

THESIS / THÈSE

DOCTOR OF SCIENCES

Engineering Configuration Graphical User Interfaces from Variability Models

Boucher, Quentin

Award date:
2014

Awarding institution:
University of Namur

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.



PRECISE RESEARCH CENTER
FACULTY OF COMPUTER SCIENCE
UNIVERSITY OF NAMUR
BELGIUM



ENGINEERING CONFIGURATION GRAPHICAL USER INTERFACES FROM VARIABILITY MODELS

QUENTIN BOUCHER

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU GRADE DE DOCTEUR EN SCIENCES

SEPTEMBER 2014

Graphisme de couverture : © Presses universitaires de Namur

© Quentin Boucher

© Presses universitaires de Namur

Rempart de la Vierge, 13

B - 5000 Namur (Belgique)

Toute reproduction d'un extrait quelconque de ce livre, hors des limites restrictives prévues par la loi, par quelque procédé que ce soit, et notamment par photocopie ou scanner, est strictement interdite pour tous pays.

Imprimé en Belgique

ISBN : 978-2-87037-873-1

Dépôt légal: D/2014/1881/65

Jury

DR. GOETZ BOTTERWECK, LERO, UNIVERSITY OF LIMERICK, IRELAND

PROF. VINCENT ENGLEBERT, UNIVERSITY OF NAMUR, BELGIUM (CHAIR)

PROF. PATRICK HEYMANS, UNIVERSITY OF NAMUR, BELGIUM (ADVISOR)

DR. GILLES PERROUIN, UNIVERSITY OF NAMUR, BELGIUM

PROF. JEAN VANDERDONCKT, CATHOLIC UNIVERSITY OF LOUVAIN, BELGIUM

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 thesis, we propose 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. A prototype generating HTML code is proposed.

RÉSUMÉ

Par le passé, les entreprises produisaient de grandes quantités de biens grâce à la production de masse, une telle approche permettant de réduire les coûts et temps de production. Bien que cette stratégie soit toujours adaptée dans certains cas (alimentation, produits ménagers), les utilisateurs ont maintenant une préférence pour les produits personnalisés. Dans un marché de plus en plus compétitif, cette personnalisation est poussée à l'extrême par les entreprises afin de gagner des parts de marché. Ces entreprises mettent des outils appelés "configureurs" à disposition des clients et de leur personnel afin de les guider dans le choix des options du produit final.

L'expérience nous a montré que les configureurs existants sont implémentés de manière ponctuelle. Ceux-ci sont d'autant plus compliqués à mettre en place et maintenir qu'il y a des contraintes non-triviales à gérer. Par exemple, nous avons observé dans deux études de cas industrielles que les dépendances entre deux options de configuration se trouvaient dans le code de l'interface graphique (et dans de nombreux autres éléments logiciels tels que des fichiers de configuration, scripts, etc.). Cette dispersion des contraintes dans le code de l'interface graphique diminue sa maintenabilité.

Dans cette thèse, nous proposons une approche de génération d'interfaces graphiques dirigée par les modèles. Nous nous reposons sur les *feature models* pour représenter et raisonner sur les options de configuration et leurs liens complexes. Une fois le *feature model* élaboré, l'interface graphique doit également être définie. Cela inclut notamment l'intégration avec les mécanismes de raisonnement sous-jacents aux *feature models* pour le contrôle et la mise à jour des éléments graphiques. Nous présentons une architecture de type modèle-vue-présentateur pour les configureurs dans laquelle nous distinguons le *feature model* (modélisation des options de configuration), le solveur associé (raisonnement automatisé) et la présentation dans l'interface graphique. Afin de combler l'écart entre le *feature model* et l'interface de configuration, les différentes constructions du premier sont traduites en objets graphiques grâce à des transformations de modèles. Ces dernières peuvent être paramétrées grâce à des langages de vue et de rendu graphique. Un prototype générant des configureurs en HTML est également proposé.

*Feeling gratitude and not expressing it
is like wrapping a present and not giving it.*

— William Arthur Ward

ACKNOWLEDGMENTS

Being a PhD student is often considered a lonely task. However, experience showed me that it is not the case. The research results presented in this manuscript would never have seen the light of day without the different persons who supported me during the last four years.

Among these persons, I would first of all like to express my gratitude and appreciation to my supervisor, Prof. Patrick Heymans, who notably gave me the opportunity to embrace the challenge of pursuing a PhD. Patrick taught me to become more autonomous and proactive in my research through the confidence and the latitude he granted me during these years. He always had an open ear, and provided good advices on my research.

Next, I would like to thank my colleague, Gilles Perrouin, for his valuable advices and the interesting discussions about my PhD topic as well as on our common research project. He also spent a lot of time to read, improve and challenge this manuscript. I am very grateful to him for all that.

Other colleagues also played a non-negligible role. Andreas Classen, Arnaud Hubaux and Raimundas Matulevičius welcomed me in their research lab, and learned me to conduct research and write bullet-proof scientific papers. Many other members (and students) of Patrick's team contributed to the ideas presented here: Ebrahim Khalil Abbasi, Mathieu Acher, Axel Boddart, Maxime Cordy, Raphaël Michel, Germain Saval and Marco Willemart. Furthermore, nothing could have been achieved without the valuable contributions of several partner companies like OSL and Rexel, to name a few.

Special thanks also go to two other colleagues, Nicolas Genon and Fabian Gilson. We spent countless time discussing professional as well as general topics during our (numerous) coffee and lunch breaks. Nico probably hated me by the time he had to manually improve parts of the configurators I had generated but I hope he does not hold it against me anymore. Do not worry, Nico, you are next on the list of PhD graduates.

I also thank my parents who made all of this possible as well as my sister, my family, and my friends for their company and encouragement. Finally, I sincerely thank my fiancée, Lowra, for her constant support and love. Conducting doctoral research comes with its fair share of upsides and downsides and she kept finding (original) ways to give me confidence that my work was worthwhile. She has always been by my side despite the distance.

In loving memory of my grandparents...

CONTENTS

List of Figures	xvii
List of Tables	xix
List of Listings	xxi
1 CONTEXT	1
1.1 What's a Configurator?	1
1.2 Building Correct Configuration Interfaces	3
1.3 Contributions	4
1.4 Reader's Guide	6
1.5 Bibliographical Notes	7
i BACKGROUND	9
2 STATE OF THE ART	11
2.1 Feature Modelling	11
2.2 User Interface Modelling and Generation	14
2.2.1 User Interface Description Languages	14
2.2.2 Feature Models and GUIs	18
2.3 Model Transformations	20
2.3.1 Model-to-Model	21
2.3.2 Model-to-Text	23
ii CONTRIBUTIONS	25
3 SOLUTION OVERVIEW	27
3.1 Architectural Pattern for Configurators	27
3.2 Generating Views from Feature Models	31
3.2.1 Widget Selection	31
3.2.2 Breaking Out the Feature Model Hierarchy	37
3.2.3 Beautifying Generated Configurators	38
3.2.4 Putting It All Together	40
3.3 Handling of Events by the Presenter	41
3.3.1 From the <i>View</i>	41
3.3.2 Back to the <i>View</i>	42
4 LANGUAGE SUPPORT	45
4.1 Textual Variability Language (TVL)	45
4.1.1 Feature Declaration and Hierarchy	47
4.1.2 Attributes	51
4.1.3 Constraints	52
4.1.4 Structuring	52

4.1.5	TVL ₂	53
4.2	Textual View Definition Language (TVDL)	54
4.2.1	Sub-tree Selection	57
4.2.2	Partial Sub-tree Selection	58
4.2.3	Feature Selection	60
4.2.4	Attribute Selection	61
4.2.5	Grouping Views	61
4.3	Featured Cascading Style Sheets (FCSS)	62
4.3.1	Global Properties	65
4.3.2	View-specific Properties	67
4.3.3	Feature-specific Properties	69
4.3.4	Attribute-specific Properties	70
5	LANGUAGE EDITORS	71
5.1	Xtext	72
5.2	TVL Editor	73
5.2.1	Grammar	73
5.2.2	Default Infrastructure	78
5.2.3	Custom Developments	80
5.3	TVDL Editor	89
5.3.1	Grammar	89
5.3.2	Default Infrastructure	91
5.3.3	Custom Developments	93
5.4	FCSS Editor	98
5.4.1	Grammar	98
5.4.2	Default Infrastructure	100
5.4.3	Custom Developments	100
6	AUTOMATION	107
6.1	HTML Interface Generation	107
6.1.1	Architectural Overview	108
6.1.2	Queries	110
6.1.3	Templates	119
6.1.4	Handling Feature Instances	127
6.2	Presenter	129
6.2.1	Initialisation	129
6.2.2	Configuration	130
6.2.3	Finalisation	133
6.3	Summary	133
iii	EVALUATION & CONCLUSIONS	135
7	EVALUATION	137
7.1	Evaluation of TVL	137
7.1.1	Evaluation Criteria	137

7.1.2	Cases	138
7.1.3	Research Protocol	142
7.1.4	Analysis of TVL	143
7.1.5	Threats to Validity	147
7.2	Evaluation of Languages and Tools	148
7.2.1	Models	148
7.2.2	Generated Configurator	155
7.2.3	Feedback from Rexel	158
7.2.4	Lessons Learned	161
7.2.5	Threats to validity	164
7.3	Further Evaluations	164
8	CONCLUSIONS	167
8.1	Summary of Contributions	167
8.2	Limitations	169
8.3	Perspectives	170
8.3.1	Reverse-engineering	170
8.3.2	Multiple Targets	172
8.3.3	Ordering Views	172
8.3.4	Workflow Configuration	174
iv	APPENDIXES	179
A	LANGUAGE GRAMMARS	181
A.1	TVL Grammar	181
A.2	TVDL Grammar	187
A.3	FCSS Grammar	189
B	PROTOTYPE GENERATOR	191
B.1	Parser Java Class	191
v	BIBLIOGRAPHY	199

LIST OF FIGURES

Figure 1.1	Audi car configurator	2
Figure 1.2	Audi car configurator for A1	3
Figure 2.1	FM of an eVoting component	12
Figure 2.2	Tree-view of pure::variant	19
Figure 2.3	Model-to-model transformation	22
Figure 2.4	Model-to-text transformation	23
Figure 3.1	Model-view-controller architecture	28
Figure 3.2	Model-view-presenter architecture	29
Figure 3.3	An MVP architecture for configurators	30
Figure 3.4	Widget types in all the configurators	34
Figure 3.5	Interface generation process	40
Figure 4.1	Package model for the TVL meta-model	47
Figure 4.2	TVL meta-model (Core)	48
Figure 4.3	TVL meta-model (Type)	49
Figure 4.4	Deep hierarchies can be split up in TVL	50
Figure 4.5	Different ways of declaring an attribute in TVL	51
Figure 4.6	TVDL meta-model	55
Figure 4.7	Stop lists for sub-tree expressions	57
Figure 4.8	Sub-tree views refined by lists	59
Figure 4.9	FCSS meta-model	63
Figure 5.1	TVL editor generated by <i>Xtext</i>	79
Figure 5.2	TVL editor with semantic highlighting	86
Figure 5.3	TVL editor outline view	87
Figure 5.4	Quick fix in the TVL editor	88
Figure 5.5	TVDL editor generated by <i>Xtext</i>	92
Figure 5.6	TVDL syntax colouring preferences	96
Figure 5.7	TVDL editor with custom highlighting	97
Figure 5.8	TVDL editor outline view	98
Figure 5.9	FCSS editor generated by <i>Xtext</i>	101
Figure 5.10	FCSS syntax colouring preferences	103
Figure 5.11	FCSS editor with custom highlighting	104
Figure 5.12	FCSS editor outline view	105

Figure 6.1	Generation process with <i>Acceleo</i>	108
Figure 6.2	Abstract architecture of our <i>Acceleo</i> solution	109
Figure 6.3	Simplified workflow of the JavaScript presenter	132
Figure 7.1	Interview protocol	142
Figure 7.2	<i>Accueil</i> tab of the <i>HTML</i> configurator for <i>Rexel</i>	155
Figure 7.3	<i>Logement</i> tab of the <i>HTML</i> configurator for <i>Rexel</i>	157
Figure 7.4	<i>Circuits</i> tab of the <i>HTML</i> configurator for <i>Rexel</i>	158
Figure 7.5	<i>Elements</i> tab of the <i>HTML</i> configurator for <i>Rexel</i>	159
Figure 7.6	<i>Tableau</i> tab of the <i>HTML</i> configurator for <i>Rexel</i>	159
Figure 7.7	Finer-grained handling of feature instances in the <i>Logement</i> tab	160
Figure 7.8	Shared feature in the <i>Circuits</i> tab	161
Figure 8.1	Reverse-engineering process for Web configurators	171
Figure 8.2	Re-engineering process for configurators	172
Figure 8.3	Illustrative YAWL workflow for a configuration GUI	173
Figure 8.4	Project planning workflow	176
Figure 8.5	Questionnaire for the project planning workflow	177

LIST OF TABLES

Table 2.1	Existing textual variability modelling languages	13
Table 2.2	Existing user interface description languages	18
Table 2.3	M2M transformation types and their main uses	22
Table 3.1	Graphical widgets mappings	35
Table 3.2	Graphical widgets for views	39
Table 7.1	Profiles of the five participants	139
Table 7.2	Results of the evaluation of TVL	143

LIST OF LISTINGS

Listing 4.1	Partial printer software TVL model	49
Listing 4.2	Printer software TVDL model	56
Listing 4.3	Printer software grouping views	62
Listing 4.4	Printer software FCSS model	64
Listing 5.1	Header of the TVL <i>Xtext</i> grammar	73
Listing 5.2	Starting elements of the TVL <i>Xtext</i> grammar	74
Listing 5.3	Types excerpt of the TVL <i>Xtext</i> grammar	74
Listing 5.4	Features excerpt of the TVL <i>Xtext</i> grammar	75
Listing 5.5	Feature contents excerpt of the TVL <i>Xtext</i> grammar	76
Listing 5.6	Attribute declaration excerpt of the TVL <i>Xtext</i> grammar	76
Listing 5.7	Constraints excerpt of the TVL <i>Xtext</i> grammar	77
Listing 5.8	Validation of attribute names in the TVL editor	80
Listing 5.9	Registration of the scope providers for the TVL editor	84
Listing 5.10	Declaration of an highlighting style for the TVL editor	85
Listing 5.11	Header of the TVDL <i>Xtext</i> grammar	89
Listing 5.12	Starting elements of the TVDL <i>Xtext</i> grammar	89
Listing 5.13	View definition excerpt of the TVDL <i>Xtext</i> grammar	90
Listing 5.14	View expressions excerpt of the TVDL <i>Xtext</i> grammar	90
Listing 5.15	Illustrative cycle example in a TVDL model	93
Listing 5.16	Cycle verification algorithm of the TVDL editor	93
Listing 5.17	Header of the FCSS <i>Xtext</i> grammar	98
Listing 5.18	High level rules of the FCSS <i>Xtext</i> grammar	99
Listing 6.1	Query retrieving the three models	111
Listing 6.2	Query retrieving an opt feature from a group	112
Listing 6.3	Main template	120
Listing 6.4	Event listener for TVL groups represented as list boxes	130
Listing 7.1	TVL model (excl. constraints) for the Rexel case	149
Listing 7.2	TVDL model for the Rexel case	153
Listing 7.3	FCSS model for the Rexel case	154
Listing 8.1	TVL model for the project planning workflow question- naire	175

CONTEXT

1.1 WHAT'S A CONFIGURATOR?

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. In August 2013, Cyledge's configurator database [Cyledge, 2013] listed more than 900 entries categorized in 16 different industries, ranging from automotive to food including apparel. The Audi car configurator displayed in Figure 1.1 is an example. Nowadays, most (if not all) other car manufacturers provide such tools to their customers.

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.

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. The different



Figure 1.1.: Audi car configurator¹(February 2014)

steps of our Audi example are visible in the bottom part of Figure 1.1. There are six: 1.Model, 2.Engine, 3.Exterior, 4.Interior, 5.Equipment, and 6.Your Audi. In this case, a step corresponds to a “part” of the car. The numbering indicates the chronological configuration order, e.g., the model has to be selected before the engine. A step is enabled as soon as the previous one is completely configured. Such an order is not always required and, in some configurators, the user is free to switch from one step to another.

Constraint verification is another guidance mechanism. Selecting an option might, for example, require the inclusion or exclusion of another one. In our Audi example, selecting the Audi A1 model line will reduce the set of available values in Body style and Model columns as depicted in Figure 1.2. Available values can be compared with those in Figure 1.1. Many more constraints examples are available around us. Configurators should

¹ See <http://configurator.audi.co.uk/controller?next=carline-page&mandant=accx-uk>

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.

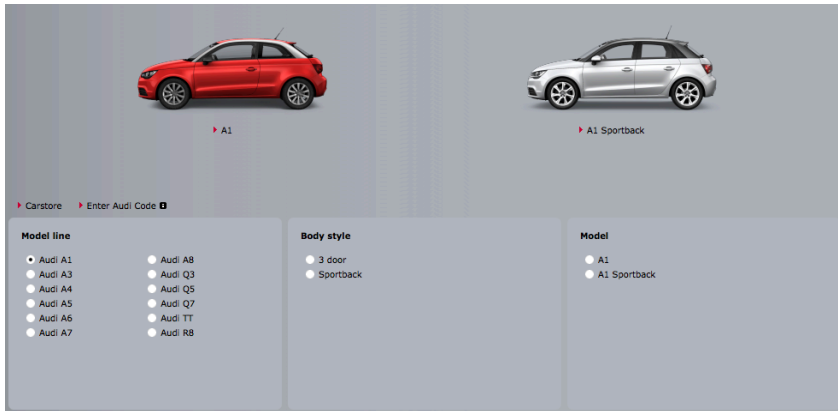


Figure 1.2.: Audi car configurator for A1

1.2 BUILDING CORRECT CONFIGURATION INTERFACES

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 [Hubaux et al., 2010a] 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. Testing such software might also prove difficult [Jin et al., 2014]. More reliable and maintainable solutions are thus needed, especially for safety-critical systems.

In this thesis, we propose a pragmatic and model-driven way to generate configuration GUIs. 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, primar-

ily in the software product line community [Kang et al., 1990]. FMs are now equipped with formal semantics [Schobbens et al., 2006], automated reasoning operations and benchmarks [Acher et al., 2012, Benavides et al., 2010b], tools [Antkiewicz and Czarnecki, 2004, Beuche, 2008, Kästner et al., 2009] and languages [Batory, 2005, Classen et al., 2011]. 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 [Batory, 2005, Benavides et al., 2010b, Hubaux et al., 2011]. 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 [Antkiewicz and Czarnecki, 2004, Beuche, 2008, Kästner et al., 2009]. But such GUIs do not consider presentation concerns and their generation process is rigid, avoiding the derivation of customised GUIs [Grechanik et al., 2004]. Furthermore existing graphical representations of FMs (e.g., FODA-like notation or tree-views) are not adapted to user-friendly configuration [Pleuss et al., 2011]. On the other hand, model-based approaches for generating GUIs simply produce the visual aspects of a GUI [Blouin et al., 2011, Blumendorf et al., 2010, Coutaz, 2010, Gomaa et al., 2005]. 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 [Boucher et al., 2010, Classen et al., 2011], a textual language for feature modelling. Transformations can be parametrised through beautification and view languages to derive specific configuration GUIs.

1.3 CONTRIBUTIONS

The contributions of this thesis are:

C1 *A model-view-presenter architecture for configurators.* Our experience shows that a lot of configurators do not separate presentation from constraint aspects. The result is that configuration options and constraints are hard to change and update since they are scattered through the GUI code. To solve those problems we propose to use a model-view-presenter architecture for configurators where the “model” part is an FM together with its solver which communicates with the GUI (i.e., the view) through a presenter. The strengths of FMs are their expressiveness and their state-of-the-art reasoning techniques based on solvers. The GUI is also derived from that FM.

