MASTER IN COMPUTER SCIENCE

Visualisation in Software Product Line Engineering

ALLARD, Kévin; ERNOUD, Julien

Award date:
2008

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
rue Grandgagnage, 21 B-5000
http://www.fundp.ac.be/facultes/info

Visualisation in Software Product Line Engineering

Julien Ernoud & Kevin Allard

Mémoire présenté en vue de l'obtention
du grade de maître en informatique

Année Académique 2007-2008

Abstract

Software Product Line Engineering is a software development paradigm which is a combination of Software Engineering and Product Line Engineering. During the last few years, this paradigm has rapidly emerged because it allows realising significant improvements in time-to-market, cost, productivity and quality.

To bring help to Software Product Lines management, many tools have been developed by the industry and the research community. A global observation is that, in a big size industrial context, the management of Software Product Lines is quite complex even with the presence of these tools. In order to deal with this complexity, new researches have been proposed for the integration of visualisation in Software Product Line Engineering. The underlying idea of this integration is to use the visual potential to manage these Software Product Lines.

Motivated by this novel concept, the purpose of this thesis is to provide a solution that could support visualisation in Software Product Lines management tools. This solution could help the engineer in order to manage these last ones in a simple and efficient manner.

Résumé

L'ingénierie des lignes de produits logiciels est un paradigme de développement logiciel qui est à la jonction entre le génie logiciel et l'ingénierie des lignes de produits. Ces dernières années, ce paradigme a rapidement émergé car, de part sa nature, il permet aux entreprises de réaliser des améliorations significatives en termes de coûts, de productivité, de qualité et de temps de réponse au marché.

Afin d'aider à la gestion d'une ligne de produits logiciels, de nombreux outils ont été développés par l'industrie et le monde de la recherche. Une observation générale est que, dans un contexte industriel de grande taille, la gestion d'une ligne de produits logiciels s'avère relativement complexe même avec l'aide de ces outils. En vue de maîtriser cette complexité, de nouvelles recherches ont été proposées afin d'intégrer la visualisation dans l'ingénierie des lignes de produits logiciels. L'idée sous-jacente à cette intégration est d'utiliser le potentiel visuel pour gérer ces lignes de produits logiciels.

Motivé par cet innovant nouveau concept, le but de ce mémoire est dès lors de fournir une solution qui pourrait supporter la visualisation dans les outils de gestion de lignes de produits logiciels et, par conséquent, aiderait l'ingénieur à gérer ces dernières d'une manière simple et efficace.

Preface

Software Product Line Engineering and *Visualisation* are both active domains. Typically some current researches are intending to use visualisation in Software Product Line Engineering. The increasing interest in the combination of these two domains led to the 1st International Workshop on Visualisation in Software Product Line Engineering (VISPLE, 2007, Kyoto, Japan) [Home]. This workshop proposes a novel and challenging research direction in Software Product Line Engineering by focusing on the idea of using visualisation techniques to achieve the management of *Software Product lines*.

The upcoming section and paragraphs provide an overview of the context in which this thesis is set and its main motivation. The preface concludes with a brief overview of the main chapters of the thesis.

Software Product Line Engineering Context

During the last few years, Software Product Line Engineering has rapidly emerged as a viable and important software development activity which results of the combination between Software Engineering and Product Line Engineering. In this novel approach, companies build different variants of their products using a *Common Platform* which encompasses similarities and differences existing between them. Typically, a Software Product Line is a "*set of software-intensive systems that share, a common, managed, set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" [PBvdL05].

For instance, we can consider the case of car security systems. These systems share common features like frontal airbags control and ABS launching. However some systems can have differences such as lateral airbags control, *Electronic Stability Program* (ESP), *Acceleration Slip Regulation* (ASR) or security switching-off button for sport mode driving. In this case, the Software Product Line approach allows car companies to develop all possible car security systems in order to reuse them in different car models. Thereby, the car companies have not to develop a particular car security system for each car models. This variety of products sets up the concept of *Customisation* which allows companies to fit better with the customer's needs

By using a Common Platform to fit with the customer's need, Software Product Line Engineering intends to review the products development of companies in order to realise significant improvements in time-to-market, cost, productivity and quality.

Motivations

To manage the Software Product Line Engineering activities, the Software Engineering community has developed some tools. These activities consist, on the one hand, of the creation and the maintenance of the Software Product Line (referred as *Domain Engineering*) and, on the other hand, of the Software Product Line use to find a product which fits at best with customer's needs (referred as *Application Engineering*). Each of these aspects requires tools with pertinent operations to facilitate these activities. Following the VISPLE idea the thesis focuses mainly a research of how the visualisation could address the problems encountered during the Software Product Line management.

Therefore, the main motivation of this thesis is to elaborate a solution to support visualisation in tools supporting the Software Product Line Engineering activities. However, as VISPLE says, the size of Software Product Lines is usually large. This leads to make tough the management of these ones and more particularly the Application Engineering activities. Thereby, the solution evolves in the remainder of the thesis to mainly focus on Application Engineering.

Overview

This work is divided into six chapters as follows.

Chapter one introduces the *Software Product Line Engineering* as a combination of *Software Engineering* and *Product Line Engineering*. The purpose of this chapter is to define the main concepts which are useful for the comprehension of the thesis and intend to clarify the terminological differences within the Software Product Line Engineering. Then the chapter introduces some visualisation concepts which are interesting for Software Product Lines management tools.

Chapter two investigates Software Product Line Engineering (literature, tools, etc.) in order to have knowledge about relevant functions for Software Product Lines management tools. This investigation is leading away to the achievement of an evaluation framework for Software Product Lines management tools.

Chapter three focuses on Application Engineering, so called *Product Derivation Process*. It analyses product derivation problems coming from our personal experience and a case study outlining two industrial Software Product Lines. This problems analysis conducts to the discovery of challenges for Software Product Lines management tools.

Chapter four achieves the main motivation of this work by proposing a solution which uses visualisation to address challenges of the chapter three. This chapter focuses mainly

on challenges which are not yet addressed by existing tool functions. Then this chapter introduces a metamodel in order to develop the proposed solution into a tool.

Chapter five presents the implementation of a prototype in order to give a concrete idea of the proposed solution in the chapter four. This chapter explains the architectural design and the different functions of this prototype.

Chapter six reviews our approach and contributions during the thesis and proposes some possible further works.

Acknowledgments

Before to start this thesis, we do not want to miss the opportunity to thank all people that have contributed to its elaboration.

To the **Lero** (the Irish software engineering research centre) for their welcome during our internship in Ireland. More particular thanks to Steffen Thiel, Daren Nestor, Ciaran Cawley, Goetz Botterweck and Padraig O'Leary for their assistance and following during our internship.

To our supervisor, Prof. **Pierre-Yves Schobbens** for giving us precious advices and continuous support to achieve our thesis.

To Prof. **Patrick Heymans** for giving us the chance to work in a convivial place and to improve our English.

To our professors and friends for their technical support and our family for their spiritual support during the elaboration of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

State Of The Art

The Merriam-Webster Dictionary [MW03] gives the following definition of a state of the art: "*the level of development (as of a device, procedure, process, technique, or science) reached at any particular time usually as a result of modern methods*". Therefore, the purpose of a state of the art consists of realising a review of the current development in a particular domain.

As we know, the purpose of this thesis is to provide a solution to support visualisation in *Software Product Line Engineering tools*. It means to split the state of the art into two distinct domains, i.e. the *Software Product Line Engineering domain* and the *visualisation* domain. However, knowledge of these two domains is quite large and is continually growing [OBTR07]. Furthermore, terminology in Software Product Line Engineering seems to vary among the practitioners. In this context, it would be unrealistic to build a complete review for these two domains.

As consequences, the state of the art just gives the principal current practices and notions of the two aforementioned disciplines in order to provide the reader a good understanding of the different concepts addressed in this thesis and in order to provide a global picture of the two disciplines for non-introduced people.

The remainder of this chapter is organised as follows: Section 1.1 introduces Software Product Line Engineering and section 1.2 presents the most important concepts of the visualisation theory and the visualisation in Software Product Line Engineering.

1.1 Software Product Line Engineering

Software Product Line Engineering is a new paradigm which is the result of the combination between *Software Engineering* and *Product Line Engineering* and where the focus is put on the concept of *Reusability*. The purpose of this section is to show this

statement. It introduces first Software Engineering. Then it presents the Product Line Engineering. Finally it shows that Software product Line Engineering is the combination of the two former and it gives a global picture of the domain.

1.1.1 Software Engineering

The late 1940s, with the establishment of the Von Neumann machine, mark the birth of software development [RH00, vN45, Mae05]. At the time, this activity is regarded as handicrafts. It is assigned to reasonable size projects with manageable complexity where the concepts of methodology and tool support are still non-existent. During this decade, software development is consequently very tacit and reserved to experts which are mainly focused on productivity [Gla97].

In the 1950s and 1960s, the software projects are beginning to be increasingly complex. Often, they include hundreds, or even thousands, of people and the developed products includes several million lines of code. The lack of methodology and quality interest for software development leads this complexity to become unmanageable [Gla97, Str96]. Thereby, many problems appear in projects resulting, in 1969, to a phenomenon, still existing today, called "software crisis" [Gla97, Neu95]. These problems are categorised in three main classes, i.e. "Cost and Budget Overruns", "Property Damage", and "Life and Death" [Neu95].

Cost and Budget Overruns: This class concerns the cost and budget overruns problems into software projects. A typical example of these problems is the development of the OS/360 operating system. The project starts in the 1960s and is, with more than thousand programmers, one of most complex systems at the time. Unfortunately, the project takes several years of delay and explodes its budget [IBM72, FPB95].

Property Damage: This class concerns the problems that cause property damages. A typical example of these problems is the intrusion, in 1996, by unidentified hackers into the Rome Laboratory, the US Air Force's main command and research facility. Using trojan horse virii, hackers obtain unrestricted access to Rome's networking systems and remove traces of their activities. The intruders obtain classified files, such as air tasking order systems data and furthermore penetrate connected networks of National Aeronautics and Space Administration's Goddard Space Flight Center, Wright-Patterson Air Force Base, some Defense contractors, and other private sector organizations, by posing as a trusted Rome center user [oD96].

Life and Death: This class concerns the problems that lead to kill people. A typical example of these problems is Therac 25 incident. Therac 25 is a radiotherapy machine which includes embedded systems. These one fail so catastrophically that they administer lethal doses to patients [Lev95].

Regarding the number of failing projects, software experts are starting to ask questions. They note that, in order to deal with the complexity of software systems, there is a strong need to have a well-defined approach, to lean on principles and methods and finally to have a tool support. This observation leads to the birth of a new discipline: Software Engineering [Str96].

The term "Software Engineering" is popularised by F.L. Bauer during the NATO Software Engineering Conference in 1968 [NR69]. The underlying idea of this denomination is to establish and use engineering principles in order to develop software, in an economic manner, which is reliable and run on real machines [Str96]. Now, both the software development process and the product of software development are not only focused on productivity but on quality as well.

Currently, Software Engineering becomes quite large. Indeed, the *Software Body of Knowledge* (SWEBOK) defines the discipline as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software [ABDM01]. Therefore, Software Engineering does not include only software development but software operation and software maintenance [Def 1.1]. The profession is trying to define its boundary and content. Since 2006, the SWEBOK is tabled as an ISO standard for this purpose (ISO/IEC TR 19759).

Definition 1.1. Software Maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment (ISO/IEC 14764).

1.1.2 Product Line Engineering

The early 1990s, with the Ford's invention of *Assembly Line* [Def 1.2], marks the birth of the *Product Line* concept [PBvdL05, Ban02, Bri03]. At the time, this invention is a revolution for the industrial world. In fact, it allows a specialisation of workers in their tasks that leads to be more productive. Therefore, it enables production for a mass market much more cheaply than handiwork. Unfortunately, it reduces possibilities for diversification [PBvdL05].

Definition 1.2. An **Assembly Line** is an arrangement of machines, equipment, and workers in which work passes from operation to operation in direct line until the product is assembled [Ban02].

Over time, industrials observe that there is a rising demand for individualised products. For example, in the industry of cars, we notice that not all people want the same kind of car for any purpose. In order to take into account the customer's wishes, the concept of *Mass Customisation* is introduced. The concept is first conceived by Stan Davis in

"*Future Perfect*" and represents the large-scale production of goods tailored to individual customers' needs [PBvdL05, Dav96, Coo00].

Unfortunately, for companies, mass customisation implies lower profits margin due to higher technological investments required for its application. To counter this problem, many companies, especially in the car industry, starts to introduce *Common Platforms* [Def 1.3]. This approach enables car manufacturers to offer a larger variety of products and to reduce costs at the same time [PBvdL05]. The first generic platform to be shared among a number of vehicles is the *Ford Fox Platform* in the 1970s [Homb].

Definition 1.3. A **platform** is any base of technologies on which other technologies or processes are built.

As a result of applying this approach, the companies using the best platform strategy increases sales by 35% within periods of three years measured from 1980 to 1991. Whereas, during the same period, the companies starting from scratch for each new series of cars have a sale loss of 7% [CK98].

The concept of product line that we know at the present time is the systematic combination of mass customisation and common platforms. Therefore, the purpose of Product Line Engineering is to establish and use the engineering principles in order to develop products tailored to individual customers' needs by using a common base of technology. The main motivations for Product Line Engineering are notably the reduction of costs, the enhancement of quality and the reduction of time to market [PBvdL05].

1.1.3 Software Product Line Engineering

Software product line engineering (SPLE) is a combination of Software Engineering and Product Line Engineering. It uses engineering principles to develop quality software tailored to individual customer's needs by using a common base of technology. In the Software Product Line Engineering context, this common base of technology is called *Software Platform*.

A Software Platform is a set of software subsystems and interfaces that form a common structure from which a set of derivative products can be efficiently developed and produced. The subsystems belonging to a software platform encompass all *Reusable Artefacts* such as requirements models, architectural models, software components, test plans and test designs.

In practice, engineers tend to mix up the terms "Reusable Artefact" and "Feature". We conserve here the term "Reusable Artefact" for a subsystem belonging to a software platform. Like *The Irish Software Engineering Research Centre* (LERO), we view the

terminology "Feature" as a high level functionality of the Software Platform. Furthermore, practitioners mix up the terms "Reusable Artefact" and "Core Asset" as well. We continue to do this crossover in the next of the thesis.

Including the concepts and the remarks mentioned before, Pohl *et al.* [PBvdL05] define a Software Product Line (SPL) as follows.

Definition 1.4. A **Software Product Line** (SPL) is a set of software-intensive systems that share, a common, managed, set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way".

1.1.3.1 Variability

With the aim of satisfying the individual customer's needs in a mass market, the Reusable Artefacts used in different products of an SPL have to be sufficiently adaptable. It means that, for exploiting the full benefits of an SPL approach, we need to define exactly the places where the products can differ so that they can have as much in common as possible. This flexibility is commonly called *Variability*.

Concretely, Variability comes to define upfront the commonalities and variabilities in products in terms of requirements, architecture, components, and test artefacts [CN01]. This definition specifies the scope of the SPL and generally depends on the business strategy of the company selling these products.

Because of its importance and because of its crosscutting nature, variability management is one of the central and most complex aspects of SPLE [BFG+02, LB07]. In order to tackle the complexity of variability management, a number of supporting modelling languages, such as *Feature Diagrams*, have been proposed.

1.1.3.2 SPLE Process

As we can see in Figure 1.1, taken from [PBvdL05], the SPLE community commonly split the SPLE process into two distinct sub-processes, i.e. *Domain Engineering* and *Application Engineering*. The differentiation is based on one proposed by Weiss and Lai [WL99]. However, the terminology varies sometimes from one practitioner to another. For instance, Clements *et al.* [CN01] use the terminology *Core Asset Development* and *Product Development* for Domain Engineering and Application Engineering respectively. We conserve the first terminology for the next of this thesis.

Domain Engineering

Pohl *et al.* [PBvdL05] define Domain Engineering as follows.

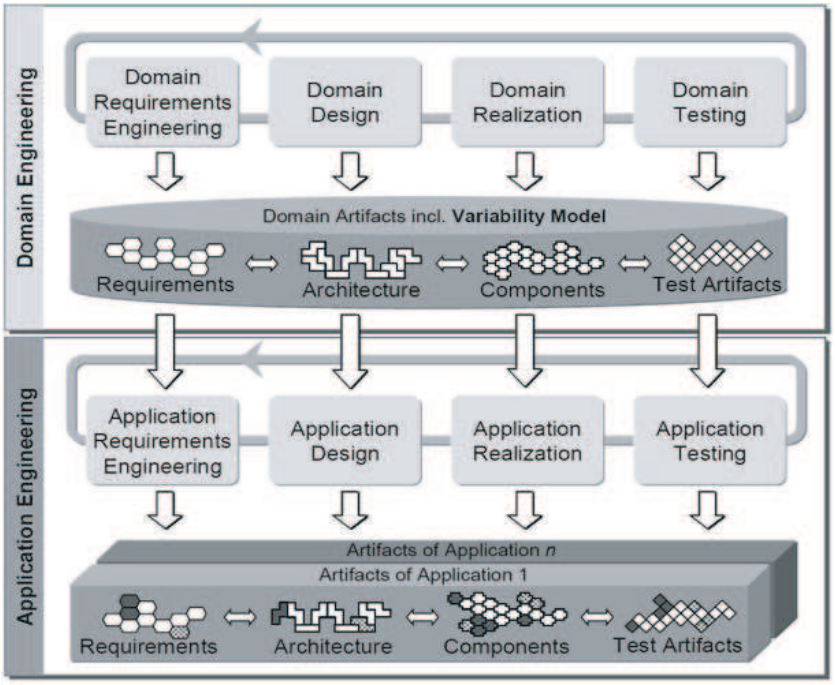

FIGURE 1.1: Schema of the SPLE process [PBvdL05]

Definition 1.5. Domain Engineering is the process of SPLE in which the commonality and the variability of the product line are defined and realised.

Thus, the purpose of this process is to decide the scope of the product line according to the company business strategy and, based on this scope, to define the commonalities and differences of products in the SPL. This definition set up the overall architecture of the SPL.

The commonalities, i.e. all properties or functionalities common to all products of the SPL, are implemented in the form of Reusable Artefacts while differences, i.e. all properties or functionalities that differ among products of the SPL, are defined in terms of *Variation Points* (a property or a functionality of the product that can vary) with their associated *Variants* (the different variations of this property or functionality) [CHW98]. For instance, a Variation Point could be the colour of the product and the variants the yellow, red, and green colours.

Application Engineering

Pohl *et al.* [PBvdL05] define Application Engineering as follows.

Definition 1.6. Application Engineering is the process of SPLE in which the applications of the product line are built by reusing reusable artefacts and exploiting the product line variability.

Indeed, as we said before, the variability of an SPL is defined in terms of Variations Points with their associated Variants making the overall architecture of the SPL. The

purpose of this process is to derive a specific product of the SPL by instantiating this overall architecture and by selecting a combination of appropriated Reusable Artefacts. In practice, engineers name this process "*Product Derivation*" as well.

The instantiation of the overall architecture is realised through the binding of each Variation Points. This means that for each Variation Points one of its Variants is chosen. Then, once all Variation Points are bound to Variants, an appropriate combination of Reusable Artefacts corresponding to these Variants is selected in order to form what is commonly called a *Product Configuration*.

In practice, the Product Derivation process is very disparate and encompasses many ways of doing. This statement can be observed in [DSB05].

1.2 Visualisation

For several years, Computer Science deals with visualisation and its problems. As MacKinlay [Mac86] said "*computer-based information plays a crucial role in our society. As a result, an important responsibility of a user interface is to make intelligent use of human visual abilities and output media whenever it presents information to the user*". This is always real in today's society because there are more and more information and it can be difficult to make sense on a huge volume of information.

Intuitively, we can see visualisation as a method which presents and stores information in an effective way. Indeed, there are many visual representations for each field (circuit diagrams, structural plans, UML models, etc) which are used by experts because they perceive quite lots of advantages. During the information system development we widely admit the assumption that the graphical representation is very effective [Pet95] and we are persuaded that slogans like "*a picture is worth a thousand words*" are true. However in the reality, most representations are a barrier to communicate effectively information [Moo06b]. Indeed, a diagram can be complex for a non-expert who is not familiar with these kinds of visualisation.

To counter this problem, the purpose of this section is to provide a global picture of visualisation principles and techniques and then, according to the subject of this thesis, to present the visualisation use in SPLE.

1.2.1 Visualisation Principles

For the question *how can we define visualisation?*, the National Science Foundation's 1987 report, *Visualization in Scientific Computing* [MDB87], answers as follows.

Definition 1.7. Visualisation is a method of computing. It transforms the symbolic into the geometric, enabling researchers to observe their simulations and computations. Visualisation offers a method for seeing the unseen. It enriches the process of scientific discovery and is revolutionising the way scientists do science.

We can feel that visualisation is perceived as a method to produce images in order to help the scientists to visualise their researches. Indeed this definition claims that visualisation is a revolution in the science world because it enables to see what scientist's could not see until now. There are more other recent definitions which are more adapted to the Computer Science.

Ware [War00] defines visualisation as "*a graphical representation of data or concepts,*" which is either an "*internal construct of the mind*" or an "*external artifact supporting decision making*". In this definition we feel more the computer science visualisation when, for instance, we have to design a future information system. In this case, visualisation assists all system stakeholders by representing all data and concept of this system visually: "*this assistance may be called cognitive support*" [TM04]. To address this assistance challenge, visualisation must use visual representations to amplify cognition [War00]. From these definitions we can conclude that the visualisation is considered as a mental activity which distinguishes from tools and techniques.

1.2.1.1 Human Graphical Information Processing

Visual representation is known in cognitive science as an external representation and it aims to enhance user cognition. As Ware [War00] says "*Most cognition is done as a kind of interaction with cognitive tools, pencils and paper, calculators and increasingly, computer based intellectual supports and information system...It occurs as a process in systems containing many people and many cognitive tools*". All around us, we find visual representations which use the external world to enhance cognition and allow converting an internal memory task into an external visual search [CMS99].

