

Human–Computer Interaction Series

Jean-Sébastien Sottet  
Alfonso García Frey  
Jean Vanderdonckt *Editors*

# Human Centered Software Product Lines

 Springer

# Human–Computer Interaction Series

## **Editors in chief**

Desney Tan

Microsoft Research, Redmond, Washington, USA

Jean Vanderdonckt

Université catholique de Louvain, Belgium

More information about this series at <http://www.springer.com/series/6033>

Jean-Sébastien Sottet • Alfonso García Frey  
Jean Vanderdonckt  
Editors

# Human Centered Software Product Lines

 Springer

*Editors*

Jean-Sébastien Sottet  
Luxembourg Institute for Science  
& Technology  
ITIS  
Esch/Alzette, Luxembourg

Alfonso García Frey  
Luxembourg Institute of Science  
and Technology (LIST)  
ITIS  
Esch/Alzette, Luxembourg

Jean Vanderdonckt  
Université catholique de Louvain  
Louvain School of Management  
Louvain-La-Neuve, Belgium

ISSN 1571-5035

Human–Computer Interaction Series

ISBN 978-3-319-60945-4

ISBN 978-3-319-60947-8 (eBook)

DOI 10.1007/978-3-319-60947-8

Library of Congress Control Number: 2017951024

© Springer International Publishing AG 2017

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature

The registered company is Springer International Publishing AG

The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

# Contents

<b>1 Engineering Configuration Graphical User Interfaces from Variability Models</b> .....	1
Quentin Boucher, Gilles Perrouin, Jean-Marc Davril, and Patrick Heymans	
<b>2 User Interfaces and Dynamic Software Product Lines</b> .....	47
Dean Kramer and Samia Oussena	
<b>3 Variability Management and Assessment for User Interface Design...</b>	81
Jabier Martinez, Jean-Sébastien Sottet, Alfonso García Frey, Tewfik Ziadi, Tegawendé Bissyandé, Jean Vanderdonckt, Jacques Klein, and Yves Le Traon	
<b>4 Feature-Based Elicitation of Cognitively Efficient Visualizations for SPL Configurations</b> .....	107
Céline Sauvage-Thomase, Nicolas Biri, Gilles Perrouin, Nicolas Genon, and Patrick Heymans	
<b>5 Addressing Context-Awareness in User Interface Software Product Lines (UI-SPL) Approaches</b> .....	131
Thouraya Sboui, Mounir Ben Ayed, and Adel M. Alimi	

# Contributors

**Adel M. Alimi** REGIM lab., University of Sfax, Sfax, Tunisia

**Mounir Ben Ayed** REGIM lab., University of Sfax, Sfax, Tunisia

**Nicolas Biri** Luxembourg Institute of Technology, Esch-sur-Alzette, Luxembourg

**Tegawendé Bissyandé** Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg City, Luxembourg

**Quentin Boucher** CETIC, Charleroi, Belgium

**Jean-Marc Davril** PReCISE Research Centre, University of Namur, Namur, Belgium

**Alfonso García Frey** Yotako S.A., Esch/Alzette, Luxembourg

**Nicolas Genon** PReCISE, Faculty of Computer Science, University of Namur, Namur, Belgium

**Patrick Heymans** PReCISE Research Centre, Faculty of Computer Science, University of Namur, Namur, Belgium

**Jacques Klein** Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg City, Luxembourg

**Dean Kramer** School of Computing and Technology, University of West London, London, UK

**Yves Le Traon** Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg City, Luxembourg

**Jabier Martinez** Sorbonne University, Paris, France

**Samia Oussena** School of Computing and Technology, University of West London, London, UK

**Gilles Perrouin** PReCISE Research Centre, Faculty of Computer Science, University of Namur, Namur, Belgium

**Céline Sauvage-Thomase** Luxembourg Institute of Technology, Esch-sur-Alzette, Luxembourg

**Thouraya Sboui** REGIM lab., University of Sfax, Sfax, Tunisia

**Jean-Sébastien Sottet** Luxembourg Institute of Science and Technology (LIST), Esch/Alzette, Luxembourg

**Jean Vanderdonckt** Louvain School of Management, Université catholique de Louvain, Louvain-la-Neuve, Belgique

**Tewfik Ziadi** Sorbonne University, Paris, France



# Chapter 1

## Engineering Configuration Graphical User Interfaces from Variability Models

Quentin Boucher, Gilles Perrouin, Jean-Marc Davril, and Patrick Heymans

**Abstract** In the past, companies produced large amounts of products through mass production lines. Advantages of such an approach are reduced production costs and time-to-market. While it is (still) appropriate for some goods like food or household items, customer preferences evolve to customised products. In a more and more competitive environment, product customisation is taken to the extreme by companies in order to gain market share. Companies provide customisation tools, more commonly called product configurators, to assist their staff and customers in deciding upon the characteristics of the product to be delivered.

Our experience reveals that some existing configurators are implemented in an ad-hoc fashion. This is especially cumbersome when numerous and non-trivial constraints have to be dealt with. For instance, we have observed in two industrial cases that relationships between configuration options are hard-coded and mixed with GUI code. As constraints are scattered in the source code, severe maintenance issues occur.

In this chapter, we present a pragmatic and model-driven way to generate configuration GUIs. We rely on feature models to represent and reason about the configuration options and their complex relationships. Once feature models have been elaborated, there is still a need to produce a GUI, including the integration with underlying reasoning mechanisms to control and update the GUI elements. We present a model-view-presenter architecture to design configurators, which separates concerns between a feature model (configuration option modelling), its associated solver (automated reasoning support) and the presentation of the GUI. To fill the gap between feature models and configuration GUIs, the various constructs of the feature model formalism are rendered as GUI elements through model transformations. Those transformations can be parametrised through beautification and view languages to derive specific configuration GUIs. We illustrate our approach on an IPv6 addressing plan configurator.

---

Q. Boucher (✉)  
CETIC, Avenue Jean Mermoz, 28, 6041, Charleroi, Belgium  
e-mail: [quentin.boucher@cetic.be](mailto:quentin.boucher@cetic.be)

G. Perrouin • J.-M. Davril • P. Heymans  
PRECISe Research Centre, University of Namur, Rue Grandgagnage 21, 5000, Namur, Belgium  
e-mail: [gilles.perrouin@unamur.be](mailto:gilles.perrouin@unamur.be); [jean-marc.davril@unamur.be](mailto:jean-marc.davril@unamur.be); [patrick.heyman@unamur.be](mailto:patrick.heyman@unamur.be)

## 1.1 Introduction

In the past, companies produced large amounts of products through mass production lines. Advantages of such an approach are reduced production costs and time-to-market. While it is (still) appropriate for some goods like food or household items, customer preferences evolve to customised products. Even car production which was a major example of mass production has moved to the customisation category. Henry Ford played a pioneering role in the mass production of cars. *Fordism* aimed to achieve higher productivity by standardizing the output, breaking the work into small well specified tasks, and using conveyor assembly lines. However, Ford's quote "Any customer can have a car painted any colour that he wants so long as it is black" already illustrates the limitations of mass production, back in 1923.

In a more and more competitive environment, product customisation is taken to the extreme by companies in order to gain market share. Companies provide customisation tools, more commonly called *product configurators*, to assist their staff and customers in deciding upon the characteristics of the product to be delivered. This trend is further strengthened by the ever-growing presence of such configurators on the Internet.

The key idea behind configurators is to provide end-users with an easy-to-use Graphical User Interface (GUI) where they can select the desired options and customise their product. The result of the configuration is then used by the manufacturer in order to produce the final product with the required options. Generally, the user is guided by the GUI in her process. That guidance manifests itself in different ways. First, configuration can be broken down into steps. Typically, a step represents a set of logically linked configuration options. That set depends on different parameters such as user requirements, application domain, etc. Constraint verification is another guidance mechanism. Selecting an option might, for example, require the inclusion or exclusion of another one. Many more constraints examples are available around us. Configurators should preclude inconsistent activation or deactivation of configuration options to avoid frustration on the user side and technically unrealistic products on the manufacturer side. Furthermore, constraints are of different natures. Some are of technical nature while others originate from business rules. Both may change over time.

Our experience reveals that some of those existing configurators are implemented in an ad-hoc fashion. This is especially cumbersome when numerous and non-trivial constraints have to be dealt with. For instance, we have observed in two industrial cases [46] that relationships between configuration options are hard-coded and mixed with GUI code. In other words, the configuration logic is not separated from the rest of the application code. As constraints are scattered in the source code, severe maintenance issues occur. For example, engineers are likely to introduce errors when updating or adding new constraints between options in the configurator. Moreover, as recognized by our industrial partners developing such configurators, the correctness and the efficiency of the reasoning operations are not guaranteed. More reliable and maintainable solutions are thus needed, especially for safety-critical systems.

We propose a pragmatic and model-driven way to generate configuration GUIs [18]. We rely on *Feature Models* (FMs) to represent and reason about the configuration options and their complex relationships. FMs have been extensively studied in academia during the last two decades, primarily in the software product line community [52]. FMs are now equipped with formal semantics [80], automated reasoning operations and benchmarks [4, 12], tools [7, 14, 54] and languages [8, 24]. In essence, an FM aims at defining legal combinations of features authorised or supported by a system. In our case, configuration options are modelled as features and each configuration (specification of a product) authorised by the configurator corresponds to a valid combination of features in an FM. A strength of FMs is that state-of-the-art reasoning techniques, based on solvers (e.g., SAT, SMT, CSP), can be reused to implement decision verification, propagation, and auto-completion in a rigorous and efficient way [8, 12, 49]. Therefore, FMs are a very good candidate to pilot the configuration process during which customers decide which features are included in a product.

Once FMs have been elaborated, there is still need to produce a GUI, including the integration of underlying reasoning mechanisms to control and update the GUI elements. On the one hand, some FM-based configuration GUIs rely on solvers [7, 14, 54]. But such GUIs do not consider presentation concerns and their generation process is rigid, avoiding the derivation of customised GUIs [43]. Furthermore, existing graphical representations of FMs (e.g., FODA-like notation or tree-views) are not adapted to user-friendly configuration [69]. On the other hand, model-based approaches for generating GUIs simply produce the visual aspects of a GUI [15, 16, 26, 42]. This is not sufficient for configurators since constraint verification is paramount for their usability and performance.

Our approach is to combine the best of both worlds, i.e., correct configurations together with user-friendly generated GUIs. We present a model-view-presenter (MVP) architecture to design configurators, which separates concerns between an FM (configuration option modelling), its associated solver (automated reasoning support) and the presentation of the GUI. To fill the gap between FMs and configuration GUIs, the different constructs of the FM formalism are rendered as GUI elements through model transformations. The transformations are based on a meta-model for TVL [20, 24], a textual language for feature modelling. Transformations can be parametrised through beautification and view languages to derive specific configuration GUIs.

The rest of the chapter is organized as follows. First, in Sect. 1.2, we give some background information about feature models and GUIs. The existing work linking feature models and GUIs is also addressed. In Sect. 1.3, an overview of our approach is proposed. Then, in Sect. 1.4, we present the implementation of the approach. It includes three different languages as well as a Web configurator generator. All the concepts are illustrated with throughout an IPv6 addressing plan configuration example. Finally, before concluding, we present the lessons learned in Sect. 1.5 and discuss some perspectives to our work in Sect. 1.6.

This book chapter is essentially based on a PhD thesis presented by the first author in September 2014 at the University of Namur (Belgium). For more detailed information about the approach, the interested reader may refer to [18].

## 1.2 Background

Here, we introduce the background required to understand the contents of this chapter as well as existing approaches that we compare to ours. Feature models being the starting point endeavour, we introduce them in Sect. 1.2.1. Then, in Sect. 1.2.2, we introduce UI-related concepts and generation.

### 1.2.1 Feature Modelling

Software Product Line Engineering (SPLE) is an increasingly popular software engineering paradigm which advocates systematic reuse across the software lifecycle. Central to the SPLE paradigm is the modelling and management of *variability*, i.e., “*the commonalities and differences in the applications in terms of requirements, architecture, components, and test artefacts*” [70]. Variability is typically expressed in terms of *features*, i.e., first-class abstractions that shape the reasoning of the engineers and other stakeholders [25].

Feature models were introduced as part of the FODA (Feature Oriented Domain Analysis) method 25 years ago [52]. They were introduced as graphical notations whose purpose is to document variability. Since their introduction, FMs have been extended and formalised in various ways [30, 80] and tool support has been progressively developed [74]. The majority of these extensions are variants of FODA’s original tree-based graphical notation.

Graphical FM notations based on FODA [52] are by far the most widely used. Most of the subsequent proposals such as FeatuRSEB [44], FORM [53] or Generative Programming [29] are only slightly different from the original graphical syntax (e.g., by adding boxes around feature names).

A number of textual FM languages were also proposed in the literature. Table 1.1 compares them against the following criteria: (i) *human readability*, i.e., whether the language is meant to be read and written by humans; (ii) support for attributes; (iii) decomposition (group) cardinalities; (iv) basic constraints, i.e., *requires*, *excludes* and other Boolean constraints on the presence of features; (v) complex constraints, i.e., Boolean constraints involving values of attributes; (vi) mechanisms for structuring and organising the information contained in an FM (other than the FM hierarchy); (vii) formal and tool-independent semantics, and (viii) tool support.

We should note that all these languages are remotely related to constraint programming, and several implementations use constraint solvers internally. Moreover,

**Table 1.1** Existing textual variability modelling languages

Language	Human readable	Attributes	Cardinalities	Basic Const.	Complex Const.	Structuring	Formal semantics	Tool support
FDL [32]	✓			✓			✓	
FMP [7]		✓	✓	✓		✓		✓
GUIDSL [8]	✓			✓				✓
FAMA [13]		✓	✓	✓	✓			✓
pure::variants [14]		✓	✓	✓	✓			✓
SXFM [60]	✓			✓				✓
VSL [78]	✓	✓	✓	✓				✓
KConfig <sup>1</sup>	✓	✓		✓		✓		✓

as pointed out by Batory [8], FMs can be seen as simplified grammars where products correspond to sentences. Similarly, FMs with attributes can be seen as a form of attribute grammar, albeit without the distinction of synthesised or inherited attribute [9, 55]. What distinguishes FMs from constraint programming and attribute grammars is their domain-specific nature and independence from any of these technologies.

## 1.2.2 User Interface Modelling and Generation

This section is decomposed into two sub-sections. In the first one, we give a short description of major user interface description languages which could be used as target languages for our generation approach. In the second, existing work combining variability models (more exactly FMs) and GUIs is presented.

### 1.2.2.1 User Interface Description Languages

In the Human-Computer Interaction (HCI) research domain, automation of UI development is an important topic. A whole spectrum of approaches ranging from purely manual design to completely automated approaches have been proposed. Manual design is of no interest to us as we seek to automate the generation of interfaces. On the other hand, fully automated approaches generate moderately usable GUIs, except for domain specific applications [64].

Most approaches propose a partially automated process which uses extra information about the UI stored in models. They are all grouped under the *Model-based User Interface Development* (MUID) denomination, generally supported by an

MBUID environment (MBUIDE). It can be defined as “*a suite of software tools that support designing and developing UIs by creating interface models*” [42]. Each MBUIDE defines its own set of models to describe the interface. The different MBUIDEs and the associated models have been surveyed by Gomaa et al. [42] and the W3C [87]. Here, we give a summary of User Interface Description Languages (UIDLs) used in MBUID. XML-based UIDLs have also been surveyed by several authors [41, 81]. Such languages can be used to represent the generated GUIs at a more “abstract” level. They are grouped in four categories.

The first category groups all languages based on the *Cameleon Reference Framework* (CRF) [22]. There, the UI development is decomposed into four abstraction levels: Task and Concepts (T&C), Abstract User Interface (AUI), Concrete User Interface (CUI) and Final User Interface (FUI), the last being the most concrete one. T&C is computing independent, AUI is modality independent and CUI is platform independent. This framework is globally well accepted by the UI community as shown by the numerous MBUID approaches which, directly or indirectly, rely on it to define their models and development processes. Among them, we can mention the *Software Engineering for Embedded Systems using a Component-Oriented Approach* [33, 73], *Model-based Language for Interactive Applications XML* (MARIA XML) [65], or *User Interface eXtensible Markup Language* (UsiXML) [58]. Among all those approaches/languages, the last one is probably the most mature while most others seem abandoned.

The *User Interface Markup Language* (UIML) [6, 45] and its derivative, the *Dialog and Interface Specification Language* (DISL) [62] make part of the second category. UIML has been defined by the OASIS consortium<sup>2</sup> which seeks to develop standards for e-business and Web services. The language must be combined with other techniques such as user task modelling or transformation algorithms in order to be able to generate a full-fledged UI. In UIML, look-and-feel, interaction and connexion of the UI with application logic can be defined.

The third category contains Web-application languages. Initially, XForms [86] was defined for HTML-XHTML documents by the W3C. Its purpose is to separate presentation from data in Web forms in order to improve re-use. Now, XForms can be used with any markup language. XForms is not an UIDL per se but allows to define GUIs at an abstract level. Second, XICL [82] is meant to develop user interface components for browsers. Lastly, the *eXtensible user-Interface Markup Language* (XIML) [72] represents interaction data for Web pages and applications at abstract and concrete levels.

Finally, we can also mention the following languages which do not fit into any of the above categories. The *Generalized Interface Markup Language* (GIML) is an UIDL used in the *Generalized Interface Tool Kit* (GITK) project [56]. The *Multiple Device Markup Language* (MDML) supports four target environments [51]: desktop, mobile, Web and voice. Similarly, the *Simple Unified Natural Markup Language* (SunML) [66] supports several target environments such as PCs, PDAs

---

<sup>2</sup>See <https://www.oasis-open.org/>

**Table 1.2** Existing user interface description languages

Language	GUIs	Other UIs	Maintained	Tools developed	Tools available
UsiXML [58]	✓	✓	✓	✓	
UIML [6, 45]	✓			✓	
XForms [86]	✓		✓	✓	✓
GIML [56]	✓			✓	
MDML [51]	✓	✓		✓	
SunML [66]	✓	✓		✓	
TADEUS-XML [63]	✓			✓	

or voice. The *Adaptable & Mergeable User Interface* (AMUSING) IDE provides tool support to edit SunML models and generate Swing software [66]. Finally, in TADEUS-XML [63], a UI description is made of two parts: a presentation component and a model component (or abstract interaction model).

None of the approaches proposed with these languages addresses the specific issues that arise when generating configurators like the integration of underlying reasoning mechanisms for controlling and propagating user choices in the GUI. Modelling techniques have been developed to support adaptations of interfaces at runtime [15, 16]. In the same way, configurators should be adapted to reflect the user interactions (i.e., selections/deselections). In our context, the kind of modifications applied to the configurator interfaces are typically lightweight (e.g., some values are greyed) and can be predicted. Moreover, we can take advantage of planned variability to make use of efficient solvers to manage the configuration process.

Our user interface description languages comparison is summarized in Table 1.2.

### 1.2.2.2 Feature Models and GUIs

In most variability-related tools, FMs are represented and configured using tree-views. We can, for example, mention pure::variants [14], FeatureIDE [54] or Feature Modeling Plug-in [7]. Those tools have a graphical interface in which users can select/deselect features in a directory-tree like interface where constraints are automatically propagated. Several visualization techniques have been proposed to represent FMs [69], but they are not dedicated to end users which are more accustomed to standard interfaces such as widgets, screens, etc. Generating such user-friendly and intuitive interfaces is the main goal of our work. An exception is the AHEAD tool suite of Grechanik et al. [43]. Simple Java configuration interfaces including check boxes, radio buttons, etc. are generated using beautifying annotations supported by the GUIDSL syntax used in the tool suite.

Pleuss et al. combine SPLs and the concepts from the MBUID domain to integrate automated product derivation and individual UI design [68]. An AUI is defined in the domain engineering phase and the product-specific AUI is calculated during the application engineering. The final UI is derived using semi-automatic approaches from MBUID. Some elements like the links between UI elements and application can be fully automatically generated while others like the visual appearance are also generated automatically, but can be influenced by the user. While we share similar views regarding MBUID, our overall goals differ. Pleuss et al. aim at generating the UI of products derived from the feature model while our interest is on generating the interface of a configurator allowing end users to derive product according to their needs. We are therefore not concerned with product derivation but rather with the link between feature model configuration and UIs.

Schlee and Vanderdonck [79] also combined FMs with GUI generation. Relying on the generative programming paradigm, the authors represent the UI options with an FM which will be used to generate the corresponding interface. Their work illustrates a few transformations between FM and GUI constructs which can be seen as patterns. Yet, they do not consider sequencing aspects which we believe to be a critical concern for complex UIs. Gabillon et al. extended that work by supporting multi-platform UIs built from FMs representing UI options [39]. However, they do not tackle UIs which allow the configuration of an FM.

Quinton et al. proposed a model-driven framework called *AppliDE* that bridges the gap between an application FM and its mobile version [75]. Their main purpose is to reduce the time-to-market between the design of the application and its availability on multiple platforms. Based on the meta-model of the configured product and the one representing the capabilities of smartphones, they can deduce which device is able to run the application. Similarly to us, they use model transformations to finally generate GUIs. However, their approach does not focus on configurators and is limited to mobile phone software.

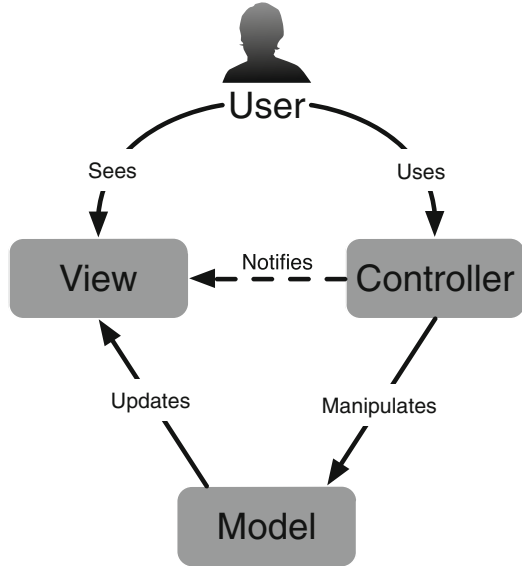
Botterweck et al. developed a feature configuration tool called *S<sup>2</sup>T<sup>2</sup> Configurator* [17]. It includes a visual interactive representation of the FM and a formal reasoning engine that calculates consequences of the user's actions and provides formal explanation. This feedback mechanism is of importance to end users. Yet, *S<sup>2</sup>T<sup>2</sup>* also presents a tree-like view on the configuration that we believe is not suited to all kinds of end users.

### 1.3 The MVP Configurator Pattern

Several architectural models have been introduced to structure modules such as the GUI in an interactive application. Among them, the model-view-controller (MVC) has wide acceptance in the development of GUIs. One reason is that it is one of the first serious attempts to structure UIs, dating back to the late 1970s. In December 1979 at the Xerox Palo Alto Research Laboratory (PARC), Trygve Reenskaug first described the MVC pattern [77].



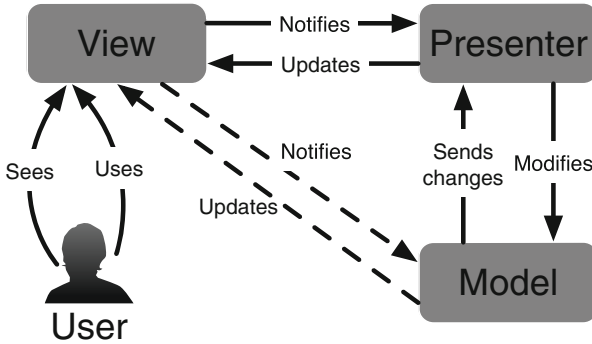
**Fig. 1.1** Model-view-controller architecture



In this paradigm, *Models* represent knowledge. They could be a single object or a structure of objects. *Views* are (visual) representations of their corresponding model. They basically highlight some attributes and suppress others, acting as a “presentation filter”. Finally, *Controllers* act as the link between a user and the system. The idea behind this pattern is to make a clear distinction between domain objects which model real world elements, and GUI elements depicted on the screen.

The MVC architecture defined by Reenskaug is depicted in Fig. 1.1. There, the *Model* manages the data and behaviour of the application domain. It responds to requests about its current state (usually from the *View*) or requests instructions to change its state (usually from the *Controller*). The *View* simply manages the layout of the information contained in the *Model*. This might require to query the state of the *Model*. Finally, the *Controller* interprets inputs from the user (keyboard, mouse, etc.) and informs the *Model/View*.

In [21], Burbeck presents two variants of the MVC pattern where the role of the model varies: active or passive. In the passive version, the model is exclusively modified by the controller (i.e., it cannot be modified by any other source). As soon as the controller detects a user action, it modifies the model and informs the view that the model has changed and should be refreshed (*Notifies* dotted line in Fig. 1.1). In this scenario, the model is unaware of the existence of the view and the controller. In the active version, the state of the model can be changed by an external component. Since only the model can detect that it has been changed, it needs to notify the view that it must be refreshed. The observer pattern [40] is generally used to keep the model independent from the other components. Views subscribe to be informed of the changes in the model.



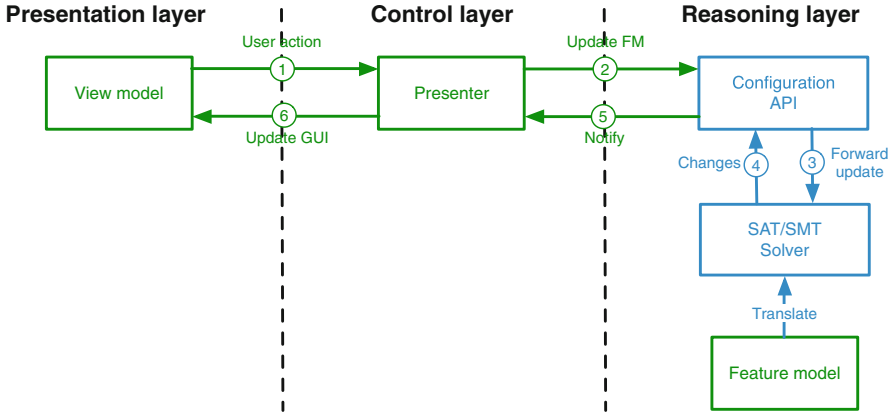
**Fig. 1.2** Model-view-presenter architecture

We rely on an MVC variant – model-view-presenter (MVP) [71] – to propose a generic architecture for configuration interfaces. It separates the responsibilities for the visual display and the event-handling behaviour into two different components named *View* and *Presenter*, respectively. The *View* detects changes in the GUI and forwards the corresponding events to the *Presenter*. That component contains the logic to handle those events. Centralizing the behaviour inside a single component makes it easier to test, and its code can be shared between different views that have the same behaviour. As for the MVC architectural pattern, MVP comes in two versions: passive view and supervising controller. They are depicted in Fig. 1.2. In the passive version, interactions between the *View* and the *Model* are handled exclusively by the *Presenter*. In the other one, the *View* can directly interact with the *Model* for simple events, more complex ones still being handled by the *Presenter*. In Fig. 1.2, dashed lines correspond to interactions specific to the supervising controller version.

The key idea of our approach is to separate variability reasoning at the FM level, event handling and the actual representation of the GUI. Thus, our architecture is inspired by the passive view version of the MVP pattern and is decomposed into three tiers (see Fig. 1.3).

Here, we focus on the MVP-related models (shown in green in Fig. 1.3) while the supporting components (in blue) are considered as third-party software. The roles involved in our adaptation of the pattern are as follows:

- **Model:** The model is an FM. The feature model is used to effectively engineer a configuration GUI. It is connected to a reasoning engine which is responsible of interactive configuration and is exposed through a generic API.
- **View:** The view contains a description of the GUI to be displayed to the user. This description is generated from the FM using transformation rules. Ideally, rather than generating the interface in its implementation language, a GUI model should be generated for it. This has two advantages; (i) GUI models are more concise and thus easier to generate and (ii) we can target several platforms from the same GUI model.



**Fig. 1.3** An MVP architecture for configurators

- **Presenter:** The presenter is the central point of our architecture. It listens to user actions, updates the FM and interacts with the reasoning engine to determine the list of changes to be propagated to the GUI. Once this list is populated, it updates the GUI model by adding, removing, hiding, making visible or updating elements affected by the changes.

From a dynamic perspective, interaction between components works according to the numbered arrows. The preliminary step is to translate the FM in a format compatible with the SAT/SMT solver. This translation is made once and allows efficient reasoning by exploiting this robust technology. Once an instance of the FM is encoded within the solver, the configurator can be used interactively. For example, ticking a check box in the GUI will trigger an event through the view model and will be propagated to the presenter (① *User action*). Depending on the nature of this action, the presenter will generate an update request (② *Update FM*) for the configuration API. This API will in turn update the FM instance (e.g., by setting a Boolean variable corresponding to the feature associated with the check box to `true` via ③ *Forward update*). The solver will compute the new list of features to be (de)selected as a result (④ *Changes*). This result will be transferred to the presenter (⑤ *Notify*) that will make decisions regarding changes in the GUI. The GUI is then updated (⑥ *Update GUI*) accordingly.

Our architecture does not use the supervising presenter version of the original MVP pattern in the sense that there is no direct link between the FM and the view model. The main reason is that interactive configuration can induce complex GUI updates for which a specific behaviour has to be provided. Since most of this behaviour can be made generic, presenters can be reused amongst different GUIs.

## 1.4 From Feature Models to MVP Configurators

### 1.4.1 Illustration

In this section, we illustrate the different languages and components of our approach by modelling a configurator for computer network topologies and IPv6 addressing plans. Preparing an IPv6 addressing plan is an important task for network managers who need to deploy IPv6 in their organizations.

One of the core networking aspects found in addressing plans is the practice of dividing a computer network into multiple networks called *subnets*. The computers that belong to the same subnet have their IP addresses prefixed by a common bit-group and the exchange of traffic between different subnets is supported by routers. The purpose of an addressing plan is to logically divide the network into subnets based on the structure of the organization so that the IPv6 addresses can be effectively managed in groups. This split can greatly simplify the management of networks, especially within large organizations.

Throughout the remainder of this section, we present the different models supporting the generation of GUIs for a configurator that can assist practitioners in their preparation of an addressing plan. We also introduce the required computer network concepts for understanding this domain-specific configuration task.

### 1.4.2 Variability Modelling

#### 1.4.2.1 General Principles and Language

As previously mentioned, FMs are the base models of our approach. However, while they are the de-facto standard for representing the variability in the scientific community, our industry partners, discussions at the 2010 variability modelling (VaMoS) workshop [11] as well as literature reviews [23, 48] suggest that in the industrial world, in contrast, FMs appear to be used rarely. In [46], some of the authors of this chapter identified their shortcomings. To overcome those shortcomings, these authors also designed TVL (Textual Variability Language), a text-based FM language. The idea of using text to represent variability in SPLE is not new [9, 32] but seems to be recently gaining popularity [3, 28]. In terms of expressiveness, TVL subsumes most existing dialects. The main goal of designing TVL was to provide engineers with a human-readable language with a rich syntax to make modelling easy and models natural. Further goals for TVL were to be *lightweight* (in contrast to the verbosity of XML for instance) and to be *scalable* by offering mechanisms for structuring the FM in various ways.

Basically, the TVL language has a C-like syntax: it uses braces to delimit blocks, C-style comments and semicolons to delimit statements. The rationale for this syntax choice is that nearly all computing professionals have come across a C-like syntax and are thus familiar with this style. Furthermore, many text editors have built-in facilities to handle this type of syntax.

In TVL, the *root* keyword is used for the root feature and each decomposition is introduced by the *group* keyword, which is followed by the decomposition type. The *and*, *or*, and *xor* decomposition types were renamed to *allof*, *someof* and *oneof* in TVL. These names are inspired by [32] and make the language more accessible to people not familiar with the Boolean interpretation of decomposition. The decomposition type can also be given by a cardinality. Cardinalities can use constants, natural numbers, or the asterisk character (which denotes the number of children in the group). The decomposition type is followed by a comma-separated list of features, enclosed in braces. If a feature is optional, its name is preceded by the *opt* keyword. Each feature of the list can declare its own children. If each feature lists its children this way, the tree structure of the FM will be reproduced in TVL with nested braces and indentation. This can become a scalability problem for deep models, something we experienced in industrial cases. To this end, TVL allows one to declare a feature in the decomposition of its parent by just providing a name. A declared feature can then be extended later on in the code. Besides the *group* block, a feature can contain constraint and attribute declarations, all enclosed by a pair of braces. If there is only a *group* block, braces can be omitted. This reduces the number of braces in a pure decomposition hierarchy. To model a Directed Acyclic Graph (DAG) structure (as in FORM [53]), a feature name can be preceded by the *shared* keyword, meaning that it is just a reference to a feature already declared elsewhere.

Attributes can be defined inside the body of a feature. They are declared like variables in C, in order to be intuitive for engineers. The attribute types supported by TVL are integer (*int*), real (*real*), Boolean (*bool*), and enumeration (*enum*) whose values set is specified with the *in* keyword. TVL further provides syntactic sugar to define the domain and the value of an attribute. If the value of an attribute depends on whether its parent feature is selected or not, the *ifIn*: and *ifOut*: keywords can be used. Furthermore, to concisely specify cases in which the value of an attribute is an aggregate of another attribute that is declared for each child, an aggregation function can be used in combination with the *children* and *selectedChildren* keywords (followed by an *ID* denoting the attribute).

In TVL, constraints are Boolean expressions inside the body of a feature. There is also syntactic sugar for guarded constraints. Constraints can be guarded using the same *ifIn*: and *ifOut*: guards as for attributes. The *ifIn*: guard means that the constraint only applies if the parent feature is selected. To facilitate specifying constraints and attribute values, TVL comes with a rich expression syntax. The syntax is meant to be as complete as possible in terms of operators, to encourage writing of intuitive constraints. For instance, to restrict the allowed values of an enum, the set-style *in* operator can be used. For enum *e* in {*a*, *b*, *c*, *d*, ...}, the constraint *e* in {*b*, *c*} serves as syntactic sugar for *e* != *a* && *e* != *d* && ..., which is much less readable.

TVL offers two mechanisms that can help engineers structure large models. The first is the *include* statement, which takes as parameter a file path. As expected, an *include* statement will include the contents of the referenced file at this point. Includes are in fact preprocessing directives and do not have any meaning beyond

the fact that they are replaced by the referenced file. Modellers can thus structure the FM according to their preferences. The second structuring mechanism, hinted at before, is that features can be defined in one place and be extended later in the code. Basically, a feature block may be repeated to add constraints and attributes to the feature. These mechanisms allow modellers to organise the FM according to their preferences and can be used to implement separation of concerns [83]. This way the engineer can specify the structure of the FM upfront, without detailing the features. Feature attributes and constraints can be specified in the second part of the file, or in other files using the *include* statement. The only restriction is that the hierarchy of a feature can only be defined at one place (i.e., the *group* keyword can only be used once for each feature).

More detailed information about TVL can be found in [20, 24].

## TVL<sub>2</sub>

Hereabove, we introduced TVL as we initially defined it [20, 24]. In the meantime, the language has been extended by other researchers in our laboratory. The purpose of those extensions is to support all constructs found in industrial cases. Basically, three main constructs were added, string attributes, feature cardinalities and feature references.

A string attribute is defined using the *string* keyword. Similarly to other attribute types, an ID is then given to the attribute. The naming convention is the same, the attribute ID has to start with a lower case letter. For example, “string myString” is a valid attribute declaration. It is also possible to define string constants in TVL<sub>2</sub>.

In the original TVL syntax, each feature can be configured (at most) once. Like most existing languages, ours lacks a construct that allows to duplicate a sub-tree of the FM to configure a product. TVL<sub>2</sub> now supports so-called feature cardinalities. Their semantics is defined elsewhere [61] and will not be addressed here. Syntactically, feature cardinalities are represented in a similar way to group cardinalities, with bounds between brackets. The cardinality directly follows the name of a feature. If it is not defined, the [1..1] cardinality is assumed. Furthermore, the root feature cannot have a cardinality, i.e., it still has to be unique. Bounds can be either an integer value or a constant, or the asterisk character. Here, the asterisk character means that the number of feature instances is unlimited.

A feature reference is an attribute which value identifies an instance of a multi-feature. It is declared by using the keyword *shared* and the type of the targeted multi-feature. For example, “shared F myFeatureRef” represents a feature reference which name is myFeatureRef and which type is F. If we assume that the cardinality of F is [0..2], then the value of myFeatureRef can be either *F-0* or *F-1* which represent the two potential instances of F.

### 1.4.2.2 Addressing Plan Example

We present a TVL model for the configuration of subnets and the allocation of IPv6 addresses. The model is visible in Listing 1.1. There, constraints have been removed in order to keep the code as compact as possible. The root feature is decomposed into four sub-features. The feature named `Subnet` (lines 18–23) contains information related to a subnet such as its name or its IPv6 prefix. It also contains two feature references that target the sibling features `UseType` (lines 24–27) and `Location` (lines 28–31). These two features represent the *groups* that are defined within an addressing plan and that determine how IPv6 address blocks will be distributed in the organization. For example, in the case of a university campus, the groups could be defined by a set of use types such as *student*, *staff* or *professors* which refer to the different types of users on the network, and by a set of locations such as *computer sciences* or *economics* which refer to the different faculty buildings on the campus. By identifying each subnet by a pair of use-type and location, the addressing plan guarantees that the IPv6 addresses will be consistently distributed. For example, it can ensure that all students in economics will be assigned an IP address from the same subnet. Below the root feature, there are six attributes (lines 6–15). The attribute `networkPrefix` represents the IPv6 prefix of the network. The attribute `strategy` indicates whether subnets are primarily identified by use types or by locations. `useTypes` indicates the total number of use types for the addressing plan and `futureUseTypes` represents the number of new use types that could emerge in the future. Likewise, `locations` indicates the total number of locations and `futureLocations` indicates the number of potential future locations. The feature `Host` (lines 32–56) contains information related to hosts on the network. The attribute `subnet` represents the subnet which the host belongs to. The feature `Interface` (lines 36–48) represents the communication interfaces through which the host sends packets to other hosts on the network. The feature `ConnectedInterface` (lines 44–46) represents the interfaces that belong to neighbour hosts and which the host can directly send packets to. Finally, the feature `RoutingTableEntry` (lines 49–54) represents lines in the routing table of the host. The attribute `destination` represents the addresse(s) that must be eventually reached by the sent packets. The attribute `sendingInterface` represents the local interface from which the host sends packets, while the attribute `nextHop` represents the neighbor interface which the host must forward the packets to.

**Listing 1.1** TVL model (excl. constraints) for the IPv6 addressing plan configurator,

```
1  enum GroupingStrategy in {LocationFirst , UseTypeFirst };
2
3  root AddressPlan {
4
5      // Address space
6      string networkPrefix;
7      GroupingStrategy strategy;
```

```

8
9 // Use types
10 int useTypes;
11 int futureUseTypes;
12
13 // Locations
14 int locations;
15 int futureLocations;
16
17 group someof {
18     Subnet [0..*] {
19         shared UseType useType;
20         shared Location location;
21         string subnetName;
22         string subnetPrefix;
23     },
24     UseType [0..*] {
25         string useTypeName;
26         string useTypePrefix;
27     },
28     Location [0..*] {
29         string locationName;
30         string locationPrefix;
31     },
32     Host [0..*] {
33         string hostName;
34         string loopback;
35         group someof {
36             Interface [0..*] {
37                 string index;
38                 string macAddress;
39                 real delay;
40                 shared Subnet subnet;
41                 string ipAddress;
42
43                 group allof {
44                     ConnectedInterface [0..*] {
45                         shared Interface connectedInterface;
46                     }
47                 }
48             },
49             RoutingTableEntry [0..*] {
50                 string destination;
51                 int metric;
52                 shared Interface sendingInterface;
53                 shared Interface nextHop;
54             }
55         }
56     }
57 }
58 }

```



### 1.4.2.3 Widget Selection

When thinking about GUI generation, the first task that comes to mind is to translate the different FM constructs into graphical widgets. In other words, the question is: How should the different TVL constructs be rendered in a configurator? For this purpose, we have analysed some existing software configurators [2]. More specifically, 111 Web-based configurators were investigated since they represent a significant share of existing GUIs today. The (less formal) analysis of configuration GUIs implemented in other technologies has confirmed most findings. “*How are configuration options visually represented and what are their semantics?*” is the research question which helped us to identify the types of widgets, their frequency of use, and their semantics (i.e., the corresponding FM constructs). In decreasing order, the most popular widgets in Web-configurators are: *combo box item*, *image*, *radio button*, *check button* and *text box*. Some of them are also combined with images, namely *check button*, *radio button* and *combo box item*. In that case, option selection is performed either choosing the image or using the widget. Other less frequent widgets are *slider*, *label*, *file picker*, *date picker*, *colour picker*, etc.

The most significant outcome of this empirical study is that the range of graphical widgets is not very large. Actually, according to our analysis, only five of them seem sufficient to represent most variability constructs. We could thus confine ourselves to those widgets, but this would too drastically limit our approach which aims to be generic. It is therefore necessary to propose a more flexible mapping in order to meet user requirements. Nevertheless, we should also impose some restrictions to ensure the generation of “coherent” GUIs. By coherent, we mean that a widget representing a given variability construct should reflect its semantics. For example, *check boxes* should be avoided to represent *xor-decompositions* to avoid confusion. Note that this could be mitigated by adding a *label* warning the user that the choices are mutually exclusive.


We thus proposed a mapping between FM constructs and GUI widgets. Customization of the interface is made possible by offering several widgets for most variability constructs. All those mappings are summarized in Table 1.3. It is divided into three main categories: Groups, Attribute types, and Features & Attributes. The second column represents the different constructs of each category. The name of the different widgets associated to each construct are displayed in the third column and illustrated in the HTML format in the last one.

## 1.4.3 View Definition

### 1.4.3.1 General Principles and Language

Previously, we presented mappings between FM constructs and GUI widgets. That might be adequate for simple FMs but the limits of such a simple transformation are rapidly reached. First, it does not take the different concerns that might be included

**Table 1.3** Graphical widgets mappings

Category	Construct	Widget	HTML example
Groups	<i>and</i> ( <i>optional features</i> )	Check button	<input type="checkbox"/>
		List box	<input type="text" value="True"/>
		Radio box	<input checked="" type="radio"/> True <input type="radio"/> False
	<i>or</i>	List box	<input type="text"/>
		Check box	<input type="checkbox"/>
	<i>xor</i>	List box	<input type="text"/>
		Radio box	<input type="radio"/>
Check box		<input type="checkbox"/>	
Attribute types	<i>integer</i>	Text box	<input type="text" value="0"/>
		Slider	<input type="range" value="5"/>
	<i>real</i>	Text box	<input type="text" value="0,0"/>
		Slider	<input type="range" value="4.9"/>
	<i>Boolean</i>	Check button	<input type="checkbox"/>
		List box	<input type="text" value="True"/>
		Radio box	<input checked="" type="radio"/> True <input type="radio"/> False
<i>enumeration</i>	List box	<input type="text"/>	
	Radio box	<input type="radio"/>	
Features & Attributes	<i>feature/attribute</i>	Label	Feature label
		Image	

in an FM [83] into account. The groups of logically linked constructs vary from person to person and should be taken into account while generating configuration GUIs. Furthermore, the structure of the generated GUI will be strongly related to the FM hierarchy. Indeed, during the generation process, the FM will, in most cases, be traversed using a *depth-first* approach in order to generate a feature together with its contents, thus resulting in “nested” and “staired” GUIs. Nested since the widgets corresponding to the contents (attributes or group) of a feature will be displayed inside (or under) the widget corresponding to the feature itself. Staired as the width of the generated GUI will depend on the depth of the FM assuming that an horizontal offset between a feature and its contents exists in the GUI. This offset will be used in most cases in order to depict the relationship between a feature and its contents. The deeper the FM, the wider the generated GUI. While those staired GUIs may be valuable in some cases, they quickly become cumbersome.

To break out of the FM hierarchy, we propose to use views on them. Views are “*a simplified version of an FM that has been tailored for a specific stakeholder, role, task, or, to generalize, a particular combination of these elements, which we call*

*a concern. Views facilitate configuration in that they only focus on those parts of the FM that are relevant for a given concern. Using multiple views is thus a way to achieve separation of concerns in FMs” [50].*

One of the benefits of views is that they allow to break the hierarchy defined in the FM. However, in some cases this hierarchy is still valuable in the configuration GUI. Consequently, the view definition language should allow to split the FM hierarchy while providing mechanisms to keep the tree structure inherent to such models, at least for sub-parts of it. In the following, some desirable characteristics of such a language are pointed out:

- **Full sub-tree** – It should be possible to select a sub-tree of the FM. This selection would preserve the structure of the original model. A sub-tree is composed of its root (which can be the FM root or any other feature) and optionally a list of features to exclude (incl. their sub-features and attributes) from the selection, a so-called *stop list*. The full FM is a specific case where the root of the sub-tree is the FM root and the feature stop list is empty.
- **Partial sub-tree** – Similarly, it should be possible to select elements in a given sub-tree. This sub-tree would also be defined by a root feature and optionally a stop list. Then, it would be possible to include or exclude some elements like a feature and its contents, an attribute, all groups, all attributes, etc. Here, the structure of the FM is not preserved since the purpose is to select some elements inside a sub-part of it.
- **Feature** – It should be possible to select a feature and its contents. Mechanisms to select only parts of feature’s contents should also be provided.
- **Attribute** – Selection of an attribute, and its sub-attributes for structured ones, should also be possible.

We propose TVDL (Textual View Definition Language), a text-based view definition language which presents those characteristics for TVL. However, it could easily be applied to any other variability modelling language. As for TVL, the goal of TVDL is to supply engineers with a human-readable and lightweight language.

In TVDL, a view model has to import a TVL FM and is composed of a collection of `Views`. Basically, a view is given a name and has contents. Its name is a character string starting either with an upper-case or lower-case character. This name must be unique and can thus be used as ID for the view. The contents are then enclosed in braces. Similarly to TVL feature extensions, there is no separator (e.g., semicolon) between the different TVDL views.

