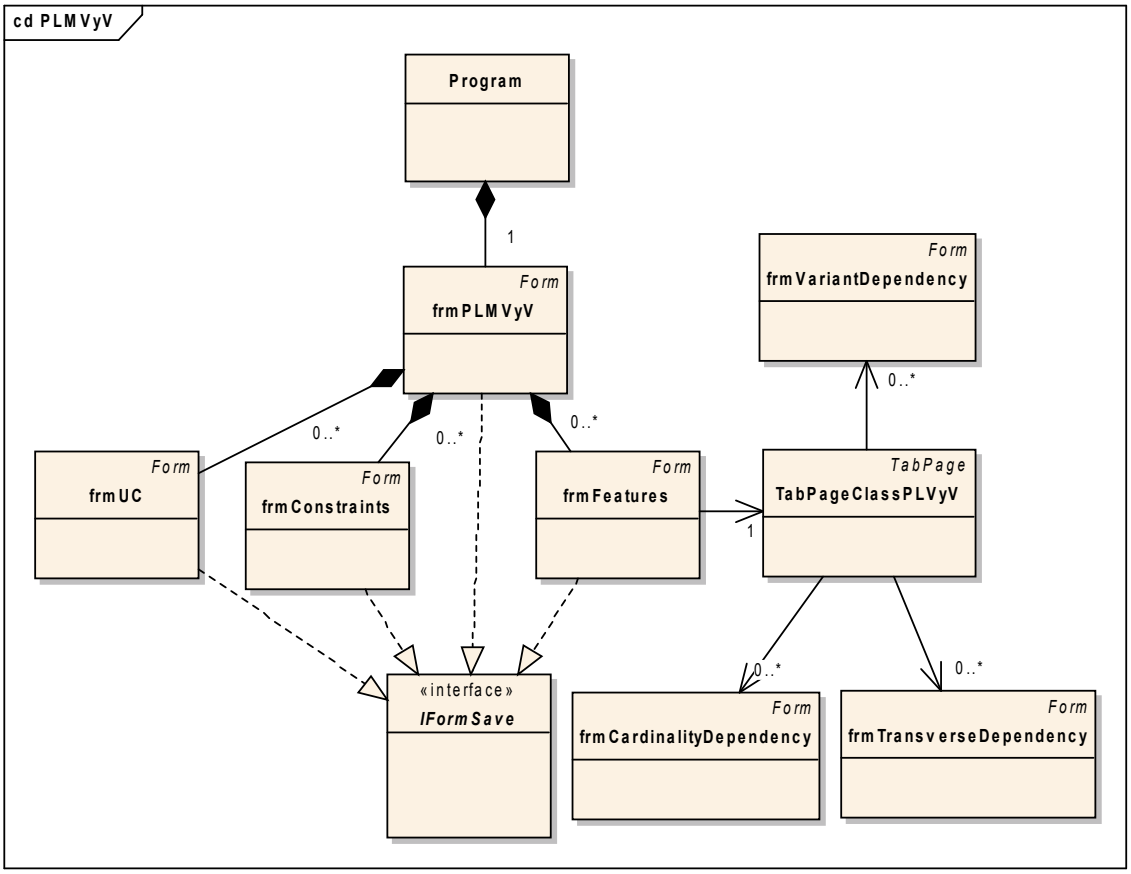


Figure 7-1-2: Class diagram of the application PLMVyV.EXE

In the class diagram, the Class Program corresponds to the main class of the application PLMVyV.EXE. This class is composed of one and only one frmPLMVyV class. frmPRMVyV classes correspond to the main MDI formulary. This class is composed of zero or several frmUC clases, each frmUC is a formulary that permits to design PLMs through use cases formalism. Also of zero or several frmConstraints classes, each frmConstraints permits to describe PLMs through textual restrictions. And it is also

composed of zero or several frmFeature clases, each class is responsible of the implementation of formularies for PLMs using features, particularly the FORE [ref] formalism. Other classes are TabPageClassPLMVyV that is in charged of panel tools which permit designing PLMs, frmVariantDependency that permits to manage variant dependencies from the model, frmTransverseDependency that permits to manage cardinality dependencies from the model and frmCardinalityDependency that permits to manage transversal dependencies of the model.

DLL: PLMVyV.Controls.DLL

Description:

In this DLL it is possible to find the controls develo ped for the project, they split into 4 groups, as follow:

Containers: in this group it is possible to find panels that can spread out and hide. These panels are employed to handle the properties of each PLM

Administrators of properties: in this group of co ntrols it is possible to find the controls permitting to edit the properties of features. NumericTextBox is a panel that permits to edit numeric elements. Some of its usages are to select the top and left of an element. SelectorValueProperty is another pan el showing a bottom that permit s to select colour and type of script for a particular PLM's feature.

Classes: are the instances of a particular class that can be located in the working space in order to list a group of properties. In this group, there are two types of controls, ClassPLVyV and FeaturePLVyV. ClassPLVyV has all the functionality to dimension and to move the controls. FeaturePLVyV is ClassPLVyV's specialization with all the functionalities of an element of the model.

Relations: these set of objects permit to do different relations on the model, like TransverseDependency, VariantDependency and Cardinality relations.

Figure 7-1-3 shows the class diagram of PLMVyV.Controls.DLL.

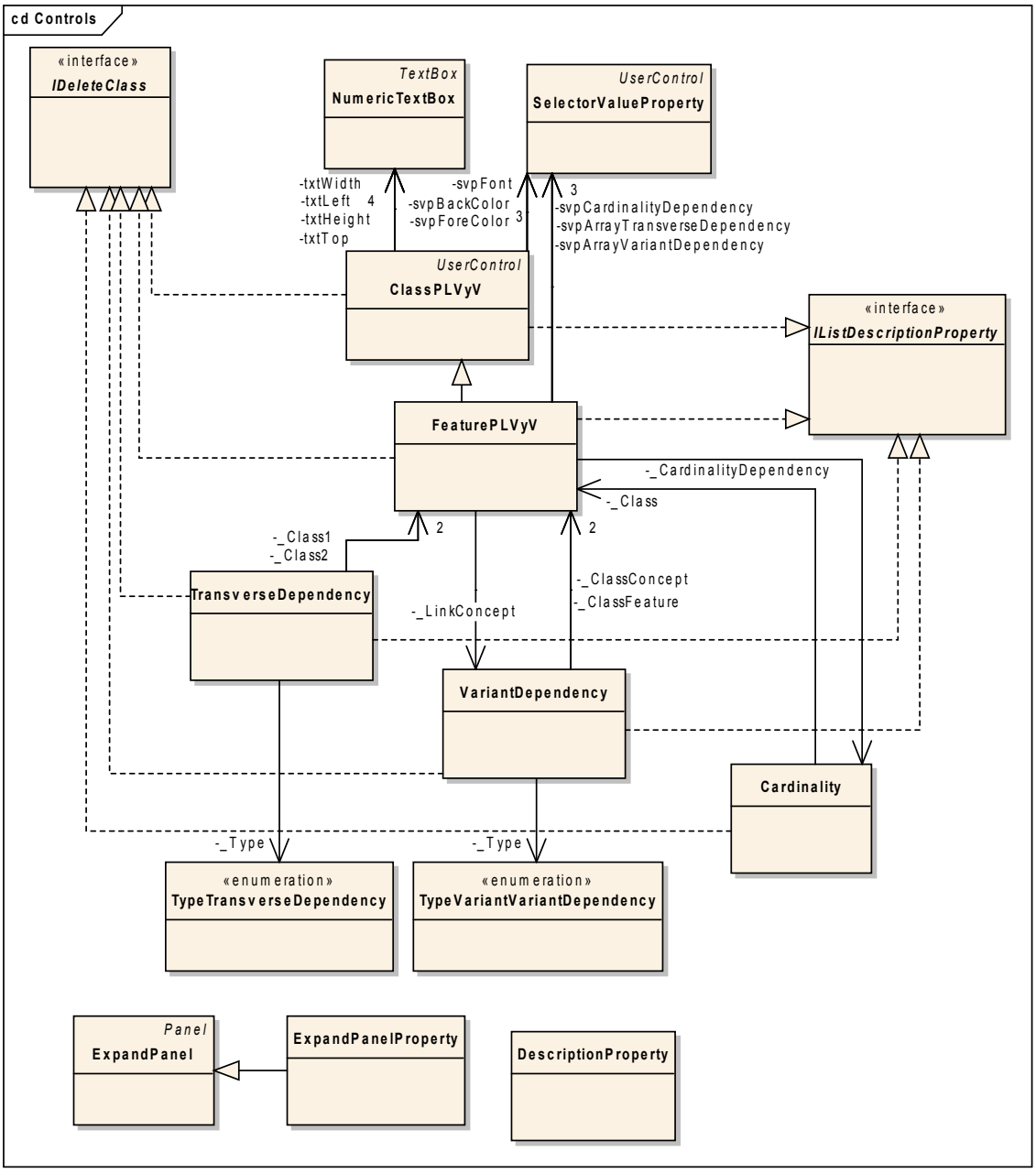


Figure 7-1-3: Class diagram of PLMVyV.Controls.DLL.