Moody [Moo06b] introduces a model of the human graphical information processing system which is represented in Figure 1.2. This model decomposes the human graphical information processing into a series of processing stages:

- **Perceptual discrimination** extracts very fast low-level properties that are detected serially and in parallel from the visual scene to parse it into separate elements. This first stage is at subconscious level and is largely independent of what we choose to attend to [War00].
- **Perceptual configuration** groups the elements (from the *Perceptual Discrimination*) together by dividing the visual field into regions and patterns. During this

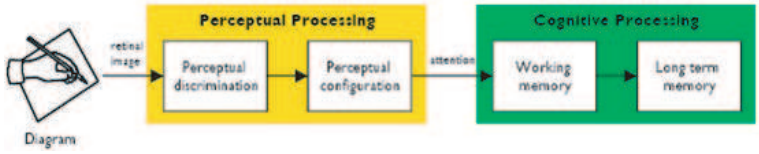

FIGURE 1.2: Model of Graphical Information Processing [BFG+02]

stage we can make mistakes if incorrect information is unintentionally encoded into the image [Moo06b].

- **Working Memory** (WM) brings the previously processed image for the active processing under the conscious control of the viewer's attention. WM is a temporary storage with very limited capacity and duration (e.g. cache memory) which synchronises rapid perceptual processes with slower cognitive processes. This stage can cause bottle-neck in graphical information processing [Moo06b].
- **Long term memory** (LTM) is quite contrary of WM. LTM is "*a permanent storage area with unlimited capacity and duration*", but is relatively slow (e.g. hard drive). Extracted and stored information in WM is integrated with knowledge from LTM. There are two types of prior knowledge relevant to diagram understanding: *Domain Knowledge* (knowledge about the represented domain) and *Notational Knowledge* (knowledge about the diagramming notation) [Moo06b].

This model shows some benefits if we design properly a visual representation because the perceptual processing time can decrease and so this representation will be more rapidly load on working memory in order to start the interpretation process [SR96].

1.2.1.2 Human Factors

Previously, we conclude that visualisation is a cognitive processing support to amplify cognition. We have seen that visualisation address this cognition amplification if we design properly with adapted visual representations. But a suitable design is not sufficient because, during the visualisation using, human factors introduce subjectivity. Because of visualisation is highly subjective, two different people could interpret the same visualisation in different ways. Moreover, a user could find that a specific visualisation is a great assistance to perform his job while another could find it unusable because it very hard to use.

Melanie Tory and Torsten Möller [TM04] strengthen the importance of the subjectivity in visualisation, saying "*how people perceive and interact with a visualization tool can strongly influence their understanding of the data as well as the system's usefulness. Human factors therefore contribute significantly to the visualization process and should play an important role in the design and evaluation of visualization tools*". Therefore,

the visualisation effectiveness depends on perception, cognition, and the users' specific tasks and goals [TM04].

In conclusion, during a work in the visualisation area we have to be taken into consideration the needs of the end users because they have a direct impact on the visualisation effectiveness.

1.2.2 Visualisation Techniques

Visualisation is derived into visual techniques which are implemented on tools in order to satisfy the end users needs. There are many visualisation techniques but we decide to present two of them which seem interesting for SPLE, i.e. focus-context and graphs, because most of current the Software Product Line Engineering tools (SPLE-tools) use these techniques.

1.2.2.1 Focus + Context

The Focus-Context concept describes the problem to find details in a larger context [War00]. We introduce some techniques which use this concept in order to address this problem. Card *et al.* [CMS99] define Focus + Context as follows.

Definition 1.8. Focus+Context starts from three premises: First, the user needs both overview (context) and detail information (focus) simultaneously. Second, information needed in the overview may be different from that needed in detail. Third, these two types of information can be combined within a single (dynamic) display, much as in human vision.

A well-known focus-context technique is *Pan and Zoom*. With this technique, users can use zooming and panning on any diagram to adjust it with the view scale [SZG+96]. However, the problem with this technique is "*when users are zoomed out for orientation, there is not enough detail to do any real work. When they are zoomed in sufficiently to see detail, context is lost*" [SZG+96]. It is not a real focus-context technique but it is an important component in any graphical system.

Another technique is the *Map-View* (or overview). This strategy uses a main view to show the details of the map and uses a miniature view to display an overview of the map with the user's current position in the map [BI90]. Beard *et al.* [BI90] conclude "*map windows significantly improved user performance [in locating an item on a binary tree]*".

The principle of *Fisheye View* (see Figure 1.3) is that "*in many contexts, humans often represent their own "neighbourhood" in great detail, yet only major landmarks further*

away" [Fur86]. People prefer to know local events than to have details on events in further area (for instance, the New Yorker's View of the United States [Fur86]). From the formalising generalised fisheye views, Furnas [Fur86] give the following algorithm:

$$DOIfisheye(x\,|.=y) = API(x) - D(x,y)$$

"*where DOIfisheye is, according to the fisheye model, the user's Degree of Interest in a point, x, given that the current point of focus is y, API(x) is the global A Priori Importance of x and D(x,y) is the distance between x and the current point y*" [Fur86].

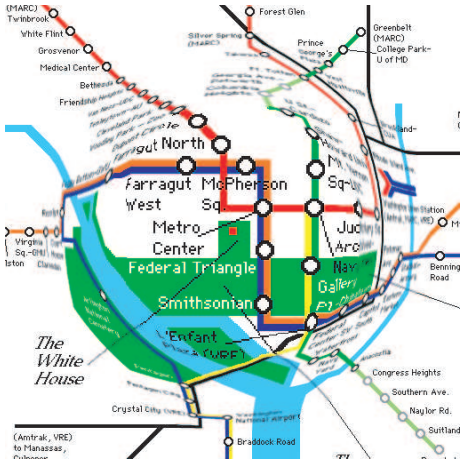

FIGURE 1.3: Map of Washington D.C.

1.2.2.2 Graphs

Graphs are useful to show data's or concepts where the relations are important. An example of a graph is shown in Figure 1.4. Graphs make explicit the entities (nodes in Figure 1.4) and their relationships (links in Figure 1.4). Moreover it is useful for both comprehension and exploration of the data.

However, Herman *et al.* [HMM00] make a point by saying "*The size of the graph to view is a key issue in graph visualization. Large graphs pose several difficult problems. If the number of elements is large it can compromise performance or even reach the limits of the viewing platform. Even if it is possible to layout and display all the elements, the issue of viewability or usability arises, because it will become impossible to discern between nodes and edges*".

By reason of that, many researches introduce new graph concepts (for more details see [HMM00]) of which the shared goal is to maximise the used space by a graph. As we can notice, a basic hierarchical tree layout which is a graph type (Figure 1.5) does not use efficiently the space and so create an unusable graph.

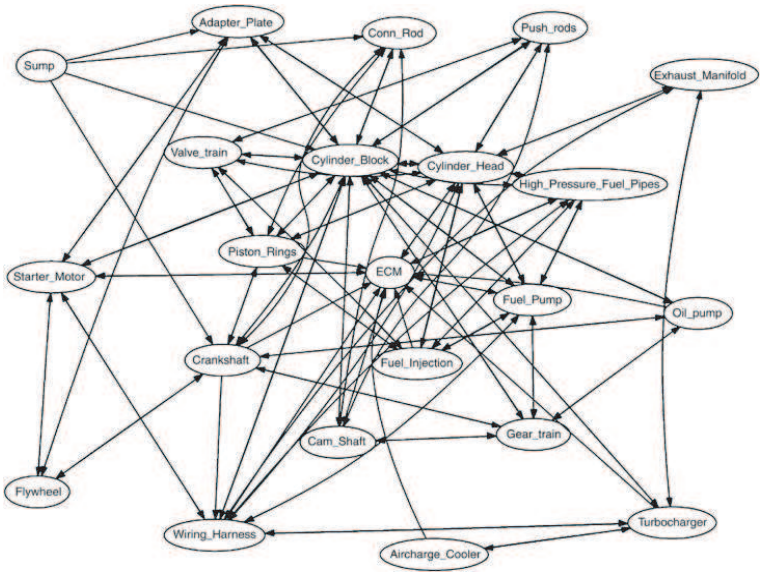

FIGURE 1.4: An Example of a graph

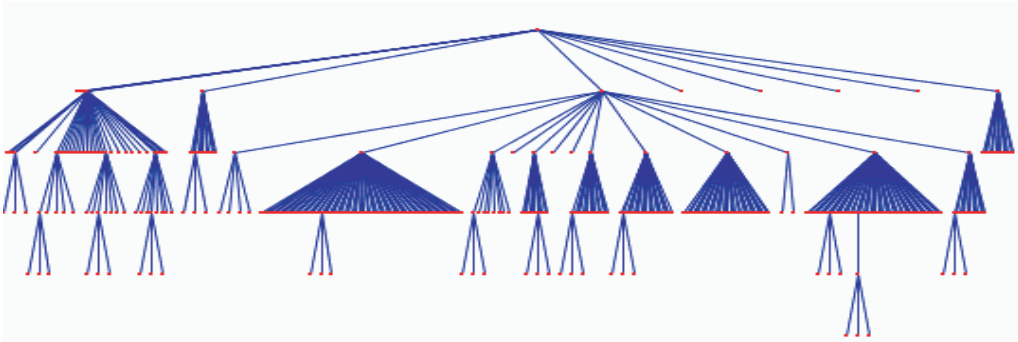

FIGURE 1.5: A tree layout for a moderately large graph [HMM00]

1.2.3 Visualisation in SPLE

Visualisation is still a non-explored area of SPLE. This is the main reason that we were affected to this work in our thesis and that we have realised our internship at the LERO which is a kind of pioneer in this area. The current performed researches are mainly focused on the Product Derivation process (see section 1.1.3.2). This is why we present exclusively the visualisation techniques that support this task.

To realise the Product Derivation process, we have to display the SPL to the users. But "*Industrial size product lines can easily incorporate thousands of variation points and configuration parameters for product customization*" [NOH+07]. So, it is important to have an appropriate visualisation in order to display this big variability and therefore to realise the product derivation tasks with a little effort.

Visualisation can be described as "*adjustable mappings from data to visual form*" [CMS99]. Nestor *et al.* [NOH+07] present a visualisation reference model which illustrates these

mappings for variability visualisation in the context of SPLs. This model is shown in Figure 1.5 and shows a series of data transformation from raw data to human perceiver (arrows from left to right). The under arrows indicates the adjustment of these transformations by user-operated controls and the dashed arrows indicates that all tasks are informed by the SPL scenarios (SPL scenarios are given in more details in [NOH+07]).

Firstly *Data Transformations* maps raw *SPL Data* (i.e., data about the SPL artefacts, their variability, and the dependencies among them) into *Data Tables* in order to achieve a set of relations that are more structured than the original data and thus easier to map to visual forms. Then *Visual Mapping* (Note that *Visual Mapping* preserves the data and that it can be perceived well by the human) transforms *Data Tables* into *Visual Structures* (combination between spatial substrates (e.g., nominal or ordinal axis), marks (points, lines, areas, volumes), and graphical properties (e.g., colour, texture or intensity)) in order to encode information. The last transformation, *View Transformations*, creates *Views* of the *Visual Structures* by specifying graphical parameters such as position, scaling, and clipping in order to interactively modify and augment Visual Structures to turn static presentations into visualisations. Finally, *Human Interaction* enables for the user to control parameters of these transformations. [NOH+07]

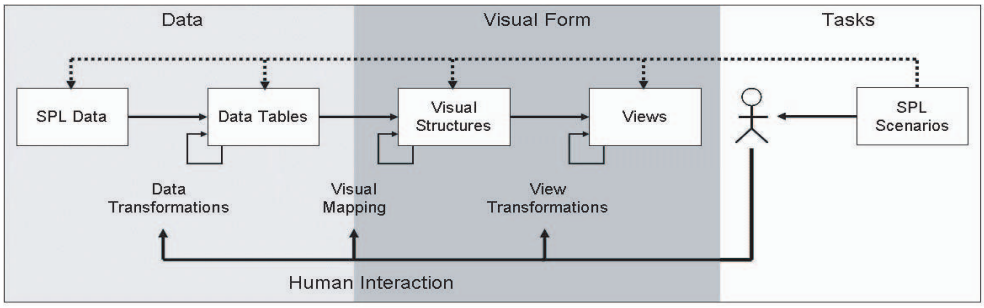

FIGURE 1.6: Reference model for variability visualisation [NOH+07]

Nestor *et al.* [NOH+07] conclude that "*There are many varieties of visualisation that could be applied in a software product line context. However, for hierarchical data (which includes computer programs) various graphs have proven useful*".

Chapter 2

Framework for SPLE-Tools Evaluation

SPLE is a new discipline of software engineering which becomes more and more important in the industry because it permits to realise significant improvements in time-to-market, cost, productivity, and system quality [NOH+07]. Thereby, SPLE-tools were developed by the commercial and research areas to support tasks of this new domain. These tasks include particularly construction of the Reusable Artefacts of the SPL, i.e. Domain Engineering, and derivation of a specific SPL product,i.e. Application Engineering (see section 1.1.3.2).

As we said in the introduction, our main motivation is to propose a possible solution to support visualisation in SPLE-tools. The first step to realise this task is to collect SPLE-tools functions. Accordingly, we need methods which could help us to collect these functions. "*An organisation needs to have an instrument which would help to analyse and compare the available tools. Such an instrument could be defined as an evaluation framework which provides understanding of the artefact quality, its application and guides through the evaluation process*" [Mat05]. The purpose of this chapter is to adopt a *top-down* approach by building an evaluation framework which may be use to find a non-exhaustive functions list for SPLE-tools. This framework could be used to compare the SPLE-tools to know how they should be improved.

The remainder of this chapter is organised as follows: Section 2.1 explains the steps of the evaluation framework construction and shows the framework with its evaluation system. Section 2.2 applies the framework on some commercial and research tools i.e. Gears (BigLever) [Homc], pure::variants [Beu03, psGH], Feature Modeling Plug-in (FMP) [AC04, iH], XFeature [Homf] and VISIT-FC (Lero tool) [BNP+07, Pre07] and we gives statistics, strengths and weaknesses for each of them. Finally, Section 2.3 concludes this chapter.

2.1 Evaluation Framework

As says Matulevicius [Mat05]: "*There is a general belief that if the process is of high quality it produces a high quality product. In this sense, it is also possible to make a hypothesis, that if a high-quality tool evaluation process is maintained, its output - the selected RE-tool*[1] *- will help to prepare high-quality requirements specifications*". If we apply this statement to the SPL context, the choice of the framework approach seems to be a significant practice to reuse in order to choose an appropriate tool according the users specific needs.

2.1.1 Guidelines for framework construction

To construct an evaluation framework, we proceed in two steps. First we create an appropriate framework skeleton to organise the framework structure. Then we determine how to fill the skeleton with functions which fit better with the evaluation context. In order to explain these two steps, we will give some approaches and examples for the skeleton construction and the skeleton filling.

2.1.1.1 Skeleton Construction

We have focused our attention on three relevant framework skeletons because these skeleton are created by their authors for the same purpose as this chapter. Firstly Nikiforova and Sukovskis [NS02] present a framework for evaluation of CASE modelling tools which includes the following functions: usability (e.g., the usage of the tools is simple, flexible printing, browser window), functionality (e.g., model relationship analysis, document generation, and generation of comments), method and language support (e.g., object orientation support, and UML support), integration with other software (e.g., Web technology support, integration with MS Office, and integration with DBMS).

Secondly, Post and Kagan [PK00] present a market-based approach which categorises their evaluation in five major criteria: graphics (e.g., ease of changing class relationships, ability to search for diagrams that use a particular object, and ease of look ups for existing definition), teamwork (e.g., version control, multi-user locking, revision history, multi-user access and data dictionary), prototyping (e.g. ability to merge the modified code with existing models, code generator, programming language support, ability to generate code based on models, and inclusion of comments and description), general features (e.g., vendor longevity, vendor stability, vendor support, quality of on-line help facilities, and quality of documentation) and object oriented (e.g. support for class

[1] "*Requirements engineering (RE) tools are software tools which provide automated assistance during the RE process*" [Mat05]. For more details about requirements engineering see [GHM08]

hierarchies and inheritance, support for encapsulation, support for polymorphism, and support for meta-classes) features.

Finally, Gallagher *et al.* [GHM08] present a qualitative framework for the assessment of software architecture visualisation tools. This framework has seven key areas for describing software architecture visualisation: Static Representation (e.g., source code, test plans, data dictionaries, and other documentation.), Dynamic Representation (i.e., the architectural information that can be extracted during runtime. For instance, inheritance and polymorphism.), Views, Navigation and Interaction (e.g., for navigation: panning, zooming, bookmarking, and rotating. For interaction: selection, deletion, creation, modification), Task Support (i.e., the ability of the visualisation to support stakeholders while they are developing and understanding the software architecture), Implementation, and Representation Quality (i.e., capability of the visualisation to adequately represent the software architecture).

2.1.1.2 Skeleton Filling

After defining each main framework sections, we have to define the specific functions to fill these sections. We have conserved three approaches to determine functions for a tool evaluation framework in a specific domain which allow filling the skeleton. First, expert-based method elicits and gathers the tool functions for the framework close to researchers and practitioners of the specific domain [NS02, HD97]. The second one realises market studies in order to find via investigations the most common trends of the specific domain [PK00]. The third approach is based on the researchers' personal experience [HKWB04]. Note that "*all the framework definition approaches are based on the techniques used for the requirements elicitation. The major limitation of all of them is availability of the stakeholders, willingness and interest in method or framework definition*"[Mat05].

2.1.1.3 GQM paradigm

The **Goal/Metric/Question** (GQM) paradigm [BCR94] is a combination of skeleton construction and filling. The QGM paradigm is used by Gallagher *et al.* [GHM08] to identify the tool functions and also the sections for the framework structure. Basili *et al.* [BCR94] define three levels in the GQM paradigm: *Conceptual level* (GOAL) where a goal is defined for an object, for a variety of reasons, with respect to various models of quality, from various points of view, relative to a particular environment. *Operational level* (QUESTION) where a set of questions is used to characterise the way the assessment/achievement of a specific goal is going to be performed based on some characterizing model. Questions try to characterize the object of measurement (product, process, resource) with respect to a selected quality issue and to determine its quality

from the selected viewpoint. *Quantitative level* (METRIC): A set of data is associated with every question in order to answer it in a quantitative way.

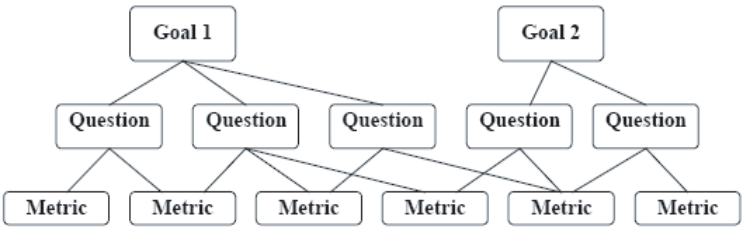

FIGURE 2.1: Illustration of QGM paradigm [BCR94]

2.1.2 Framework Construction Steps

The framework construction is based on the previous guidelines:

- **First step** is to elaborate a structure for the framework, i.e. the skeleton.
- **Second step** is to elicit the functions in order to fill this skeleton.

SPLE is a difficult task due to the youthfulness of the discipline [DSB05]. Indeed, this characteristic brings a lack of formal scoping in the domain and thus hardness to draw up skeleton for the framework. In order to help us in the framework construction, we have used the GQM paradigm to build the framework skeleton and to find the appropriate functions.

During the **first step**, based on the SPLE literature, we have set up a list of general goals. This list includes two sections, i.e. *domain* and *application engineering*. This last one is implicitly split into two sub-sections, i.e. a *derivation* sub-section (it is linked to the instantiation of the overall architecture; see section 1.1.3.2) and a *product construction* sub-section (it is linked to the Reusable Artefacts selection to build the product, see section 1.1.3.2). Then we have identified sub-goals which are common to all main sections. This leads to define three categories inside sections (note that all categories are not necessary present in every section because some of them don't require as much.):

- **Interactivity** means to perform an action in order to satisfy a purpose.
- **Automatic management** means that some SPL aspects like constraints that are automatically managed by the SPLE-tools.
- **Visual facilities** means visual information in order to perform a further action.

During the **second step**, we have found questions for each category thanks to investigation into commercial and research tools, to our personal experience which is based on our readings and on an elicitation close to Lero's researchers. These questions investigations conduct to identify two other main sections, i.e. *common functions* which are tool functions present in all steps of domain and application engineering and *quality requirements* (according to ISO 9126) especially pertinent for SPLE-tools. From these questions we conclude a functions list to evaluate each category. Here is an example of the application of GQM paradigm:

Goal (Sub-goal)	**Questions**
Domain engineering (Interactivity)	The tool allows adding feature?
	The tool allows the parametrization of features?
	The tool allows adding alternative choice?
	The tool allows adding cardinality?
	The tool allows adding require constraint?
	The tool allows adding exclude constraint?
	The tool allows adding mandatory constraint?

TABLE 2.1: GQM application to domain engineering interactivity

In order to reduce the framework size we have gathered questions into some themes. In this example, the two first questions have been gathered in the *Feature structure* function, the two next questions have been gathered in *Alternative choices* function and the last three questions have been gathered in *Constraints* function.

2.1.3 Framework Presentation

In the remainder, we present the complete framework (Table 2.2) which respects the skeleton built in the framework construction step. Each function is described in order to understand their meaning.

2.1.4 Evaluation System

Following the GQM paradigm, we create a metric in order to answer to the questions in a quantitative way. We have chosen a metric system as we can see in Gallagher *et al.* [GHM08]. This metric (Table 2.3) is a quotation included between absent (0) and excellent (5) feature.

Our weight system is based on a point calculus. On the top, a SPLE-tool can have 1000 points. These 1000 points are distributed among the five main sections as follow. The *domain and application engineering* sections get respectively 300 points and 400 points because these are large essential activities for SPLE-tools. Then the *common functions* section gets 200 points because it contains some important functions for SPLE-tools.