We implemented the four different types of views introduced above in TVDL. Additionally, we propose grouping views which are composed of a set of sub-views previously declared. The name of grouping views is preceded by the dollar sign. Each view is composed of one or several view expressions which can be combined using the `&&` symbol. And each view expression references either a TVL feature or attribute.

The first view expression type, full sub-tree, is defined using the asterisk character. A so-called *stop list* can be defined to determine the branches of the FM which are not covered by the view expression. The branch is thus pruned before the

stop list element, i.e., it is not included. A stop list is composed of stop elements which are a TVL feature name or its (fully) qualified name preceded by the slash character. A stop list is composed of at least one stop element.

In the partial sub-tree expression, the sub-tree is used as search space. Its purpose is to select attributes only, to exclude some features or attributes, to exclude all attributes or groups, etc. contained in a given sub-tree. In this kind of view, the hierarchy is not preserved since one can exclude some elements, so breaking the hierarchy and creating confusion about the semantics of the partial FM. The difference with full sub-tree views is the filter added to the partial sub-tree selection. Three different sub-tree filters exist. They all start with the pipe character. The first one is lists. A list can either be an inclusion or an exclusion (preceded by the exclamation mark) one. The coverage of an inclusion list is the union of elements covered by each of the list elements. Conversely, the coverage of a sub-tree expression refined by an exclusion list is the difference between the set of elements covered by the sub-tree expression and the set of elements covered by the list elements. List elements can be, regardless of the list type, IDs of TVL features or attributes, *attributes* or *groups* keywords. Those elements can be mixed inside the same list and TVL IDs must refer to constructs covered by the sub-tree expression. If a feature ID is included in an exclusion list, this feature as well as all its contents (attributes and group) will be excluded from the view. Conversely, in an inclusion list, the feature and its contents only will be included in the view coverage. Attribute IDs included in an exclusion (resp. inclusion) list will be excluded (resp. included) in the view coverage, as well as sub-attributes for structured attributes. The *groups* keyword in an exclusion (resp. inclusion) list will exclude (resp. include) all groups from the view coverage. The same principle applies to the *attributes* keyword. Attributes are the second kind of refinement for sub-tree expressions. The *attributes* keyword is used for this purpose. It means that the view covers all attributes contained in the sub-tree expression. It is also possible to further refine the view with a refinement list which can either be an inclusion or exclusion one but, in this case can contain only IDs of TVL attributes covered by the sub-tree expression. This refinement list is also preceded by the pipe character. Finally, it is also possible to select all feature groups contained in a sub-tree of an FM with the *groups* keyword. In this case, the view coverage is a set of feature groups. It is possible to refine this groups expression with an inclusion/exclusion list (preceded by the pipe character). But, in this case, the list contains TVL feature IDs only. We chose to allow features since it is the only way to identify feature groups in TVL. If a feature is covered by an inclusion (resp. exclusion) list, its group will (resp. will not) be covered by the groups expression.

In TVDL, it is also possible to select a single feature in a view. Similarly to partial sub-trees, refinements exist for those feature selections. The only difference is that the *group* keyword has to be used instead of *groups* in the case of partial sub-trees given that each feature contains (maximum) one group in TVL. Finally, refinement lists can also be defined on features. As for partial sub-trees, it can either be an inclusion or exclusion list. This list can contain the TVL ID of the feature's attributes, and/or the *group* or *attributes* keywords. For inclusion lists, the view will cover the feature itself plus the elements mentioned in the list.

The last kind of expression, namely attributes, is the simplest one. Indeed, we have chosen to disallow their refinement. The only way to refine attributes would be to select only some sub-attributes of a TVL structure attribute. But, given our experience in variability modelling, it makes no sense to split such attributes. Indeed, if they had to be split, they would have been represented as a feature with attributes.

### 1.4.3.2 Addressing Plan Example

The TVDL model for the addressing plan configurator is given in Listing 1.2. At line 1, the TVL model previously introduced is imported before defining the four different views. The first one, `MainTab` (line 3) contains the global properties of the addressing plan that is currently configured. The second view, `SubnetTab` (line 5), displays information related to subnets and groups in the organization (i.e. use types and locations). The third view, `InterfaceTab` (line 7), shows information related to the communication interfaces of the hosts that are on the network. Finally, the fourth view `RoutingTableTab` (line 9) shows the routing table entries of the interfaces. Stop lists are used for defining `MainTab`, `InterfaceTab` and `RoutingTableTab`.

**Listing 1.2** TVDL model for the addressing plan example

```

1  import "addressing_plan_demo.tvl"
2
3  MainTab { AddressPlan :*/ Subnet/ UseType/ Location/ Host }
4
5  SubnetTab { Subnet && UseType && Location }
6
7  InterfaceTab { Host :*/ RoutingTableEntry }
8
9  RoutingTableTab { Host :*/ Interface }
```

## 1.4.4 Widget Selection

As for FM constructs, we propose a mapping between views and GUI widgets. Each view can be depicted either as a `Tab` or as a `Window`. `Tabs` could be nested in other `Tabs` or `Windows`, but not conversely.

## 1.4.5 Beautification

### 1.4.5.1 General Principles and Language

In the previous sections, our focus was on the direct translation of FM constructs into GUI elements. Even if this translation is technically feasible, the result would be rough as it relies only on information contained in FMs which is rather technical. For example, using feature and attribute names as label for the input fields might not be expressive enough to understand their meaning.

A first solution would be to extend existing languages. Missing information would be directly added in TVL and TVDL. At the first glance, this solution seems to be the best one in the context of configuration GUI generation. All information would be located in the same place. While this might help to design configuration GUIs, variability and view models would be cluttered with GUI-related information. This information is completely irrelevant in other contexts and might disturb variability modellers. We want to keep TVL and TVDL languages independent of the GUI generation process in order to preserve the separation of concerns [83]. For all those reasons, we chose to propose a new language dedicated to GUI-specific information.

This language plays the same role as CSS (Cascading Style Sheets) [85] for HTML pages, i.e., it contains beautification information. For this reason, we called our language FCSS, standing for *Featured Cascading Style Sheets*. As in usual CSS, properties include layout information but also feature-specific visualisation strategies. Other properties are related to the rendering of TVL attributes and groups, and TVDL views. The availability of certain options may also depend on the target language.

An FCSS Beautification Model refers to a TVL model and optionally to a TVDL one. Then, it is composed of four different kinds of parts, namely Global Properties, View Properties, Feature Properties and Attribute Properties.

Global properties definition sections start with the dot character and are, like the three other categories, delimited by curly braces. Several global sections can exist. However each global property can only be defined once in the whole model, i.e., it can neither be defined several times in the same global part nor in different global parts.

A property has a name, and a value separated by a colon. It is closed by a semicolon. Fourteen global properties exist, five are related to feature groups, four to features, another four to attributes, and a single one for views.

#### Global Properties

A global group property exists for each kind of TVL decomposition. For *and*-decompositions, it is named *andGroup* and can take a single value, namely *textbox*,

at the moment. Setting this property might thus be useless. Our intent is to extend the language in the light of experience with Web configurators, requests from customers, etc. It can be seen as a variation point whose variants still have to be defined. *orGroup* is a second global group property which can take either *listbox* or *checkbox* as value. *xorGroup* is the third property and its available values are *listbox* and *radiogroup*. The last kind of groups, *card*-decompositions, is represented by the *cardGroup* property and can, at the moment, take a single value, namely *checkbox*. Finally, the Boolean *groupContainer* property is used to determine whether groups and their sub-features have to be visually grouped together in the rendered configuration GUI. This is typically done with a bordered box.

The first property dedicated to features is simply called *feature* and determines how they are rendered in the GUI. Available values are *text* and *image*. Those values speak for themselves. The *optFeature* property determines how optional features have to be rendered. Three values exist, *checkbox*, *listbox*, and *radiogroup*. With a check box, the optional feature is selected if (and only if) it is checked. The listbox contains two values, true and false. Similarly the radio group contains two radio buttons labelled with the same Boolean values. Note that, optional features are generally used with *and*-decompositions. That may help explain why the *andGroup* property has a single value. *unavailableContent* is the third feature property. It can take three values, *hidden*, *greyed*, or *none*. This value determines the strategy to apply with the contents of a feature when the latter is not selected. It can either not be visible to the user (*hidden*), or visible but not editable (*greyed*), or visible and editable (*none*). With this last option, the user can select any option at any time. Given the structure of an FM, setting the value of a construct (attribute or feature) will automatically select all its ancestors in the configuration GUI. Finally, a *selectFeature* property exists and can take the same values as *optFeature*, namely *checkbox*, *listbox*, or *radioGroup*. In TVDL, we allow to not cover a group if all its sub-features are covered. As a consequence, the group is not rendered in the configuration GUI. Given that all its sub-features are depicted, we propose to use a selection widget in front of all of them, similarly to optional features. In this way, the user is still able to select group's sub-features and the group cardinality will be verified by the solver (the *presenter* in our architecture). The group is scattered all over the configuration GUI but it is still possible to select its sub-features while sticking to its cardinality.

The four attribute properties correspond to the four attribute types available in TVL. Their purpose is to determine the graphical widget of the corresponding type. The *intAttribute* and *realAttribute* properties represent integer and real attributes. They have the same set of values, namely *textbox* (a box containing the value) or *slider*. The rendering of Boolean attributes is influenced by the *boolAttribute* property. It can take three values, namely *checkbox*, *listbox*, or *radioGroup*. Note that this set of values is the same as optional features given the Boolean type of both constructs. The last attribute type available in TVL is enumeration. Its corresponding global property is named *enumAttribute* and can take *listbox* or *radiogroup* as values.

Finally, it is also possible to influence the rendering of views defined in the TVDL model with the *view* property. As introduced in previous section, available values are *tab* and *window*. The *tab* value means that all views will be represented by tabs in the same window. With the other value, *window*, each view will be rendered in its own window. In the latter case, navigation links between windows should be made available in each window.

Properties defined inside this global part can be seen as “default” values which can be overridden by other ones defined at a lower (i.e., more specific) level. As a case in point, properties defined at the view level have priority over global ones. Conversely, if a global property is not refined for a given construct, it will be used as default behaviour to generate the corresponding widget in the configuration GUI.

## View Properties

View-specific definition sections start with the dollar sign followed by the TVDL ID of the corresponding view. The different properties can then be defined inside the block delimited by curly braces. As for global properties, view-specific ones end with a semicolon. We can classify view-specific properties into two categories: those which apply to the view itself and those which apply to elements covered by the view.

We propose four properties which directly relate to the view referenced in the view-specific definition section (i.e., the TVDL view ID directly following the dollar sign). Using the Boolean *generate* property, one can define whether or not a view has to be rendered in the configuration GUI. This might, for example, be useful if the user has defined a view which is relevant in some contexts (technical, commercial, etc.) but should not be displayed in the GUI. It means that the TVDL model can contain views which are irrelevant for GUI generation. We also propose to define labels and help texts for views. Those properties are named *label* and *help*, respectively. They both take a double quoted string as value. The *label* property makes it possible to not use the view ID which might be too technical for the end-user. The help text might help the user understand the meaning or the purpose of a view. It is designer’s responsibility to choose the right words to help configuration GUI users in their task. Finally, we propose the *unavailable* property which determines what to do with the view contents when the view is not available. Values for this property are *hidden*, *greyed* and *none*, and their meaning is the same as for the *unavailableContent* global property.

The other category of view-specific properties is similar to the global properties. Indeed, properties falling in this category will influence the rendering of constructs covered by the view. For this reason, the proposed properties are exactly the same as global ones presented earlier. The 14 properties will not be recalled here for the sake of conciseness. However, we would like to draw the attention to one of them, *view*. As a reminder, this property allows to define the widget corresponding to views. Setting this property will have an influence on the views contained in the view corresponding to the view-specific definition section, not on that view itself. The



*view* property thus only makes sense for *grouping views*. In our opinion, all views declared at the same level should be depicted by the same widget. This explains why we did not propose a *widget* property in the first category. However, if needed, this property could be easily added.

## Feature Properties

The goal of this third category is to set properties for a given feature. Contrarily to the two previous categories, this one covers a single element which is a TVL feature. A feature-specific definition section starts with the ID of a feature in the referenced TVL model. It is the single category which has no starting symbol (like the dot character for global parts, or the dollar sign for views). Its contents are then delimited by curly braces. Seven different feature-specific properties are available.

Among the seven feature-specific properties, three are shared with view-specific ones, namely *label*, *help*, and *unavailable*. Available values and semantics are similar. For this reason, they will not be detailed here.

Four properties that are really specific to TVL features are given. *widget* is the first one and allows to set the widget for the feature in the rendered configuration GUI. It is the feature-specific counter-part of the *feature* global and view-specific properties. The same two values are available at the moment, *text* and *images*. Similarly, the *opt* feature-specific property has the same role as *optFeature* discussed earlier. As a reminder, available values are *checkbox*, *listbox*, and *radiogroup*. The role of this property is to determine the widget depicting the optionality of the feature in the GUI. This property only makes sense for optional features. The *select* property is equivalent to *featureSelect* and takes the same three values, *checkbox*, *listbox*, and *radiogroup*. Its role is to set the selection widget for features whose group is not covered by TVDL views. It should thus only be defined for features falling in this category.

The last feature-specific property, *group*, is a little more complex and has a different syntax. It can contain other properties. In this sense, a parallel can be drawn with TVL *struct* attributes. Its contents, replacing its value, are delimited by curly braces. There, six properties can be defined. Three of them are the common ones, *label*, *help*, and *unavailable*. Our experience with existing Web configurators and discussions with industrial partners showed that, in some cases, it should also be possible to define this information for groups. The *widget* property defines the widget for the group. Available values are *textbox*, *listbox*, *checkbox*, and *radiogroup*. They will depend on the decomposition type, *textbox* only for *and*-decompositions, *listbox* and *checkbox* for *or*-decompositions, *listbox* and *radiogroup* for *xor*-decompositions, and *checkbox* for *card*-decompositions. The Boolean *container* property has the same role as the *groupContainer* global property, that is determine whether the group and its sub-features have to be graphically enclosed together, for example using a box. Finally, the *default* property defines which group's sub-feature will be selected in the configuration GUI. Available values will be the group's sub-features. Ideally, default values should be defined in another language which is out of the scope of this thesis. For this reason, it is temporarily included in FCSS.

## Attribute Properties

The last category of properties, attribute-specific ones, is the simplest one. This is due to the nature of attributes which are the simplest TVL constructs. An attribute-specific definition section starts with the # symbol directly followed by the TVL ID of an attribute. The properties are then declared inside a block delimited, like other categories, by curly braces.

The *label*, *help*, and *unavailable* properties are the same as the ones previously discussed. A single property really specific to TVL attributes exists. It is called *widget* and can take *textbox*, *listbox*, *checkbox*, *radiogroup*, and *slider* as value. As for group widgets, values will depend on the attribute type. *textbox* and *slider* for *int* and *real* TVL attributes, *checkbox*, *radiogroup*, and *listbox* for *bool* attributes, and *listbox* and *radiogroup* for enumerations.

### 1.4.5.2 Addressing Plan Example

In our addressing plan example, the FCSS model contains only labels for views (e.g. line 6), features (e.g. line 46) and attributes (e.g. line 49). Due to space constraints, only the beginning of the FCSS model is visible in Listing 1.3. All other entries are similar to those depicted in the code excerpt.

**Listing 1.3** FCSS model for the addressing plan example

```

1  import "addressing_plan_demo.tvl"
2  import "addressing_plan_demo.tvd1"
3
4  // Views
5
6  $MainTab {
7      label: "main";
8  }
9
10 $SubnetTab {
11     label: "Subnets and Groups";
12 }
13
14 $InterfaceTab {
15     label: "Interfaces";
16 }
17
18 $RoutingTableTab {
19     label: "Routing Tables";
20 }
21
22 // Features and attributes
23
24 AddressPlan {
25     label: "Address plan properties";
26 }

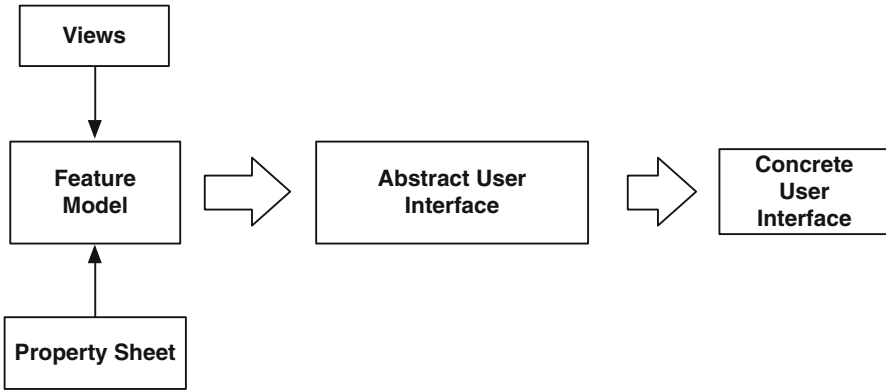
```

```
27 #AddressPlan.networkPrefix {
28     label: "Numero client";
29 }
30 #AddressPlan.strategy {
31     label: "Strategy";
32 }
33 #AddressPlan.useTypes {
34     label: "Number of use types";
35 }
36 #AddressPlan.futureUseTypes {
37     label: "Number of future use types for expansion";
38 }
39 #AddressPlan.locations {
40     label: "Number of locations";
41 }
42 #AddressPlan.futureLocations {
43     label: "Number of future locations for expansion";
44 }
45
46 Subnet {
47     label: "Subnet";
48 }
49 #Subnet.useType {
50     label: "Use type";
51 }
52 #Subnet.location {
53     label: "Location";
54 }
55 #Subnet.subnetName {
56     label: "Name";
57 }
58 #Subnet.subnetPrefix {
59     label: "Prefix";
60 }
```

## 1.4.6 Putting It All Together

### 1.4.6.1 General Principle

After having made the role of each model of our approach explicit, we explain here how they fit together. Our vision is based on the decoupling of the FM and the configuration GUI by combining separation of concerns [83] and generative techniques [79]. The base process is sketched in Fig. 1.4 and relies on the notion of AUI [22]. According to the W3C [89], an AUI is “*an expression of a UI in terms of interaction units without making any reference to implementation neither in terms of interaction modalities nor in terms of technological space (e.g., computing platform, programming or markup language)*”. In other words, an AUI is a language- and target platform-independent description of the UI, which allows



**Fig. 1.4** Interface generation process

considering mappings from the feature model in a unique and reusable manner. This AUI can be directly generated from the FM with the possibility to use *Views* to tweak configuration interface decomposition. The layout of the elements composing the UI can be guided by a *Property sheet* containing beautification information. Once created, the AUI can then be transformed into a CUI. Depending on the required sophistication level of the interface, different combinations of views and property sheets can be envisioned.

Based on the FM (TVL) and the associated *Property sheet* (FCSS), an AUI can be defined for the configurator. AUI languages describe UIs in terms of *Abstract Interaction Objects* (AIOs). Those AIOs present the advantage of being independent of any platform and any modality of interaction (graphical, vocal, virtual reality and so on). In this way, we keep our approach as generic as possible. This AUI will finally be translated into a CUI which is the implementation of the UI in a given language for a specific platform. Views can also intervene in this generation process (using TVDL). Once they have been defined, views-related beautifying information similar to FM-related one can be defined in the *Property sheet*. It is meant to beautify the UI with views-related information like their display name, help text, colours and styles.

#### 1.4.6.2 Addressing Plan Example

Our original intent was to generate configuration GUIs encoded in a given UIDL. They could then be transformed into multiple target implementations (e.g., HTML, GWT, etc.). As mentioned in Sect. 1.2.2, UIDL support is still immature or proprietary. As a reminder, we can mention that existing UIDLs either do not fit our requirements or tool support for transforming models into final GUIs are not available to us. This last point is really important to evaluate the quality of the generated configurators. Indeed, it is easier to show a final GUI than a model describing it to an end-user.



Fig. 1.5 Generation process with *Acceleo*

The screenshot shows a web-based configuration interface for IPv6 addressing plans. At the top, there are four tabs: 'Address Plan', 'Subnets and Groups', 'Interfaces', and 'Routing Tables'. The 'Address Plan' tab is selected. Below the tabs is a section titled 'Address plan properties' with a teal header. The properties are as follows:

Network prefix	<input type="text" value="2000:de4:abe7::/48"/>
strategy	<input type="text" value="LocationFirst"/>
Number of use types	<input type="text" value="3"/>
Number of future use types for expansion	<input type="text" value="2"/>
Number of locations	<input type="text" value="5"/>
Number of future locations for expansion	<input type="text" value="2"/>

Fig. 1.6 *Address plan* tab of the *HTML* configurator for IPv6 addressing plans

We thus had to skip the UIDL model in our MDE transformation chain to prefer a direct generation approach. For the target interface technology, we chose the *HTML5* language [88], the latest version of the *HTML* standard. As previously mentioned, a lot of configuration interfaces are Web-based, as illustrated by Cyledge’s configurators database [27]. By choosing *HTML*, we thus cover a lot of configurators. For other target languages, we depend on the availability of UIDLs, especially UsiXML which is in the standardization process [84]. In addition to the *HTML* target language for the static part of configuration GUIs, the presenter is developed in JavaScript, its natural complement.

No detail will be provided about the generator which is based on a model-to-text approach. Interested reader can refer to [18]. Basically, our implementation of model transformations takes the three models (TVL, TVDL, and FCSS) as input (see Fig. 1.5) and generates an *HTML* document.

The *HTML* page generated by our *Acceleo* tool is depicted in Figs. 1.6, 1.7, 1.8, and 1.9. Each figure represents the same *HTML* file with a different tab selected. The content of each page is automatically rendered by our generator.

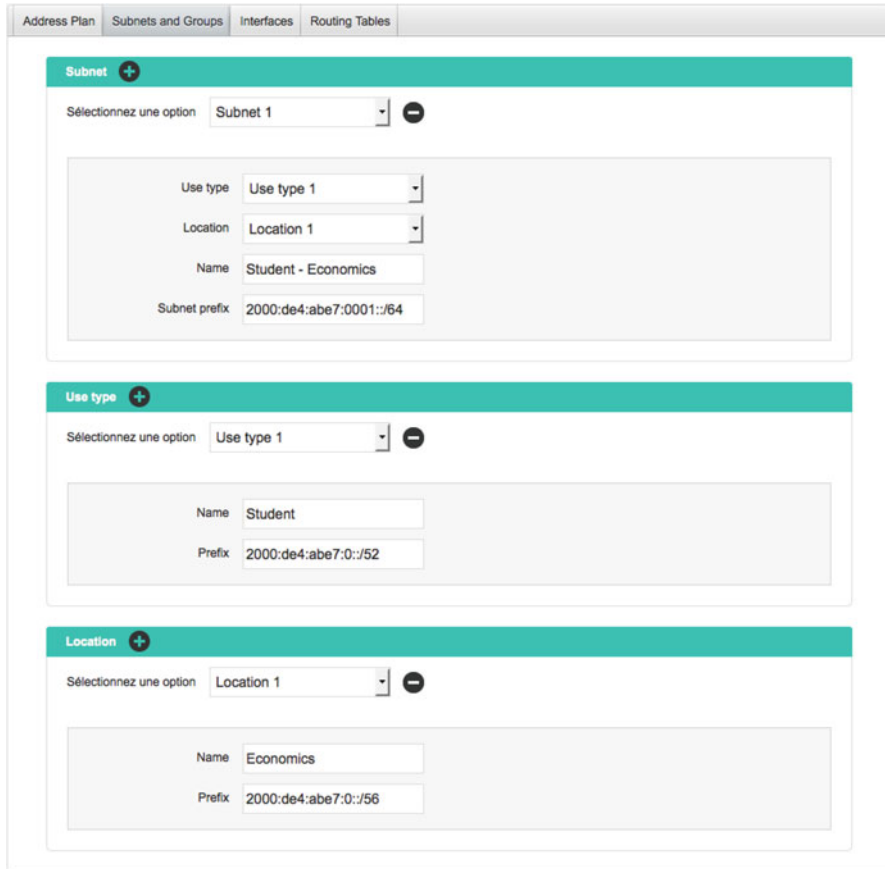


Fig. 1.7 Subnets and groups tab of the HTML configurator for IPv6 addressing plans

The first view (see Fig. 1.6) presents the user with the general properties of the addressing plan. It allows her to specify the number of locations and use types that are present in the organization, as well as the number of locations and use types that may potentially arise in the future. It also allows the use to select which strategy should be applied for the identification of the subnets that form the network.

The second view (see Fig. 1.7) allows the user to configure subnets and groups. She can instantiate use types, locations and associate them to subnets. The view offers to specify the IP prefixes that will identify the subnets and their hosts. In the example shown in Fig. 1.7, which follows the creation of an addressing plan for a university campus, all students from the faculty of economics will be grouped in the subnet identified by the prefix `2000:de4:abe7:0001::/64`.

The third view (see Fig. 1.8) enables the configuration of hosts and their interfaces. This is the view where the user can manage information related to hosts on the network and where she can associate hosts to their subnets. The panel

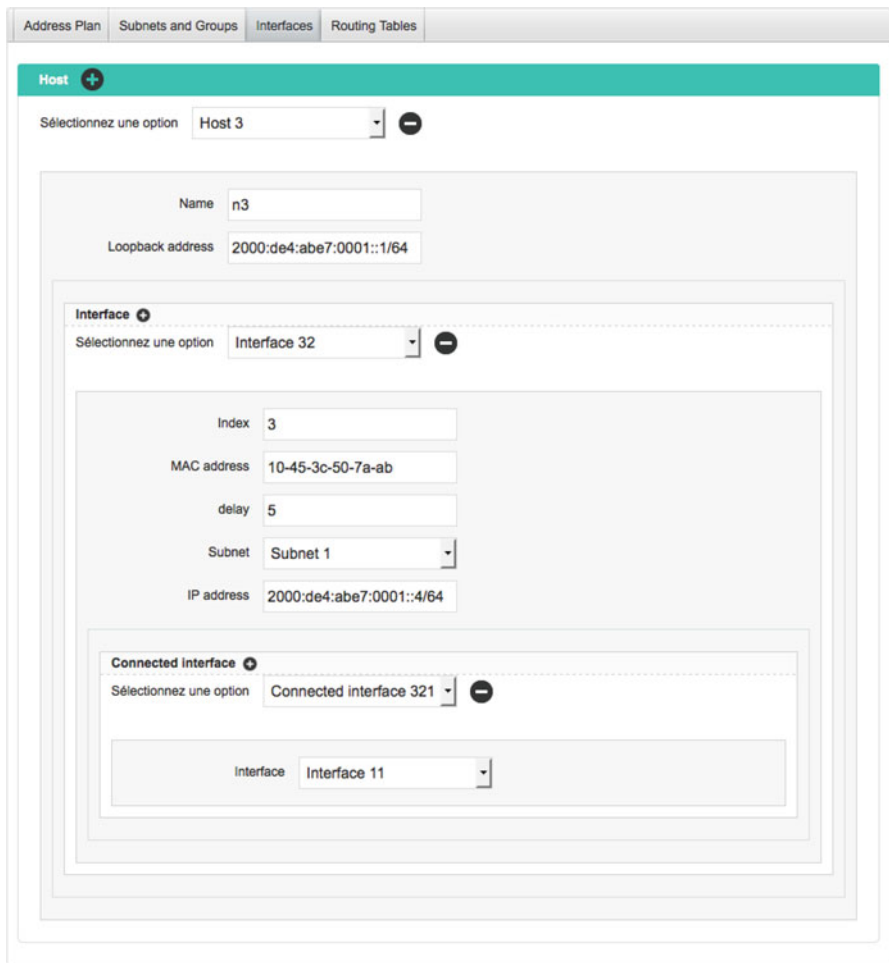


Fig. 1.8 Interfaces tab of the HTML configurator for IPv6 addressing plans

labelled “*Connected interface*” allows the user to configure the direct connections between interfaces that belong to distinct hosts. Figure 1.8 also offers an example of three nested features rendered into the panel labelled Host, Interface and Connected interface.

Finally, the fourth view (see Fig. 1.9) addresses the configuration of routing tables. In our example, the panel labelled Routing table entry shows a line in the routing table that indicates how packets directed to hosts identified by the prefix 2000:de4:abe7:2::/56 should be routed.

The screenshot displays the 'Routing Tables' tab in the HTML configurator. At the top, there are navigation tabs: 'Address Plan', 'Subnets and Groups', 'Interfaces', and 'Routing Tables'. The main content area is titled 'Host' and contains a dropdown menu labeled 'Sélectionnez une option' with 'Host 3' selected. Below this, there are two main sections:

- Host Configuration:**
  - Name: n3
  - Loopback address: 2000:de4:abe7:0001::1/64
- Routing table entry:**
  - Sélectionnez une option: Routing table entry 31
  - Destination: 2000:de4:abe7:2::/56
  - Metric: 2
  - Interface: Interface 32
  - Next hop: Interface 11

Fig. 1.9 Routing table tab of the HTML configurator for IPv6 addressing plans

## 1.5 Lessons Learned

We applied our approach on several research (such as the IPv6 addressing plan) as well as industrial cases.<sup>3</sup> Globally, our interlocutors were pleased with the generated HTML interfaces even if none of them used the full power of the FCSS model. We could thus conclude that the default behaviour of our generator matches the expectations of our first partners. The ease and speed with which interfaces could be generated allowed us to easily interact with people without variability modelling background. The different models changed a lot over time and all required changes were supported by the proposed languages. Some even challenged us and were not able to find weak points for TVDL.

However, our partners missed three things in the generated configuration Web page. First, they would like an additional “summary” tab. Finalisation being case-specific, we decided to not handle it in our generator. Instead, it should be developed based on user requirements. A possible implementation would be a Web service which, for a given configuration, returns the expected summary.

<sup>3</sup>Unfortunately, these cases could not be reported here for confidentiality reasons.



A much finer-grained handling of feature instances was also required by one of our interlocutors. In the interfaces currently generated, the number of clones is handled by a number input. Decreasing (resp. increasing) the number of feature instances will delete (resp. add) the *HTML* code corresponding to those instances, starting from the last. It is thus not possible to delete a given instance. This functionality can easily be added to our generator. Ideally, a button to create a new instance should also be added after the current last one.

One of our partners required to be able to define features having several parents. Theoretically, this request is supported by TVL through the *shared* feature construct. Those constructs are also supported by our *Acceleo* generator. However, we did not use them given that the current version of the solver does not support such features. This specific case study allowed us to get accurate requirements for shared constructs. The generator should be modified accordingly.

We now report our findings about the approach, including the solver, the TVL, TVDL and FCSS languages, the presenter or the generated configurator based on our collaborations.

**Completeness of TVL.** In the biggest TVL model we had to produce so far, we count four duplicable features. The same comment applies to string attributes added in TVL<sub>2</sub> and used ten times in the same case study, that is 17,9% of the attributes. Generally speaking, TVL offered the required expressiveness. Shared features also proved relevant, even if they are currently not supported by the underlying solver.

**Completeness of TVDL.** The view definition language has been assessed. It turned out that it supports all views required by our partners with one exception. To deal with this weakness, an abstract feature was added right under the root feature. In the future, TVDL should be extended in order to avoid such collateral effects on other models.

**Completeness of FCSS.** We did not use a lot of FCSS properties and focused mainly on labels. On the one hand, it does not allow us to thoroughly evaluate the language. On the other hand, it implies that the default behaviour corresponds to actual user needs. There is room for improvement. First, it should be possible to define the position of a label, before or after the TVL construct with which it is associated. Second, several FCSS properties should be made available for more fine grained TVL constructs. For example, it is not possible to define a label for the values of an enum attribute. The same comment applies to sub-attributes of structures. For such attributes, it is even not possible to change the widget, which is somewhat restricting. Defining the step for number attributes, the break point between radio groups and list boxes, etc. worth exploring according to our interlocutors. Finally, colours could also be defined for elements to be rendered in the GUI. An interesting feature would be to generate the same interface in several languages with different FCSS models. For this purpose, we could use the *include* mechanism of TVL in FCSS.

**Communication with the solver.** The JavaScript presenter fulfils its role of interface between the *HTML* page and the solver perfectly and behaves as expected.

Behind the scene, this component is probably the most complex one and should be simplified. At the moment, it handles some behaviours which should be on solver side. Migrating them would make the JavaScript much simpler and respect the separation of concerns. For example, the presenter currently handles transactions. Changing the value of a select box representing a *xor*-decomposition is an example of such a transaction. It can be decomposed into two tasks: (1) unassign the previously selected value and (2) assign the new one to *true*. After the first step, the solver randomly selects an option to comply with the group cardinality and returns it to the presenter. That value is ignored by the presenter as it knows that, in the second step, another value will be sent to the solver. In the future, the solver should handle requests containing multiple changes. The solver might be in an invalid state while the transaction is processed. At the end of it, the solver should be in a valid state. Otherwise, it means that the transaction is an invalid one.

**Role of generated GUIs.** In our different use cases, the generated interfaces provided valuable input to initiate discussion. Working only at TVL level seemed abstract for most of our interlocutors. *HTML* interfaces generated in less than one minute made the process more interactive. *TVDL* views were even tailored according to the audience. Indeed, high level managers do not have the same concerns as technicians. As expected, none of our interlocutors envisions to reuse the generated configuration GUI as-is in their final products. There are several reasons for this, including the graphical charter, legacy tools, etc. These reasons motivated us to focus on the correctness of the interface (with respect to configuration) and its structure (tabs, views) and to not aim for 100% automation neither possible nor desirable.

**Propagation strategies.** In the current solution, there are two possible outcomes to user changes. Either it is not valid and the previous state is reset, or it is acceptable and propagations are automatically applied in the GUI. While, in the first case, the implemented behaviour seems the single viable one, several strategies should be made available for valid changes. At the moment, the user is not informed of the consequences of her choices which are automatically propagated in the interface. Providing an explanation mechanism could minimize user's lack of comprehension concerning a propagated value. Such information requires modifications at the solver level. Alternatively, the set of propagations could be displayed to the user before applying them in the configuration GUI. If she confirms her choice, the configuration is updated according to the values in the set. Otherwise, the previous GUI state is reset, i.e., like for invalid changes. The two behaviours can co-exist.

**Source of propagations.** Initially, the presenter was able to handle values propagated by the solver in a specific way. In the prototype version, they were greyed out in order to prevent user changes. But this approach was rather restrictive with respect to the results sent back by the solver. For example, if a feature is selected, the propagation set contains its parent which will be greyed out in the configuration GUI. While this behaviour respects the semantics of FMs, it is not adapted to GUIs. In such a case, the user would have to deselect all sub-features to unblock the parent

one. Instead, it should be possible to set the parent to false with the unassignment of child features as side effect. We identify three categories of propagation sources: cross-tree, hierarchy, and siblings constraints. The first category should trigger the disabling mechanism (e.g., grey out). The second one has been illustrated by the example earlier in this paragraph. Finally, siblings constraints should be handled differently by the presenter depending on the widget representing the group. For example, *xor*-decompositions rendered as a list box or a radio group are automatically handled by the widget, contrarily to those depicted by a set of check boxes. In the future, the solver should return three propagation sets, differently handled by the presenter.

**Display strategies.** A top-down strategy is applied in our generator. By this, we mean that the contents of a feature are displayed in the configuration GUI as soon as it is selected. The Web page is thus populated as the user makes choices. However, some configurators might require a different display strategy. Theoretically, our approach can support other strategies with mechanisms such as the *unavailable* property in the FCSS model. We will require other case studies to evaluate the alternative behaviours.

## 1.6 Perspectives

### 1.6.1 *Multiple Targets*

We envision two solutions to target multiple output languages. The critical point is to have an UIDL suited to our configuration needs. The first solution consists in selecting a relevant subset of an UIDL like UsiXML to meet our requirements while still taking advantage of existing code generators. The second approach is to define our own UIDL dedicated to configuration GUIs. In that case, UI concepts would be strongly connected to FM concepts.

### 1.6.2 *Ordering Views*

In the GUI generation approach, the different views are rendered in the GUI in the same order as in the TVDL model. These views are all accessible to the user at any time. Such a behaviour is not suited to all situations. In the future, generated configurators should support explicit view ordering and activation/deactivation.

To describe those behaviours, feature configuration workflows [47] or multi-step SPL configuration [91] could be used. There, the workflow defines the configuration process and each view on the FM is assigned to a task in the workflow. A view is configured when the corresponding workflow task is executed. A feature configuration workflow is thus a combination of views on the FM, workflow and

the mapping between them. Up to now, feature configuration workflows focused on distributed configuration among several stakeholders but one might easily adapt them to other purposes like the dynamic behaviour of a GUI in our case.

After having defined views, the workflow representing the dynamic aspect of the GUI thus has to be modelled and its tasks attached to the different views to create a so-called FCW. FCW-related beautification information can also be stored in the FCSS along with information related to the FM and views.

### ***1.6.3 Re-engineering***

In our previous empirical study of 111 Web configurators, we were able to identify several common bad practices among online configurators, such as incomplete reasoning over configuration constraints, counter-intuitive representation of options or the loss of all the user's decisions when navigating backwards. The study reveals that developing an online configurator like any other typical Web application (i.e. without specific, adapted, and rigorous engineering methods) can lead to issues in reliability, runtime efficiency, and maintainability. These issues could be addressed though the migration from a legacy ad-hoc configurator to a better model-driven engineered configurator.

In this chapter, we focus on the creation of new configurators through the elaboration of feature models and the generation of configuration GUIs. We believe that our approach can also be useful in the context of re-engineering existing configurators. The re-engineering process consists of two steps.

1. The configuration models of the existing configurator are recovered by applying reverse-engineering techniques.
2. A new configurator is created from the recovered models.

The approach described in this chapter can be used to implement the second step of this re-engineering process as it can support the engineering of configuration GUIs from recovered feature models. As for the first step, Abassi et al. [1] propose a supervised semi-automatic process for reverse-engineering TVL code from web-based configurators. The user starts with the definition of variability data extraction patterns (vde patterns) which specify the variability information to be extracted from a given Web page written in an HTML-like language. A Web Wrapper is then used to extract the variability data from a given page, based on a given a set of vde patterns, and save it into an XML format. Some manual user configurations are also simulated in order to extract dynamic content such as configuration constraints. The information extracted through steps 2 and 3 can be edited and transformed into a feature model. Typically, several FMs are extracted from one Web configurator, e.g., one FM for each configuration step. These FMs are merged by FAMILIAR, a tool-supported language to merge multiple FMs into a single one [5].

Figure 1.10 presents the full re-engineering process, the contribution of this chapter being represented on the right side by the forward-engineering steps. The

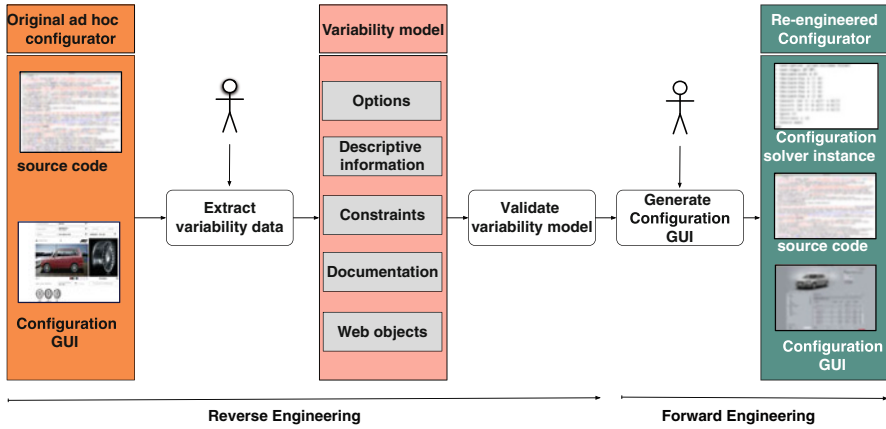


Fig. 1.10 Re-engineering process for configurators

interested reader can refer to [19] for more detailed discussion of the re-engineering process.

### 1.6.4 Product Selection

The configuration model of a configurator is a concise representation of the technical and functional properties of all variants for a product line. Typically, the configurator is the system through which the particular requirements of the current user are collected, and which exploits the configuration model to derive the product with the properties meeting these requirements. This means that the system gradually refines the features and attributes that will be included in the single final product at each user interaction.

For some product lines, while the configuration process still consists in collecting requirements from the user and verifying their consistency, the resulting configuration is used to compute a set of candidate products which all meet the user's expectations. This can be the case when a customer needs to select a candidate product from a catalogue, but is first asked to complete a configuration task in order to define the environment in which it will be deployed. In this scenario, the purpose of this first configuration task is to filter out the invalid products, that is those which do not hold the properties that would make them suitable for the configured environment. For such product lines, the process of choosing the final product can thus be divided into two phases. Firstly, a configuration phase that determines the valid products. Secondly, a selection phase through which the user selects one final product among all valid products. Figure 1.11 illustrates the two phases for a catalogue of servers.

In the previous sections, we propose a generative approach for supporting configuration tasks. An interesting research direction would be the extension of our



**Fig. 1.11** The configuration phase and the selection phase for a catalogue of servers

work to the engineering of selection phases in order to help users rank competing valid products and evaluate trade-offs. In the remainder of this section, we discuss *product comparators* and *knowledge-based recommender systems*, two types of systems which could benefit from a model-driven development approach.

#### 1.6.4.1 Product Comparators

Product comparators aim at assisting customers during the evaluation of product assortments. These systems help their users to visualize the similarities and differences between competing products within product comparison matrices (PCM). A PCM offers a tabular representation of the characteristics of competing products that helps customers to rapidly compare them and evaluate trade-offs between them.

While the structure of PCMs may appear simple, they can contain heterogeneous data and be frequently updated as new products and features emerge. For these reasons, practitioners can benefit from a model-driven approach for maintaining PCMs.

The interested reader can refer to Bécan et al. [10]. The authors propose a meta-model for PCMs and discuss model-based techniques as well as automated tools for developing PCMs.

#### 1.6.4.2 Knowledge-Based Recommender Systems

Like product comparators, a recommender system aims at helping customers to navigate competing product ranges. Knowledge-based recommender systems (KBRS) are a particular type of recommender systems that share common characteristics with configurators. Indeed, they also collect requirements from their users and exploit knowledge about the products to provide purchase recommendations (see Felfernig et al. [36] for a more detailed coverage of KBRS). Similarly to configurators and their configuration models, KBRS operate a knowledge base which synthesizes the knowledge about the product properties and their relationships with customer requirements.

Oftentimes the development of KBRS gives rise to a domain knowledge acquisition bottleneck, a challenge also encountered by developers of configurators. This

problem refers to the need for practitioners to encode knowledge about the products into the formalism used within the knowledge base of the system. This acquisition phase is critical as the resulting knowledge base will determine the behaviour of the system. The term *bottleneck* refers to the fact that this phase often proves to be both time-consuming and error-prone. It thus requires particular effort and cautiousness from practitioners when searching product documentation or engaging with domain experts in order to ensure the completeness, accuracy and consistency of the knowledge base.

This challenge has led to previous research effort. Felfernig et al. [35] discuss an environment for engineering KBRS and their user interfaces. The motivation for a model-driven approach is to accelerate the acquisition of domain knowledge through fast prototyping, and to reduce maintenance costs.

### 1.6.5 Recommendations

Web configurators often have to cope with domain knowledge related to complex products (i.e., products with numerous features, attributes, as well as business and technical constraints to satisfy). Due to this complexity, users can be exposed to an overwhelming number of configuration steps to resolve, to the extent that the benefits of the co-creation process risks to be offset. This has serious managerial implications as tedious co-creation processes can make vendors undesirable for customers [67]. Franke and Schreier [38] show that the enjoyment and perceived effort of the co-design process have a direct impact on the willingness to pay for customized products. Configuration complexity can also make customers miss the product that best meets their expectations as they shift towards simplifying decision heuristics [31].

For these reasons, it is important to assist users of Web configurators during their configuration tasks. Previous works have addressed the development of recommendation techniques to help users resolve configuration steps, that is recommendations for feature selections and attribute values. Researchers have proposed the use of *defaults values* which denote predefined recommendations that are applied based on the current user preferences [34, 59]. Other approaches consist in analyzing past configurations to infer recommendations [34, 90]. Felfernig et al. [37] analyze the current partial configuration of the user and use a similarity-based approach to recommend the complete configurations that are the closest to the already specified user requirements.

In this chapter we discussed languages to model variability and build GUI elements. It would be interesting to investigate language extensions or additions to support practitioners during the elicitation of configuration recommendations and the generation of corresponding GUI elements through a model-based approach.

### ***1.6.6 Evolution of Configuration Interfaces***

In addition, to the generation of configuration interfaces, another concern is their evaluation. As noted by Leclercq et al. [57] there is limited knowledge on what are the general guidelines and principles guiding the design of configuration interfaces. Indeed, most of the works criticise existing interfaces or practices (e.g., [2, 76]), focus on specific configuration interfaces (Web) or business-to-consumer (B2C) applications. As our case-study suggests, not all configuration interfaces are dedicated to a general audience, and the specific needs and skills of intended users have to be taken into account when designing interfaces for them. We are therefore in search for grounded theories and guidelines that could assist the design of such interfaces and in the long term incorporate these principles in our generative approaches.

## **1.7 Conclusion**

The explosion of e-commerce applications and the need for customized products tailoring user needs make the development of configurators a concern in a variety of domains. Configurator engineering is a difficult activity: configurators both need to be consistent while handling user's decisions and their graphical user interfaces should meet usability and aesthetics requirements of consumers. This difficulty is often amplified in ad-hoc configurators in which the variability model, graphical user interface concerns and reasoning engine are all implicit and/or entangled. The software product line community has developed conceptual models and concrete tools to perform configuration through (simple) feature models. However, the engineering of configuration graphical interfaces has been much less addressed.