C2 *A generative approach for configuration GUIs.* In our model-view-presenter architecture, the configuration GUI is derived from an FM. However, such models do not contain sufficient data to be properly rendered graphically. We thus allow to split them using views on FMs and add beautification information. The FM, its views and beautification information are used as input for GUI generation through model transformations.

C3 *A textual variability language and its editor.* While graphical FM languages may be more intuitive, they are not always adapted to large FMs involving attributes and complex constraints [Hubaux et al., 2010b]. To overcome these shortcomings, we propose TVL, a textual variability modelling language with a C-like syntax. The goal of the language is to be scalable, by being concise and by offering mechanisms for modularity. TVL is also meant to be comprehensive so as to cover a wide range of FM dialects proposed in the literature. An Eclipse editor for TVL is also proposed.

C4 *A textual view definition language and its editor.* In our approach views are used to break the hierarchy of the TVL model and make the configuration GUI independent from the FM structure. In [Hubaux, 2012], Hubaux *et al.* made use of XPath [W3C, 2010b] expressions for that purpose. However, they do not support the TVL language and do not provide access to all FM constructs. We thus propose TVDL, a textual view definition language with an XPath-like syntax. TVDL relies on TVL models and grants access to all their constructs in a more concise way than the language from which it was inspired. As for TVL, TVDL comes with an Eclipse editor developed with *Xtext* [Xtext, 2013].

C5 *A beautification language and its editor.* TVL and TVDL models do not contain the required information to be properly rendered in a GUI. They miss properties such as a label, a help text, etc. In order to preserve the separation of concerns [Tarr et al., 1999], we propose the FCSS language which allows specifying such beautification information. It is inspired by the CSS [W3C, 2008] language (Cascading Style Sheets) for HTML pages. FCSS

stands for *Featured Cascading Style Sheets*. An editor is also available for that language.

C6 *An end-to-end application of the GUI generation approach.* A prototype GUI generator has been developed that renders *HTML* configurators. The presenter is implemented in JavaScript and the solver (i.e., the model) is proposed as a Web service. That prototype has been evaluated on an industrial case study in the electrical domain.

1.4 READER'S GUIDE

The rest of the thesis is organized as follows.

Chapter 2 presents the background. It is composed of two types of information, background and state of the art. There, we give some background information about feature models and GUIs. The existing work linking feature models and GUIs is addressed. Finally, model transformations (used in the proposed approach) are introduced.

Chapter 3 gives an overview of the proposed approach. It can be decomposed into two steps. In the first one, the configuration GUI is generated based on three input models: the feature model, the view model and the beautification model. The second part is the runtime environment of the generated GUI. It is based on the model-view-presenter pattern, a variant of model-view-controller. In this pattern, the FM plays the role of “model” while the “view” role is assigned to the GUI.

Chapter 4 introduces the three input languages we developed for the GUI generator. The feature modelling language is called *Textual Variability Language* (TVL). Views are defined on TVL models using the *Textual View Definition Language* (TVDL). And the *Featured Cascading Style Sheets* (FCSS) contain beautification information, i.e., feature- and view-related information for the GUI. For each of them, we describe its syntax based on the same example. An intuition of the semantics is also given.

Chapter 5 presents the editors for the three languages introduced in Chapter 4. All three are based on *Xtext*, a framework for developing programming and domain specific languages [Xtext, 2013]. We present the editors generated by *Xtext* as well as the additional custom developments.

Chapter 6 describes a prototype implementation of the proposed approach. The generator produces *HTML* GUIs and is implemented with *Acceleo*, a model-to-text transformation tool [Obeo, 2014]. The TVL, TVDL and FCSS models are used as inputs to the tool. In the second part of the chapter, we give the general principles of the controller developed in JavaScript.

Chapter 7 validates the proposed languages and the approach based on the prototype implementation. It is decomposed into three distinct parts.

In the first one, we evaluate the proposed languages. The second part, the “industrial” one, relates our experience on an industrial application of our approach. In the last part, we explain how to properly evaluate the approach.

Chapter 8 concludes the thesis and highlights the future work.

1.5 BIBLIOGRAPHICAL NOTES

The research presented in this thesis is based upon, reuses and extends publications of the author. We list below the most relevant peer-reviewed papers published during the PhD:

Journals

- A. Classen, Q. Boucher, and P. Heymans. A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science Computer Programming*, volume 76, pages 1130–1143, 2011. (**Chapter 4**)
- P. Heymans, Q. Boucher, A. Classen, A. Bourdoux, and L. Demonceau. A Code Tagging Approach to Software Product Line Development: An Application to Satellite Communication Libraries. *International Journal on Software Tools for Technology Transfer*, volume 14, number 5, pages 553–566, 2012.

Conferences

- T. Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans. Relating Requirements and Feature Configurations: A Systematic Approach. In *Proceedings of the International Software Product Line Conference (SPLC'09)*, pages 201–210, San Francisco, USA, 2009. ACM. (**Chapter 3**)
- Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, L. Demonceau. Tag and Prune: A Pragmatic Approach to Software Product Line Implementation. In *Proceedings of the International Conference on Automated Software Engineering (ASE'10)*, pages 333–336, Antwerp, Belgium, 2010. ACM.
- A. Hubaux, Q. Boucher, H. Hartmann, R. Michel, and P. Heymans. Evaluating a Textual Feature Modelling Language: Four Industrial Case Studies. In *Proceedings of the International Conference on Software language engineering (SLE'10)*, pages 337–356, Eindhoven, The Netherlands, 2010. Springer. (**Chapter 7**)
- Q. Boucher, G. Perrouin, J-C. Deprez, P. Heymans. Towards Configurable ISO/IEC 29110-Compliant Software Development Processes for Very Small Entities. In *Proceedings of the European System, Software & Service Process Improvement & Innovation Conference (EuroSPIP'12)*, pages 169–180, Vienna, Austria, 2012. Springer. (**Chapter 8**)

- E. Abbasi, A. Hubaux, M. Acher, Q. Boucher, and P. Heymans. The Anatomy of a Sales Configurator: An Empirical Study of 111 Cases. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE'13)*, pages 162–177, Valencia, Spain, 2013. Springer. **(Chapter 3)**

Workshops

- Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a Text-based Feature Modelling Language. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, pages 159–162, Linz, Austria, 2010. Universität Duisburg-Essen. **(Chapter 4)**
- C. Gauthier, A. Classen, Q. Boucher, P. Heymans, M-A. Storey, M. Mendonca. XToF - A Tool for Tag-based Product Line Implementation. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, pages 163–166, Linz, Austria, 2010. Universität Duisburg-Essen.
- R. Michel, A. Classen, A. Hubaux, and Q. Boucher. A Formal Semantics for Feature Cardinalities in Feature Diagrams. *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'11)*, pages 82–89, Namur, Belgium, 2011. ACM. **(Chapter 4)**
- Q. Boucher, G. Perrouin, and P. Heymans. Deriving Configuration Interfaces from Feature Models: A Vision Paper. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12)*, pages 37–44, Leipzig, Germany, 2012. ACM. **(Chapters 3 and 7)**
- Q. Boucher, E. Abbasi, A. Hubaux, G. Perrouin, M. Acher, and P. Heymans. Towards More Reliable Configurators: A Re-engineering Perspective. In *Proceedings of the International Workshop on Product Line Approaches in Software Engineering (PLEASE'12)*, co-located with ICSE'12, pages 29–32, Zurich, Switzerland, 2012. IEEE. **(Chapter 3)**

Part I

BACKGROUND

STATE OF THE ART

Here, we introduce the background required to understand the contents of this thesis as well as existing approaches that we compare to ours. Feature models being the starting point endeavour, we introduce them in Section 2.1. Then, in Section 2.2, we introduce UI-related concepts and generation. Finally, model transformations used to generate configurators are introduced in Section 2.3.

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*” [Pohl et al., 2005]. Variability is typically expressed in terms of *features*, i.e., first-class abstractions that shape the reasoning of the engineers and other stakeholders [Classen et al., 2008].

Feature models were introduced as part of the FODA (Feature Oriented Domain Analysis) method 24 years ago [Kang et al., 1990]. They are a graphical notation whose purpose is to document variability. Since their introduction, FMs have been extended and formalised in various ways [Czarnecki et al., 2005, Schobbens et al., 2006] and tool support has been progressively developed [pure-systems GmbH, 2006]. The majority of these extensions are variants of FODA’s original tree-based graphical notation. Figure 2.1 shows an example of graphical tree-shaped FM that describes the variability of an eVoting component. The *and*-decomposition of the root feature (Voting) implies that all its sub-features have to be selected in all valid products. Similarly, the *or*-decomposition of the Encoder feature means that at least one of its child features has to be selected, and the *xor*-decomposition of the Default VoteValue feature means that one and only one child has to be selected. Cardinality-based decompositions can also be defined, like for VoteValues in the example. In this case, the decomposition type implies that

at least two, and at most five sub-features of `VoteValues` have to be selected. Finally, two `<requires>` constraints impose that the feature corresponding to the default vote value (Yes or No) is part of the available vote values.

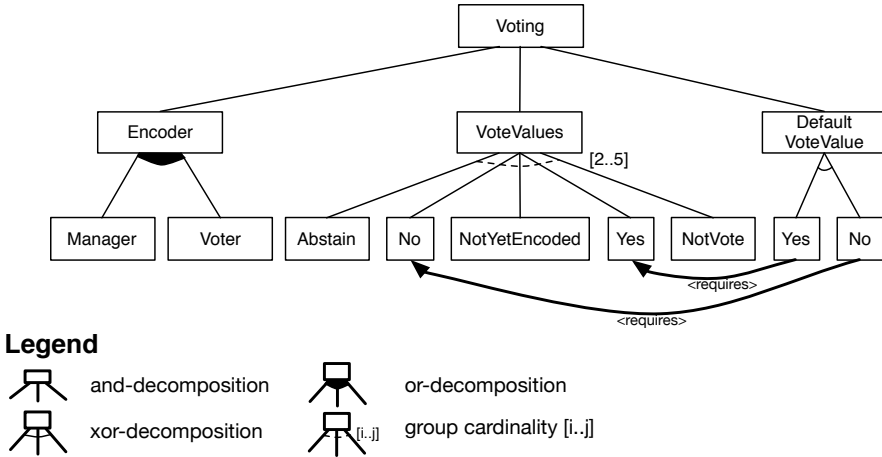


Figure 2.1.: FM of an eVoting component

Graphical FM notations based on FODA [Kang et al., 1990] are by far the most widely used. Most of the subsequent proposals such as *FeaTuRSEB* [Griss et al., 1998], *FORM* [Kang et al., 1998] or *Generative Programming* [Czarnecki and Eisenecker, 2000] 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 2.1 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 (vii) tool support.

To our knowledge, the first textual language was *FDL* [van Deursen and Klint, 2002]. It is the only language for which a formal semantics exists. It does not support attributes, cardinality-based decomposition nor other advanced constructs.

XML-based file formats to encode FMs are used by the *Feature Modelling Plugin* [Antkiewicz and Czarnecki, 2004], the *FAMA* framework [Benavides et al., 2007] and *pure::variants* [Beuche, 2008]. XML-based

Table 2.1.: Existing textual variability modelling languages

Language	Human readable	Attributes	Cardinalities	Basic Const.	Complex Const.	Structuring	Formal semantics	Tool support
FDL [van Deursen and Klint, 2002]	✓			✓			✓	
FMP [Antkiewicz and Czarnecki, 2004]		✓	✓	✓		✓		✓
GUIDSL [Batory, 2005]	✓			✓				✓
FAMA [Benavides et al., 2007]		✓	✓	✓	✓			✓
pure::variants [Beuche, 2008]		✓	✓	✓	✓			✓
SXFM [Mendonca, 2009]	✓			✓				✓
VSL [Reiser, 2009]	✓	✓	✓	✓				✓
KConfig ¹	✓	✓		✓		✓		✓

languages are not intended for human analysis, and both FAMA and pure::variants offer APIs and editors to create, manipulate or reason about models. Nevertheless, their semantics is directly implemented in the respective tools, which limits the understanding of language features and capabilities to outsiders. While our TVL editor ultimately adopts an XML representation (XMI) to store models in Eclipse, we provide language grammar and semantics in an XML-independent way [Classen et al., 2011].

Batory [Batory, 2005] proposed the GUIDSL syntax, in which the FM is represented by a grammar. The GUIDSL syntax is used as a file format of the feature-oriented programming tools AHEAD [Batory, 2005] and FeatureIDE [Kästner et al., 2009]. The GUIDSL format is aimed at the engineer and is thus easy to write, read and understand. However, it does not support arbitrary decomposition cardinalities, attributes, or the representation of the FM as a hierarchy.

The SPLOT [Mendonca et al., 2009] and 4WhatReason [Mendonca, 2009] tools use the SXFM syntax and file format. While the format uses XML for metadata and the overall file structure, its representation of the FM is entirely text-based with the explicit goal of being human-readable. It differs from the GUIDSL format in that it makes the tree structure of the FM explicit through

¹ See <http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt>

(Python-style) indentation. It supports decomposition cardinalities but not attributes.

The CVM framework [Reiser, 2009, Abele et al., 2010] supports text-based variability modelling with VSL which has support for many constructs. Attributes, however, can only be used as feature parameters and not in constraints.

KConfig is the configuration language of the Linux kernel. It is a configuration interface description language and a KConfig file can be interpreted as an FM. KConfig supports structuring with file includes. It only supports basic constraints which define presence of features. KConfig is also hard to reuse in other domains due to its specialization. For example, it provides a three-valued logic for features (yes, no, module) which is not suitable in all cases.

Bak *et al.* [Bak et al., 2010] proposed the **class**, **feature**, **reference** (Clafer) language. Even if Clafer can be used to encode feature models augmented with complex constraints, it also supports class and meta-models. This might make it too complex for people seeking to define a feature model only.

We should note that all these languages are remotely related to constraint programming, and several implementations use constraint solvers internally. Moreover, as pointed out by Batory [Batory, 2005], 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 [Knuth, 1971, Batory and Geraci, 1996]. What distinguishes FMs from constraint programming and attribute grammars is their domain-specific nature and independence from any of these technologies.

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 generator. In the second, existing work combining variability models (more exactly FMs) and GUIs is presented.

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 [Myers et al., 2000].

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* (MBUID) 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” [Gomaa et al., 2005]. 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.* [Gomaa et al., 2005] and the W3C [W3C, 2010a]. Here, we give a summary of User Interface Description Languages (UIDLs) used in MBUID. XML-based UIDLs have also been surveyed by several authors [Souchon and Vanderdonckt, 2003, García et al., 2009]. 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) [Calvary et al., 2003]. 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. A short description of each level follows:

- **Task and concepts** – Describes the UI based on user tasks and the concepts of the application domain they involve.
- **Abstract user interface** – Describes the rendering of the domain concepts and functions independently of available interactors on the target platform(s). The AUI is independent of the target platform(s) and the interaction mode.
- **Concrete user interface** – Describes the rendering of the domain concepts and functions with interactors from the target platform(s). Even though it uses elements of a target platform, the CUI is still a mock-up that runs only in development environments.
- **Final user interface** – Contains source code derived from the CUI for specific target languages. That code can then be compiled or interpreted in a run-time environment.

In other words, T&C is computing independent, AUI is modality independent and CUI is platform independent. Each abstraction level contains one model or more. For example, an AUI can be composed of a presentation model describing the interactors, their location in the UI, etc., and a dialogue model describing the behaviour.

The default development process starts with the definition of the task and concepts models. Those user-defined models are then successively refined

into AUI, CUI, and FUI. However, the user is allowed to start her development at any of the four levels. For example, she could ignore the task and concepts level and directly start with the AUI.

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* [Eisenstein et al., 2001, Puerta and Eisenstein, 2003], *Model-based lAnguage foR Interactive Applications XML* (MARIA XML) [Paterno' et al., 2009], or *USer Interface eXtensible Markup Language* (UsiXML) [Limbouurg et al., 2005]. Among all those approaches/languages, the last one is probably the most mature. It is currently being evaluated for standardization² while most others seem abandoned. UsiXML proposes a language for each model of the Cameleon framework. Some editors have been developed: GraphiXML, VisiXML, SketchiXML, IdealXML, KnowiXML, ReversiXML, and TransformiXML. Rendering also has its tool support with RenderXML and InterpiXML. However, we were not able to get access to all those tools and test them in order to get a UI directly usable by the final user. Another disadvantage of UsiXML is that it supports all interaction modes (graphical, voice, etc.) while we focus on the graphical one. Consequently, the different languages proposed by UsiXML contain irrelevant information in our case which might become cumbersome.

The *User Interface Markup Language* (UIML) [Ali et al., 2002, Helms et al., 2009] and its derivative, the *Dialog and Interface Specification Language* (DISL) [Mueller et al., 2004] make part of the second category. UIML has been defined by the OASIS consortium³ 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. Three code generators are available. UIML.net seems abandoned, JUIML has not been updated since 2009 and PyUIML (Python UIML XML Parser) has yet to mature. The *Transformation-based Integrated Development Environment* (TIDE) supports UIML code writing and Java/HTML code generation. It has been developed to help developers write UIML models and see how its final UIs are rendered. However, we were not able to find that IDE. DISL extends UIML and supports several interaction modes. It has no tool support.

The third category contains Web-application languages. Initially, XForms [W3C, 2009] was defined for HTML-XHTML documents by the W3C.

² See http://usixml.eu/sites/default/files/Issue_6_Winter_2012.V2.pdf

³ See <https://www.oasis-open.org/>

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. XHTML forms are decomposed into three models: the XForms model which defines the form and model elements, the instance data which allow to send collected data in the XML format, and the user interface which defines concepts such as *output*, *input*, *submit*, etc. XForms is not an UIDL per se but allows to define GUIs at an abstract level. Second, XICL [de Sousa and Leite, 2003] is meant to develop user interface components for browsers. New components are created by combining HTML and XICL elements. They are then translated into Dynamic HTML (DHTML). An XICL document is composed of a description of the UI using HTML and XICL constructs as well as information such as properties, events, etc. Lastly, the *eXtensible user-Interface Markup Language* (XIML) [Puerta and Eisenstein, 2002] represents interaction data for Web pages and applications at abstract and concrete levels. We distinguish three different constructions in the language: components, relations and attributes. Components are categories of interface elements. User tasks, domain objects, user types, presentation elements, or dialogue elements are examples of such categories. Relations define links between elements within the same component or across several. Attributes are properties of elements which can be assigned a value. The drawback of all those approaches is that they focus on Web GUIs while our goal is to cover more runtime environments.

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 [Kost, 2006]. There, dialogue functions are represented by GIML while the presentation is derived from XSL documents. XSLT is used to combine both information and derive a description of the corresponding final user interface. The project seems abandoned since 2004⁴. The *Multiple Device Markup Language* (MDML) supports four target environments [Johnson and Parekh, 2003]: desktop, mobile, Web and voice. The language allows to define navigation, structure and components concepts of an AUI. That information is used by an XML-based rule engine in order to tailor the FUI for a given target environment. We were not able to find the tools associated to MDML. Similarly, the *Simple Unified Natural Markup Language* (SunML) [Picard et al., 2003] supports several target environments such as PCs, PDAs or voice. It is an XML-based language to describe UI elements in an intuitive way. The *Adaptable & Mergeable User Interface* (AMUSIng) IDE provides tool support to edit SunML models and generate Swing software [Picard et al., 2003]. As for MDML, we were not able to find the IDE. Finally, in TADEUS-XML [Müller et al., 2001], a UI description is made of two parts: a presentation component and a

⁴ See <http://gitk.sourceforge.net/>

model component (or abstract interaction model). The XML-based interaction model is composed of User Interface Objects (UIOs) whose behaviour is defined by their attributes. Then, an XML-based Device Definition can transform the UIO model into a device-dependent model, e.g., by mapping abstract UIOs to concrete ones. Finally, an XSL-based model description can be derived from the second model, based on the running environment. As in [Souchon and Vanderdonckt, 2003], we were not able to find tool support.

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 [Blouin et al., 2011, Blumendorf et al., 2010]. 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.