Section	Categories	Functions Descriptions
Domain Engineering	**Interactivity**	**Feature structure**. Actions to manage a feature structure (add feature, parameterise feature, etc.), feature cardinalities to clone features, feature attributes, etc.
		Alternative choice. Actions to manage alternative choices (add alternative choice, etc.), alternative choice cardinalities, etc.
		Constraints. Actions to manage feature constraints (mandatory, optional, etc.), constraints between features (require, exclude, etc.), etc.
		Multiple feature models. Creation of multiple feature models in the same workspace.
		Product line hierarchy. Creation of hierarchy to manage a set of products lines in a structural way.
		Save/Load a feature model. Get a feature model that we have saved before.
	Automatic Management	**Editor Generation**. Generate an editor based on a meta-model and a display-model to draw a feature model.
		Model Validation. Validate the current feature model according to a meta-model.
		Constraints cancellation. When a feature is removed from a model, its linked constraints are automatically cancelled.
	Visual Facilities	**Commonalities**. See the commonalities between different features.
Application Engineering: Derivation	**Interactivity**	**Actions on a feature**. Set of actions to perform a derivation like feature selection, feature elimination, feature attribute instantiation, etc.
		Load/save a configuration. Get a product configuration that we have saved before.
		Interactive process. Process to conduct the product derivation by reporting problems and automatic actions due to constraints.
		Specific adaptation. Save the need of a specific adaptation of a feature to prevent it later during the implementation.
		Chose a configuration. Get a product configuration within a set of pre-set configurations.
	Automatic Management	**Constraints**. Automatic check of all constraints (requires, excludes, etc.) after all user actions (feature selection, etc.).
		Block selection. Block the selection of a non selectable feature because for instance it is a pre-set feature for all configurations.
		XML file. Mechanisms to generate XML files in order to be reuse further by other tools.

	Visual Facilities	**Visual mechanisms**. Intuitive mechanisms to see problems during the derivation process.
		Configuration number. See the number of possible remaining product configurations.
		Auto-selected feature. See difference between auto-selected feature and manually selected feature.
Application Engineering: Product Construction	**Automatic Management**	**Assembling**. Different artefacts are put together automatically to build the final products according to the product configuration.
		Software assets. Automatic products management if changes appear in software assets.
		Adaptation. Automatic adaptations of the feature model based on the number of specific adaptations realised for features of this model.
		Prevention. Automatic prevention of specific adaptations of features in a product configuration.
Common Functions	**Interactivity**	**Feature Information**. Manage information about features (name, description, etc.).
		Dependencies. Manage dependencies between features and development artefacts.
		Feature model views. Choose from different views in order to display the feature model according to the user needs.
		Search. Mechanism to search features in the model.
		Software assets change. Know the impact of a software assets change.
		Undo & redo actions. Undo and redo actions is possible.
	Visual Facilities	**Graphical notation**. Clear and complete graphical notation which supports feature models.
		Context. Visualisation mechanisms which do not lose the context of a large feature model.
		Big information quantity. Visualisation mechanisms to display a big information quantity in a very small space.
		Feature information. See information about a feature.
Quality requirements	**Interoperability**	The capability of the software to interact with one or more specified systems.
	Portability	The capability of software to be transferred from one environment to another.
	Co-existence	The capability of the software to co-exist with other independent software in a common environment sharing common resources.

Table 2.2: Framework for SPLE-tools evaluation

Quotation	Meaning
0	The function is absent or unknown
1	The function is present but very poor
2	The function is present but bad
3	The function is present but middling
4	The function is present and good
5	The function is present and excellent

TABLE 2.3: The metric and its meaning

Finally the *quality requirements* section gets 100 points because it is not really important for the SPL management but it can be an extra-value for SPLE-tools. Each section weight is assigned to their functions according to their weight in the SPL management. Note that this weight system is totally subjective and may be changed according to the evaluation team needs.

The metrics and the weight system enable to compute a value for each function of SPLE-tools framework. The formula in Figure 2.4 calculates the value for each quoted function where Valuefunction represents the global quote of the function for the tool, Quotefunction represents the quote that user gives for the function and Weightfunction represents the balancing of the function in the tool. The tool score is got by adding all the function values.

$$Value_{function} = \left(\frac{Quote_{function}}{5}\right) \times Weight_{function}$$

2.2 Tools evaluation

We have applied the framework to some commercial and research tools i.e. Gears (BigLever) [Homc], pure::variants [Beu03, psGH], Feature Modeling Plug-in (FMP) [AC04, iH], XFeature [Homf] and VISIT-FC (LERO tool) [BNP+07, Pre07]. We have tried to evaluate all tools on the same equal footing. Yet some tools are not accessible because they are not free of charge. So we have decided to use all available documentations (white papers, tools demonstrations, etc.) because tools documentations are as much important. The evaluation is totally subjective and could lead to opposite results if it was realised by other evaluators. The detailed tools evaluation is available in the Appendix A.3 where we have divided some functions into sub-functions in order to be more accurate in the evaluation. Depending on their needs, any evaluation teams may set their own sub-functions list for any functions of the SPLE-tools framework.

We present in the Appendix A.1 our detailed interpretation of the tools evaluation results. In this appendix we find for each tool a strengths and weaknesses report and a diagram looking the quota between missing and present functions for each main sections

of the SPLE-tools framework. To summarise the main results of the tools evaluation, Figure 2.5 shows the score for each evaluated tool and therefore shows the tool podium where the winner is pure::variants.

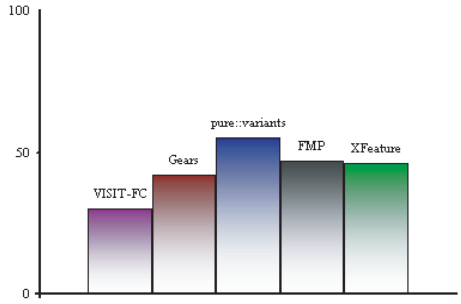

FIGURE 2.2: The tools score

2.3 Conclusion

In this chapter we have tried, through a top-down approach, to find required functions of SPLE-tools in order to provide a good base in the construction of an abstract solution that could encompass some of these functions. We have chosen to build an evaluation framework which seemed to be an appropriate method to find most of functions. Our evaluation framework is not proposed as the best method to find all SPLE-tools functions and is not proposed as the best formal representation for SPLE-tools functions. But this evaluation framework is necessary for discussion about these required functions. Of course this framework is an open question and a subject of future research.

Subsequently, we have applied the framework in practice on five tools coming from the commercial and research areas. Based on our tool investigation, a global observation is that SPLE-tools are disparate and contain common functions as well as various. In this context, it is not easy to characterise and to compare them.

Chapter 3

Challenges Identification

Over the past few years, most of the research efforts are focused on methodological supports for Domain Engineering in order to facilitate the tasks of the product derivation process. However, most of the approaches have not bear fruit and consequently there is a lack of methodological supports for Application Engineering. This situation causes troubles in organisations and these ones fail to exploit the full benefits of SPLs [DSB05]. Therefore, there is a strong need to bring support for Application Engineering. Thereby, we leave Domain Engineering to only focus on the product derivation process in the remainder of this thesis.

Based on our tool investigation, we have seen in chapter 2 that the SPLE-tools are disparate and contain common functionalities as well as various. This tool investigation was a good basis to know the main functions required by a SPLE-tool. However it is not enough to fulfil our motivation of providing a solution to support visualisation in SPLE-tools. Consequently, the purpose of this chapter is to find some tool challenges that will be used to conduct our visual solution in the best way. These tool challenges coupled with the principal tool functionalities investigated in chapter 2 will give us a complete basis to realise our motivation. Naturally, as we said before, we are now focusing on the product derivation process. Thus, the tool challenges and the future visual solution will be focused on this process as well. Furthermore, only the functionalities investigated in chapter 2 regarding this process will be considered.

Naturally, we have to consider the requirements of SPLE-tools users in order to provide a useful and practicable solution. As a result, to find good tool challenges requires finding challenges for the product derivation process. To realise this task, we adopt a bottom-up approach by analysing product derivation problems, i.e. industry product derivation problems and problems of our personal experience in the product derivation process. Then we try to draw from that product derivation challenges. As says the SWEBOK [ABDM01], it is widely acknowledged within the software industry that software engineering projects are critically vulnerable when the requirements collecting activities are

performed poorly. Thus, we think that this way of doing will help us to guide our visual solution in the good direction.

For synthesizing concerns we outline the product derivation framework presented by O'Leary *et al.* [OBTR07] which introduces a best practice approach for product derivation by mixing most of the current practices. For each task of the framework we present the associated product derivation problems by putting them into the context of the framework.

The remainder of this chapter is organised as follows: Section 3.1 outlines the product derivation framework presented by O'Leary *et al.* [OBTR07]. Section 3.2 acquaints, for each task of the framework, the associated product derivation problems putted into the context of this framework. Section 3.3 gives, based on the section 3.2, the product derivation challenges and the tool challenges required to realise these product derivation challenges. Finally, Section 3.4 concludes the chapter.

3.1 Product Derivation Framework

This section presents the product derivation framework introduced by O'Leary *et al.* [OBTR07]. This framework was realised in order to have a best practice approach for product derivation and is based on the results of an extensive literature review, lengthy discussions with SPL practitioners and researchers, and reviews of industrial product derivation practices. Eventually, this framework is expected to assist organisations in using a structured approach for product derivation activities. However, it is not in the perspective of providing a best practice approach that we choose to introduce this framework in this chapter but more for synthesising concerns. Indeed, as we have said in chapter 1, the product derivation process is very disparate in practice and we think that this framework is appropriated to have a synthetic view of the product derivation process which may be used as basis to present the product derivation problems.

As we can see in Figure 3.1, the framework has four main tasks:

- **Impact Analysis** where the *product specific requirements* are derived based on the *customer requirements* and negotiation with the platform team. This task is called "Impact Analysis" because it is crucial for the effectiveness of the product derivation process that, during negotiation with the platform team, the impact of implementing *customer requirements* which cannot be satisfied by a configuration of the *platform assets* is well know.
- **Reusability Analysis** where a *partial product configuration* is created based on the *product specific requirements* and by using the available *platform assets*. This task is called "Reusability Analysis" because it is crucial for the effectiveness of

the product derivation process that the possibility of reusing an existing *product configuration* or not is well know.

- **Component Development and Adaptation** where new components are developed (if required) and existing components are adapted to satisfy requirements which could not be achieved through a *configuration of the platform assets.*
- **Product Integration and Validation** where the *partial product configuration* is integrated with the *developed and adapted components.* The *integrated configuration* is then validated through *integration and system testing.*

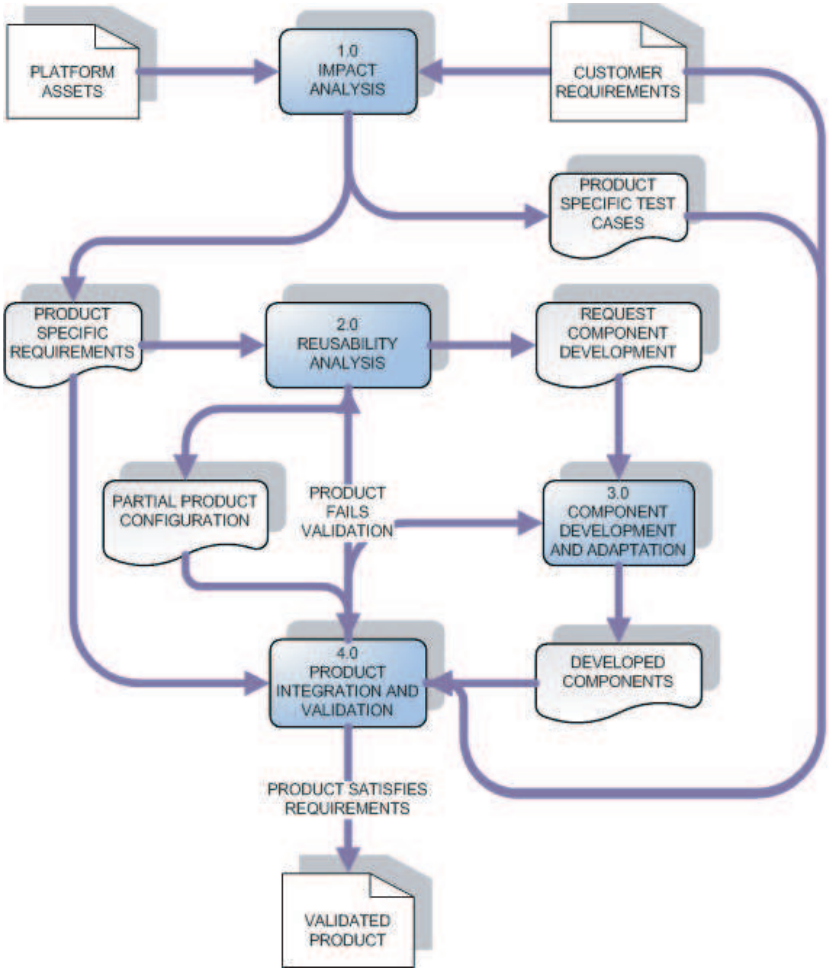

FIGURE 3.1: Product derivation framework overview [OBTR07]

3.1.1 Impact analysis

In this task (see Figure 3.2) the customer requirements are mapped to platform features. The product team determines the list of the customer requirements which can be satisfied by a configuration of the platform assets. Customer requirements, which cannot be satisfied by existing assets, are negotiated with the customer and platform architect.

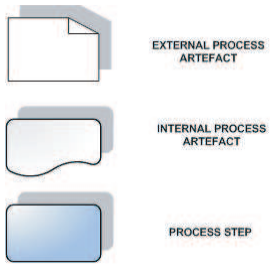

FIGURE 3.2: Product derivation framework legend [OBTR07]

Customer negotiation is an important and critical aspect of product derivation. The trade-off here is to meet ideally all of customer's needs while retaining the profitability of the platform assets for the whole product line. Time-to-market requirements can cause the product team to make their own product-specific modifications to core assets.

The satisfied customer requirements and the negotiated requirements are merged to form the product specific requirements. These product specific requirements are used to create the product specific test cases. The product specific test cases are used during system testing in Product Integration and Validation and assist the product team in the verification of requirements. The product team uses the platform test cases artefacts as a basis for the creation of the product specific test cases.

3.1.2 Reusability analysis

The main goal of the Reusability Analysis (see Figure 3.3) is to create a partial product configuration that maximises the benefits of the platform artefacts and minimises the amount of product specific development required. Based on the product specific requirements, the platform team identifies if an old configuration can be reused.

This case is especially viable where a large system is developed for repeat customers, i.e. customers who have purchased similar types of systems before. Typically, repeat customers desire new functionality on top of the functionality they ordered for a previous product. This way of doing is particularly interesting because it allows speed up the development process by choosing a previously tested solution especially in instances when two or more configurations can be used. In situations where no appropriate existing configuration can be selected, the product team must derive a new configuration from a subset of the overall platform architecture. Deriving a new configuration includes selecting components from the collection of platform components and setting the parameters of these components as well.

In both situations, i.e. configuration selection and new configuration derivation, a base configuration which represents a subset of platform components is produced. At this stage, it is still possible that some requirements are not satisfied by the base configuration. As consequences the product team selects components from the collection of

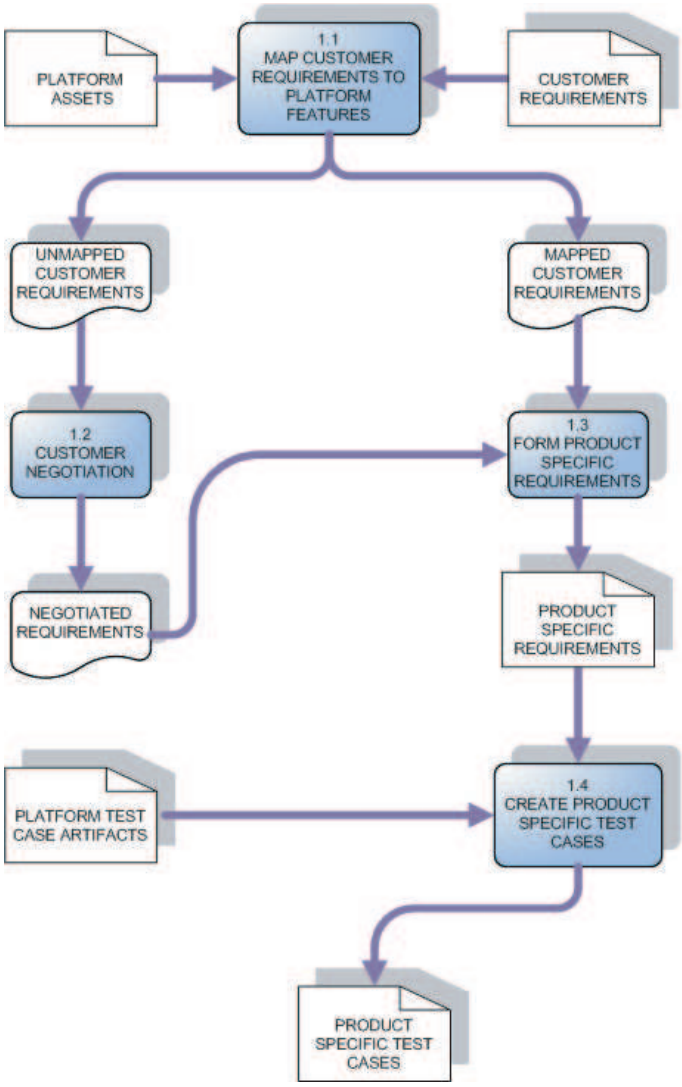

FIGURE 3.3: Impact analysis of the product derivation framework

platform components for, addition to or replacement of, components in the base product configuration (like in the derivation of a new configuration that includes setting component parameters as well). Components selection is particularly delicate because the fact that a component fits in the configuration depends on the fact whether the component correctly interacts with the other components in the configuration and no dependencies or constraints are violated.

A partial configuration is now created and the product team identifies which product specific requirements are satisfied by this partial configuration. It is possible again that there are requirements that are not respected by the partial configuration. This means that these requirements are not available in the platform assets and need to be developed from scratch. The Product Team is responsible for the development of new components and adaptation of existing components in order to conform to the new components.

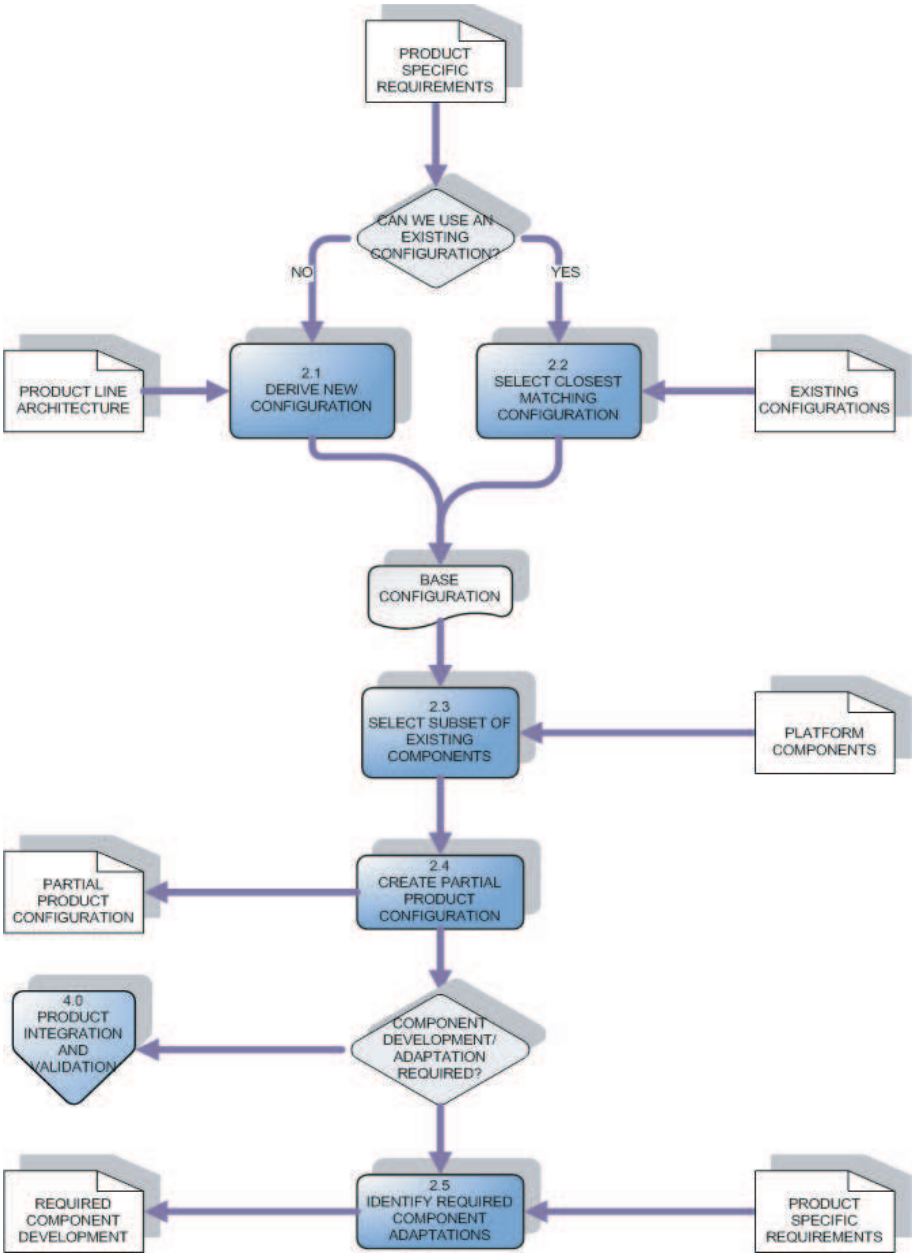

FIGURE 3.4: Reusability analysis of the product derivation framework

3.1.3 Component development and adaptation

In the Component Development and Adaptation task (see Figure 3.4) the product team facilitates the achievement of requirements which could not be satisfied by the partial product configuration by adapting existing and developing new components. The decision of whether the required component development or adaptation will result in product-specific code or in adaptation of the entire product line is determined through a Change Control Board. The Change Control Board is usually constituted of members of the product team and the platform team. Scoping new development is a difficult task. Practical arguments such as time-to-market and short term cost frequently cause

scoping solutions to be selected that are neither optimal for the product itself nor for the product line as a whole. While platform development must provide a consistently high-quality platform, product development must meet delivery dates and customer requirements. So, with every software product line development you must decide whether to integrate a given requirement into the platform or into an individual product only.