In this chapter, we present a model-based perspective. We rely on (advanced) feature models to formally specify configuration options and automate reasoning. We developed a model-based solution to generate graphical user interfaces from feature models while relying on SAT/SMT solvers to perform reasoning to react to user selections/deselections. We propose a model-view-presenter architecture to separate variability, reasoning and presentation. In our approach, the model is a feature model and its solver, and the view is a graphical user interface. The presenter will depend on the target graphical user interface technology. Its main role is to enable communication between the model and the view.

As existing feature modelling languages are not providing the expressiveness required to cover our needs, we developed a new language: it is a textual language named TVL and supports constructs such as feature attributes or group cardinalities which are not supported by most existing variability modelling languages. Furthermore, the language provides two mechanisms for structuring large models: an include statement to split the model into several files and the possibility to define a feature in one place and extend it later in the code. These mechanisms allow modellers to organise the feature model according to their preferences and can be used to implement separation of concerns.



In order to split the hierarchy of feature models, we propose a view definition language called TVDL. It is inspired by the XPath language previously used by Hubaux et al. in the context of feature configuration workflows. The advantage of TVDL is that it is not XML-based and allows to select any (combination of) TVL model construct(s). Four kinds of views are supported: grouping, sub-tree, feature and attribute. Grouping views are syntactic sugar to group the three other kinds of views. Sub-tree views allow to select TVL constructs in a sub-tree of a TVL model, feature views allow to select a feature and its contents (or a part of them), and attribute views cover TVL attributes (and their sub-attributes for structured ones).

As TVL and TVDL models do not focus on styling information, we propose FCSS. FCSS is a beautification language which contains information related to the graphical user interface such as labels or help texts, for example. The language has been named after CSS which plays a similar role for *HTML* Web pages. FCSS models can be decomposed into three levels. The highest one, called *global*, defines properties which should be applied to all constructs of imported TVL and TVDL models. They can be seen as default values. The second level defines the default properties for all constructs contained in a view. Finally, the last level allows to define properties for a specific feature or attribute.

Configuration interfaces are generated through model transformations, of which TVL, TVDL and FCSS models are the inputs. Our initial intent was to use a user interface description language as target, more specifically an abstract user interface model. In that case, model-to-model transformations would have been used. However, we did not find such a language meeting all our criteria. Consequently, our prototype generator produces *HTML* code through model-to-text transformations. The workload to move from a model-to-text to a model-to-model transformation should not be too high given that the most intricate part can be massively reused.

The languages and the generator were evaluated together on several cases. Our approach and the generator were used iteratively to demonstrate and evaluate the capabilities of the tool to (re)design and (re)generate a configurator on-the-fly. This could be done at such speed that the tools can be used during workshops in order to dynamically adapt the configurator based on the participants' input. Our experiences demonstrated the utility of the approach and allowed to identify various improvement opportunities.

**Acknowledgements** This work was partly supported by the European Commission (FEDER IDEES/CO-INNOVATION). Jean-Marc Davril is supported by the FNRS under a FRIA doctoral grant.

## References

1. Abbasi EK, Acher M, Heymans P, Cleve A. Reverse engineering web configurators. In: 2014 software evolution week-IEEE conference on software maintenance, reengineering and reverse engineering (CSMR-WCRE). IEEE; 2014. p. 264–273.

2. Abbasi EK, Hubaux A, Acher M, Boucher Q, Heymans P. The anatomy of a sales configurator: an empirical study of 111 cases. In: Salinesi C, Norrie MC, Pastor O, editors. Proceedings of the 25th international conference on advanced information systems engineering (CAiSE'13), vol 7908. Springer; 2013. p. 162–177.
3. Abele A, Papadopoulos Y, Servat D, Törnngren M, Weber M. The CVM framework – a prototype tool for compositional variability management. In: Proceedings of the 4th international workshop on variability modelling of software-intensive systems (VaMoS'10) [12]. p. 101–106.
4. Acher M, Collet P, Lahire P, France R. Separation of concerns in feature modeling: support and applications. In: Proceedings of the 11th annual international conference on aspect-oriented software development (AOSD'12). ACM; 2012, to appear.
5. Acher M, Collet P, Lahire P, France R. Familiar: a domain-specific language for large scale management of feature models. Science of Computer Programming (SCP) Special issue on programming languages; 2013. p. 55. doi:<http://dx.doi.org/10.1016/j.scico.2012.12.004>
6. Ali M, Pérez-Quiñones MA, Abrams M, Shell E. Building multi-platform user interfaces with UIML. In: Kolski C, Vanderdonck J, editors. Computer-aided design of user interfaces III. Netherlands: Springer; 2002. p. 255–266.
7. Antkiewicz M, Czarnecki K. FeaturePlugin: feature modeling plug-in for eclipse. In: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange. 2004.
8. Batory DS. Feature models, grammars, and propositional formulas. In: Proceedings of the 9th international conference on software product lines (SPLC'05). 2005. p. 7–20.
9. Batory D, Geraci BJ. Validating component compositions in software system generators. In: Proceedings 4th international conference on software reuse (ICSR'96). 1996. p. 72–81.
10. Bécan G, Sannier N, Acher M, Barais O, Blouin A, Baudry B. Automating the formalization of product comparison matrices. In: 29th IEEE/ACM international conference on automated software engineering (ASE'14). Västerås, Suède. 2014. doi:[10.1145/2642937.2643000](https://doi.org/10.1145/2642937.2643000). <http://hal.inria.fr/hal-01058440>.
11. Benavides D, Batory DS, Grünbacher P, editors. Proceedings of the 4th international workshop on variability modelling of software-intensive systems, Linz.ICB-research report, vol 37. Universität Duisburg-Essen. 2010.
12. Benavides D, Segura S, Ruiz-Cortés A. Automated analysis of feature models 20 years later: a literature review. Inf Syst. 2010;35(6):615–36.
13. Benavides D, Segura S, Trinidad P, Cortés AR. FAMA: tooling a framework for the automated analysis of feature models. In: Proceedings of the 1st international workshop on variability modelling of software-intensive systems (VaMoS'07), 2007. p. 129–134.
14. Beuche D. Modeling and building software product lines with pure: :variants. In: Proceedings of the 12th international software product line conference (SPLC'08). Washington, DC: IEEE Computer Society; 2008. p. 358.
15. Blouin A, Morin B, Nain G, Beaudoux O, Albers P, Jézéquel JM. Combining aspect-oriented modeling with property-based reasoning to improve user interface adaptation. In: Proceedings of the 3rd ACM SIGCHI symposium on engineering interactive computing systems (EICS'11). ACM; 2011. p. 85–94.
16. Blumendorf M, Lehmann G, Albayrak S. Bridging models and systems at runtime to build adaptive user interfaces. In: Proceedings of the 2nd ACM SIGCHI symposium on engineering interactive computing systems (EICS'10). ACM; 2010. p. 9–18.
17. Botterweck G, Janota M, Schneeweiss D. A design of a configurable feature model configurator. In: Proceedings of the 3rd international workshop on variability modelling of software-intensive systems (VaMoS'09). 2009. p. 165–68.
18. Boucher Q. Engineering configuration graphical user interfaces from variability models. Ph.D. thesis, University of Namur (2014).
19. Boucher Q, Abbasi EK, Hubaux A, Perrouin G, Acher M, Heymans P. Towards more reliable configurators: a re-engineering perspective. In: Proceedings of the 3rd product Line approaches in software engineering (PLEASE'12). 2012. p. 29–32.

20. Boucher Q, Classen A, Faber P, Heymans P. Introducing TVL, a text-based feature modelling language. In: Proceedings of the 4th international workshop on variability modelling of software-intensive systems (VaMoS'10). Universität Duisburg-Essen; 2010. p. 159–62.
21. Burbeck S. Applications programming in smalltalk-80: how to use model-view-controller (MVC). 1992. <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>
22. Calvary G, Coutaz J, Thevenin D, Limbourg Q, Bouillon L, Vanderdonckt J. A unifying reference framework for multi-target user interfaces. *Interact Comput.* 2003;15:289–308.
23. Chen L, Babar MA, Ali N. Variability management in software product lines: a systematic review. In: Proceedings of the 13th international software product line conference (SPLC'09). 2009. p. 81–90.
24. Classen A, Boucher Q, Heymans P. A text-based approach to feature modelling: syntax and semantics of TVL. *Sci Comput Program.* 2011;76:1130–1143.
25. Classen A, Heymans P, Schobbens PY. What's in a feature: a requirements engineering perspective. In: Proceedings of the 11th international conference on fundamental approaches to software engineering (FASE'08). 2008. p. 16–30.
26. Coutaz J. User interface plasticity: model driven engineering to the limit! In: Proceedings of the 2nd ACM SIGCHI symposium on engineering interactive computing systems (EICS'10). ACM; 2010. p. 1–8.
27. Cyledge. Cyledge configurator database. 2013. <http://www.configurator-database.com>. Last consulted: Aug 2013.
28. Czarnecki K. Variability modeling: state of the art and future directions (keynote). In: Proceedings of the 4th international workshop on variability modelling of software-intensive systems (VaMoS'10) [12]. p. 11.
29. Czarnecki K, Eisenecker UW. Generative programming: methods, tools, and applications. Boston: Addison-Wesley; 2000.
30. Czarnecki K, Helsen S, Eisenecker UW. Formalizing cardinality-based feature models and their specialization. *Softw Process Improv Pract.* 2005;10(1):7–29
31. Dellaert BG, Stremersch S. Marketing mass-customized products: striking a balance between utility and complexity. *J Market Res.* 2005;42(2):219–27.
32. van Deursen A, Klint P. Domain-specific language design requires feature descriptions. *J Comput Inf Technol* 2002;10(1):1–18
33. Eisenstein J, Vanderdonckt J, Puerta A. Applying model-based techniques to the development of UIs for mobile computers. In: Proceedings of the 6th international conference on intelligent user interfaces (IUI'01). New York: ACM; 2001. p. 69–76.
34. Falkner A, Felfernig A, Haag A. Recommendation technologies for configurable products. *AI Mag.* 2011;32(3):99–108.
35. Felfernig A, Friedrich G, Jannach D, Zanker M. An integrated environment for the development of knowledge-based recommender applications. *Int J Electron Commer.* 2006;11(2):11–34.
36. Felfernig A, Friedrich G, Jannach D, Zanker M. Constraint-based recommender systems. In: Ricci F, Rokach L, Shapira B, editor. *Recommender systems handbook*. New York: Springer; 2015. p. 161–90.
37. Felfernig A, Mandl M, Tiihonen J, Schubert M, Leitner G. Personalized user interfaces for product configuration. In: Proceedings of the 15th international conference on intelligent user interfaces. ACM; 2010. p. 317–20.
38. Franke N, Schreier M. Why customers value self-designed products: the importance of process effort and enjoyment\*. *J Prod Innov Manag.* 2010;27(7):1020–31.
39. Gabillon Y, Biri N, Otjacques B. Methodology to integrate multi-context UI variations into a feature model. In: Proceedings of the 17th international software product line conference co-located workshops (SPLC'13). ACM; 2013. p. 74–81.
40. Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. Boston: Addison-Wesley; 1995.
41. García JG, González-Calleros JM, Vanderdonckt J, Arteaga JM. A theoretical survey of user interface description languages: preliminary results. In: Proceedings of the 2009 Latin American web congress (La-web 2009). 2009. p. 36–43.

42. Gomaa M, Salah A, Rahman S. Towards a better model based user interface development environment: a comprehensive survey. In: Proceedings of the 38th Midwest instruction and computing symposium (MICS'05). 2005.
43. Grechanik M, Batory DS, Perry DE. Design of large-scale polylingual systems. In: Proceedings of the 26th international conference on software engineering (ICSE'04). 2004. p. 357–66.
44. Griss ML, Favaro J, Alessandro MD. Integrating feature modeling with the RSEB. In: Proceedings of the 5th international conference on software reuse (ICSR'98). 1998. p. 76–85.
45. Helms J, Schaefer R, Luyten K, Vermeulen J, Abrams M, Coyette A, Vanderdonckt J. Human-centered engineering of interactive systems with the user interface markup language. In: Seflah A, Vanderdonckt J, Desmarais MC, editors, Human-centered software engineering. London: Springer; 2009. p. 139–71.
46. Hubaux A, Boucher Q, Hartmann H, Michel R, Heymans P. Evaluating a textual feature modelling language: four industrial case studies. In: Proceedings of the 3rd international conference on software language engineering (SLE'10). 2010. p. 337–56.
47. Hubaux A, Classen A, Heymans P. Formal modelling of feature configuration workflows. In: Proceedings of the 13th international software product line conference (SPLC'09). 2009. p. 221–30.
48. Hubaux A, Classen A, Mendonca M, Heymans P. A preliminary review on the application of feature diagrams in practice. In: Proceedings of the 4th international workshop on variability modelling of software-intensive systems (VaMoS'10) [12]. p. 53–9. <http://www.vamos-workshop.net/2010>
49. Hubaux A, Heymans P, Schobbens PY, Deridder D, Abbasi EK. Supporting multiple perspectives in feature-based configuration. *Softw Syst Model*. 2011;12:1–23.
50. Hubaux A, Heymans P, Schobbens PY, Deridder D, Abbasi EK. Supporting multiple perspectives in feature-based configuration. *Softw Syst Model*. 2013;12(3):641–63.
51. Johnson PD, Parekh J. Multiple device markup language: a rule approach. Technical report, DePaul University. 2003.
52. Kang K, Cohen S, Hess J, Novak W, Peterson S. Feature-oriented domain analysis (FODA) feasibility study. Technical report, SEI, CMU. 1990.
53. Kang KC, Kim S, Lee J, Kim K, Kim GJ, Shin E. FORM: a feature-oriented reuse method with domain-specific reference architectures. *Ann Softw Eng*. 1998;5:143–68.
54. Kästner C, Thüm T, Saake G, Feigenspan J, Leich T, Wielgorz, F, Apel S. FeatureIDE: a tool framework for feature-oriented software development. In: Proceedings of the 31th international conference on software engineering (ICSE'09). 2009. p. 311–20.
55. Knuth DE. Semantics of context-free languages. *Math Syst Theory*. 1971;5(1):95–6.
56. Kost S. Dynamically generated multi-modal application interfaces. Ph.D. thesis, Dresden University of Technology. 2006.
57. Leclercq T, Davril JM, Cordy M, Heymans P. Beyond de-facto standards for designing human-computer interactions in configurators. In: Workshop on engineering computer-human interaction in recommender systems (EnCHIReS) co-located with EICS. 2016.
58. Limbourg Q, Vanderdonckt J, Michotte B, Bouillon L, López-Jaquero V. UsiXML: a language supporting multi-path development of user interfaces. In: Bastide R, Palanque P, Roth J, editors, Engineering human computer interaction and interactive systems. Lecture notes in computer science, vol 3425. Berlin/Heidelberg: Springer; 2005. p. 200–20.
59. McSherry D. Incremental nearest neighbour with default preferences. In: Proceedings of the 16th Irish conference on artificial intelligence and cognitive science. 2005. p. 9–18.
60. Mendonca M. Efficient reasoning techniques for large scale feature models. Ph.D. thesis, University of Waterloo. 2009.
61. Michel R, Classen A, Hubaux A, Boucher Q. A formal semantics for feature cardinalities in feature diagrams. In: Proceedings of the 5th workshop on variability modeling of software-intensive systems (VaMoS'11). 2011. p. 82–9.
62. Mueller W, Schaefer R, Bleul S. Interactive multimodal user interfaces for mobile devices. In: Proceedings of the 37th annual Hawaii international conference on system sciences (HICSS'04). Washington, DC: IEEE Computer Society; 2004. p. 90286.1–.

63. Müller A, Forbrig P, Cap CH. Model-based user interface design using markup concepts. In: Proceedings of the 8th international workshop on interactive systems: design, specification, and verification-revised papers (DSV-IS'01). London: Springer; 2001. p. 16–27.
64. Myers BA, Hudson SE, Pausch RF. Past, present, and future of user interface software tools. *ACM Trans Comput-Hum Interact*. 2000;7:3–28.
65. Paterno F, Santoro C, Spano LD. MARIA: a universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans Comput-Hum Interact*. 2009;16(4):19:1–19:30.
66. Picard E, Fierstone J, Pinna-Déry AM, Riveill M. Atelier de composition d'IHM et évaluation du modèle de composants. Tech. Rep. Livrable 3, Réseau National des Technologies Logicielles. 2003.
67. Piller FT, Blazek P. Core capabilities of sustainable mass customization. Burlington, Massachusetts: Morgan Kauffman; 2014.
68. Pleuss A, Botterweck G, Dhungana D. Integrating automated product derivation and individual user interface design. In: Proceedings of the 4th international workshop on variability modelling of software-intensive systems (VaMoS'10). 2010. p. 69–76.
69. Pleuss A, Rabiser R, Botterweck G. Visualization techniques for application in interactive product configuration. In: Proceedings of the 15th international software product line conference, (SPLC'11), vol 2. 2011. p. 22.
70. Pohl K, Böckle G, van der Linden FJ. Software product line engineering: foundations, principles and techniques. Berlin/Heidelberg: Springer; 2005.
71. Potel M. MVP: model-view-presenter the taligent programming model for c++ and Java. Cupertino, California: Taligent Inc. 1996.
72. Puerta A, Eisenstein J. XIML: a common representation for interaction data. In: Proceedings of the 7th international conference on intelligent user interfaces (IUI'02). New York: ACM; 2002. p. 214–15.
73. Puerta A, Eisenstein J. Developing a multiple user interface representation framework for industry. In: Multiple user interfaces: engineering and application framework. Wiley; 2003. p. 119–48.
74. Pure-systems GmbH. Variant management with pure::variants. 2006. Technical White Paper. <http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf>
75. Quinton C, Parra CA, Mosser S, Duchien L. Using multiple feature models to design applications for mobile phones. In: Proceedings of the 15th international software product line conference (SPLC'11), vol 2. 2011. p. 23.
76. Rabiser R, Grünbacher P, Lehofer M. A qualitative study on user guidance capabilities in product configuration tools. In: Goedicke M, Menzies T, Saeki M, editors. IEEE/ACM international conference on automated software engineering, ASE'12, Essen, 3–7 Sept 2012. ACM; 2012. p. 110–19. doi:10.1145/2351676.2351693, <http://doi.acm.org/10.1145/2351676.2351693>
77. Reenskaug T. Models-views-controllers. 1979. <http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf>
78. Reiser MO. Core concepts of the compositional variability management framework (cvm). Technical report, Technische Universität Berlin. 2009.
79. Schlee M, Vanderdonck J. Generative programming of GUIs. In: Proceedings of the 7th international working conference on advanced visual interfaces (AVI'04). ACM; 2004. p. 403–06.
80. Schobbens PY, Heymans P, Trigaux JC, Bontemps Y. Generic semantics of feature diagrams. *Comput Netw*. 2006;51(2):456–79.
81. Souchon N, Vanderdonck J. A review of XML-compliant user interface description languages. In: Jorge J, Jardim Nunes N, Falcão e Cunha J, editors. Interactive systems. Design, specification, and verification. Lecture notes in computer science, vol 2844. Berlin/Heidelberg: Springer; 2003. p. 377–91.
82. de Sousa LG, Leite JC. XICL: a language for the user's interfaces development and its components. In: Proceedings of the Latin American conference on human-computer interaction (CLIH'03). New York: ACM; 2003. p. 191–200.

83. Tarr P, Ossher H, Harrison W, Sutton SMJ. N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the 21st international conference on software engineering (ICSE'99). 1999. p. 107–19. <http://doi.ieeecomputersociety.org/10.1109/ICSE.1999.841000>
84. UsiXML Consortium. USer Interface eXtensible Markup Language (UsiXML). Submitted to the W3C Model-Based UI Working Group. 2012.
85. W3C: Cascading style sheets. 2008. <http://www.w3.org/TR/REC-CSS1/>. Last consulted: Oct 2013.
86. W3C. XForms 1.1. 2009. <http://www.w3.org/TR/2009/REC-xforms-20091020/>
87. W3C. Model-based UI XG final report. 2010. <http://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui-20100504/>
88. W3C. HTML5. 2014. <http://www.w3.org/TR/html5/>. Last consulted: Dec 2013.
89. W3C. MBUI – abstract user interface models. 2014. <http://www.w3.org/TR/abstract-ui/>
90. Wang Y, Tseng MM. Customized products recommendation based on probabilistic relevance model. *J Intell Manuf.* 2013;24(5):951–60.
91. White J, Galindo JA, Saxena T, Dougherty B, Benavides D, Schmidt DC. Evolving feature model configurations in software product lines. *J Syst Softw.* 2014;87(0):119–136. doi:<http://dx.doi.org/10.1016/j.jss.2013.10.010>. <http://www.sciencedirect.com/science/article/pii/S0164121213002434>

# Chapter 2

## User Interfaces and Dynamic Software Product Lines

Dean Kramer and Samia Oussena

**Abstract** In the modern world of mobile computing and ubiquitous technology, society can interact with technology in new and fascinating ways. To help provide an improved user experience, mobile software should be able to adapt itself to suit the user. By monitoring context information based on the environment and user, the application can better meet the dynamic requirements of the user. Similarly, it is noticeable that programs can require different static changes to suit static requirements. This program commonality and variability can benefit from the use of Software Product Line Engineering, reusing artefacts over a set of similar programs, called a Software Product Line (SPL). Historically, SPLs are limited to handling static compile time adaptations. Dynamic Software Product Lines (DSPL) however, allow for the program configuration to change at runtime, allow for compile time and runtime adaptation to be developed in a single unified approach. While currently DSPLs provide methods for dealing with program logic adaptations, variability in the Graphical User Interface (GUI) has largely been neglected. Due to this, depending on the intended time to apply GUI adaptation, different approaches are required. The main goal of this work is to extend a unified representation of variability to the GUI, whereby GUI adaptation can be applied at compile time and at runtime. In this chapter, we introduce an approach to handling GUI adaptation within DSPLs, which provides a unified representation of GUI variability.

### 2.1 Introduction

In the modern world of mobile computing and ubiquitous technology, society can interact with technology in new and fascinating ways. To help provide an improved user experience, mobile software should be able to adapt itself to suit the user. By monitoring context information based on the environment and user, the application can better meet the dynamic requirements of the user. Similarly, it is noticeable that programs can require different static changes to suit static requirements. This

---

D. Kramer (✉) • S. Oussena  
School of Computing and Technology, University of West London, London, UK  
e-mail: [d.kramer@mdx.ac.uk](mailto:d.kramer@mdx.ac.uk)

program commonality and variability can benefit from the use of Software Product Line Engineering, reusing artefacts over a set of similar programs, called a Software Product Line (SPL). Historically, SPLs are limited to handling static compile time adaptations. Dynamic Software Product Lines (DSPL) however, allow for the program configuration to change at runtime, allow for compile time and runtime adaptation to be developed in a single unified approach. While currently DSPLs provide methods for dealing with program logic adaptations, variability in the Graphical User Interface (GUI) has largely been neglected. Due to this, depending on the intended time to apply GUI adaptation, different approaches are required. The main goal of this work is to extend a unified representation of variability to the GUI, whereby GUI adaptation can be applied at compile time and at runtime.

In this chapter, we introduce an approach to handling GUI adaptation within DSPLs, which provides a unified representation of GUI variability. In Sect. 2.2, we motivate this chapter and the need for handling GUI adaptation in DSPLs. Section 2.3 discusses previous work in the fields. An introduction into the challenges of GUIs and DSPLs is discussed in Sect. 2.4. Our approach to handling GUI variability is described in Sect. 2.5. In Sect. 2.6, we describe how a DSPL is generated using our approach. Section 2.7 outlines our implementation. Next, we introduce examples using our approach, and discuss some of the current limitations in Sect. 2.8. Finally, Sect. 2.9 concludes this chapter.

## 2.2 Motivation

To help drive this chapter, let us consider a mobile content store application as a scenario application, like the Google Play store. These applications allow different content to be bought, and consumed by the user, including different applications & games, movies & TV shows, and music. Different methods of content consumption can often be handled. For example, music and movies can be streamed to the device, or downloaded.

These applications often require tailoring to suite different user requirements and situations. These changes can be static including tailored content stores designed for education, or user interfaces designed for particular user groups. Static tailoring remains constant during use, and only changes during explicit application updating on the device. Other changes can be more dynamic, based on the changing situations the user is in. For example, sales and distribution licensing is often required for selling different content in different countries and geographic regions. If a person takes their mobile device on a plane to another country, they might be legally unable to purchase a specific piece of content. For these reasons, this content should not be shown to the user, and can require changes in the user interface when whole types of content are not available, for example music. Other dynamic changes can be related to network connectivity and unused storage capacity of the device.



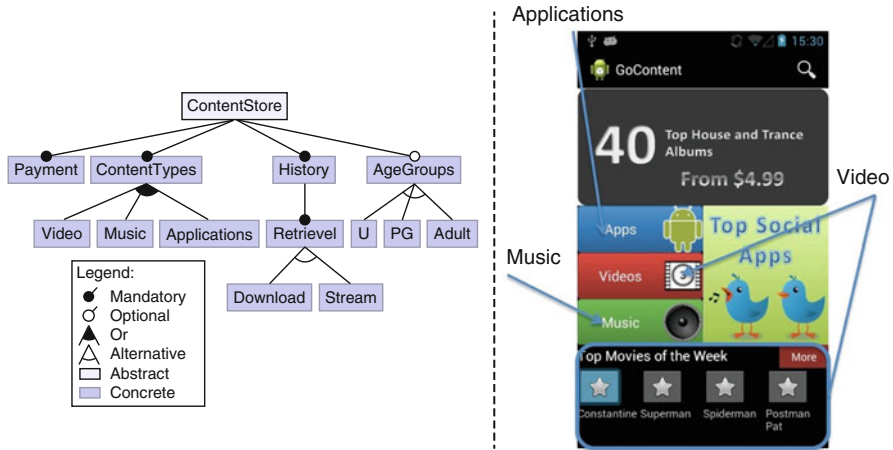


Fig. 2.1 Content store SPL feature model, and example variability

For such a DSPL, a simplified feature model as seen in Fig. 2.1 could be considered. In this feature model, the features *Payment*, *ContentTypes*, *History*, and *Retrieval* are seen to be required in every configuration of this DSPL. The *Payment* feature handles all payment transactions when content is bought or rented. The *ContentTypes* feature contains the different components for browsing, and buying different types of content. For contextual changes, we can make use of feature model propositional logic. Propositional logic can be viewed as a logical representation of the feature model. By the use of logical connectives including  $\vee$ ,  $\wedge$ ,  $\neg$ ,  $\Rightarrow$ , and  $\Leftrightarrow$  with primitive variables (Boolean values), a formula can be determined to be either satisfiable, or non-satisfiable. Formula satisfiability is determined by whether the formula evaluates to true, given a set of variable assignments. Using this satisfiability, context adaptation can be bound to specific application features at runtime. For example, the two *Retrieval* features can be chosen based on space remaining on the mobile device, whereby if the device has sufficient storage remaining and on Wi-Fi, the file can be downloaded on the device. This can be represented using the following constraint:

$$WifiConnected \wedge HighStorageSpace \Rightarrow Download$$

### 2.2.1 Document-Oriented GUIs

Historically, GUIs have been described and implemented within code. Different elements of the GUI including visual properties and controls are created using program statements. This approach requires programming knowledge from the developer creating the GUI.

Within recent years, we have seen the emergence of GUI representation being implemented using documents instead of code [20]. Using this approach, GUI representation is implemented in a more declarative fashion, commonly in markup based languages [30]. GUI documents are often used in conjunction with different UI design patterns for example the Model-View-Controller (MVC) [31]. When using these patterns, the GUI documents are used as a method of implementing the View. Examples of these languages includes Mozilla XUL, QML used in QT, Microsoft XAML, Apple XNib, and Android XMLBlock. While there are differences in capabilities across these implementations, all essential share the ability to declare the layout of different GUI screens in terms of GUI elements/widgets, and their positions. Once the document has been created, it is normally referenced within the main application code, and at runtime, the document is interpreted.

By using a document-oriented approach, there are many advantages discussed in [20] including separation of concerns, compatibility, editability, non-universality and abstraction. Also, with many development platforms, *What You See Is What You Get* (WYSIWYG) editors are included for the GUI documents, allowing developers to preview their GUI without the need for compilation and testing.

### 2.2.2 *Types of Variability*

Central to adaptive GUIs and SPLs is variability. Variability can affect the GUI in several different ways. GUI variability can occur to suit different localisation/internationalisation and design needs. Particularly in the instance of internationalisation and localisation, it has been proposed that products translated for new cultures become entirely new products [33, 40]. It is this variability that allows applications to be tailor made, and make dynamic changes to the application GUI at runtime. Because we wish to consider the ability to handle static and dynamic GUI variability, it is necessary we first analyse the distinct types of variability that need to be handled within such system.

The types of GUI variability that should be handled include the following extension of [37]:

- **Presentation Units:** A collection of UI elements. Presentation units can differ in the quantity of screen real state they utilise, covering a whole screen, or just a particular fragment. Using GUI documents, developers can declare either single UI elements, or more often, a collection of different UI elements in a given layout.
- **UI Elements:** GUIs are composed as a tree of different widgets, or UI elements. These widgets can handle user input, output, or both. An example of these widgets include buttons, editable text fields, labels, and checkboxes. Widgets do not always need to be a visual element, but can also be used for shaping the GUI.

These type of widgets are known as container widgets, as they by default contain widgets. Container widgets can be used to shape the GUI by allowing widgets to be arranged in a specific style including linear layouts, that is placing widgets either vertically or horizontally after each other.

- **Properties of UI Elements:** A property of a UI element corresponds to different logical settings of that element. If we consider UI elements for example checkboxes, and radio buttons, there can be default selections, also textual hints in text fields to add to the user.
- **Dialogue of UI Elements:** By dialogue, we include reactions that require intervention by a user. Interventions required by the user can include input validation or verification. An example can include maximum input lengths, and character types in text fields.
- **Layout:** GUI layouts primarily concerns the positioning and sizes of the different UI elements contained in it. This includes ordering of widgets, when contained in order dependent layouts. Layout variability can be used for several reasons. For instance, it could be used to ensure input field ordering is consistent across corporate applications. It could also be used for handling different localisation concerns, including left-to-right and right-to-left reading users [40].
- **Visual Appearance:** GUI elements by default to be visible require visual properties. Depending on the GUI element in question, different properties can be used to set the visual appearance. Different examples include colours, text font type, borders, background images etc.
- **Orientation:** GUI applications often run on a variety of different screen sizes including mobile phones, and tablets. Different orientations for a given application can be required based on the horizontal and vertical requirements of the GUI for example a game. Screen orientations can be manual set permanently in portrait or landscape, or can also switch orientations based on the orientation of the mobile device. In cases where both orientations are supported, it can be necessary to also alter the orientations of GUI elements, or their positions to increase usability.
- **Behaviour & Interaction:** Behaviour and interaction are both important aspects of an GUI. GUIs carry out tasks for the user, which can include gesture interpretation, and completing actions when particular events occur. Similar to GUI layouts, interactions can be related to localisation issues. For example, if a slide out menu that is designed on the left the screen for left-to-right reading users, it makes sense to intercept sliding actions from the left edge of the screen to the right. Alternatively, for right-to-left users, not only should the menu be displayed on the right side of the screen, sliding actions should be intercepted from the right edge to the left of the screen.
- **Compound Variability:** The previously introduced variability categories can be used in isolation, or compounded and used together. In some cases, to ensure consistency across a GUI, variability in the GUI can include more than a single dimension of variability.

## 2.3 Previous Work

GUI variability is an important issue when handling user facing applications. Therefore, in this section, we present related work in the fields of SPLs, Model-Driven GUIs and Software Engineering, Adaptive GUIs, and Plastic GUIs.

### 2.3.1 *Static GUI Variability*

Systematic reuse of static GUI variability can be traced back to [42], who took advantage of generative programming techniques to generate GUIs from refinement fragments of GUIs. For this method, various parts of the GUI were abstractly modelled within a conventional feature model. Deriving GUI variants was carried out using the tool, ANGLE-Based GUI Generator (ABA), which takes an XML specification of the GUI, generated from a dialog-based graphical-interactive DSL. Model-Based UI Development (MBUID) has been used in conjunction with SPLs for handling UI variability [24]. This work firstly applied this approach to web applications. In this approach, the author proposed a methodology for handling UI within domain and application engineering. In the domain engineering, two types of artefacts are created, a feature model, and domain artefacts. Domain artefacts created include the application core, which is expressed in models for platform independence including a data model & operations, and an Abstract User Interface (AUI) models. Each AUI model can be seen as a combination of task model, and abstract user interfaces from model-based UI development. AUI models are structured in tree hierarchies of different nodes, along with relationships between them using temporal operators. Next, elements in the AUI model are mapped to feature in the feature model, and linked with the application core using a linking element. In the application engineering, the processes of product configuration and derivation is carried out. Product configuration involves the picking of what features should be added to a specific product. Product derivation produces a product model, made up of a UML based Web Engineering (UWE) content model, a UWE user model, a UWE process model, and AUI model. These are then transformed using a semi-automatic stepwise approach using model-based user interface development. This methodology was then applied to web applications creating JSP pages and some limited form of business logic of the application. This work was then described by [36]. The authors also considered the need for manual customisation with automatic UI variant generation. This problem is caused by the need for manual modifications by of an applications UI after product derivation by customers of the system. The authors describe different aspects of MBUID commonly requiring customisation and how these aspects can be customised.

Automated UI generation to tackle the difficulties in creating UIs for different devices and appliances has been proposed by [32]. Generation was carried out in

two stages. The first stage took an appliance specification, written in a DSL, and produces an abstract user interface (AUI). Following this, interface modifications are carried out to ensure consistency among other interfaces created for that device. These modifications can be functional, or structural. From here, a concrete UI (CUI) is created, using platform specific UI objects. This is carried out by traversing the AUI tree, applying CUI rules. Lastly, the CUI is modified for consistency using rules.

Other work concentrating on GUIs within an SPL considered how to re-engineer configurators [13]. In this work, the authors present challenges regarding the reverse engineering of existing configurators analysing GUI, webpage source, and code base to extract variability information. It is proposed that variability information can be extracted by searching for variability and constraint patterns in the GUI, with a few patterns already supported. A TVL model is generated after the user is satisfied with the extracted data in a post-processing step. Additionally, the challenge regarding forward engineering and generating a tailored GUI and codebase is discussed.

Other approaches proposed include rule-based approaches [34]. Using this approach, GUI descriptions are stored within database tables, that are loaded and transformed into object hierarchies. These objects are then converted to CLIPS facts for an engine proposed in previous work [35]. The CLIPS facts are script files that describe different GUI parameters. Two processes are handled by the specified rules, first reconfiguring the UI for a given event, and then adapting the UI. Currently, only rules for hiding and removing certain GUI elements have been realised.

The need for scaling the GUI on mobile devices based on variable screen sizes was presented by [10]. To help graphical scaling, the author proposes the use of Scalable Vector Graphics (SVG). By using SVG graphics for different UI elements on the screen, these can be scaled to suit the screen size more easily, while retaining image quality. To use vector graphics instead of the default raster graphics, the authors propose to override the standard widget drawing methods. These override methods then instead translate the vector graphics to a drawable that can be used by the widget, scaling to suit the given display size.

Toolkits to handle transparent GUI migration and adaptation have been proposed, including those by [22]. This approach was designed to be used with the Mozart Programming System,<sup>1</sup> based on the Oz language providing declarative, object oriented, and constraint programming. Dynamic adaptation of the GUI is carried by each widget having different representations, that can be switched at runtime depending on a given context. Each of these representations is supported by individual renderers, which in turn represent a variant of that widget. These renderers can be distributed and transferred between applications, and even devices over a network interface. Granularity of adaptation supported ranges from the entire screen, to single widgets, and to an arbitrary pixel area.

---

<sup>1</sup><http://www.mozart-oz.org/>

### 2.3.2 *Dynamic GUI Variability*

GUI support in SPLs has predominantly been handled only statically. Handling GUI adaptation at runtime has therefore been mostly from the adaptive GUI community, to which we consider some of the following works. Middleware and language extensions have been used for implementing dynamic variability [18]. This approach used the Context-Oriented Programming language, ContextJ [8], for implementing UI changes in the application. This approach was found to reduce development and testing time, while reducing the number of lines of code when compared with a standard Android application. Other language extensions include the work of [41], who bring adaptivity to static GUIs using a refactoring process in C++. This approach is based on three distinct steps. In the first step, user requirements are analysed, with roles and requirements identified. Next, interface profiles are modelled, expressing variations of the user-interface behaviour. Lastly, adaptation decision logic is identified, and context events and their adaptation rules are specified. In the second step, alternative adaptations are encapsulated, using general superclasses, and each alternative adaptation implemented as a subclass. Replacement at runtime is carried out by component termination, and substitution activation. During component replacement, state is passed between the original component to the constructor of the substitute. Other approaches to dynamic GUI variability include the use of Model-Based GUIs. Hanumansetty [23] proposed a framework for handling web applications, including context processing and interface adaptation off device. Client side contexts are collected from the device and are sent to a context server, where with sensors and other system contexts are aggregated, and interpreted. Context events, are then sent to the business components and interface server. Based on different task model adaptation rules specified, the task model is regenerated for the GUI. Once the task model is updated, the abstract UI is generated using the dialogue model, and then the concrete UI is generated using the presentation model. Generated UIs using approach were XHTML documents.

### 2.3.3 *Mixed Variability*

Now, having introduced static and dynamic variability approaches, we will discuss approaches aimed at dealing with both. One such work that has considered both static and dynamic variability includes *Plastic User Interfaces* (PUI) [14]. Calvary et al. recognised both the need for dealing with both variability in devices in which an application many run on, and the need to deal with environmental changes of the device. The level of plasticity of a user interface is defined as its ability to adapt to different contexts, whereby the more contexts the UI can adapt to, the higher the plasticity. PUIs have been proposed as method for handling both adaptive, and

adaptable UIs. A reference framework for plasticity was proposed, adopted from model-based UI development. Lastly a tool named ARTStudio was developed for developing all models except the environment and evolution models.

Plasticity has been proposed to be modelled as finite state machines, using mealy machines [16] to handle UI resizing operations. Each state in a mealy machine is a resizing operation, with each transition being composed of source and destination of the GUI. This approach then uses UsiXML, a language for defining the final GUI, and is expanded for adaptivity and multi-presentation UIs. The specification language is expanded by adding different concepts including a set of *plasticitydomain*. This set of plasticity domains includes the different conditions that are required for a particular variant of the UI. These conditions include characteristics of the platform, user and/or environment. Each plasticity domain is mapped to a specific GUI representation using inter-model relationships.

Sottet et al. [43] proposed an approach using MDE and SOA for developing PUIs. This approach is based on two principles. The first principle is that an interactive system is a graph of models. These models while developed at design time, still should be available at runtime, and linked by mappings. Concrete UI interactors should be mapped to the platform input and output devices, whereas task and concepts are mapped to the functional core entities. Transformations and mappings are models too, which are expressed in ATL. The second principle is that close-adaptiveness and open-adaptiveness cooperate. This is based around the need for self-contained and sometimes runtime extendable adaptation. Context use and UI adaptation is handled by services in the Distribution-Migration-Remolding middleware. Context observers gather contextual information that is processed by the situation synthesizer. New situations are then sent to the evolution engine to start adaptation. Adaptation can target either a section of, or the whole UI, using a mix of specifications, defined by the developer. The evolution engine then provides the configurator with what components need to be replaced and/or suppressed. These components are then retrieved from the storage space if needed. Further work by [45] promoted a third principle of the keeping the user in the loop. This principle is based on the proposition that the user should remain in control of the UI, even if the UI operation is automatic. To support this, three types of adaptation are suggested including *automated*, *semi-automated*, and *manual transformations*. Designers and users can perform manual and semi-automated transformations. In semi-automated transformations, the designer can adjust the transformation target models at runtime.

The use of Model-Based UI design, and how it can contribute to End-User Software Engineering has been investigated by [19]. Firstly, the concept of Extra-UI, a UI that represents and provides control over a UI, is considered from a End User Programming (EUP) perspective. Secondly, a design methodology is proposed. This methodology includes the design of the core UI, specification of design spec constraints for end users, design of Extra UI, and the coupling of core UI and Extra UI to the final EUP UI.

Context-sensitive user interface creation using DynaMo-AID was proposed by [15]. This included design support and runtime architecture. The DynaMo-AID design process involves the definition of a Dynamic Task Model, Dynamic Dialog Model, a Dynamic Environment Model, Dynamic Application Model, and Presentation Model. The Dynamic Task Model allows for the user to specify temporal relations between tasks including abstract tasks, interaction tasks, user tasks, and application tasks. The Dynamic Dialog Model primarily defines transitions between user interface states. These transitions can be caused by user action, a call from the application core, or the current context of an actor. The Dynamic Environment Model represents context changes, which can also include services. The DynaMo-AID runtime architecture was developed to execute the results of the design process. Once started, the UI can change due to three actors: the actor, the application and the Context Control Unit (CCU). The CCU has the following tasks: detection of context changes, recalculations of mappings from the Concrete Context Object to Abstract Context Objects, selection of the current context-specific task model, and execution of the inter-dialog transition.

A complementary approach to DynaMo-AID has been proposed by [38]. In this approach, the focus was primarily on the creation of dialog graph models. Using the described strategies, abstract prototypes can be generated and refined. With these prototypes, task models can be mapped to the dialog graphs. In addition, task models can be used to control prototype animation, through the use of temporal description of the task model.

Model-Driven approaches have enabled the development of multiple user interfaces, including MANTRA [12]. Using this approach, Abstract UI Models (AUI) are annotated with automated mode flow, tailoring dialogue and logical presentation structures. To generate presentation units, UI elements are clustered together by the identification of suitable UI composites. This stage is carried out first automatically, and then can be refined by human designer. Navigation elements needed to traverse between presentations units are generated. These are implemented as ATL model transformations. The AUI models are then transformed into several Concrete User Interfaces (CUI)s using platform specific model transformations.

## 2.4 Challenges

In previous work, we can see that while there has been research attention on GUI variability, there lacks a truly unified approach. DSPLs offer the promise of unified adaptation. However, it lacks the ability to deal with GUI adaptation in modern platforms. Therefore, in this section we consider the challenges that exist in having unified GUI adaptation using DSPLs. First, we consider the different adaptation approaches that could be used.



### 2.4.1 *Annotative and Composition Approaches*

In conventional FOSD, *annotative* and *compositional* approaches provide two different solutions to the separation of concerns. Annotative approaches focus around the use of virtual separation of concerns [27]. Examples of tools that support annotative approaches include the C preprocessor, and CIDE tool [21]. Using these tools, the developer can annotate parts of the source code that are part of each feature in the SPL. This approach can allow very fine grained adaptation including extra statements to methods, and parameter alterations in method declarations. Product derivation is carried out by negative variability. Using negative variability, parts of the system are removed based on which features are present in the product configuration [46]. This causes code to be removed from the final variant of the source code if its associated feature is not included in a product.

Compositional approaches on the other hand focus around physical separation of concerns. By physically separating code into multiple modules, these can be composed into different variants at configuration [28]. This approach normally makes use of positive variability because elements that are variable to the product are added to the base product [46]. Many languages supporting software composition exist include Aspect-Oriented Programming, Delta-Oriented Programming, and Feature-Oriented Programming. Using a compositional approach for GUI variability, the following paths can be used:

- **Compile-Time Composition:** This process functions on the derivation and generation of all foreseeable valid variations of a given GUI resource at or before application compilation. Then at runtime, the correct variant is used when it is required, based on the current DSPL feature configuration. This approach can lead to fast reconfiguration, as no composition takes place at runtime. However, this can lead to scalability limits regarding space and time at program compilation. Furthermore, the final DSPL application can eventually becoming restrictively large, if there are high amounts of GUI variability.
- **Runtime Composition:** This process functions on the derivation and generation of the correct GUI resource at runtime. Using this approach, the DSPL program is compiled with the GUI refinements for each of the selected features. Then at runtime, based on the current feature configuration, each of the refinements of a given GUI resource are composed together to create that GUI resource variant. This approach unlike the compile-time approach removes the need for generating and storing all foreseeable variants, which will need to be installed with the application. However, this approach does require the DSPL requires a far higher static overhead including all composition tools required to compose the GUI resources. This can lower performance of the adaptation, and with many modern platforms, GUI resources require preprocessing during compilation, further complicating the process. In proprietary software development kits, including these tools is of considerable difficulty.

### 2.4.2 *Configuration Timing*

During the reconfiguration of a DSPL, the running application is adapted instantly. While logic adaptations can be applied instantly, this might not be desirable for all GUI adaptation. Particularly when considering the usability of the GUI, it can become increasingly confusing to the user when UI elements appear and disappear on the screen [26]. It could still be the case that updating the properties of already existent UI elements on the GUI may not cause this issue. Therefore, we propose that the developer should be capable of applying GUI adaptation at different times to suit different adaptations.

To support different GUI adaptation times, we consider two separate phases, where adaptation in a dynamic GUI may be applicable:

1. **On Inflation.** The inflation of a GUI can be regarded as when the GUI is created, either when the application first starts, or during a GUI transition from one screen to another. It is possible that substantial amounts of adaptation for example presentation units, may be best suited for this time in the GUI lifecycle.
2. **While Active.** We consider that a GUI is active when it is currently visible to the user in the foreground of the device. During this stage in the GUI, smaller adaptation for example UI element properties could be more suitable for deactivating a download button in the situations where an internet connection is not present.

### 2.4.3 *GUI State*

GUIs in a mobile application are rarely stateless. State in the GUI can be altered directly by user input for example text in an editable text field, or can be indirectly altered for example the position of a video playing in a video container. When an adaptation occurs, we need to ensure that state of the different UI element remains. State in GUI during adaptation can be retained in a number of ways. The first method is that adaptations of UI elements only updates the properties of an instance, leaving the rest of the properties the same. The second is to store the state of the widget, carry out the adaptation, and copy the state back. In the context of using GUI documents, the second option will have to be used. Having said this, it may not always be possible to store all UI state data, due to different platform constraints. If this the case, the developer will have to decide if the variability will have to remain static.