DLL: PLMVyV.Buss.DLL

Description:

Control all business rules of different types of models to implement (feature, use case and restriction models). Into its functionalities are: to verify availability of necessary files and objects corresponding to each model.

Figure 7-1-4 shows the class diagram of PLMVyV.Buss.DLL.

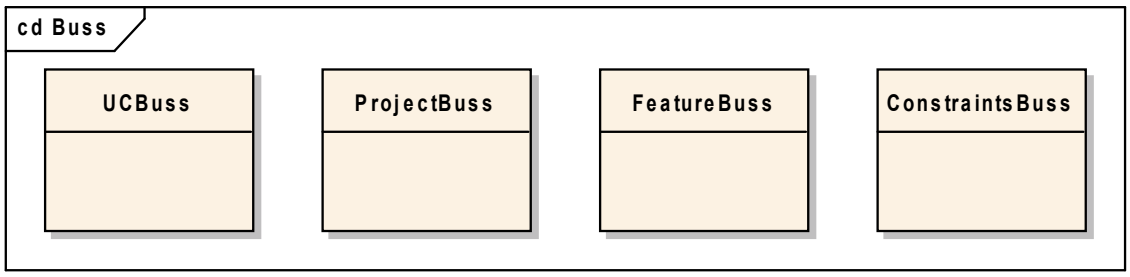


Figure 7-1-4: Class diagram of PLMVyV.Buss.DLL.

DLL: PLMVyV.Data.DLL

Description:

Control the storages in the different files that compose a project, the file of each model as well the contents of the aforementioned files.

Figure 7-1-5 shows the class diagram of PLMVyV.Data.DLL.

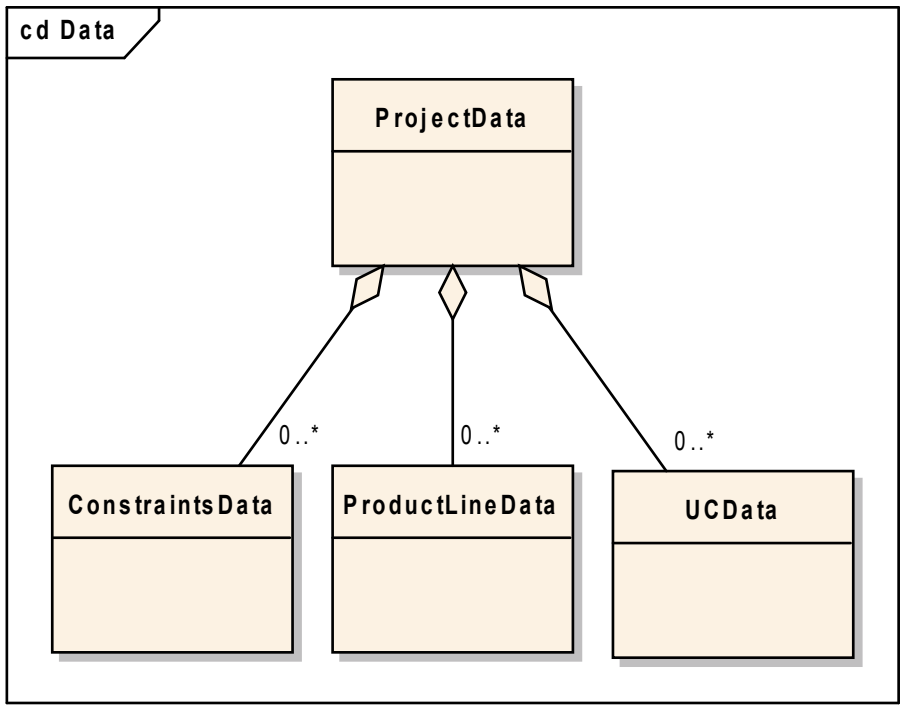


Figure 7-1-5: Class diagram of PLMVyV.Data.DLL

Persistence of Constraints Data and UC Data is making through text files, whereas Product Line Data is made through a file XML that must be in accordance with schema described in Figure 7-1-6.

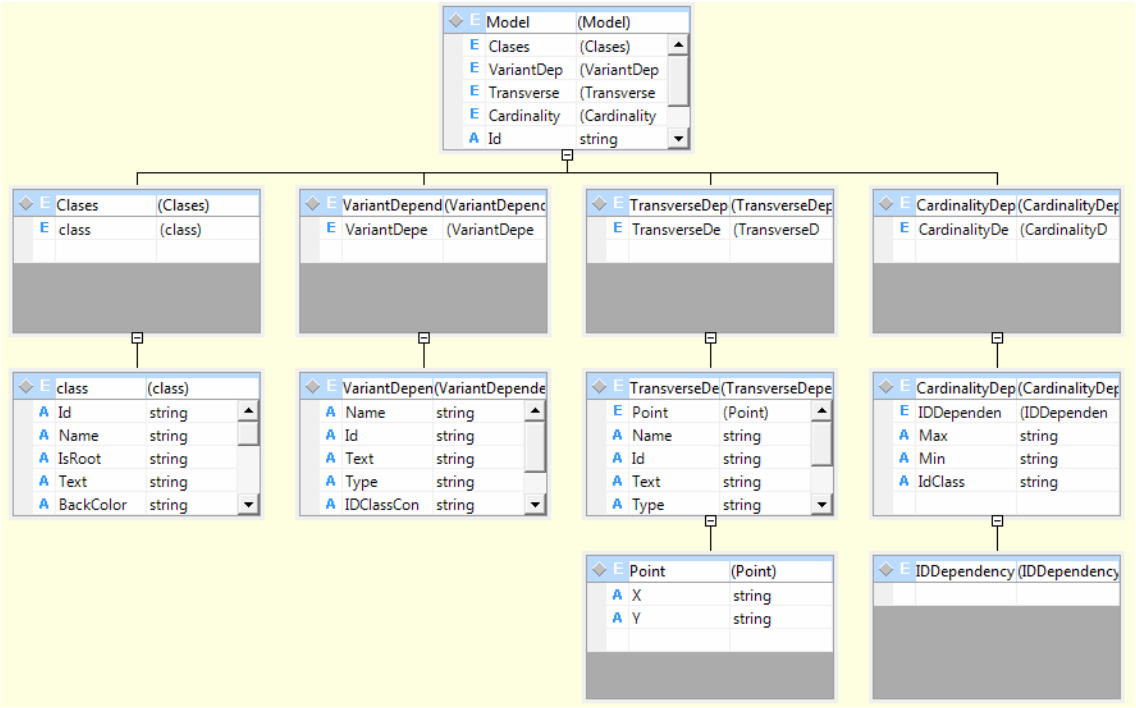


Figure 7-1-6: Schema for storage of Constraints Data and UC Data content.

DLL: PLMVyV.Utilities.DLL

Description:

Manage general functions of the project. Some of utilities are : administration of especial types of data, utilities to access disk, utilities for handling XML files, utilities for handling types of personalized exceptions and, in general, to manage all constants of the project.