If the Change Control Board decides that the component development should occur at the platform level then the platform team has to adapt or develop new shared artefacts and release a new version of the platform. Based on the new platform, the product team must repeat Reusability Analysis for the products under consideration. If the development or adaptation is designated to be product-specific then it is the responsibility of the product development team to implement the required component changes at the product level.

When a component is built or adapted, initial or tailored versions of a component will need to be tested rigorously through unit testing. Conventional unit test methods must be utilised as no product line specific methods have been developed so far.

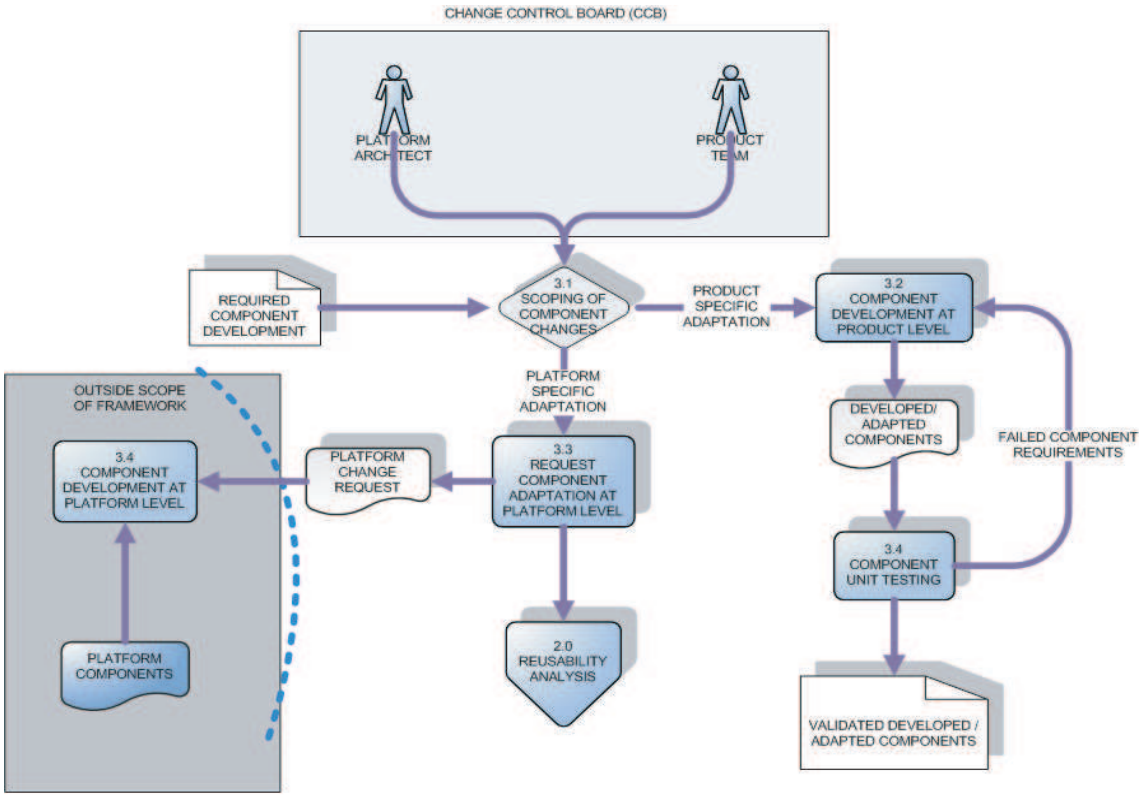

FIGURE 3.5: Component development and adaptation of product derivation framework

3.1.4 Product integration and validation

In Product Integration and Validation (see Figure 3.5), a final integrated product is created from the partial product configuration and the developed components. The product is validated through integration and system testing.

During product integration, the newly developed/adapted components are integrated with the partial product configuration. The Product Team integrates the developed/adapted components and the partial product configuration by writing sufficient "glue" code to interface with the components.

The integrated product configuration must undergo product validation. Before product validation can begin the product team must confirm that no changes in the customer requirements have occurred. If the customer requirements have changed, the product team must return to the Impact Analysis and perform a second iteration of the framework to reflect the new requirements in the product. If the customer requirements are consistent then the product team begins product validation. During product validation the product is checked for the consistency and correctness of the component configuration in integration testing and for compliance with the product specific requirements in system testing.

Due to the variability defined in the platform assets, completely testing the platform assets is impossible except for trivial cases. Integration testing validates the platform assets for this particular configuration. The integration tests should reuse platform test artefacts. This also ensures that no new errors appear due to the integration of core assets with product specific assets.

After integration testing, system testing is performed. System testing verifies if a product conforms to the product specific requirements. System test artefacts such as the product specific test cases are already derived from the product specific requirements in the Impact Analysis.

If the product fails integration testing or system testing then the current configuration may not provide the required functionality, or some of the selected components simply do not work together. In this case the product team should repeat either the Reusability Analysis task or the Components Development and Adaptation task depending on the required changes.

There are two main reasons why a product may fail integration or system testing. Firstly the requirements set may change or expand during product derivation, for example, if the organization uses a subset of the customer requirements to derive the initial configuration, or if the customer has new wishes for the product. Secondly, if the configuration may not completely provide the required functionality or some of the selected components simply do not work together at all.

If the product is validated through system and integration testing the process is complete and the customer product has been derived.

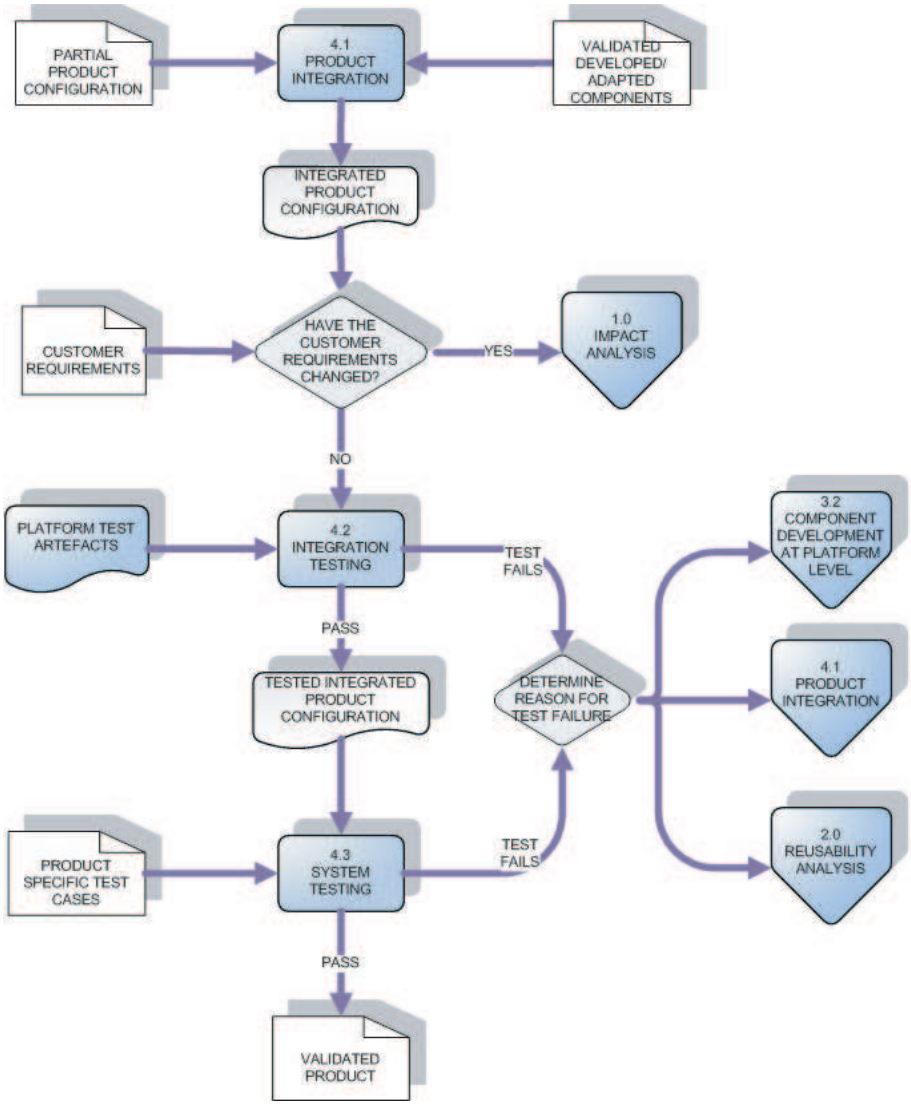

FIGURE 3.6: Product integration and validation of product derivation framework

3.2 Product Derivation Problems

Now that we have a synthetic view of the product derivation process, we present the main product derivation problems put in the context of the framework. The problems presented in this section come from a combination of our personal experience in the product process and of a review of the industry problems raised in a case study outlined by Deelstra *et al.* [DSB05] (a synthetic view of these problems can be seen in [DSB04] as well). This case study presents the problems of two societies, i.e. Robert Bosh GmbH and Thales Nederland B.V., which are large and relatively mature organisations that

develop complex software systems and face issues during product derivation that are comparable to other companies. In this sense, we think that the issues of these societies combined with our personal experience in the product derivation process will give a good picture of the product derivation problems to guide our research of product derivation challenges.

In the same vision of the product derivation framework presented before, we give a synthetic view of the product derivation issues by putting the mixed problems, i.e. the problems coming from our personal experience combined to the analysed industry problems of the case study, in the context of the framework. To present the contextualised problems, we have followed the categorisation given by Deelstra *et al.* [DSB05, DSB04] which seems the most logical to focus our attention on problem of particular SPLE concepts. There are three problem categories, i.e. *knowledge Externalisation*, *variability*, and *scoping and Evolution*.

- **Knowledge Externalisation**. "*The process of converting tacit knowledge to documented or formalized knowledge is referred to as externalization*" [NT95]. The problems of this category are associated with tacit knowledge and knowledge externalization.
- **Variability**. This category presents the product derivation problems which come from the variability of the SPL.
- **Scoping and Evolution**. There are several approaches in product derivation to identify the platform assets that should be made reusable in order to deal with the continuous evolution of a SPL, i.e. reactive product evolution and proactive product evolution [CN01]. The problems of this category are related to the using of these evolution approaches in product derivation.

Let us now describe the product derivation problems for each framework task (except for the production integration and validation where we have decided to not present problems because they are not linked to the product derivation process but more to the overall Software Engineering) classified according to the three categories mentioned above.

3.2.1 Impact analysis problems

Knowledge externalization

[KE1]. During the mapping of customer requirements to platform features, it is possible that requirements are considered like mapped while there are incompatible. This problem occurs when we choose incompatible requirements due to a bad documentation and especially a bad documentation of the dependencies between requirements.

Variability

[V1]. In industrial context, where there are hundreds or even thousands requirements, it becomes often difficult to do the mapping between the customer requirements and the platform features due to the very high cognitive complexity that is hard to manage for the human brain for. That leads to situations where the product team does not know what requirements are mapped or not. As consequences the product team must do much verification which could be avoided and that are expensive and time-consuming.

[V2]. A follow-up problem is that platform features are not really structured. This increases the cognitive complexity of the mapping activity.

Scoping and evolution

[SE1]. It is important to know the effort that will be required to change the platform assets in order to conform to the unmapped customer requirements while retaining the profitability of the platform assets for the whole product line. Unfortunately there are not industry mechanisms to do this activity. For time-to-market reasons, this causes, like we suggest in the description of the impact analysis, the product team to make their own product-specific modifications to core assets but sometimes in a none financially beneficial manner.

3.2.2 Reusability analysis problems

Knowledge externalization

[KE2]. During the derivation of a new configuration and the selection of components from the collection of platform components for, addition to or replacement of, components in the base product configuration, we remark that the selected components are often incompatibles. This is due to the fact that all compatibility aspects between these components are not externalised. Generally, this problem is observed in the Product Validation and Integration task and forces product engineers to return at the Reusability task in order to select other components. Consequently, this problem leads to wasted time.

[KE3]. Furthermore, we see that there are a large number of errors in component parameter settings due to large amount of parameters with implicit dependencies that are not externalised. In the same way of the previous one, this problem is generally observed in the Product Validation and Integration task and forces product engineers to return at the Reusability task in order to reset parameters leading to wasted time as well.

[KE4]. Finally, due to the apparent lack of externalising important tacit knowledge, we observe that there is an occurrence of the two previous problems [KE2 and KE3] in

successive projects. However, it is also important not to have an over explicit documentation because it decrease the traceability of information between successive projects.

Variability

[V3]. Similarly to the unmanageable number of requirements referred in the Impact Analysis, one problem of deriving a new configuration is the complexity of the SPL in terms of number of variation points and variants. This cognitive complexity causes the process of binding each variation points (see 1.1.3.2) to become unmanageable by individuals.

[V4]. Furthermore, another problem, which is already referred in the Impact Analysis and is a cause of the previous problem, is the fact that neither SPLs explicitly organised variation points. Thereby, product engineers have to deal with many variation points that are not all relevant for the product which is currently being derived. This problem increases the complexity of the binding activity.

Scoping and evolution

[SE2]. Generally, when changes in the platform artefacts are realised by a reactive manner, new versions of components are added to the platform artefacts. This hampers the components selection because product engineers have to find out which versions of the components can be connected together leading to wasted time.

3.2.3 Component development and adaptation problems

As we have seen before, this task manages the development of components and adaptation of existing components. This task is generally realised through a reactive or a proactive evolution. Thereby, even if it appears that the problems of this task are linked to a Knowledge Externalisation or a Variability problem, the nature the problem of this task are more linked to Scoping and Evolution problems. It is due to the fact that, in this task, Knowledge Externalisation and Variability problems are the consequences of Scoping and Evolution problems.

Scoping and evolution

[SE3]. As we said in the description of this task, the decision of whether the required component development or adaptation will result in product-specific code or in adaptation of the entire product line is determined through a Change Control Board. This decision is very hard to regulate in practice. Practical arguments such as time-to-market and short term cost cause frequently to select solutions that were neither optimal for the product itself nor for the product family as a whole from an engineering perspective.

3.3 Challenges

In the previous section we have presented many product derivation problems. These problems reflect the product engineer's needs that could help to fully benefit from a SPL approach. We have formalised these needs in the form of product derivation challenges.

Naturally these product derivation challenges need a support to be tackle. This support can be made through tools. As consequences, the product derivation challenges will be accompanied with tool challenges. These ones coupled with the principal tool functionalities investigated in chapter 2 will give us a complete basis to realise our motivation. We first describe the product derivation challenges and then the tool challenges.

3.3.1 Product derivation challenges

- Map the customer requirements with the platform features (e.g. the platform requirements) in an efficient way, keeping in mind the maximisation of the platform reusability and the customer satisfaction [KE1, V1, V2, and SE1].
- Select the platform components and set component parameters in an efficient way in order to avoid inconsistencies in configurations and occurrences of identical errors/developments in successive projects KE2, KE3, KE4, V3, V4, SE2, and SE3].
- Have an acceptable quality to quantity ratio documentation in order to avoid the traceability failure and the ambiguity of information [KE4].
- Have a structured and manageable set of platform features in order to reduce the cognitive complexity of the SPL [V1, V2, V3, and V4].
- Have an efficient scoping process in order to optimise the quality of the SPL [SE1, SE2, and SE3].

3.3.2 SPLE-Tool challenges

So as to tackle these industry challenges, the tool support must include:

- A relevant externalisation of the dependencies between all platform features including dependencies between their parameters and dependencies between their different available versions.
- A relevant externalisation of the platform feature information including information of their parameters and information about their different available versions.
- A relevant externalisation of information to facilitate the scoping activity.

- An interactive process for platform features selection (e.g. platform requirements), including selection of their different available versions, that shows the consequences of feature choices.
- A panel of relevant views in accordance with the relevance of the accomplished tasks (e.g. view for the mapping of the customer requirements).
- View mechanisms which reduce the complexity of the SPL (e.g. packaging system [PBvdL05]).

3.4 Conclusion

In order to fulfil our motivation of providing a solution to support visualisation in SPLE-tools, we have adopted a bottom-up approach by analysing product derivation problems, i.e. industry product derivation problems and problems of our personal experience in the product derivation process and by trying to draw from that product derivation challenges which represents the product engineers needs. Then, we have found tool challenges which address these product derivation challenges. In accordance with the SWEBOK [ABDM01] ideas, we have used this way of doing in order help us to guide our visual solution in the good direction.

However, we can criticise our problems analysis in the sense that we could use more case studies to support our product derivation challenges. It is principally for a lack of time that we have not extended our analysis. This can be the purpose of a further work in order to find many other problems that could help to refine or improve our product derivation challenges and consequently the tool challenges.

The tool challenges coupled with the principal tool functionalities investigated in chapter 2 now give us a complete basis to realise our motivation. An overall observation is that the SPLE-tools functions investigated in the previous chapter are already covering the tool challenges of this chapter except for the complexity tool challenge, i.e. View mechanisms which reduce the complexity of the SPL. Thereby, while trying respecting all tool challenges, the visual solution, that will be outlined in chapter 4, will mainly focused on the complexity problem.

Chapter 4

Addressing Of Challenges Using Visualisation

Keeping in mind our main motivation, i.e. to propose a solution to support visualisation in SPLE-tools, we will try to find a solution that uses visualisation to address most of SPL-tool challenges from chapter 3 and most of SPL-tool functions from chapter 2. Among the tool challenges, we focus mainly on the **complexity challenge** (see Section 3.3) because "*an important issue in this area is to overcome the problem of communicating information effectively in a high information density environment. Extracting information from representations of high variability structures can lead to information overload*" [NOH+07]. Furthermore, as we said in chapter 3, an overall observation is that the SPLE-tools functions investigated in chapter 2 are already covering the SPLE-tool challenges except the complexity challenge.

Nestor *et al.* [NOH+07] claim that hierarchical structures including, listings, outlines and tree diagrams (e.g. [CMS99, HMM00]) can help to manage and visualise information from a complex structure. But the big issue of the complexity in the visualisation is to display an amount of information in a limited space. In this way Nestor *et al.* [NOH+07] say that "*the presentation of hierarchical information can be improved even if the display space is limited*". Indeed it exists some visualisation techniques that allow to improve display space usage, e.g. Venn Diagrams and Tree-Maps [CMS99]. Furthermore, we can use clustering and semantic zoom on hierarchical structures to reduce the amount of information on display [NOH+07].

All of these techniques aim to resolve the visualisation problems of the complexity challenge. But, as seen in the chapter 3, the variability of some SPLs is so high that such variability becomes hard to manage. Such visual techniques are not very appropriated to resolve this problem because the complexity makes the visual techniques ineffective and complex to use. This statement is verified in Figure 4.1. Indeed the variability of this SPL is so high that it is not easy to manage and visualise it whereas Nestor *et al.* say

that this visual technique (tree structure) is appropriated for SPL. This example reflects exactly the *map shock* issue [Moo06a] and thus that becomes impossible to interact with this visualisation.

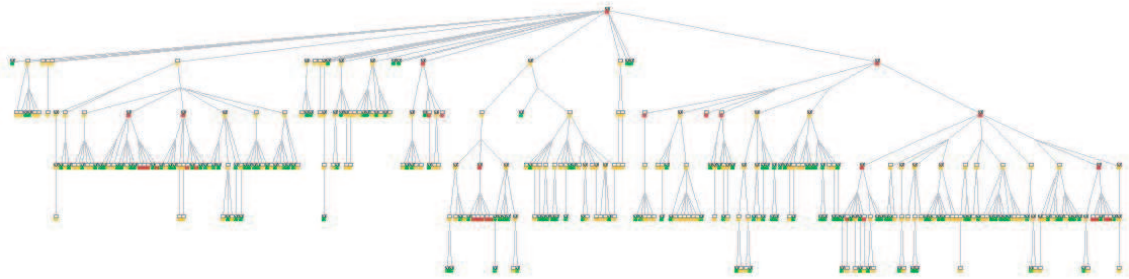

FIGURE 4.1: An example of ineffective visual technique for a high SPL variability

Our solution does not try to resolve the current complexity with a new visual technique or by applying an existing visual technique, e.g. fisheye view, in order to know the effectiveness of this kind of visual techniques on SPLs. However our solution tries to reduce the current complexity of SPL variability. If the complexity is reduced, we can use visual techniques in a more effective way. We can find such an approach in Pohl *et al.* [PBvdL05] with the *packaging system* and in the *Dopler* tool [Homa] with the decision model. The purpose of this chapter is to provide an abstract visual solution which gives a description of the new concepts introduced to reduce the complexity of SPL variability.

The remainder of this chapter is organised as follows: Section 4.1 explains our abstract visual solution explaining each of its concepts. Section 4.2 introduces a metamodel which includes Decisions, Features and Components. Section 4.3 shows the adapted product derivation framework of O'Leary *et al.* [OBTR07] presented in chapter 3 in order it fits better with our solution. Section 4.4 gives the working of our solution in the shape of use cases. Finally section 4.5 concludes this chapter criticising our solution.

4.1 Abstract Visual Solution

Our solution focuses on reducing the complexity of SPL variability. This variability can incorporate many variation points and so it is hard to display all of them in a simple and clear way to achieve a product derivation. To deal with this important visualisation issue, i.e. the complexity challenge, we introduce a concept which will allow reducing variability. Thanks to this reduced variability, we could achieve a product derivation with current visual techniques more easily.

Our solution focuses mainly on the *complexity challenge* but tries to respect the other challenges found in chapter 3, i.e. the *dependencies challenge* (see section 3.3). Furthermore it will try to contain the product derivation functions found in chapter 2 as well,

i.e. automatic management of constraints or intuitive mechanisms to see problems in the derivation process (see section 2.1.3).