### 2.4.4 *Adaptation Isolation*

A GUI can require different amounts of adaptation ranging from a UI element property alteration to whole presentation units. These adaptations should be carried

out only on the GUI widgets requiring adaptation. Widgets in the GUI can take many different roles, and can carry out tasks for example video playback. If during an adaptation video playback is stopped, or affected by pausing, this could potentially be frustrating to the user. Therefore, unless a widget is being adapted, it should be left to function normally, even during adaptation. This should lead to less noticeable adaptation transitions. While for many adaptive approaches, this isolation can be a straight forward to handle, for GUI documents this is less trivial. Existing GUI frameworks commonly parse each GUI document at runtime when it is required by the system. These are parsed whole and create the GUI from all the properties that exist in that document. There is often no ability to parse fractions of these documents, applying only what properties need to be changed. We therefore need to ensure that despite reading variants of the GUI as a whole document, we still only adapt the UI elements that require adaptation.

### **2.4.5 Consistency**

GUIs in modern platforms are often proposed to use specific design patterns including the Model-View (MV), Model-View Controller (MVC), and Model-View-ViewModel (MVVM). These patterns normally involve multiple linked artefacts. In terms of a SPL, this can equate to each of these artefacts containing variability relating to either the same cross cutting feature, or different features. For this reason, it is important to ensure that all artefacts are consistent in terms of shared elements to prevent inconsistency errors. Examples of these errors include attempting to add event listeners to a non-existent button, attempting to change the visual properties of a GUI element, or declaring an event listener which is yet to be implemented in the controller. It is not safe to assume a targeted platform will statically check for these inconsistencies, as some including the Android platform deal with artefact bonds at runtime, not at compile-time.

## **2.5 Variability Design and Implementation**

Considering conventional DSPL approaches, handling GUI variability would require GUI implementation in the platform host language. These approaches therefore result in the developer not being capable to leverage the use of GUI documents for both static and dynamic variability. This either leads a developer to either use different implementation for static and dynamic, or they choose not to use GUI documents at all. In our approach, we consider how static and dynamic support can be reached while using GUI documents to implement GUIs. It is our goal to support a single code base for GUI variability using GUI documents.

### 2.5.1 Document Refinement

To implement variability using GUI documents, we propose to follow a refinement approach used in Feature-Oriented Programming (FOP) [3]. Refinements to GUI documents bring about changes to the GUI they represent. Refinements allow for widgets displayed to the user to either be added, or be altered to have a different property e.g. colour, shape, and size. Each refinement is implemented using physical separation of concerns, whereby each refinement is contained within its own file. These refinements are then contained within feature modules, which also contain source code refinements and documentation for feature cohesion.

**Listing 2.1** An android GUI document refinement

```

1 <FrameLayout
2     android:id="@+id/mainFrame">
3     <LinearLayout
4         android:id="@+id/mainlayout">
5         <LinearLayout
6             android:id="@+id/contenttypes">
7             <Button
8                 android:id="@+id/videos"
9                 android:layout_width="160dp"
10                android:text="@string/videos" />
11            </LinearLayout>
12        </LinearLayout>
13    <LinearLayout
14        android:id="@+id/adverts">
15        <LinearLayout
16            android:id="@+id/videoAd
17            .... >
18        <TextView
19            android:id="@+id/TopMovies"
20            android:text="@string/TopMovies"
21            .... / >
22        </LinearLayout>
23    </LinearLayout>
24 </LinearLayout>
25 </FrameLayout>

```

Each GUI document is made up by GUI nodes forming a tree. Each of these nodes is expected to have a unique identifier, which can be used for searching and identifying in the tree. In Listing 2.1, we depict a GUI refinement that refines the content store application main screen. This refinement adds additional buttons to allow the user to browse video content types. As this refinement is implemented for the Android platform, the GUI element identifiers are defined with the `android:id` XML

attribute. To add additional GUI element to a GUI tree, all parent nodes are needed. In the given example, we can see that a button named with `@+id/videos` has been added to the tree. Refinements are not designed just for compositions of new GUI elements, they can be used for the adaptation of ones currently included. This is accomplished using node overriding. During refinement composition, when the base document has a node also present in a refinement, the properties of the node in the refinement are either added, or used to replace existing properties of the same name in the base document.

### 2.5.1.1 Refinement Ordering

In GUI document refinements, ordering can be an important consideration. In many platforms, the relative position of GUI elements is guided by the GUI document structure. Therefore, if a button in a vertical fragment of the GUI is defined before a text field in the GUI document, it will also be placed that way on the GUI. Compositional ordering can partially accomplish this need; however it does not assist in conditions where a UI element is required before an element in the base feature. This consideration has been partially addressed in existing tools, including XAK [1], which gives the ability to place items either at the beginning (prepend) or at the end (append) of the parent node. Having said this, when handling shallow trees, this still does not give enough inter-branch precision. We propose the ability to compose nodes either before, or after a given node within a shared parent node using a set of keywords. This approach can be used on multiple nodes, by placing the list of nodes within that particular keyword block.

**Listing 2.2** Refinement ordering tags

```

1 <!-- @start before android:id="@+id/btnApps" -->
2 <Button
3     android:id="@+id/btnVideos"
4     android:layout_width="match_parent"
5     android:layout_height="59dp"
6     android:contentDescription="@string/videos"
7     android:text=@string/videos />
8 <!-- @end before android:id="@+id/btnApps" -->
9
10 <!-- @start after android:id="@+id/btnApps" -->
11 <Button
12     android:id="@+id/btnVideos"
13     android:layout_width="match_parent"
14     android:layout_height="59dp"
15     android:contentDescription="@string/videos"
16     android:text=@string/videos />
17 <!-- @end after android:id="@+id/btnApps" -->

```

Many software development kits allow for graphical editing of GUI documents, allowing the developer to preview their GUIs before compilation. To avoid parsing errors within these tools caused by unknown tags, we embed these keywords within source code comments. Source code comments are readable annotations within source code, which are often used as form of documentation. In Listing 2.2, we depict two examples of placing a button before, and after a particular widget in an Android GUI document. In this motivating application, we would use the `after` comment for both Music and Video buttons to make sure each button is underneath the Applications menu button. When using ordering comments, each set of widgets need to be placed with a `@start` and `@end` comment. Each comment block requires the position needs to be specified, `before` or `after`, followed by the identifier of the widget involved. In the cases where a widget identifier is not given, or if the specified widget does not exist in the existing parent node, the comments behaviour is identical to XAK, whereby the widgets will be place either at the start or end of the parent node set. Using this approach in conjunction with composition ordering, widgets can be placed far more precisely within a GUI document.

### 2.5.2 Source Code Variability

So far, we have described how we express variability in GUI documents, which form the *View* as part of the Model-View Controller pattern. As part of the GUI, other elements handled in source code may also exhibit variability including behaviour e.g. button on click listeners. To handle this variability, we do not propose a novel approach to source variability, but instead use existing language approaches. These existing approaches include Feature-Oriented Programming [3] and Context-Oriented Programming [25]. Feature-Oriented Programming (FOP) is a programming paradigm for modularising software according to *features* [9]. Using FOP, classes are implemented as standard classes within a base language e.g. Java, C++. To implement variability to classes defined within a feature, *refinements* are used. Refinements contain adaptation for a particular class which allows extra class members, methods, and method extensions to be added. Method extensions allow the addition of instructions by method overriding, and using the `super` keyword to create an execution chain. The position of the `super` can dictate when the execution of the extensions should take place.

## 2.6 Generating the DSPL

In the last section, we described how the developer can design, and implement their DSPL variability. Now we describe the steps required to generate a DSPL that incorporates GUI variability. The first step is *configuration*. In a static SPL, a configuration determines the features that are present and are always bound in a product. For a DSPL, configuration can include both what features need to be

available to bind, and what features of them should be bound by default. These features must be chosen, and the initial product feature binding configuration needs choosing. DSPLs do not always need to contain dynamic features, but can also incorporate features that have been statically bound to a product. The use of static features in a DSPL through the use of composition which are the base application or composition to other dynamic features are called *dynamic binding units* [39]. Dynamic binding units are essentially compound features which are created by statically composing features together. These compound features are then used at runtime for dynamically binding feature refinements.

### 2.6.1 Static Variability

Firstly, when generating the DSPL, we can handle any static variability in the SPL. Static variability is handled using the principles of *superimposition*. For many different approaches, superimposition has proven to be a successful feature composition technique [2, 4–7]. Depicted in Fig. 2.2, we illustrate how the GUI trees are superimposed, producing the final GUI document. With the opening screen of the content store, various buttons are used for adverts and to specific content types for example video, music etc. As introduced in the Sect. 2.2, because of different distribution rights, not all content types may be accessible in every country of use. We handle this change by adding the buttons designed to take the user to that specific content. While superimposition has been acknowledged as not being a silver bullet, it has also been applied to other non-programming languages including UML [2].

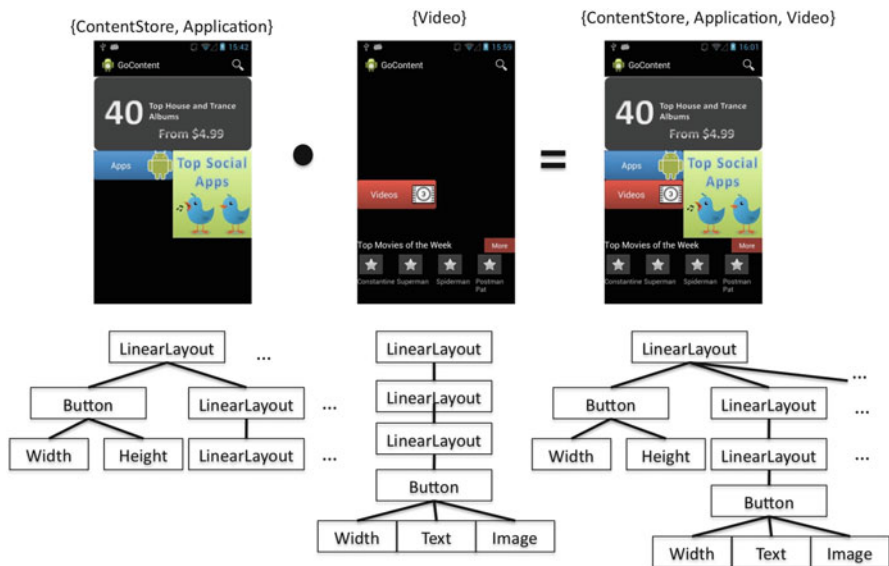


Fig. 2.2 GUI composition

As there will be features that require binding at runtime, based on certain contextual conditions, there can also be features that can be statically bound. An example of where this can be of use include the age group set of features in our scenario application. The content store can be derived to suit a distinct set of users. As part of a program variant designed for school devices, the GUI and functionality can be designed to suit that group of users. This variability may not need to be decided at runtime, and therefore can be statically bound. By combining static and dynamic feature binding, we can apply an approach used in FOP, whereby features are joined to create composite features, also known as dynamic binding units [39]. Using dynamic binding units can be seen to follow a staged configuration [17], whereby product derivation is not carried out within a single step, but more over several steps, each specialising the product more. This staged configuration leads to a final feature model, which exhibits only dynamic variability needed by the system.

## 2.6.2 *Dynamic Variability*

Next, after static variability has been handled, dynamic variability must be processed. This is handed with a combination of GUI variant generation, and source code generation and transformations.

### 2.6.2.1 GUI Variant Generation

To compose all variants, we first need to generate all needed runtime variants, as depicted in Algorithm 1. When considering variant generation, it is important to produce only unique documents. If variants are generated by every possible configuration of the system, it is likely that there will be many document duplicates. Duplicates can occur because it is unlikely that a document will be refined in every

---

#### **Algorithm 1** Generate all GUI variant configurations

---

**Input:** A set of relative ordered *Features*  
*Guis* ← GETALLGUIREFINEMENTS(*Features*)  
**foreach** *Gui* ∈ *Guis* **do**  
    *combinations* ← GETREFINEMENTCOMBINATIONS(*Gui*)  
    **foreach** *comb* ∈ *combinations* **do**  
        *config* ← NEWCONFIGURATION(*comb*)  
        *config.propagateFeatures*()  
        *valid* ← *config.isValid*()  
        **if** *valid* = true **then**  
            ADDCONFIGURATION(*Gui*, *Config*)  
        **end if**  
    **end foreach**  
**end foreach**

---



single feature, leading to more than one feature configuration for a given variant. We avoid this by only generating unique and valid GUI variants.

We therefore find all GUI refinements (line 1). Refinements are found by traversing through each feature module in composition order. Composition order is a relative order that can be set by the user to specify in what order each feature should be composed in a stepwise fashion. This step ends with a two-way mapping between what features refine each document. Next, for each GUI document name (line 2), we compute all combinations of features that refine that document (line 3). By carrying this out, we get every unique variant of the document, in terms of features in a configuration.

While all document variants must be unique in their content, they must also be valid in terms of the feature model. Currently, we only compute all combinations of features for each dynamic document. We then need to filter all variants that contain an invalid configuration, in terms of the feature model. For each combination (line 4), we then create a configuration that can be tested for validity. To do this, we initialise a new configuration, manually selecting every feature in the combination (line 5). This is followed by propagating all automatic feature selections (line 6). Automatic feature selections include feature selections based on either model structural relationships, or using model constraints. An example of this include selecting any feature parents of manually selected features. The step should end with features being selected explicitly, or implicitly. We then check the given configuration for satisfiability (line 7), using a SAT Solver. The configuration is checked by encoding it into conjunctive normal form [11, 44], creating a SAT problem that can be reasoned over by an off-the-shelf SAT solver, for example SAT4J.<sup>2</sup>

### 2.6.3 Source Code Generation and Transformation

Following GUI variant generation, other source code transformations are required to produce a working DSPL. The first includes variant management.

#### 2.6.3.1 Variant Management

The prime responsibilities of the variant manager are to handle at runtime which GUI variants should be used based on the system configuration. Within this generated class, there includes:

- **Feature-Variant Map:** As all valid unique variants of each GUI document, we need to map to which configuration a particular variant is required. This structure

---

<sup>2</sup><http://www.sat4j.org/>

is a key-value structure, with the collection of active features as the key, and the variant reference as the value. To avoid duplicate variants across different configurations, each key only contains the active features from a configuration which actually adapt that GUI document.

- **Feature Array:** This array compliments the feature-variant map, by which is lists which features are associated with a particular GUI document. During variant retrieval, this list is used to remove unrelated features from the key looked up in the feature-variant map.

When a GUI document variant is required during GUI adaptation, or when the GUI is first inflated and shown to the user, it is requested via a single method. This entry point takes the name of the GUI document in question, which then invokes a specialised method for that particular GUI document name. If the GUI document for that request name only has a single variant, and is therefore not managed by the manager, the original reference is returned and is used normally as before. If, however a specialised method is invoked, first we copy the list of active features in the current configuration. Next, Next, all features not associated with the GUI document in question are removed from that list. To remove the un-associated features, we use the GUI document feature array, and remove features from the configuration list that are not in the feature array. Next, we sort the features into alphabetical order, and output the list as a single string. But ensuring the different features are in alphabetical order, we ensure that a single configuration can only produce a single string. Lastly, a map lookup using the string of features as the key, with the resulting GUI document reference being returned.

#### ***2.6.4 Transformations***

The first transformation that takes place is the need to update references to GUI documents within the source code. To use the correct variant of any GUI document required, all calls to any GUI document should be made through the variant manager instead of direct calls. This ensures that if there are multiple variants, it will use the correct variant. In Android, two methods calls including `setContentView`, and `inflate` read GUI documents. The `inflate` method reads a GUI document and returns a GUI tree which can be made viewable to the user. The method `setContentView` on the other hand can be seen to both inflate the GUI document, and then set the currently viewable GUI to that inflated GUI tree. It is these methods that require the correct GUI variant when they called. Therefore, if these methods are used within classes or class refinements, we simply alter the method parameter to get the variant from the variant manager instead, as shown in Listings 2.3.

**Listing 2.3** Java method transformation

```

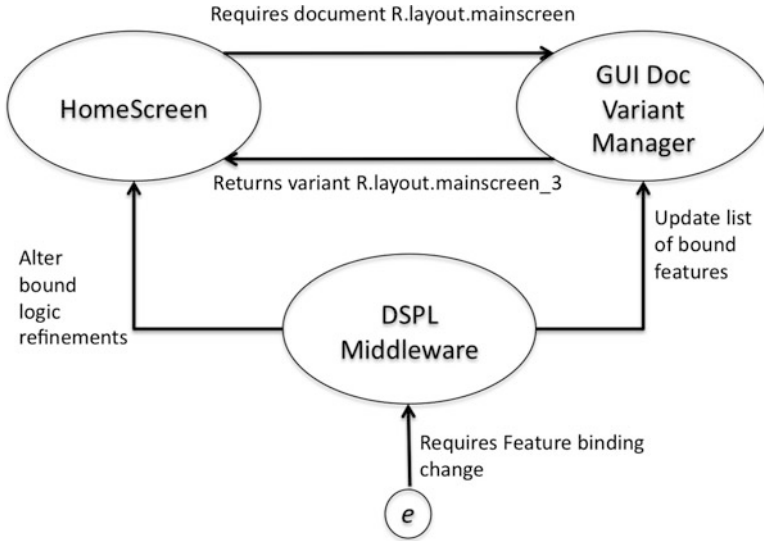
1 // Original Implementation of onCreate() method
2 @Override
3 protected void onCreate(Bundle savedInstanceState ) {
4     super.onCreate( savedInstanceState );
5     setContentView(R.layout . activity_main );
6 }
7
8 //Implementation after transformation
9 DSPLResourceGetter dsplrg = (DSPLApp)getApplicationContext().
10     getDSPLRG();
11
12 @Override
13 protected void onCreate(Bundle savedInstanceState ) {
14     super.onCreate( savedInstanceState );
15     setContentView( dsplrg . getResourceVariant (
16         R.layout . activity_main ));
17 }

```

## 2.6.5 Runtime System Behaviour

During program execution, depending on which SPL features are in the active configuration, the correct GUI variant needs to be chosen when requested. To achieve this, the variant manager maintains a list of currently bound features, which is updated when feature binding changes.

In Fig. 2.3, we illustrate an adaptation scenario in the home screen. In this scenario, different contextual events ( $e$ ) can be received by the DSPL middleware that handles context-awareness, and SPL feature and configuration management. Contextual events can cause a configuration change in the system, meaning different features can become bound and unbound. After a valid feature reconfiguration has taken place, logic refinements in the main screen can be altered. This means different refinements containing adaptation can either become bound, or unbound and removed. During reconfiguration, the GUI document variant manager receives the updated list of currently bound features in the SPL. During GUI document variant is needed, in the case of GUI adaptation, or when the GUI is first inflated, it is requested via a single method. This entry point takes the name of the GUI document in question, and then invokes the specialised method needed for that GUI document name. If the GUI document for that name has only a single variant, and is therefore not managed by the manager, the original reference is returned and can be used as before. If a specialised method is invoked, first we copy the list of active features in the current configuration. Next, all features not associated with the GUI document



**Fig. 2.3** Graphical Illustration of the runtime behaviour for the home screen

in question are removed from that list. To remove the un-associated features, we use the GUI document feature array, and remove features from the configuration list that are not in the feature array. Next, we sort the features into alphabetical order, and output the list as a single string. But ensuring the different features are in alphabetical order, we ensure that a single configuration can only produce a single string. Lastly, a map lookup using the string of features as the key, with the resulting GUI document reference being returned.

## 2.7 Implementation

In this chapter, we have described an approach for handling GUI variability in DSPLs. This approach combines variant generation, code generation and transformations. As part of our validation, we implemented modelling tool support and specific Android implementations to support our approach.

### 2.7.1 Tools

To validate our compile time composition approach, tool support was implemented on top of *FeatureIDE* [29]. *FeatureIDE* is an Eclipse plugin to support Feature-Oriented Software Development. GUI document composition is handled through

our extension of *FeatureHouse* [4]. *FeatureHouse* provides a language independent solution to software composition. When *FeatureHouse* composes a project, it creates a Feature Structure Tree (FST) model. Within a FST model, different FST nodes represent different elements of a given artefact, for example package imports, classes, and methods. There are specifically two types of nodes, nonterminal, the inner nodes of the tree with recursively further nodes and terminal, the leaves in the tree. These FSTs are then composed by superimposition, in a stepwise fashion. Because of the tree structure form of GUIs, it is straight forward to create FSTs of GUIs. Within the GUI, different GUI widgets and layout holders are nonterminal nodes, and each widget property is a terminal node. During composition, widget properties are either added, or replaced if a new property value is found.

### 2.7.2 *Android Implementations*

Our approach while applicable to most GUI frameworks that use GUI documents or GUI description languages, has been implemented for the Android platform. For our implementation, a number of classes are generated including the Variant Manager. In Android, resources including GUI documents are referenced using static integer values that are within a generated class named R. For every handled GUI document by the manager, a generated method is added. Because GUI document are referenced using integers, we can delegate which method is required by use of a simple switch statement. This method does the variant lookup for a specific GUI document, by which it gets the list of active features that are known to refine that GUI document, and then do a map lookup. Only the active features that refine GUI document are used because as said earlier, it is possible to have duplicate variants for multiple configurations. This would require far more variants to configuration maps, which is unnecessary. By only storing configurations of features that refine a specific GUI document, and its variant, we can greatly reduce the overall map structure.

In addition, for runtime DSPL management, a reusable manager for handling feature selection at runtime was developed. This middleware is designed to function as part of a DSPL application, or be used externally by many applications. When feature change for the application happens, the active feature list within the resource manager are updated either by Inter process communication when used externally, or by direct object alterations when used as part of the DSPL application.

## 2.8 Examples and Discussion

The goal of DSPLs is to be able to handle both static and dynamic variability. Our work attempts to bridge the gap closer to also handling the GUI also. In this section, we aim to highlight some examples using the approach described earlier, and discuss some of the current limitations of the approach, with future and ongoing work described last.

## 2.8.1 Examples

Having described our approach earlier in this chapter, we now attempt to show its applicability using examples from the SPL introduced at the beginning. In Sect. 2.2.2, we introduced nine categories of GUI variability. These categories are based on different aspects of the GUI. Many of these categories however are often implemented in the same way. As an example, if we consider the categories *properties of UI elements* and *visual appearance*, they are both implemented on Android by either adding or refining different XML node attributes. As a result, we show applicability of the approach through the following two examples:

- **GUI Elements.** This type of variability includes adding and removing whole GUI elements e.g. buttons, and text fields. These GUI elements can be singular, or can be containers of other GUI elements. This implementation variability can be used for implementing presentation units, UI element, and layout variability described in Sect. 2.2.2.
- **GUI Element Properties.** This type of variability encompasses alterations to different properties of GUI elements. When considering GUI documents, this means altering what attributes exist, and or, the attribute values. This implementation variability can be used for UI element property, dialogue, and visual appearance variability in Sect. 2.2.2.

### 2.8.1.1 GUI Elements

In this example, we show an example of handling whole GUI elements within your GUI refinements. These GUI elements can be both single widget e.g. buttons, and text fields, or can be layout widgets. For this type of variability, we use the ContentStore application home screen. In this main menu, depending on the geographical location of the device, the user has access to different types of content including applications, videos, and music. This screen therefore has different elements related to different content types. In Fig. 2.4, we illustrate screenshots of the screen with and without the videos feature. In the application feature model, we need the features *ContentStore*, *Applications*, *Music*, and *Videos*. The ContentStore feature in terms of the home screen has the base structure of the GUI including advertisements and GUI containers. Buttons and screen areas for specific content types are contained within *Applications*, *Music*, and *Videos*. Since the Applications feature is default in all geographical regions, a specific content rule to enable it is not required, and instead have it active in the initial configuration.

Now, let us consider the different source refinements required by this example. For the home screen, a number of refinements are required for the Videos feature. These refinements include a button to take the user to the section of the application destined for selling video content. In our application, the video button should always be directly after the button for applications. To ensure this, we use the `after` refinement ordering statement. Also, a group of video advertisements to show popular videos is included in the refinement.

**Listing 2.4** Main activity document refinement

```

1 <FrameLayout xmlns:android="http://schemas.android.com/apk/res/
  android"
2   android:id="@+id/framelayout" >
3   <LinearLayout android:id="@+id/mainlayout" >
4     <LinearLayout android:id="@+id/corebuttons" >
5       <LinearLayout android:id="@+id/contenttypes" >
6
7       <!-- @start after android:id=''@+id/apps'' -->
8         <Button
9           android:id="@+id/videos"
10          android:background="@drawable/videos"
11          android:text="@string/videos"
12          .../ >
13       <!-- @end after android:id=''@+id/apps'' -->
14     </LinearLayout>
15   </LinearLayout>
16   <LinearLayout
17     android:id="@+id/videoads"
18     android:orientation="vertical"
19     .... >
20     <LinearLayout
21       android:layout_width="match_parent"
22       android:layout_height="wrap_content" >
23       <TextView
24         android:id="@+id/videoAdvertTitle"
25         android:text="@string/advert_title"
26         .../ >
27       <Button
28         android:id="@+id/btnVideoMore"
29         android:onClick="movieAdClick"
30         android:text="@string/More"
31         .../ >
32     </LinearLayout>
33     <LinearLayout
34       android:id="@+id/videoadcontainer"
35       .../ >
36   </LinearLayout>
37 </LinearLayout>
38 </FrameLayout>

```

In Listing 2.4, we have the GUI document refinement to add the Videos button, and the container for holding video advertisements in the `LinearLayout` named `videoads`. This container contains a title, a button to take the user to a larger list of videos in order of popularity, and an empty container named

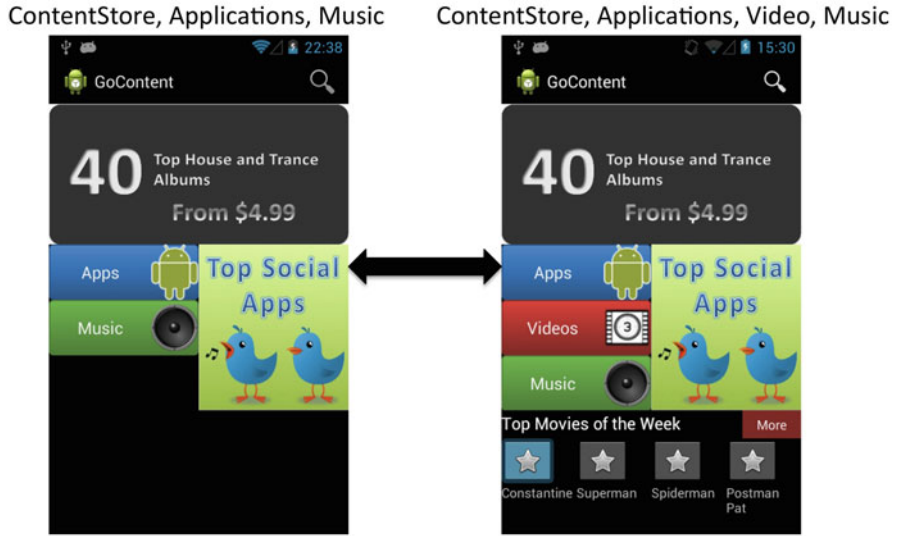


Fig. 2.4 Application main menu

videoadcontainer. This container will be used to hold each individual advertisement, each of which are instances of a GUI document named videoadview.

Next, we describe the supporting class refinements for this example.

In Listing 2.5, we show the refinement added to the MainScreen activity. This refinement primarily extends the Activity class onCreate method, and includes other methods linked to logic additions including the method goToVideoStoreScreen. Included in this method refinement are the operations to add a listener to handle touch events on the main video button, and logic to populate the videoadcontainer element container with instances of the videoadview GUI document for each movie.

Listing 2.5 MainScreen class Refinement

```

1  public class MainScreen extends Activity {
2  public void onCreate(Bundle savedInstanceState) {
3  original ();
4  Button btnVideo = (Button)vg.findViewById(R.id.video);
5  btnVideo.setOnClickListener(new OnClickListener() {
6  public void onClick(View v) {
7  gotoVideoStoreScreen ();
8  }
9  });
10 ArrayList<VideoAdvert> videoads = getVideoAdvertisements ();
11 ViewGroup root = (ViewGroup) this.findViewById(R.id.
    videoadcontainer );
12 for (VideoAdvert ad : videoads) {

```



```

13     LinearLayout newMovie = (LinearLayout)View.inflate ( this , R.layout
        .videoadview, null );
14     TextView movieName = (TextView) newMovie.findViewById(R.id.
        videoName);
15     movieName.setText(ad.getName());
16     ImageView movieImg = (ImageView) newMovie.findViewById(R.id.
        videoImage);
17     movieImg.setImageBitmap(ad.getImage());
18     root .addView(newMovie);
19 }
20 }
    
```

### 2.8.1.2 GUI Element Properties

This next example illustrates how the approach can adapt properties of GUI elements of a window. In this example, we consider a window designed to allow a user to download, install (in the case of applications), stream (in the case of videos), and review that content. This window is affected by network connectivity and battery capacity remaining on the device. In Fig. 2.5, we illustrate two different configurations of the GUI, with each of the relevant features stated above.

In Listing 2.6, we have excerpts of different GUI document refinements. We do not have room to have all refinements, so we illustrate just some of them. As this

History, Retrieval, Stream, Download, ContentReviews, UserReview

History, Retrieval, NoStream, NoDownload, ContentReviews, NoUserReview

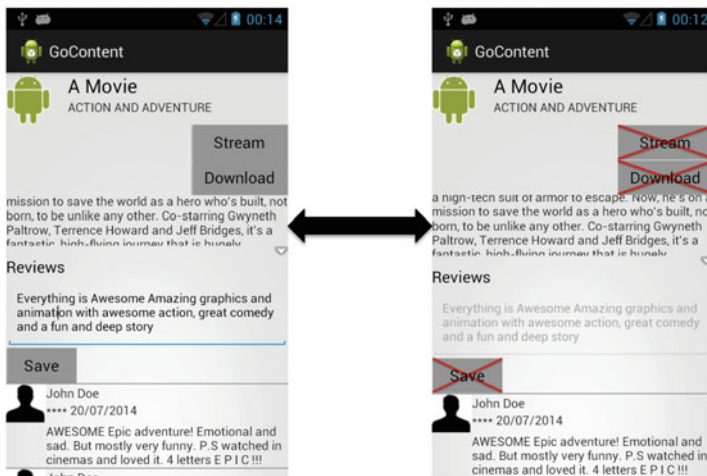


Fig. 2.5 Content detail screen

GUI is broken down over multiple GUI documents, we need to refine multiple GUI documents. Two of the refinements are for the `contentdetailheader.xml` document, which both add a download button, but with different properties. The first is designed for when the Download feature is activated, and the user can download that specific content. The second is when the NoDownload feature is activated, and the user cannot download content due to specific contexts. The last refinement disables a button responsible for saving user reviews for a specific item. The next set of refinements required for this scenario include refinements to the Android activity source code.

**Listing 2.6** Main activity document refinement

```

1 // contentdetailheader .xml refinement in feature "Download"
2 <LinearLayout android:id="@+id/contentdetailheader" >
3   <Button android:id="@+id/downcloudcontent"
4         android:layout_width="wrap_content"
5         android:layout_height="wrap_content"
6         android:background="@drawable/can"
7         android:text="@string/download" />
8 </LinearLayout>
9 // contentdetailheader .xml refinement in feature "NoDownload"
10 <LinearLayout android:id="@+id/contentdetailheader" >
11   <Button android:id="@+id/downcloudcontent"
12         android:layout_width="wrap_content"
13         android:layout_height="wrap_content"
14         android:background="@drawable/cant"
15         android:text="@string/download" />
16 </LinearLayout>
17 // contentreviews .xml refinement in feature "NoUserReview"
18 <LinearLayout android:id="@+id/contentreviews" >
19   <EditText android:id="@+id/txtreviewsValue"
20         android:enabled="false" />
21   <Button android:id="@+id/btnSaveReview"
22         android:background="@drawable/cant" />
23 </LinearLayout>

```

We consider how the GUI should behave in Listing 2.7. This excerpt implements the behaviour required by the system if the application cannot connect to the content store because of a lack of internet connectivity. This source code refinement is used along with the visual changes made to `contentreviews.xml`. This refinement implements an error message to the user by use of an Android Toast. To implement this behaviour, we had to add additional logic to the GUI document initialisation method `onCreate`.

**Listing 2.7** GUI document initialisation refinement for ContentDetails.java class in feature NoUserReview

```
1 public class ContentDetails extends Activity {
2     public void onCreate(Bundle savedInstanceState ) {
3         original ();
4         btnSaveReview = (Button) vg.findViewById(R.id.btnSaveReview);
5         btnSaveReview.setOnClickListener(new OnClickListener() {
6             @Override
7             public void onClick(View arg0) {
8                 Toast toast = Toast.makeText(mContext, R.string .
9                     error_cantsendreview ,
10                                     Toast.LENGTH_LONG);
11         }
12     }
```

## 2.8.2 Limitations

The approach described proposed a solution for handling GUI variability at runtime. However, this solution does contain limitations which we aim to outline next.

### 2.8.2.1 Configuration Timing

Configuration timing is an important challenge as we set out earlier. For full runtime adaptation, any approach needs to handle both phases of the GUI lifecycle, *on inflation* and *while active*. Currently in the proposed approach, adaptation while the GUI are only realised during the inflation phase of the interface. This means that changes can only take place during transitions between GUIs. Work is currently being carried out to allow adaptation while GUIs are in the active phase too. These adaptations are based on the adaptation of the tree of GUI elements, not just reinflation of a fresh GUI. This will lead to a far faster, and less noticeable transition.

### 2.8.2.2 Artefact Inconsistencies

With the use of GUI documents, separation of concerns regarding UI development can be achieved. This separation of concerns always the visual representation to be separated from the business logic, using design patterns such as Model-View Controller. By separating these concerns, you have however linked software artefacts. This link however is often implicit, and due to the how GUI documents are often designed to be interpreted and used at runtime, these links are not statically checked. This means that the developer must be sure that no inconsistencies between

the elements declared in the GUI document, and the business logic occur. These inconsistencies can include business logic regarding non-existent GUI elements, or declaring event handlers within a GUI document with no implementation within the GUI controller. This inconsistency is often easily manageable by the developer when developing single applications, as often exception handling within developer frameworks can give the developer feedback if and when the application fails during runtime. For an SPL engineer however, when many unique features can refine such software artefacts, this inconsistency can become increasingly difficult to manage by the developer. As part of future work, we suggest the following checks:

- **Controller to View:** This checks between view references made within a controller to the view. The duty of this check is to ensure widgets referenced in controllers exist within the GUI document that is used. To do this, we can check which GUI documents are used within a controller through the `setContentview`, and then for each View referenced within that controller, check it exists in any of the used GUI documents.
- **View to Controller:** This checks between controller references made in the GUI document to the controller. In this check, we ensure that event handlers set in a GUI document are implemented within the controllers to which use those GUI documents.

### 2.8.3 *Performance and Storage Consumption*

Because of the particular approach taken, one of the areas we wish to examine is *application bloat*. For this, we describe application bloat as the amount of storage needed by repetitive code found in more than one GUI variant. To gain a sense of the size of the GUI documents, we consider both before pre-processing, and afterwards, where they would be part of a compiled application. To obtain the size of the pre-processed documents, we compile the Android application to an application APK ready for deployment, to which we then inspect the file contents of the application resources. Non pre-processed GUI documents were all reduced to a condensed format using a single line, thus avoiding discrepancies due to differences in XML tag formatting, and indentation. As a running example, here we consider how much bloat is found related to the first example of GUI elements, shown earlier. In this example, we consider the age groups to be set static, and only content type to be dynamically variable, based on the location of the user. Because there are three refining “Or” features, *Applications*, *Music*, and *Video*, this equates to seven variants of this GUI.

The size of each variant varies from 1.8 KB (2 KB when pre-processed), when only Applications or Music are bound, up to 3.2 KB (3.3 KB when pre-processed) when all three features are bound. This combined size can be considered the optimal

amount of storage required for this variability. When considering that the size of all the variants is 17.6 KB (18.7 KB pre-processed), and the size of the base GUI document and the refinements are 4 KB, we can assume there is 13.6 KB of bloat. This means that of the space required to store those different variants, 77% of that space is repetitive source. While these results can appear as not very encouraging in terms of scale, we should put these sizes in perspective with the general size of mobile applications, which can be larger than several megabytes.

When considering runtime performance, there are two principal areas that we can consider, resource manager initialisation, and GUI lookups. Resource management initialisation is carried out when the application starts, during which, the data structures containing the GUI variants are filled. How long this initialisation takes really depends on how many GUI documents are being managed, and how many variants are there of each GUI document being managed. The GUI lookup is purely the task of the GUI document reference being queried and looked up. To compare static selection of GUIs with the resource manager, we set up a micro-benchmark on a Samsung Nexus S running Android 4.2.1. When comparing lookup times from the resource manager to a purely static solution, we found it took on average 4 ms longer for the manager. At least in this simple case, this is close to negligible regarding our target use. It is foreseeable that it will be slower if there are more features that it needs to check against, but we do not expect more GUI documents to affect speed considerably. This is because for each managed GUI, a separate lookup method is used. When a resource is requested, depending on which GUI type e.g. home screen, will determine which lookup method to use. Since GUI references on Android are stored as Integers, we can carry this out within a simple switch.

## 2.9 Conclusions

The Graphical User Interface, like the rest of containing program can exhibit static variability realised at compile time [36, 37]. There also exists the need for runtime variability [14, 18, 23], however DSPLs traditionally consider only business logic changes. In this chapter, we presented an approach to supporting static and dynamic variability of GUI documents within features. This support gives the possibility to apply adaptation at different realisation times without the need for multiple implementations. When dynamic configuration is required, runtime management software is generated combined with code transformations using tool support implemented during validation.

While this approach currently has the likelihood to not scale to high levels of runtime variability, this solution introduces a step forward for mixed GUI variability in DSPLs for GUI documents. This approach therefore provides greater flexibility to the developer, by which it is not tied to single language solutions.

## References

1. Anfurrutia FI, Díaz O, Trujillo S. On refining XML artifacts. In: Proceedings of the 7th international conference on Web engineering, ICWE'07. Berlin/Heidelberg: Springer; 2007. p. 473–78.
2. Apel S, Janda F, Trujillo S, Kästner C. Model superimposition in software product lines. In: Proceedings of the 2nd international conference on theory and practice of model transformations, ICMT '09. Berlin/Heidelberg: Springer; 2009. p. 4–19.
3. Apel S, Kästner C. An overview of feature-oriented software development. *J Object Technol.* 2009;8(5):49–84.
4. Apel S, Kästner C, Lengauer C. Featurehouse: language-independent, automated software composition. In: Proceedings of the 31st international conference on software engineering, ICSE '09. Washington, DC: IEEE Computer Society; 2009. p. 221–31.
5. Apel S, Leich T, Rosenmüller M, Saake G. Featurec++: on the symbiosis of feature-oriented and aspect-oriented programming. In: Proceedings of the 4th international conference on generative programming and component engineering, GPCE'05. Berlin/Heidelberg: Springer; 2005. p. 125–40.
6. Apel S, Lengauer C. Superimposition: a language-independent approach to software composition. In: Proceedings of the 7th international conference on software composition, SC'08. Berlin/Heidelberg: Springer; 2008. p. 20–35.
7. Apel S, Lengauer C, Möller B, Kästner C. An algebra for features and feature composition. In: Meseguer J, Rosu G, editors. Algebraic methodology and software technology. Volume 5140 of lecture notes in computer science. Berlin/Heidelberg: Springer; 2008. p. 36–50.
8. Appeltauer M, Hirschfeld R, Masuhara H. Improving the development of context-dependent Java applications with contextj. In: International workshop on context-oriented programming, COP '09. New York: ACM; 2009. p. 5:1–5:5.
9. Batory D, Sarvela J, Rauschmayer A. Scaling step-wise refinement. *IEEE Trans Softw Eng.* 2004;30:355–71.
10. Behan M, Krejcar O. Adaptive graphical user interface solution for modern user devices. In: Pan J-S, Chen S-M, Nguyen N, editors. Intelligent information and database systems. Volume 7197 of lecture notes in computer science. Berlin/Heidelberg: Springer; 2012. p. 411–20.
11. Benavides D, Segura S, Ruiz-Cortés A. Automated analysis of feature models 20 years later: a literature review. *Inf Syst.* 2010;35:615–36.
12. Botterweck G. A model-driven approach to the engineering of multiple user interfaces. In: Proceedings of the 2006 international conference on models in software engineering, MoDELS'06. Berlin/Heidelberg: Springer; 2006. p. 106–15.
13. Boucher Q, Abbasi E, Hubaux A, Perrouin G, Acher M, Heymans P. Towards more reliable configurators: a re-engineering perspective. In: 2012 3rd international workshop on product line approaches in software engineering (PLEASE), 2012. p. 29–32.
14. Calvary G, Coutaz J, Thevenin D. A unifying reference framework for the development of plastic user interfaces. In: Little M, Nigay L, editors. Engineering for human-computer interaction. Volume 2254 of lecture notes in computer science. Berlin/Heidelberg: Springer; 2001. p. 173–92.
15. Clerckx T, Luyten K, Coninx K. DynaMo-AID: a design process and a runtime architecture for dynamic model-based user interface development. Berlin/Heidelberg: Springer; 2005. p. 77–95.
16. Collignon B, Vanderdonck J, Calvary G. Model-driven engineering of multi-target plastic user interfaces. In: Proceedings of the fourth international conference on autonomic and autonomous systems, ICAS '08. Washington, DC: IEEE Computer Society; 2008. p. 7–14.
17. Czarniecki K, Helsen S, Eisenecker UW. Staged configuration through specialization and multilevel configuration of feature models. *Softw Process: Improv Pract.* 2005;10(2):143–69.

18. David L, Endler M, Barbosa SDJ, Filho JV. Middleware support for context-aware mobile applications with adaptive multimodal user interfaces. In: Proceedings of the 2011 fourth international conference on Ubi-media computing, U-MEDIA '11. Washington, DC: IEEE Computer Society; 2011. p. 106–11.
19. Dittmar A, García Frey A, Dupuy-Chessa S. What can model-based UI design offer to end-user software engineering? In: Proceedings of the 4th ACM SIGCHI symposium on engineering interactive computing systems, EICS '12. New York: ACM; 2012. p. 189–94.
20. Draheim D, Lutteroth C, Weber G. Graphical user interfaces as documents. In: Proceedings of the 7th ACM SIGCHI New Zealand chapter's international conference on computer-human interaction: design centered HCI, CHINZ '06. New York: ACM; 2006. p. 67–74.
21. Feigenspan J, Kästner C, Frisch M, Dachselt R, Apel S. Visual support for understanding product lines. In: Proceedings of the 2010 IEEE 18th international conference on program comprehension, ICPC '10. Washington, DC: IEEE Computer Society; 2010. p. 34–35.
22. Grolaux D. Transparent migration and adaptation in a graphical user interface toolkit, PhD thesis, Université catholique de Louvain. 2007.
23. Hanumansetty RG. Model based approach for context aware and adaptive user interface generation, Master's thesis, Virginia Polytechnic Institute and State University. 2004.
24. Hauptmann B. Supporting derivation and customization of user interfaces in software product lines using the example of web applications, Master's thesis, University of Augsburg. 2010.
25. Hirschfeld R, Costanza P, Nierstrasz O. Context-oriented programming. *J Object Technol* Mar–Apr 2008, ETH Zurich 2008;7(3):125–51.
26. Holzinger A, Geier M, Germanakos P. On the development of smart adaptive user interfaces for mobile e-business applications – towards enhancing user experience – some lessons learned. In: DCNET/ICE-B/OPTICS. 2012. p. 205–14.
27. Kästner C, Apel S. Virtual separation of concerns – a second chance for preprocessors. *J Object Technol*. 2009;8(6):59–78.
28. Kästner C, Apel S, Kuhlemann M. A model of refactoring physically and virtually separated features. In: Proceedings of the eighth international conference on generative programming and component engineering, GPCE '09. New York: ACM; 2009. p. 157–66.
29. Kastner C, Thum T, Saake G, Feigenspan J, Leich T, Wielgorz F, Apel S. Featureide: a tool framework for feature-oriented software development. In: Proceedings of the 31st international conference on software engineering, ICSE '09. Washington, DC: IEEE Computer Society; 2009. p. 611–14.
30. Kim J, Lutteroth C. Multi-platform document-oriented guis. In: Proceedings of the tenth Australasian conference on user interfaces – volume 93, AUIC '09. Darlinghurst: Australian Computer Society, Inc.; 2009. p. 27–34.
31. Krasner GE, Pope ST. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J Object Oriented Program*. 1988;1(3):26–49.
32. Nicols J. Automatically generating high-quality user interfaces for appliances, PhD thesis, Camegie Mellon University. 2006.
33. Nielsen J. Usability testing of international interfaces. In: Nielsen J, editor. *Designing user interfaces for international use*. Essex: Elsevier Science Publishers Ltd.; 1990. p. 39–44.
34. Paskalev P. Rule based GUI modification and adaptation. In: Proceedings of the international conference on computer systems and technologies and workshop for PhD students in computing, CompSysTech '09. New York: ACM; 2009. p. 93:1–93:7.
35. Paskalev P, Nikolov V. Multi-platform, script-based user interface. In: Proceedings of the 5th international conference on computer systems and technologies, CompSysTech '04. New York: ACM; 2004. p. 1–6.
36. Pleuss A, Hauptmann B, Dhungana D, Botterweck G. User interface engineering for software product lines: the dilemma between automation and usability. In: Proceedings of the 4th ACM SIGCHI symposium on engineering interactive computing systems, EICS '12. New York: ACM; 2012. p. 25–34.
37. Pleuss A, Hauptmann B, Keunecke M Botterweck G. A case study on variability in user interfaces. In: Proceedings of the 16th international software product line conference – volume 1, SPLC '12. New York: ACM; 2012. p. 6–10.