Figure 7-1-7 shows the class diagram of PLMVyV.Utilities.DLL

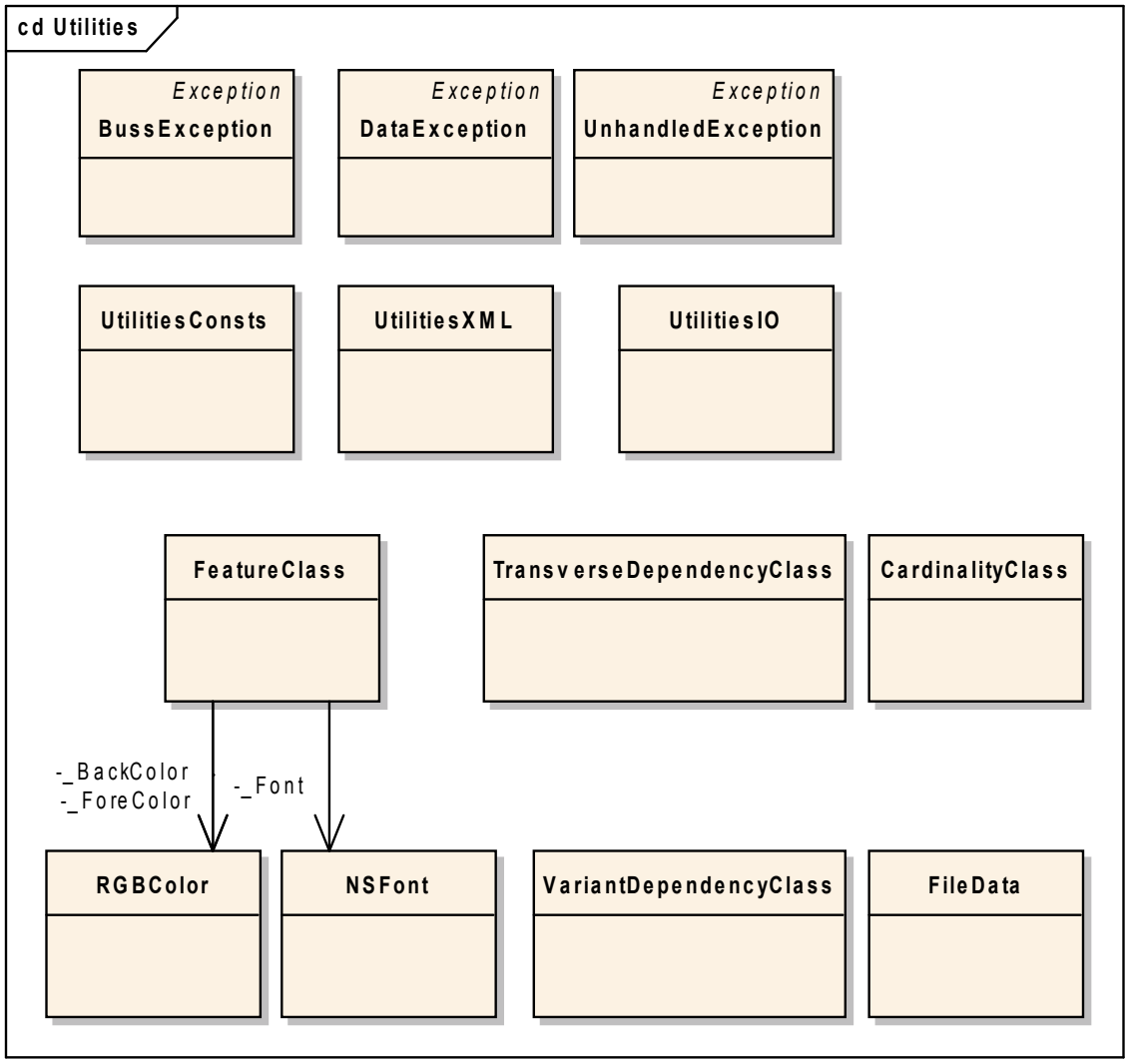


Figure 7-1-7: Class diagram of PLMVyV.Utilities.DLL

7.3 System Functionality

In broad strokes, the system permits to create fourth types of models:

- Feature product line models
- Particular configurations of feature product line models.
- Use case product line models
- Restriction product line models

In this release we have implemented components for PLMs and PLCs construction. A project is a set of one or several models, one by default. We have implemented the following functions:

1. To create a PLM, where it can be applicable:

2. Exporting and importing the model to XMI file. Due to these functions it is possible to communicate with and from other applications. The second function is not still implanted.
3. On PLMV&V it is also possible to validate some characteristics on a PLM, like unbound cardinality, root unicity and cyclic relations. Other validations issues that we want to implement on future releases are: contradiction in optionally relation, contradiction in mandatory relation, contradiction in transversal dependency – cardinality, contradiction in transversal dependency – variant dependency, contradiction between transversal dependencies, identification of features that we can not access.
4. To validate a particular product line model configuration. This functionality allows the verification of a PLC model compared to a PLM. The set of verified characteristics on a PLC are: root unicity, correctness (same PLM's root, PLM's mandatory dependencies are fulfilled in the PLC model, inclusion of PLC's features into PLM's feature set, all transverse dependencies defined in the PLM are included in the PLC model, all cardinalities defined in the PLM are respected in the PLC model)
5. PLMV&V also permits to create a particular configuration of the product line model. On this model we can export and import it in a XMI file, the second function is not still implemented.

Functionalities of the system are represented in Figure 7-1-8.

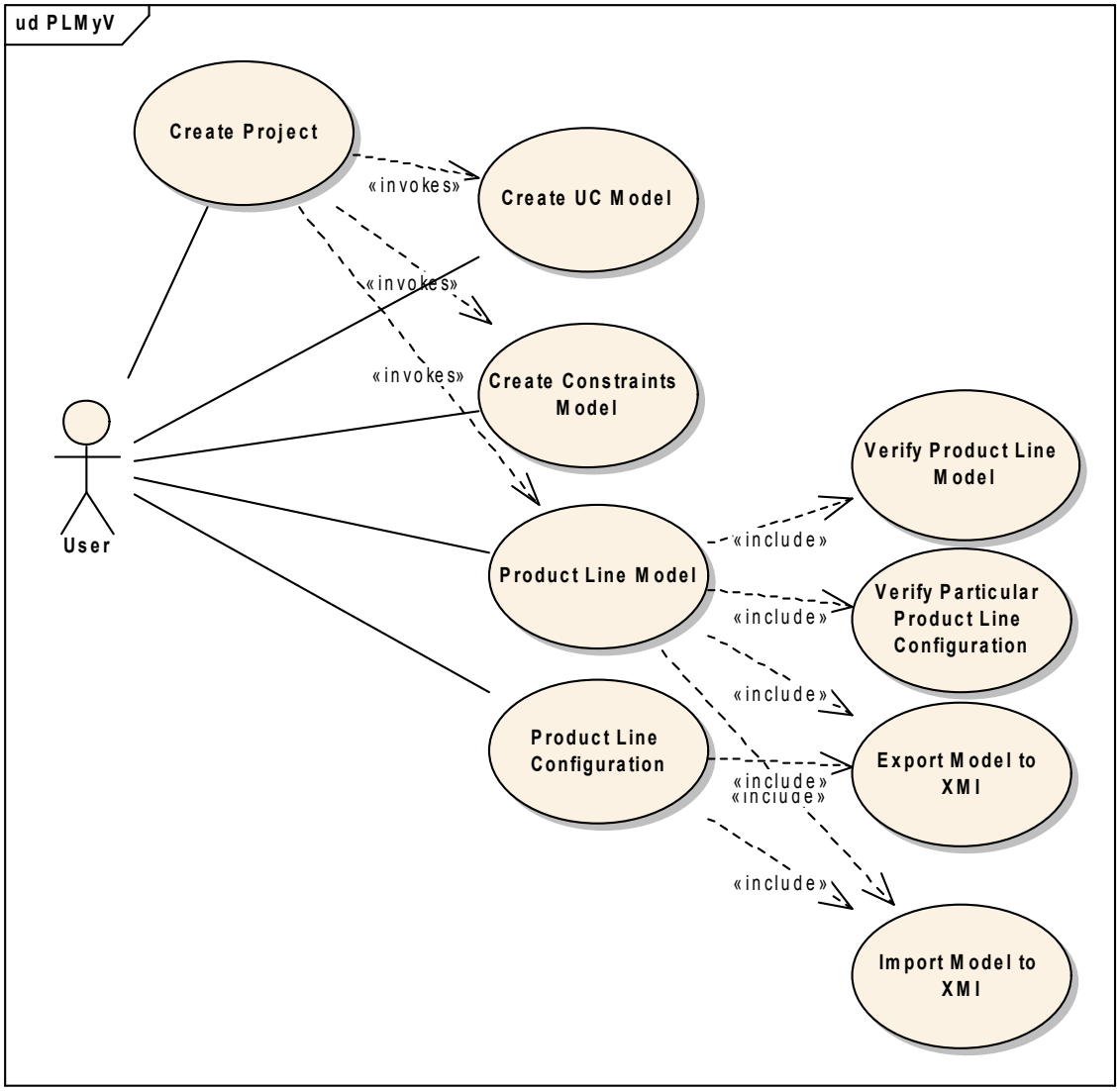


Figure 7-1-8: PLMV&V's use cases diagram.

Example 7-1-1:

This example shows how an actor can create a project and use the most important functionalities of the system.

As soon as the application is opened, the user can create either a project or a specific model. By default, when the project is created, three interfaces are available to create different types of PLM.