Table 2.2.: Existing user interface description languages

Language	GUIs	Other UIs	Maintained	Tools developed	Tools available
UsiXML [Limbouurg et al., 2005]	✓	✓	✓	✓	
UIML [Ali et al., 2002, Helms et al., 2009]	✓			✓	
XForms [W3C, 2009]	✓		✓	✓	✓
GIML [Kost, 2006]	✓			✓	
MDML [Johnson and Parekh, 2003]	✓	✓		✓	
SunML [Picard et al., 2003]	✓	✓		✓	
TADEUS-XML [Müller et al., 2001]	✓			✓	

As summarised in Table 2.2, a lot of languages have been proposed but none of them meets all our needs. Most of them are either deprecated, or under development, or not available anymore. An alternative is necessary for our prototype generator. In this thesis, though recognizing the potential of a UIDL for multi-platform generation, we will focus on *HTML* Web con-

figurators in our implementation. The underlying approach is nevertheless applicable to any rendering technology.

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* [Beuche, 2008], FeatureIDE [Kästner et al., 2009] or Feature Modeling Plug-in [Antkiewicz and Czarnecki, 2004]. Those tools have a graphical interface in which users can select/deselect features in a directory-tree like interface where constraints are automatically propagated. See Figure 2.2 for an example in *pure::variants*. Several visualization techniques have been proposed to represent FMs [Pleuss et al., 2011], 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.

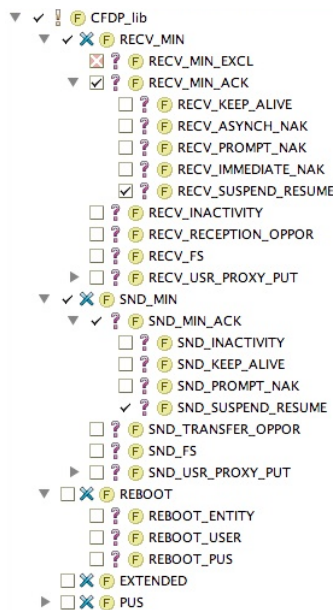


Figure 2.2.: Tree-view of *pure::variant*

An exception is the AHEAD tool suite of Grechanik *et al.* [Grechanik et al., 2004]. 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. Exam-

ples of annotations are *disp* which corresponds to the displayed name of a feature, *help* which stores help information for a feature, *tab* which defines a new tab rooted by the associated feature in the UI, *hidden* which allows to hide a feature, etc.

Pleuss *et al.* combine SPLs and the concepts from the MBUID domain to integrate automated product derivation and individual UI design [Pleuss et al., 2010]. 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.* aims 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 Vanderdonckt [Schlee and Vanderdonckt, 2004] 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 [Gabillon et al., 2013]. 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 [Quinton et al., 2011]. 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^2T^2 *Configurator* [Botterweck et al., 2009]. 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^2T^2 also presents a tree-like

view on the configuration that we believe is not suited to all kinds of end users.

2.3 MODEL TRANSFORMATIONS

In software engineering, models are traditionally used for information and communication purposes. In Model-Driven Engineering (MDE), they are further used in the development process and support code generation [Kent, 2002]. Models have to be formally defined and comply with the definition of their meta-model in order to be automatically handled by a transformation engine. Meta-models and model transformations are two core concepts of MDE. Here, we focus on the second one. We refer the interested reader to [Kleppe et al., 2003] for meta-models.

Several model transformation approaches and languages are available, each having their own characteristics. Choosing one of them depends on several parameters such as input models, required outputs, functionalities, etc.

General-purpose languages such as Java or C++ could be used to transform models. However, dedicated languages provide more powerful facilities. The complexity of model transformation is hidden by a dedicated language. Generally speaking, such languages are more efficient and accurate as they are tailored to specific needs [Greenfield and Short, 2004].

Model transformations can be classified into three categories:

1. **M2M (Model-to-Model)**: transforming input model(s) into output model(s). The OMG's QVT standard establishes a normative framework for such transformations [OMG, 2011]. QVT defines a set of languages covering a set of transformation paradigms. ATL [Jouault et al., 2008], QVTd [QVT Declarative, 2014], and QVTo [QVT Operational, 2014] are examples of M2M languages.
2. **M2T (Model-to-Text)**: transforming input model(s) into output text, e.g., source code or documentation. MOFM2T is the standard defined by the OMG for such transformations [OMG, 2008]. Aceleo [Obeo, 2014] and Jet [Jet, 2007] both implement that standard.
3. **T2M (Text-to-Model)**: transforming input text (e.g., source code or documentation) into output model(s). This category is probably the least common one. It is principally used in reverse-engineering tasks in order to extract models from existing artefacts. In this thesis, we focus on the "forward-engineering" part. Consequently, the T2M approach will not be further developed hereunder.

2.3.1 Model-to-Model

Figure 2.3, based on [Czarnecki and Helsen, 2006], gives an insight of M2M transformations with single input and output models. The Transformation engine executes Transformation definitions which refer to source and target meta-models. A model conforming to its meta-model is given as input to the transformation tool which translates it into a target model, also conforming to its own meta-model. In some cases, Transformation definitions also conform to their own meta-model. In that case, transformation definitions could also be used as source or target models, so allowing higher-order transformations, i.e., transformations of transformations.

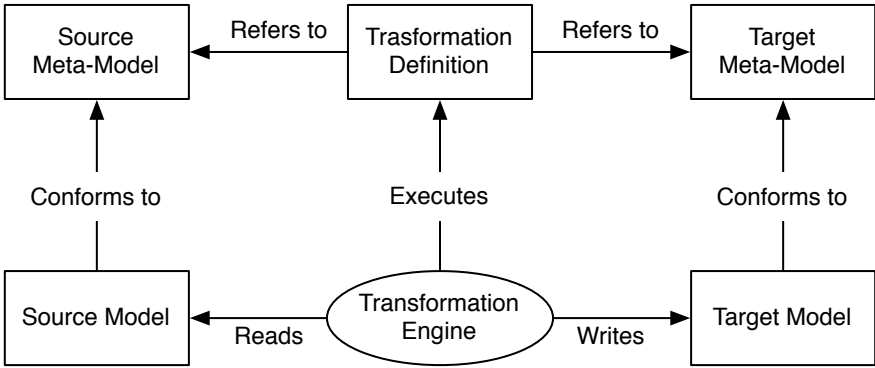


Figure 2.3.: Model-to-model transformation [Czarnecki and Helsen, 2006]

M2M transformations can be classified according to different criteria [Czarnecki and Helsen, 2006, Mens and Gorp, 2006]. One of them relates to the source and target meta-models. In *endogenous transformations*, source and target models conform to the same meta-model. Such transformations are used, e.g., to update or refine existing models in order to keep track of the changes in software. Oppositely, *exogenous transformations* use different meta-models for source and target models. One can also distinguish *vertical* from *horizontal* transformations. In the former ones, source models defined in an abstract formalism are transformed into models conforming to a less abstract formalism. That kind is used, e.g., to transform platform independent models (PIM) into platform specific models (PSM) in the context of the MDA initiative⁵. In *horizontal* transformations, source and target models have the same abstraction level. Table 2.3 summarises the main uses of M2M transformation types.

⁵ See <http://www.omg.org/mda/>

Table 2.3.: M2M transformation types and their main uses

	Horizontal	Vertical
Endogenous	Restructuring	Refinement
	Normalisation	
	Patterns integration	
Exogenous	Software migration	Generation
	Models merging	Reverse-engineering

Furthermore, several paradigms exist for model transformation languages [Czarnecki and Helsen, 2003]. We can mention the declarative one. It is based on patterns. A pattern describes which elements of the source meta-model (left-hand side, LHS) are translated into elements in the target meta-model (right-hand side, RHS). Execution of the transformation is non-deterministic in general and matches the LHS with model elements conforming to it. These elements are then replaced by newly created model elements conforming to the RHS. QVTd [QVT Declarative, 2014] is an example of declarative transformation language. Imperative transformations are close to imperative programming languages. Models are processed in a procedural way: one codes explicitly how model elements are transformed within transformation operations. QVTo [QVT Operational, 2014] fits into this category. Both declarative and imperative approaches have pros and cons. Patterns are generally small and easy to write: they can even be shown graphically as for graph transformation approaches. Yet, non-determinism can cause problems (pattern ordering) and can be slow due to the NP-completeness of the matching problem. Imperative approaches are faster but can be difficult to write and to maintain (as for long monolithic source code) in some cases. It has been acknowledged that a universal transformation language should mix both. Such languages are called *hybrid*, such as ATL [Jouault et al., 2008].

2.3.2 Model-to-Text

Figure 2.4 presents M2T transformations with a single input model. Compared to Figure 2.3, the right side is different. The generated text (e.g., source code or documentation) does not necessarily conform to a meta-model but rather to a programming language grammar or a natural language one.

The explanation for Figure 2.4 is the same as for Figure 2.3 except that the first generates text, not models conforming to a meta-model. It is also theoretically possible to define higher-order transformations. In that case, the source and target models are model transformations.

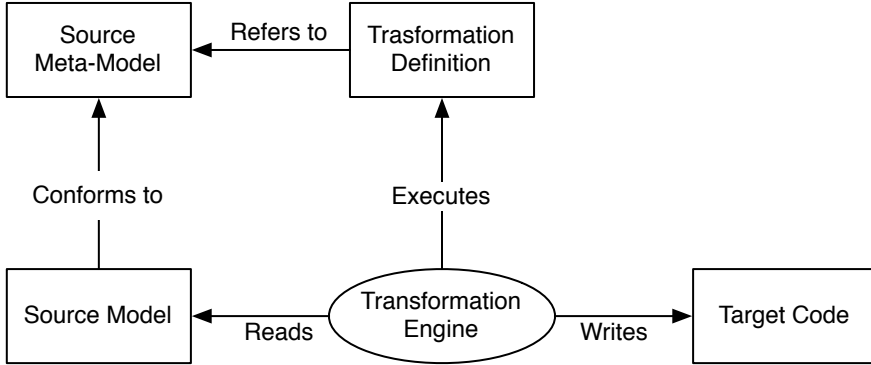


Figure 2.4.: Model-to-text transformation [Czarnecki and Helsen, 2006]

There are two kinds of M2T transformations approaches: visitor-based and template-based [Czarnecki and Helsen, 2003].

The visitor-based approach is very basic. It provides “*some visitor mechanism to traverse the internal representation of a model and write code to a text stream*” [Czarnecki and Helsen, 2003]. Jamda is an example of this approach [Jamda Project, 2014]. It is an object oriented framework which provides classes to represent UML models, an API to manipulate models and CodeWriters, a visitor mechanism.

Template-based approaches make up the majority of the model-to-text transformations. A *template* can be defined as a transformation rule. Generally, it is composed of meta-code providing access to the contents of source model(s) elements which will be replaced in the generated text, and target text. The first component has to be evaluated and depends on the contents of the input model(s) while the second is independent of the inputs.

Accessing the contents of source models is handled in different ways, depending on the transformation language. Some use Java combined with the API provided with the source model, others use declarative queries such as XPath [W3C, 2010b] or OCL [Object Management Group, 2012].

The advantage of template-based approaches is that templates are closer to the code to generate. As a consequence, they are probably easier to learn. On the negative side, they could contain “incorrect” code. By this, we mean that the generated source code could be syntactically or semantically incorrect given that the text generated by a template is not validated. In the context of documentation generation, this flexibility is an advantage.

Part II

CONTRIBUTIONS

SOLUTION OVERVIEW

As we have seen in the previous chapter, existing work does not apply on configuration GUIs. On the one hand, some FM-based configuration GUIs, typically proposed as part of feature modelling tools, implement some of the reasoning mechanisms included in configurators. But such GUIs do not consider presentation concerns and their generation process is rigid, preventing the derivation of customised GUIs. Furthermore, existing graphical representations (e.g., FODA-like notations or tree-views) are not adapted to user-friendly configuration, as encountered in real-world configurators. On the other hand, model-based approaches for generating GUIs simply produce the visual aspects of a GUI. This is not sufficient for configurators since constraint verification is paramount for their usability and performance.

In this thesis, we address the open challenge of generating a comprehensive configurator. Our approach is to combine the best of both worlds, i.e., correct management of the configuration process together with user-friendly generated GUIs. We propose a model-view-presenter architecture to design configurators (Section 3.1). One of the advantages of this architecture is the clear separation between the logical and visual concerns of a configurator. To map FMs to configuration GUIs, we propose that the different syntactical constructs of an FM are rendered as GUI elements (Section 3.2). User event management in the GUI are presented in Section 3.3.

3.1 ARCHITECTURAL PATTERN FOR CONFIGURATORS

Several architectural models have been introduced to structure modules such as the GUI in an interactive application: PAC, MVC, Oberon. The last one has become a reference model for this purpose: Slinky Arch Metamodel. 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 1970's. In December 1979 at the Xerox Palo Alto Research Laboratory (PARC), Trygve Reenskaug first

described the MVC pattern [Reenskaug, 1979a] inspired by his thing-model-view-editor [Reenskaug, 1979b] published earlier that year.

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 Figure 3.1. 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*.

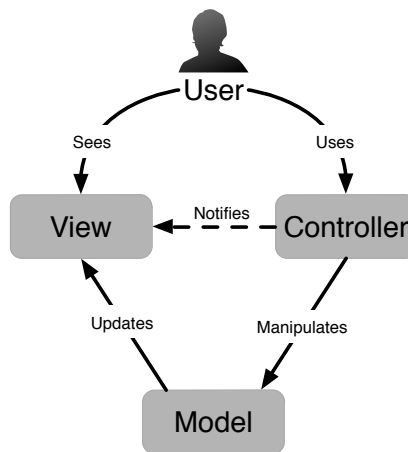


Figure 3.1.: Model-view-controller architecture

In [Burbeck, 1992], Burbeck presents two variants of the MVC pattern applied to Smalltalk-80 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 Figure 3.1). 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 (i.e., not the controller). 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 [Gamma et al., 1995] is generally used to keep the model independent from the other components. Views subscribe to be informed of the changes in the model. When such event happens, the model iterates through the list of registered observers (i.e., views) and notifies them.

We rely on an MVC variant – model-view-presenter [Potel, 1996] – 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, i.e., it in turn updates the states of the *Model* and the *View*. Centralizing the behaviour inside a single component (i.e., the *Presenter*) 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 Figure 3.2. 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 Figure 3.2, dashed lines correspond to interactions specific to the supervising controller version.

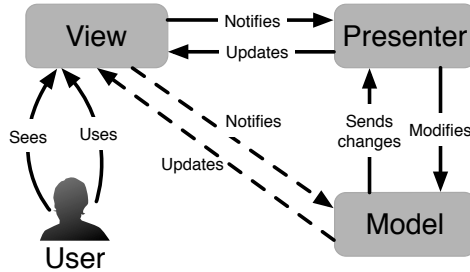


Figure 3.2.: Model-view-presenter architecture

The key idea of our approach is to separate variability reasoning at the FM level, event handling (user actions) 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 Figure 3.3).

In this thesis, we focus on the MVP-related models (shown in green in Figure 3.3) 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:** In our case, the model is an FM. The feature model is used to effectively engineer a configuration GUI. It is connected to a reasoning engine (e.g. SAT or SMT solvers), which is responsible of interactive configuration 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 (e.g., HTML, Swing, etc.) a 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, extending the applicability of the generation. This point will be further discussed in Section 3.2.
- **Presenter:** Finally, the presenter is the central point of our architecture. It listens to user actions, updates the FM (selected features, attribute values, etc.) 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.

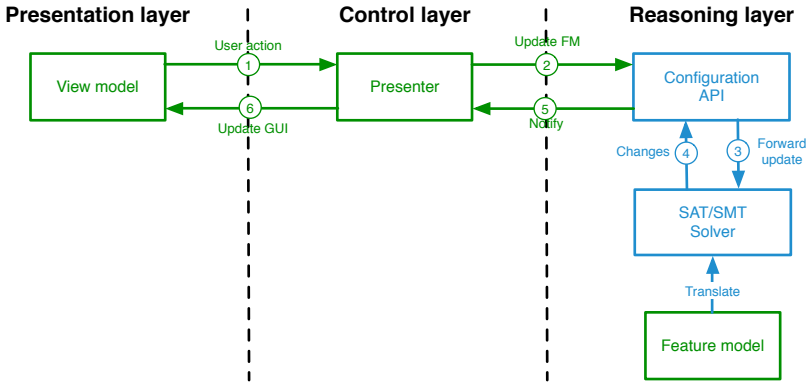


Figure 3.3.: An MVP architecture for configurators

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 (e.g., as a CNF problem). 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. In the following, we will focus on the generation of the static view.

3.2 GENERATING VIEWS FROM FEATURE MODELS

In Section 2.2.1, we have seen that some UIDLs allow to generate non-graphical user interfaces like, e.g., vocal ones. Despite the inherent usefulness of those interfaces, we focus only on graphical ones in this thesis. One of the primary reasons for this choice is that most configurators are graphical, other interfaces being more domain-specific. However, we realize that other kinds of UIs might become more widespread in the future. Consequently, the proposed solution should be easily adjustable to fit new interface representations.

This section is divided into 4 parts. First, we propose a mapping between FM constructs and GUI widgets. Then we show how to break the FM structure in the GUI. We also elaborate on the beautification of generated GUIs before wrapping up the different parts.

3.2.1 *Widget Selection*

When speaking of GUI generation, the first task that comes to mind is to translate the different FM constructs into graphical widgets. In other words, how should the different variability concerns be rendered in a configurator. For this purpose, we have analysed some existing software configurators [Abbasi et al., 2013]. More specifically, 111 Web-based configurators were investigated since they represent a significant share of existing GUIs today. They all come from Cyledge’s configurator database [Cyledge, 2013] which listed more than 900 entries categorized in 16 different industries in

August 2013. The listed configurators vary significantly, each with its own characteristics, spanning visual aspects (i.e., GUI elements) to constraint management. Although Web-based configurators do not represent the full range of software configurators, they provide valuable insights into current software configurators. 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.

In this thesis we follow the terminology provided in [UsiXML Consortium, 2012] for CUI widgets for graphical modality, currently under standardisation at the W3C:

Combo box item	Item of a combo box list
Combo box	Combination of a text label and a list box: [...] a button on the right allows showing the entire list for selection
List box	Entry allowing to select one or more items from a list
Radio button	Item representing an option in a radio box
Radio box	Entry allowing to make single selection from a number of options
Check button	Item representing an option of a check box
Check box	Entity allowing the user to make multiple selections from a number of options
Text box	Component allowing to insert text
Slider	Bar with a cursor that the user can move in order to specify a value

Spin box	Entry allowing to select a value from a set of related but mutually exclusive choices. It has an increment arrow and a decrement arrow
Label	Simple component allowing to display text
File picker	Component allowing the user to upload a file
Date picker	Component allowing the user to select a date
Colour picker	Component allowing the user to select a colour

The most significant outcome of this empirical study is that the range of graphical widgets is not very large. Actually, according to our analysis [Abbasi et al., 2013], 5 of them seem sufficient to represent most variability constructs. Figure 3.4 represents the most common widgets we found. 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, *checkboxes* 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 3.1. It is divided into 3 main categories: *Groups*, *Attribute types*, and *Features & Attributes* which will be discussed in the following paragraphs. 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 column.