38. Reichart D, Forbrig P Dittmar A. Task models as basis for requirements engineering and software execution. In: Proceedings of the 3rd annual conference on task models and diagrams, TAMODIA '04. New York: ACM; 2004. p. 51–8.
39. Rosenmüller M, Siegmund N, Apel S, Saake G. Flexible feature binding in software product lines. *Autom Softw Eng.* 2011;18(2):163–97.
40. Russo P, Boor S. How fluent is your interface?: designing for international users. In: Proceedings of the INTERACT '93 and CHI '93 conference on human factors in computing systems, CHI '93. New York: ACM; 1993. p. 342–47.
41. Savidis A, Stephanidis C. Software refactoring process for adaptive user-interface composition. In: Proceedings of the 2nd ACM SIGCHI symposium on engineering interactive computing systems, EICS '10. New York: ACM; 2010. p. 19–28.
42. Schlee M. Generative programming of graphical user interfaces, Master's thesis, University of Applied Sciences of Kaiserslautern. 2002.
43. Sottet J-S, Ganneau V, Calvary G, Coutaz J, Demeure A, Favre J-M, Demumieux R. Model-driven adaptation for plastic user interfaces. In: Proceedings of the 11th IFIP TC 13 international conference on human-computer interaction, INTERACT'07. Berlin/Heidelberg: Springer; 2007. p. 397–410.
44. Thüm T. Reasoning about feature model edits, Master's thesis, Otto-von-Guericke-University Magdeburg. 2008.
45. Vanderdonckt J, Calvary G, Coutaz J, Stanculescu A. Multimodality for plastic user interfaces: models, methods, and principles. In: Tzovaras D, editor. Multimodal user interfaces, signals and communication technologies. Berlin/Heidelberg: Springer; 2008. p. 61–84.
46. Voelter M, Groher I. Product line implementation using aspect-oriented and model-driven software development. In: Proceedings of the 11th international software product line conference, SPLC '07. Washington, DC: IEEE Computer Society; 2007. p. 233–42.



# Chapter 3

## Variability Management and Assessment for User Interface Design

**Jabier Martinez, Jean-Sébastien Sottet, Alfonso García Frey, Tewfik Ziadi,  
Tegawendé Bissyandé, Jean Vanderdonckt, Jacques Klein, and Yves Le Traon**

**Abstract** User Interface (UI) design remains an open, wicked, complex and multi-faceted problem, owing to the ever increasing variability of design options resulting from multiple contexts of use, i.e., various end-users, heterogeneous devices and computing platforms, as well as their varying environments. Designing multiple UIs for multiple contexts of use inevitably requires an ever growing amount of time and resources that not all organizations are able to afford. Moreover, UI design choices stand on end-users' needs elicitation, which are recognized to be difficult to evaluate precisely upfront and which require iterative design cycles. All this complex variability should be managed efficiently to maintain time and resources to an acceptable level. To address these challenges, this article proposes a variability management approach integrated into a UI rapid prototyping process, which involves the combination of Model-Driven Engineering, Software Product Lines and Interactive Genetic Algorithms.

---

J. Martinez (✉) • T. Ziadi  
Sorbonne University, UPMC Univ Paris 06, CNRS, Paris, France  
e-mail: [jabier.martinez@lip6.fr](mailto:jabier.martinez@lip6.fr); [tewfik.ziadi@lip6.fr](mailto:tewfik.ziadi@lip6.fr)

J.-S. Sottet  
Luxembourg Institute of Science and Technology (LIST), 5 Avenue des Hauts-Fourneaux,  
Esch/Alzette, Luxembourg  
e-mail: [jean-sebastien.sottet@list.lu](mailto:jean-sebastien.sottet@list.lu)

A.G. Frey  
Yotako S.A., 9 Avenue des Hauts-Fourneaux, Esch/Alzette, Luxembourg  
e-mail: [alfonso@yotako.io](mailto:alfonso@yotako.io)

T. Bissyandé • J. Klein • Y. Le Traon  
Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg,  
Luxembourg City, Luxembourg  
e-mail: [tegawende.bissyande@uni.lu](mailto:tegawende.bissyande@uni.lu); [jacques.klein@uni.lu](mailto:jacques.klein@uni.lu); [yves.letraon@uni.lu](mailto:yves.letraon@uni.lu)

J. Vanderdonckt  
Louvain School of Management, Université catholique de Louvain, Place des Doyens, 1, 1348,  
Louvain-la-Neuve, Belgique  
e-mail: [jean.vanderdonckt@uclouvain.be](mailto:jean.vanderdonckt@uclouvain.be)

### 3.1 Introduction

The development life cycle of User Interfaces (UIs) often requires producing several versions of the same UI intended for different targets. This development may span from different versions targeting different contexts of use to a single version exhibiting a multi-layer UI [39] offering different levels of sophistication depending on the end-user's expertise. Interaction design is understood as a complex and multi-faceted problem [8] that is always open [45] (new end-user requirements may appear while designing), iterative (there is no optimal solution at first glance), and intrinsically incomplete (not all end-user requirements are elicited from the beginning). When designing interaction, variability is manifold depending on the context of use [10]: variability of end-users, of their devices, computing platforms, and of environments in which they are working. Moreover, end-user requirements are difficult to evaluate precisely upfront in UI design processes. Therefore, the main UI design processes, such as User-Centred Design [15], implement an iterative design cycle in which a UI variant is produced, tested on end-users, and their feedback is integrated into design artifacts (e.g., part of the UI, requirements, etc.). Since these processes are mostly based on trial and error, some parts of the UI have to be re-developed many times to fulfill all the different user requirements. When different UIs are produced for various contexts of use, their respective usability to be achieved is also varying, thus posing the problem of multi-target usability evaluation [1]. Moreover, these UI design processes involve multiple stakeholders playing different roles (e.g., software developers, UI/User eXperience designers, business analysts, end-users) that demand a great amount of time to reach consensus. UI variability has thus a significant impact on the design, development, and maintenance costs of the UI, thus redistributing the total cost of ownership of a UI depending on these parameters.

To overcome variability issues in software engineering, researchers have successfully relied on the paradigm of Software Product Lines (SPLs) [12]. The SPL paradigm allows to manage variability by producing a family of related product configurations (thus leading to product variants) for a given domain. Indeed, the SPL paradigm proposes the identification of common and variable sets of features, to foster software reuse in the configuration of new products [34].

Model-Driven Engineering (MDE) has already been used to effectively support the UI design process [40]. According to [5, 13], SPL and MDE are complementary and can be combined in a unified process that aggregates the advantages of both methods without suffering too much from their respective shortcomings.

A MDE-SPL approach could produce rapidly many variants of the same UI. However, some of these variants could not satisfy their respective level of usability: a same UI deployed on different computing platforms could have different levels of usability [1]. Testing a very large collection of similar variants could be exhausting for the end-users and may lead to inappropriate results. As such, an innovative way to evaluate only a relevant portion of the variant needs to be defined. This solution could provide a reasonably good (but not necessarily optimal) variant that satisfy the end-user requirements elicited for a set of contexts of use.

This article proposes an approach to manage UI variability jointly based on MDE and SPL. Our approach relies on model transformations that support the expression of the variability. This approach enables the separation of concerns of the different stakeholders when expressing the UI variability and their design choices (UI configurations). A Multiple Feature Model approach is considered in which each feature model represents a particular concern allowing, if needed, each stakeholder to work independently. A partial and staged configuration process [14] is proposed in which a partial UI configuration is produced that can be refined by all stakeholders including the end-user feedback. Once the UI is produced, an innovative mean for evaluating the variants with end-users allowing to select the best products is presented. These concepts will be exemplified with a concrete example of UI variability.

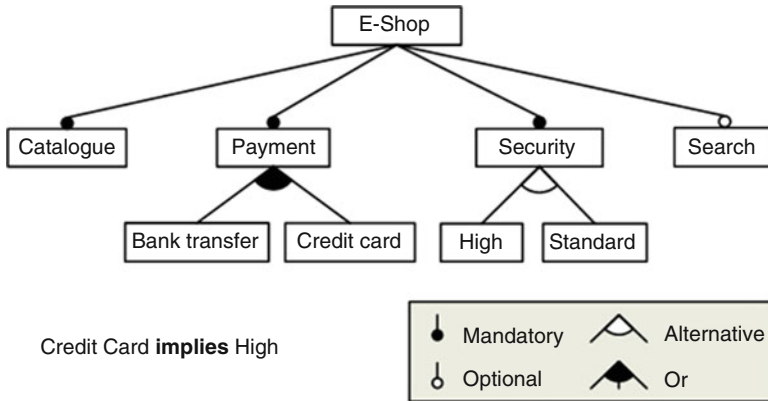
The chapter is structured as follows: First, in Sect. 3.2, we introduce the related work on the three domains: feature modelling and configuration, user interface variability and variant testing. Next, we introduce our UI-SPL approach (Sect. 3.3) that help in deriving many products from initial design models and variability models. Then, in Sect. 3.4, we explain the assessment of the produced UI variants using a genetic approach. Before concluding the chapter, we illustrate (Sect. 3.5) and evaluate (Sect. 3.6) our approach based on a case study: evaluating the UI variants of a contact list.

## 3.2 Related Work

### 3.2.1 Feature Modelling

Feature models (FM) [34] are popular SPL assets that describe both variability and commonalities of a system. They express, through some defined operators, the decomposition of a product related features. The feature diagram notation used in this article is explained in Fig. 3.1. The E-Shop FM consists of a mandatory feature “catalogue”, two possible payment methods from which one or both could be selected, an exclusive alternative of security levels and an optional search feature. FM constraints can be defined. In this case “credit card” implies a high level of security.

Features composing a FM depict different parts of a system without any clear separation of concerns. The absence of feature types makes these models popular as there are no limits for the expression of design artifacts. But at the same time, [9] have demonstrated that depicting information in a single FM leads to feature redundancies due to the tree structure. As a result, separation of variability concerns into multiple FMs seems to be crucial for understanding [28] and manipulating [2] the many different faces of variability. Each of these FM focuses on a viewpoint on variability which makes easier to handle variability for each stakeholder. An early example of this approach is MiniAba [38], which models UI adaptation based on



**Fig. 3.1** Feature model from an E-shop (Source: Wikimedia commons)

a FM: the complete UI configuration is represented by a FM tree, whose branches and/or leaves may be edited or removed in order to produce an instantiated FM. This FM then gives rise to a project configuration that will automatically generate a corresponding adapted UI. The big win is that, when one changes any feature in the FM, it is automatically reflected in the generated UI after re-compiling.

### 3.2.2 SPL Configuration

The configuration process is an important task of SPL management: producing a particular product variant based on a selection of features to fit the end-users' requirements. In this context, a configuration is hereby defined as a specific combination of FM features satisfying all the FM constraints. When designers and developers configure a system according to requirements, the enforcement of FM constraints can limit them in their design choices [48]. Moreover, the separation of the variability in multiple FMs is also a source of complexity due to many dependencies across FMs. The fusion of all FMs into one for configuration purposes seems to solve this issue but results in a large FM that mixes different facets: this may lead to invalid configurations and thus invalid or inefficient products. Some solutions exist to overcome these problems. The work by [35] proposes an implementation of a configuration composition system defining a step-by-step configuration [14] using partial configurations [7]. Thus, some portion of a FM can be configured independently, without considering all the constraints (coming from other configurations) at configuration time. Then, constraints amongst configurations may be solved by implementing consistency transformations [2].

### 3.2.3 *Model-Driven User Interfaces Variability*

Model-Driven UI calls for specific models and abstraction. These models address the flow of UIs, which could range from a mono-path flow to multi-path workflow [19], the domain elements manipulated during the interaction, the models of expected UI quality, the layout, the graphical rendering, etc. In addition, each model corresponds to a standard level of abstraction as identified in the CAMELEON Reference Framework (CRF) [10]. The CRF aims at providing a unified view on modelling and adaptation of UIs. In the CRF, each level of abstraction is a potential source of variability. Modifying any model fragment at any specific level of abstraction induces a specific adaptation of the UI.

For instance, the task model, considered in CRF as the topmost model, depicts the interaction between the user and the features offered by the software in a way that is computing-independent. Adding or removing a task results in modifying the software features. Considering this, we can assume that there is a direct link between classical feature modelling and task modelling such as presented in [33]. In this work, a task model is derived from an initial FM. However the authors did not go any further in describing the variability related to interaction and UI (e.g., graphical components, behavior).

In [18], the authors present an integrated vision of functional and interaction concerns into a single FM. This approach is certainly going a step further by representing variability at the different abstraction levels of the CRF. However, this approach has several drawbacks. On the one hand, this approach derives functional variability only from the task model, limiting the functional variability of the software. On the other hand, all the UI variations are mixed into a single all encompassing FM which blurs the various aspects for comprehension and configuration [28].

Finally, Martinez et al. [29] presented a preliminary experience on the usage of multiple FMs for web systems. This work showed the feasibility of using multiple FMs and the possibility to define a process around it. It implements FMs for a web system, interaction scenario, a user model (user impairments), and device. However, this approach does not consider the peculiarities of UI design models and their variability.

A few works in SPL for UI have been published. A large part is dedicated to the main variability depiction (using FMs) but they do not directly address the configuration management. Configuration is a particular issue when considering end-user related requirements which may be fuzzily defined.

### 3.2.4 *Testing Many Variants*

Selecting optimal SPL product variants based on some criteria has been studied in SPL Engineering (SPLE). To achieve this, it is a common practice to enhance the

variability model by what is referred to as quality attributes [6, 11, 23, 25]. Assessing the quality of produced variants requires to deal with a potential large set of similar UI to be evaluated and with subjectivity (i.e., considering users feedback could also encompass some aesthetic aspects).

Genetic algorithms have been used for guiding the analysis of configuration space [17, 36]. A key operator for evolutionary genetic algorithms is the *fitness function* representing the requirements to adapt to. In other words, it forms the basis for selection and it defines what improvement means [16]. In our case, the fitness function is based on user feedback which is a manual process in opposition to automatically calculated fitness functions (e.g., the sum of the cost of the features). Because we deal with these user assessments as part of the search in the configuration space, we leverage Interactive Genetic Algorithms (IGA) where humans are responsible to interactively set the fitness [16, 47].

### 3.3 UI-SPL Approach

Model-driven UI design is a multi-stakeholder process [21, 22] where each model – representing a particular sub-domain of UI engineering – is manipulated by specific stakeholders. For instance, the choice of graphical widgets to be used (e.g. radio button, drop-down list, etc.) is done by a graphical designer, sometimes in collaboration with the usability expert and/or the client. Our model-driven UI design approach [40] relies on a revised version of the CRF framework (Fig. 3.2). It consists of two base meta-models, the Domain meta-model -representing the domain elements manipulated by the application as provided by classical domain analyst- and the Interaction Flow Model (IFM) [31]. From these models we derive the Concrete User Interface model (CUI) which depicts the application “pages” and their content (i.e., widgets) as well as the navigation between pages. The CUI meta-model aims at being independent of the final implementation of any graphical element. Finally, the obtained CUI model is transformed into an Implementation Specific Model (ISM) that takes into account platform details (here platform refers to UI tool-kits such as HTML/jQuery, Android GUI, etc.). Finally, a Model-to-Text (M2T) transformation generates the code according to the ISM. This separation allows for separate evolution of CUI metamodel and implementation specific metamodel and code generation.

We propose a multiple feature models (multi-FM) approach (see Sect. 3.3.1) to describe the various facets of UI variability (e.g., UI layout, graphical elements, etc.). In a second phase, (see Sect. 3.4) we introduce our specific view on configuration on this multi-FM and its implementation in our model-driven UI design approach (see Sect. 3.3.2).

**Fig. 3.2** Model-driven UI design process



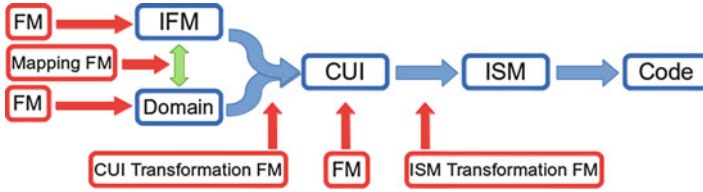


Fig. 3.3 Variability models (FM) coverage on our UI modeling framework

### 3.3.1 Multi-FM Approach

Classical FM approaches combine different functional features [33]. In the specific context of UI design, we propose to rely on a similar approach for managing variability of each UI design concern. UI variability is thus decomposed into FMs (Fig. 3.3). Each of these FMs is related either to a model, a meta-model, a mapping or a transformation depending on the nature of the information it conveys.

- *Models*: Three FMs in Fig. 3.3 manage variations at the model level (IFM, Domain and CUI). The FM responsible for the Domain configuration can be used to express alternatives of a same concept, e.g., using or not the address, age or photo of a given class “Person”. The IFM variability can express for instance the possible navigation alternatives to be activated or not. For instance on a shopping website, shortcuts providing quickly access to a given product can be configured. The CUI FM represents alternative representations of a widget: a panel (i.e., portion of UI displayed on a screen) can become a full window (i.e., displayed has full-screen) on mobile phones.
- *Mappings*: The variability of the mapping between IFM and Domain can be managed by a mapping FM. Interaction flow elements (UI states) can involve different concepts of the domain model.
- *Transformations*: Variability can be expressed also at the level of transformations. The variability that a transformation could convey (i.e., multiple output alternatives) can be expressed with FMs. The transformations impacted are (1) between IFM, Domain and CUI, (2) between CUI and ISM. The variability of (1) expresses the possible UI design choices: how an IFM state selection can be represented: a simple list, an indexed list, a tile list, etc. The variability in (2) depends on the target ISM and configures the final representation to be provided to end-users. For instance, a CUI simple list can be represented using, as output of the transformation, the following HTML markup alternatives: “<select>” or “<ul>”.

By scoping the FM to a specific concern, our approach allows to focus only on the variations related to the underlying concern. In our approach, the different FMs enrich the existing UI design process accompanying, step-by-step, the design of models and their variability.

### 3.3.2 Implementation: Model Transformation

Before starting the whole transformation process (up to the first executable prototype), the various FMs that describe the possible configurations of the product line have to be aggregated. More details about this step is available in [43]. As a result, only one large FM is built allowing to produce, through model to model and model to text transformations the variants to be assessed. To aggregate the many FM, we can use the insert operator of Familiar [3] using the following expression see Listing 3.1.

**Listing 3.1** Familiar insertion operator for building complete partial configuration

```
fml> insert MappingIFMDomain into CUITransformation.CUIList with mand
```

The implementation of our approach relies on an existing system that derives a UI from IFM and domain models using successive model transformations [41]. This initial system was not taking into account the configuration of the variability. The difference is that it uses the FM as a transformation configuration and generate a UI for each possible configurations depicted in the aggregate FM. We have built a small algorithm that produce a configuration model for each valid combination of the FM. Then this configuration model (conforms to a configuration metamodel) is used as an additional input model for the transformation. Nevertheless, we keep our tool initial behavior if no FM is specified: a default transformation is executed if it has no configuration (in ATL syntax: `i.getConfiguration.oclIsUndefined()`). The “getConfiguration” helper uses the explicit link between the input models (IFM/Domain) and the current configuration to be generated.

**Listing 3.2** Excerpt of the default transformation used if no configuration is defined

```
rule selectionListViewDefault extends widgetEvents {
from
  i : SC! SelectionState ( i . getConfiguration . oclIsUndefined () )
to
  o : CUI! ListView (
    name <- i . name ,
    id <- i . name . regexReplaceAll ( ' ' , '' ) ,
    widgets <- i . domainElements -> select ( e | e . Type = #Image ) -> collect ( e |
      ↪ thisModule . image ( e ) )
  )
}
```

We reused the rest of the transformation chain up to the application generation: CUI to ISM and ISM to Code. For each type of ISM (i.e., interaction State) a set of rules are produced corresponding to the possible variants. Each particular attribute of the widget (i.e., indexed and filtered) is also dependent on the configuration thus introducing additional conditional expressions (ListFilters and ListDividers conditions in Listing 3.3).

**Listing 3.3** Excerpt of selection to tile list in CUI transformation including configuration helpers

```
helper context OclAny def: hasConfig ( config : String ) : Boolean =
if ( self . getConfiguration . oclIsUndefined () )
```



```

    then
      false
    else
      self.getConfiguration.WidgetName=config
  endif;
  ...
  rule selectionTileList extends widgetEvents {
  from
    i : SC! SelectionState(i.hasConfig('TileList'))
  using {
    conf: Configuration! Configuration = i.getConfiguration;
  }
  to
    o : CUI! TileList (
      name <- i.name,
      id <- i.name.regexReplaceAll(' ', ''),
      icons <- i.domainElements->select(e| e.Type = #Image)->collect(e|thisModule
        =>.image(e)),
      Listfilters <- if(conf.filtered) then filter else OclUndefined endif,
      listDivider <- if(conf.indexed) then divider else OclUndefined endif
    ),
    filter : CUI! Filter(
      filterRevealedList <- false),
    divider : CUI! Divider(
      autodivider <- true )
  }

```

### 3.4 Evaluation of Configuration: Rapid Prototyping

End-user requirements are crucial in user centered design. They are often not formally defined: most of the time they are expressed as remarks on a portion of the produced or prototyped UI. Thus, in order to capture end-user requirements, UI designers have to propose various product versions (prototypes) to end-users. A common practice is to use rapid prototyping. Rapid prototyping is a user-centered iterative process where end-users give feedback on each produced prototype. Prototypes are usually mock-ups of UI drawn with dedicated tools (e.g., see balsamiq).<sup>1</sup> In order to show to the users an interaction experience closest to reality we should rely on higher fidelity and on living prototypes (i.e., the user should be able to interact with it). As a result, MDE provides us with semi-automatic generation capabilities. It allows a quick production of prototypes and many assessments in a limited amount of time. End-users will thus elicit the way they prefer interacting with the system, the best widgets and representations for their tasks. In fact, through these iterations they elicit the product configuration that best fits their needs.

In previous work [40, 42], we have tested the global usability of particular generated UI prototypes. We have also proposed a version that evaluate only a portion of the UIs produced by an SPL [43]. Configuration reconciliation can be a time consuming task, which may delay the product elicitation. Indeed the time for configuring and aligning all the partial configurations together can be a very consuming task even if we relax some FM constraints (i.e., getting an end-user feedback on generated UI after a one or two minutes configuration).

---

<sup>1</sup><http://balsamiq.com>

Here we will try to evaluate the different UIs obtained by an SPL. The approach we have selected is an interactive genetic algorithm (IGA). It aims at reducing drastically the number of UI configurations to evaluate as well as the number of person necessary for this test. The idea is to converge quickly to good/acceptable UIs with a small number of algorithm iterations: limiting the effect of user fatigue. At each iteration the IGA will take into account a user ranking (e.g., like/dislike) of the proposed UI and propose, a mutated configuration for the next step. This will allow to test potentially many configurations whilst keeping only the convenient ones.

The approach we have followed here is decomposed in two steps. The first step consists in generating the set of UIs according to the possible configurations depicted in the FM. For the sake of performance (i.e., rapid evaluation between each IGA iteration) and because of interoperability issues we choose to produce first the set of evaluable UIs. Nevertheless the MDE-SPL generation mechanism could have been directly driven by the IGA produced configurations. The second step is iterative and consists in the evaluation of the produced UI variant and then production of next UIs to be evaluated thanks to the IGA.

### ***3.4.1 Step 1: Deriving All Relevant Configurations***

The idea is to generate all the possible configurations and then apply the UI generation process for each of them, as depicted in Sect. 3.3.2. Some constraints can also be added to existing ones [4]. They allow to remove product variants which are, predictably, not relevant for the task at hand. For instance, tile lists are not necessarily relevant without photos or images.

### ***3.4.2 Step 2: Variant Assessment***

We followed the recommendations of Nielsen regarding the minimal number of end users to involve in iterative user tests [30]. As such, we decided to run user tests with 5 participants in order to collect their feedback. For each iteration of the genetic algorithm, 10 UI configurations are suggested to testing (i.e., 2 per end-user). According to the predictions of the Poisson model [30], involving 5 users gives rise to an expected probability of reporting 85% of the usability problems for a simple UI. Given that usability problems impact their assessments, having at least 5 users should enable us to gracefully evolve to a reasonable good UIs at a reasonable cost regarding the number of involved end-users. The IGA is taking this user evaluation to propose, to all users, for the second round of evaluation a mutation of the best variants. A mutant is a new configuration that slightly different from the previous one, by e.g., changing one feature (add/remove).

The details of the implemented IGA is shown in Algorithm 1. First, the population is randomly initialized at line 1. After this, the evolution starts at line

---

**Algorithm 1** Interactive genetic algorithm for data set creation in the contact list case study

---

**input:** Genetic representation of a configuration = 20 bits, Population = 10 configurations, Users = 5

**output:** Data set of user assessments

- 1: population  $\leftarrow$  initializePopulation()
- 2: **while** stopConditionNotSatisfied() **do**
- 3:     **foreach**  $conf_i \in$  population **do**
- 4:          $conf_i.assignedUser \leftarrow$  assignUser(users,  $conf_i$ )
- 5:     **end foreach**
- 6:     **foreach**  $conf_i \in$  population, **in parallel do**
- 7:          $conf_i.fitness \leftarrow$  getUserFeedback( $conf_i$ )
- 8:         registerDataInstance( $conf_i$ )
- 9:     **end foreach**
- 10:     parents  $\leftarrow$  parentSelection(population)
- 11:     offspring  $\leftarrow$  crossover(parents)
- 12:     offspring  $\leftarrow$  repair(offspring)
- 13:     offspring  $\leftarrow$  mutate(offspring)
- 14:     offspring  $\leftarrow$  repair(offspring)
- 15:     population  $\leftarrow$  survivorSelection(offspring)
- 16: **end while**

---

2 until the stop condition is satisfied. In our case we used the termination condition when reaching a fixed number of generations in order to avoid user fatigue and time consumption. We can also limit the number of generations (iteration of the IGA): it limits the number of evaluations (against user-fatigue), but also stop before the evaluation curve is reverted (i.e., when producing mutant the quality of UI could regress because of user fatigue).

From line 3 to 5, each member of the population is assigned to one user for assessment. In our experimental settings, each of the 5 users is assigned to 2 members to cover the whole population. The user feedback for all the population is obtained from line 6 to 9. Once the fitness of the whole population is set, we can proceed to the parent selection for the next generation. The *parent selection operator* is based on a fitness proportionate selection (line 10). At line 11, the *crossover operator* is based on the half uniform crossover scheme [46]. The crossover, as well as the mutation operators of the GA, can end up with invalid configurations because of FM constraints. Existing works have solved this by penalizing the fitness function or trying to recover the configuration to a valid state. In our case, at line 12 and 14 we repair the offspring if hard constraints are violated. The *mutation operator* used at line 13 is uniform with  $p = 0.1$ . This mutation factor is meant to prevent a loss of motivation from users (i.e., user fatigue) by reducing the likelihood that they will keep assessing very similar UI configurations from the population, while thus enabling us to explore new regions of the configuration space. Finally, at line 15, the *survivor selection operator* is based on a complete replacement of the previous generation with the new generation.

## 3.5 Case Study

Contact Lists are widely used Human-Computer Interaction (HCI) applications to obtain personal information such as telephone numbers or email addresses. We can find them on mobile phones for personal use, communication systems for elderly people, corporate intranets or web sites. Despite of sharing the same objective, the final UI implementations are very diverse. We focus in this case study in building contact lists with corporate information.

User involvement is one of the key principles of the user-centered design method [15]. The literature includes a number of early works leveraging SPL techniques to deal with the management of HCI-specific variability using SPL-based approaches [32, 33]. In this case study dealing with UIs, we focus on early binding variability [26] where design alternatives are chosen at design-time. Therefore, this scenario does not focus on customization options which are performed by end-users themselves within the UI, nor to self-adaptive systems which corresponds to run-time binding of variability. The ISATINE framework [27] structures the adaptation life cycle into seven stages: goal specification (when the end-user defines her goals for adaptation), adaptation initiative (who is taking the responsibility to trigger the adaptation), adaptation specification (what could be performed to ensure an adequate adaptation), adaptation application (performed by the system itself), adaptation transition (how to convey the transition between the stage before adaptation and after), adaptation interpretation (how to interpret the results of the adaptation), and adaptation evaluation (how to evaluate the results of the adaptation with respect to the initial goals). If these seven stages are considered, the adaptation specification and application are particularly ensured by our approach. The transition should be ensured by other techniques.

### 3.5.1 *The Contact List Example*

Figure 3.4 presents the FM defining the variability of the Contact List application domain. The FM was created by HCI experts from the Luxembourg Institute of Science and Technology (LIST) with whom we collaborated in this case study. This FM encodes knowledge about the interface design defining a configuration space of 1365 valid configurations. UI design choices, even for this apparently simple case, give raise to voluminous configuration spaces.

The `ContactList` variability is decomposed into three main features: `List` depicting the possible choices to be made in terms of widgets for representing the list, `Master Detail Interface` which states the global layout of the application and `Details Grid` which sets the layout for the detailed information of a person. The `ListType` variability defines the different alternatives of `List` widgets: `DropDownList` is a select box showing only one item when inactive, `ListView` is a classic navigation list and `TileList` is a list of thumbnails represented as tiles. The `Indexed` optional feature separates and ranks the list

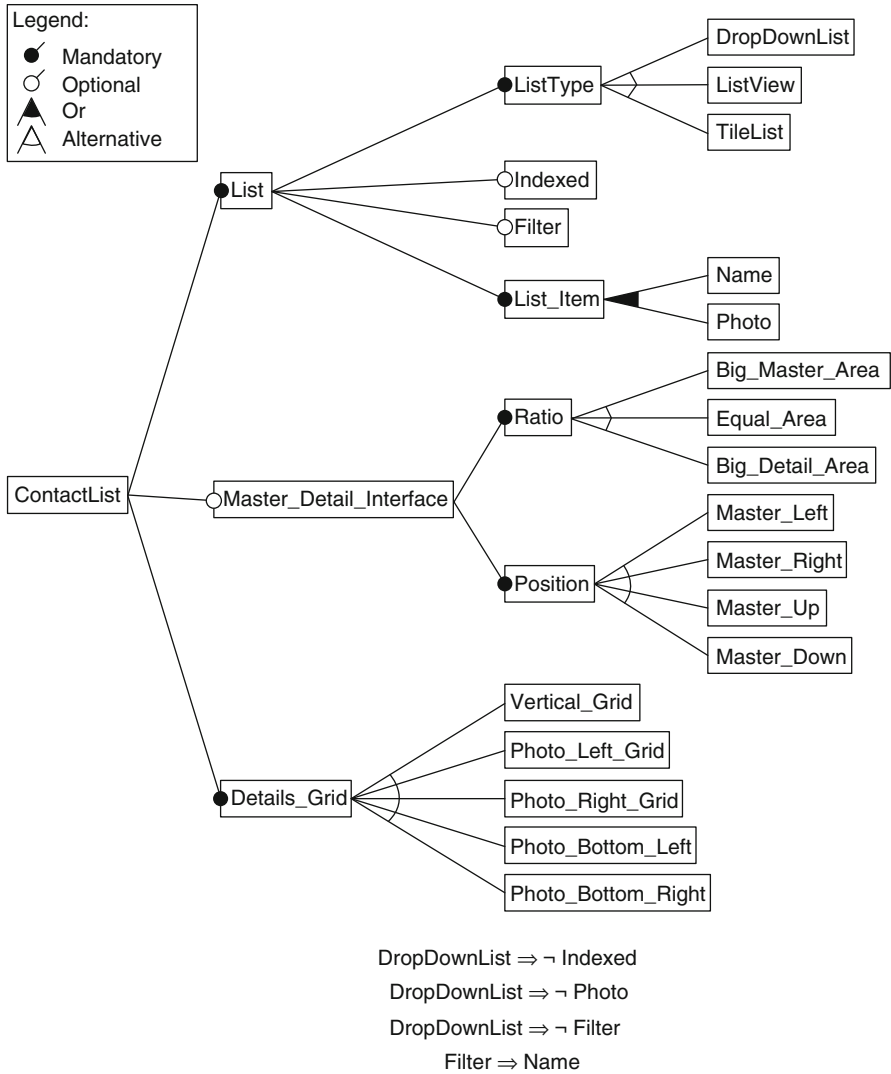


Fig. 3.4 Contact list feature model

items by the first letter of the name. The *Filter* optional feature adds a search functionality implemented through a text box that automatically filters the list items according to the text introduced by the user. The *ListItem* consists of the *Name* of the person or the *Photo*, or both. Four cross-tree constraints are shown in the feature diagram which are related to these features. Concretely, the *DropDownList* feature excludes *Indexed*, *Photo* and *Filter* features. Also, the *Filter* feature requires the *Name* in the *ListItem* feature.

The Master Detail Interface is an optional feature that split the screen in two parts: the master and the detail. The master contains the list while the details interface, after a selection in the master, shows the corresponding contact information. There is variability concerning the Ratio of the screen split and the Position of the master interface. Finally, the Details Grid feature represents different alternatives to organize the contact information on the screen (e.g., telephone number, address etc.) as for example including all information in one column or determine the position of the textual information with respect to the photo.

The SPL has been implemented using the Variability-aware Model-Driven UI design framework [44] based on AME (Adaptive Modeling Environment) [20] which is able to derive, through source code generation, any configuration of the presented FM. The target framework for the derived products is the JQueryMobile web framework.<sup>2</sup>

Screenshots demonstrate the diversity of UIs that can be obtained (they have been anonymized to avoid displaying personal information): Figs. 3.5 and 3.6 present UI variants from which we enumerate their corresponding features. Figure 3.5 shows an example that includes ListView with only the Name (see left side of the figure). The list view is neither Indexed nor has a Filter feature. It has Master Detail Interface with Equal Area and Master Left given that the screen is split in two identical parts with the list (master) at the left and the details grid at the right. The details are displayed with Photo Right Grid.

Figure 3.6a, b show how the TileList is realized (see right side of the figure) and Fig. 3.6c how the DropDownList is displayed (see left side of the figure). Figure 3.6d shows a UI variant whose configuration does not have a master detail. It only displays on the screen either master or detail (note the presence of the back button at the top left of the screenshot for coming back to the master). In the case of master detail, the ratio indicates whether we have a big master interface with a small details interface (e.g., Fig. 3.6c) or vice-versa. Alternatively, we can have the split into two equal parts (Figs. 3.5 and 3.6a).

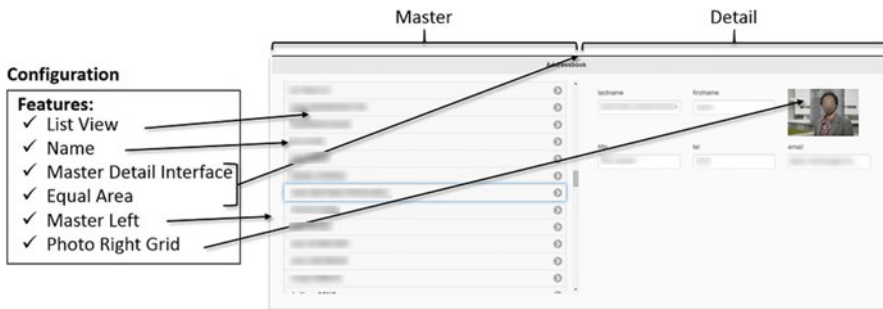
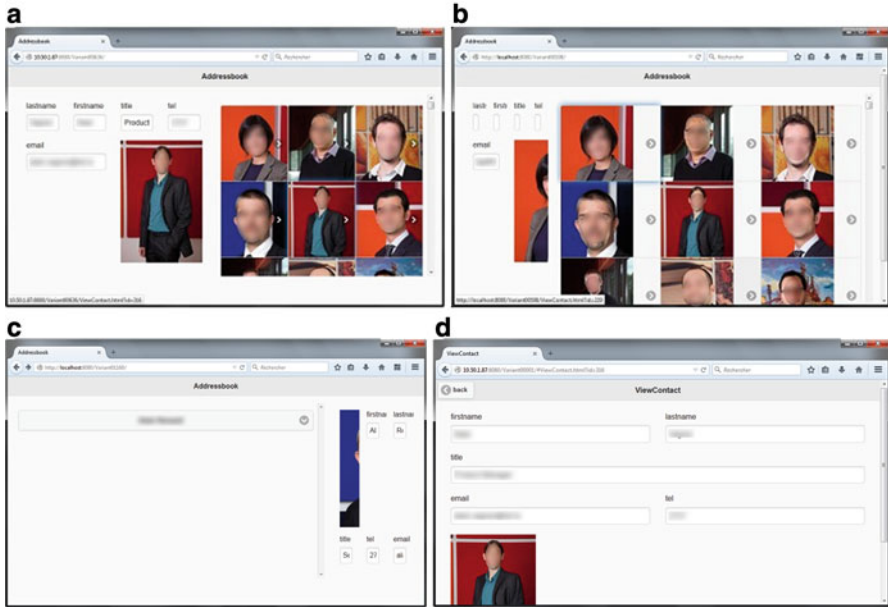


Fig. 3.5 Configuration and screenshot of its associated contact list UI variant

<sup>2</sup>JQueryMobile web framework: <https://jquerymobile.com>



**Fig. 3.6** Screenshots of derived variants from the contact list SPL and enumeration of their associated features. (a) Tile List, Photo, Master Detail (with Equal Area and Master Right) and Details Grid with Photo Bottom Right. (b) Tile List, Photo, Master Detail (with Big Master Area, Master Right). (c) DropDownList, Name, Master Detail (with Big Master Area, Master Left). (d) Master Detail Feature (The list variability is not illustrated in this figure) and Details Grid with Photo Bottom Right

The `Position` variability is related to a horizontal or vertical split of the screen and whether the master is in one side or the other (in Fig. 3.6b the master and detail have been swapped). If the `Master Detail Interface` feature is not selected in a configuration, the window split will be replaced during the navigation: one first window for selecting the person to be displayed and the other one for seeing the details. Finally, the `Details Grid` feature represents different alternatives to organize the contact information on the screen. For instance, in Fig. 3.6a the grid is four columns and two rows whereas in Fig. 3.6d it is two columns and four rows.

### 3.5.2 Defining Configurations Chromosome

One important decision to implement the genetic algorithm is how to represent the individuals (the configurations in our case). We consider a binary array. Figure 3.7 shows an example of the chromosome of an individual that conforms to the Contact List SPL. The phenotype consists of the non-abstract features of the FM. Concretely, the leaves of the FM and the `Master Detail Interface` feature (see Fig. 3.4) are coded on a binary string of 20 bits. The features are the fixed indexes of the array

**Fig. 3.7** Example chromosome of an individual of the contact list SPL

DropDownList	0
ListView	1
TileList	0
Indexed	0
Filter	1
Name	1
Photo	1
MasterDetailInterface	1
BigMasterArea	0
EqualArea	1
BigDetailArea	0
MasterLeft	1
MasterRight	0
MasterUp	0
MasterDown	0
VerticalGrid	0
PhotoLeftGrid	0
PhotoRightGrid	1
PhotoBottomLeft	0
PhotoBottomRight	0

where the value 1 means that the feature is activated and 0 that it is not. Representing a FM configuration chromosome as an array of bits is a common practice in the use of genetic algorithms in SPLE [17, 24]. As presented before, the details of the implemented IGA are shown in Algorithm 1.

### 3.6 Case Study: Experimentation

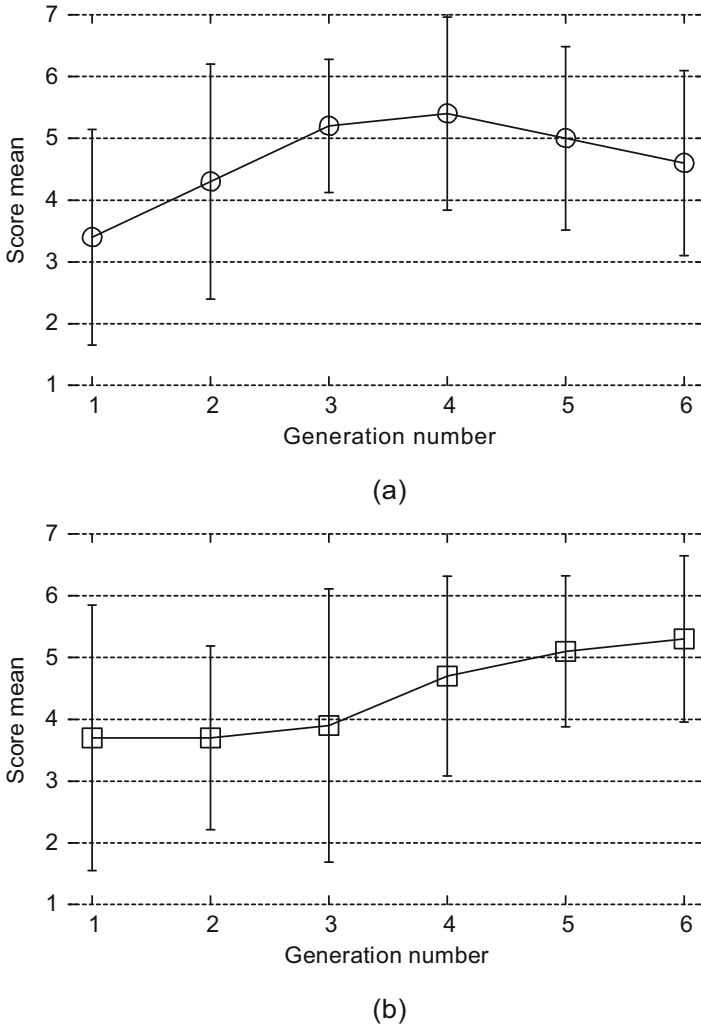
The objectives in this case study is to investigate whether:

1. The variant assessment selects better configurations than a randomized algorithm for a given number of iterations. To that end:
  - We quantitatively evaluated the improvement of the user scores through the IGA compared to random selection within the configuration space.
  - We study the diversity of the population along the generations to show the convergence of the IGA towards relevant UI designs.
2. The best UIs are configurations that usability experts confirm as relevant UI designs. We qualitatively discuss the findings with a usability expert and we checked if the top ranked configurations are close to configurations elicited by usability experts.

We conducted two independent experiments in two organizations: LIST and SnT. Figure 3.8 presents the results of the IGA for the two organizations. On the horizontal axis we have the different generations and the vertical axis is the mean of the scores of the user assessments for this generation. We show the score mean along the six generations including the standard deviations. An ascendant progression means that for each new generation, globally, the UI variants are being better appreciated by the pool of users. In Fig. 3.8a we observe a quick ascension until generation four while in Fig. 3.8a we observe the ascendant progression starting at generation two.

Despite that we do not have the explanation for the descending effect in LIST case for generations five and six, we consider that it is caused by the experience

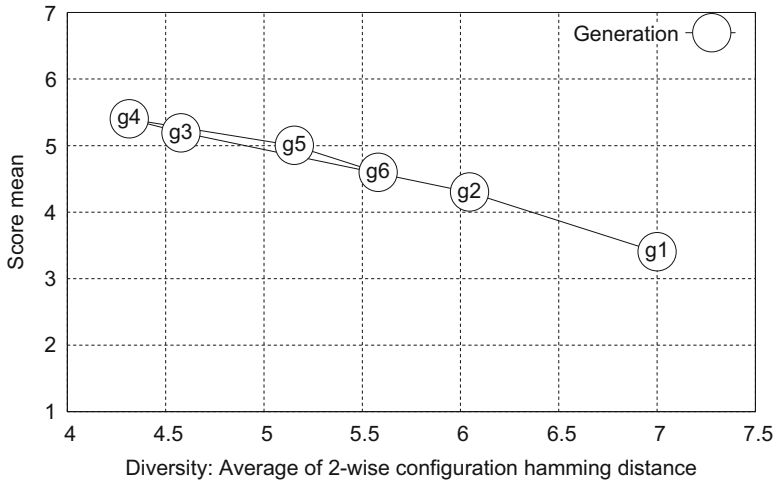




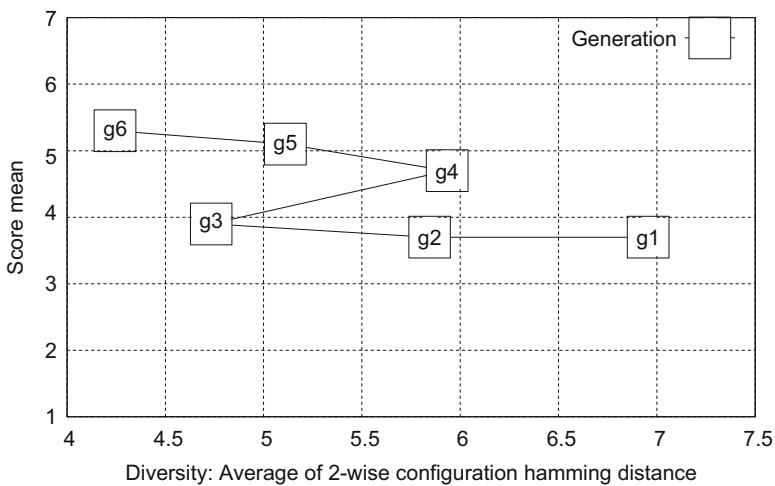
**Fig. 3.8** Results of the genetic algorithm evolution in the two organizations. (a) LIST. (b) SnT

that the users had in UI design. The score mean improved quickly from generation one to four by filtering really inappropriate variants, and then decreased a little bit because of their capacity to criticize the proposed variants. They may not evaluate the variant itself but its capability to be different from what they already evaluated. Furthermore, some of these critics were related to non-variability related issues. Other possible explanation could be user fatigue.

In order to observe whether the IGA tends to converge, Fig. 3.9 shows the progression of the generations in the two dimensional space of mean score and diversity. We calculated the genotype diversity along the different generations ( $g1$  to  $g6$ ) as the average of the Hamming distance of all pair of configurations in the



(a)



(b)

Fig. 3.9 Generations progress in terms of mean score and diversity. (a) LIST. (b) SnT

generation. The diversity decreases if we approach to the left side of the horizontal axis. For example, we can observe how the diversity is not increasing more than its value at  $g1$  which is the randomly created population. For LIST, as shown in Fig. 3.9a,  $g4$  has both the lowest diversity and the maximum mean score. In the case of SnT, as shown in Fig. 3.9b, the last generation ( $g6$ ) has both the lowest diversity and the maximum mean score. In the LIST case, the user pool was able to reach better variants for them earlier.