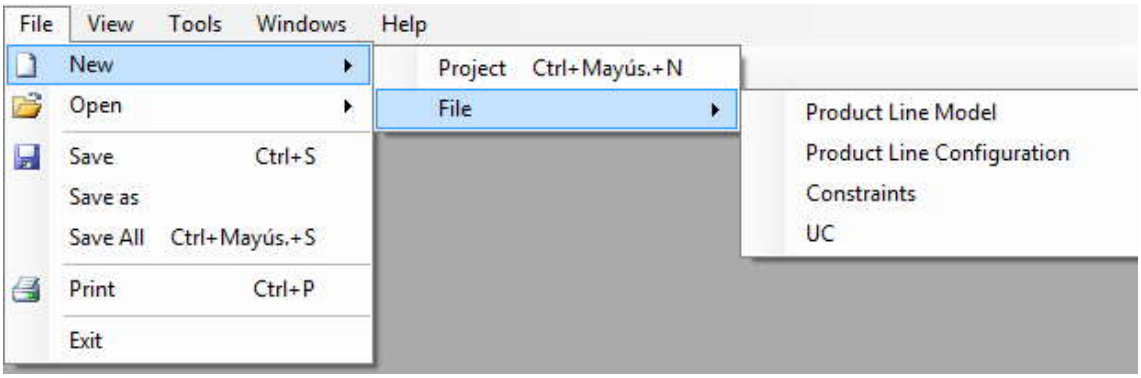


Figure 7-1-9: Interface to create a new project or a new model

In our example, user has created a new project called Demo and saved it in “documents and settings” folder.

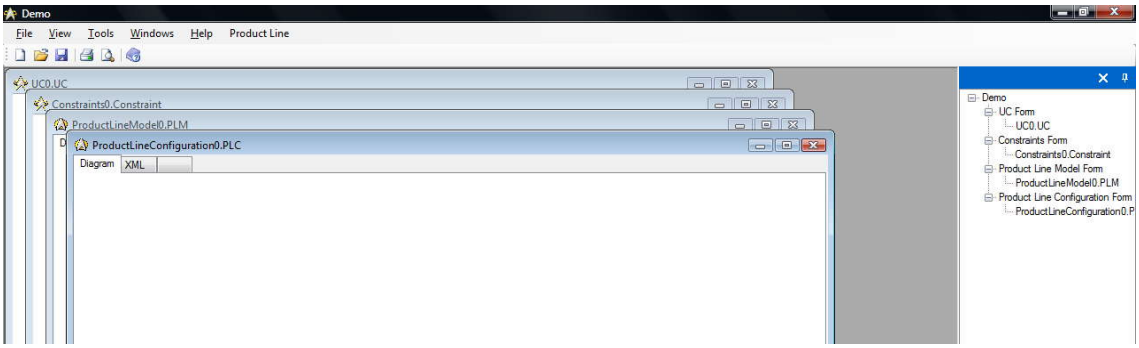


Figure 7-1-10: New project interface

In Figure 7-1-11, user active ToolBox and Properties windows in order to create a new PLM in feature notation.

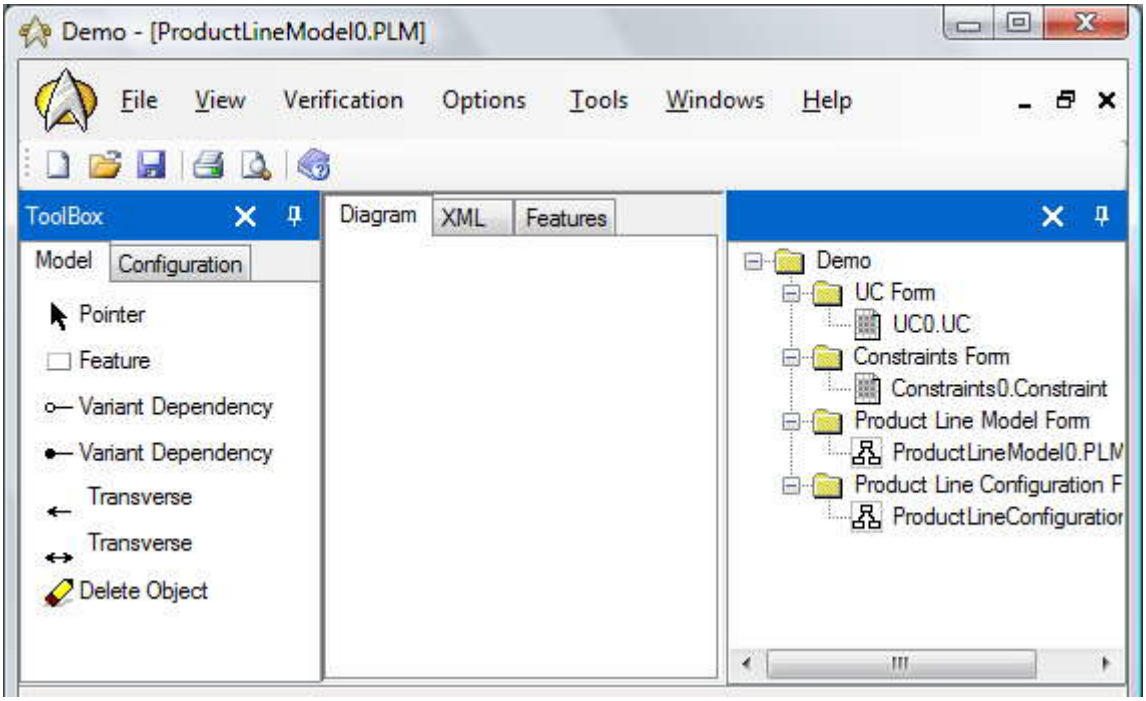


Figure 6-1-11: Interface permitting to create and configure a PLM

The next figure shows a PML in construction using features notation.

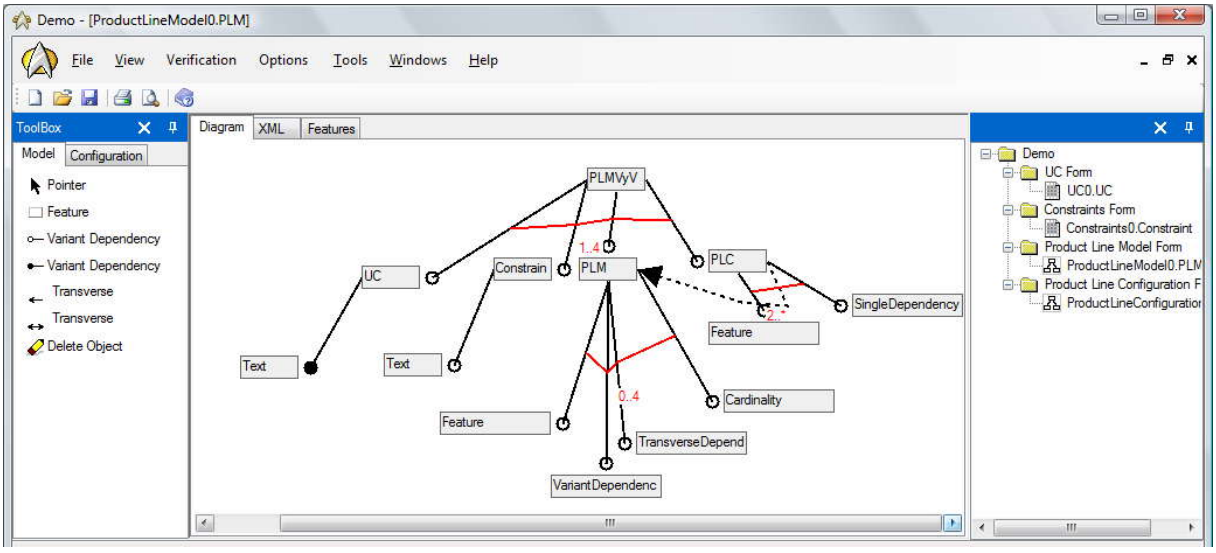


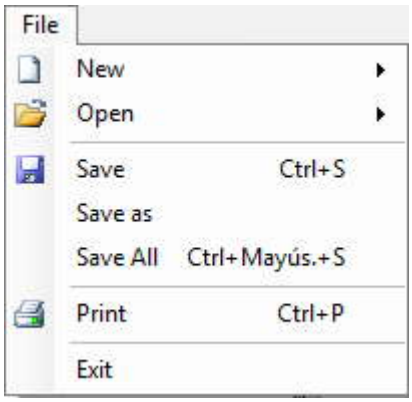
Figure 7-1-12: DAG of PLM in construction.

7.4 Manual of the Application

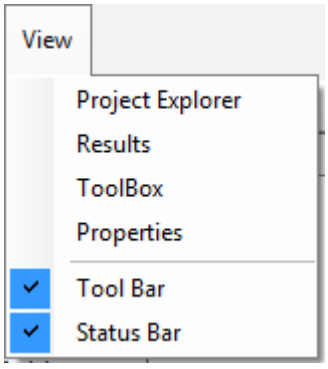
Menu bar is composed of:

1. File, that contains: (i) New, this option allows to create a new project or model. When a new model is created in an open project, the model becomes a new element of the project. (ii) Open,

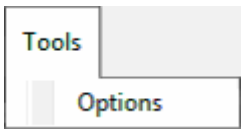
when a model is opened in a project, this model becomes a new element of the project. (iii) Save, “Save as” and “Save all”, permit to store the current active model with the same name, with different name and the entire project, respectively. (iv) Print, this option shows a preliminary view of the current model and lets us print it. (v) Exit permit to get out of the application, if a change is still pending to save, an alert message is showed.