The *Groups* category contains the different decompositions allowed in an FM. The first one, *and*, is a special case. Indeed, features contained in such groups have to be displayed in the configuration GUI in only two cases since mandatory options should always be selected (i.e., do not require any action on the part of a user). The first case is mentioned in Table 3.1, namely optional features defined in an *and*-decomposition. Even if they are declared inside an *and*-decomposition, the user still has the choice to not include them in her configuration. Optional features in an *and*-decomposition

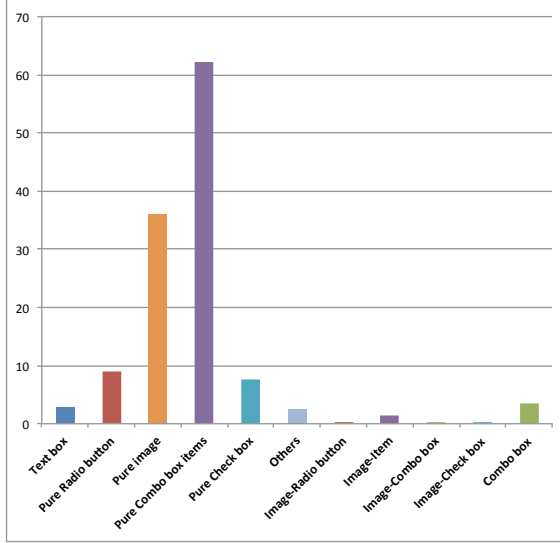



Figure 3.4.: Widget types in all the configurators [Abbasi et al., 2013]

can either be represented by a `Check` button, or a `List` box with two values (`True` and `False`) or a `Radio` box composed of two radio buttons, one for each value. Obviously, a checked `Check` button means that the corresponding optional feature should be selected. Displaying non-optional features contained in an *and*-decomposition is also required on a single condition, namely that at least one of its descendants has to be configured. By configurable descendant, we mean that the user is allowed to select or not a feature, set a value to an attribute, etc. In this way, the full hierarchy is displayed to facilitate user's understanding. Other representations like *greyed*, *pruned*, or *collapsed* [Hubaux et al., 2013] could also be of interest but are not addressed in the context of this thesis. As they heavily depend on the capabilities of the rendering language, they are not considered here. Our generic architecture can accommodate such additions easily.

The second kind of groups, *or*-decompositions, can be represented either by a `List` box or a `Check` box. There are two kinds of `List` boxes, *single* and *multiple*. As the names imply, they differ in the number of items which can be selected in the list. While *single* lists are used for optional features in *and*-decompositions introduced in the previous paragraph, *multiple* ones must be used for *or*-decompositions. This allows to comply with the semantics of such groups which, as a reminder, require to select at least one sub-feature. The other widget, `Check` box, does not enforce the selection of at least one sub-feature. This constraint will be verified by the solver itself via the con-

Table 3.1.: Graphical widgets mappings

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

figuration API. *Xor*-decompositions are represented either by a `List` box or a `Radio` box. In this case, the `Radio` box has to be a *single* one given that exactly one sub-feature has to be selected in a *xor*-decomposition. This widget thus enforces the group constraint on its own. The same advantage applies to the other widget, namely `Radio` box, as it only entitles to select a single `Radio` button in the box. *Cardinality*-decomposition can only be represented by `Check` boxes given their semantics. Here, the group semantics will also have to be enforced by the solver itself as the selected widget can not ensure the group cardinality on its own.

The different attribute types also have to be rendered in configuration GUIs. In this thesis we discuss only 4 of them, namely *integer*, *real*, *Boolean* and *enumeration*. Those 4 types cover a fifth one, *structures* which can be defined as a set of different base attribute types. *Integer* and *real* both being of number type are rendered with the same two widgets, namely `Text` box and `Slider`. The first is not a simple text box but a typed one which accepts only integer or real values, depending on the represented attribute type. This could be further improved by adding increment and decrement arrows to the `Text` box, similarly to *spin boxes*. `Sliders` represent a viable alternative. In both cases, the increment will depend on the attribute type. It should be an integer value for integer attributes and a real value for attributes of the same type. Number attributes whose domain is an enumeration are not covered by the mappings in Table 3.1. Indeed, the proposed widgets are probably not suitable in that case. Instead, we propose to use a `List` box or a `Radio` box, depending on the number of values in the enumeration. Logically, the same principle applies to *enumeration* attributes. The cut-off number of values to either select a `List` box or a `Radio` box is deliberately vague as it will depend on many factors. The last type of attribute covered in this thesis, *Boolean*, is handled in exactly the same way as optional features contained in an *and*-decomposition.

Generating a feature group or an attribute type is not sufficient per se, features contained in a group, and attributes, independently of their type also have to be displayed in the configurator. In the previous paragraphs, we have tackled the translation of group cardinalities into widgets but not the generation of the feature contained in such groups. The same observation can be made for attributes. There we presented the mappings between attribute types and their corresponding widgets. To put it in simple (and naïve) terms, in previous paragraphs we have tackled group cardinalities and attribute types, and in this one we target features and attributes on their own. Our analysis of existing Web configurators has shown that configuration options, regardless of whether they correspond to features or attributes, are almost always depicted either by a simple `Label` or an `Image`. Therefore, we propose to use those two widgets to represent features as well as attributes.

3.2.2 *Breaking Out the Feature Model Hierarchy*

In the previous section, 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 in an FM [Tarr et al., 1999] into account. Here, we do not tackle the abstraction level of the FM which we have already discussed in [Tun et al., 2009] but the constructs which might be logically linked from the user perspective even if they are not depicted as such by the FM hierarchy. Those logical groups of 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*” [Hubaux et al., 2013]. They were defined by Hubaux *et al.* to make FM-related tasks less complex in the context of Feature Configuration Workflows (FCWs). An FCW is a combination of views on an FM, each view being stakeholder-specific, and a workflow used to drive the configuration of the views. Each view allows stakeholders to concentrate on the parts of the FM that are relevant to them. Here, we go one step further and focus on the task of a given stakeholder. An FM covering stakeholder’s specific concerns only is a pre-requisite for GUI generation. Views are defined on that user-specific FM to split it into sets of logically linked information.

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 struc-

ture 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. 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.

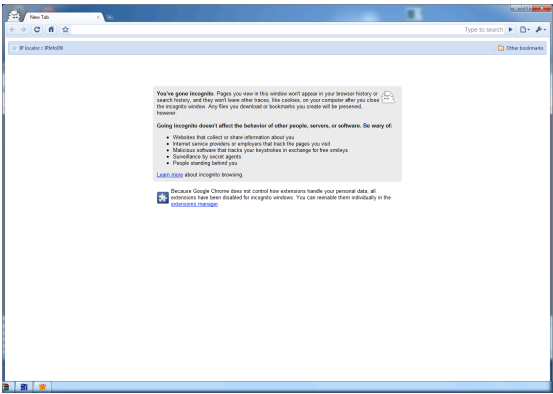
As for FM constructs, we propose a mapping between views and GUI widgets. It is summarized in Table 3.2. Each view can be depicted either as a Tab or as a Window. Tabs could be nested in other Tabs or Windows, but not vice versa.

3.2.3 *Beautifying Generated Configurators*

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. To tackle this problem we propose to add a so-called *Property Sheet*. This document is meant to store display information to beautify the GUI, by attaching a display name, help text, etc. to features, attributes and views. The property sheet links GUI properties to constructs of the FM.

Here, we propose different properties which would help to beautify the generated GUI. Those properties can be grouped into five categories, one for each kind of FM construct (attribute, group and feature), one for views on FMs, and one for properties common to the 4 other categories. The first

Table 3.2.: Graphical widgets for views

Construct	Widget	Example
View	Tab	<div>Tab 1</div> <div>Tab 2</div>
	Window	

category, attributes, is actually empty since all its properties are covered by the common category. A similar comment applies to the view category.

We propose two properties for groups. The first one is Boolean and called `container`. It allows to define a physical container (e.g., a border) for the group and its contents (i.e., sub-features). This allows to group together elements logically linked in the GUI. The second property is `default` and helps to define the default value(s) of the group. Available values are sub-features of the group only. The number of selected values for this property will depend on the cardinality of the group.

A single property is proposed for features, the others falling within the common category. It is the `optionality` property of a feature. As previously explained, this value makes sense only for optional features contained in an *and*-decomposition. For the sake of conciseness, we will not discuss this point here and we refer the reader to the first part of this chapter. In line with what has been proposed in Table 3.1, the available values for the *optionality* property are `Check button`, `List box` and `Radio box`.

As we have seen in previous sections, most FM constructs and views can be translated into several widgets. A widget property is thus required to chose a specific widget. Available widget values will depend on the corresponding element (attribute type, group cardinality, feature, view). For example, `List box` and `Check box` will be the only available values associated to the *widget* property of an *or*-decomposition, as mentioned in Table 3.1. A `label` property is also available for all constructs in order to replace its (usually)

unreadable name. Help texts allow to define additional information about the construct if its name is not self-explanatory, for example. Finally, an unavailable property may define the behaviour in cases where a construct is not available, generally due to FM constraints. Available values will be discussed in Section 3.3. Generate might also be a useful Boolean property. A *false* value indicates that the corresponding construct does not have to be rendered in the generated configuration GUI.

3.2.4 Putting It All Together

After having made the role of each element of our architecture 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 [Tarr et al., 1999] and generative techniques [Schlee and Vanderdonckt, 2004]. The base process is sketched in Figure 3.5 and relies on the notion of AUI [Calvary et al., 2003]. According to the W3C [W3C, 2014b], 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 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 be then transformed in a CUI. Depending on the required sophistication level of the interface, different combinations of views and property sheets can be envisioned.

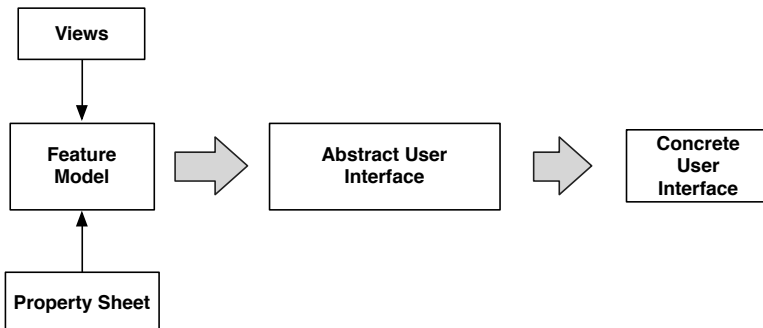


Figure 3.5.: Interface generation process

Based on the FM and the associated *Property sheet*, 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. 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.

3.3 HANDLING OF EVENTS BY THE PRESENTER

As mentioned in Section 3.1, the *Presenter* will be the same for all configurators implemented in the same CUI language. Its role is to handle the behaviour of *View* events as well as the answers from the *Model*. We now discuss those two flows of interactions in turn.

3.3.1 From the View

In our MVP architecture, a change event is sent to the *Presenter* as soon as it is detected by the *View*. The detection mechanism will depend on the chosen CUI and is not further discussed here. The event detection mechanism is a pre-requisite for target CUI languages of generated configurators. This seems not too stringent since, as far as we know, most GUI languages offer such mechanisms. For example, in HTML, such events are *onchange*, *onselect*, *onsubmit*, etc.

The *Presenter* then takes the change event over. It has to identify the FM construct which has been changed, e.g., by its ID. Consequently, the *View* has to keep track of it, either by also using this ID as its own elements' IDs, or by applying a naming convention allowing the *Presenter* to retrieve the ID of the construct in the FM. For example, the FM construct ID could be used as value for the *id* tag of its corresponding HTML widget for Web-based configurators. The way in which the mapping between GUI widgets and FM constructs is made will depend on many different factors. The target CUI language is one of them, but it might also depend on the widget type, even within the same language.

The second task of the *Presenter* is to identify the new value(s) of the changed widget. Again, this will depend on the CUI language and the specific widget. Basically, possible values are *Booleans* (for features and Boolean

attributes), *numbers* (for integer and real attributes), and *strings* (for enumeration attributes).

Once the *Presenter* has collected the ID of the FM construct together with its new value(s), it sends them to the solver associated to the chosen feature modelling language through its API. That API will not be addressed here as it has been, and always is, covered by many scientific works. The first part of *Presenter's* task is thus simple.

One final important note is *when* the constraints are sent to the *Model*. We propose two approaches. In the first, the *Presenter* directly transfers the changes from the *View* to the *Model*. This is called *interactive* setting. In the second, the *Presenter* can store the changes in the *View* and send them when the user mentions that she has finished configuring her product, e.g., via a *confirm* button. We call this *batch* mode. Both techniques are not mutually exclusive and could be used in the same configuration GUI, depending on the type of constraint, the views, etc.

3.3.2 Back to the View

Once the FM construct and the associated value passed by the *Presenter* have been handled by the *Model* (more exactly, the FM API), the list of FM constructs impacted by this change is sent back by the configuration API. This list contains pairs composed of the ID of an FM construct and its value(s) propagated by the solver. Its non-trivial task is to update the *View* to reflect those changes.

The first decision is to propagate or not the list of propagated values in the *View*. Indeed, in some cases, the validation of those changes by the user might be desirable. The user should be given the opportunity to either confirm her change knowing its consequences or discard it. In the latter case, the *Presenter* will have to reset the *View* in its previous state, i.e., before the user's intervention. This means that it will have to keep track of old values for each widget. We call it *controlled* propagation. The *Presenter* could also apply a *guided* propagation. By this, we mean that, if the solver is unable to select a value on itself, it will require the user's intervention. For example, if option A requires to select option B or C, the reasoning engine cannot decide whether B or C should be selected knowing A. Consequently, the *Presenter* requires a decision from the user. Finally, the *Presenter* can also automatically propagate the required changes to all impacted options. Those three propagation strategies are not mutually exclusive and could be applied on different widgets in the same configuration GUI, for example.

Once the impacted values have been propagated in the *View*, the next question is to decide how to handle those values in cases where the user would like to modify them. The first solution is to *prevent such a situation* by dis-

abling values corresponding to solver decisions in the configuration interface. This could be done by greying out the corresponding widget, for example. An alternative is to *hide elements which are not available and show those that are available*. This is possible only in cases where an element becomes (un)available. Finally, *changes to propagated values could be displayed in the same way as user-defined ones*. Then, if a user wishes to change it, the *Presenter* should either display an error message explaining the current value and prevent its change or allow to change it while clearly stating the user's decisions which will be modified. However, this last solution is not always feasible, especially when FM constraints are complex or the user has already made a lot of choices and changing them will completely modify her product. In all cases, an explanation mechanism should be provided by the *Model* to help the user understanding why a value has been set to a given value. This would further reduce her frustration.

The propagation "problem" could be further refined. Indeed, the handling of propagated values might depend on its origin. We identified three types of constraints in an FM:

1. **Hierarchical constraints** state that a child construct is available if and only if its parent has been selected. Oppositely, if a construct is selected, all its ancestor features must be selected.
2. **Group constraints** determine the number of features which can be selected in a group.
3. **Cross-tree constraints** cover all other constraints, generally expressed using Boolean formulae.

Hierarchical constraints might be handled in two modes: descending or ascending. In the first one, the contents of a feature are displayed if it is selected. In the other one, the user can select any FM construct at any time. Each strategy has particular benefits and drawbacks. The choice is left to the user wishing to develop a configurator for her product. In the ascending mode, the *Model* will automatically select the ancestors of a changed value. In some cases, this could prove problematic. For example, if a "disable" strategy has been chosen for solver-propagated values, the user will not be able to deselect parents. Whilst this might be acceptable for some cases, it means that a user changing her mind about a higher level option will have to unset one by one all its descendant options. Again, both strategies should be supported by the *Presenter* and left to the configurator engineer's discretion.

The question of *group constraints* has already been discussed in Section 3.2.1. The semantics of some groups is enforced by the nature of their corresponding widget in the GUI. This is, for example, the case of *Radio* boxes depicting *xor*-decompositions. Such widgets allow the user to select a single option, so

complying with the group cardinality. However, in other cases the chosen widget does not enforce the group cardinality and calls have to be made to the *Model*. In that case, one of the strategies discussed above will have to be applied. The same remark applies to the last category of constraints, *cross-tree* ones, which are not enforced by GUI widgets. In our approach, the origin of a constraint propagation is thus required from the FM API.

LANGUAGE SUPPORT

As introduced in Chapter 3, our configuration GUI generation approach requires three input languages. The first one represents the variability of the product to be configured. The two other languages depend on it as they allow to define view and beautification information on the variability model. In this chapter, we present the different languages we have defined for our approach. Although several feature modelling languages exist, they do not cover all constructs such as feature attributes, for example. Furthermore, while we have seen in Section 2.1 that FMs are the *de-facto* standard for representing the variability in the scientific community, several sources suggest that in the industrial world, in contrast, FMs appear to be used rarely. We tried to find the causes of this little industrial use and tackle them in a new variability modelling language called TVL. View definition and beautification languages are defined on top of TVL. They are called TVDL and FCSS, respectively.

In this chapter, we first introduce the TVL language on a simple example before presenting TVDL and FCSS, in turn using that same example. Grammars are deliberately left out here as they will be presented in the next chapter dedicated to tool support. Formal semantics is not considered here but discussed in Raphaël Michel’s PhD thesis (being written concurrently). Instead, we will give an introduction to its semantics.

4.1 TEXTUAL VARIABILITY LANGUAGE (TVL)

While FMs are the *de-facto* standard for representing the variability in the scientific community, our industry partners, discussions at the 2010 variability modelling (VaMoS) workshop [Benavides et al., 2010a] as well as literature reviews [Chen et al., 2009, Hubaux et al., 2010b] suggest that in the industrial world, in contrast, FMs appear to be used rarely.

One reason for this, we believe, is their lack of conciseness and naturalness when it comes to modelling realistic SPLs. Various industrial experiences have shown us that one of the most efficient tools for making FMs intuitive and concise are feature attributes [Benavides et al., 2005], that is, typed pa-

rameters attached to features similar to attributes attached to classes in UML class diagrams. Discussions with engineers also showed that the resulting diagrams are more natural and easier to understand [Hubaux et al., 2010a]. For instance, without attributes, we are forced to model alternative choices that are not further decomposed as a *xor*-decomposition. A more concise solution would be to use an enumerated attribute. The semantics is exactly the same, but the notation is much more concise. In spite of all that, the semantics of FMs with attributes is not well understood and most existing notations and tools do not support them at all [Michel et al., 2011].

Another likely reason for the difficulty of using FMs in practice is the graphical nature of their syntax. Almost all existing FM languages are based on the FODA notation [Kang et al., 1990] which uses graphs with (sometimes boxed) nodes and edges in a 2D space. *Feature attributes*, to begin with, are intrinsically textual in nature and do not easily fit into this representation. Furthermore, *constraints* on the FM are often expressed as textual annotations using Boolean operators. If they were given a graphical syntax, attributes and constraints would only clutter an FM. When working with engineers, we also observed that a graphical syntax is a psychological barrier (having to draw models is deemed tedious and cumbersome by engineers) and poses a tooling problem. Existing tools for graphical FMs are generally research prototypes and are inferior in many regards to tool support for text-based formats (e.g., text editors, source control systems, diff tools, no opaque file formats and so on).

To overcome these shortcomings, we designed TVL (Textual Variability Language), a text-based FM language. The idea of using text to represent variability in SPLE is not new [Batory and Geraci, 1996, van Deursen and Klint, 2002] but seems to be recently gaining popularity [Abele et al., 2010, Czarnecki, 2010]. 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.

An overview of the three most important high level parts of the TVL meta-model is given in Figure 4.1. Indeed, our language being quite complex, we had to start with such an overview. The main package is the one containing the Core concepts of TVL. It imports classes declared in the two other packages, namely Types and Expressions. Details about the core concepts meta-model are provided in Figure 4.2. Basically, a TVL model is composed of Record Types, Custom Attribute Types, Constants and Features. Each Record Type is composed of a set of Record Fields which have a name and a type (described by the meta-model in Figure 4.3). Custom Attribute Types

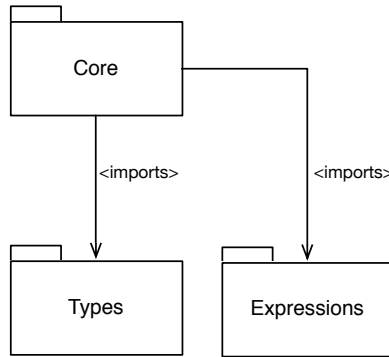


Figure 4.1.: Package model for the TVL meta-model

and Constants also have those two properties. The last one also has a value of type Expression declared in its own package (not detailed here due to its complexity). Record Types, Custom Attribute Types and Constants can be used in features. The last meta-class of a TVL model is Feature and is the most complex one. A feature can be the parent of several children features via a decomposition Operator like All of, Some of, One of or Cardinality. Features can also be optional (Opt) or have several parents if they are declared as Shared. A feature can also contain Several Attributes, each of them having a name, a type and optionally a value. Constraints are also attached to features. They can have a guard and are defined by Expressions. Finally, Data (key/value pairs) can also be declared inside features.