4.1.1 Three abstraction levels

Our solution represents variability into three abstraction levels which are the **decisions** i.e. the higher level of variability, the **features** , i.e. the middle level, and the **components** , i.e. the lower level.

4.1.1.1 Decisions

The decision represents a choice in the higher level of variability. A decision gathers a set of features and could be described by the following logic formula:

$$Dec \Leftrightarrow f_1 \wedge f_2 \wedge ... \wedge f_n$$

This high level choice can be realised by customers or application engineers during the derivation process. They have to traduce their requirements into decision to take on the SPL. When they select a decision, a set of features is automatically selected as well. Thanks to decisions, customers or application engineers can easily configure a particular application of the SPL because a great part of the complexity in the variability is reduced.

The decisions visualisation could display the different possible decisions to take on the SPL (these decisions have been created beforehand by the domain engineering). It enables to select decisions to make a choice. It enables also to parameterise some decisions and to show some information about decisions, e.g. name, description, state, dependencies and constraints with others decisions, features and components. The display style can be in the shape of a structured and hierarchical data list in order to classify and gather decisions in theme, e.g. list of questions in a directory-style tree representation (see Figure 4.2), and also in the shape of query-reply sequence where each sequence represents a step of product configuration, e.g. you can find such sequences on the *Mini* car website where you can customise your future Mini car [cH].

4.1.1.2 Features

The feature represents a choice in the middle level of variability. A feature gathers a set of components and could be described by the following logic formula:

$$f \Leftrightarrow c_1 \wedge c_2 \wedge ... \wedge c_n$$

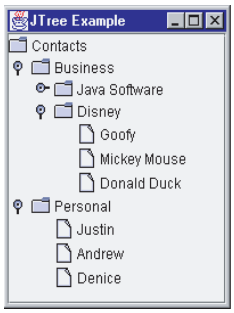

FIGURE 4.2: Directory-style tree example with Java

This middle level choice can be realised by application engineers during the derivation process. When they select a feature, a set of components is automatically selected as well. Thanks to the features, application engineers can improve easily the product configuration by resolving the last dependencies conflicts between features or by selecting some required features.

The features visualisation could display the different SPL features organised in hierarchy. It enables to select features, to show some information about features, e.g. name, description, state, dependencies and constraints with decisions, features and components. As seen in chapter 1, tree structures can be "*a useful visual metaphor for representing hierarchical structures*" [BNP+07].

4.1.1.3 Components

The component represents a choice in the lower level of variability. A set of components materialise a feature. This low level choice can be realised by application engineers during the derivation process. Thanks to the components, application engineers can improve easily the product configuration by resolving the last dependencies conflicts between components or by selecting some required components.

The components visualisation could display components which have to be used in order to build the configured product. It classifies the components by Reusable Artefacts (see section 1.1.3) types into a hierarchy. It enables to show some information about components, e.g. name, description, versions, parameters, dependencies and constraints with others decisions, features and components. The display style can be in the shape of structured and hierarchical list.

4.1.2 Dependencies management

Our solution needs a dependencies management in order to help the customers and the application engineers during the derivation process. This dependencies management allows avoiding incompatible choices during the derivation of one particular application

of the SPL. If this management is not achieve, the customers and the application engineers could make incompatible choices during product derivation which could lead to an invalid state.

Dependencies management includes the management of require and exclude dependencies between decisions, between features and between components. Furthermore we need a dependencies management between decisions - features and between features - components in order to choose appropriated features for the chosen decisions and in order to choose appropriated components for the chosen features. All of these dependencies allow an automatic management which will help the users during the product derivation process. This automatic management will be necessary in order to have beneficial user interactivity during the derivation process. Moreover it will be very interesting to have a mechanism to visualise each dependencies. The dependencies visualisation can be graphical as seen on Figure 4.3 (red line: exclude dependency, blue/green line: require dependencies) or textual with more information on the dependencies, e.g. name, dependencies actors.

4.1.3 User interactivity

During the product derivation process customers and application engineers have to choose among decisions, features and components in order to realise the product configuration. These choices are only possible if the tool has an interactive process to conduct the user. Such an "*interaction is important to get the most out of a visualisation*" [BNP+07]. Therefore interaction includes:

- **Actions** on visualisation, e.g. selection or elimination of features.
- **Intuitive mechanisms** to locate easily problems during the derivation process, e.g. features which are mandatory but not selected.
- An **automatic management** of constraints which clearly manages users actions consequences, e.g. verify require and exclude constraints.
- A complete and clear **information** appropriated to guide the user across the derivation process.

Thanks to the interactivity combined with the three abstraction levels, our solution can then find out if the desired user requirements can be satisfy by the SPL. Indeed our solution determines whether the desired requirements conform to the SPL. Moreover, our solution guides customers and application engineers during the search of a valid product configuration.

You can find an example of such an user interactivity (see Figure 4.3) where there are different kinds of information which allow a quick overview of the product configuration

state, e.g. colour information can be pre-attentively processed to have a faster interpretation [BNP+07]. We reuse this interactive tree structure in our implementation (see chapter 5).

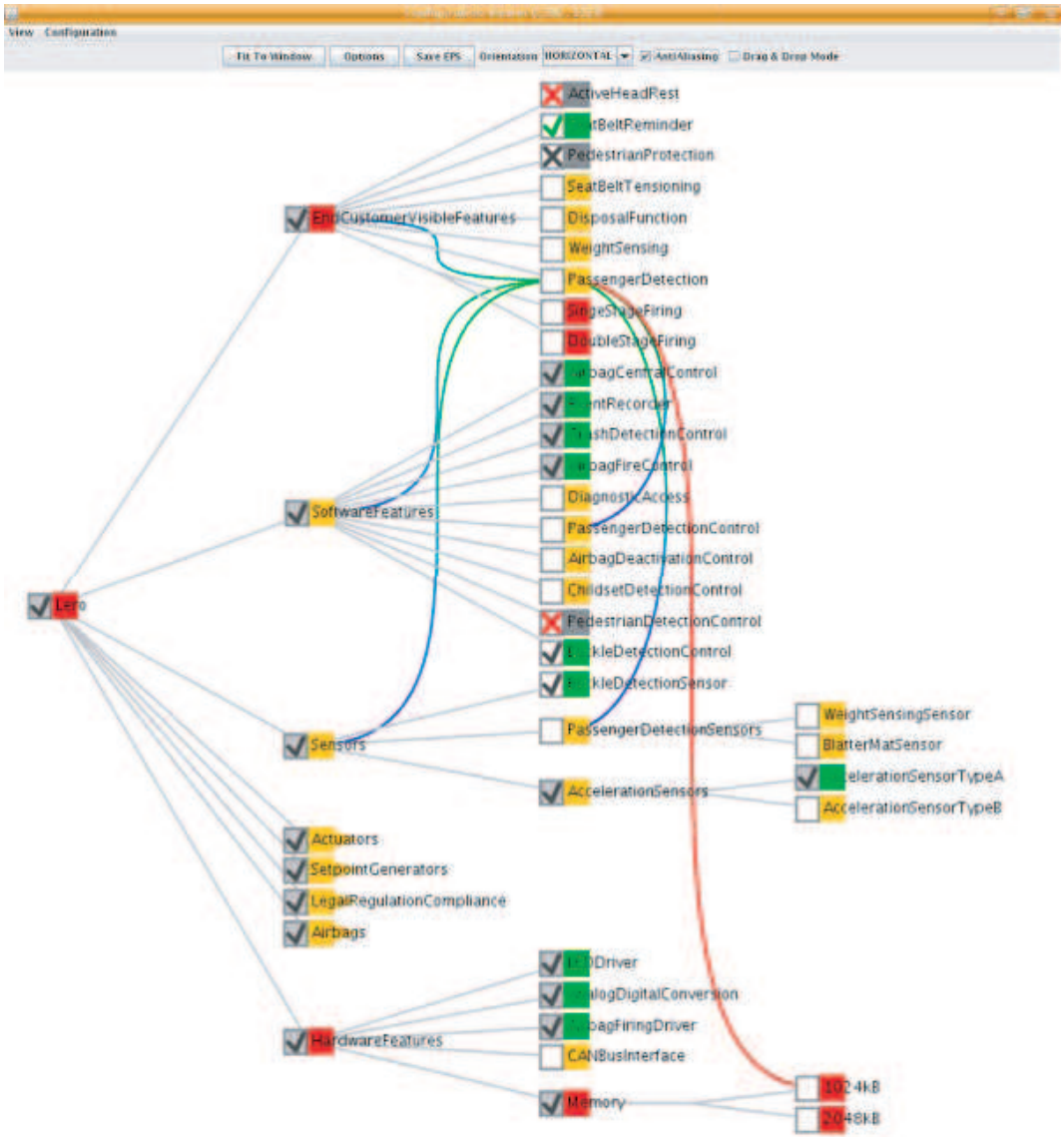

FIGURE 4.3: VISIT-FC Configuration Viewer Showing Features of the RESCU Product Line [BNP+07]

4.2 The DFC Metamodel

According to our abstract visual solution, we have designed a metamodel, called DFC Metamodel, including the Decisions, Features, and Components concepts. It proposes a tripled version of the Feature Metamodel presented by Cawley *et al.* (for more information about the Feature Metamodel see [NOH+07, CNP+08]) by adapting this one for the decisions and the components. Therefore, as we can see in Figure 4.4, the DFC

Metamodel contains an overall model which includes three sub-models, i.e. a Decision Model, a Feature Model, and a Component Model.

As we have seen in the previous section, our abstract visual solution manages three dynamical views where an action in one view has consequences on the two other views. Thereby, in order to make possible this management, dependencies between the three sub-models have been added (see Figure 4.4), i.e. ImplementedBy_DF between the Decision and Feature Model, and ImplementedBy_FC between the Feature and Component Model. The dependencies between the Decision and Component Model are implicit by following the ImplementedBy_DF dependencies and then the ImplementedBy_FC dependencies.

Concretely, these inter-model dependencies are one-to-one. The ImplementedBy_DF dependencies materialise a feature implementing a decision. Therefore, if a decision is implemented by many features, there will be as much ImplementedBy_DF dependencies as there are features implementing the decision. In the same way, the ImplementedBy_FC dependencies materialise a component implementing a feature. Thereby, if a feature is implemented by many components, there will be as much ImplementedBy_FC dependencies as there are components implementing the feature.

The cardinalities of these dependencies can be criticised in the sense that they are not optimal. Indeed, if a decision (respectively feature) is implemented by many features (respectively components), we have to create as much ImplementedBy_DF (respectively ImplementedBy_FC) dependencies as there are features (respectively components) implementing the decision (respectively feature). This redundancy could be avoided by using one-to-many cardinalities.

Furthermore, the metamodel does not contain any mechanisms that avoid inconsistencies in the SPL. For instance, as we can see in Figure 4.5, it is possible to create a feature B which implements a decision A and a feature D which implements a decision C while there is a Requires dependency between the features B and D, and an Excludes dependency between C and A. In this case, if we choose the decision A, the linked feature B will be selected as well. Then, thanks to the Requires dependency, the feature B selects the feature D and this last one selects the decision C which deselect the decision A. Therefore, the decision A is deselected while it should be selected. This situation is possible because there is a cycle in the dependencies. This should be avoided through other constructions in the metamodel.

Unfortunately, these problems do not relieve of our responsibility. This is a choice of the LERO (with whom we have collaborated to construct this metamodel) who did not take into account these problems in order to make the meta-model as simple as possible. The improvement of this metamodel could be the subject of a further work.

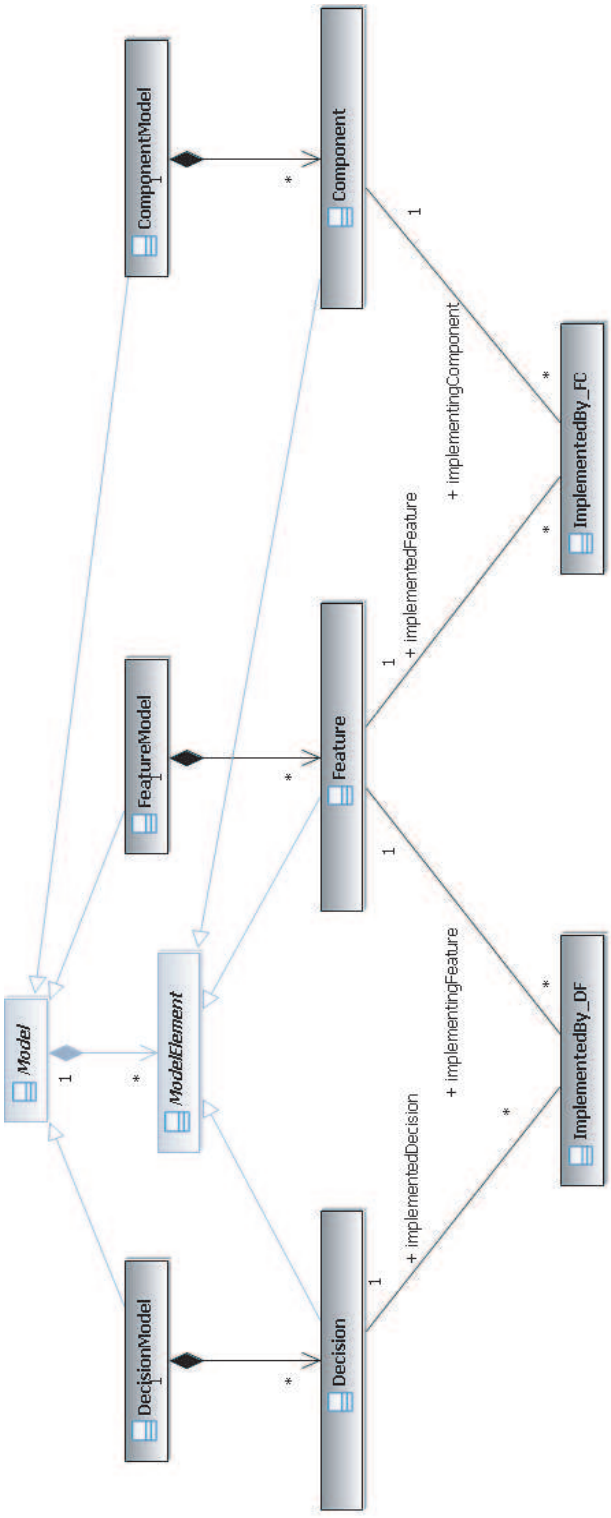

FIGURE 4.4: DFC Metamodel Overview

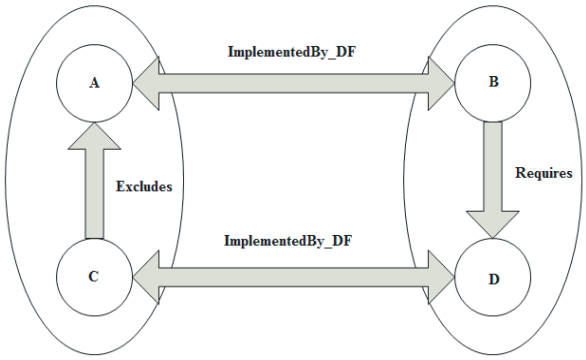

FIGURE 4.5: Example of inconsistency in the DFC Metamodel

4.3 Adapted Product Derivation Framework

In this part we present an adaptation of the product derivation framework presented in the chapter 3 in order to fit to our abstract visual solution needs. We describe each step of this adapted framework. For some steps, use cases also explain how it is possible to use our visual solution following this framework. We have to be aware that this use cases list is not exhaustive but it shows the most interesting use cases for the visualisation.

As we can see in Figure 4.6, the adapted framework has three main tasks:

- **Impact Analysis** where the product specific requirements are derived based on the customer requirements and negotiation with the platform team.
- **Reusability Using** where a validated (tested and accepted by the customer) partial or final product is created based on the platform artefacts.
- **Iterative Construction and Testing** where some components are adapted or created from scratch in order to fit at best to the customer needs.

4.3.1 Impact analysis

This task maps the customer requirements with the SPL in order to find a configuration that satisfies most of customer requirements. Either it already exists a configuration which is closest with the customer requirements or we derive a new configuration from the SPL. Then we analyse if all requirements are mapped with the previous configuration. Otherwise we start the customer negotiation in order to come to a compromise between the customer requirements and the needed changes on the SPL. Finally we form the product specific requirements which are prioritised for the development.

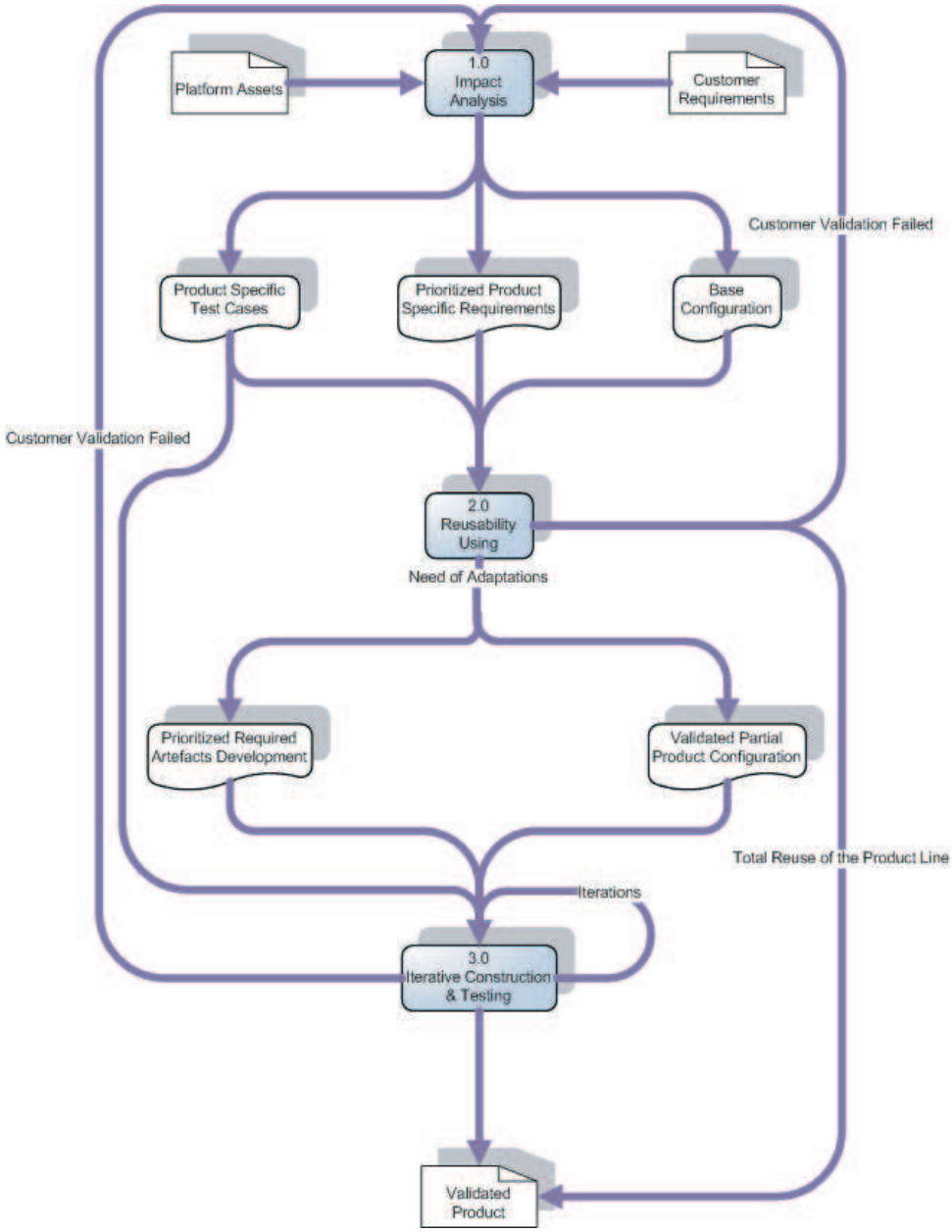

FIGURE 4.6: Adapted product derivation framework overview

4.3.1.1 Select closest matching configuration

Description: When the application engineer gets the customer requirements he realises that an existing product configuration respects all or most of customer requirements.

Pre-condition:

- The application engineer knows the customer requirements.
- There is at least one existing product configuration corresponding to the customer requirements.