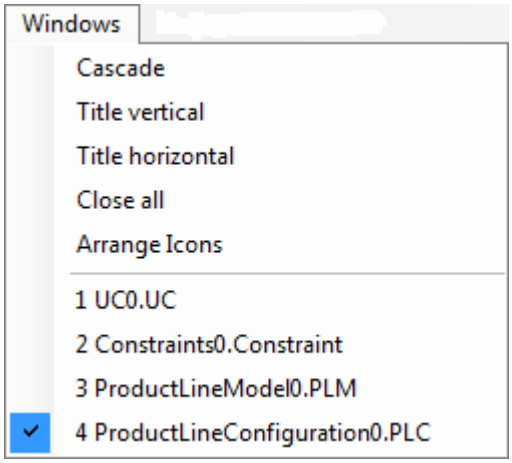
2. View is composed of: (i) Project Explorer, this option shows the project exploration bar. (ii) Results show the bar with this name. (iii) ToolBox, show the tool box, only if a feature PLM is active. (iv) Properties show the properties box, only if a PLM based on features is active.



3. Tools: this option allows showing the system’s configuration options, but it is not still implemented.

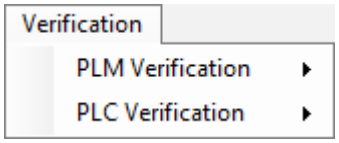


4. Window is composed of: (i) Cascade, it allows organizing all the project's models in the form of cascade. (ii) Title vertical, it allows organizing all the project's models in a vertical way. Title horizontal, it allows organizing all the project's models in a horizontal way. Close all is an option that allows closing of the project's models, if a model has been changed, an alert message is showed. Arrange Icons is an option that organizes all icons of the project. The model that is currently active is showed in blue.

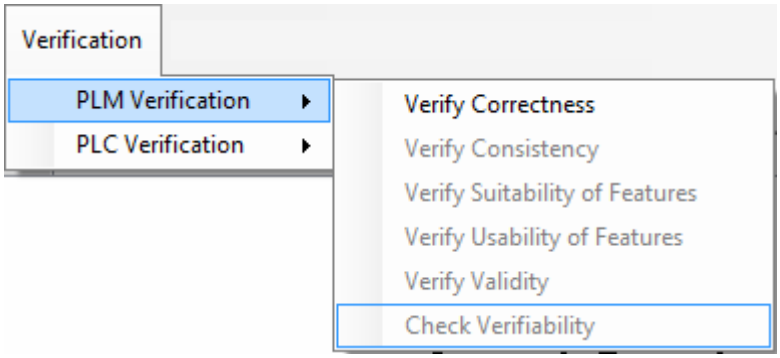


5. Verification

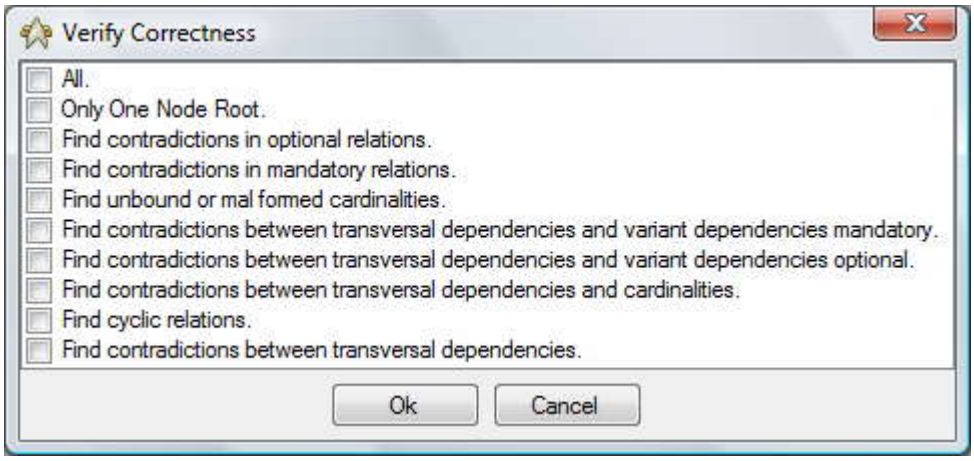
This menu's element is activated only when a model is activated .



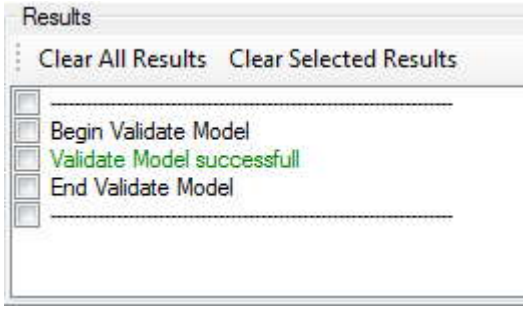
5.1 PLM Verification



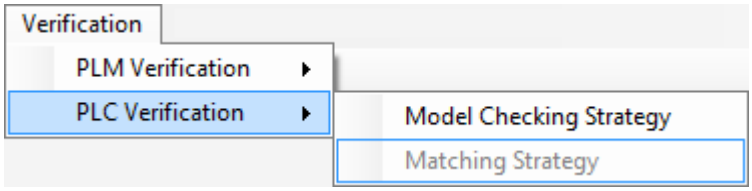
This option allows verifying correctness property in PLMs and PLCs models. In future releases the rest of properties will be implemented. When “Verify Correctness” i s chosen, next window is automatically showed permitting select different options to be verified on the current model.



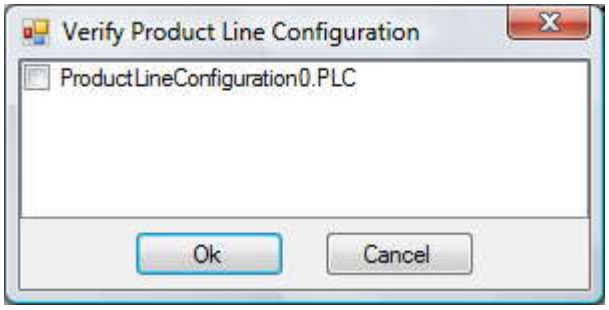
And in the output bar, the resulting verification will be showed.



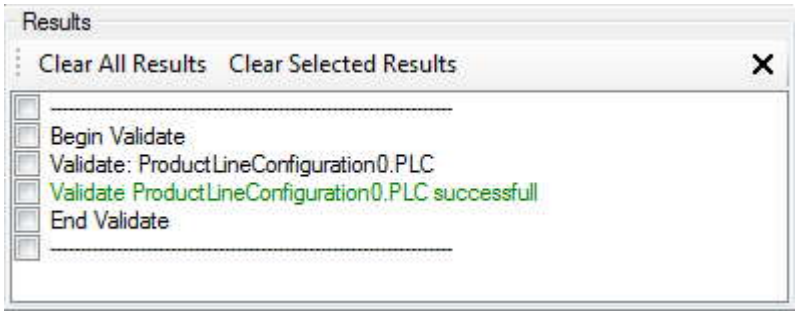
5.2 PLC Verification



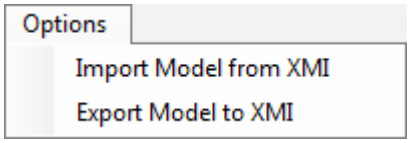
PLC Verification allows verifying a PLC model against its corresponding PLM. At the moment “Model Checking Strategy” option is only available. “Matching Strategy” will be implemented in future releases. Selecting “Model Checking Strategy”, a list of PLCs models members of the project will be showed.



And in the output bar, the resulting verification will be showed.

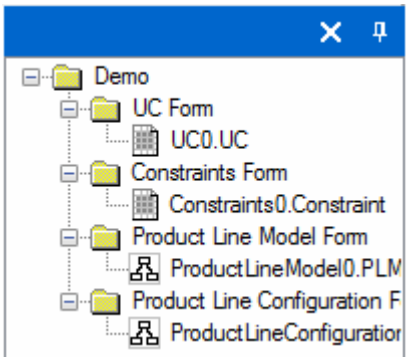


6. Options Menu is composed of: (i) Export model to XMI, this functionality allows exporting a feature model in a XMI file. (ii) Import model to XMI allows to open a XMI file and to show the content of the file graphically.