Those meta-models give an overview of the language. More detailed information about TVL (e.g., its syntax) will be provided in the following sub-sections using the TVL model contained in Listing 4.1. It is an excerpt of a larger FM representing the variability of a tool to prepare jobs for professional printers. The following sub-sections introduce the major parts of the language: features, attributes, constraints and structuring mechanisms.

4.1.1 Feature Declaration and Hierarchy

We will ignore the first two declarations in Listing 4.1 for the time being. The FM itself starts at line 10, with the declaration of the root feature. The feature hierarchy follows on lines 13 through 22.

The root feature, Document, is decomposed into two sub-features by an *and*-decomposition: Spine Caption and Sheet. In TVL, each decomposition is introduced by the *group* keyword (line 12), which is followed by the decomposition type. The *and*, *or*, and *xor* decomposition types were

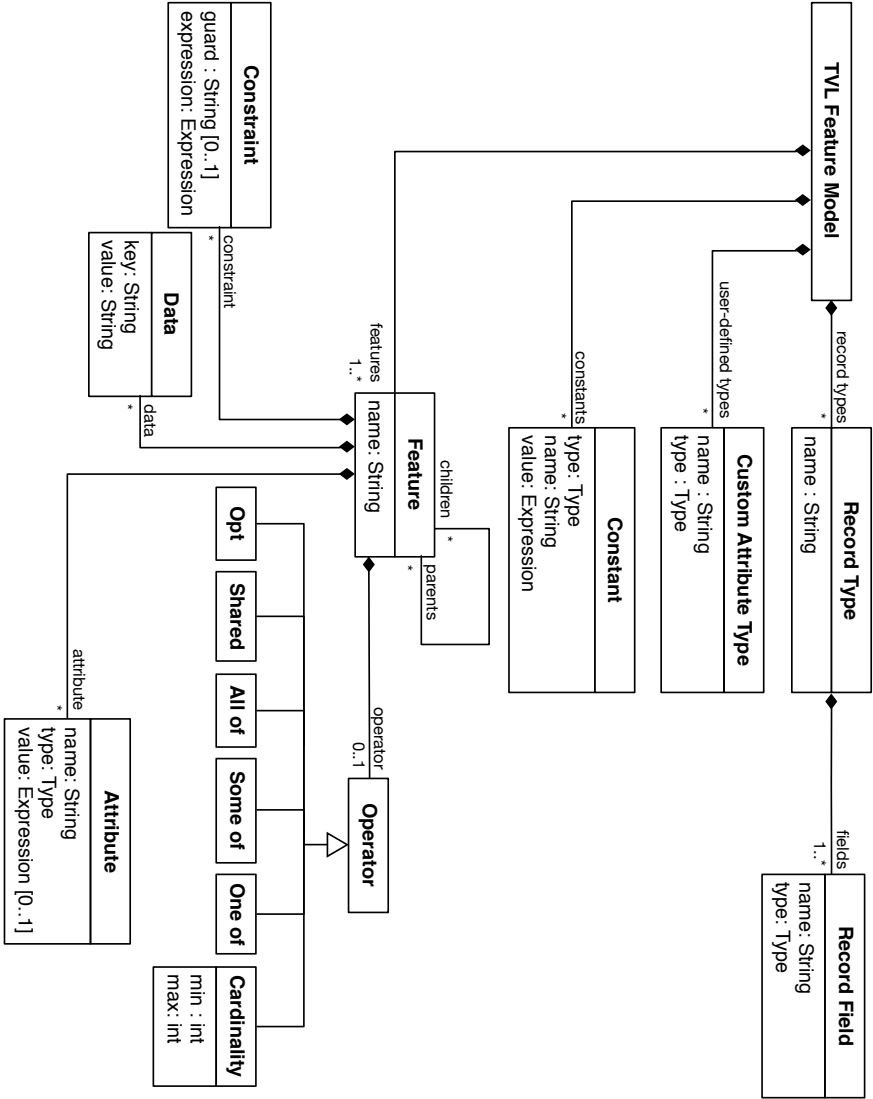


Figure 4.2.: TVL meta-model (Core)

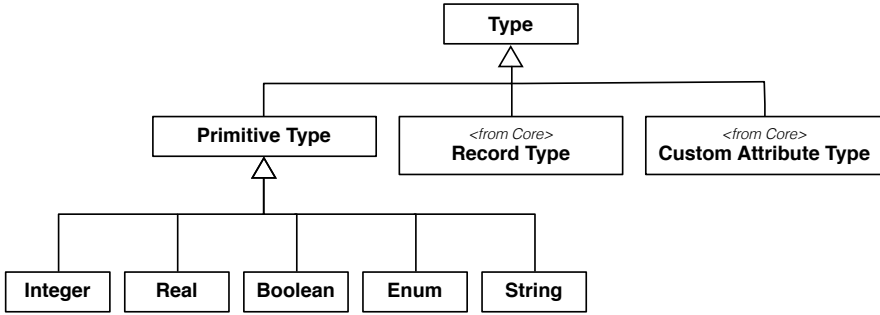


Figure 4.3.: TVL meta-model (Type)

renamed to *allOf*, *someOf* and *oneOf* in TVL. These names are inspired by [van Deursen and Klint, 2002] 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, as is done for the Sheet feature on line 13. Cardinalities can use constants, natural numbers, or the asterisk character (which denotes the number of children in the group). In our example, `group [*..*]` on line 13 is thus equivalent to `group [6..6]` or `group allOf`.

Listing 4.1: Partial printer software TVL model

```

1  // Declaring a custom type:
2  enum orientation in {horizontalLeft, horizontalRight, vertical};
3  // Declaring a structured type:
4  struct coord {
5      int x;
6      int y;
7  }
8
9  // The feature model:
10 root Document {
11     // And-decomposition of the root feature:
12     group allOf {
13         Sheet group [*..*] {
14             opt Tab, // an optional feature
15             Page,
16             opt Hole,
17             Media,
18             Staple,
19             NumberingMethod
20         },
21         opt SpineCaption
22     }
23
24     // Attribute declarations of the root feature:

```



```

25  enum type in {normal, booklet, perfectBinding};
26  enum stackMethod in {none, offset, mixed};
27
28  // A constraint:
29  Document.type == booklet -> !Sheet.Hole;
30 }
31
32 // The features SpineCaption and Hole are extended:
33 SpineCaption {
34   orientation orient;
35   ifIn: Document.type in {booklet, perfectBinding};
36 }
37
38 Hole {
39   coord position {x is 3;}
40 }

```

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 (see, e.g., line 14). Each feature of the list can declare its own children, such as *Sheet* on line 13. If each feature lists its children this way, the tree structure of the FM will be reproduced in TVL with nested braces and indentation, as shown in Figure 4.4a. 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, as in Figure 4.4b.

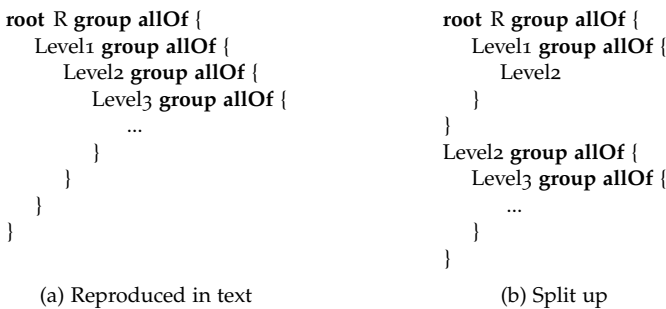


Figure 4.4.: Deep hierarchies can be split up in TVL

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 [Kang et al., 1998]), a feature name can be preceded by the *shared*

keyword, meaning that it is just a reference to a feature already declared elsewhere. (This is not illustrated in the example.)

4.1.2 Attributes

In our example, the Document feature has two attributes, indicating the types of document (line 25) and the stacking method (line 26) supported by the printer. Both attributes are of type *enum*, meaning that their value is one in a fixed set of values. This set of values is specified with the *in* keyword. The other attribute types supported by TVL are integer (*int*), real (*real*) and Boolean (*bool*). While Boolean and enumerated attributes could be encoded with features, this often results in unnecessary clutter of the diagram.

Attributes are declared like variables in C, in order to be intuitive for engineers. TVL further provides syntactic sugar to define the domain and the value of an attribute as illustrated in Figure 4.5. If the value of an attribute depends on whether its parent feature is selected or not, the declaration shown in Figure 4.5d can be used. Note that declarations in Figures 4.5b, 4.5c and 4.5d could be equally expressed by the declaration in Figure 4.5a followed by a constraint. However, syntactic sugar such as this allows the engineer to make models concise and to express her intentions clearly and intuitively.

<pre>Feature { int x; }</pre> <p>(a) Basic</p>	<pre>Feature { int x in [0..10]; }</pre> <p>(b) Domain restriction</p>
<pre>Feature { int x is 10; }</pre> <p>(c) Fixed value</p>	<pre>Feature { int x, ifIn: is 10, ifOut: is 0; }</pre> <p>(d) Conditional value</p>

Figure 4.5.: Different ways of declaring an attribute in TVL

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). This is not used in the example, but a classical example is an attribute *price* declared for each feature. The attribute has a fixed value in the leaf features: *real price is 12.34*; and is calculated for all other features: *real price is*

`sum(selectedChildren.price);`. Intuitively, this corresponds to a synthesised attribute in an attribute grammar.

4.1.3 Constraints

A constraint in the printing tool is that sheets of a booklet may not be punched. This constraint is expressed at line 29 of the `model`. As expected, the `==` denotes equality and `->` implication.

Constraints in TVL are Boolean expressions inside the body of a feature. There is also syntactic sugar for guarded constraints. For instance, if a document has a spine caption, it has to be a booklet or a perfect binding (otherwise it is impossible to write on the spine of the document). This means that the domain of the enumerated type attribute is restricted if there is a spine caption. In TVL, this can be expressed with the guarded constraint at line 35. The `ifIn:` guard works just as for attributes: the constraint only applies if the parent feature, `SpineCaption`, is selected. While `ifIn:..` is equivalent to `SpineCaption -> ..`, it is more concise and requires less “decoding” from the reader.

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}` is syntactic sugar for `e != a && e != d && ..`, which is much less readable.

4.1.4 Structuring

TVL offers various mechanisms that can help engineers structure large models. For instance, custom types can be defined at the top of the file and then be used throughout the FM. This allows to factor out recurring types and can thus reduce consistency errors. In our example FM, the type `orientation` is defined on top (at line 2) as it appears in several places in the model (e.g., at line 34). Defining a custom type in this case increases the maintainability of the model as changes will be required only in the type declaration. It is also possible to define structured types to group attributes that are logically linked. In the example, the type `coord` (at line 10) represents a pair of coordinates. It is used as a type for the `position` attribute of the `Hole` feature (at line 39) which represents the position of the hole.

There are two mechanisms for structuring. The first is the `include` statement, which takes as parameter a file path.

```
include(../some/other/file);
```

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. This mechanism was not used in our example as it is relatively small.

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 which adds constraints and attributes to the feature. In our example, the features *Hole* and *SpineCaption* are declared on lines 16 and 21, respectively. Their attributes and constraints are defined at lines 38-40 and 33-36, respectively. These could have been defined at lines 16 and 21, too, but this would have just cluttered the model.

These mechanisms allow modellers to organise the FM according to their preferences and can be used to implement separation of concerns [Tarr et al., 1999]. 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 (as in our example), 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).

Finally, features can have the same name provided they are not siblings. Qualified (or fully qualified) feature names must be used to reference features whose name is not unique. Relative names *root*, *this* and *parent* are also available to modellers.

4.1.5 TVL₂

In the previous sections, we introduced TVL as we initially defined it [Boucher et al., 2010, Classen et al., 2011]. 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 (see Chapter 7.1). Basically, two main constructs were added, string attributes and feature cardinalities.

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 TVL₂.

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. In our illustrative TVL example, each Document (at line 10) contains a single Sheet (at line 13) which is

quite restrictive. In fact, a document is composed of several sheets, each of them being configured independently. Some constraints might also exist between sheets. TVL₂ now supports so-called feature cardinalities. Their semantics is defined elsewhere [Michel et al., 2011] 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 an integer value or constant, or the asterisk character. Here, the asterisk character means that the number of feature instances is unlimited. To stick to reality, line 13 of our TVL model should thus be rewritten as follows: Sheet [1..*] group [*..*] {.

4.2 TEXTUAL VIEW DEFINITION LANGUAGE (TVDL)

The role of a view depends on its purpose. In our approach it is used to break the hierarchy of the FM and make the configuration GUI independent from its structure. For FCWs [Hubaux, 2012], they offer a way to split the FM between the different stakeholders. In other contexts of use, they will have a different role.

Independently of their role, views on an FM can be defined in two ways [Hubaux, 2012]. The first option is to enumerate all FM constructs, typically features and attributes, belonging to each view. This is called an *extensional* definition. Whilst this may be convenient for small FMs, it quickly becomes very time-consuming and error-prone for larger ones. *Intentional* definitions are an alternative. Such definitions take advantage of the FM's tree structure to select sub-parts of it. While a subset of XPath was sufficient in the context of FCWs, it is not adapted to our configuration GUI generative approach. First, XPath [W3C, 2010b] does not allow to work with TVL models as it does not allow to select all its constructs, e.g., attributes. Second, it does not meet all the requirements for the view definition language presented in Section 3.2.2.

To tackle the problems of XPath, we propose TVDL (Textual View Definition Language), a text-based view definition language 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. We chose an XPath-like language and avoided the verbosity of XML-based languages. The advantage of TVDL is to propose *intentional* as well as *extensional* view definitions. The language even allows to combine them into the same view declaration.

The meta-model of TVDL is depicted in Figure 4.6. A view model has to import a TVL Feature Model and is composed of a collection of Views which

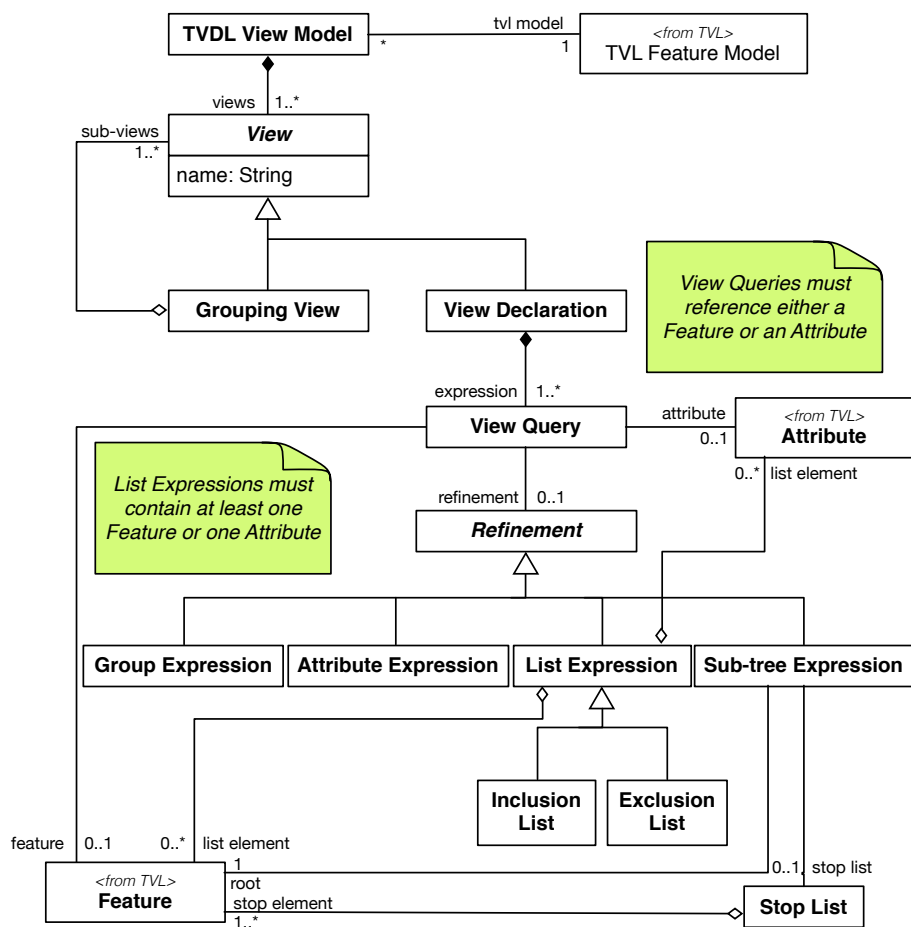


Figure 4.6.: TVDL meta-model

all have a name. A View is either a Grouping View or View Declaration. In the first case, it is composed of a set of sub-views previously declared. In the other case, a View Declaration is composed of View Queries which determine its coverage. Each View Query either refers to a TVL Attribute or a Feature and can be refined¹. A Refinement is either a Group Expression (selects groups only), an Attribute Expression (attributes only), a non-empty List Expression containing TVL attributes and/or features to include (Inclusion List) or exclude (Exclusion List), or a Sub-tree Expression. The last one has a Feature as root and can have a Stop List which is itself composed of TVL Features.

As for TVL, the TVDL meta-model depicted in Figure 4.6 gives an overview of the language structure. The detailed syntax of our view definition language will be introduced in the following sub-sections using the TVDL model contained in Listing 4.2. It is an example of a view definition model for the printing tool FM presented in Listing 4.1.

Listing 4.2: Printer software TVDL model

```

1 //Referencing the corresponding TVL model:
2 import "Printer.tvl"
3
4 //Full sub-tree selection
5 pageSubtree {Page:*}
6
7 //Partial sub-tree selection
8 partialDocument {Document:*/SpineCaption | ![position]}
9
10 //Feature selection
11 DocumentAttributes {Document: attributes}
12
13 //AttributeSelection
14 AttributesView {orient && position }
```

The first line of the TVDL model always contains the import declaration of the TVL model it refers to. The name of the file (including the .tv1 extension) is enclosed in quotation marks and preceded by the `import` keyword. In our example, the TVL FM introduced in the previous section (and subjectively named `Printer.tv1`) is referenced at line 3. In the following subsections, we discuss the other parts of the view model that actually contain views on the FM.

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. Our illustrative TVDL model contains four views, each of them having a unique name:

¹ Actually, View Expressions referencing TVL attributes cannot be refined. This point will be discussed in the following sub-sections.

pageSubtree, partialDocument, DocumentAttributes, and AttributesView. The contents are then enclosed in braces. Similarly to TVL feature extensions, there is no separator (e.g., semicolon) between the different TVDL views.

The contents of a view are discussed in the following sub-sections. Each sub-section presents a category of queries on the TVL model. The first four sub-sections correspond to the queries introduced in Section 3.2.2 while the fifth contains information related to the definition of what we call *grouping* views.

4.2.1 Sub-tree Selection

As a reminder, the goal of this kind of view is to select a sub-tree of the FM. An example is depicted at line 5 of Listing 4.2. There, we define a view named `pageSubtree` which is composed of a single sub-tree expression. Such an expression is defined using the asterisk character. It is possible to combine different expressions inside the same view. They are separated by the `&&` symbol. See line 14 for an example. The `pageSubtree` view covers a single sub-tree of our example FM. The Page feature is the root of this sub-tree which includes all its children, grand-children, etc. elements (features and attributes). In this snippet of the complete FM, it includes Page only as it does not have any child. However, in the complete model, this feature is further decomposed and contains a group as well as attributes.