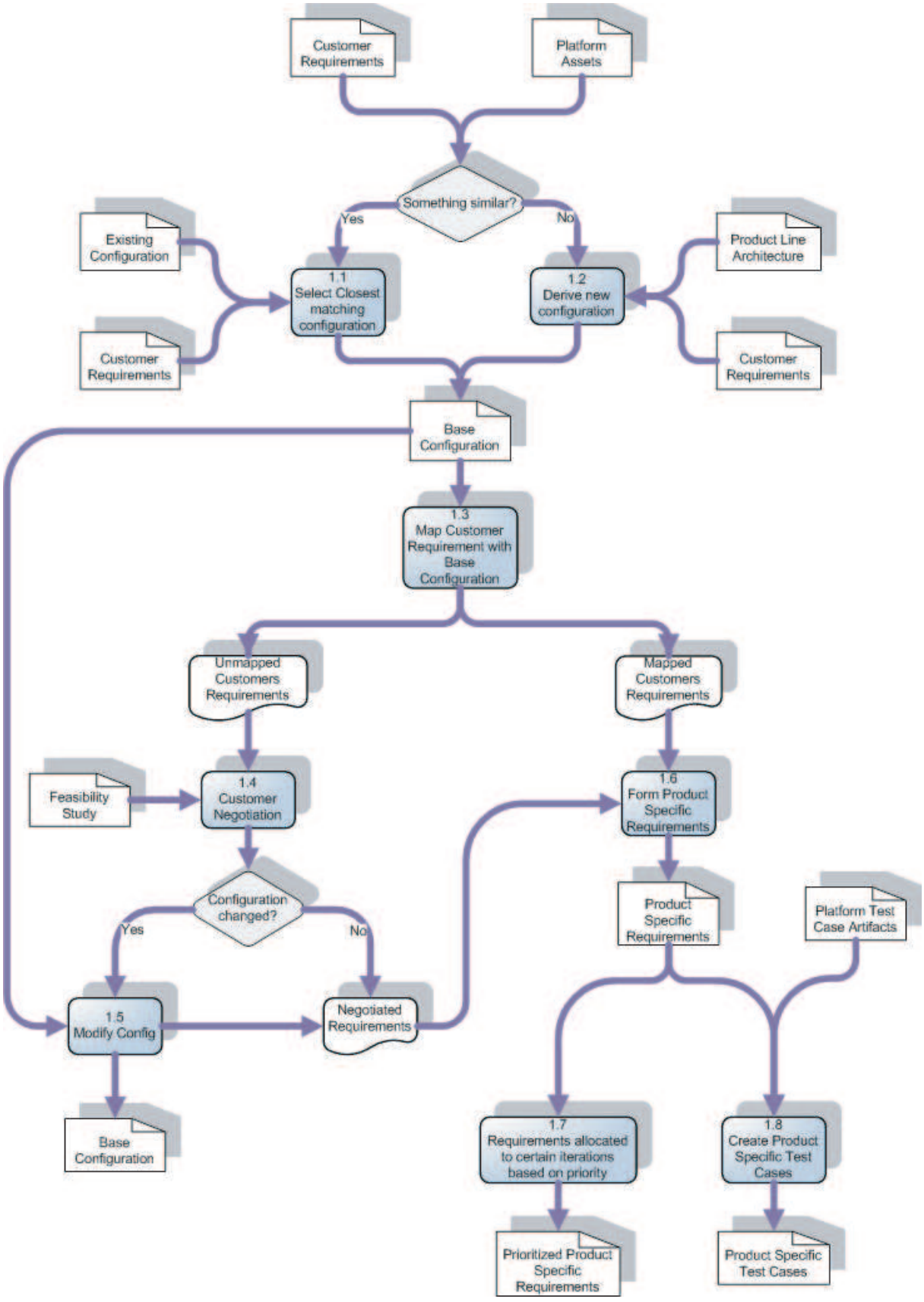

FIGURE 4.7: Impact analysis of the adapted product derivation framework

- The tool runs.

Post-condition:

- The application engineer knows the customer requirements.
- There is at least one existing configuration corresponding to the customer requirements.
- The tool runs again and an existing configuration corresponding to the customer requirements was opened.

Flow:

	Application Engineer	**SPLE-Tool**
1	Application engineer asks to reuse an existing product configuration	
2		Tool opens a list of all existing product configurations
3	Application engineer can see information about existing product configurations and differences between them	
4	Application engineer chooses an existing product configuration	
5		Tool opens the existing product configuration into 3 visualisations: decisions, features and Components

(End of UC)

TABLE 4.1: Use case - Select closest matching configuration

4.3.1.2 Derive new configuration

Description: When the application engineer gets the customer requirements he realises that none existing configuration respects all or most of customer requirements. So he decides to create a new product configuration.

Pre-condition:

- The application engineer knows the customer requirements.
- The tool runs.

Post-condition:

- The application engineer knows the customer requirements.

- The tool runs again and a new product configuration was started.

Flow:

	Application Engineer	**SPLE-Tool**
1	Application engineer asks to create a new product configuration	
2		Tool opens a new product configuration into 3 visualisations: decisions, features and Components

(End of UC)

TABLE 4.2: Use case - Derive new configuration

4.3.1.3 Derive new configuration

Description: When the application engineer gets a base configuration (an existing product configuration or a new product configuration[1]) he has to derive a product configuration from this base configuration. Either he has to derive from scratch or he has to continue the derivation of an existing product configuration. In the first case he has to find a product configuration that respects the most of customer requirements. In the second case he has to verify if the product configuration respects all of customer requirements and he has to continue the derivation if necessary in order to respect the most of customer requirements. After this step we will know mapped customers requirements and unmapped customers requirements.

Overall pre-condition:

- The application engineer knows the customer requirements.
- The tool runs and a base configuration is open.

Overall post-condition:

- The application engineer knows the customer requirements.
- The tool runs again and a base configuration is still open.

1 Normal case:
Name: Decision acceptance
Pre-condition: /
Post-condition:

[1]These two kinds of product configurations are the same except that an existing product configuration has already pre-selected decisions, features and components while it is not the case for a new configuration.

- A decision is accepted.

Flow:

	Application Engineer	**SPLE-Tool**
1	Application engineer accepte a decision	
2		Tool validates this decision
3		Tool manages automatically decisions, features and components dependencies

(End of UC)

TABLE 4.3: Use case - Decision acceptance during the mapping of customer requirements

2 Alternative case 1:
Name: Decision non-acceptance
Pre-condition: /
Post-condition:

- A decision is not accepted.

Flow:

	Application Engineer	**SPLE-Tool**
1	Application engineer does not accept a decision	
2		Tool validates this decision
3		Tool manages automatically decisions, features and components dependencies

(End of UC)

TABLE 4.4: Use case - Decision non-acceptance during the mapping of customer requirements

3 Alternative case 2:
Name: Map customer requirement using a query-reply sequence
Description: In this case the application engineer derives with the customer by using the query-reply sequence. He can switch or back to another step in order to satisfy the customer requirements. Each step contains some decisions.
Pre-condition: /
Post-condition:

- A derivation step is done

Flow:

	Application Engineer	**SPLE-Tool**
1	Application engineer chooses to do the derivation in a query-reply sequence mode	
2		Tool opens a new window and proposes several steps to do the derivation
3	Application engineer selects a step and accepts (does not accept) decisions in this step	
4		Tool manages automatically decisions, features and components dependencies
5	Application engineer go to the next step	

(End of UC)

TABLE 4.5: Use case - Map customer requirement using a query-reply sequence

4.3.1.4 Customer negotiation and modify configuration

In this step the application engineer tries to improve the product configuration with the customer based on a feasibility study (evaluate the time and costs for changes in the SPL in order to satisfy unmapped customer requirements). The customer has to make choices and the application engineer changes the product configuration according to customer choices and the feasibility study.

It is interesting to keep traces of such requirements because another customer can later ask the same requirements. In this case the application engineer could take back the feasibility studies to gain time in the derivation process. Moreover the domain engineer could implement an unmapped requirement because it was often requested but not accepted during the customer negotiation due to its impacts on the SPL (time consuming and costs for the customer).

4.3.1.5 Form product specific requirements

The application engineer will have to open the base configuration in the 3 initial views to see all taken decisions and all selected features in order to make, in parallel, his product specific requirements document in an editing tool.

4.3.1.6 Create product specific test cases

The tool will already have automatically selected the specific test cases linked to each validated decisions. For an unmapped requirement that was accepted during the customer negotiation, the specific test cases for this requirement will have to be designed

by the application engineer and will maybe be added later in the product line if the domain engineering decides to integrate this requirements in the SPL.

4.3.1.7 Requirements allocated to certain iterations based on priority

In this step the customer and the application engineer prioritise all validated decisions and the application engineer estimates the effort required to realise these ones. The end dates of iterations are specified and decisions are allocated to specific iterations (in the step 23 of the framework) based on their priority.

4.3.2 Reusability using

This task creates and tests the partial product with the platform artefacts. It wonders if the customer is satisfied with this partial product. Then we consider if we need to make adaptations of some platform artefacts in order to finalise the product.

4.3.2.1 Select subset of existing components

Description: In this step the application engineer chooses a subset of validated decisions according to the prioritised validated decisions list (step 1.8) in order to create the partial product.

Pre-condition:

- The tool runs and a base configuration is open.

Post-condition:

- The tool runs again and a base configuration is still open.
- A subset of existing components was chosen.

Flow:

4.3.2.2 Create partial product configuration

In this step the application engineer has to create the partial product that respects the subset of chosen components in step 2.1. In this case the visual assistance aims to help the application engineer to find easily the necessary components to realise the partial product in order to gain time.

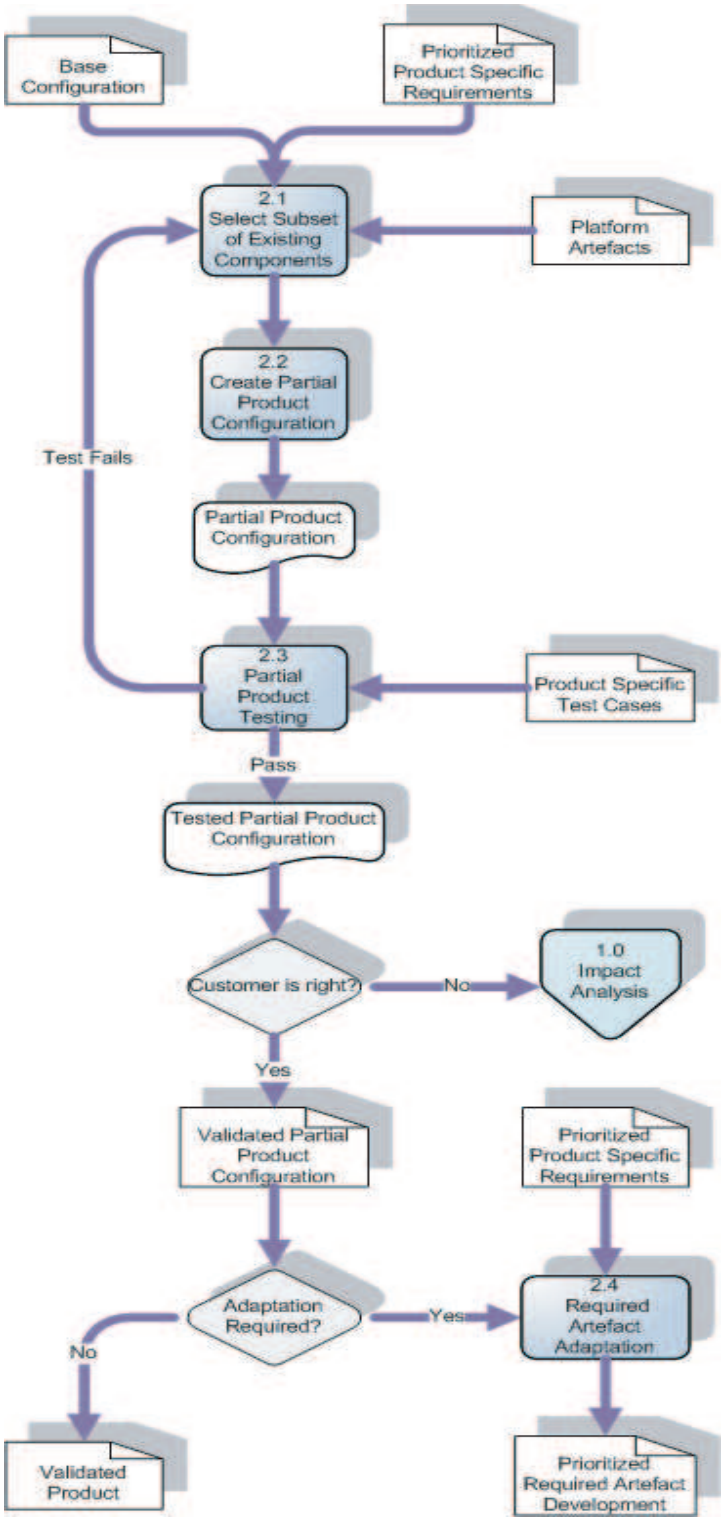

FIGURE 4.8: Reusability using of the adapted product derivation framework

	Application Engineer	**SPLE-Tool**
1	Application engineer selects all validated decisions to implement the partial product	
2		Tool selects automatically the components linked to these decisions except the ones which have to be adapted

(End of UC)

TABLE 4.6: Use case - Select subset of existing components

4.3.2.3 Partial product testing

In this step the application engineer has to test the partial product in order to verify if it works correctly and respects the subset of chosen validated decisions.

4.3.2.4 Identify required component adaptation

In this step the application engineer selects a validated decision for which components have to be adapted or developed from scratch according to the prioritised validated decisions list. Thereby we obtain new components adaptations to realise or new components to develop.

4.3.3 Iterative construction and testing

This task represents iterations for adaptations/constructions of artefacts. During these iterations we achieve and test the artefact adaptations/constructions. Then we integrate the artefacts adaptations/constructions in the partial product. When a test failure takes place, it is possible to come back in the reusability using task in order to analyse and resolve the problem. Finally when the customer requirements don't fit with the artefacts adaptations, we come back in the analysis impact in order to make a new elicitation of customer requirements.

4.3.3.1 Components development

In this step the application engineer has to develop (adaptation or from scratch) one or several component(s). This step requires special visualisation that is provided with some tools, i.e. tool for design or for implementation. So we think that a product derivation tool have not to support this step.

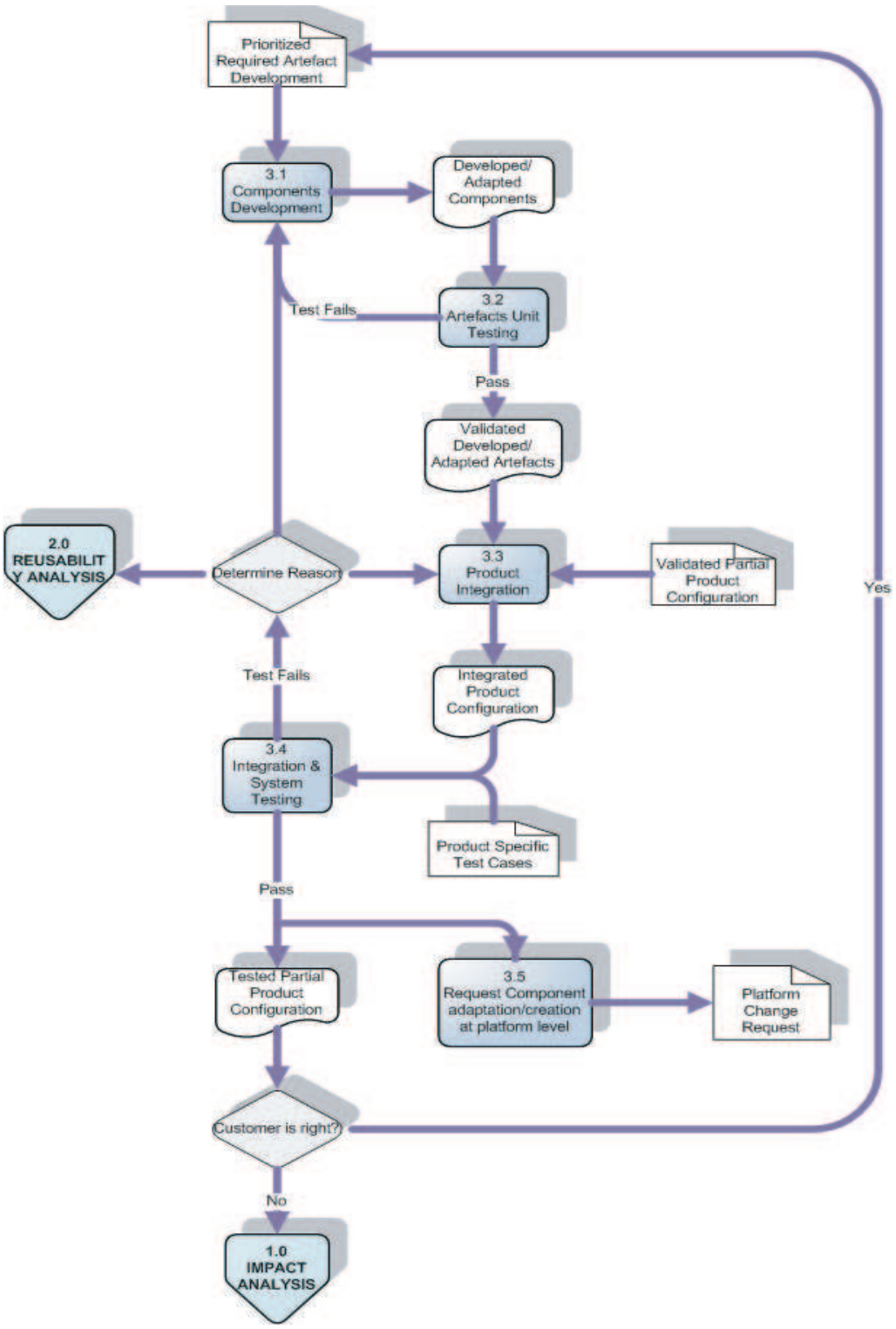

FIGURE 4.9: Iterative construction and testing of the adapted product derivation framework

4.3.3.2 Components unit testing

In this step the application engineer has to test the developed or adapted components in order to verify they work correctly. Thus this step does not require a special visualisation because it is present in others tools that support the editing of such components.

4.3.3.3 Product integration

In this step the application engineer has to integrate the validated developed /adapted components to the partial product. This step does not require a special visualisation because it is present in others tools that support such process.

4.3.3.4 Integration and system testing

In this step the application engineer has to test the integrated product configuration in order to verify if it works correctly and respects all implemented validated decisions. Therefore this step does not require a special visualisation because it is present in others tools that support such testing.

4.3.3.5 Request component adaptation/creation at platform level

It would be interesting that the application engineer saves the components adaptations/creations that he made. Therefore the components adaptations/creations will be integrated in the components visualisation for a future derivation. That means the new or adapted components have to appear in the decisions and features visualisation. This has to be achieved by the domain engineer. The application engineer job is limited to draw up a report on adapted/created components and to add components in the component view.

In a future derivation, if there are unmapped requirements, the application engineer will have to look for a new or adapted existing component that does not still appear in the decisions and features visualisation. This investigation will take place during the feasibility study realised before the customer negotiation. Later the domain engineering will make the choice to integrate or not the new added components based on the application engineer report.

4.4 Conclusion

This abstract visual solution intends to prove that a potential exists for the visual techniques by attempting to reduce what made them ineffective: the *complexity* presents

in SPL variability. To address this challenge we introduce three abstraction levels to represent the variability: *decisions*, *features* and *components*.

Decisions give a hand to application engineer to support an important step forward through the derivation process. Then application engineer can refine the derivation thanks to features and components. Even if these two last abstraction levels may be still complex we will not have to achieve all the derivation in these two levels but only some refinement choices. Furthermore if the features and components abstraction levels have a visualisation where we can see intuitively problems (see Figure 4.3), that facilitates the derivation tasks from a visual point of view.

Our solution is not proposed as the best to address all SPLE-tools challenges. However this solution intends to address some challenges and that is necessary for discussion about solutions which want to address SPLE-tools challenges. Of course this solution is an open question and a subject of future researches.

Chapter 5

Proof Of Concept

In order to illustrate our abstract visual solution, we have developed our own tool. This tool does not pretend to have a commercial or research value. It is just a prototype used as support for our thesis. As we are mainly focused on the realisation of a configuration during product derivation, we have decided to call this tool **VISIT-DFC** for **VIS**ual and **I**nteractive **T**ool for **D**ecisions, **F**eatures and **C**omponents configuration.

VISIT-DFC is based on the metamodel described in chapter 4 and attempts to respect our abstract visual solution. However, due to the lake of time for implementing the totality of this solution, we are just focusing on some parts on this one. Consequently, we describe first the scope of our implementation in relation to the abstract solution and the visualisation techniques used in our tool. Then we describe the design architecture of the tool in order to give a global picture of the implementation for further developments and we give the main VISIT-DFC functions. Finally we conclude by giving a critic of VISIT-DFC and further developments to realise.

The remainder of this chapter is organised as follows: Section 5.1 gives the scope of the implementation. Section 5.2 describes the tool architectural design. Section 5.3 presents the functions of VISIT-DFC. Finally section 5.4 concludes by giving a critic of the tool and the further developments to realise.

5.1 Implementation Scope

Our tool implements the three visualisations which were introduced in the chapter 4, i.e. decisions, features and components. Each visualisation has its own view in the tool. Based on the Lero's request, we reuse the visual techniques of the VISIT-FC tool to implement the features and components views. We understand the Lero's request because these visual techniques have friendly user interactivity (see Figure 4.3). For the decisions view, we implement it ourselves in the form of questions and possible answers

(see Figure 5.1). As our solution asks to display maximum information about decisions, features and components, we implement a fourth view, called information view, whose the goal is to display information about a pre-selected element.

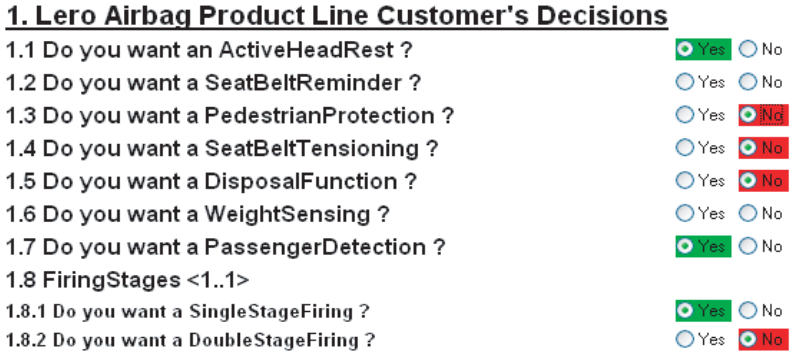

FIGURE 5.1: Decisions view of the VISIT-DFC tool

In order to respect the dependencies management, we implement an automatic management of all dependencies presented in the metamodel of chapter 4. Thanks to this automatic management, all actions performed on one of the three views, for instance the decisions view, generate other automatic actions on the two others views, features and components views. The display of these automatic actions will be implemented in a different colour to differentiate them from the user actions. Moreover we implement a dependencies display mechanism: firstly a textual visualisation which is displayed in the information view. Secondly a graphical visualisation which is already implemented in the VISIT-FC visualisation and which is displayed in the features and components views.

Regarding the user interactivity of our tool, it is already present with the VISIT-FC visualisation in the features and component views. Indeed this visualisation gathers all qualities for user interactivity as seen in chapter 4. The decisions view does not require such user interactivity because the user will only have to answer to some questions. Problems appear mainly in the features and components views after that the customer have given his own decisions. Indeed the application engineer needs an interactivity which allows easily locating problems.