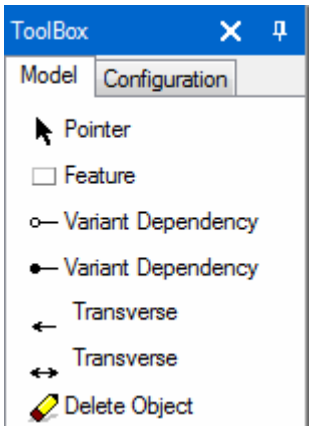


Next, we will present the project's bars of PLMV&V:

1. Project administrator allows seeing the different models organised by type of project. In order to open a model, it is only necessary to click on it.

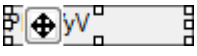


2. Tool box - Model: provides all elements that allow the construction of a feature model.



Is useful for selecting an element of the model.

Is useful for selecting an element that will be moved from one place to another one or from a model to a different one.

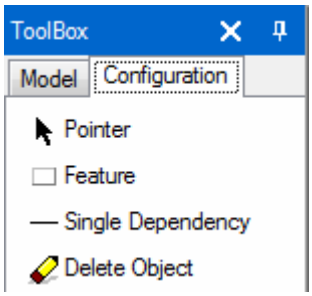
When an element is selected it is ready to be moved: 

Optional and Mandatory dependencies, respectively. These ones are useful for creating Variant-Dependency relationships between features. The link must be made between parent and child features.

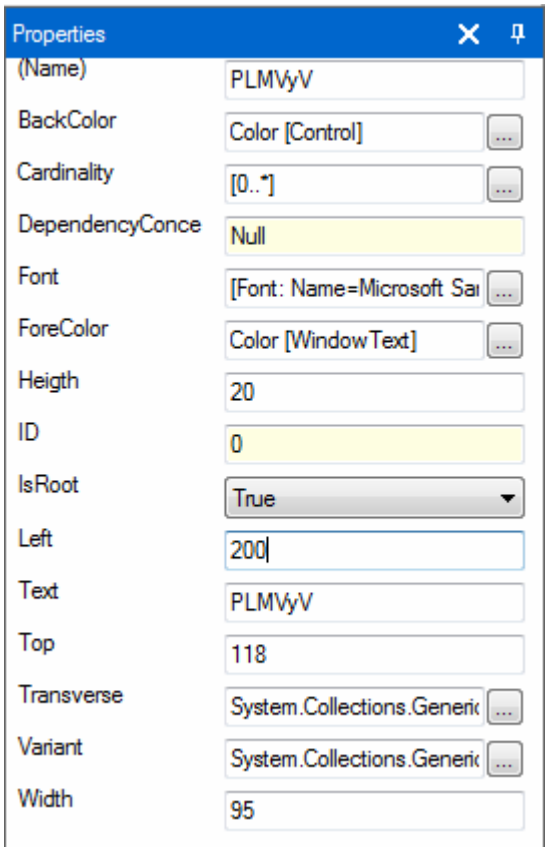
Require and Exclude dependencies, respectively. These ones are useful for creating Transverse-Dependency relationships between two features. Any click made on an empty place of the model, make a flexion point in the line of relation.

This control lets us delete an element from the model.

3. Tool box – Configuration: Lists all components that can be added in a particular product line configuration model. Used elements are discussed above.



4. Feature properties bar: permits to edit all properties of a model's element.



(Name): is the mane of the corresponding feature in the model.

BlackColor: feature's background color.

Cardinality: cardinality of the selected child and optional features.

DependencyConcept: Allows giving the name of a dependent concept.

Font: Type of font features.

ForeColor: font's color.

Height: height of features box.

ID: features identificator, is used to establish relations between features.

IsRoot: this field allows indicating if the current feature is the model's root .

Left: left location features.

Text: textual description features showed on the model .

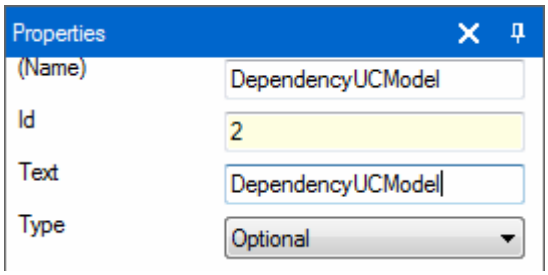
Top: vertical location features.

Transverse: this field allows to do or to add a TransverseDependency from the current feature.

Variant: this field allows to do or to add a VariantDependency from the current feature.

Width: features width.

5. Relationship properties bar: permits to edit all properties of a relationship



(Name): is the name of the corresponding relationship.

Id: relationship identificator, is used to establish cardinalities and a list of model's relationships.

Text: text of the associated relationship.

Type: the type of a relationship can be optional or mandatory.

7.5 Limitations of application

- Product Line models using use cases formalism will not be implemented in this release.
- Product Line models using constraints formalism will not be implemented in this release.
- PLM is not modifiable from XML file
- PLMV&V is only available as a desktop application
- PLMV&V is only available in English language
- The manipulation of carnalities must be done using the editor of properties

7.6 Conclusion

In the second part of this chapter we have presented the automation of our PLM correctness verification process. In order to automat it, we have developed a computational tool called PLMV&V, and we have presented its architecture, functionality, as well as a use guideline, some limitations and proposals to be implemented in future releases. With this application we thus showed that both, the approach and the tool, are operational and can be applied to an industrial case.

Part VII

Perspectives and Conclusion

Perspectives

At the present time we work on PL defined by the features models with simple configuration's restraints (If F1 so F2 or if F1 so not F2). Besides, new PL models appear in which configuration restraints are different (for instance, restraints on attribute value, on groups of attributes, etc) that can not be solve by simples SAT solvers. It is on this direction our search must persist.

Conclusion

The goal of this paper was to investigate the verification of product lines models. We believe that this goal was reached. Even though there are still open questions, the present paper can be seen as a proof of affirmation that it is possible to reduce every verification problem of a PLM designed with feature notation, to a validation of constraint problem.

We have focused in definition of a multi-process method for verify correctness in feature product line models. There are two aspects that must have been taken into account in a verification process of PLM's correctness, the structural and the semantic correctness of the model. From the point of view of the structural correctness, we have enriched the range of properties to verify in a PLM and we have proposed the corresponding logic invariants to evaluate each one. We have included not only the properties of a model designed with the notation FORE, but also another one like: correctness in optional relations, correctness in mandatory relations, correctness in cardinalities, correctness in transversal dependencies and variant dependencies interactions, correctness in transversal dependencies interactions, every feature must be possible to achieve and no cyclic relations are permitted. And from the point of view of the semantic correctness, we have unified dispersed criteria found in literature and have select some of the most important characteristics to be verified in a PLM. Some logic invariants to evaluate usability, suitability, validity and verifiability properties have been proposed. The different paths to execute each one of the previous logic invariants on a PLM are defined in the multi-process model in Figure 5-2-1.

For the verification of each of correctness's characteristics, we have used propositional logic and first-order logic for writing out of every invariant to verify. In order to evaluate a propositional logic formula, we have used in some cases satisfiability criterion and validity criterion in others.

Benefits of our approach to PLM verification are (i) its foundations in the well accepted requirements engineering framework, which allows the approach to be very general; (ii) this approach not only gathers the proposals of verification found in literature, but proposing another innovative rules and standardizes them through a same language and a same multi-model of verification; (iii) this is an approach not only focusing on single-system models (PLC models like the most of the literature do), but extended to the evaluation of PLMs.

Based on the general approach, we validate it making use of a case study and automate it through a computational tool called PLMV&V.

As limitation, we have that formulas or invariants proposed can not been directly used by any available SAT tools. Although the aforementioned invariants have been used in our PLMV&V tool, its syntax is not enough independent of its implementation as for being used by some SAT solver standard.

Part VIII

Appendix

PLMV&V tool and its explanation video available in:

http://sites.google.com/site/plmcommunity/

In this URL you can download free version of:

PLMV&V.exe

PLMV&V_explanation_video.wmv

Bibliography

[Alexandria] http://www.theoinf.tu-ilmenau.de/~riebisch/pld/index.html

[Alférez *et al*. 07] M. Alférez, U. Kulesza, A. Garcia, A. Moreira, J. Araújo, and V. Amaral. *Towards Volatility Analysis in Software Product Line Engineering,* presented at Second International Workshop on Aspect-Oriented Product Line Engineering held in conjunction with GPCE'07 (Generative Programming and Component Engineering), Saltzburg, Austria, 2007.