A so-called *stop list* can be defined to determine the branches of the FM which are not covered by the view expression. Figure 4.7 contains an example of a sub-tree view from which one branch has been pruned, namely B. Figure 4.7a contains the TVL FM and Figure 4.7b, the associated view. B and all its sub-features (H and I) are not covered by the view. The branch is thus pruned before the stop list element, i.e., it is not included. Conversely, features R, A, C, D, E, F and G are all covered by this view expression. A stop element is a TVL feature name or its (fully) qualified name preceded by the slash character. `"/R.B"` is equivalent to the stop element defined in our example. A stop list is composed of at least one stop element.

With this kind of query, it is possible to select the full FM. This can be done via a sub-tree expression which has the FM root feature as root and has an empty stop list. For example, `"Document:*`" covers the full FM of our running example.

By selecting a full sub-tree in the context of configuration GUI generation, the user expresses her agreement with the structure defined in the FM. This hierarchy will be directly rendered in the GUI. However, semantics might depend on what purpose TVDL is used for.

<pre> root R group allOf { A group oneOf { D group allOf {F,G}, E }, B group someOf {H, I}, C } </pre>	<pre> import "exampleFM.tvl" stopListView {R:*/B } </pre>
(a) TVL FM	(b) Sub-tree view with stop list

Figure 4.7.: Stop lists for sub-tree expressions

4.2.2 Partial Sub-tree Selection

In this view category, a partial FM 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 `partialDocument` view declaration at line 8 of the TVDL model introduced earlier is an example of partial sub-tree selection. There, the search space is defined by the sub-tree with `Document` as root and excludes the `SpineCaption` branch. It covers all constructs of our TVL model example with the exception of `SpineCaption` and its attribute, namely `orient`. The difference with full sub-tree views is the filter added to the partial sub-tree selection. This filter starts with the pipe character and is followed by a refinement expression. In our example, it is an exclusion list which contains a single element, the position attribute. The list is delimited by square brackets and the exclamation mark indicates that all its elements will be excluded. The `partialDocument` view thus covers `Document`, `Sheet`, `Tab`, `Page`, `Hole`, `Media`, `Staple` and `NumberingMethod` features as well as type and `stackMethod` attributes, not position as it is included in the exclusion list.

Three different sub-tree refinements exist. They all start with the pipe character.

The first one, lists, has already been introduced in our example. 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 in-

side 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. Figure 4.8b introduces some examples of sub-tree views refined by a list. There, the first and third sub-tree views are refined by an exclusion list while View2 is refined by an inclusion list. The coverage of View1 is the set of all features contained in the TVL FM (and their groups) given that all attributes have been excluded. In other words it covers features R, A, B, C, D, E, F and G, and the three groups but not attributes a, b and r. View2 covers all groups (three in total) as well as feature D and its contents, i.e., the a integer attribute. Finally, the last view covers features R, B, C, D, E, F and G, attributes a and r, and two groups (R and B). Excluding a feature, like A in View3, will exclude its group and attributes, but not its sub-features (D and E). Sub-tree stop lists introduced earlier should be used for this purpose.

<pre> root R group allOf { A group oneOf { D, E }, B group someOf {F, G}, C } B { bool b;} D { int a;} G { real r;} </pre>	<pre> import "exampleFM.tvl" View1 {R:* ![attributes]} View2 {R:* [groups, D]} View3 {R:* ![A,b]} </pre>
(a) TVL FM	(b) Views with refinement lists

Figure 4.8.: Sub-tree views refined by lists

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. “Document:* | attributes” thus covers all attributes (and sub-attributes) of our running example, namely type, stackMethod, orient and position. 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 char-

acter. “Document:* | attributes | ![orient,position]” is a valid view definition which covers all attributes of our running example with the exception of *orient* and *position*. Note that “Document:* | [orient,position]” is equivalent (i.e., has the same coverage) to “Document:* | attributes | [orient,position]”, so making the refinement of the *attributes* keyword by an inclusion list less relevant.

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. For example, “Document:* | groups” covers all groups (i.e., two) of our running example. Again, 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. For example, “Document:*/Sheet|groups|![Document]” has an empty coverage. The group of Sheet is not covered by the sub-tree expression as it is in its stop list, and the other group (contained in Document) is in the exclusion list. Contrarily to attribute expressions, the refinement of group expressions with inclusion lists is not redundant with simple list refinements. For example, it is harder to define a view equivalent to “Document:* |groups| [Document]” with a list refinement. “Document:*/Sheet | [groups]” may be an option but is more complicated to find in complex FMs.

4.2.3 Feature Selection

In TVDL, it is also possible to select a single feature in a view. For example, “DocumentView {Document}” defines a view named DocumentView which covers a single feature, namely Document, and all its contents (i.e., its group and attributes). In this specific case, the view covers feature Document, its group as well as *type* and *stackMethod* attributes. Similarly to partial sub-trees, it is possible to refine 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. The coverage of a view defined as “Document:group” just contains Document’s group, not the Document feature itself. As illustrated at line 11 of Listing 4.2, it is also possible to select all attributes of a feature with the *attributes* keyword preceded by the colon character. In our running example, the DocumentAttributes view covers the *type* and *stackMethod* attributes. Again, the Document feature itself is not covered. 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. For example, “Document:[attributes]” covers the Document feature as well as `stackMethod` and `type` attributes. Contrarily to the example at line 11 of Listing 4.2, this view expression covers the Document feature itself. Feature view expressions refined by an exclusion list will also include the feature itself plus its contents which are not covered by the elements contained in the list. For example, “Document:![attributes]” covers the Document feature and its group, not its two attributes. We can note that “Document: [attributes]” is equivalent (i.e., covers the same elements) as “Document:![group]”. This statement is not specific to our example and can be generalised to any feature of any TVL FM given that, in our variability modelling language, a feature contains only attributes and a group.

4.2.4 Attribute Selection

The last kind of query, 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. As illustrated at line 14 of Listing 4.2, an attribute query just refers to the TVL ID of an attribute in the imported FM. There, we also illustrate another construct supported by the TVDL language, the ability to combine queries in a single view. The `AttributeView` is composed of two attribute queries separated by the `&&` symbol. This operator allows to combine queries. In our example, combined queries are of the same type (attributes) but can also be used to compose different kinds. The coverage set of `AttributeView` is composed of two attributes, `orient` and `position`.

4.2.5 Grouping Views

In the previous paragraphs, we have introduced the available queries. Those queries can even be combined to declare more complex views using the `&&` operator. Even if all those constructs should be expressive enough, we introduced another kind of view, so-called *grouping views*. As the name implies, they allow to group different views together. These may be view declarations or grouping views, so allowing to define a kind of view hierarchy. In order to facilitate the distinction between the two kinds of views, the IDs of grouping ones start with the dollar sign. The code excerpt of Listing 4.3 introducing the `GroupingView` grouping view at line 2 could be appended to the TVDL model introduced earlier.

In this example, `GroupingView` combines two view declarations using the `&&` operator, namely `DocumentAttributes` and `AttributesView`. Its coverage is the union of the coverage sets of the contained views, type, `stackMethod`, `orient` and `position` attributes in this case.

Listing 4.3: Printer software grouping views

```

1 //Grouping view
2 $GroupingView {DocumentAttributes && AttributesView}
3
4 //View declaration equivalent to the grouping view
5 EquivalentView {Document:attributes && orient && position}
```

A view covering the same set of TVL constructs has been defined at line 5. Basically, `EquivalentView` is defined as the combination of all queries defined in the view declarations grouped by `GroupingView`. This example helps to illustrate the fact that grouping views are syntactical sugar. Everything expressed with grouping views could be done with view declarations, but in a more complex way. Their purpose is to help the designer to structure her model and facilitate its writing. She might want to define views of elements which are, in her opinion, logically linked. Then, she would combine those “simple” views together. The latter could in turn be combined, and so on. The emerging hierarchy should help her easily find her way through the TVDL model which can otherwise quickly become complex.

4.3 FEATURED CASCADING STYLE SHEETS (FCSS)

Information contained inside the variability (TVL) and views (TVDL) models could be used directly to generate configuration GUIs. However, the result would be rather rough. All group decompositions of a given type (*and*-, *or*-, *xor*-, or *card*-decompositions in TVL) would be represented by the same pre-defined widget. Similarly, all attributes of a given type (*int*, *real*, *bool*, *enum*, or *struct* in TVL) would be depicted by the same widget. More importantly, the ID of each TVL construct would be used as label in the GUI. Similar problems could arise with TVDL models, especially with view labels and widgets, but with a reduced impact. The root cause of all those problems is that TVL and TVDL models do not contain all the necessary information. A solution had to be found to address this expressiveness problem.

A first solution is 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

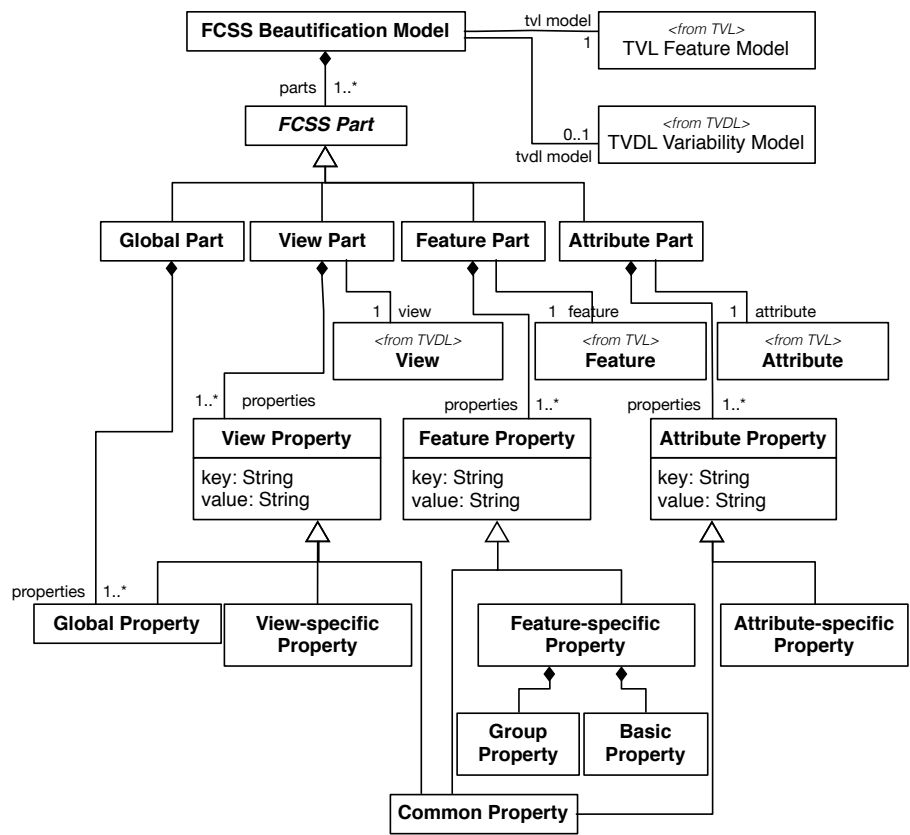


Figure 4.9.: FCSS meta-model

TVDL languages independent of the GUI generation process in order to preserve the separation of concerns [Tarr et al., 1999]. 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) [W3C, 2008] 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.

Figure 4.9 gives an overview of the FCSS language using its meta-model. 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 Part, View Part, Feature Part and Attribute Part. The Global Part gives beautification information (in Global Property) which should be applied by default. That information can then be refined by the three other kinds of parts. A View Part refers to a TVDL View and contains several properties, all represented by a key-value pair. View Properties are either View-specific ones, the same as the Global Properties or properties common to View, Feature and Attribute parts. Feature Parts (resp. Attribute Parts) define beautification information for a specific TVL feature (resp. attribute). Feature-specific and Attribute-specific properties both cover Common ones. They also declare their own properties. Feature-specific Properties can be further refined as they are composed of properties for the feature itself (Basic Property) and properties dedicated to its group (Group Property).

In what follows, we illustrate some of those concepts of TVDL based on our printing software running example. Similarly to TVL and TVDL, the formal syntax is deliberately left out (see Chapter 5). The FCSS model visible in Listing 4.4 could be applied to the TVL and TVDL models introduced in the previous sections.

Listing 4.4: Printer software FCSS model

```

1 //Referencing the corresponding TVL model:
2 import "Printer.tvl"
3 //Referencing the corresponding TVDL model (optional):
4 import "Printer.tvd1"
5
6 //Global properties definition section
7 .{
8     optFeature:checkbox;
9     view:tab;
10 }
11
12 //View-specific properties section for "DocumentAttributes"
13 $DocumentAttributes {
14     label: "Properties of the document";
15     help: "This tab contains all properties of the document";
16     generate: true;
17     unavailable: greyed;
18 }
19
20 //View-specific properties section for "GroupingView"
21 $GroupingView {
22     view:window;
23 }
24
25 //Feature-specific properties section for "Sheet"
26 Sheet {
27     help:"Properties of a sheet";

```

```

28     unavailable: hidden;
29     //Group properties
30     group {
31         label: "Components of the sheet";
32         container: true;
33     }
34 }
35
36 //Feature-specific properties section for "Hole"
37 Hole {
38     opt: listbox;
39 }
40
41 //Attribute-specific properties section for "orient"
42 #orient {
43     label: "Orientation";
44     help: "Orientation of the spine caption";
45     widget: radiogroup;
46 }

```

Similarly to TVDL, the first line of an FCSS model imports the TVL model it refers to (see line 2). Then, a TVDL model can be imported. Contrarily to the first import, the second one is not mandatory. Indeed, in our approach, a TVDL model is not required. Its absence means that the user agrees with the hierarchy defined in the TVL model. This hierarchy will be rendered in the configuration GUI. In our example, a TVDL model randomly named “Printer.tvdl” is imported at line 4. As for TVDL, imported models are enclosed in quotation marks (including their *.tol* or *.tvd* extension).

The different FCSS entries are then defined throughout the model. We have identified four categories of properties, namely *global*, *view-specific*, *feature-specific* and *attribute-specific*. The *global* category covers all constructs while *feature-* and *attribute-specific* ones cover a single construct. They will be discussed in turn in the following sections.

4.3.1 Global Properties

Global properties definition sections start with the dot character and are, like the three other categories, delimited by curly braces. See lines 7-10 of Listing 4.4 for an example. 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 (e.g., `optFeature` at line 8 of our example), and a value (checkbox) 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.

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. The latter have been defined in Chapter 3 and will not be discussed here for the sake of conciseness. *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 list-box 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*. It comes from some choices we made in TVDL and, consequently, has not been introduced in Chapter 3. 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 Chapter 3, 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. For example, at line 8 of our FCSS example in Listing 4.4, we state that optional features will be depicted by check boxes in the configuration GUI. Two optional features exist in our TVL model, *Tab* and *Home*. On the one hand, the *Home* feature is refined at lines 37-39 (see following sections for feature properties). There, the *opt* property indicates that the optional characteristic of the feature has to be rendered using a Boolean-valued list box. This property overwrites *optFeature* defined at line 8. On the other hand, the *Tab* feature is not refined. As a consequence, the value of the global attribute will be used to generate a check box for that feature.

4.3.2 View-specific 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. In our example, we have defined two of them. One for *DocumentAttributes* from line 13 through line 18 of Listing 4.4, and one for *GroupingView* (lines 21-23). 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. An example of view-specific definition section dedicated to the `DocumentAttributes` is visible at lines 13-18 of Listing 4.4. There, *label*, *help*, *generate*, and *unavailable* properties are set. The generated tab (inherited from the global attribute view at line 9) will thus be labelled "Properties of the document" and its help text will be "This tab contains all properties of the document". If, for any reason, the view is not available, its contents will be greyed out. Note that if the *generate* property is set to *false*, other properties make no sense anymore given that the view will not be rendered in the configuration GUI.

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 in the previous section. The fourteen 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* properties thus only make sense for *grouping views*. For example, defining a *view* property inside the `DocumentAttributes` view block (lines 13-18 of Listing 4.4) is useless given that it contains no sub-view. Yet, it makes sense to define it for `GroupingView` (lines 21-23) as it contains two sub-views, namely `DocumentAttributes` and `AttributesView`. Both views will be generated in separate windows (so overriding the global *view* property defined at line 9). 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.

4.3.3 Feature-specific 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. See the FCSS entry referencing the Sheet feature of our printing tool TVL model example (lines 26-34 of Listing 4.4).

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. In our FCSS example, we have defined the help text (see line 27) and set the unavailable property to hidden (line 28). Given that a *label* property has not been defined, the TVL feature ID should be used in the generated configuration GUI. Here, we considered that the feature ID (i.e., Sheet) was speaking for itself.

Four properties really specific to TVL features are available. *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. In our FCSS example, it has not been defined for Sheet which is not optional (lines 26-34), but for Hole (line 38). 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. An example of group properties is visible from line 30 through line 33 of our FCSS example in Listing 4.4. There, we define a label (line 31) for our group which will be enclosed inside a container (*container* set to true at line 32). In this case, setting the *widget* property is useless as the group is an *and*-decomposition (single value available). Similarly, defining a *default* sub-feature makes no sense given that all features have to be selected.

4.3.4 Attribute-specific 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 *orient* example is visible from line 42 through line 46 of Listing 4.4.

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. In our illustrative FCSS model, we have defined a label (line 43), a help text (line 44) and set the *widget* property to *radiogroup* (line 45).

LANGUAGE EDITORS

The three languages presented in Chapter 4 are so-called *Domain Specific Languages* (DSLs). Such languages support the user with a language designed for specific tasks. While several language-agnostic tools like text editors, source control systems, diff tools, etc. exist, language-specific tool support is essential to make other people adopt those languages [Fowler, 2005]. Language-specific tooling facilitates reading, writing, browsing, navigating, searching, editing and code comparison for developers.

Originally, we developed TVL syntax highlighting plug-ins for some text editors: *Notepad++* for Microsoft Windows, *Smultron/Fraise* and *Textmate* for Apple Mac OS, and *gedit* for Linux. All of them are available on the TVL Web page¹. While this provides tool support to users, it is still basic. More advanced tool support should be made available like Textual Language Workbenches as suggested by Martin Fowler [Fowler, 2005]. Following his opinion that “*for a wide adoption of DSL in day to day developments, IDEs for DSL should be easy to create (for language designers) and easy to use (for end-users)*” [Merkle, 2010], several workbenches have been developed. Among them, we can mention the *Textual Editing Framework* (TEF) [TEF, 2013], *Xtext* [Xtext, 2013], *Textual Concrete Syntax* (TCS) [Jouault et al., 2006], or *EMF-Text* [Devboost, 2013]. Although Merkle was not able to declare a winner in his language workbenches comparison [Merkle, 2010], we have selected *Xtext*. Our decision was motivated, amongst others, by the tool stability, its active community, its ease of learning, and its integration in Eclipse. The latter allows the interoperability with model transformation and analysis tools (see Chapter 6) and offers extension facilities. Efftinge and Zarnekow demonstrated that it has several advantages over XML-based solutions [Efftinge and Zarnekow, 2010]. Furthermore, the tool is quite extensively used in academic as well as industrial contexts, as evidenced by the *2010 Eclipse Community Award for the most innovative Eclipse project*².

¹ See <https://info.unamur.be/tvl/>

² See http://www.eclipse.org/org/press-release/20100322_awardswinners.php

5.1 XTEXT

On the Eclipse Web page, *Xtext* is defined as “a framework for development of programming languages and domain specific languages” [Xtext, 2013]. Furthermore, it covers a broad range of language infrastructure, including parsers, linker, compiler as well as Eclipse IDE integration.

At first, *Xtext* was part of the openArchitectureWare (oAW) project developed by *itemis AG* which provides tools to develop model driven software development infrastructures. In this context, *Xtext* was developed to support the creation of textual DSLs. Since June 2009, it is part of the *Eclipse Modeling Project* and developed by the global Eclipse community. More specifically, it is part of the *Textual Modeling Framework* (TMF) that “provides tools and frameworks for developing textual syntaxes and corresponding editors based on EMF”. EMF stands for *Eclipse Modeling Framework*, a “modelling framework and code generation facility for building tools and other applications based on a structured data model”. More detailed information about the *Eclipse Modeling Project* is available online [Eclipse, 2013]. In the literature, the (old) openArchitectureWare version is referenced as *oAW Xtext* while the newest (and current) one is named *TMF Xtext*.