Our tool mainly manages the first task of the adapted product derivation framework presented in the chapter 4. Indeed our tool with its mechanisms allows mainly identifying the right product configuration based on the customer requirements. The other tasks of the adapted framework are not implemented in our tool because of a lack of time.

5.2 Tool Architectural Design

As we said before, due to the lake of time for implementing the totality of our abstract visual solution, we are just focusing on some parts of this one. As a result, we have to choose an architectural design that allows a further completion of the tool user interface according to the non-implemented parts of our abstract visual solution or according to new visual needs of SPLE. The best architectural design addressing this requirement is the layered architecture (for more details on the different types of architectural design, see [LPR97]).

Indeed, in a layered architecture, a modification in a layer just affects the bottom layer and the top layer, not the entire architecture. This characteristic makes easier the maintainability of an implementation. In our case, as we can see in Figure 5.2, the visualisation part supposed to be extended in the future is situated at the top of the architecture. It is an advantage in the sense that the visualisation layer has not a top layer and thus, only the bottom layer, i.e. the visual mapping, will have to be adapted.

Furthermore, a layered architecture increases the readability and the elegance of an implementation. It allows incremental development and benefit of the object oriented paradigm mechanisms as well. In this optic, we make easier the work of people who will have to extend the tool by making the implementation understandable and practicable.

Finally, it is important to note that the architecture respects the reference model for variability visualisation which follows a series of data transformations from raw data to human perceiver (see Section 1.2.3).

In the next of this section we describe the different layers of the tool architecture as represented in Figure 5.2. There are five layers, i.e. the *Metamodel* layer, the *Parsing* layer, the *Datagraph* layer, the *Visual Mapping* layer, and finally the *Visualisation* layer.

5.2.1 Metamodel layer

This layer is our tool basis. It is based on the metamodel described in chapter 4 and is implemented using the *Eclipse Modeling Framework* (EMF) [BBM03]. Its main role is to provide the tool a data model representing a SPL and the operations required to manipulate this data model. In this intention, EMF provides an editor to define its own metamodel using the file extension *ecore*. This editor is not user friendly but a graphical editor, called *Topcased* [Homd], with an interactive process to construct the metamodel is available.

Once the metamodel is constructed, EMF generates the java classes representing the objects of this metamodel and two java classes allowing manipulating these object types,

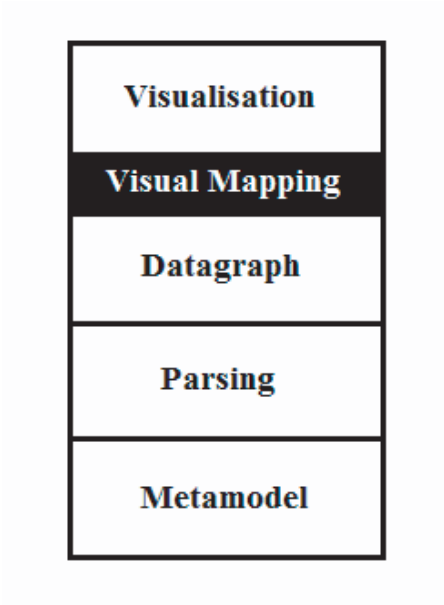

FIGURE 5.2: The layered tool architectural design

i.e. *EPackage* and *EFactory*. The manipulation on these object types consists mainly on methods for object types creation and access.

Finally EMF allows creating instances of the metamodel which represent models of particular SPLs. The objects of these instances have the types of the metamodel objects. Thereby the two classes *EPackage* and *EFactory* allow manipulating them. In the remainder we call the instances of a metamodel *SPL models*. These one are conserved under the XML format [Homg] and are used for the loading and saving of SPL data's for our tool.

As we can see in Figure 5.3, our implementation uses two SPL models, i.e. the *initial SPL* model and the *configuration SPL* model. The first one represents the base model of the product line which will be never changed. The second one represents a copy of the initial SPL model that will be changed through the configuration of the SPL and will conserve the modifications.

5.2.2 Parsing layer

The parsing layer is responsible of data transfer, i.e. loading and saving, between the configuration SPL model contained in the metamodel layer and the objects contained in the datagraph layer (see Figure 5.3). To realise this task, the parsing layer uses the methods offered by the two classes EPackage and EFactory. Furthermore, this layer is responsible to create the configuration SPL model by copying the initial SPL model.

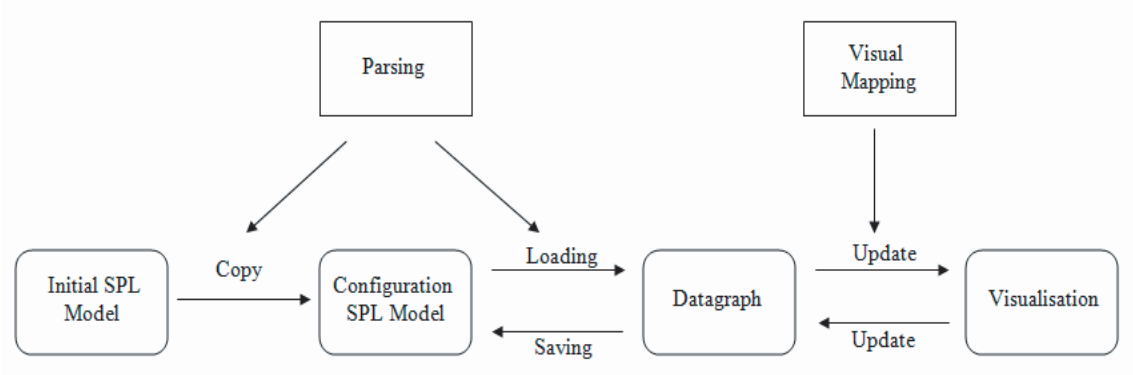

FIGURE 5.3: Global tool mechanism

5.2.3 Datagraph layer

The datagraph layer is another data representation of the configuration SPL model loaded in the memory cache by the parsing layer in the form of a graph. It is responsible to manage the consequences of the user interactivity transmitted by the visual mapping layer on the datagraph (see Figure 5.3). The datagraph layer has two main advantages.

First its data representation is loaded in the cache memory. This property allows being more high-performance in terms of tool speed execution. In opposition to permanent calls to the parsing layer which have to access to the configuration SPL model situated on the hard drive, this way of doing interacts just two times with the parsing layer, one time for the loading and one time for the saving, and so interacts just two times to the hard drive which is slower than the cache memory.

Then its data representation is totally independent of any visualisation techniques. That allows completing the maintainability objective of the tool by giving a basis that could be reused for further visual developments and, in our case, to develop the non-implemented parts of the abstract visual solution in an easy way.

5.2.4 Visual mapping layer

The visual mapping layer has two main roles. First the transformation of the datagraph objects into visual objects for the visualisation layer. Then ensuring the communication of the user interactions applied on the visualisation layer to the datagraph layer and to reflect the consequences of the datagraph management to the visualisation. This layer can be seen as a mapping layer between the datagraph layer and the visualisation layer (see Figure 5.3).

5.2.5 Visualisation layer

The visualisation layer is the tool user interface (see Section 5.1). It displays visually the datagraph objects and communicates the user interactions performed on the interface to the datagraph layer through the visual mapping. When the datagraph layer receives user interaction messages through the visual mapping, it manages the datagraph and reflects the consequences of the management to the visualisation through the visual mapping again (see Figure 5.3).

5.3 Tool functions

In this section we present most of the VISIT-DFC functions. We describe each view and all actions which can be made on these views. To describe the VISIT-DFC functions, we reuse the instantiation of the metamodel presented in [BNP+07]: Restraint System Control Unit (RESCU) product line. The RESCU model includes decisions, features and components of an automotive restraint system. The VISIT-DFC tool interface is represented in Figure 5.4. We can see the four views, i.e. the **decisions** view (5.3.1), the **features** and **components** views (5.3.2) and the **information** view (5.3.3). Of course this presentation focuses mainly on view functions and does not explain anymore what we find in these views. See the tool demonstration on the enclosed CD for more information about tool functions.

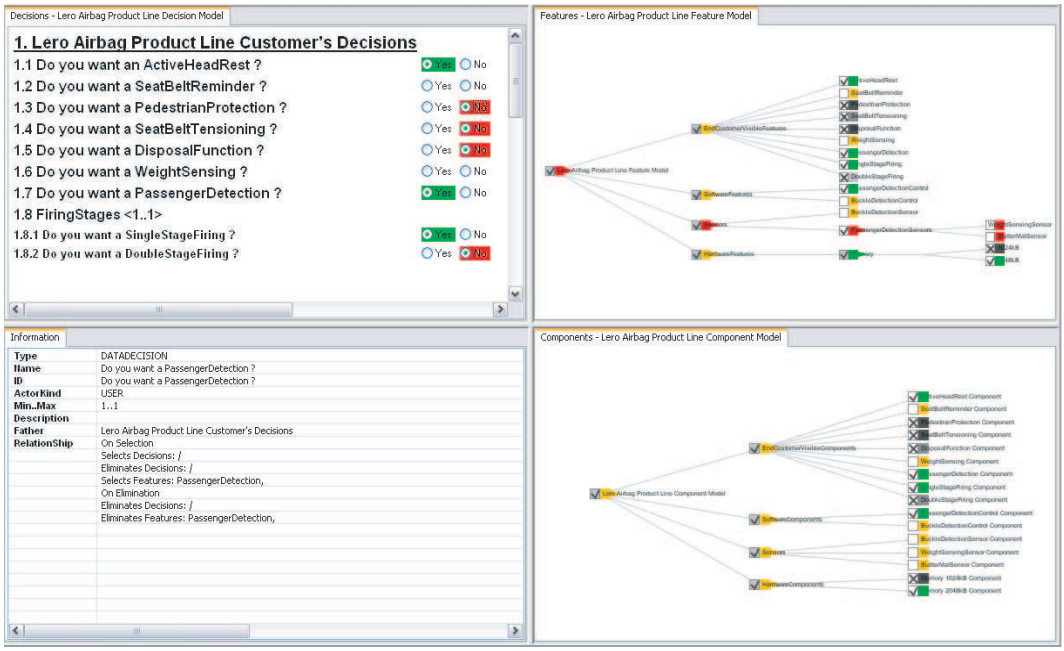

FIGURE 5.4: VISIT-DFC interface

5.3.1 Decisions view

In this view user has to answer to some questions. He has two possible responses: yes (green) or no (red). Each action on the decisions view has an impact, i.e. selection or elimination on the features and components views. This impact is well-visualised thanks to a highlighting mechanism implemented in the features and components views (see 5.3.2). Furthermore User can see information about a decision by a right double click on the desired decision. Decision information will be displayed in the information view. We can see an example of the decisions view in Figure 5.1.

5.3.2 Features and components views

In these views user has to select or eliminate features/components and there are many information styles as follows.

- **Colour coding** to indicate features and components state:
  - Green: selected
  - Grey: eliminated
  - Orange: optional
  - Red: mandatory but not selected
- Graphical symbols (cross or tick) with colours to indicate if features and components were selected or eliminated by the customer or automatically based on a dependency:
  - Green tick: selected by the customer
  - Black tick: selected automatically based on a dependency
  - Red cross: eliminated by the customer
  - Black cross: eliminated automatically based on a dependency.
- Shaded boxes to indicate that features and components have been pre-configured at an earlier stage of configuration and are no longer changeable.

In conclusion information can be pre-attentively processed and allow a faster interpretation. For instance, as seen in Figure 5.5, **node 1** is an automatically selected node (black tick) but this node is a variation point with a mandatory (1..1) choice (red colour).**Node 2** is a pre-configured node (shaded boxes) but this node is a variation point with an optional (1..2) choice (orange colour). **Node 3** is a pre-configured node (shaded boxes) but this node is a variation point with a mandatory (1..1) choice (red colour). These

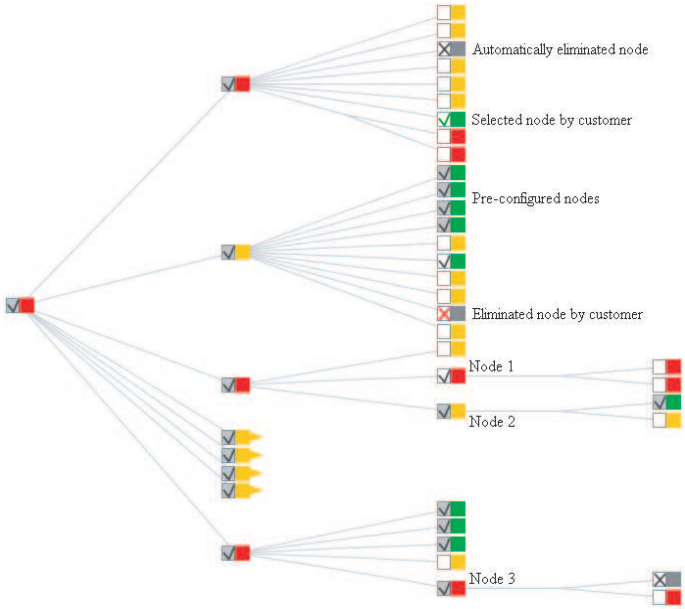

FIGURE 5.5: Faster interpretation example

colour codes allow a quick overview of the features/components views to show its current states, for instance to see where there are problems within the configuration (red colour).

Moreover these views include following mechanisms in order to improve the user interactivity:

- **Highlighting** allows visualising rapidly all user actions impacts in the features and components views (see Figure 5.6).

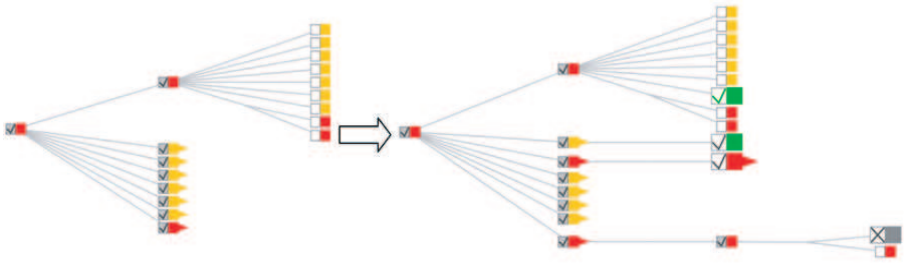

FIGURE 5.6: Highlighting example

- **Dependencies display** allows visualising rapidly dependencies of a feature or a component. *Mutual exclusion* is red and *requires* dependency is green/blue (the green side shows the feature which requires the other feature) (see Figure 5.7).
- **Group cardinalities display** allows visualising rapidly the group cardinality by pointing the mouse on the group (see Figure 5.8).

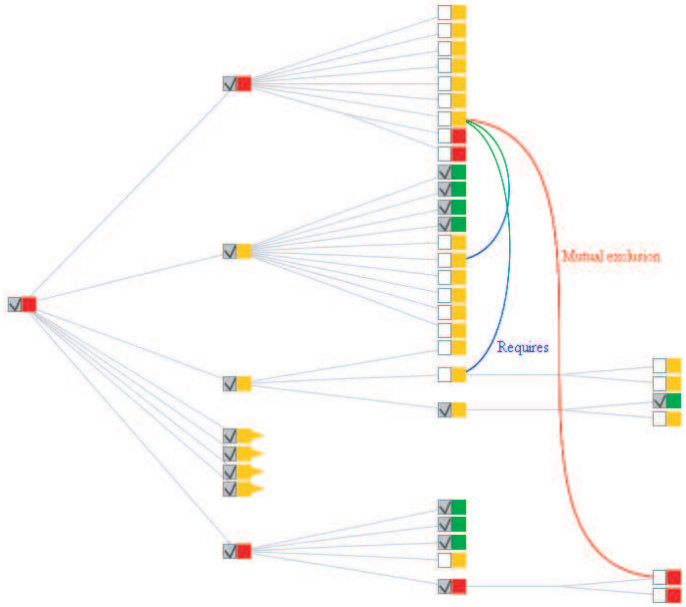

FIGURE 5.7: Dependencies display example

FIGURE 5.8: Group cardinalities display example

5.3.3 Information view

This view allows visualising all information about decisions, features and components (in our implementation, information about decisions are only implemented). Textual information like name, description, dependencies is displayed.

5.4 Conclusion

We have implemented a tool which is a proof of concept of our abstract visual solution. This tool is not yet complete and is not supposed to be used as basis for a commercial purpose. It just gives a good overview of *what can be done* to make easier the work of the product derivation engineers. We have seen that this fact passes through the using of friendly visualisation techniques and a good interactive process with the user.

However, VISIT-DFC covers only one task of the adapted product derivation framework outlined in chapter 4 and the automatic management of dependencies is yet partial. Therefore, further visual developments have to be done. This is the reason we have choose a layered architectural design in order to make easier these visual developments.

We keep in mind that the layered architectural design is not the best because a change in the metamodel could have reflects on the whole architecture but in the sense that

this metamodel was designed to be the largest as possible, we hope that such change will not happen in a near future. Thus, the chosen architectural design is a good deal for the moment.

Chapter 6

Conclusion and Discussion

This chapter aims to conclude this thesis. Its remainder is organised as follows: Section 6.1 reports our overall thesis approach and its linked pros and cons. Section 6.2 presents our personal contributions in the thesis. Finally, section 6.3 concludes the chapter and the thesis by giving the possible further works to realise.

6.1 Our Approach

It would be very interesting to report our feedback on the approach used along this thesis. We have adopted a dual approach which mixes a top-down approach and a bottom-up approach. For each of them we have registered the pros and the cons as follows.

6.1.1 Top-down approach

It requires a theoretical analysis of the Software Product Line Engineering domain in order to find main theories, properties, etc. for the SPLs management. Thereafter, these elements are changed into SPLE-tools functions which have to be implemented in order to manage SPLs in a relevant way. By giving significant functions, this approach aims to address most of developments which have to create effective SPLE-tools. Indeed this approach allows developers to find most of important and mandatory functions in order to manage SPLs in a relevant way. However these functions do not maybe address the user's needs because this approach does not elicit the requirements close to them. Moreover this approach does not take into account SPLs problems because it does not survey existing SPLs management to find their problems.

6.1.2 Bottom-up approach

It requires a survey on existing SPLs in order to find problems in their management. This approach allows investigators to discover more specific problems or goals during the SPLs management. These problems or goals are changed into requirements that SPLE-tools have to respect to satisfy user's needs. Thanks to this approach the developed functions in the tools are more relevant for user's needs. However the developed solution which addresses the found problems or goals during the survey could be too specific. Therefore, this solution would not be reusable for another SPL.

Nevertheless the combination between the top-down and the bottom-up approaches intends to find a collection of functions, problems and user's needs for SPLs management.

6.2 Contributions

Firstly we have built a framework for SPLE-tools evaluation. This framework has resulted in a non-exhaustive list of SPLE-tools functions via an investigation of the SPLE domain. By dint of this investigation we were aware of some current functions of SPLE-tools and of the lack of quality in some functions. Thanks to this top-down approach, we have made a step forward in the problem discovering which could be addressed using visualisation.

With the bottom-up approach we have studied some SPLs problems which complete the problems found in the top-down approach. Thanks to this case study we have been aware that most of problems take place during the product derivation process. As a result we are coming to find some SPLE-tools challenges for this process.

From these challenges we have introduced an abstract solution which addresses the SPLE-tools challenges. We have focused mainly on a challenge which was not yet addressed by the existing tool functions: the Complexity challenge. We have noticed that some visual techniques intended to address this complexity challenge but they were not really helpful for the user during the product derivation process on an industrial size SPL.

Our solution introduces three abstractions levels, i.e. decisions, features and components, which enable to reduce the cognitive complexity during a product derivation on a large SPL. Indeed, thanks to the decisions, we realise a forward step in the derivation process. Then we can refine the product configuration thanks to features and components. For this solution, we have updated the feature metamodel of the LERO in order to fit with these three abstraction levels.

Finally we have developed a prototype which implements the three abstraction levels and some SPLE-tools functions. This prototype does not claim to be usable but it

illustrates in concrete terms our solution in order to know if it addresses actually some challenges.

6.3 Future Work

This section presents the possible further works to achieve on our thesis.

6.3.1 Evaluation Framework

In order to find more SPLE-tools functions and improve the evaluation framework, we could perform more investigations in the SPLE domain. Furthermore, to realise a more relevant tool evaluation, we could use more concrete SPLE-tools. The classification matrix could be improved basing the ranking on the quality and price of the tools as well.

6.3.2 Challenges

In order to find more problems and user's needs, we could analyse more existing case studies. Of course we could realise our personal case study even if that would be a very heavy task. From these case studies and the SPLE domain theory, we could try to build a framework for product derivation process evaluation.

6.3.3 Solution using visualisation to address challenges

We could consider a different approach to address challenges. For instance, we could find a new visual technique or improve an existing visual technique to address some challenges. In the remainder of our solution, we could try to find how this solution would be usable in the domain engineering. For instance, how can we add decisions on an existing SPL?

Furthermore, the DFC Metamodel should be improved to avoid product derivation problems induced by inconsistencies created during the construction of the SPL. This metamodel could be improved to avoid redundancy as well. For instance, it should be necessary to avoid creating as much dependencies as there are features implementing a decision.

6.3.4 Prototype

We could complete the DFC-tool prototype based on the missing parts of the abstract visual solution that are not yet implemented. Furthermore, it would be necessary to improve the interface of this prototype because we are not java interface experts.

Appendix A

Detailled Tools Evaluation

A.1 Results Interpretation

Here is the strengths and weaknesses of each tool evaluated with the framework. Each tool score gets a graph where we can see the proportion between absent (purple side) and present functions (blue side).

A.1.1 VISIT-FC strengths and weaknesses

Strengths

- Derivation process:
  - A good interactive process to conduct the product derivation with colour coding and automatic management of constraints.
  - A very good intuitive mechanism to see problems in the derivation process. Indeed we can see:
    - Automatically selected features after to have to choose a feature.
    - Mandatory non-selected feature (red colour).
    - Hidden sub-feature problems of a feature (red colour).
- In general, Visit-FC have a clear and complete graphical notation during the derivation and there are some visualisation mechanisms to display a big quantity of information in a very small space (zooming, panning, drag drop, collapsing).
- Finally, the tool has a good autonomy (do not use Eclipse for instance) and a good portability (jar file can be use on many operating systems).