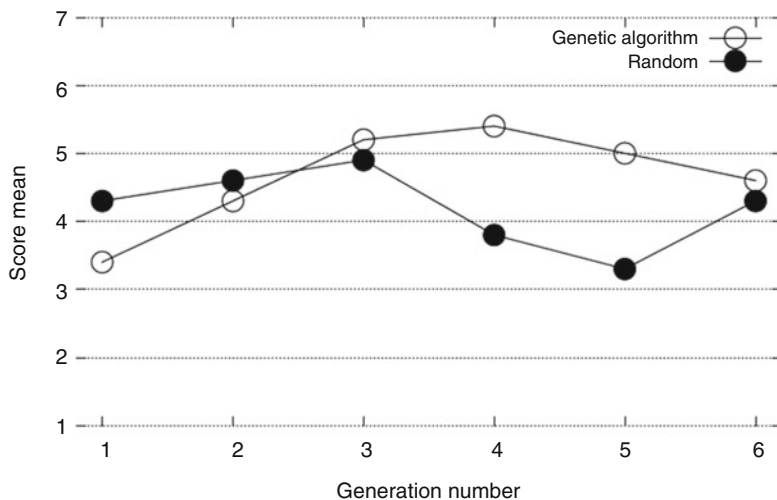
**UI Quality Improvement.** Regarding the first hypothesis, we evaluate if our process based on evolutionary techniques selects better variants than a randomized algorithm for a given number of iterations. We repeated the experiments with the same participants using random selection. In this approach, for each generation, 10 configurations were automatically selected from the viable space which is the same size of the population that we used for the IGA. Basically, for the random selection, we used the same operator as the one used for seeding in the IGA. Despite that we still call each group of 10 random configurations a generation, no genetic information was propagated from one generation to the next. Figure 3.10 shows the results of the genetic algorithm and the random selection in order to compare them. The most important observation is that random selection failed to obtain a global score mean greater than 5 in any of the generations while the genetic algorithm did achieve it. We can see how the genetic algorithm outperforms the random selection approach except in the first two generations where the effect of evolution is still trying to find relevant regions of the configuration space.

Table 3.1 presents the representative improvements obtained in the two independent experiments by comparing the global score mean. The global score mean is the mean of the assessment scores in all the generations. The genetic algorithm approach has a global score mean which is around 0.5 points better (i.e., 0.45 in LIST and 0.55 points in SnT).

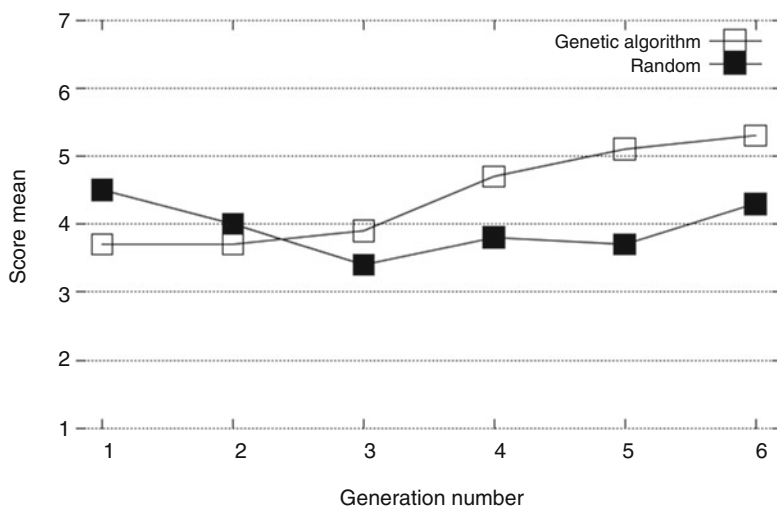
The proposed genetic algorithm produced better results over the generations than the random approach. The algorithm tends to converge to a solution in this search of better UI configurations. To prove this, we computed the diversity of the members of each generation. If the diversity has a tendency to decrease, it is a sign of convergence. Figure 3.11 shows the graph of the results at organization LIST and SnT for both the genetic algorithm and the random process. We can see how the random approaches in both organizations do not decrease the diversity while, for these 6 generations, we observe how the genetic algorithm performs better than the random approach to reduce the diversity. The random approach failed to decrease the diversity to values lower than 5 while this was achieved by the genetic algorithm approach. As result, we can conclude that, compared to the random approach, we both increase the global mean score and we reduce the diversity along the generations. These two aspects allow the genetic algorithm to try to converge to optimal or sub-optimal solutions which means to relevant UI designs.

**Usability Expert Analysis.** In order to confirm our second hypothesis we required a usability expert with nine years of experience to assess that the better variants found by our approach satisfy usability criteria. This expert is independent in order to provide an impartial assessment. He does not belong to the team that developed the considered project, nor participated during the variant assessment. We summarize the expert qualitative evaluations on the three relevant variants shown in Fig. 3.12:

- The first variant, shown in Fig. 3.12a, is the simplest list with no master detail. It addresses several usability criteria [37] such as low workload, explicit control,



(a)

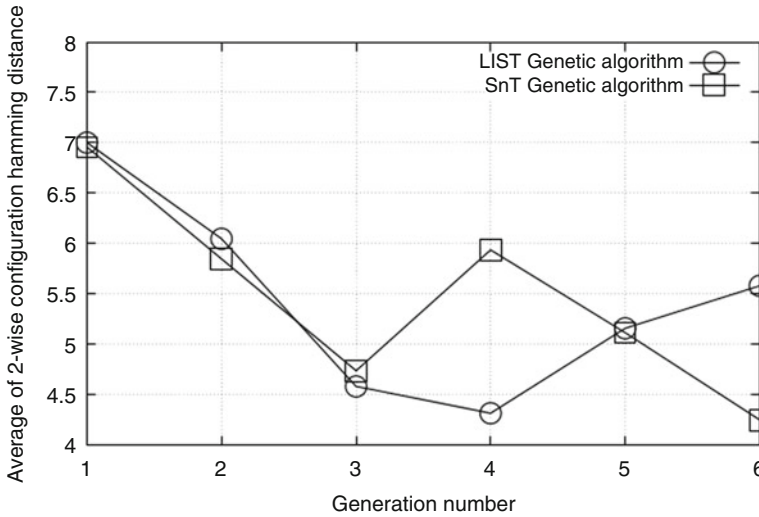


(b)

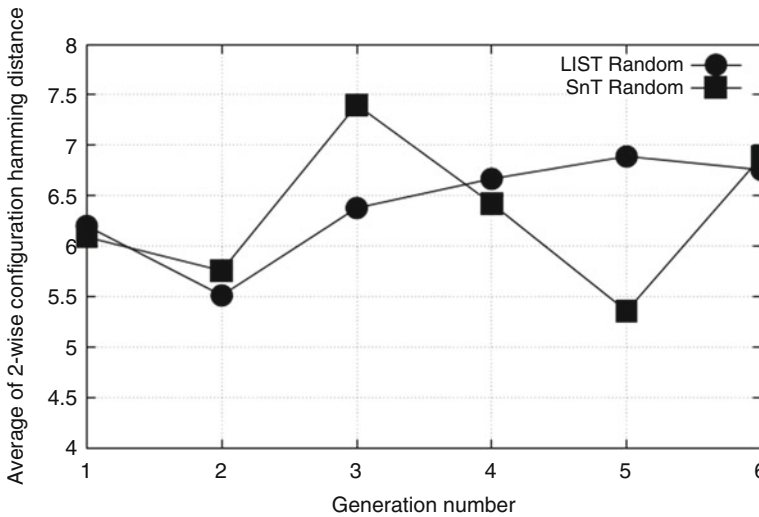
**Fig. 3.10** Comparing variant selection based on genetic algorithm and random. (a) LIST. (b) SnT

**Table 3.1** Global score mean evaluation

	GA	Random
LIST	4.65	4.20
SnT	4.40	3.95



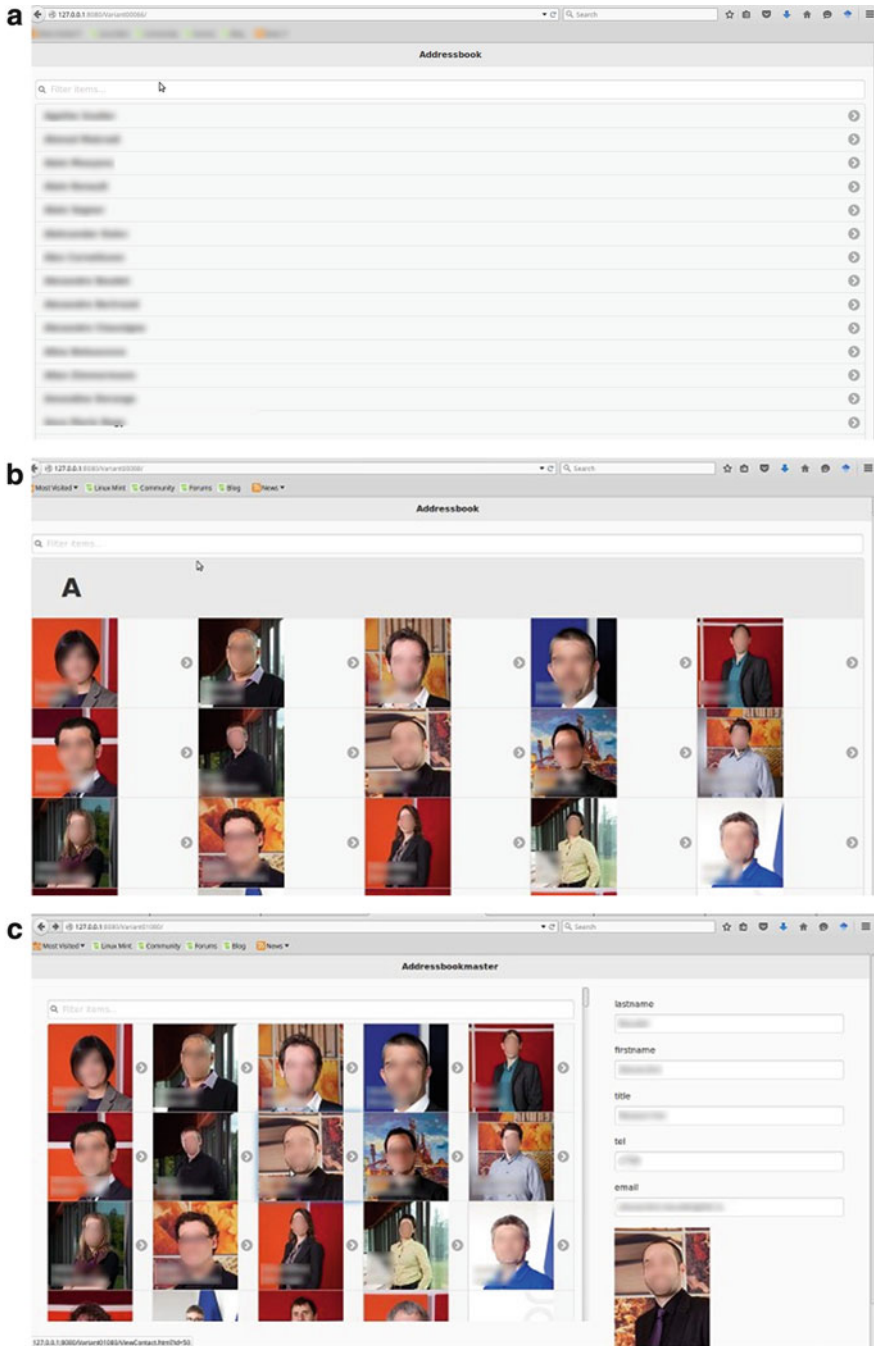
(a)



(b)

Fig. 3.11 Genotype population diversity. (a) Genetic algorithm. (b) Random

homogeneity/consistency or compatibility with traditional contact applications. The search bar and the simplicity of the UI allows the end user to go directly to what he/she is looking for. However, as drawback, it is not possible to browse through the contacts' photos or to do a visual research if the name of the person is unknown.



**Fig. 3.12** Screenshots of relevant variants. (a) ListView, Name and no MasterDetailInterface. (b) TileList, Name, Photo, Indexed, Filter and no MasterDetailInterface. (c) TileList, Name, Photo, Filter, MasterDetailInterface (with Big Master Area and Master Left) and Vertical Grid

- The second variant, shown in Fig. 3.12b, has a better appearance (aesthetic consideration) and has more information (i.e., photo and index). It also complies well with Scapin and Bastien’s usability criteria [37]. Notably the adaptability criteria is well implemented here: the application can be convenient to the different situations of use (e.g., on large screen and small screen display, etc.). However, it seems visually overloaded. Reducing the number of persons displayed in the list can be an option. Another important point noticed is that the users can just play with the UI (e.g., browse through colleague photos) and be distracted from the prescribed task.
- The third variant, reproduced in Fig. 3.12c, is very close to the previous one, except for the master/detail pattern. It also complies with most of the usability criteria. The list of persons is more compact than the previous variant (Fig. 3.12b) giving a better impression. The information is accessible directly without the need to navigate which is a plus for large screens but not necessarily the best solution. In the configuration with a Master Detail interface, the layout is important, and in this variant the vertical grid fits perfectly with this layout.

The usability expert claimed that the relevant variants that have emerged from applying the approach satisfy most of the usability criteria.

### 3.7 Conclusion and Perspectives

In this article, we addressed the issue related to UI variability. UI variability holds numerous facets (e.g., graphical design, development, usability, etc.) due to the diversity of stakeholder profiles whom may contribute to the UI development life cycle, such as, but not limited to, end-users. Moreover, in UI design, one encounters frequently the difficulty to align the products with fuzzily defined user requirements. This complexity can lead to an inefficient UI design process, which has an impact on the UI design costs.

Therefore, we proposed an approach to manage UI variability based on MDE and SPL, integrating SPL management into our current MDE UI design process. In order to build a viable product, the stakeholders have to confront their viewpoints when configuring products. This is the general approach we have illustrated here, adopted for rapid prototyping.

This article focused on the end-users as peculiar stakeholders of the design process. Such stakeholders can intervene in some specific parts of the process using partial FM: i.e., help in choosing some features. Nevertheless the potential indirection between feature and the resulting UI could make things unclear. This is why we should still rely on user assessments.

When using SPL, we can obtain many variants of the same UI, thus making it difficult to assess for end-users. How end-users would assess more than 800 variants? Even, if they had time to evaluate it, when assessing many similar UIs the user-fatigue will provide biased results. We could have used partial configuration to split the problem and focus on a specific element such as in [43]. However, it will not have provided results for the overall interaction experience.

We have experimented with an Interactive Genetic Algorithm that helps to assess and deal with many variants produced by an SPL approach. It helps to reduce the number of UI to be tested by end-users and to find consensus on a relevant versions (i.e., of a good quality). First, it facilitates the exploration of the design space (as defined in the FM) and with a rather small portion of the possible configurations. Second, it helps to assess the variants with a group of users reducing some personal subjectivity.

One future direction of this work would be to propose a ranking of the features which influence user decisions the most. As such we would be able to predict the configurations which are not relevant or the most adapted to specific interaction situations.

**Acknowledgements** This work has been partially supported by the FNR CORE Project MoDEL C12/IS/3977071. The work of Jabier Martinez is funded by the AFR grant agreement 7898764. The work of Alfonso García Frey is partially co-funded by Yotako S.A. and the FNR Luxembourg under the AFR grant agreement 7859308. Special thanks to Alain Vagner for his contributions.

## References

1. Abrahão S, Iborra E, Vanderdonck J. Usability evaluation of user interfaces generated with a model-driven architecture tool. In: *Maturing usability*. London: Springer; 2008. p. 3–32.
2. Acher M, Collet P, Lahire P, France RB. Separation of concerns in feature modeling: support and applications. In: *Proceedings of the 11th Conference on Aspect-Oriented Software Development*. 2012.
3. Acher M, Collet P, Lahire P, France RB. Familiar: a domain-specific language for large scale management of feature models. *Sci Comput Program*. 2013;78(6):657–81.
4. Barreiros J, Moreira A. Soft constraints in feature models. In: *Proceedings of ICSEA 2011: The Sixth International Conference on Software Engineering Advances*. IARIA XPS Press; 2011. p. 136–141. ISBN: 978-1-61208-165-6.
5. Batory D, Azanza M, Saraiva J. The objects and arrows of computational design. In: *Model driven engineering languages and systems*. Berlin: Springer; 2008. p. 1–20.
6. Benavides D, Martín-Arroyo PT, Cortés AR. Automated reasoning on feature models. In: *CAiSE*. 2005. p. 491–503.
7. Benavides D, Segura S, Ruiz-Cortés A. Automated analysis of feature models 20 years later: a literature review. *Inf Syst*. 2010;35(6):615–36.
8. Brummermann H, Keunecke M, Schmid K. Variability issues in the evolution of information system ecosystems. In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. 2011.
9. Bühne S, Lauenroth K, Pohl K. Why is it not sufficient to model requirements variability with feature models. In: *Workshop on Automotive Requirements Engineering (AURE04)*, at RE04. 2004.
10. Calvary G, Coutaz J, Thevenin D, Limbourg Q, Bouillon L, Vanderdonck J. A unifying reference framework for multi-target user interfaces. *Interact Comput*. 2003;15(3):289–308.
11. do Carmo Machado I, McGregor JD, de Almeida ES. Strategies for testing products in software product lines. *ACM SIGSOFT Softw Eng Notes*. 2012;37(6):1–8.
12. Clements P, Northrop L. *Software product lines*. Boston/London: Addison-Wesley Boston; 2002.
13. Czarnecki K, Antkiewicz M, Kim CHP, Lau S, Pietroszek K. Model-driven software product lines. In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM; 2005. p. 126–27.



14. Czarniecki K, Helsen S, Eisenecker U. Staged configuration through specialization and multilevel configuration of feature models. *Softw Process Improv Practice*. 2005;10(2): 143–69.
15. DIS I. 9241-210: 2010. Ergonomics of human system interaction-part 210: human-centred design for interactive systems. Geneva: International Standardization Organization (ISO); 2009.
16. Eiben AE, Smith JE. Introduction to evolutionary computing. Berlin/London: Springer; 2003.
17. Ensan F, Bagheri E, Gašević D. Evolutionary search-based test generation for software product line feature models. In: Ralyté J, Franch X, Brinkkemper S, Wrycza S, editors. *Advanced information systems engineering. Lecture notes in computer science*, vol. 7328. Berlin/Heidelberg: Springer; 2012. p. 613–28.
18. Gabillon Y, Biri N, Otjacques B. Designing multi-context UIs by software product line approach. In: *ICHCI'13*. 2013.
19. García JG, Vanderdonckt J, González-Calleros JM. Flowixml: a step towards designing workflow management systems. *Int J Web Eng Technol*. 2008;4(2):163–82. <http://dx.doi.org/10.1504/IJWET.2008.018096>
20. García Frey A, Sottet JS, Vagner A. Ame: an adaptive modelling environment as a collaborative modelling tool. In: *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York: ACM; 2014. p. 189–92.
21. García Frey A, Sottet JS, Vagner A. Towards a multi-stakeholder engineering approach with adaptive modelling environments. In: *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York: ACM; 2014. p. 33–8.
22. García Frey A, Sottet JS, Vagner A. A multi-viewpoint approach to support collaborative user interface generation. In: *19th IEEE International Conference on Computer Supported Cooperative Work in Design CSCWD*. 2015.
23. Henard C, Papadakis M, Perrouin G, Klein J, Traon YL. Multi-objective test generation for software product lines. In: *SPLC*. 2013. p. 62–71.
24. Henard C, Papadakis M, Perrouin G, Klein J, Heymans P, Traon YL. Bypassing the combinatorial explosion: using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans Softw Eng*. 2014;40(7):650–70.
25. Johansen MF, Haugen Ø, Fleurey F, Eldegard AG, Syversen T. Generating better partial covering arrays by modeling weights on sub-product lines. In: *MoDELS*. 2012. p. 269–84.
26. Kang KC, Lee J, Donohoe P. Feature-oriented project line engineering. *IEEE Softw*. 2002;19(4):58–65.
27. López-Jaquero V, Vanderdonckt J, Simarro FM, González P. Towards an extended model of user interface adaptation: the isatine framework. In: Gulliksen J, Harning MB, Palanque PA, van der Veer GC, Wesson J, editors. *Engineering Interactive Systems – EIS 2007 Joint Working Conferences, EHCI 2007, DSV-IS 2007, HCSE 2007, Salamanca, Mar 22–24, 2007. Selected Papers. Lecture notes in computer science*, vol. 4940. Springer; 2007. p. 374–92. [http://dx.doi.org/10.1007/978-3-540-92698-6\\_23](http://dx.doi.org/10.1007/978-3-540-92698-6_23)
28. Mannion M, Savolainen J, Asikainen T. Viewpoint-oriented variability modeling. In: *COMP-SAC'09*. 2009.
29. Martinez J, Lopez C, Ulacia E, del Hierro M. Towards a model-driven product line for web systems. In: *5th Model-Driven Web Engineering Workshop MDWE*. 2009.
30. Nielsen J, Landauer TK. A mathematical model of the finding of usability problems. In: *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*. ACM; 1993. p. 206–13.
31. OMG. IFML – interaction flow modeling language. 2013.
32. Pleuss A, Botterweck G, Dhungana D. Integrating automated product derivation and individual user interface design. *VaMoS*. 2010;10:69–76.
33. Pleuss A, Hauptmann B, Dhungana D, Botterweck G. User interface engineering for software product lines: the dilemma between automation and usability. In: *EICS*. New York: ACM; 2012. p. 25–34.

34. Pohl K, Böckle G, Van Der Linden F. *Software product line engineering: foundations, principles, and techniques*. Berlin: Springer; 2005.
35. Rosenmüller M, Siegmund N. Automating the configuration of multi software product lines. In: VaMoS. 2010. p. 123–30.
36. Sayyad AS, Menzies T, Ammar H. On the value of user preferences in search-based software engineering: a case study in software product lines. In: ICSE. 2013. p. 492–501.
37. Scapin DL, Bastien JC. Ergonomic criteria for evaluating the ergonomic quality of interactive systems. *Behav Inf Technol*. 1997;16(4–5):220–31.
38. Schlee M, Vanderdonckt J. Generative programming of graphical user interfaces. In: *Proceedings of the Working Conference on Advanced Visual Interfaces, AVI'04*. New York: ACM; 2004. p. 403–6. <http://doi.acm.org/10.1145/989863.989936>
39. Shneiderman B. Promoting universal usability with multi-layer interface design. In: *Proceedings of the 2003 Conference on Universal Usability, CUU'03*. New York: ACM; 2003. p. 1–8. <http://doi.acm.org/10.1145/957205.957206>
40. Sottet JS, Vagner A. Genius: generating usable user interfaces. arXiv preprint arXiv:1310.1758; 2013.
41. Sottet JS, Vagner A. Defining domain specific transformations in human-computer interfaces development. In: *2nd Conference on Model-Driven Engineering for Software Development*. 2014.
42. Sottet JS, Calvary G, Coutaz J, Favre JM. A model-driven engineering approach for the usability of plastic user interfaces. In: *Engineering Interactive Systems*. Berlin/New York: Springer; 2008. p. 140–57.
43. Sottet JS, Vagner A, García Frey A. Model transformation configuration and variability management for user interface design. In: *International Conference on Model-Driven Engineering and Software Development*. Springer International Publishing; 2015. p. 390–404.
44. Sottet JS, Vagner A, García Frey A. Variability management supporting the model-driven design of user interfaces. In: *Modelsward*. 2015.
45. Sumner T, Davies S, Lemke AC, Polson PG. Iterative design of a voice dialog design environment. In: Wixon DR, editor. *Posters and Short Talks of the 1992 SIGCHI Conference on Human Factors in Computing Systems, CHI 1992, Monterey, 3–7 May 1992*. New York: ACM; 1992. p. 31. <http://doi.acm.org/10.1145/1125021.1125050>.
46. Syswerda G. Uniform crossover in genetic algorithms. In: Schaffer JD, editor. *Proceedings of the 3rd International Conference on Genetic Algorithms (ICGA)*. Morgan Kaufmann; 1989, p. 2–9. ISBN: 1-55860-066-3.
47. Takagi H. Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation. *Proc IEEE*. 2001;89(9):1275–96.
48. White J, Dougherty B, Schmidt DC, Benavides D. Automated reasoning for multi-step feature model configuration problems. In: *Proceedings of the 13th International Software Product Line Conference*. 2009.

# Chapter 4

## Feature-Based Elicitation of Cognitively Efficient Visualizations for SPL Configurations

Céline Sauvage-Thomase, Nicolas Biri, Gilles Perrouin, Nicolas Genon,  
and Patrick Heymans

**Abstract** Configuring a SPL is a cognitively difficult activity that requires a deep understanding of the features and their constraints to be performed effectively. To this end, SPL configurators have been equipped with various visualizations to assist users in their tasks. However, there are many ways to visualize data: the process of associating an efficient visualization to a given (configuration) task is neither well-understood nor systematically applied, resulting in confusing visualizations yielding configuration errors. In this chapter, we offer such a process, based on theories of the visualization community for data representation. The first step consists in choosing the data to be visualized. This selection induces restrictions on the types of visualization that are then computed based on the data characteristics and best practices from semiology and visual languages. Designers can then select an efficient visualization for the intended task. Our process is supported by feature models and FAMILIAR to merge and constrain the set of applicable visualizations.

### 4.1 Introduction

The activity of configuring a product in the context of a Software Product Line (SPL) is cognitively challenging. Indeed, the engineer has for instance to understand the complex relationships of the features involved in the configuration of his product. Even if solving these dependencies can be automated (e.g. by using a SAT solver in the back-end), it does not help regarding configuration understanding. To this end, current configurations tools integrate some visualization supports intended to help users in their tasks. However there is more than a single way to visually represent a configuration [16] and we argue in this paper that the process of associating a

---

C. Sauvage-Thomase (✉) • N. Biri  
Luxembourg Institute of Technology, Esch-sur-Alzette, Luxembourg  
e-mail: [celine.thomase@list.lu](mailto:celine.thomase@list.lu); [nicolas.biri@list.lu](mailto:nicolas.biri@list.lu)

G. Perrouin • N. Genon • P. Heymans  
PReCISE, Faculty of Computer Science, University of Namur, Namur, Belgium  
e-mail: [gilles.perrouin@unamur.be](mailto:gilles.perrouin@unamur.be); [nicolas.genon@unamur.be](mailto:nicolas.genon@unamur.be); [patrick.heyman@unamur.be](mailto:patrick.heyman@unamur.be)

visualization to a given task is not well-understood and mostly results in generic visualizations. These visualizations are clearly sub-optimal to perform some tasks such as the understanding of propagations during configuration.

To provide dedicated visualizations, we present a method to explicitly model and guide visualization choices in terms of feature models (FM) and relate them with the kind of data involved for a particular task. The kinds of data involved in the configuration task (e.g. features, constraints) are defined on a first FM named dataset FM. The possible visualization designs (e.g. trees, maps, etc.) are also represented with FMs, one by visualization design, named visualization designs FMs. The dataset FM configuration induces restrictions on the possible configurations of the visualization design FMs thanks to a set of mapping rules. These rules are based on the types of the configured data and their possible representations by the visual properties [2] of the visualization designs where a visual property is a graphical element that can be perceived by the human eye (e.g color, shape, size). Each visualization design FM offers different choices for visual properties assignment. To ease the configuration task, these visualization FMs are merged using the FAMILIAR environment [1] in a single visualization FM and configuration constraints are automatically generated. Hence, the valid configurations of the visualization FM forms thus the set of appropriate visualizations for the considered input data.

The remainder of this chapter is organized as follows: Sect. 4.2 illustrates our approach on a configuration task issue example. Section 4.3 describes the different steps of our method. Section 4.4 applies the method through our running example. Section 4.5 provides an overview of the visualization designs guidance approaches existing in the literature. Finally, Sect. 4.6 concludes this paper by wrapping up our contributions and highlighting some future perspectives.

## 4.2 Example

To illustrate the challenge of visualization choice, we focus on issues occurring during product configuration in SPL. In an industrial environment, feature models tend to be complex, involving an important number (thousands) of features and complex crosstree constraints [18, 20].

In [4], Deelstra et al. pointed out that in large feature models involving a lot of complex cross-tree constraints, engineers can not accurately see how their choices impact other features. This difficulty is twofold: on the one hand, numerous cross-tree constraints make the consequences of a choice difficult to forecast, on the other hand, a large number of features can make the configuration impact difficult to spot. Our task is thus to select an adapted visualization to ease the comprehension of a feature selection during the configuration of the FM.

We take the Graph Product Line (GPL) [7] as a running example of product line configuration throughout this chapter. On the GPL feature model, we start a configuration with the selection of the following features `Gpl`, `MainGpl`, `Test`, `StartHere` and `Base`. From this point, we continue with the selection

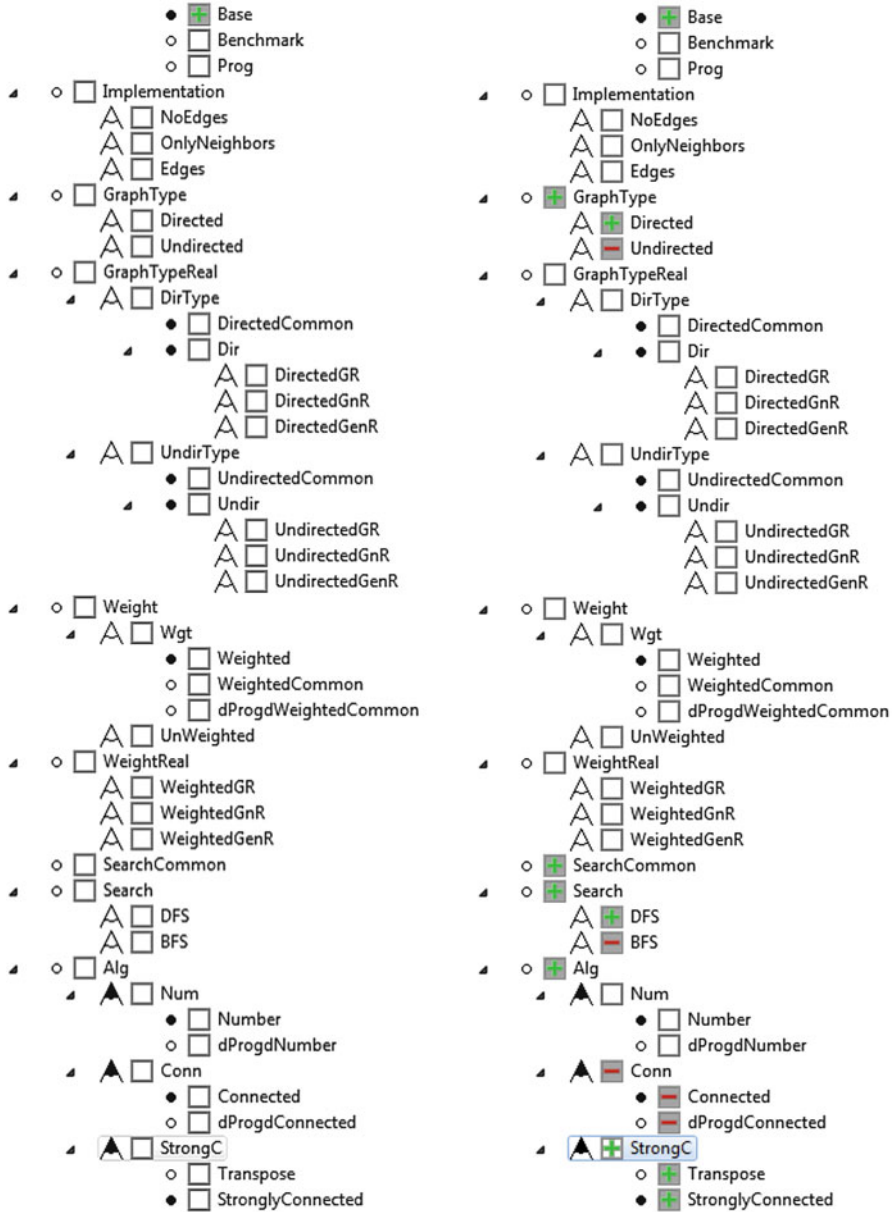


Fig. 4.1 Feature IDE configuration views

of the feature `StrongC`. In Fig. 4.1, the screenshots on the left hand side and on the right hand side display respectively the configuration *before* and *after* the feature selection in the Feature IDE configuration tool. Similarly, in Fig. 4.2, the screenshots

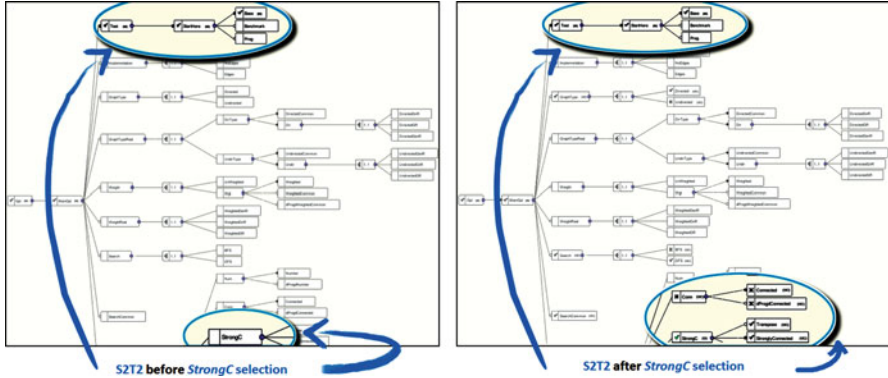


Fig. 4.2 S2T2 configuration views

on the left and on the right show respectively the state of the S2T2 configuration tools *before* and *after* the selection. The distinction of the feature names in Figs. 4.1 and 4.2 is not important. The focus is on the fact that a feature is selected/rejected.

First, we must acknowledge that none of the proposed tools explicitly claims to tackle the choice understanding issue. Nevertheless, none of the two editors makes a visual difference between the features that are selected or rejected *before* the StrongC selection and those becoming selected or rejected *after* the selection. As a consequence, it is difficult to figure out the differences *before* and *after* the selection. S2T2 differentiates the user choices from constraint propagation, which can partially support the user, but it does not directly address the issue.

In Sect. 4.4, we will apply our visualization selection method to this example and show how we can explicitly represent the choice consequences in a both immediate and meaningful way.

### 4.3 Visualization Elicitation Method

We propose an SPL-backed method that guides the selection of an appropriate visualization for the data involved in a configuration user task. Though the models offered are dedicated to the support SPL configuration tasks, the method can be adapted to any data visualization needs.

The method follows a traditional two-fold SPL approach [13]. First, the domain engineering phase, led by a *visualization domain engineer*, consists in the definition of the FMs that will support the selection of the visualization. Second, the application engineering phase, led by the *configuration visualization engineer* that will configure these models and fine tune the selected visualization who will be provided to the *end-user*. The process is illustrated in Fig. 4.3.

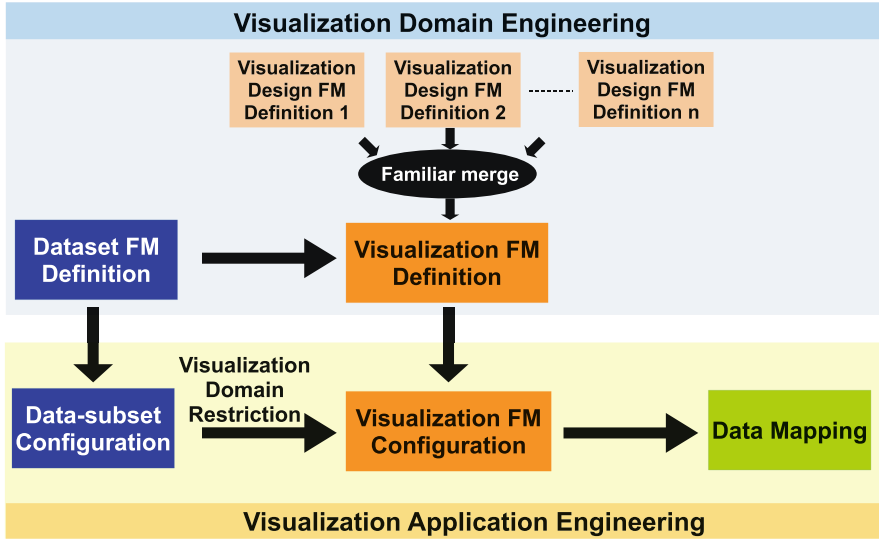


Fig. 4.3 Visualization engineering process

### 4.3.1 Visualization Domain Engineering

The visualization domain engineering phase consists in the formalization of two feature models. A *dataset FM*, specific to an application domain, which allows the configuration of the dataset to visualize, and the *visualization FM*, a variability model that represents all the visualization possibilities that the visualization domain engineer will offer to the configuration engineer. This second model is data agnostic and thus can be reused from one domain to the other.

#### 4.3.1.1 Dataset FM

During this phase, the visualization domain engineer defines a FM that represents the datasets that can potentially be interesting to provide insight to the end user.

The definition of the dataset FM is built on the multidimensional concepts formalised in the context of OnLine Analytical Processing (OLAP) research [14]. These concepts contain the notions of *dimensions* and *measures*. A *dimension* is somehow a concern on the data. More precisely it defines an analysis axis formed by a set of data with the same datatype, providing a base on which the other data are analyzed. A *measure* describes data that have to be analyzed over the dimensions. Bearing in mind these definitions, the dataset FM is set with three subtrees (see Fig. 4.4). One for the set of *dimensions* involved in a configuration process, another for the set of *measures* associated with these dimensions and the last for the set of *derived measures* deduced from these *measures*.

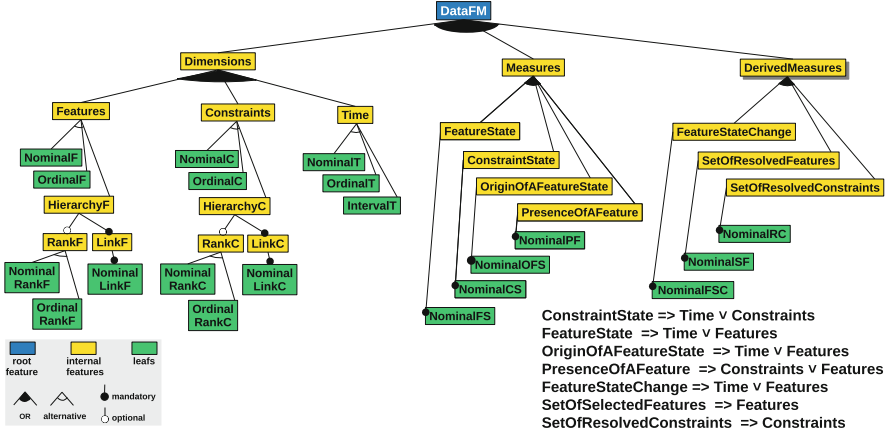


Fig. 4.4 Dataset FM

To ease the mapping with visual solutions, we also need to define the datatypes associated to the data constituting the dimensions and measures. To do so, we use the four types of measurement scale cited by Stevens in [21], which provide precise types of scale for the dimensions and the measures [24]. The first type is *nominal* and defines the possibility for elements to be distinguishable. The second type is *ordinal* and means that the elements can be sorted depending on a rank order. The third type is named *interval* and adds the capacity to calculate the difference between two elements. Finally, the type *ratio* is the more expressive. It defines the additional possibility to calculate a ratio between two elements. These datatypes are defined on the leaf features of the three subtrees *dimensions*, *measures* and *derived measures*. It has to be noticed that several types can be associated with the same feature to let the possibility to represent a *dimension*, a *measure* or a *derived measure* with a less expressive datatype than initially needed. The prefixes Interval, Ordinal and Nominal of the leaf features will be used as types to restrict the possibilities in the visualization models during the application engineering phase (see Sect. 4.3.2.3).

In the following paragraphs, we will detail the set of *dimensions*, *measures* and *derived measures* with their associated datatypes that are involved in a configuration process.

### Dimensions

Configuring a feature model means making successive choices on features in accordance with the constraints expressed on the feature model. From this definition, three dimensions can be deduced and expressed on the data feature model by the visualization domain engineer.



**Features dimension.** Given that a configuration is carried out by selecting features, the set of features forms an obvious dimension which is identified as `Features` in Fig. 4.4. The dimension `Features` has three children in the dataFM: `NominalF`, `OrdinalF` and `HierarchyF`. The `HierarchyF` child is defined to reflect the hierarchical organization of the features on a feature model. Actually, the notion of hierarchy encompasses a mandatory parent-child relationship (`LinkF` in the dataset FM) and an optional rank of the elements inside the hierarchy (`RankF` in the dataset FM), this decomposition is motivated by the visual representation of hierarchy detailed in Sect. 4.3.1.2. The parent-child relationship has a nominal datatype (`NominalLinkF` in the dataset FM) because the information to convey is the existence of a relationship between the linked elements. The rank requires an ordinal datatype (`NominalRankF` in the dataset FM) as there is a specific order between the ranks of the hierarchy (i.e., rank 1 < rank 2 < ...). However, a representation of this hierarchical organization is not always necessary. The children `NominalF` and `OrdinalF` define the datatypes nominal and ordinal and allow to represent only the feature names or respectively an ordered list of the feature names.

**Constraints dimension.** The feature choices are restricted by the feature model constraints. Thus, another relevant dimension is the `Constraints`. The `constraints` dimension is composed by two types of constraints. The first type is formed by the hierarchical constraints between the child nodes and the parent nodes of the feature tree. The second type is the cross-tree constraints scattered over the whole feature model. All the constraints can be represented with an algebraic tree where the internal nodes are logical operators and the leafs are logical literals involving a feature. Resultantly the `constraints` dimension has also a hierarchical organization and its datatypes are defined in the same way as those of the `constraints` dimension.

**Time dimension.** During a configuration process successive choices are made. The `Time` dimension allows the analysis of the configuration process at different points in time. We consider that the configuration process is constituted by a set of configuration steps where each step contains a user choice which may be followed by an automatic propagation of constraints. The `Time` dimension is formed with this set of configuration steps. The relevant associated datatype for the time dimension is interval, named as `IntervalT` in the dataset FM. However if the time difference between two time data does not need to be represented, the datatype ordinal is adequate and the feature `ordinalT` is selected. Similarly, if the representation of the order between two time data is not useful, the feature `NominalT` representing the datatype nominal is chosen.

## Measures

The visualization domain engineer defines on the dataset FM the set of measures that are involved during a configuration process. As previously mentioned, the measures are associated with one or more dimensions. To visualize a measure, at least one of its dimensions must also be present in the dataset. Consequently, to ensure the

data feature model consistency, cross-tree constraints are needed expressing that the measures can not exist without at least one of their dimensions. Hence, for each measure added in the data feature model, a cross-tree constraint is created involving the measure and its associated dimensions.

**Feature state.** This measure is straightforward as it denotes the state of a feature during a configuration. A feature is selected, rejected or choice free. Its associated dimensions are the `Time` and the `Features`. Its datatype is nominal. In Fig. 4.4 this measure is identified as `FeatureState` and its datatype as `NominalFS`.

**Constraint state.** This measure identifies the state of a constraint which can be resolved or not resolved during a configuration process. Its associated dimensions are `Time` and `Constraints`. Its datatype is nominal. This measure is named as `ConstraintState`, its datatype as `NominalCS`.

**Feature state origin.** This measure distinguish whether a feature is selected or rejected following a user action (chosen) or following an automatic propagation of constraints (deduced). Its associated dimensions are `Time` and `Features`. Its datatype is nominal. This measure is named as `OriginOfAFeatureState` and its datatype as `NominalOFS`.

**Feature-constraint involvement.** This measure determines if a feature is part of a given constraint. Its associated dimensions are `Constraints` and `Features`. Its datatype is nominal. This measure is identified as `PresenceOfAFeature` and its datatype as `NominalPF`.

### The Derived Measures

These measures are deduced from the main measures that we have just described. It has to be noticed that this list of deduced measures is not exhaustive.

**Feature state change.** This measure allows to know whether a feature becomes selected or rejected between two given configuration steps. It is deduced from the measure `FeatureState` by identifying the feature states which are modified between the two given configuration steps. Thus, it is associated with the same dimensions: `Time` and `Features`. Its datatype is nominal. In Fig. 4.4, this measure is named as `FeatureStateChange` and its datatype as `NominalFSC`.

**Set of selected features.** This measure shows the set of selected features at a given configuration step. It is deduced from the measure `FeatureState`. Resultantly, it is associated with `Time` and `Features`. Its datatype is nominal. This measure is named as `SetOfSelectedFeatures` and its datatype as `NominalSF`. According to the measure semantics, this measure is irrelevant without the selection of the `Features` dimension. Consequently, while for the other measures one of the dimensions associated with is indifferently mandatory, for this measure the `Features` dimension is specifically mandatory. The following constraint is added to the feature model:

`SetOfSelectedFeatures`  $\Rightarrow$  `Features`.

**Set of resolved constraints.** This measure defines the set of resolved constraints at a given configuration step. It is deduced from the measure `ConstraintState`. Thus, it is associated with `Time` and `Constraints`. Its datatype is nominal. This measure is named as `SetOfSelectedConstraints` and its datatype as `NominalSC`. Similarly to the measure `SetOfSelectedFeatures`, the following constraint is added to the feature model:

`SetOfResolvedConstraints`  $\Rightarrow$  `Constraints`.