In *Xtext*, the user starts by defining her grammar in a `.xtext` file. The language has to be defined using a variant of the context-free *Extended Backus-Naur Form* (EBNF) notation [ISO/IEC, 1996]. The latter does not support left-recursive grammars. Production rules and terminal symbols are mixed into the same `.xtext` file. Common terminal rules such as, e.g., IDs (ID terminal in *Xtext*), integers (INT), strings (STRING), and comments (ML_Comment for multiple lines and SL_CCOMMENT for single line) are available by default in *Xtext*. The grammar definition editor itself is supported by syntax highlighting, keyword completion and outline view just like it will be the case for the defined language. Grammar examples will be discussed in the following sections.

The whole language infrastructure is simply derived from the language grammar via a *Modeling Workflow Engine* (`.mwe`) file. This file describes the necessary building steps (i.e., loading the model, running the checkers and code generators, etc.) for the language. This file is the core of *Xtext*’s code generation approach.

The Abstract Syntax Tree (AST) is rendered as an *Ecore* model, the EMF implementation of meta-models. Thereby, integration with tools from the *Eclipse Modeling Project* is facilitated, including M2M and M2T transformations. Every grammar rule corresponds to a meta-type in the *Ecore* model. The corresponding Java API to access the EMF classes is also made available. The generated artefacts include an ANTLR parser [ANTLR, 2013] that can read the textual syntax and returns the corresponding AST.

In our case, the most interesting component generated by *Xtext* is the editor. It is an Eclipse plug-in built on top of components previously introduced. The generated editor offers most (if not all) functionalities that one would expect from language editors “out of the box” (i.e., with .mwe file default values). It comes with *syntax highlighting* which allows faster reading of the code and *content assist* which speeds up code writing by suggesting keywords, names of elements, etc. Instantaneous code validation is also available through *syntax* and *linking* checkers. Syntax errors occur if the produced code cannot be parsed correctly. Linking errors appear when a reference to another element cannot be found in the document (or the scope available for this reference). An outline view of the document is also part of the language editor. It comes with many other features not detailed here. Interested readers can refer to the *Xtext* documentation for more information [Xtext, 2013].

For most users, the default editor generated by *Xtext* shall be sufficient. For others, *Xtext* can be configured and editors can be customized through an easy-to-use API. For example, the outline view can be completely rewritten, the scope for references extended or refined, qualified names customized, quick fixes implemented, syntax colouring modified, etc. In short, almost everything can be tailored according to the language designers’ preferences. *Xtext* relies on the *Google Guice* dependency injection framework [Vanbrabant, 2008] to assemble all its components. An external module is used to bind a component whenever it is required. If such a component is refined, *Google Guice* will fetch it (almost) transparently. This point will be illustrated on TVL, TVDL and FCSS editors generated by *Xtext* in the following sections.

The remainder of this chapter is devoted to the editors of the three languages presented in Chapter 4. We keep the same presentation order for the languages, i.e., we start with TVL, continue with TVDL and end with FCSS. The grammar, the generated editors, and the customization developments are presented for each language, TVL being the most complex one.

5.2 TVL EDITOR

We start by presenting the TVL grammar as it is central to the *Xtext* framework. Then, we discuss the different custom implementations on top of the generated editor in the following sections.

5.2.1 Grammar

Some properties of the language can be found in the header of the EBNF grammar file. The header of TVL is described in Listing 5.1.

Listing 5.1: Header of the TVL *Xtext* grammar

```

1 grammar be.unamur.TVL with org.eclipse.xtext.common.Terminals
2
3 generate tvl "http://www.unamur.be/TVL"

```

The name of the language is defined at line 1 using Java-like qualifiers. Existing grammars can be reused and are declared using the *with* keyword followed by the name of the grammar. For TVL, we reused the common terminals mentioned in Section 5.1 (see line 1). Ecore models are another cornerstone of the *Xtext* framework. As a reminder, they are used to represent the AST of a document. It can be derived directly from the grammar. The *generate* declaration performs such services. The example at line 3 means that the EPackage (the Ecore model plus some utilities) named tvl will be generated with the *nsURI* `http://www.unamur.be/TVL`. Conversely, it is possible to import an existing EPackage but this option was not used here.

Next comes the TVL grammar itself. Here, we present only excerpts of it. The complete grammar is available in Appendix A.1. The *Xtext* EBNF grammar is structured into rules, identified by their name (followed by a colon). The starting elements of the TVL grammar are visible in Listing 5.2.

Listing 5.2: Starting elements of the TVL *Xtext* grammar

```

1 Model: model+=ModelElement*;
2
3 ModelElement:
4     Type
5     | Constant
6     | Root_Feature
7     | Feature_Extension;

```

The `Model` rule is the grammar entry point (see line 1 of Listing 5.2). A model is composed of zero or more `ModelElements`, as indicated by the asterisk character. All model elements are assigned to the `model` list (depicted by the `+=` operator). This information will be used in the Ecore model to get access to `ModelElements` of the `Model`. A `ModelElement` (see line 3 through line 7) can either be a `Type`, a `Constant`, the `Root_Feature`, or a `Feature_Extension`. Those rules are discussed hereunder.

We start with types which are defined in Listing 5.3.

Listing 5.3: Types excerpt of the TVL *Xtext* grammar

```

1 Type:
2     SimpleType
3     | Record;
4
5 SimpleType:
6     type='int' name=ID ('in' domain=Set_Expression)? ';'
7     | type='real' name=ID ('in' domain=Set_Expression)? ';'

```

```

8 | | type='enum' name=ID 'in' enumDomain=Enum_Expression ';'
9 | | type='bool' name=ID ';'
10 | | type='string' name=ID ';;';
11
12 Record: type='struct' name=ID '{' fields+=Record_Field+ '};'
13
14
15 Record_Field:
16 | type='int' name=ID ('in' domain=Set_Expression)? ';'
17 | type='real' name=ID ('in' domain=Set_Expression)? ';'
18 | type='enum' name=ID 'in' enumDomain=Enum_Expression ';'
19 | type='string' name=ID ';'
20 | type='bool' name=ID ';'
21 | typeref=[SimpleType|ID] name=ID ';;';

```

In TVL, types can either be simple ones (lines 5-10 of Listing 5.3) or records (line 12). As mentioned in Chapter 4.1, a `SimpleType` has a type, a name, and optionally a domain for some of them. The `Set_Expression` rule will be discussed later. The `Record` rule states that a record is composed of one or more `fields`. (plus sign). Each field can be a simple type or a reference to a user-defined type (line 21). The reference is depicted by the `[SimpleType|ID]` construct, meaning that `typeref` refers to a `SimpleType` by its ID. Even with type references, records cannot be contained inside other records. The `Record_Field` rule is quite similar to the `SimpleType` one. We had to duplicate the first one due to grammar ambiguities. Constants being similar to types, their grammar will not be discussed here.

Next come the rules dedicated to all features, i.e., root, hierarchical, extension, and shared ones. They are declared by the EBNF grammar snippet visible in Listing 5.4.

Listing 5.4: Features excerpt of the TVL *Xtext* grammar

```

1 Root_Feature: 'root' name=ID body=Feature_Content;
2
3 Hierarchical_Feature:
4 | optional?=('opt')? name=ID cardinality=BasicCardinality? body=Feature_Content?;
5
6 Feature_Extension: ref=Long_ID body=Feature_Content;
7
8 Shared_Feature: 'shared' ref=Long_ID;

```

The first two, namely `Root_Feature` at line 1 of Listing 5.4 and `Hierarchical_Feature` at line 3, are feature declarations. The two others refer to existing feature declarations using their `Long_ID`. This rule, not visible here, defines (fully) qualified names using a Java-like notation as discussed in Section 4.1. It is possible to define a cardinality (`BasicCardinality` at line 3) or the optionality (`opt` keyword) for `Hierarchical_Features` only. Root features cannot be optional and there can only be one instance of it. The cardinality of feature extensions is defined by the feature declaration they refer

to. `Shared_Features`, unlike others, have no contents. They have to be defined in the original feature declaration. The others have a `Feature_Content` (optional for `Hierarchical_Features`) which is defined in Listing 5.5.

Listing 5.5: Feature contents excerpt of the TVL *Xtext* grammar

```

1 Feature_Content:
2   '{' bodyItems+=Feature_Body_Item+ '}'
3   | group=Feature_Group;
4
5 Feature_Body_Item:
6   Data
7   | Constraint
8   | Attribute
9   | Feature_Group;
10
11 Feature_Group:
12   'group' cardinality=Cardinality '{' sub_features+=Sub_Feature (',' sub_features+=
    Sub_Feature)* '}' ;
13
14 Sub_Feature:
15   Hierarchical_Feature
16   | Shared_Feature;

```

A `Feature_Content` is either a set of (at least one) `Feature_Body_Items` delimited by curly braces or just a `Feature_Group`. A feature body item can either be a data block (Data rule not defined here), a constraint (defined hereunder), an attribute (defined hereunder) or a group. The definition of a `Feature_Group` is given at lines 11-12 of Listing 5.5. It has a `Cardinality` (as defined in Chapter 4) and a comma-separated list of sub-features. A group contains at least one sub-feature which can be a hierarchical or shared one (line 14 through line 16).

Attributes are another type of `Feature_Body_Item`. Their grammar is visible in Listing 5.6.

Listing 5.6: Attribute declaration excerpt of the TVL *Xtext* grammar

```

1 Attribute:
2   Base_Attribute
3   | {Structure_Attribute} type=[Record] name=ID cardinality=BasicCardinality? '{'
    sub_attributes+=Sub_Attribute+ '}' ;
4
5 Base_Attribute::
6   type='int' name=ID cardinality=BasicCardinality? attr_body=Attribute_Body? ';'
7   | type='real' name=ID cardinality=BasicCardinality? attr_body=Attribute_Body? ';'
8   | type='bool' name=ID cardinality=BasicCardinality? attr_body=Attribute_Body? ';'
9   | type='string' name=ID cardinality=BasicCardinality? attr_body=Attribute_Body? ';'
10  | type='enum' name=ID cardinality=BasicCardinality? 'in' domain=Enum_Expression ('is'
    attr_value=Expression | ',' attr_condition=Attribute_Conditional)? ';'
11  | predefined_type=[SimpleType|ID] name=ID cardinality=BasicCardinality? attr_body=
    Attribute_Body? ';' ;

```

```

12
13 Sub_Attribute: sub_id=[Record_Field] attr_body=Attribute_Body ';;

```

As for types, TVL attributes can be split into two categories: `Base_Attributes` and `Structure_Attributes`. Here, the `Structure_Attribute` rule is defined directly inside the `Attribute` one at line 3. Its type must be a user-defined record (`[Record]` reference). Similarly, its `Sub_Attributes` (line 13) refer to the user-defined `Record_Fields` of the record. `Base_Attributes` for their part have a type, an ID, and optionally a cardinality and/or a body (from line 5 to line 11). The type can either be one of the five basic ones or a reference to a user-defined one as defined by the `[SimpleType|ID]` reference at line 11. The definition of the `Attribute_Body` is deliberately left out here but is available in Appendix A.1.

Finally, comes the `Constraint` rule, the last kind of `Feature_Body_Item` (see Listing 5.7).

Listing 5.7: Constraints excerpt of the TVL *Xtext* grammar

```

1 Constraint:
2   condition=('ifin ':'|' ifIn :) expression=Expression12';
3   | condition=('ifout ':'|' ifOut :) expression=Expression12';
4   | expression=Expression12';
5
6 Expression12 returns ComplexExpression: ;
7   Expression11 =>((If. left=current) '?' right+=Expression12' right+=Expression11)*;
8
9 Expression11 returns ComplexExpression:
10   Expression10 =>((LeftImplication.left=current)' <- ' right=Expression10)*;
11
12 Expression10 returns ComplexExpression:
13   Expression9 =>((RightImplication.left=current)' -> ' right=Expression9)*;
14
15 ...
16
17 Expression2 returns ComplexExpression:
18   Expression =>((Excludes.left=current)' excludes' | {Requires.left=current} requires'
19   right=Expression)*;;
20
21 Expression:
22   value='true'
23   | value='false'
24   | value=Integer
25   | value=Real
26   | ref=Long_ID
27   | op='!' expression=Expression
28   | op='-' expression=Expression
29   | op='(' expression=Expression12 ')'
30   | op='abs' '(' expression=Expression12 ')'
31   ....
32   | op='xor' '(' (expression_list=Expression_List | child=Children_ID | values=Values_Set)
33   ');

```

A Constraint optionally has a condition (*ifIn* or *ifOut*), and the constraint itself expressed by the `Expression12` rule. We had to define different rules for the expressions in order to force the operator precedence, which explains the numbered `Expression` rules from line 6 through line 31. For the sake of conciseness, we mentioned some of them only in the above grammar excerpt. New *Xtext* grammar language facilities are used here. First, the `returns` statement for all `ExpressionX` stipulates that they will all be of type `ComplexExpressions` in the `Ecore` model. Then, the `=>` instruction of those same rules guides the parser generator. It forces to check the second (optional) part of the complex expression before going to the lower level expression. This prevents wrong TVL model parsing, like, e.g., confusion between the “-” symbol of the minus unary operator (line 27 of Listing 5.7) and the right implication `->` (line 13). `{RightImplication.left=current}` at line 13 is an example of the last *Xtext* facility newly used. It means that `left Expression9` of `Expression10` will be rendered as the left child of the right implication in the AST. In this way, `Expression9` is marked as such if and only if it is really the case, so avoiding to have left parts without right ones.

5.2.2 Default Infrastructure

As stated in the above section dedicated to *Xtext*, the framework generates some default infrastructure components like the parser or the `Ecore` model based on *MWE2*. That is a DSL to configure the generator.

We will not get into the details of the *MWE2* syntax as, for TVL, we basically used the default behaviours. In short, variables are first declared. They include, amongst others, the grammar URI or the file extension of the generated language (`.tv1`). Variable declarations are shown in the following code excerpt. The `grammarURI` variable, for example, will then be referenced as `#{GrammarURI}` in the *MWE2* workflow.

```
1 var grammarURI = "classpath:/be/unamur/TVL.xtext"
2 var file .extensions = ".tv1"
```

Then follows the root element of the *MWE2* file, the workflow itself. It is composed of beans and components. Beans “*do nothing but provide a means to apply global side-effects*” [Xtext, 2013]. For TVL, we defined two beans, `StandaloneSetup` and `be.unamur.TVLSupport`. The first one initializes a bunch of elements in order to allow to use the language infrastructure independently of the editor plug-in while the second allows, among others, to reference language constructs from any *Xtext* language. The latter is useful for TVDL and FCSS languages.

Workflow component elements directly follow bean declarations. In TVL, we used two kinds of components. The first one cleans the directories of the previously generated infrastructure (`DirectoryCleaner` com-

ponent). The second generates the whole infrastructure and is consequently named Generator. Furthermore, it is composed of generator fragments. Standard fragments handle the generated code for EMF models (EcoreGeneratorFragment), Model validation (JavaValidatorFragment), etc.

The result of running the *MWE2* workflow is a whole infrastructure for TVL. The tip of the iceberg visible to the TVL designer is the editor depicted in Figure 5.1. It contains the printer software example introduced in Section 4.1. Given that the generated editor offers, by default, syntax checking, it shows that our TVL model is syntactically correct. Furthermore, default syntax colouring helps to read the model. Code folding is also generated by *Xtext* (⊖ in the left column). Other facilities like (basic) auto-completion, syntax checking, etc. are also available by default but not visible in Figure 5.1.

```
Printer.tvl
// Declaring a custom type:
enum orientation in {horizontalLeft,horizontalRight,vertical};
// Declaring a structured type:
struct coord {
  int x;
  int y;
}

// The feature model:
root Document {
  // And-decomposition of the root feature:
  group allOf {
    Sheet group [...] {
      opt Tab, // an optional feature
      Page,
      opt Hole,
      Media,
      Staple,
      NumberingMethod
    },
    opt SpineCaption
  }

  // Attribute declarations of the root feature:
  enum type in {normal, booklet, perfectBinding};
  enum stackMethod in {none, offset, mixed};

  // A constraint:
  Document.type == booklet -> !Sheet.Hole;
}

// The features SpineCaption and Hole are extended:
SpineCaption {
  orientation orient;
  ifIn: Document.type in {booklet, perfectBinding};
}

Hole {
  coord position {x is 3;}
}
```

Figure 5.1.: TVL editor generated by *Xtext*

5.2.3 Custom Developments

Although the TVL editor generated by default is an excellent starting point, it has its limits such as incomplete validation, permissive scoping or overwhelmed outline tree. To tackle them, we took advantage of the extension and customization facilities offered by *Xtext*. The custom developments can be split into two categories: TVL model validation and TVL editor.

Model Validation

TVL model validation is required at two levels: syntax and semantics.

Xtext could handle all syntactical checks. But, due to one of our grammar choices, namely construct IDs, some custom developments had to be made. In the grammar presented earlier, features, attributes, types and constants all use the ID terminal, although some differences exist. Feature names should start with an upper-case letter while the three other constructs should have a leading lower-case letter. We could have defined different IDs but, in such a case, TVL qualified names would have been far more difficult to handle. Our choice has less of an impact as it is sufficient to implement five additional checks, namely for attribute, constant, user-defined type, record field, and feature names.

In *Xtext*, such checks are defined in Java inside the validation package of the language (`be.unamur.tvl` in our case) and extends default validation methods generated by the framework which allows subclasses to specify invariants in a declarative manner using the `@Check` annotation. The code excerpt presented in Listing 5.8 contains an example of validation for attribute names. Returned errors contain a message (“Attribute names should start with a lower case” in our example), the incriminated TVL construct (`TvlPackage.Literals.SHORT_ID__NAME` representing the attribute ID), and the cause defined as a string constant (`INVALID_LOWER_CASE`). The latter can be used in other contexts as discussed in the following sections.

Listing 5.8: Validation of attribute names in the TVL editor

```

1 package be.unamur.validation;
2
3 public class TVLJavaValidator extends AbstractTvlJavaValidator {
4
5     public static final String INVALID_LOWER_CASE = "be.unamur.tvl.
        InvalidLowerCaseName";
6
7     @Check
8     public void checkAttributeStartsWithLowerCase(Attribute attribute){
9         if (attribute.getName()!=null && !Character.isLowerCase(attribute.getName().charAt(0)))

```

```

10         error("Attribute_names_should_start_with_a_lower_case", TvIPackage.Literals.
11             SHORT_ID__NAME, INVALID_LOWER_CASE);
12     }

```

Semantic validation is more difficult to handle. The root cause is the structuring mechanism offered by TVL to declare a feature at one place and extend it later in the code (at several places). This facility is not supported by default in *Xtext*. The framework generates an API to get access to the different constructs (attributes, root feature, hierarchical features, feature extensions, etc.) of the model but do not provide means to get access to the whole contents of a feature. It is the union of the contents from the feature declaration and its potential extensions. Getting access to this information is essential to check the semantics of TVL models. We developed our own accessors and custom navigation mechanisms relying on some methods to resolve *Long_IDs* used to reference TVL constructs. They are located in the `be.unamur.utils.TVLUtils` class and developed in Java.

Methods for resolving TVL qualified names are the following:

1. **resolveLong_ID(Long_ID id)**: Returns the TVL construct referenced by a qualified name (*Long_ID* in the grammar). It can be a feature declaration, an attribute (or a sub-attribute), a constant, or a value of an enumerated attribute. This method depends on the others presented in this list which handle *root*, *this* and *parent* keywords.
2. **getRoot(EObject tvlConstruct)**: Returns the model root feature given any construct. In TVL, it must be unique.
3. **getThis(EObject tvlConstruct)**: Returns the feature declaration which contains *tvlConstruct* if it is not a feature declaration, itself otherwise.
4. **getParent(EObject tvlConstruct)**: Returns the parent feature declaration of the feature declaration where *tvlConstruct* is declared.