Weaknesses

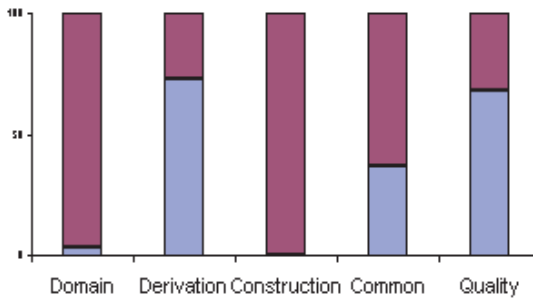

FIGURE A.1: VISIT-FC score

- Do not manage the domain engineering and the product construction.
- Do not manage information about features.
- View choice is poor: only tree matrix view (the last one does not bring much value).
- Effective search of a feature in the model is bad because it is based on the strict syntax of feature names (i.e. we must respect capital letter).
- There are no visualisation mechanisms that allow to not losing sight of the domain context. When zooming is switching on, the domain context is completely lost.

Conclusion

It is a good tool to make configurations based on a small model. For the rest it is very poor.

A.1.2 Gears strengths and weaknesses

Strengths

- An automatic building of the software products after the product derivation.
- A management of changes in software assets by applying them to every product of the product line.
- Perfect automatic management of constraints due to the effectiveness of the formal language of the tool.
- Possibility to create software product line hierarchy.
- The autonomy of the tool is good because it does not require specific software's to run.

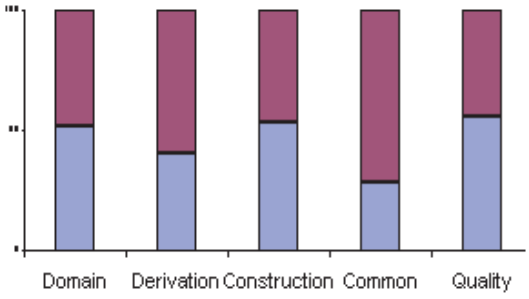

FIGURE A.2: Gears score

Weaknesses

- A very bad software product line construction process by using a formal language specific to the tool.
- A very bad product derivation by using variable instantiations into the formal language of the tool.
- Visual poverty of the product line management.
- Bad interoperability of the tool because it produces file formats specific to the tool and so, these can not be reused by other tools.

Conclusion

This tool has a good management of the product construction but is above all an expert tool and thus hasn't good visualisation mechanisms for domain engineering and product derivation.

A.1.3 Pure::variants strengths and weaknesses

Strengths

- Good management of feature information's by using tabs in the Eclipse environment.
- Good management of domain engineering and product derivation with a lot of views to facilitate these tasks.
- Special view to compare commonalities between configurations.
- Very good automatic management of constraints during the product derivation.
- Possibility to manage linkage between the feature model and the implementation.

- Explicit mechanisms to find problems in the product derivation very quickly.
- Possibility to do undo and redo actions.
- Possibility to call functionalities of other programs, i.e. code generation and compilation.
- Good portability because it runs with Eclipse.

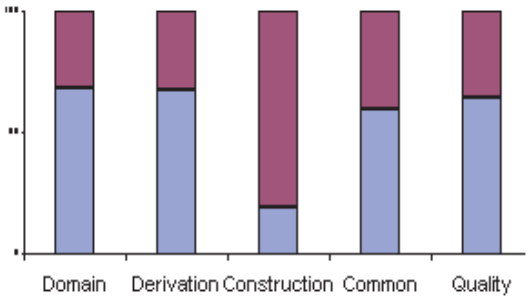

FIGURE A.3: Pure::variants score

Weaknesses

- No possibility to choose the metal-model which will be used by the tool.
- Only a linkage between the feature model and the implementation but not with other types of software artefacts (requirements, design, tests,).
- View mechanisms which become unsuitable in a software product line of industry size.

Conclusion

With its good management of domain engineering and product derivation, this tool seems to be very good. However there are still some improvements to do for software product line management in complex systems and for product construction.

A.1.4 Feature Modeling Plug-in (FMP) strengths and weaknesses

Strengths

- Good interactivity during the domain and application engineering. Especially with its wizard-based configuration that is a very good interactive process to conduct the product derivation.
- A complete automatic management during the derivation.

- A good management of the information about feature.
- A good graphical notation except for the constraints between features (see weaknesses).
- The choice of different feature model views is not bad because there some additional view of Eclipse (properties, console, Outline, Navigator,) but the main repository view of the feature model is very bad.
- There are some visualisation mechanisms to display a big quantity of information in a very small space (additional view of Eclipse).
- The portability of the tool is good because it runs with Eclipse and the interoperability of the tool is good thanks to the creation of XML file that can be reuse by another tool.

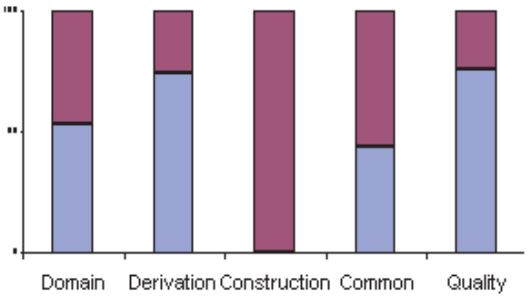

FIGURE A.4: Feature Modeling Plug-In score

Weaknesses

- Automatic management in domain engineering is absent (or unknown in the documentation).
- Poor intuitive mechanisms during the derivation process to see problems.
- Do not manage the product construction.
- The display of the constraints is very bad. It is not easy to see the constraints between features and it is very hard to see the different types of constraints.
- There are no visualisation mechanisms that allow to not losing sight of the domain context.

Conclusion

This tool has a good average in its functionality. It allows managing the domain engineering and the derivation process but with a poor visualisation.

A.1.5 XFeature strengths and weaknesses

Strengths

- Nearly perfect management of domain engineering.
- The choice of different feature model views is not bad because there are some additional views of Eclipse (properties, console, Outline, Navigator,) but there are no choices for the main feature model view.
- A good graphical notation except for the constraints between features (see weaknesses).
- There are some good visualisation mechanisms that allow to not losing sight of the domain context as graphical outline view.
- There are some visualisation mechanisms to display a big quantity of information in a very small space (additional view of Eclipse).
- The portability of the tool is good because it runs with Eclipse and the interoperability of the tool is good thanks to the creation of XML file that can be reuse by another tool.

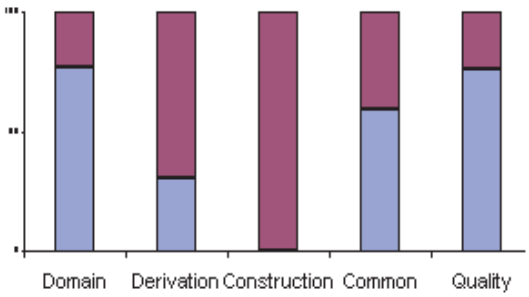

FIGURE A.5: XFeature score

Weaknesses

- Very poor derivation because we must create a new model (the application model) to do a derivation. Then we must validate it with the family model that we created before based on a meta-model. After the validation, you must correct the errors on the application model and then to validate it again. It's a too long process and moreover the visual facilities are totally absent during this process.
- Do not manage product construction.

- The display of the constraints is very bad. It is not easy to see the different types of constraints and it is very hard to see the constraints between features. Moreover, it is impossible to see constraints during the derivation process. So if there are constraints problems, we could see them after the application model validation (see to first weakness).

Conclusion

This tool is perfect to manage the domain engineering but when we want to make a derivation, it is very long in its process and is very poor in its interactivity and visualisation facilities.

A.2 Tools Classification Grid

As a result of this tools evaluation, we have created a grid for tools classification (Figure 6) that could be used by other evaluators to classify their evaluated tool in order to select the best tool for their company. This grid ranks the tools horizontally based on functions number that tools do not own and vertically based on the functions quality that the tools contain.

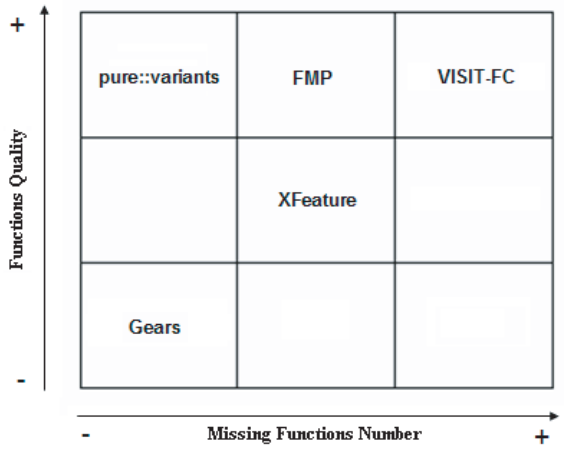

FIGURE A.6: Tools classification grid

A.3 Tools Quotation

Features	VISIT-FC	Gears	Pure:: variants	FMP	Xfeature
Domain Engineering					
Interactivity					
Feature structure					
Add feature	0	2	4	4	5
Delete feature	0	2	4	4	5
Feature parametrization	0	0	0	0	0
Add feature cardinalities to support cloning	0	2	4	5	5
Delete feature cardinalities to support cloning	0	2	4	5	5
Update feature cardinalities to support cloning	0	2	4	5	5
Add attribute	0	2	0	4	5
Delete attribute	0	2	0	4	5
Update attribute	0	2	0	4	5
Alternative choices					
Add alternative choice	0	2	4	4	5
Delete alternative choice	0	2	4	4	5
Add cardinality	0	2	4	5	5
Delete cardinality	0	2	4	5	5
Update cardinality	0	2	4	5	5
Constraints					
Add require constraint	0	2	4	4	5
Add exclude constraint	0	2	4	4	5
Add mandatory constraint	0	2	4	4	5
Add optional constraint	0	2	4	4	5
Delete require constraint	0	2	4	4	5
Delete exclude constraint	0	2	4	4	5
Delete mandatory constraint	0	2	4	4	5
Delete optional constraint	0	2	4	4	5
Multiple feature models	0	5	5	4	4
Product line hierarchy	0	5	0	0	0
Save/Load a feature model	1	5	5	5	5
Automatic Management					
Editor Generation	0	0	0	0	5
Model Validation	0	5	5	0	5
Constraints cancellation	0	0	0	0	0
Visual Facilities					
Commonalities	0	0	5	0	0
Application Engineering: Derivation					
Interactivity					
Actions on a feature					
Select feature	5	1	4	4	0
Deselect feature	5	1	4	4	0
Instantiate attributes of a feature	0	1	0	5	0
Update instantiated attributes	0	1	0	5	0

FIGURE A.7: Tools Quotation - Part 1

Load/save a configuration	5	5	5	5	5
Interactive process	4	2	3	5	1
specific adaptation	0	0	0	0	0
Chose a configuration	3	0	5	3	3
Automatic Management					
Constraints					
Verify require constraint	5	5	5	5	3
Verify exclude constraint	5	5	5	5	3
Verify mandatory constraint	5	5	5	5	3
Verify optional constraint	5	5	5	5	3
Verify alternative choice cardinality	4	5	5	5	3
Verify clone cardinality	0	5	5	5	3
Verify instantiated attribute type	0	5	5	5	0
Block selection	5	2	5	5	0
XML file	0	0	0	5	5
Visual Facilities					
Visual mechanisms					
See automatically selected features after to have to choose a feature	5	2	5	4	0
See mandatory non-selected feature	5	2	5	0	0
See hidden sub-feature problems of a feature	5	0	0	0	0
Configuration number	0	0	0	5	0
Auto-selected feature	5	0	5	4	0
Application Engineering: Product Construction					
Automatic Management					
Assembling	0	5	3	0	0
Software assets	0	5	0	0	0
Adaptation	0	0	0	0	0
Prevention	0	0	0	0	0
Common functions					
Interactivity					
Feature information					
Feature description	0	2	4	5	5
Feature priority	0	0	0	5	0
Occurrence number of a feature	0	0	0	5	0
Specific adaptation number	0	0	0	0	0
Project information	0	0	0	3	3
Dependencies					
Add artefact dependency	0	5	4	1	1
Delete artefact dependency	0	5	4	1	1
Update artefact dependency	0	5	4	1	1
feature model views	1	2	3	3	3
Search	2	2	4	0	0
Software assets change	0	5	0	0	0
Undo & redo actions	0	0	5	0	5
Visual Facilities					
Graphical Notation					

FIGURE A.8: Tools Quotation - Part 2

Distinct a feature from a sub-feature	5	2	5	5	5
Distinct a feature group	4	2	5	5	5
See constraints between features	4	2	3	3	1
Distinct constraint types between features	4	2	3	1	1
See name of a feature	5	2	5	5	5
Context	1	0	3	1	3
Big information quantity					
Drag & drop feature	5	0	0	0	5
Collapse feature	5	3	3	5	5
Panning	5	0	0	0	0
Zooming	5	0	0	0	5
Animation	2	0	0	1	1
Degree of user interests	0	1	5	5	5
3D view mechanism	0	0	0	0	0
Feature information	0	3	5	4	4
Tool quality requirements					
Interoperability	1	0	4	5	5
Portability	4	3	4	5	5
Co-existence	0	0	0	0	0

FIGURE A.9: Tools Quotation - Part 3

Bibliography

[ABDM01] Alain Abran, Pierre Bourque, Robert Dupuis, and James W. Moore, editors. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, Piscataway, NJ, USA, 2001.

[AC04] Michal Antkiewicz and Krzysztof Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM.

[Ban02] Russ Banham. *The Ford Century: Ford Motor Company and the Innovations that shaped the World*. Artisan, 2002.

[BBM03] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric paradigm. 2:528–532, 1994.

[Beu03] Danilo Beuche. Variant management with pure::variants. Technical report, 2003.

[BFG+02] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink, and Klaus Pohl. Variability issues in software product lines. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 13–21, London, UK, 2002. Springer-Verlag.

[BI90] D. V. Beard and J. Q. Walker II. Navigational techniques to improve the display of large two-dimensional spaces. 9(6), 1990.

[BNP+07] Goetz Botterweck, Daren Nestor, André Preußner, Ciarán Cawley, and Steffen Thiel. Towards supporting feature configuration by interactive visualisation. In *VISPLE '07: 1st International Workshop on Visualisation in Software Product Line Engineering*, 2007.

[Bri03] Douglas G. Brinkley. *Wheels for the World: Henry Ford, His Company, and a Century of Progress, 1903-2003*. Viking, New York, NY, USA, 2003.

[cH] Mini car Homepage. `http://www.mini.com/`.

[CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Softw.*, 15(6):37–45, 1998.

[CK98] Michael A. Cusumano and Nobeoka Kentaro. *Thinking Beyond Lean: How Multi-Project Management is Transforming Product Development at Toyota and Other Companies*. Free Press, New York, NY, USA, 1998.

[CMS99] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. *Readings in information visualization: using vision to think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[CN01] Paul Clements and Linda M. Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[CNP+08] Ciarán Cawley, Daren Nestor, André Preußner, Goetz Botterweck, and Steffen Thiel. Interactive visualisation to support product configuration in software product lines. In *VaMoS '08: the First International Workshop on Variability Modelling of Software-intensive Systems*, 2008.

[Coo00] Alan Cooper. *Itinerary to Mass Customisation*. Pool Spring edition, 2000.

[Dav96] Stan Davis. *Future Perfect, 10th anniversary edition*. Addison-Wesley Pub Co, Harlow, England, 1996.

[DSB04] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Experiences in software product families: Problems and issues during product derivation. In *Software Product Lines, Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September, 2004, Proceedings*, volume 3154 of *Lecture Notes in Computer Science*, pages 165–182. Springer, 2004.

[DSB05] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194, 2005.

[FPB95] Jr. Frederick P. Brooks. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Fur86] G. W. Furnas. Generalized fisheye views. *SIGCHI Bull.*, 17(4):16–23, 1986.

[GHM08] Keith Gallagher, Andrew Hatch, and Malcolm Munro. Software architecture visualization: An evaluation framework and its application. *Software Engineering, IEEE Transactions on*, 34(2):260–270, 2008.

[Gla97] Robert L. Glass. *In the Beginning: Recollections of Software Pioneers*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1997.

[HD97] Elizabeth Haywood and Philip Dart. Towards requirements for requirements modelling tools. In *Proceedings of the 2nd Australian Workshop on Requirements Engineering*, pages 61–69, 1997.

[HKWB04] Matthias Hoffmann, Nikolaus Kuhn, Matthias Weber, and Margot Bittner. Requirements for requirements management tools. In *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International*, pages 301–308, Washington, DC, USA, 2004. IEEE Computer Society.

[HMM00] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[Homa] Dopler Homepage. `http://ase.jku.at/dopler`.

[Homb] Ford Fox Homepage. `http://www.ford-fox.org`.

[Homc] Gears (BigLever) Homepage. `http://www.biglever.com/`.

[Homd] Topcased Homepage. `http://www.topcased.org/`.

[Home] VISPLE Homepage. `http://www.lero.ie/visple2007/`.

[Homf] XFeature Homepage. `http://www.pnp-software.com/XFeature/`.

[Homg] XML Homepage. `http://www.xml.com/`.

[IBM72] IBM. Os/360 introduction. Technical report, 1972.

[iH] Feature Modeling Plug in Homepage. `http://gp.uwaterloo.ca/fmp/`.

[LB07] Reed Little and Randy Blohm. A plea for help with variability, in two acts. In Klaus Pohl, Patrick Heymans, Kyo-Chul Kang, and Andreas Metzger, editors, *VaMoS '07: the First International Workshop on Variability Modelling of Software-intensive Systems*, Limerick, Ireland, 2007.

[Lev95] Nancy G. Leveson. *Safeware: system safety and computers.* ACM, New York, NY, USA, 1995.

[LPR97] Bass Len, Clements Paul, and Kazman Rick. *Software Architecture in Practice.* Addison-Wesley Professional, December 1997.

[Mac86] Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, 1986.

[Mae05] Libero Maesano. La crise du logiciel et le développement de la production communautaire de logiciel ouvert. 2005.

[Mat05] Raimundas Matulevicius. *Process Support for Requirements Engineering: A Requirements Engineering Tool Evaluation Approach*. PhD thesis, NTNU, Faculty of Information Technology, Mathematics and Electrical Engineering, Department of Computer and Information Science, June 2005.

[MDB87] B. McCormick, T. DeFanti, and M. Brown. Visualization in scientific computing-a synopsis. *IEEE Comput. Graph. Appl.*, 7(7):61–70, 1987.

[Moo06a] Daniel Moody. Dealing with "map shock": A systematic approach for managing complexity in requirements modelling. In *REFSQ '06: in Proceedings of the 12th Working Conference on Requirements Engineering: Foundation for Software Quality*, 2006.

[Moo06b] Daniel L. Moody. What makes a good diagram? improving the cognitive effectiveness of diagrams in is development. In *ISD '06: the 15th International Conference in Information Systems Development*, pages 481–492. Springer US, 2006.

[MW03] Merriam-Webster. *Merriam-Webster's Collegiate Dictionary, 11th Edition thumb-notched with Win/Mac CD-ROM and Online Subscription*. Merriam-Webster, July 2003.

[Neu95] Peter G. Neumann. *Computer related risks*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[NOH+07] Daren Nestor, Luke O'Malley, Patrick Healy, Aaron Quigley, and Steffen Thiel. Visualisation techniques to support derivation tasks in software product line development. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 315–325, New York, NY, USA, 2007. ACM.

[NR69] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

[NS02] Oksana Nikiforova and Uldis Sukovskis. Framework for comparison of system modeling tools. In *Proceedings of the Baltic Conference, BalticDB&IS 2002*, pages 63–70. Institute of Cybernetics at Tallin Technical University, 2002.

[NT95] Ikujiro Nonaka and Hirotaka Takeuchi, editors. *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press, New York, NY, USA, 1995.

[OBTR07] Padraig O'Leary, Muhammad Ali Babar, Steffen Thiel, and Ita Richardson. Product derivation process and agile approaches: Exploring the integration potential. In *Central and East European Conference*, Poznan, Poland, 2007.

[oD96] United States Department of Defense. Information security: Computer attacks at department of defense pose increasing risks. Technical report, 1996.

[PBvdL05] Klaus Pohl, Gunter Bockle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer, New York, NY, USA, 2005.

[Pet95] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, 1995.

[PK00] Gerald Post and Albert Kagan. Oo-case tools: an evaluation of rose. *Information & Software Technology*, 42(6):383–388, 2000.

[Pre07] André Preußner. Visualization-supported product derivation. Master's thesis, Brandenburgische Technische Universitat, Cottbus, Germany, October 2007.

[psGH] pure-systems GmbH Homepage. `http://www.pure-systems.com/`.

[RH00] Raúl Rojas and Ulf Hashagen, editors. *The first computers: history and architectures.* MIT Press, Cambridge, MA, USA, 2000.

[SR96] Mike Scaife and Yvonne Rogers. External cognition: how do graphical representations work? *Int. J. Hum.-Comput. Stud.*, 45(2):185–213, 1996.

[Str96] Alfred Strohmeier. Cycle de vie du logiciel. In Alfred Strohmeier and Didier Buchs, editors, *Genie logiciel: principes, methodes et techniques*, pages 1–28. Presses Polytechniques et Universitaires Romandes, 1996.

[SZG+96] Doug Schaffer, Zhengping Zuo, Saul Greenberg, Lyn Bartram, John Dill, Shelli Dubs, and Mark Roseman. Navigating hierarchically clustered networks through fisheye and full-zoom methods. *ACM Trans. Comput.-Hum. Interact.*, 3(2):162–188, 1996.

[TM04] Melanie Tory and Torsten Möller. Human factors in visualization research. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):72–84, 2004.

[vN45] John von Neumann. First draft of a report on the edvac. Technical report, 1945.

[War00] Colin Ware. *Information visualization: perception for design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software product-line engineering: a family-based software development process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.