### 4.3.1.2 Visualization FM

The second task of the visualization domain engineer is to characterize the visualizations in the form of a FM called the visualization FM. In the following, a visualization refers to a complete visualization design. A treemap [6] illustrated in Fig. 4.5 or a sunburst [19] illustrated in Fig. 4.6 are some examples of visualization designs. A visualization design is defined in accordance with a set of visual properties and a series of constraints. The visualization FM is defined in several steps. First, we will define a FM that characterizes the set of visual variables and their capacity expression in terms of data. Then, on this base, the visualization domain engineer provides, for each proposed visualization design, a visualization design FM allowing the configuration of the visual properties. Finally, the set of defined visualization design FMs are merged to obtain the visualization FM unifying the set of valid configurations of each visualization design FM.



Fig. 4.5 Treemap designed with calluna, a LIST visualization solution

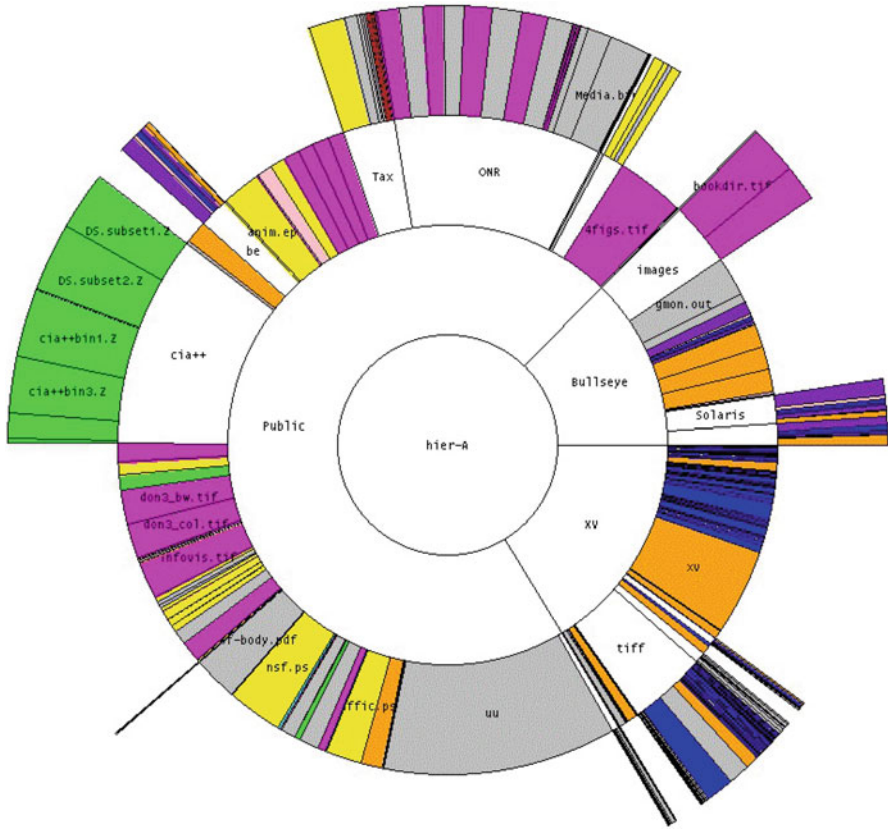


Fig. 4.6 Sunburst (depiction of a file hierarchy [19])

**FM for visual variables.** We start with the definition of a reference FM to describe the levels of measurement of each of the visual properties. The level of measurement denotes the highest datatype that the property can express. There exists several classifications of the visual properties according to their levels of measurements [2, 3, 8, 9, 23]. We settle on Bertin's proposal [2]. At the current stage, we do not claim the FM to be complete, and we define it in a scalable way to support further extensions of the mapping rules.

Bertin defines eight visual variables that are the building blocs to design any symbol (see Fig. 4.7). The variables spread out into two categories: the planar variables ( $x$  and  $y$  position on a 2D plan) and six retinal variables (shape, color, orientation, size, texture and value (aka., brightness)). Each visual variable is suitable to express a certain level of measurement. More precisely, this referent FM maps the dimensions and measures datatypes onto the visual variables' levels of measurement. As represented by Fig. 4.8, the root of this FM is the Mapping feature. This feature has seven children that denote each of the visual

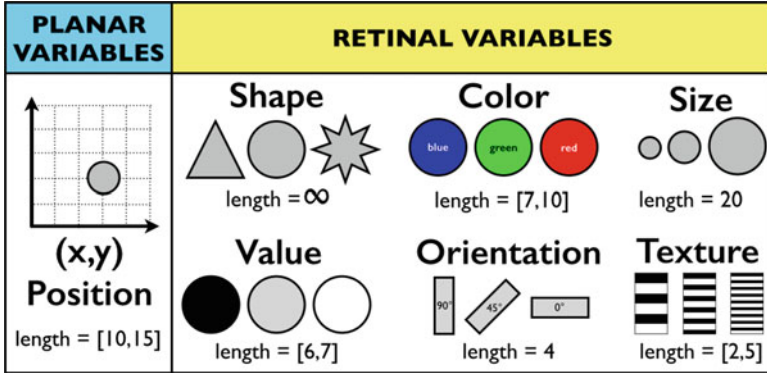


Fig. 4.7 The eight visual variables defined by Bertin [2]

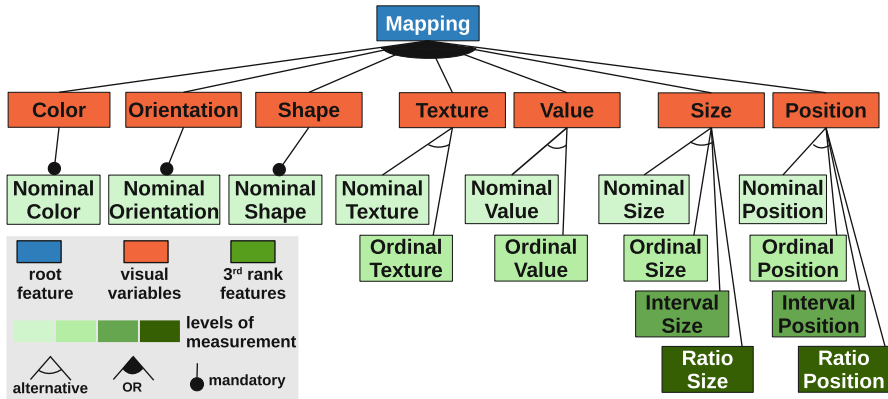


Fig. 4.8 Mapping between the visual variables and their levels of measurement

variables (except for the planar variables that are gathered under the feature Position). The Shape, Orientation and Color have respectively one child that indicates a nominal level of measurement (i.e., the features NominalShape, NominalOrientation and NominalColor). Texture and Value are ordinal variables and hence have respectively two children: NominalTexture, OrdinalTexture and NominalValue, OrdinalValue. The variables of Size and (x,y) Position have the highest level of measurement: each of them have four children that correspond to the nominal, ordinal, interval and ratio levels of measurement.

The hierarchy notion elicited in Sect. 4.3.1.1 has no immediate correspondence to the levels of measurement of the visual variables. However, as the hierarchy is expressed in the dataset FM with a mandatory nominal datatype and an ordinal datatype, it can be mapped to a pair (nominal, ordinal) of visual variables' levels of measurement or to a simple nominal level. For example, in the usual representation

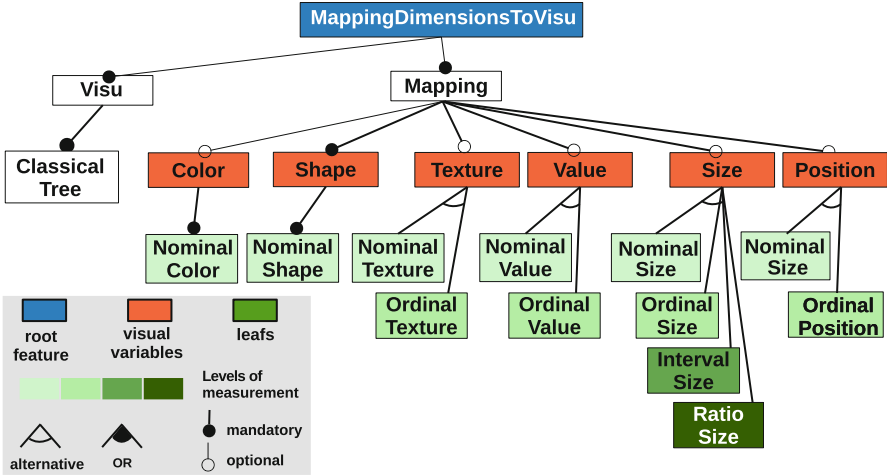


Fig. 4.9 Classical tree FM

of FM, the hierarchy is expressed by the *shape* and *y position* visual variables: parent elements are linked to each child element by a line that is depicted with some *shape*. Moreover, the parent elements are located on the top of the diagram, while their children lie below, which corresponds to distinct *y positions* (and all elements of a same rank have the same *y position*).

**FMs for visualization designs.** The second step aims at defining, with the support of the reference mapping FM, a FM specific to each visualization designs that the visualization domain engineer wants to integrate. For example, the FM specific to the classical tree designs is represented in Fig. 4.9. They describe the different visual variables contained in the visualization and their associated datatypes following the FM for visual variables. Features marked as optional correspond to visual variables that can be used in such visualizations but are not mandatory. For example, a treemap can be colored but it’s not a requirement. On the contrary, to be a meaningful choice, the size of treemap elements must carry an information, thus the size is required for this visualization.

**FM for Visualization.** The third step is a merge operation of all the FMs specific to each visualization design. This operation is managed by the *Familiar* tool [1] and allows to obtain a single FM that integrates all the variability of the involved FMs. The use of Familiar in this case is particularly interesting as it provides a straightforward solution to propose different FMs for different visualization subsets. The Fig. 4.10 illustrates the result of the merge of the treemap FM, the classical tree FM and the sunburst FM, as depicted by Familiar. Some constraints allowing to manage the variability of the visual variables depending on the type of visualization designs have been generated by the tool. The merge resulting FM is the visualization FM that the designer has to configure to select the adapted visualization design.

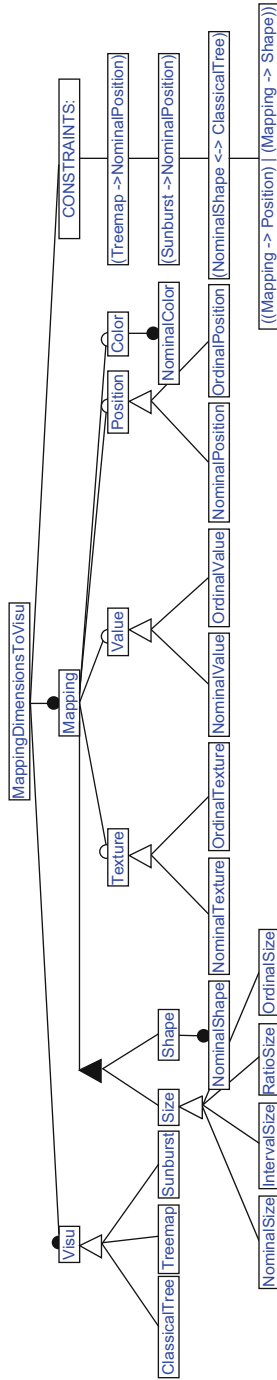


Fig. 4.10 Visualization mapping FM (exported from FAMILIAR)

### 4.3.2 Visualization Application Engineering

Once the visualization domain engineering is complete, the visualization application engineering phase consists mainly in the configuration of the different feature models. The phase is composed of the following steps:

1. The configuration visualization engineer will decide which subset from the original dataset must be present in the resulting visualization.
2. The visualization FM is *automatically* restricted thanks to extra-constraints to reduce the configuration space. Consequently, it allows only visualizations that are relevant for the selected data subset.
3. The configuration visualization engineer configures the restricted visualization FM to choose the visualization and its details among those that are still available.
4. The configuration visualization engineer maps the selected data to the adequate visual variables.

#### 4.3.2.1 Data-Subset Configuration

The configuration visualization engineer configures the dataset FM defined during the visualization domain engineering phase (see Sect. 4.3.1.1) depending on the dimensions and measures involved in the specific configuration user task considered. The result is a subset of the data with their corresponding datatypes.

#### 4.3.2.2 Visualization Domain Restriction

This phase is performed automatically by the system after the data subset configuration. The configuration of the dataset FM leads to establish a combination of different datatypes to visualize. Resultantly, it is necessary to generate constraints on the visualization FM about the datatypes that are mandatory following the configuration of the dataset FM.

The generation of the additional constraints is based on the subset of datatypes contained in the valid configuration of the dataset FM. The subset of datatypes is defined as follows:

**Definition 4.1 (Datatypes subset)** Given a configuration  $\mathcal{C}$ , the datatypes subset  $\mathcal{D}_{\mathcal{C}}$  of this configuration is the tuple  $\{n, o, i, r\}$  (with  $n, o, i, r \in \mathbb{N}$ ) where:

- $n$  is the number of features in  $\mathcal{C}$  prefixed by `Nominal`;
- $o$  is the number of features in  $\mathcal{C}$  prefixed by `Ordinal`;
- $i$  is the number of features in  $\mathcal{C}$  prefixed by `Interval`;
- $r$  is the number of features in  $\mathcal{C}$  prefixed by `Ratio`;

Having  $x$  features of a given datatype in the data subset implies that the designer must choose  $x$  visual variables that support the same datatype in the visualization



feature model to obtain a valid configuration. Suppose that the visualization feature model holds  $y$  features being of this datatype, the designer must form a combination of  $x$  elements out of  $y$  (with  $y \geq x$ ) for this datatype. The number of the possible combinations is consequently  $\binom{y}{x}$ .

Given a visualization feature model  $V$  and a datatype  $d$ , the set of visual features for this datatype,  $\mathcal{V}_{\{V,d\}}$  is the set of features of  $V$  that are prefixed by the datatype name  $d$ .

Given a set of features  $F$  we define  $C(F, n)$  as the disjunction of all the conjunctions of  $n$  distinct elements of  $F$ .

With these definitions, the set of generated constraints is defined as follow:

**Definition 4.2 (Datatypes subset constraints)** Given a datatype subset  $\mathcal{D}_{\mathcal{C}} = \{n, o, i, r\}$  and a visualization feature model  $V$ , the datatypes subset constraints of  $\mathcal{D}_{\mathcal{C}}$  and  $V$  is the following:

$$\{C(\mathcal{V}_{\{V, \text{Nominal}\}}, n), C(\mathcal{V}_{\{V, \text{Ordinal}\}}, o), C(\mathcal{V}_{\{V, \text{Interval}\}}, i), C(\mathcal{V}_{\{V, \text{Ratio}\}}, r)\}$$

As example, let `NominalViz` the set of features representing a `Nominal` datatype in the visualization FM:

`NominalViz = (NominalTexture, NominalColor, NominalShape)`

If the configuration visualization engineer chooses 2 `Nominal` datatypes in the dataset FM, he has to choose exactly 2 features among `NominalViz`. The number of possible combinations is  $\binom{3}{2}$ . The constraint can be expressed as follows:

$$\begin{aligned} &(\text{NominalTexture} \wedge \text{NominalColor} \wedge \neg \text{NominalShape}) \\ &\vee (\text{NominalTexture} \wedge \text{NominalShape} \wedge \neg \text{NominalColor}) \\ &\vee (\text{NominalColor} \wedge \text{NominalShape} \wedge \neg \text{NominalTexture}) \end{aligned}$$

Once the data subset constraints are generated, they are inserted into the visualization FM with `Familiar` and the configuration visualization engineer can then configure the visualization FM.

For a given datatype, the configuration visualization engineer can select in the dataset FM a number of features representing this datatype greater than the number of existing visual variables expressing this datatype in the visualization FM. In a such situation, none of the proposed visualizations would fit his needs. Thus, either the configuration visualization engineer needs to narrow the data subset or the visualization domain engineer needs to extend the visualization catalog with new visualization designs in order to obtain a visualization FM with valid configurations for the generated constraints.

#### 4.3.2.3 Visualization Configuration

The constraints induced by the configuration of the data feature model and the constraints resulting from the merge of the feature models specific to each visualization designs (see Fig. 4.10) restrict the possible configurations of the visualization FM. The configuration visualization engineer chooses his visualization design among the visualization designs which are still available after this restriction.

#### 4.3.2.4 Data Mapping

The configuration of the visualization FM allows to determine the visualization design, its visual variables and the datatypes that can be expressed by these visual variables. We can define a matrix for each datatype involved: along one axis, we find the candidate visual variables for this datatype, and along the other axis, the data of this type that were selected during the configuration of the dataset FM. The configuration visualization engineer has to designate for each matrix how the data are mapped to the visual variables.

To support this task, the configuration engineer can rely on the length of the visual variables. Each visual variable has a specific *length*, which is the number of distinct values that it can take and that can be effectively perceived by the human's perceptual system. The length of each variable is given in Fig. 4.7. The designer has to ensure that the length of the selected variable is equal or larger than the number of elements of its mapped datatype. Stated another way, the visual variables must at least be able to depict all the elements of the datatype. By applying this rule, the number of candidate variables can be reduced.

Another factor that can reduce the set of candidate variables is the choice of variables that are deliberately not bind to any selected data from the data feature model. This choice is supported by the fact that the configuration engineer wants to keep free some of the visual variables to allow extra information to be added later on the diagram. Indeed, every variable that is mapped to a datatype cannot be easily reused to convey a new meaning on the same diagram. The set of free variables forms the secondary notation, while the primary notation is defined by the set of bound visual variables. For instance, if we consider that it would be important to be able to point out one or several specific elements on a diagram, the variable *color* should be kept free. Color is a variable that is easily and almost instantly perceived. Moreover, the human's perceptual system is able to isolate all occurrences of a given color on a diagram, and hence the color is really appropriate for pointing out elements.

### 4.4 Application

In this section we apply our approach on the GPL product line introduced in Sect. 4.2. In particular, we want to select an appropriate visualization for the following task: “understanding the consequences of feature selection”.

#### 4.4.1 Visualization Domain Engineering

The visualization domain engineering phase consists in the definition of the dataset FM and the visualization FM. The dataset FM and the visualization FM used for this application case are those illustrated in Figs. 4.4 and 4.10. The visualization

FM allows to configure a classical tree visualization design, a treemap or a sunburst but can be extended to accommodate new visualizations.

## 4.4.2 Visualization Application Engineering

The same steps as those described in Sect. 4.3.2 have to be followed in order to complete the visualization application engineering phase for our example.

### 4.4.2.1 Data-Subset Configuration

This step consists in selecting only the features on the dataset FM which are useful for the analysis of the consequences of a feature choice on the other features. Understanding the consequences of a choice implies to visualize the FM and to be able to detect the feature states that become selected or rejected following this choice.

We describe the features selected on the dataset FM as dimensions, then the features selected as measures and finally, the features selected as derived measures.

**Dimensions.** As we need to visualize the FM, the `Features` dimension is necessary. Consequently the set of following features are selected:

```
{DataFM, Dimensions, Features, HierarchyF,
LinkF, NominalLinkF, RankF, OrdinalRankF}
```

**Measures.** In order to know the state of the features after the choice, the measure `FeatureState` has to be selected on the dataset FM. Consequently, the following features are selected:

```
{Measures, FeatureState, NominalFS}
```

**The derived measures.** Finally, to be able to detect a feature state that become selected or rejected, the derived measure `FeatureStateChange` is needed. The following features are thus selected:

```
{DerivedMeasures, FeatureStateChange, NominalFSC}
```

### 4.4.2.2 Visualization Domain Restriction

The configuration of the dataset FM leads to the generation of some constraints on the visualization FM in accordance with the explanations given in Sect. 4.3.2.2. They imply that the final configuration of the visualization FM contains exactly three visual variables representing a nominal datatype and one representing an ordinal datatype.

### 4.4.2.3 Visualization Configuration

The visualization domain restriction does not restrict the choices of global visualization designs in the visualization FM. Hence, in Fig. 4.10, the three children of the feature `Visu` (`ClassicalTree`, `Treemap`, `Sunburst`) can be selected. We select the `ClassicalTree` feature in order to compare the design resulting from our methodology with the classical tree visualization of the existing configuration tools. As explained in Sect. 4.3.1.2, in the classical tree design, the relationship between the parent and the children is usually mapped to the `shape` as nominal. The rank is usually mapped to the `Position` visual variable as ordinal. Resultantly, we select these visual variables in order to map them to the features `LinkF` and `RankF` of the dataset FM. The constraints generated in Sect. 4.4.2.2 require three visual variables to be able to express the nominal datatypes. Given that `Shape` and `Position` are already selected, we need to select two other visual variables among `Texture`, `Color`, `Value` and `Size`. We choose `Texture` and `Color`. The constraints require also one visual variable to depict the datatype ordinal. This constraint is already resolved due to the previous selection of the `Position` visual variable as ordinal. Finally, we obtain the following configuration:

```
{MappingDimensionsToVisu, Visu, ClassicalTree,
Mapping, Shape, NominalShape,
Texture, NominalTexture,
Position, OrdinalPosition,
Color, NominalColor}
```

### 4.4.2.4 Data Mapping

As described in Sect. 4.3.2.4, for each datatype existing in the visualization configuration, we define a matrix with, along one axis, the possible visual variables for the datatype concerned and along the other axis, the set of features of this type that were selected during the configuration of the dataset FM.

We decided in Sect. 4.4.2.3 to map the feature `LinkF` to the `Shape` visual variable. Consequently, concerning the datatype nominal, two features from the dataset FM have to be mapped to the visual variables `Texture` and `Color`: `FeatureState` and `FeatureStateChange`. The matrix obtained is illustrated in Fig. 4.11 and the crosses indicate the resulting mapping. Given that the length of the visual variables `Texture` and `Color` (see Fig. 4.7) are both adapted to represent the measures `FeatureState` and `FeatureStateChange`, we could choose another mapping.

**Fig. 4.11** Mapping nominal datatype/visual variables

	Texture	Color
Feature State		×
Feature State Change	×	

Concerning the datatype ordinal, we decided in Sect. 4.4.2.3 to map the feature RankF to the Position visual variable. Hence, a matrix definition is irrelevant for the type ordinal.

### 4.4.3 Resulting Visualization

The dataset FM configuration, the visualization FM configuration and the data mapping lead us to build a visualization design holding three essential characteristics.

The first characteristic is that the visualization design is a classical tree representing the feature dimension. We choose to represent a feature by a circle with an affixed text label indicating the feature name.

The second characteristic is the color of the node circle circumference that represents the measure FeatureState. We choose the green and the red color to visualize the feature state *selected* and respectively *rejected*. We take a light blue for the feature state *free*.

The third characteristic is the circle circumference line texture that indicates the measure FeatureStateChange. We choose to design a solid line when a feature state becomes selected or rejected after the choice and a dashed line when a feature state was already selected or rejected before the choice.

By following our method with the aim to visualize the impacts of a feature choice, we obtain the visualization design illustrated in Fig. 4.12. The Figure presents the FM configuration view after the selection of the feature StrongC as detailed in Sect. 4.2.

We can compare our result with the configuration views proposed by the configuration tools Feature IDE and S2T2 illustrated in Figs. 4.1 and 4.2. Whereas on our configuration view, the impacts of the feature StrongC selection are clearly identifiable due to the use of the color and texture on the feature nodes, the impacts are not pointed out on the two other configuration views. S2T2 offers an animation to get a glimpse of the impacts when the stakeholder selects a feature. However, this furtive glimpse is not suitable for the analysis of the selection consequences, particularly if all the consequences are not visible on the same space screen. Feature IDE uses the red and green colors to represent the feature state but does not make the distinction between a selection/rejection of features before/after a given feature choice.

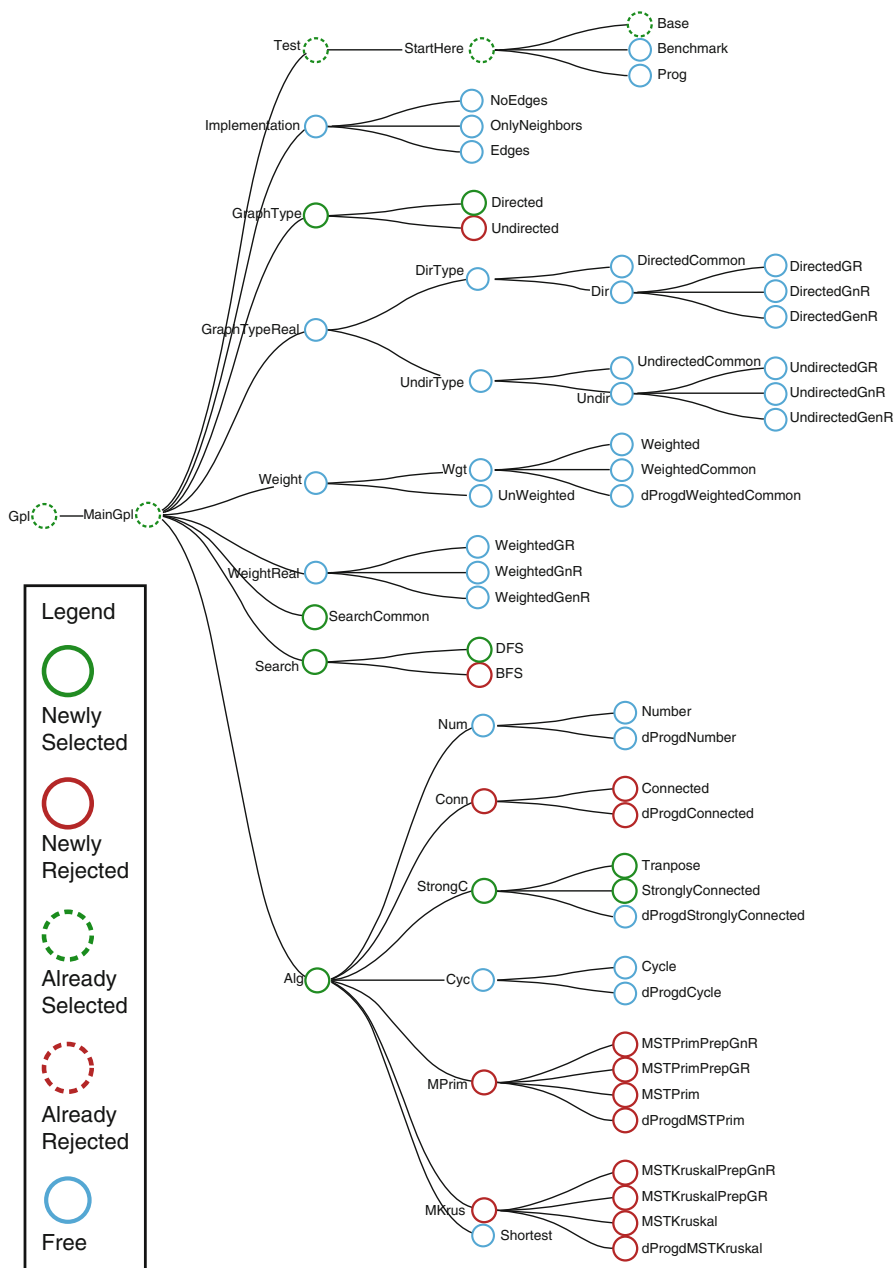


Fig. 4.12 FM configuration view

## 4.5 Related Work

Among the common existing configuration tools, some contain basic FM configuration views [22], others are the result of specific research aiming to integrate more visual support in the configuration tools. They offer additional views [10, 11] or advanced configuration views in order to help the stakeholders to make their choices. S2T2, the tool presented in [12] and illustrated in Fig. 4.2 offers several interesting visualization mechanisms such as, for example, different tree layouts, zoom and pan mechanism, or color usage to indicate the configuration progress. Whilst the designed choices are well documented, there is, to the best of our knowledge, no explicit method to support them. The objective of our method is to (at least partially) automate the reasoning process that leads to such design choices, as shown in the example.

These various visual propositions for the configuration tools reveal a growing interest among SPL researchers to find adapted visualization techniques for the configuration tasks.

For its part, the visualization community has come up with approaches intended to guide the designer towards a solution visualization. In [9], Mackinlay bases his work on the expressiveness and effectiveness of a set of primitive graphical languages. He presents a framework for the development of tools which are able to automatically generate a design for relational information. Whereas the algorithm presented decides on a unique relevant visualization design, our approach leaves the final decision to the configuration visualization engineer, given that several visualizations may satisfy constraints. A methodology based on a connection between the data interpretation aims and the data representation possibilities is presented in [15]. In [24], Zhang gives a classification of relational information displays that can guide the designers to select a visualization solution. More recently, [5] presented a tool that allows to select a visualization design in accordance with the data characteristics and the user's objectives. This study intends to be used in a visual data mining context where the user's objectives are to discover interesting structure inside the data. In contrast, the general aim of a user in your case study is to take the right configuration decision. For that purpose, a clear display of the appropriate data and of their existing intrinsic relationships is needed. Based on these considerations, our approach is suitable for our goal.

## 4.6 Conclusions

In this chapter, we presented a SPL approach guiding the visualization engineer to a cognitively optimal visualization design choice for SPL configuration tasks. By following the method described, the engineer is not restricted to a specific visualization solution but can choose among a set of suitable designs for the input data involved. Moreover, the visualization solution thus designed ensures consistency in terms of mapping between the visual variables and the data depicted.

Behind the interest for the engineers of a such approach, the use of feature models to support it deserves to be pointed out. Indeed, this particularity allows an easy extensibility of the basic principles we laid down. Moreover, by adapting the dataset FM, our approach can be useful for application domains other than the SPL configuration tasks. Finally, the fact that FMs can be encoded as formal decision models opens the possibility to partial automation of the approach as applied with FAMILIAR in this paper.

Future work will consist in improving the method according to three different aspects. The first concerns the dataset FM. An extension of its definition is necessary in order to take into account the configuration tasks carried out on FMs with a greater expressiveness (i.e. attributes, cardinalities). Moreover, we would like to integrate some quantitative metrics such as the size of the FM and manage the consequences on the resulting visualizations.

The second aspect relates on the visualization solutions proposed. At the current step of our approach, the design depends only on the data involved in the considered user task. As explained in [24], we need to take into account the user task characteristics to improve our propositions of solutions. Furthermore, the dynamic parts of the visualization solutions were deliberately set aside.

The third aspect concerns the improvement of the approach itself by adding guidance to the decisions left at the moment to the visualization engineer. Additionally, we would like to validate our approach by performing an empirical study as described in [17].

**Acknowledgements** This work was partly supported by the European Regional Development Fund (ERDF IDEES/CO-INNOVATION).

## References

1. Acher M, Collet P, Lahire P, France RB. Familiar: a domain-specific language for large scale management of feature models. *Sci Comput Program*. 2013;78(6):657–81.
2. Bertin J. *Semiology of graphics: diagrams, networks, maps* (wj berg, trans.). Madison: The University of Wisconsin Press, Ltd; 1983.
3. Cleveland WC, McGill ME. *Dynamic graphics for statistics*. 1st ed. Boca Raton: CRC Press, Inc.; 1988. ISBN: 053409144X.
4. Deelstra S, Sinnema M, Bosch J. A product derivation framework for software product families. In: *Software product-family engineering*. Berlin/Heidelberg: Springer; 2004. p. 473–84.
5. Guettala AET, Bouali F, Guinot C, Venturini G. A user assistant for the selection and parameterization of the visualizations in visual data mining. In: *2012 16th International Conference on Information Visualisation (IV)*. Los Alamitos: IEEE; 2012. p. 252–57.
6. Johnson B, Shneiderman B. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In: *Proceedings of the IEEE Conference on Visualization, Visualization'91*. Los Alamitos: IEEE; 1991. p. 284–91.
7. Lopez-Herrejón RE, Batory D. A standard problem for evaluating product-line methodologies. In: *Generative and component-based software engineering*. Berlin/Heidelberg: Springer; 2001. p. 10–24.



8. MacEachren AM. How maps work: representation, visualization, and design. New York/London: Guilford Press; 2004.
9. Mackinlay J. Automating the design of graphical presentations of relational information. *AcM Trans Graph. (Tog)* 1986;5(2):110–41.
10. Murashkin A, Antkiewicz M, Rayside D, Czarnecki K. Visualization and exploration of optimal variants in product line engineering. In: *Proceedings of the 17th International Software Product Line Conference*. New York: ACM; 2013. p. 111–15.
11. Nöhler A, Egyed A. C2o: a tool for guided decision-making. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. New York: ACM; 2010. p. 363–64.
12. Pleuss A, Botterweck G. Visualization of variability and configuration options. *Int J Softw Tools Technol Transfer*. 2012;14(5):497–510.
13. Pohl K, Böckle G, van der Linden FJ. *Software product line engineering: foundations, principles, and techniques*. Berlin: Springer; 2005. doi:10.1007/3-540-28901-1.
14. Ravat F, Teste O, Zurfluh G. *Algebre olap et langage graphique*. 2010. arXiv preprint arXiv:1005.0213.
15. Robertson PK. A methodology for choosing data representations. *IEEE Comput Graph Appl*. 1991;11(3):56–67.
16. Schulz HJ. Treevis. net: a tree visualization reference. *IEEE Comput Graph Appl*. 2011;31(6):11–5.
17. Sedlmair M, Meyer M, Munzner T. Design study methodology: reflections from the trenches and the stacks. *IEEE Trans Vis Comput Graph*. 2012;18(12):2431–40.
18. She S, Lotufo R, Berger T, Wasowski A, Czarnecki K. The variability model of the linux kernel. *VaMoS*. 2010;10:45–51.
19. Stasko J, Catrambone R, Guzdial M, McDonald K. An evaluation of space-filling information visualizations for depicting hierarchical structures. *Int J Hum Comput Stud*. 2000;53(5):663–94.
20. Steger M, Tischler C, Boss B, Müller A, Pertler O, Stolz W, Ferber S. Introducing pla at bosch gasoline systems: experiences and practices. In: *Software product lines*. Berlin/Heidelberg: Springer; 2004. p. 34–50.
21. Stevens SS. On the theory of scales of measurement. *Science*. 1946;103(2684): 677–80.
22. Thum T, Kstner C, Benduhn F, Meinicke J, Saake G, Leich T. FeatureIDE: an extensible framework for feature-oriented software development. *Sci Comput Program*. 2014;79:70–85.
23. Wilkinson L. *The grammar of graphics (statistics and computing)*. New York: Springer; 2005. ISBN: 0387245448.
24. Zhang J. A representational analysis of relational information displays. *Int J Hum Comput Stud*. 1996;45(1):59–74.

# Chapter 5

## Addressing Context-Awareness in User Interface Software Product Lines (UI-SPL) Approaches

Thouraya Sboui, Mounir Ben Ayed, and Adel M. Alimi

**Abstract** The development of context aware UIs has become a major requirement to ensure their efficiency and improve their usability, thus inducing many variations of the same UI depending on the constraints imposed by the context of use. To address this challenge, many works decided to rely on the Software Product Line Engineering (SPLE) paradigm. Software product lines (SPLs) were widely used due to their advantages, such as, but not limited to: the reuse of shared artifacts, the generation of specific products from shared artifacts, the reduction in time, effort and cost of development. Others works have relied on Dynamic Software Product Line (DSPL) approaches to develop a family of adaptive UIs. The DSPL exploits the knowledge acquired in SPLE to develop systems that can be context-aware, post-deployment reconfigurable, or adaptable at run-time. In order to address the challenge of context awareness, this paper presents an overview of UI-(D)SPL approaches which specifically focus on context-aware adaptation of user interfaces.

### 5.1 Introduction

Usability problems continue to create a challenge for interactive system designers, especially when a User Interface (UI) should be created for different computing platforms [1], different contexts of use [6], which are often characterized as a triplet (user, platform, environment) [6, 14]. One response has been an increasing interest for and emphasis on context consideration during UI development by gaining more knowledge about the context of use. The user element typically includes user models or profiles, user preferences, user goals and tasks. The platform element covers the software/hardware technological space in which the interactive system must run. While the environment element often includes many physical properties such as the location (e.g., in location-aware user interfaces), the temperature, the light, the noise, it could also encompass psycho-sociological factors such as organization type, stress level, etc.

---

T. Sboui (✉) • M.B. Ayed • A.M. Alimi  
REGIM lab., University of Sfax, Sfax, Tunisia  
e-mail: [sboui.thouraya@gmail.com](mailto:sboui.thouraya@gmail.com); [mounirbenayed@ieee.org](mailto:mounirbenayed@ieee.org); [adel.alimi@ieee.org](mailto:adel.alimi@ieee.org)

Software Product Lines (SPL) [10, 29] consist of 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. Furthermore, Dynamic software product lines (DSPL) [8, 9, 15] engineering exploits the knowledge acquired in SPL to develop systems that can be context-aware, post-deployment reconfigurable, or adaptable at run-time. From these works emerges the idea of using (D)SPL paradigms to develop a family of similar UIs, especially for context-aware adaptation.

This paper presents an overview of UI-(D)SPL approaches which specifically focus on context-aware adaptation of user interfaces. The first section presents a state of the art of existing (D)SPL approaches used in the field of UI development. The second section focuses on context-aware UI-SPL approaches. The first two sections are concluded by a list of challenges defined based on the gaps in existing approaches. In the third section, we propose a schema which summarizes approaches that combine Model Driven Engineering (MDE) [34] core assets, in particular Model-Based User Interface Design (MBUID) [7] concepts with SPL core assets to implement a UI-SPL process. This schema will provide a baseline for future UI designers/developers. In the fourth section, and to show the use of the summary schema, we propose a novel UI-DSPL approach, which encompasses a design phase and a runtime phase and is illustrated by a case study.

## 5.2 UI-SPL Approaches

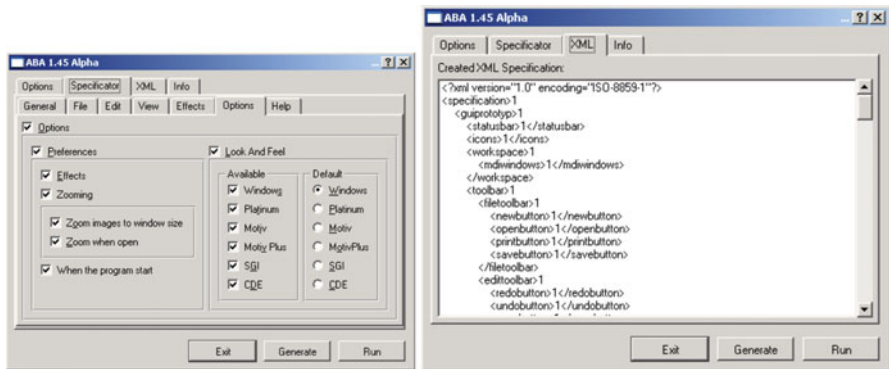
This section presents the first part of the overview. Table 5.1 lists existing UI-SPL proposals, which have been selected and analyzed according to the following criteria:

- *SPL approach type*: specifies whether the used approach is a conventional SPL process, a dynamic SPL process, a model driven engineering (MDE) approach or a model driven SPL (MD-SPL) approach [35].
- *Implementation Technology*: specifies the software technology type (e.g., component based programming, aspect programming, model based design, model driven engineering) used to implement the SPL process.
- *UI type*: specifies the type of the developed UI (e.g., graphical user interfaces).
- *Context consideration*: check whether the context of use was explicitly considered within the approach.

Schlee & Vanderdonck's approach [33] was the first work that used the SPL family to investigate how different GUIs could be adapted for different contexts of use. For the variability modeling, the authors use feature models and for the generation of customized GUIs, they used the frame technology to implement GUI features. MiniABA [33] manipulates a Concrete User Interface (CUI) model expressed as a feature diagram by adding, deleting, updating, moving any feature (Fig. 5.1a). This configuration is then saved in a XML file (Fig. 5.1b) that

**Table 5.1** Comparative Table of UI-SPL approaches

Approach	Approach type	Implementation technology	UI type	Context consideration
Schlee & Vanderdonck (2004) [33]	SPL	Frame Technology	GUI	None
Garcés et al. (2007) [13]	MD-SPL	Models-feature models weaving	GUI	None
Quinton et al. (2011) [31]	MD-SPL	MDE models	Mobile GUI	None
Muller (2011) [23]	SPL	MBUID models	GUI	None
Boucher et al. (2012) [5]	SPL	MBUID Models	Not specified	None
Pleuss et al. (2012) [27]	MD	–	GUI	Yes
Pleuss et al. (2013) [29]	MD-SPL	MBUID Models	GUI	Yes
Arboleda et al. (2013) [3]	MD-SPL	MDE	GUI	None
Logre et al. (2014) [21]	SPL	Aspects	Dashboard	None
Kramer (2014) [19, 20]	DSPL	Document	GUI	Yes
Gabillon et al. (2015) [12]	DSPL	Components	GUI	Yes
Sottet et al. (2015) [35]	SPL	MDE model	GUI	None



**Fig. 5.1** MiniABA options (a) and resulting XML definition (b) [33]

automatically generates a project file to be compiled in C++. Therefore, the adaptation entirely relies on the designer’s ability to tailor the feature model.

Garcés et al. [13] proposed a multi-levelled approach in which they weave SPL artifacts (e.g., architecture and Java feature model) and MDE artifacts (e.g., business, architecture, Java meta-models/models and transformations connecting the different levels) in order to develop a music store application. To generate the application including its GUI, designed models are progressively transformed from a high abstraction level (business level) to the low abstraction level (the generation of Java source code).

Quinton et al. [31] defined two separated feature models: the first one defines the functional variability and the second one defines the device variability of a mobile application. To generate a specific product, features were implemented using MDE models. On another hand, to identify the device on which the application can run, the authors use the meta-model of the AppliDE framework.

Müller [23] adopted Pleuss's approach [28] to implement graphical user interfaces. In his proposal, Müller put the focus on the layout (disposition of widgets in the container) design.

Boucher et al. [5] proposed a SPL approach to develop a configuration interfaces. To define the UI feature model, authors use the configuration workflow (defined separately from the UI feature model), the UI views and the property stylesheets. To generate the concrete structure of the UI, the feature model is mapped into an Abstract User Interface model (AUI), and then the AUI model is mapped, in turn, into a Concrete User Interface (CUI) model, both being specified according to the Cameleon Reference Framework (CRF) [6], as recommended in [7].

In the approaches mentioned above, the context of use was not explicitly considered within the development process. The context consideration within UI-SPL processes started with Pleuss's proposal [27, 29]. Pleuss et al. [29] used Model-Based User Interface Design (MBUID) models [7] to support their approach: a task model representing what the end-user wants to achieve, a domain model representing the data manipulated by the tasks, an Abstract User Interface (AUI) model, and a Concrete User Interface (CUI) model to develop a family of customized UIs. These authors also used MBUID models in [27] to implement the domain and the application engineering levels of a UI-SPL process. The context of use was considered at the design phase and the target context element was the customer element.

Arboleda et al. [3] proposed a SPL proposal in which they combine MDE artifacts and SPL artifacts to implement both engineering level of the SPL process [30]. To derive a specific product, authors use the decision model.

Logre et al. [21] proposed a SPL approach to develop dashboards. A meta-model which supports the dashboards design was defined. Then, based on the meta-model, the dashboard feature model is defined. To generate a specific dashboard, features are implemented using aspects, and then the source code is generated according to the HTML platform.

The context of use was also considered in [12, 19] proposals. Gabillon et al. [12], as well as Kramer [19, 20], used a DSPL process to develop a platform-adaptive UI. The context of use was considered at the design phase and at the runtime phase. To implement UI variability, the authors in [12] have used components while authors of [19, 20] have used GUI documents as source elements for initiating the process.

Sottet et al. [35] used a MBUID approach to generate the Final User Interface (FUI) [7] and used a multi-feature models approach to generate the configuration interface which allows the dynamic configuration of the final UI.

### 5.3 Context Awareness in UI-SPL Approaches

In this section, only approaches that deal with context aware UIs were retained. Table 5.2 lists UI-SPL proposals which were analyzed from the perspective of context consideration according to the following criteria:

- *Time of context consideration*: specifies the phase during which the context of use was considered: the design time phase, the run time phase or in both phases (mixed).
- *Context type*: if the context of use was considered at runtime, the type of the managed context (derived, sensed or profiled) is specified.
- *Context element*: specifies which context element (user, platform or environment) was targeted [7].
- *Context consideration/adaptation techniques*: specifies which technique was used for context design and UI adaptation (if the approach deals with UI adaptation) [22].

#### 5.3.1 Context Consideration at the Design Phase

All approaches reported in Table 5.2 have considered the context of use at least at the design time. However this consideration differs from one approach to another

**Table 5.2** Context consideration in UI-SPL approaches

Approach	Time of Context consideration	Context type	Context element	Context consideration/adaptation techniques
Pleuss et al. (2013) [29] Pleuss et al. (2012) [28]	Design time	None	<user, platform, environment, <b>customer</b> >	MDE models
Kramer (2014) [19, 20]	Mixed	Sensed	<user, <b>platform</b> , environment>	Design time: Context feature model separately designed/UI configuration Runtime:Features/ document-based compositional technique
Gabillon et al. (2015) [12]	Mixed	Sensed	<user, <b>platform</b> , environment>	Design time: Context and UI features combined under the system FM/ UI configuration Runtime: Feature/a component-based compositional technique

according to the purpose of context consideration and according to the technique used to design the context.

In Pleuss et al. [29], the context awareness was performed at the design time phase in order to generate a customized UIs. The standard context triplet [11] was extended with a fourth element which is the customer element: "In addition to the end user there is now also the customer who owns a concrete product. Often, this is not the end user itself". The UI customization was performed during the development process by addressing different UI aspects such as the layout aspect, the navigation aspect, presentation unit aspect, UI elements aspect and so on. The UI customization was mainly designed using additional context models. For instance, we find the navigation model, the clustering model, and the arrangement model. Others aspects, such as the abstract user interface elements, are customized during the derivation of a specific UI or at within (the concrete user interface elements) the transformation connecting the AUI model and the CUI model.

In Gabillon et al. [12] and Kramer [19], the design time context consideration was performed in order to adapt the UI to the context of use. The adaptation was performed at the time of the configuration of the UI feature model. The UI is configured according to the target device on which the application runs. In both proposals, the context was presented using feature models. In Kramer [19], the context feature model was separately defined from the UI feature model. While in [12], context feature and UI features were combined under the umbrella of the system feature model.