[Antkiewicz, Czarnecki 04] M. Antkiewicz, K. Czarnecki. FeaturePlugin: feature modeling plug-in for Eclipse. Proceedings of the OOPSLA workshop on eclipse technology eXchange, 2004. http://swen.uwaterloo.ca/~kczarnec/

[Bachmann *et al.* 2003] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig; "A Meta-Model for Representing Variability in Product Family Development", In: *Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5), Siena, Italy*, 2003, pp. 66–80.

[Balzer 91] R. Balzer, Tolerating inconsistency, in: Proceedings of the 13th IEEE International Conference on Sofware Engineering (ICSE13), IEEE Computer Society Press, Austin, Texas, 1991, pp. 158-165.

[Bass *et al.* 99] Bass, L., Clements, P., Donohoe, P., McGregor, J., and Northrop, L. 1999. *4th Product Line Practice Workshop Report*, CMU/SEI-2000-TR-002, Software Engineering Institute, CMU.

[Batory 05] Don S. Batory. Feature Models, Grammars, and Propositional Formulas. In Proceedings of the 9th International Software Product Line Conference (SPLC), pages 7–20, 2005.

[Batory, Thaker 06] Batory, D.; Thaker, S.: Towards Safe Composition of Product-Lines. Dept. Computer Sciences, University of Texas, TR-06-33, 2006.

[Bayer *et al*. 99] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, and Tanya Widen. *PuLSE: A Methodology to Develop Software Product Lines.* Fraunhofer Institute for Experimental Software Engineering, Germany and Lucent Technologies Software Product Line Engineering Laboratories, U.S.A. May 1999.

[Benavides, Ruiz 05] D. Benavides, A. Ruiz-Cort´es, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.

[Benavides *et al.* 06a] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feture models. In *Jornadas de Ingenier´ıa del Software y Bases de Datos (JISBD)*, 2006.

[Benavides *et al.* 06b] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using java csp solvers in the automated analyses of feature models. *LNCS*, 4143:389–398, 2006.

[Benavides *et al.* 06c] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: WorkingWith Variability Mechanisms*, 2006.

[Benavides *et al.* 07] David Benavides, Sergio Segura, Pablo Trinidad and Antonio Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. First International Workshop on Variability Modelling of Software-intensive Systems (VaMoS) January 16–18, 2007, Limerick, Ireland, pp 129-134

[Benjamen 99] Benjamen, A. (1999). Une approche multi-démarches pour la modélisation des démarches méthodologiques. Unpublished doctoral dissertation, University of Paris 1–Sorbonne, France.

[Bjorner 06] Dines Bjorner Software Engineering 3 Domains, requirements and Software Design. Springer-Verlag 2006

[Boehm 81] Barry Boehm. Software Engineering Economics. Prentice Hall, NJ, USA, 1981

[Boehm 84] Barry Boehm, Verifying and validating software requirements and design specifications, IEEE Software 1 (1) (1984) 75-88.

[Bosch *et al.* 2002] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, and K. Pohl; "Variability Issues in Software Product Lines" In: *Proceedings of the 4th International Workshop on Product Family Engineering (PFE-4), Bilbao, Spain, October 3–5, 2001*, Springer, Berlin Heidelberg New York, LNCS 2290, 2002, pp. 13–21.

[Bradley, Manna 07] Aaron Bradley, Zohar Manna. The calculus of computation – Decision procedures with applications of verification. Stanford University. Springer-Verlag Berlin Heidelberg 2007

[Bryant 86] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[Bühne *et al.* 03] S. Bühne, G. Halmans, and K. Pohl; "Modeling Dependencies between Variation Points in Use Case Diagrams", In: *Proceedings of the 9th International Workshop on Requirements Engineering – Foundation for Software Quality (REFSQ'03), Klagenfurt/Velden, Österreich, June* , 2003.

[Bühne *et al.* 06] Bühne, S.; Halmans, G.; Lauenroth, K.; Pohl, K.: Scenario based Application Requirements Engineering. In: Software Product Lines - Research Issues in Engineering and Management. Springer, Heidelberg, 2006, pp. 161-194.

[Clark, Wheelwright 1995] K. Clark and S. Wheelwright; *Leading Product Development*, Free Press, New York,1995.

[Classen 07] Andreas Classen. Master work: Problem-Oriented Modelling and Verification of Software Product Lines. Facultés Universitaires Notre-Dame de la Paix, Namur Institut d'Informatique. 2007

[Cockburn 00] Alistair Cockburn, Writing Effective Use Cases, Addison-Wesley, 2000.

[Cook 71] S. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing* , pages 151–158, 1971.

[Coplien *et al.* 98] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. IEEE Software, 35: 8:705–754, 1998.

[Czarnecki, Kim 05] K. Czarnecki and C. H. P. Kim. Cardinality-based feature modeling and constraints: a progress report. In *International Workshop on Software Factories*, San Diego, California, Oct 2005. Paper available at http://www.ece.uwaterloo.ca/ kczarnec/sf05.pdf.

[Czarnecki, Pietroszek 06] Krzysztof Czarnecki, Krzysztof Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints 2006

[Davis 92] Davis, P.K., "Generalizing concepts of verification, validation, and accreditation (VV&A) for military simulation," RAND, October 1992a (to be published as R-4249-ACQ).

[Easterbrook *et al.* 95] S. Easterbrook, B. Nuseibeh, Managing inconsistencies in an evolving specification, in: Proceedings of the Second International Symposium on Requirements Engineering (RE95), York, England, 1995, pp. 48-55.

[Easterbrook 96] Steve Easterbrook. The role of independent v&v in upstream software development processes. In Proceedings, 2nd World Conference on Integrated Design and Process Technology (IDPT), Austin, Texas, December 1996.

[Eriksson *et al.* 05] Magnus Eriksson, J¨urgen B¨orstler, and Kjell Borg. The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In Proceedings of the 9th International Software Product Line Conference (SPLC), pages 33–44, 2005.

[Fantechi *et al.* 04] Fantechi, A.; Gnesi, S.; Lami, G.; Nesti, E.: A Methodology for the Derivation and Verification of Use Cases for Product Lines. In Proceedings of SPLC 2004, 2004, pp. 255 –265.

[Faulk 01] Faulk, S.R.: Product-line requirements specification: An approach and case study. In Proceedings of RE01, 2001.

[Fey *et al.* 02] D. Fey, R. Fajta, and A. Boros; “Feature Modeling - A Meta-Model to Enhance Usability and Usefulness”; In: *Proceedings of the 2nd International Conference on Software Product Lines (SPLC-2), San Diego, USA*, Springer, Berlin Heidelberg New York, LNCS 2379, 2002, pp. 198–216.

[Firesmith 03] Donald Firesmith, Specifying Good Requirements, Journal or Object Technology (JOT), Vol. 2, No. 4, July-August 2003

[Griss *et al.* 98] M. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In Proceedings of the Fifth International Conference on Software Reuse (ICSR), pages 76–85, Vancouver, BC, Canada, June 1998.

[Halmans, Pohl 03] G. Halmans and K. Pohl; "Communicating the Variability of a Software Product Family to Customers", *Software and Systems Modeling*, vol. 2, no. 1, March 2003, pp. 15–36.

[Heitmeyer *et al.* 96] Heitmeyer, C.; Jeffords, R.; Labaw, B.: Automated consistency checking of requirements specifications. In: ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 3, 1996, pp. 231-261.

[Hooks 93] Ivy Hooks, Writing Good Requirements, Published in the Proceedings of the Third International Symposium of the NCOSE – Volume 2, 1993.

[Hunter, Nuseibeh 98] Hunter, A.; Nuseibeh, B.: Managing inconsistent specifications: reasoning, analysis, and action. In: ACM TOSEM. Vol. 7, No. 4, 1998, pp. 335 -367.

[Huzar *et al.* 05] Huzar, Z. et al.: Consistency Problems in UML-Based Software Development. In: N. Jardim Nunes et al. (Eds.): UML 2004 Satellite Activities, LNCS 3297, 2005, pp. 1–12.

[IEEE 98] IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications, IEEE 1998.

[IEEE 04] IEEE Computer Society, Guide to the Software Engineering Body of Knowledge (SWEBOK), version 2004, (chapter 2).

[ISO/IEC 01] ISO, "ISO/IEC 9126-1 – Software engineering – Product quality – Part 1: Quality Model", 2001.

[Jacobson *et al.* 97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley Publishing Co., 1997.

[Jackson 01] Michael A. Jackson. Problem frames: analyzing and structuring software development problems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Jaffe *et al.* 91] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, B. E. Melhart, Software requirements analysis for real-time process-control systems, IEEE Transactions on Software Engineering 17 (3) (1991) 241-258.

[Jarke *et al.* 99] Jarke, M., Rolland, C., Sutcliffe, A., & Domges, R. (1999). *The NATURE requirements engineering*. Aachen: Shaker Verlag.

[Kaiya 02] H. Kaiya, H. Horai, M. Saeki, AGORA: Attributed goal-oriented requirements analysis method, in: Proceedings of the Tenth IEEE Joint International Requirements Engineering Conference (RE02), Essen, Germany, 2002, pp. 13-22.

[Kang *et al.* 90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[Kang *et al.* 98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. Annales of Software Engineering, 5:143–168, 1998.

[Kang *et al.* 02] K. Kang, J. Lee, and P. Donohoe; "Feature-Oriented Product Line Engineering", *IEEE Software*, vol. 19, no. 4, 2002, pp. 58–65.

[Kuloor, Eberlein 02] Chethana Kuloor, Armin Eberlein. Requirements Engineering for Software Product Lines. 2002

[Krzysztof *et al.* 05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice, 10(1):7–29, 2005.

[Lami 05] Giuseppe Lami, QuARS: A Tool for Analyzing Requirements, Technical Report, September 2005

[Lamsweerde *et al.* 98] A. van Lamsweerde, R. Darimont, E. Letier, Managing conicts in goal-driven requirements engineering, IEEE Transactions on Software Engineering: special issue on Managing Inconsistency in Software Development 24 (11) (1998) 908-926.

[Landry, Oral 93] Landry, M., and Oral, M., "In search of a valid view of model validation for operations research", *European Journal of Operational Research* 66/2, 161-167. 1993

[Lauenroth, Pohl 07] Kim, Klaus Pohl. Dynamic Consistency Checking of Domain Requirements in Product Line Engineering

[Letier, Lamsweerde 02] E. LetiLauenroth, K.; Pohl, K.: Towards Automated Consistency Checks of Product Line Requirements Specifications. In Proc. of ASE07, 2007, pp. 373-376er, A. van Lamsweerde, Requirements analysis: Deriving operational
software specifications from system goals, in: Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM Press, Charleston, South Carolina, 2002, pp. 119-128.

[Leveson 00] N. Leveson, Completeness in formal speci_cation language design for processcontrol systems, in: Proceedings of the Third Workshop on Formal Methods in Software Practice, Portland, Oregon, 2000, pp. 75-87.

[Macaulay 96] L. A. Macaulay. Requirements Engineering. Springer- Verlag. 1996.

[Mader *et al.* 07] Angelika Mader, Hanno, Wupper, Mieke Boon, Jelena Marincic. Taxonomy of Modelling Decisions for Embedded Systems Verification. 2007.

[Mannion 02] M. Mannion, Using first-order logic for product line model validation, in: SPLC2, LNCS, vol. 2379, Springer, Berlin, 2002, pp. 176–187.

[Mannion, Camara 03] Mike Mannion, Javier Camara, Theorem Proving for Product Line Model Verification. Software Product-family Engineering: 5th International Workshop, PFE 2003

[Metzger *et al.* 07] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. From 15th IEEE International Requirement Engineering Conference. India Habitat Centre, New Delhi, October 15 -19 2007.

[McGregor *et al.* 2002] J.D. McGregor, S. Jarrad, L.M. Northrop, and K. Pohl; "Initiating Software Product Lines", *IEEE Software*, vol. 19, no. 4, July 2002, pp.24 –27.

[Mylopoulos *et al.* 99] J. Mylopoulos, L. Chung, E. Yu, From object-oriented to goal-oriented requirements analysis, Commun. ACM 42 (1) (1999) 31-37.

[Mueller 06] Erik T. Mueller. Commonsense Reasoning. Morgan Kaufmann, 2006.

[Nuseibeh 96] B. Nuseibeh, To be and not to be: on managing inconsistency in software development, in: Proceedings of the Eight IEEE International Workshop on Software Speci_cations and Design (IWSSD'96), IEEE Computer Society Press, 1996, pp. 164-169.

[Nuseibeh, Easterbrook 00] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In Proceedings of the 22nd International Conference on Software Engineering (ICSSE) - Track: The Future of Software Engineering, pages 35 –46, New York, NY, USA, 2000. ACM Press.

[Padmanabhan, Lutz 05] Padmanabhan, P.; Lutz, R. R.: Tool-Supported Verification of Product Line Requirements. In: Automated Software Engineering, Vol. 12, No. 4, 2005, pp. 447 -465.

[Pohl *et al.* 05] Klaus Pohl, Gunter Bockle, and Frank van der Linden. Software Product Line Engineering: Foundations, Principles and Techniques. Springer, July 2005

[Pohl, Metzger 06] Klaus Pohl and Andreas Metzger. Variability management in software product line engineering. 28th International Conference on Software Engineering (ICSE'06), May 2006.

[Ponsard *et al.* 05] C. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde and H. Tran Van. Early Verification and Validation of Mission Critical Systems. Electronic Notes in Theoretical Computer Science 133 (2005) 237–254, 2005 Elsevier B.V. Belgium

[Probert *et al.* 03] Robert L. Probert, Yanping Chen, Behrad Ghazizadeh, D. Paul Simsb, Maurus Cappa. Formal verification and validation for e-commerce: theory and best practices. Information and Software Technology 45 (2003) 763–777

[PureVariants] [pure-systems GmbH] http://www.software-acumen.com/purevariants/feature-models/ http://www.pure-systems.com/fileadmin/downloads/pv-clearquest-whitepaper-en.pdf

[Riebisch *et al.* 02] M. Riebisch, K. Böllert, D. Streitferdt and I. Philipp ow, Extending Feature Diagrams with UML Multiplicities, in Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, CA, June 2002.

[Rolland *et al.* 96] Rolland, C., & Prakash, N. (1996). A proposal for context-specific method engineering. *Proceedings of the IFIP WG 8.1 Conference on Method Engineering* (pp. 191-208), Atlanta, Georgia. Chapman and Hall.

[Rolland *et al.* 99] Rolland C., Prakash N., Benjamen A.: A Multi-model View of Process Modelling, Requirements Engineering J. 4(4), pp.169-187, 1999.

[Savolainen, Kuusela 01] Savolainen, J., Kuusela, J. Consistency Management of Product Line Requirements. In Proceedings of RE01, 2001.

[Schobbens *et al.* 06] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. Computer Networks (2006), doi:10.1016/j.comnet.2006.08.008, special issue on feature interactions in emerging application domains, page 38, 2006.

[Shanahan 99] Murray Shanahan. The event calculus explained. Lecture Notes in Computer Science, 1600:409–430, 1999.

[Sommerville, Kotonya 98]. Sommerville, I. Kotonya, G. Requirements Engineering: Processes and Techniques. John Wiley & Son Ltd. 1998.

[Spinczyk, Beuche 04] O. Spinczyk, D. Beuche, Modeling and Building Software Product Lines with Eclipse, in proceedings of the international Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004.

[Streitferdt 03] D. Streitferdt, Family-Oriented Requirements Engineering, PhD Thesis, Technical University Ilmenau, 2003.

[Svahnberg et al., 2001] Svahnberg, M., Gurp, J. V., and Bosch, J. (2001). "On the notion of variability in software product lines". In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture* (WICSA'01), pages 45–54, Amsterdam, The Netherlands.

[Tsang 95] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1995.

[van Gurp *et al.* 01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), 2001.

[V.d. Maßen and Lichter 02] T. von der Maßen and H. Lichter; "Modeling Variability by UML Use Case Diagrams", In: *Proceedings of the International Workshop on Requirements Engineering for Product Lines (REPL'02)*, 2002, pp. 19–25.

[Wang *et al.* 05] Hai Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, Verify Feature Models using Protégé-OWL. May 10–14, 2005, Chiba, Japan. ACM 1595930515/05/0005.

[Weiss, Lai 1999] D.M. Weiss and C.T.R. Lai; *Software Product-Line Engineering – A Family-Based Software Development Process*, Addison-Wesley, Reading, Massachusetts, 1999.

[Wiegers 99] Karl E. Wiegers, Writing Quality Requirements, Software Development magazine, May 1999

[Zave, Jackson 97] Pamela Zave and Michael A. Jackson. Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology, 6(1):1 –30, 1997.

[Zave 01] Zave, P. 2001. Requirements for evolving systems: a telecommunications perspective. 5th Int'l Symp. Requirements Eng., pp. 2-9.

[Zhang *et al.* 04] Wei Zhang, Haiyan Zhao and Hong Mei. A propositional Logic-based Method for Verification *of Feature Models.* 6th International Conference on Formal Engineering Methods, ICEFEM 2004, held in Seattle, WA, USA in November 2004.

[Zowghi, Gervasi 03] Didar Zowghi, Vincenzo Gervasi. On the Interplay Between Consistency, Completeness, and Correctness in Requirements Evolution. Preprint submitted to Elsevier Science 1 April 2003