### 5.3.2 *Context Consideration at Run-Time*

Only two approaches [12, 19] have addressed the context of use consideration at the runtime phase. The context of use was handled in order to adapt the UI to any context change that is considered significant enough to trigger some adaptation. The sensed data are relative to the platform element. For example, sensed data could be gathered about the battery, the connectivity, the telephony, internet and the data synchronization [19, 20]. Similarly, sensed data could capture relevant data from the computing platform or device used by the end-user, such as screen resolution, screen size, multi-monitor configuration, amount of color palettes, interaction modalities, and computational capabilities [7]. In both approaches, the context of use was represented using features. To recompose the adapted UI, both approaches have used the same technique (compositional technique) but different technologies (in [19], Kramer opted for the use of a document-based compositional technique [32], while in [7], authors have discussed component-based composition. Based on Tables 5.1 and 5.2, the following shortcomings could be mentioned:

- (1) The context of use consideration within UI-SPL approaches is partially covered: only 4 approaches from 11 approaches deal with context-awareness in UI-

SPL approaches. Furthermore, authors use different techniques to approach the context of use. In Pleuss's approaches [27, 29], the context of use was expressed using various declarative models, while in Kramer [19] and Gabillon [12] approaches, the context of use was represented itself using feature models, thus meaning that any variation of the context of use could be explicitly represented in the feature model as opposed to Event-Condition-Action (ECA) adaptation rules for other systems [14].

- (2) To implement the SPL process, existing approaches primarily relied on different technologies such as aspects, components, documents, and models. The use of models rather than any other implementation technology makes the SPL process more abstract and more reusable in principle. However, the use of models, in particular specific UI models (MBUID models) [7] is still not widely used, partially because a modeling activity replaces a development activity.

The next section defines a reference schema for context-aware adaptation which combines the abstraction capability of MDE/MBUID paradigms and the variability management capability of SPLE paradigm in order to develop a family of similar UIs. This combination is unprecedented: existing frameworks usually represent the context of use by models [14], while this reference schema is intended to be used with SPLE. The reference schema highlights how the context of use could be explicitly considered both at the design phase and at the runtime phase. To unify the technique of context design, the feature modeling technique is used instead of models. The reference schema will serve as a reference pattern for designers/developers who want to develop a context sensitive UI-SPL approach.

## 5.4 A Reference Schema for Context-Aware Adaptation

To make the reference schema easier to read, it is presented following the Domain Engineering, and Application Engineering processes which are typically used in SPL. The domain engineering level defines the domain analysis level and the domain design level while the application engineering level defines the application analysis and design level and the application implementation level. The summary schema aggregates SPL artifacts, MDE/MBUID artifacts and context consideration techniques made to the development of a family of context-aware UIs.

### 5.4.1 Design Elements

The design elements are logically distributed throughout the schema levels (Fig. 5.2). The domain analysis level is the process of identifying, eliciting and modeling the requirements of a family of products. The major design elements that can be included in this phase are:



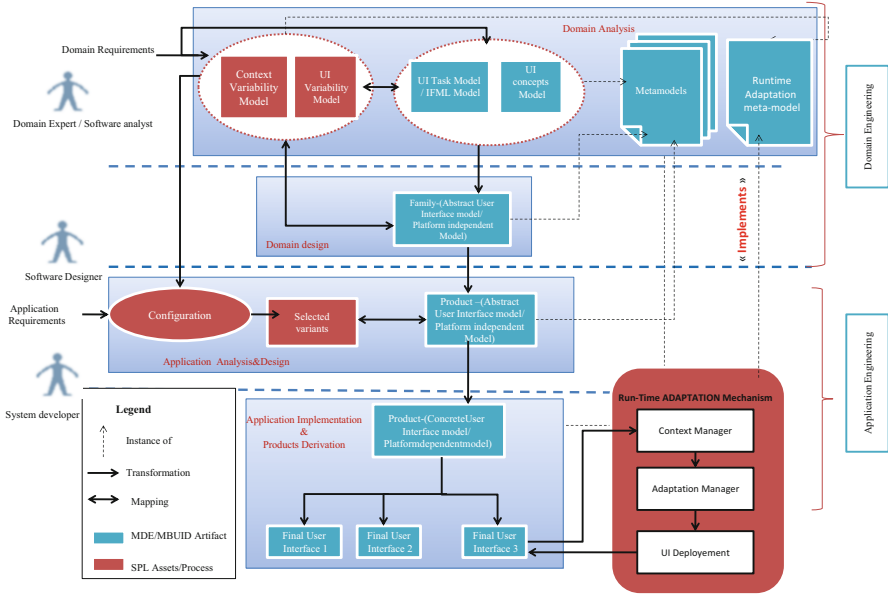


Fig. 5.2 Summary/Reference schema of UI-SPL development process

- *Meta-models*: define the metadata of SPL/MDE artifacts. We find meta-models describing variability models, meta-models describing MBUID models, and meta-models describing MDE models. Another meta-model can take place is that which describes the runtime adaptation mechanism, this meta-model serves to facilitate the design and the development of the runtime adaptation mechanism;
- *Variability models*: define the commonalities and the variabilities of the user interface and the context of use. Feature modeling by means of Feature Diagrams (FDs) is a popular technique for capturing the commonality and the variability in software product lines;
- *UI domain model*: defines the meaningful real-world concepts pertinent to user interface domain, usually by a UML class diagram.
- *Task model/Interaction Flow Modeling Language model*: the task model is the corresponding artifact of the use case diagram in UML language and it represents the logical activities that should support users, interacting with the interface, in reaching their goals. The Concur Task Trees (CTT) [26] is a visual notation used to describe the task model. Regarding the IFML [24] model and beyond the description of user interactions, the IFML was designed for expressing contents, control behavior of the front-end of software applications, as well as the binding to the persistence and business logic layers. The IFML is a Domain Specific

Language (DSL) that has been adopted as a standard by OMG in March 2013 and has its own graphical notation described in the IFML specification document [21]. IFML only covers however the Concrete User Interface (CUI) level and is mainly designed to provide abstractions relevant for web applications more than any other interactive application. Since IFML is a DSL serving for describing a user interface, it is referred to as a User Interface Description Language (UIDL). Other examples of UIDL are UIML [17], MariaXML based on CTT [26], AbaXML [33], and UsiXML [16, 37].

The domain design level takes domain models as described above and aims to produce a generic architecture to which all UIs can be compliant. This architecture is described using the Abstract User Interface (AUI) model or the Platform Independent Model (PIM):

- *Family-specific AUI/PIM models*: both models describe the UI family in terms of interaction spaces (or presentation units), independently of which interactors are available and even independently of the modality of interaction [7]. These models are designed at the domain engineering level to describe presentation units composing the whole family of UIs;

The application engineering level is characterized by the derivation of the user interface product. This derivation satisfies specific application requirements. At this level, the following entities could be identified:

- *Selected variants*: specifies selected and deselected variants of UI and context variability models. At this phase, selected context variants as well as specific application requirements impacts the configuration of the variability model of the user interface;
- *Product-specific AUI/PIM models*: these models are an instantiation of the family-specific AUI/PIM. At the application engineering level, these models describe a specific UI in terms of interaction spaces and independently of which interactors are available and the modality of interaction. At the same time, these models present assets used to implement the selected variants of the UI variability model.
- *Concrete user interface model/platform specific model*: these models describe the interface in terms of concrete interactors that depend on the used modality;
- *Final user interface*: depending on the target platform, this model specifies the source code of the UI in any programming language or mark-up language. Thus, the source code can be interpreted and/or compiled.

Regarding processes, the application engineering level defines three types of processes:

1. *The configuration process*: is the customization of the variability models by selecting and deselecting the appropriate variants in order to meet specific user requirements;
2. *The runtime adaptation mechanism*: this process is responsible of UI adaptation to context changes when the UI is running [22];

3. *Transformation*: is the connection linking the design elements of the different levels of the development process.

### 5.4.2 Transformations

The summary schema defines four types of transformations that connect the design elements of higher level of abstractions to the design elements of lower level of abstractions. These transformations may be automated (performed by the computer autonomously), semi-automated (requiring human intervention) or manually (performed by humans) and are described as follows:

- *Instantiation*: this transformation specifies the meta-models to which SPL and MDE models must conform. An instantiation is an automated transformation which may be performed using Integrated Development Environments (IDEs) (e.g., Eclipse IDE, Feature IDE);
- *MDE/MBUID connections*: the transformation which connects MDE/MBUID models is an automated way that transforms a source model to a target model or to text (e.g. source code). This kind of transformation is defined using transformation languages, collectively known as QVT (Query/View/transformation) [25] languages;
- *SPL connections*: include connections between the variability models and connections between variability models and their configurations. The connection between variability models is supported by a set of composition and decomposition operators (e.g. aggregate, merge, slice) defined by variability management DSLs (e.g., FAMILIAR DSL [2]). While the connection between the variability model and its configuration are performed manually using variability modeling IDEs (e.g., Feature IDE);
- *SPL/MDE connection*: this type of transformation is called a mapping and he links SPL and MDE artifacts. A first mapping links variability models and task/domain models. That means, if we already designed variability models, this will help designers to model the task/domain model and vice-versa. The two other mappings connect variability models to AUI/PIM models, this mapping presents variant's implementation into assets.

### 5.4.3 Actors

The common process supports four types of actors, they include:

- *Domain Expert*: this actor has a thorough knowledge of the domain. After expertise training, the domain expert delivers the glossary of UI terms to the software analyst.

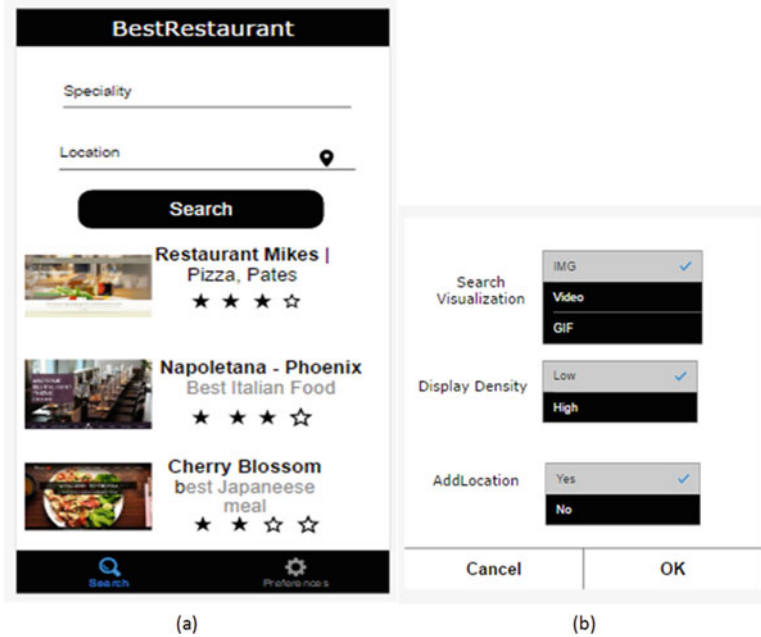
- *Software Analyst*: the analyst studies the domain knowledge provided by the domain expert, and identifies the functional and non-functional specifications based on user requirements.
- *Software Designer*: with the collaboration of the software analyst and the domain expert, the software designer is responsible of the elaboration of domain analysis models/meta-models and the family-specific AUI/PIM model.
- *Software developer*: is responsible of the product derivation. The software developer starts with the configuration of variability models, then she generates the product-specific AUI/PIM and CUI/PSM which will be implemented according to a specific technology (e.g., the FUI model).
- *Final Users*: these are the people who have a stake or interest in the use of interactive systems. They are invited by the analyst to specify their requirements and they are represented, on the schema, by their requirements (domain requirements/application requirements).

To show how the summary schema may be used, we propose a UI-DSPL approach which combines SPL and MBUID artefacts in order to develop a family of adaptable UIs. The approach is illustrated with an example highlighting the UI adaptation according to the user preferences change. User preferences are a contextual data provided manually by the end-user and which may be related to different UI aspects namely, the layout, the visual appearance, the navigation, and UI elements.

## 5.5 Illustrative Case Study

The illustrative example is about a “search for restaurant” application (Fig. 5.3). The application consists of two interface parts: a first main UI for searching any restaurant according to criteria and a second dialog interface for context settings. The search UI (Fig. 5.3a) includes a text field to enter the restaurant speciality, another text field to enter the current location and a search button to validate the search request. By default, the search result is displayed as `imglinks` describing the restaurants which correspond to the search request. The preference UI (Fig. 5.3b) includes three combo Box controls. The first combo Box (labeled “Search Visualization”) presents the user preference towards the visualization element and includes three choices (“video”, “GIFing” and “IMG”). The second combo Box (labeled “Display Density”) presents the user preference towards the displaying density and includes two choices (“high” and “low”) and the third combo Box (labeled “Add Location”) presents the user preference towards specifying or not his current location. The preferences UI as shown in Fig. 5.3b is relative to the following context of use [6]:

- *User*: a novice user with these preferences: an “IMG” visualization search, a “low” display density and an “Add Location” set to “Yes”;



**Fig. 5.3** Application UIs before adaptation. (a) The default search UI, (b) the default context settings

- *Platform*: an average smartphone;
- *Environment*: an outdoor location.

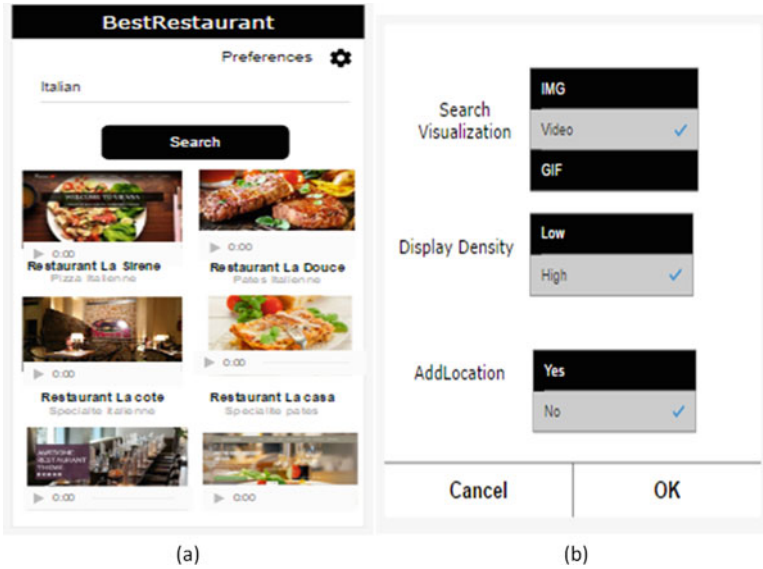
Let us now consider another context of use described as follows:

- *User*: an experimented user who prefers visualizing the search result as video, displayed in a high density format (e.g., many videos per line) and who prefers to hide the location text field (the user is familiar with the location);
- *Platform*: an average smartphone.
- *Environment*: an outdoor location.

The user set his preferences as shown in Fig. 5.4b. After the validation of user preferences change, the search UI is adapted (Fig. 5.4a). Alternatively, the user may choose to visualize the search result in the form of GIF images displayed in a low density format (e.g., one GIF image per line).

### 5.5.1 The Development Process (Design Phase)

The interfaces reproduced in Figs. 5.3 and 5.4 are resulting from the design phase, which phase includes the two levels of the conventional SPL process (Fig. 5.5):



**Fig. 5.4** Application UIs after adaptation. (a) Search UI after adaptation, (b) context settings change

the domain engineering level and the application engineering level. At the domain engineering level, the commonality and the variability of the search and context UIs are defined and modelled. At the application engineering level, specific UIs are derived (Fig. 5.3) by exploiting the commonality and binding variability built into the domain engineering phase.

The domain engineering stage is decomposed into two distinct phases: the domain analysis phase and the domain implementation phase. In the domain analysis phase, the commonalities and variabilities of UIs are defined using feature models. A feature model is organized hierarchically and is graphically depicted as a feature diagram. A feature model is modeled as a tree (Figs. 5.6 and 5.7) in which every node in the tree has one and only one parent except the root feature, a terminal or a concrete feature is a leaf and a non-terminal or compound or abstract feature is an interior node of a feature diagram. Connections between features and its groups of children are classified as “And”, “Or”, and “Alternative groups”. The members of And-groups can be either mandatory or optional.

As depicted in Fig. 5.5, the context feature model and the UI feature model are defined separately. This separation is aimed at minimizing the complexity of the system, facilitating the design process and ensuring its reusability [18, 36]. The UI feature model describes UI variability. The UI variability has been previously investigated by Pleuss et al. [28]. Authors analysed a university management web application by measuring the variability across some concepts, namely presentations units (window, frame) [4], UI elements (widget), the layout aspect,

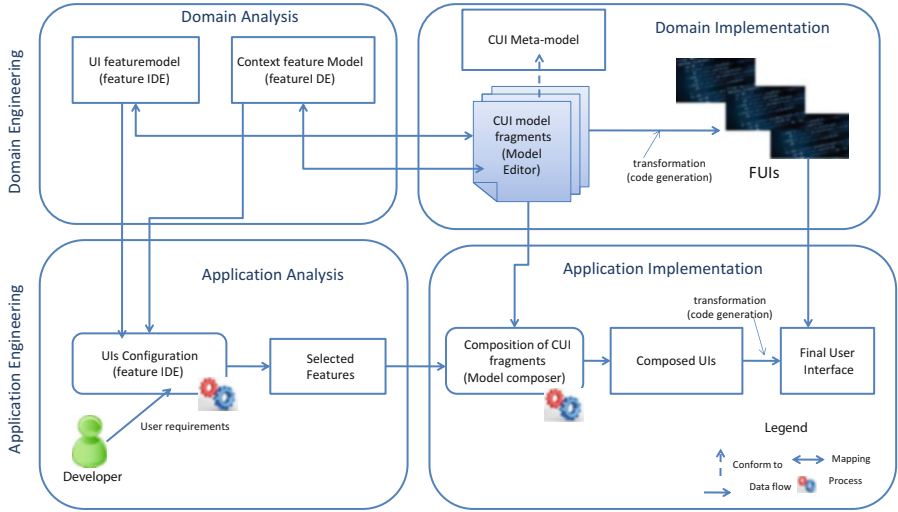


Fig. 5.5 The UI-DSPL approach (design phase)

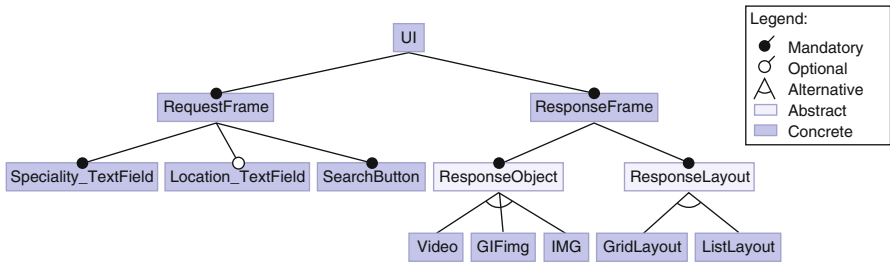


Fig. 5.6 UI feature model

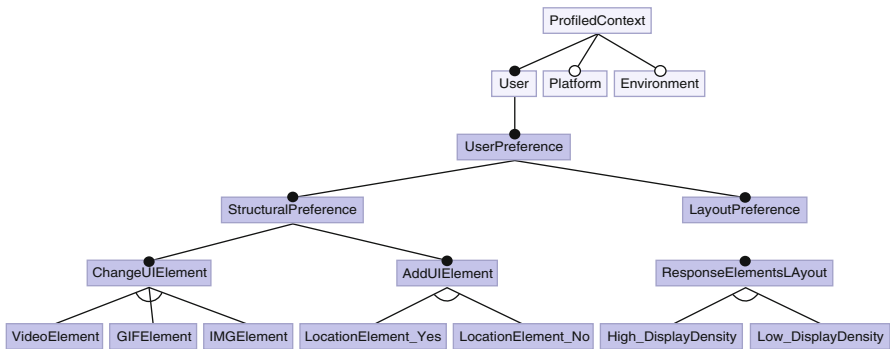


Fig. 5.7 Context feature model

and visual appearance (e.g., foreground color, background color, font size, font type). The feature model presenting the search UI of the case study in Fig. 5.6 was defined in terms of presentations units, UI elements and layout aspects. The features related to the presentation unit aspect include the root feature “UI”, the “requestFrame” feature, and the “responseframe” feature. The features related to UI elements are: “specialityTextField”, “locationTextFiled”, “SearchButton”, “video”, “GIFimg” and “IMG” features. The features related to the layout cover the “listlayout” feature and the “Gridlayout” feature.

In order to adequately cover the three dimensions of the context of use [6], the context feature model (Fig. 5.7) is defined across the triplet <user, platform, and environment> [14]. User preferences feature has as parent the “user” feature and has two sub-features presenting respectively the UI structure (the UI is recursively decomposed into presentation units until final widgets are reached) and the UI layout (how are final widgets laid out in order to produce the Final User Interface). The “structuralpreference” feature has two sub-features: the “changeUIElement” feature which, in its turn, defines three variants: “videoElement”, “GIFelement” and “IMGelement” features. While the “addUIelement” feature defines two variants: the “locationelement\_Yes” and the “locationelement\_No”. Finally, the “layout-preference” feature presents the disposition of the response elements (presented by the “responseElementslayout” feature). Two layout are defined, a high layout (presented by the “high\_displayDensity feature) and a low layout (presented by the “lowDisplayDensity” feature).

Aside from the definition of features models, the definition of adaptation rules is also achieved at the design phase. Adaptation rules are the means of UI adaptation at the design and at the runtime phases. A sample of adaptations rules is listed in Table 5.3 and is defined using Event-Condition-Action (ECA) rules which describe the link between context features and UI features.

For example, the adaptation rule AR1 means that if the user prefers viewing the search result displayed in a high density (in this case, the feature “High\_DiplayDensity” of the context feature model is selected), then several response elements will be displayed per line, which corresponds to the selection of the feature “GridView” of the UI feature model).

**Table 5.3** Adaptation rules

Adaptation Rule	Event	Condition	Action
AR1	Context change	High_DiplayDensity	Gridlayout
AR2	Context change	GifElement	GIF
AR3	Context change	LocationElement_Yes	LocationTextField



In the domain implementation phase, features of context and UI feature models are implemented according to the principles and requirements of Model-based Design of User interfaces [5]. The use of models, instead of aspect, component or any other implementation technology makes the SPL process more abstract and more reusable.

As a major artefact of this paradigm, the Concrete User Interface (CUI) model expresses the UI in terms of “concrete interactors”, “Concrete Interaction Objects” (CIOs) [6], or more recently “concrete interaction units” that are modality dependent (in this case, the graphical interaction modality has been selected and remains fixed) but implementation technology independent (although the graphical modality has been preferred, no reference is made towards any particular programming or markup language). Fig. 5.8 depicts how CUI model fragments could be associated to features.

The CUI model conforms to the CUI meta-model sketched in Fig. 5.9. The CUI meta-model describes the main concepts related to the graphical interaction modality. The main concepts are: the “CUIobject”, the “listener” and the “style”. The “CUIobject” class presents “CUIinteractor” and the “CUIcontainer”. The listener class handles “event” applied on the “CUI interactors” and defines “actions” to be performed. The “style” class is intended to capture all the presentation attributes for a CUI Object.

At the Application Engineering level, feature models are configured in order to produce any specific UI which is intended to be context-aware. Feature configuration is declaratively specified by selecting or deselecting features according to user requirements or in principle according to any requirement coming from the context of use. This approach is similar to MiniAba [33] where the designer interactively selects or deselects UI options by manipulating model features, this creating a new configuration that will produce a newly generated UI. After feature model configuration, a model composer tailors the CUI model which correspond to selected features. Composed CUI models/model fragments constitute the desired UI (e.g., context UI or search UI). To generate the source code, the CUI model is transformed into a Final User Interface (FUI) model according to any Model-to-code (M2C) transformation. A FUI is a representation of the UI in any programming language (e.g., Java UI toolkit) or mark-up language (e.g., HTML). To generate the FUI from the CUI, code generators techniques are hereby used.

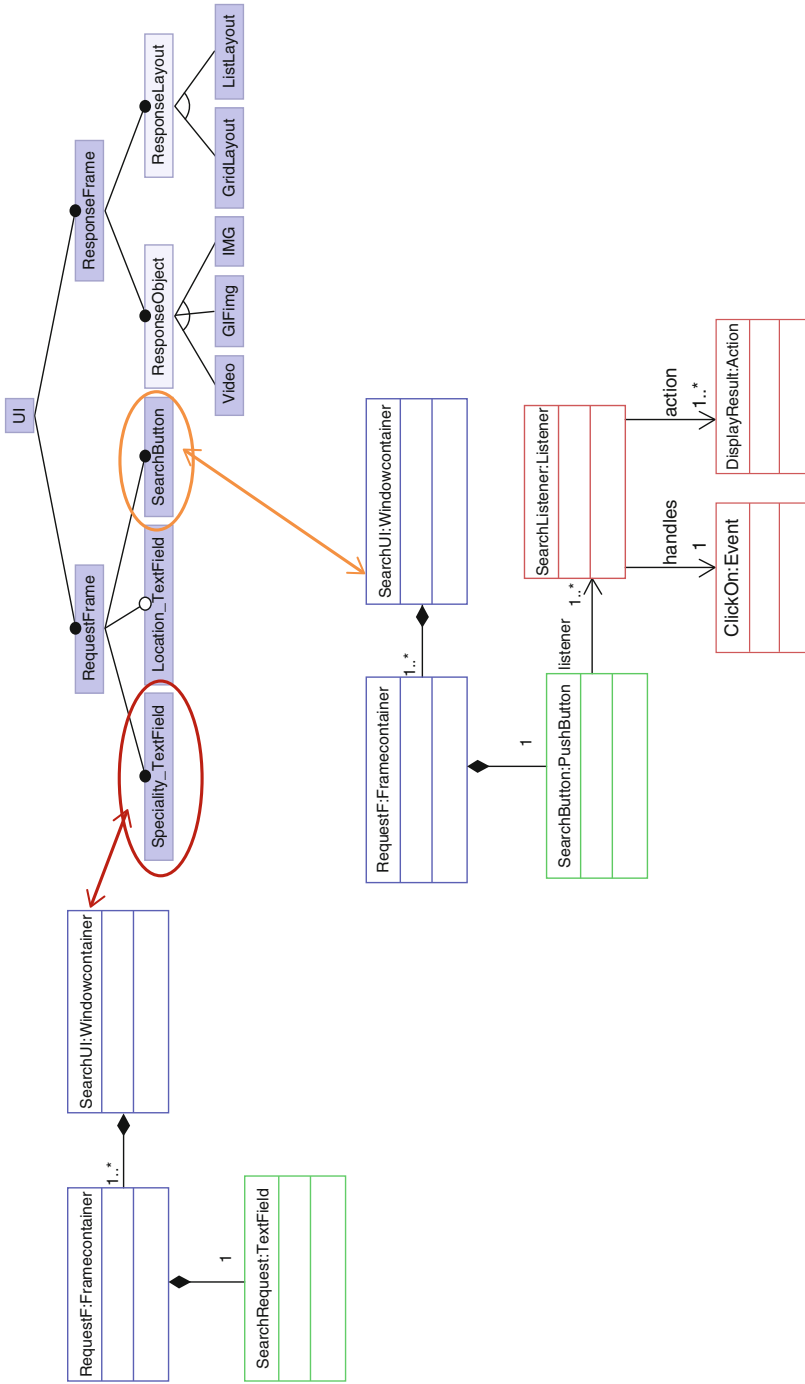


Fig. 5.8 Feature implementation

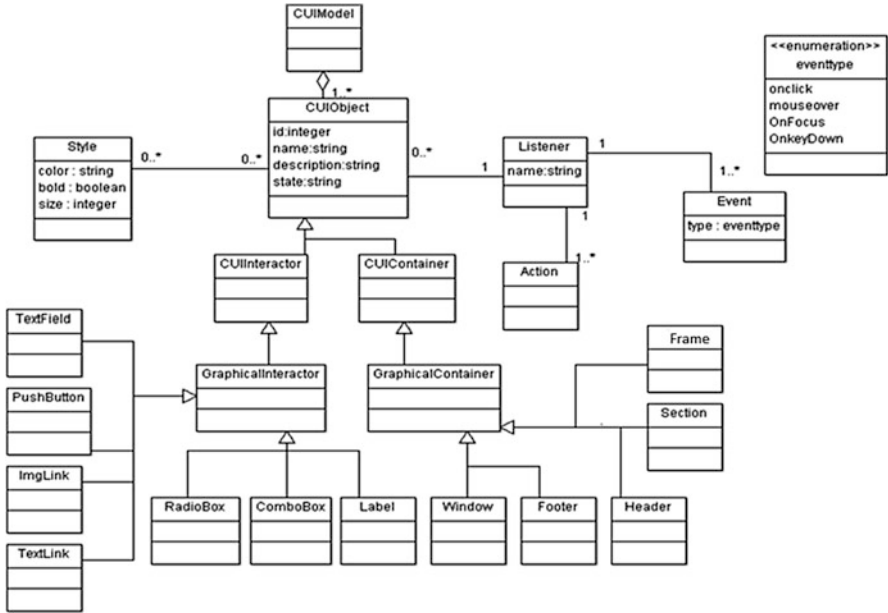


Fig. 5.9 Excerpt of the CUI Meta-model

### 5.5.2 The Runtime Adaptation Mechanism

The runtime phase or execution phase is the post-deployment phase during which the UI is running. In this phase, the end-user can set her preferences. Following these preferences settings, an adaptation mechanism is triggered based on three main components (Fig. 5.10):

- The *context manager*: is responsible for context acquisition and context storage.
- The *adaptation manager*: is responsible for the reconfiguration and the recomposition of the new UI. In addition to the acquired context data, to function properly, the adaptation manager will needs design knowledge such as: adaptations rules, the current running configuration, and the implementation artefacts.
- The UI code source generator: after its recomposition, the new UI is transformed into a FUI ready for compilation/interpretation on the device.

At a high level of abstractions and to automate the runtime mechanism, a meta-model describing the UI adaptation may be defined. Such a meta-model has to define the main adaptation concepts such as: the context of use, the user interface aspects and the adaptations rules. In this context, Fleury et al. [11] propose a meta-model which describes the runtime adaptation in self adaptive system. The proposed meta-model is generic in that the context of use was presented using contextual variables, the user interface using variants and the used artefacts through aspect technology.

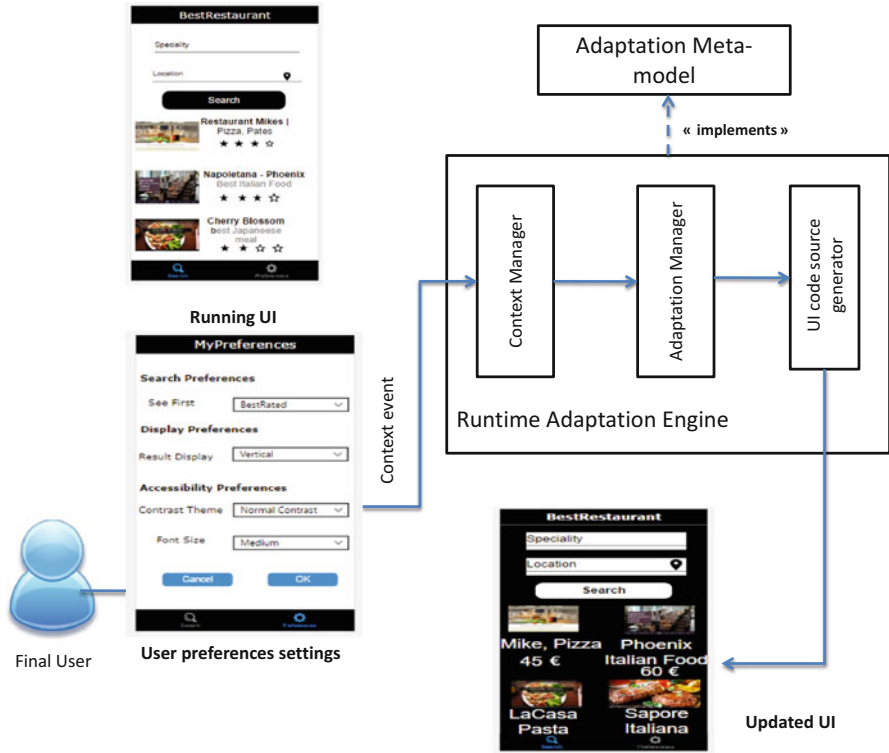


Fig. 5.10 The UI-SPL approach at run-time

## 5.6 Conclusion

In order to address the challenges posed by context-awareness in Software Product Lines (SPL), this paper firstly presented an overview of UI-SPL approaches with a specific focus on context of use consideration within these approaches. To overcome the shortcomings identified in existing literature, a more general reference schema was defined that combines the capabilities of Model-Driven Engineering (MDE) and with SPL concepts. This combination makes the SPL process more abstract and more reusable in theory, but also more difficult to produce in practice with respect to a classical development approach, which makes sense for supporting context-awareness. If the whole approach has to produce only one final UI or a small set of such UIs, probably the Model-Based Design of User Interfaces approach would not be efficient: the time required to model the entire interactive system would probably exceed the total amount of time required to develop the same system. However, when it comes to ship an interactive system exhibiting context-awareness with perhaps a significant amount of different final user interfaces, this approach could offer significant wins over the classical approach. The big win is when

there is a need for context-awareness, context-aware adaptation, the corresponding feature models only need to be updated, then been propagated to produce the corresponding final user interfaces. It is likely that modifying a few options in two feature models will require less effort than re-developing more various final user interfaces corresponding to the intended contextual changes. In order to validate the reference schema, a UI-SPL approach is defined with a design time phase that generates a UI and a runtime phase that makes the UI adaptable to any context change. The approach was illustrated with a case study that highlights the use of user preferences.

## References

1. Abrahão S, Iborra E, Vanderdonck J. Usability evaluation of user interfaces generated with a model-driven architecture tool. In: Law E, Hvannberg E, Cockton G, editors. *Maturing usability: quality in software, interaction and value*, Chapter 1. HCI Series, vol. 10. London: Springer; 2008. p. 3–32. doi=[http://dx.doi.org/10.1007/978-1-84628-941-5\\_1](http://dx.doi.org/10.1007/978-1-84628-941-5_1).
2. Acher M, Collet P, Lahire P, France RB. Familiar: a domain-specific language for large scale management of feature models. *Sci Comput Program*. 2013;78(6):657–81. doi=<https://doi.org/10.1016/j.scico.2012.12.004>
3. Arboleda H, Romero A, Casallas R, Royer JC. Product derivation in a model-driven software product line using decision models. In: *Memorias de la XII Conferencia Iberoamericana de Software Engineering (CIBSE'2009, Medellín, April 13-17) 2009*. pp. 59–72. Accessible at <http://ai2-s2-pdfs.s3.amazonaws.com/57ae/634647fcf7e610df3d106eb2a3cd0f152733.pdf>
4. Bodart F, Hennebert A-M, Leheureux J-M, Vanderdonck J. Computer-aided window identification in TRIDENT. In: Nordbyn K, Helmersen PH, Gilmore DJ, Arnesen SA, editors. *Proceedings of 5th IFIP TC 13 international conference on human-computer interaction INTERACT'95 (Lillehammer, 27–29 June 1995)*. London: Chapman & Hall; 1995. p. 331–6.
5. Boucher Q, Perrouin G, Heymans P. Deriving configuration interfaces from feature models: a vision paper. In: *Proceedings of the 6th international workshop on Variability Modeling of Software-Intensive Systems (VaMOS'2012, Leipzig, January 25–27, 2012)*. New York: ACM Press; 2012. p. 37–44. doi=<https://doi.org/10.1145/2110147.2110152>.
6. Calvary G, Coutaz J, Thevenin D, Limbourg Q, Bouillon L, Vanderdonck J. A Unifying reference framework for multi-target user interfaces. *Interact Comput*. 2003;15(3):289–308.
7. Cantera Fonseca JM (ed), González Calleros JM, Meixner G, Paternò F, Pullmann J, Raggett D, Schwabe J, Vanderdonck J. *Model-based UI XG Final Report*, W3C Incubator Group Report, 4 2010. Accessible at <http://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui-20100504/>
8. Capilla R, Bosch J, Trinidad P, Ruiz-Cortés A, Hinchey M. An overview of dynamic software product line architectures and techniques: Observations from research and industry. *J Syst Soft*. 2014;91:3–23. doi=<https://doi.org/10.1016/j.jss.2013.12.038>
9. Capilla R, Bosch J. Dynamic variability management supporting operational modes of a power plant product line. In: *Proceedings of the 6th international workshop on Variability Modeling of Software-Intensive Systems (VaMOS'2012, Leipzig, January 25–27, 2012)*. New York: ACM Press; 2012. p. 49–56. doi=<https://doi.org/10.1145/2866614.2866621>.
10. Clements P, Northrop L. *Software product lines*. Boston: Addison-Wesley; 2002.
11. Fleury M, Reverbel F. The JBoss extensible server. In: Endler M, Schmidt D, editors. *Proceedings of middleware 2003, Lecture notes in computer science*, vol. 2672. Berlin: Springer; 2003. p. 344–73. doi=[http://dx.doi.org/10.1007/3-540-44892-6\\_18](http://dx.doi.org/10.1007/3-540-44892-6_18).

12. Gabillon Y, Biri N, Otjacques B. Designing an adaptive user interface according to software product line engineering. In: Proceedings of 8th international conference on advances in computer-human interactions (ACHI'2015, Lisbon, February 22–27, 2015). International Academy, Research, and Industry Association (IARIA), 2015, pp. 86–91. Accessible at [https://www.thinkmind.org/download.php?articleid=achi\\_2015\\_5\\_20\\_20128](https://www.thinkmind.org/download.php?articleid=achi_2015_5_20_20128)
13. Garcés K, Parra C, Arboleda H, Yie A, Casallas R. Variability management in a model-driven software product line. *Avances en Sistemas e Informática* 4(2), 2007. Accessible at <http://www.bdigital.unal.edu.co/15155/>
14. Genaro Motti V, Vanderdonck J. A computational framework for context-aware adaptation of user interfaces. In: Proceedings of 7th international conference on Research Challenges in Information Science (RCIS'2013, Paris, May 29–31, 2013). Los Angeles: IEEE Computer Society; 2013. p. 1–12.
15. Goma H, Hussein M. Dynamic software reconfiguration in software product families. In: Proceedings of international workshop on software Product-Family Engineering (PFE'2003, Siena, November 4–6, 2003). Lecture Notes in Computer Science, vol. 3014. Berlin: Springer; 2003. p. 435–44. doi=[http://dx.doi.org/10.1007/978-3-540-24667-1\\_33](http://dx.doi.org/10.1007/978-3-540-24667-1_33).
16. Guerrero J, Vanderdonck J, Gonzalez J. FlowXML: a step towards designing workflow management systems. *Int J Web Engineering Technol.* 2008;4(2):163–82. doi=<https://doi.org/10.1504/IJWET.2008.018096>
17. Helms J, Schaefer R, Luyten K, Vermeulen J, Abrams M, Coyette A, Vanderdonck J. Human-centered engineering with the user interface markup language. In: Seffah A, Vanderdonck J, Desmarais M, editors. Human-centered software engineering, Chapter 7. HCI Series. London: Springer; 2009. p. 141–73. doi=[http://dx.doi.org/10.1007/978-1-84800-907-3\\_7](http://dx.doi.org/10.1007/978-1-84800-907-3_7).
18. Hubaux A, Acher M, Tun TT, Heymans P, Collet P, Lahire P. Separating concerns in feature models: retrospective and support for multi-views. In: Reinhartz-Berger I, Sturm A, Clark T, Cohen S, Bettin J, editors. Domain engineering: product lines, languages, and conceptual models. Berlin: Springer; 2013. p. 3–28. doi=[http://dx.doi.org/10.1007/978-3-642-36654-3\\_1](http://dx.doi.org/10.1007/978-3-642-36654-3_1).
19. Kramer DM. Unified GUI adaptation in dynamic software product lines. Doctoral dissertation, University of West London, London, 2005. Accessible at <http://repository.uwl.ac.uk/1270/>
20. Kramer D, Oussena S, Komisarczuk, Clark T. Using document-oriented GUIs in dynamic software product lines. *ACM SIGPLAN Notices* 49(3), 2014, pp. 85–94. doi=<https://doi.org/10.1145/2637365.2517214>
21. Logre I, Mosser S, Collet P, Riveilli M. Sensor data visualisation : a composition-based approach to support domain variability. In: Proceedings of the 10th European Conference on Modelling Foundations and Applications (ECMFA'2014, York, July 21–25, 2014). Lecture Notes in Computer Science, vol. 8569. Berlin: Springer; 2014. p. 101–16. doi=[https://doi.org/10.1007/978-3-319-09195-2\\_7](https://doi.org/10.1007/978-3-319-09195-2_7).
22. López-Jaquero V, Vanderdonck J, Montero F, González P. Towards an extended model of user interface adaptation: the ISATINE framework. In: Proceedings of engineering interactive systems 2007 (IFIP WG2.7/13.4 10th conference on engineering human computer interaction jointly organized with IFIP WG 13.2 1st conference on human centred software engineering and DSVIS - 14th conference on design specification and verification of interactive systems) EIS'2007 (Salamanca, 22–24 March 2007) Gulliksen J, Harning MB, Palanque Ph (eds). Lecture Notes in Computer Science, Vol. 4940. Springer, Berlin, 2008, pp. 374–392. doi=[http://dx.doi.org/10.1007/978-3-540-92698-6\\_23](http://dx.doi.org/10.1007/978-3-540-92698-6_23)
23. Muller J. Generating graphical user interfaces for software product lines: a constraint-based approach. In: Alt et al. (eds), Tagungsband 15. Interuniversitäres Doktorandenseminar-Wirtschaftsinformatik der Universitäten Chemnitz, Dresden, Freiberg, Halle-Wittenberg, Jena und Leipzig, Leipzig, 2011, pp 64–71. Accessible at <https://pdfs.semanticscholar.org/186d/7f0907852cbaaf798513ea1f2e347a63b342.pdf>
24. Object Management Group. IFML: the Interaction Flow Modeling Language, 2015. Accessible at: <http://www.ifml.org/>
25. Object Management Group, Query/View/Transformation. Accessible at <http://www.omg.org/spec/QVT/>

26. Paternò F, Santoro C, Spano LD. Concur Task Trees (CTT). Accessible at: <https://www.w3.org/2012/02/ctt/>, 2012.
27. Pleuss A, Hauptmann B, Dhungana D, Botterweck G. User interface engineering for software product lines: the dilemma between automation and usability. In: Proceedings of the 4th ACM SIGCHI symposium on Engineering Interactive Computing Systems (EICS'2012, Copenhagen, June 25–26, 2012). New York: ACM Press; 2012. p. 25–34. doi=<https://doi.org/10.1145/2305484.2305491>.
28. Pleuss A, Hauptmann B, Keunecke M, Botterweck G. A case study on variability in user interfaces. In: Proceedings of the 16th international Software Product Line Conference (SPLC'2012, Salvador, September 2–7, 2012), vol. 1. New York: ACM Press; 2012. p. 6–10. doi=<https://doi.org/10.1145/2362536.2362542>.
29. Pleuss A, Wollny S, Botterweck G. Model-driven development and evolution of customized user interfaces. In: Proceedings of the 5th ACM SIGCHI symposium on Engineering Interactive Computing Systems (EICS'2013, London, June 24–27, 2013). New York: ACM Press; 2013. p. 13–22. doi=<https://doi.org/10.1145/2494603.2480298>.
30. Pohl K, Boeckle G, van der Linden F. Software product line engineering. Berlin: Springer; 2005.
31. Quinton C, Mosser S, Parra C, Duchien L. Using multiple feature models to design applications for mobile phones. In: Proceedings of the 15th international Software Product Line Conference (SPLC'2011, Munich, August 21–26, 2011), Article No. 23, vol. 2. New York: ACM Press; 2011. doi=<https://doi.org/10.1145/2019136.2019162>.
32. Rosenmüller M, Siegmund N, Pukall M, Apel S. Tailoring dynamic software product lines. ACM SIGPLAN Notices. 2012;47(3):3–12. doi=<https://doi.org/10.1145/2189751.2047866>
33. Schlee M, Vanderdonckt J. Generative programming of graphical user interfaces. In: Proceedings of 7th international working conference on Advanced Visual Interfaces (AVI'2004, Gallipoli, May 25–28, 2004). New York: ACM Press; 2004. p. 403–6. doi=<https://doi.org/10.1145/989863.989936>.
34. Schmidt DC. Model-driven engineering. IEEE Computer. 2006;39(2):25–31. doi=<https://doi.org/10.1109/MC.2006.58>
35. Sottet JS, Vagner A, GarcíaFrey A. Model transformation configuration and variability management for user interface design. In: Proceedings of 3rd international conference on Model-Driven Engineering and Software Development (MODELSWARD'2015, Angers, February 9–11, 2015), Communications in Computer and Information Science, vol. 580. Berlin: Springer; 2015. p. 390–404. doi=[http://dx.doi.org/10.1007/978-3-319-27869-8\\_23](http://dx.doi.org/10.1007/978-3-319-27869-8_23).
36. Ubayashi N, Nakajima S. Separation of Context Concerns— Applying Aspect Orientation to VDM. In Talk at the 2nd Overture Workshop, FM, Vol. 6, 2006. Accessible at <https://www.yumpu.com/en/document/view/27958389/applying-aspect-orientation-to-vdm-wiki>
37. Vanderdonckt J. A MDA-compliant environment for developing user interfaces of Information systems. In: Proceedings of 17th Conference on Advanced Information Systems Engineering CAiSE'05 (Porto, 13–17 June 2005).O. Pastor & J. Falcão e Cunha (Eds.). Lecture Notes in computer Science, Vol. 3520. Springer, Berlin, 2005, pp. 16–31.