Based on those accessors related to qualified names, all feature content queries can be defined. In the following list, *Common_Feature* represents either a feature declaration (root or hierarchical) or an extension.

1. **getAllInstancesFeature(Common_Feature feature)**: Returns all instances of a feature (the declaration plus its extensions, if any).
2. **getCommonFeature_Group(Common_Feature feature)**: Get, given a common feature, its group. All feature instances corresponding to *feature* will be explored to find a group, if any.
3. **getFeatureBodyItems(Feature_Declaration feature)**: Get the list of items included in the bodies of a feature and its extensions. Such items include data, constraints, attributes and groups.

4. **getAllAttributes(*Feature_Declaration* feature):** Get the list of attributes declared in the bodies of feature and its extensions.
5. **getAllConstraints(*Feature_Declaration* feature):** Get the list of constraints declared in the bodies of feature and its extensions.

All semantic validation methods are defined in the same class as the syntactical ones, `be.unamur.utils.validation.TVLJavaValidator`. Here we give, an intuition of the different checks performed. For the Java implementation, interested readers might refer to the TVL editor source code. The first validation, uniqueness of the root feature, is quite direct: the count of `Root_Features` should be equal to one (see `checkSingleRootFeature` method). A similar approach applies to the number of groups (maximum one) in each hierarchical feature. We have to explore the hierarchical feature and its extensions to count the number of groups. An error is raised when it is greater than one. See `checkSingleGroupDeclaration` method for the Java source code.

The names of the different constructs also have to be checked to guarantee their uniqueness. As a reminder, features can have the same name provided they are not siblings. Consequently, it is sufficient to check that a feature has a unique name in its group. The root feature is unique and does not have to be checked. Uniqueness of feature names is checked directly in group declarations. For each sub-feature (hierarchical or shared one), we compare its name to its siblings in `checkHierarchical_FeatureHasUniqueName` method. The same principle applies to attributes, their name has to be unique inside their parent feature. The first step is thus to collect all attributes declared by the parent via the `getAllAttributes` method and compare them (see `checkAttributeHasUniqueName`). The uniqueness of the constant names is performed in a similar way in the `checkConstantHasUniqueName` method. Finally, we also check that an attribute does not have the same name as a constant as there is no way to distinguish them. For each attribute (resp. constant), we get the list of constants (resp. attributes) declared in the model and compare their names (see `checkCollidingAttributeConstant` methods).

Given that shared features cannot be extended (the referenced one should be extended instead), we check that no feature extension references a shared feature using the `resolveLong_ID` method introduced earlier. A similar approach checks that a shared feature does not refer to a shared feature. It should refer to its original declaration instead.

Finally, we also check elements referenced by qualified names. First, we count the number of elements it refers to. If it is greater than one, it means that the qualified name is ambiguous (see `checkLong_ID` method). One way to solve this problem is to use the fully qualified name which is, by construction, unique. Second, we check that qualified names used to reference a

feature declaration in a feature extension actually refer to a feature declaration (root or hierarchical one) in `checkFeatureExtensionRef`. The first step verifies that *this* and *parent* keywords are not used as they make no sense at that place. Then, we check that the qualified name actually refers to a feature declaration. Finally, we check that a `Children_ID` (used to reference attributes in expressions) actually refers to an attribute in a similar manner (see `checkChildren_ID` method).

Another important point is the scoping, i.e., TVL constructs accessible given a context. It is decomposed into two levels: global and construct-specific.

One limitation of the current implementation of the TVL editor is that it does not support the *include* mechanism. In this first version, visible elements are those defined inside the same *.tvl* file. This is defined in the `TVLGlobalScopeProvider` class of the `be.unamur.scoping` package. There, we created a filter which excludes all elements that are not defined in the same file as the current context object. Otherwise, Eclipse would grant access to all constructs from all TVL models inside a given project.

The construct-specific scoping is limited to two of them in our case, record fields and qualified names. They are defined in the `TVLScopeProvider` class.

The qualified names scoping method follows the following naming convention:

```
IScope scope_<DeclaringEClass>_<Reference>(<Context> ctx, EReference ref)
```

It is used when evaluating the scope of a specific cross-reference named `Reference` declared in `DeclaringEClass`. In the signature, they are separated by the underscore character and always preceded by “scope_”. The `ctx` parameter represents the context in which the scope should be applied and the `ref` one, the cross-reference object itself. For example, “`scope_Long_IDTail_head(Long_IDTail context, EReference ref)`” defines the scope of the head part of a `Long_IDTail` when the user is editing a `Long_IDTail` (`ctx` parameter). Please, refer to the grammar introduced at the beginning of this section for more detailed information about those TVL rules.

As for validation, we just give an intuition of the scoping method. The first step is to determine whether the current element (`ctx`) is the second element or located later in the qualified name. In the first case only the preceding element can be a keyword. In that case, it has to be resolved and the scope corresponds to the set of TVL constructs it contains. For example, if it is a feature, return all its sub-features, attributes, etc. using the utilities previously introduced. In all other cases, the scope will depend on the type of the preceding element in the qualified name. It can either be an attribute, a sub-attribute, an enumerated attribute element, a constant, a shared feature, or a feature declaration. The scope for an attribute (excluding *struct* ones), a sub-attribute, an enumerated attribute element, or a constant is empty given

that they have no relevant contents. The scope of a structure is the set of its sub-attributes. For shared features, it corresponds to the scope of the feature it references. For feature declarations, we use the utilities previously defined to get all their sub-features and attributes which might be scattered over several feature extensions.

The scoping of struct attributes follows a different pattern:

```
IScope scope_<TypeToReturn>(<Context> ctx, EReference ref))
```

In our case, the scope of a *struct* attribute is the set of its sub-attributes, regardless of the attribute. The signature of our method is the following:

```
IScope scope_Record_Field(Structure_Attribute struct, EReference ref)
```

The scope is composed of `Record_Fields` given an attribute (struct parameter) and a cross-reference (ref). Its implementation is quite direct: retrieve the record corresponding to the type of struct and return all its record fields.

Finally, `TVLGlobalScopeProvider` and `TVLScopeProvider` have to be declared in the `TVLRuntimeModule` in order to override the default behaviour (see Listing 5.9).

Listing 5.9: Registration of the scope providers for the TVL editor

```

1 public class TVLRuntimeModule extends be.unamur.AbstractTVLRuntimeModule {
2     ...
3     @Override
4     public Class<? extends IScopeProvider> bindIScopeProvider() {
5         return TVLScopeProvider.class;
6     }
7
8     @Override
9     public Class<? extends IGlobalScopeProvider> bindIGlobalScopeProvider() {
10         return TVLGlobalScopeProvider.class;
11     }
12 }
```

Editor

Customization is also available at the graphical interface level. Three custom developments were made: improvement of the syntax highlighting, rewriting of the outline tree and development of two basic quick fixes.

Syntax highlighting can be done at two levels: lexical and semantic. The lexical level was not modified as the default behaviour (depicted in Figure 5.1) fits our needs. Custom developments were made at the semantic level only in order to highlight more TVL model constructs, so improving its readability.

The first step, regardless of the highlighting level, is the definition of the available styles. This information is located inside the `TVLHighlighting-Configuration` class of the `be.unamur.ui.highlighting` package. The code

excerpt contained in Listing 5.10 declares a style name (line 2), registers it (line 4) and defines its visual aspect (line 6 through line 10):

Listing 5.10: Declaration of an highlighting style for the TVL editor

```

1 public class TVLRuntimeModule extends be.unamur.AbstractTVLRuntimeModule {
2     public static final String FEATURE_ID = "feature";
3     public void configure(IHighlightingConfigurationAcceptor acceptor) {
4         acceptor.acceptDefaultHighlighting(FEATURE_ID, "Feature", featureTextStyle());
5     }
6     public TextStyle featureTextStyle() {
7         TextStyle textStyle = new TextStyle();
8         textStyle.setColor(new RGB(65,105,225));
9         return textStyle;
10    }
11 }

```

Next comes the semantic highlighting calculation. The framework will pass the current *XtextResource* (the TVL model) and an *IHighlightedPositionAcceptor* to the calculator. Given the provided resources, we have to iterate through the elements contained in the TVL model and determine their type. Upon completion of this first step, the positions to be highlighted are calculated. Determining the TVL constructs to highlight is made available in the *provideHighlightingFor* method of the *TVLSemanticHighlightingCalculator* class. There, we also find five methods which compute the characters to highlight in the GUI, one for each relevant TVL construct (feature, constant, type, attribute and reference). A given construct can be covered by several highlighting configurations. In such a case, they will be combined. For example, the qualified name of a feature extension will be highlighted as a feature as well as a reference. Figure 5.2 shows the improved semantic highlighting and should be compared to Figure 5.1 to assess the added value.

The *Xtext* framework generates a default outline view for the editor. It is based on the different classes of the *Ecore* model representing the AST. It is depicted in Figure 5.3a. A noticeable characteristic of this Eclipse view is that it contains a lot of intertwined levels, a lot of them presenting the *<unnamed>* label. The number of levels in the tree is determined by the grammar rules, each level corresponding to a rule. However, in TVL we had to define some intermediary constructs to group similar elements together. And those rules should not be rendered in the outline tree. *Xtext* uses the text in grammar rules to set outline tree labels. Given that all rules do not contain a terminal rule (i.e., text), it is impossible for the framework to get a label. This explains the *unnamed* elements in Figure 5.3a. This default outline was customized and the result is visible in Figure 5.3b.

The first noticeable difference is that the file (*Printer*) is not covered anymore. Our goal is to reduce the depth of the tree by removing this useless in-

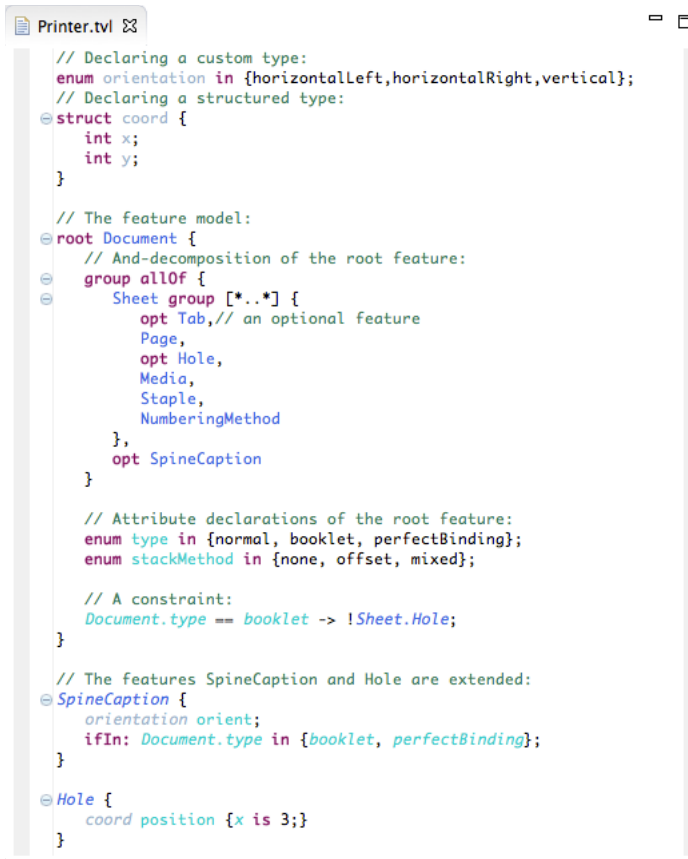


Figure 5.2.: TVL editor with semantic highlighting

formation. Model designers know that the outline corresponds to the file they are currently editing. Additionally, outline levels are now labelled with the type of the TVL construct they represent between square brackets. Available values are *Type*, *Constant*, *Feature root*, *Feature*, *Group*, *Attribute*, and *Constraint*. Most of them are displayed in Figure 5.3b. Their purpose is to give readers a quick preview of the TVL model, more specifically its structure. For this reason, details have been omitted. The outline view could be seen as an intermediary representation somewhere between the TVL syntax and the FODA notation. All unnecessary levels have been removed and a label defined for each of the remaining ones.

As for other custom developments, we just provide a high level description for the customization of the TVL editor outline view. Java implementation is

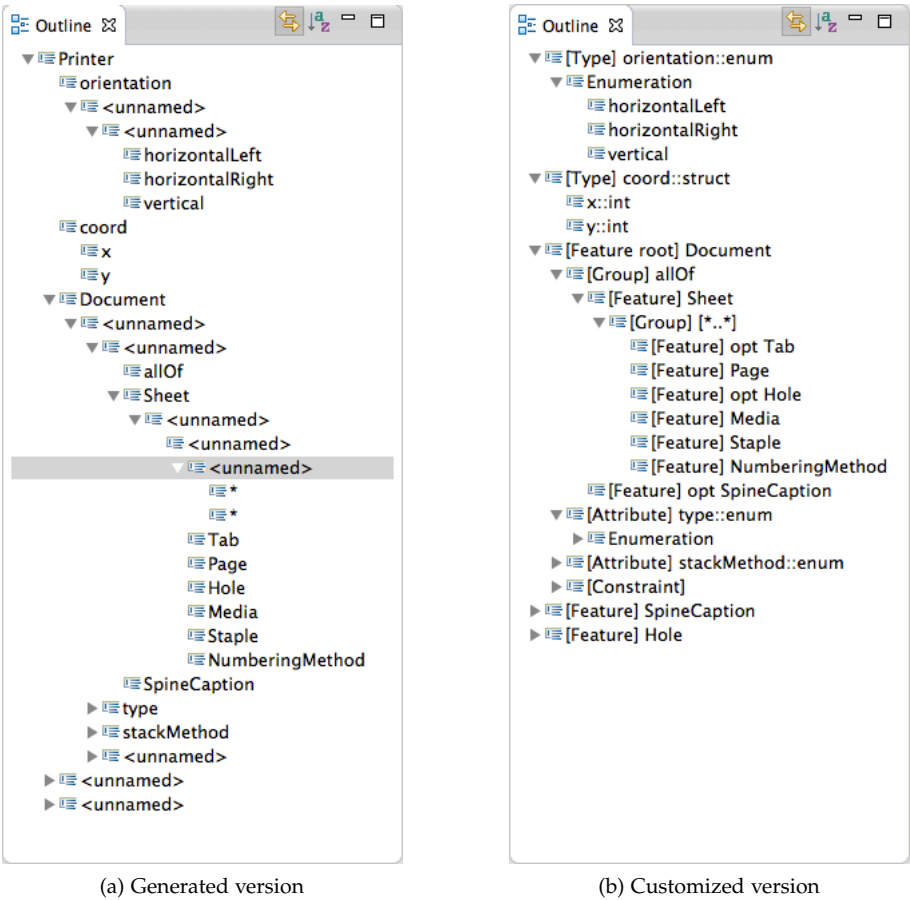


Figure 5.3: TVL editor outline view

available in TVLOutlineTreeProvider class of be.unamur.ui.outline class. There, four categories of methods exist and are automatically called by the Xtext framework (using naming conventions):

1. **Node creation:** all methods named `_createNode(IOutlineNode parentNode, <TVLConstruct> construct)` define how a TVL construct is rendered in the outline tree. Basically, it creates a node. The `parentNode` parameter is used to attach the created node to its ancestor.
2. **Children creation:** navigation in the descendant TVL constructs is handled by methods named `_createChildren(IOutlineNode parentNode, <TVLConstruct> construct)`. The role of `parentNode` is the same here.

In our case, those methods were used mainly to discard irrelevant nodes in the hierarchy, so reducing the depth of the outline hierarchy.

3. **Label definition:** Methods named `_text(<TVLConstruct> construct)` are used to set the label of a given TVL construct. They are called directly from `_createNode` methods. If such a custom method has not been defined for the given TVL construct, the label method will be called by default by the *Xtext* framework.
4. **Leaf definition:** The last category is the simplest one, it defines whether a TVL construct has children to explore. The naming convention is the following: `_isLeaf(<TVLConstruct> construct)`. Such methods return a Boolean value and are mainly used to decide whether the descendants of `construct` have to be explored or not. It thus also has a role in the shrinking of the outline tree. Similarly to *Text* methods, *isLeaf* ones are called from node creation ones.

Finally, two basic quick fixes were implemented: 1) for feature name not starting with an upper-case letter and 2) for attributes, constant and types not beginning with a lower-case letter. Both fixes are defined in the `TVLQuickfixProvider` class of the `be.unamur.ui.quickfix` package. They all start with the `@Fix` keyword followed by an error constant defined in the `TVLJavaValidator` class introduced earlier. For example, `@Fix(TVLJavaValidator.INVALID_LOWER_CASE)` is the header of the fix method for attributes, constant, and types not starting with a lower-case letter. Then comes the Java code of the fix itself. It includes, among other things, a short description of the proposed solution. The result is visible in Figure 5.4.

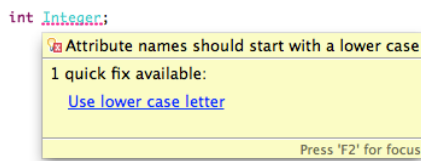


Figure 5.4.: Quick fix in the TVL editor

As for language-related custom developments, some editor-related ones have to be registered. It is the case for syntax highlighting and quick fixes. They are declared in the `TVLUiModule` class of the `be.unamur.ui` package. The syntax is the same as for language-custom developments and will not be recalled here.

The editor has been packaged and is available on the following Eclipse update site: <https://staff.info.unamur.be/qbo/Tools/>.

5.3 TVDL EDITOR

As for TVL, we first introduce the TVDL grammar before discussing the language infrastructure generated by *Xtext* and the improvements.

5.3.1 Grammar

The TVDL grammar header is similar to the TVL one. Only the name of the language has been changed (see Listing 5.11). As views rely on TVL constructs, the language has to be imported. It is done at line 3 of the code excerpt. The *nsURI* `http://www.unamur.be/TVL` declared for TVL in its own grammar is used to identify the language. In the TVDL grammar, TVL constructs will be referred to using the `[tv1::TVLConstruct]` syntax where TVLConstruct is, for example, `Attribute`, `Feature_Declaration`, etc.

Listing 5.11: Header of the TVDL *Xtext* grammar

```

1 grammar be.unamur.TVDL with org.eclipse.xtext.common.Terminals
2
3 import "http://www.unamur.be/TVL" as tvl
4
5 generate tvdl "http://www.unamur.be/TVDL"
```

Next comes the TVDL grammar itself. Contrarily to TVL, here we show the full grammar given that it is far less complex.

The first statement of any TVDL model is the import of the corresponding TVL model (see line 2 of Listing 5.12). This is achieved by declaring the name of the TVL file (including the *.tvl* file extension) between double quotes preceded by the *import* keyword. Only one import must be defined, as stated at line 5. Then come the views (line 3). As stated in Section 4.2, they are either grouping or declaration ones.

Listing 5.12: Starting elements of the TVDL *Xtext* grammar

```

1 TVDLModel:
2     tvl=Import
3     views+=View*;
4
5 Import: 'import' importURI=STRING;
6
7 View: View_Grouping | View_Declaration;
```

Listing 5.13 covers the grammar of both view types. They both have a name which matches the *Xtext ID* common terminal. The dollar sign is used to differentiate grouping views. The latter are composed of views declared between brackets and separated by the "&&" operator (line 1). As for TVL, *[View]* is a reference to an existing view. It can either be a view declaration

or a grouping one. View declarations, on their side, are composed of an &&-separated list of `ViewExpressions` also delimited by curly braces (line 3).

Listing 5.13: View definition excerpt of the TVDL *Xtext* grammar

```

1 View_Grouping: '$' name=ID '{' subViews+=[View] ('&&' subViews+=[View])* '}';
2
3 View_Declaration: name=ID '{' expressions+=ViewExpression ('&&' expressions+=
  ViewExpression)* '}';

```

Central to the TVDL grammar is the `ViewExpression` defined in Listing 5.14. A `ViewExpression` refers to a TVL feature or attribute using its qualified name, represented by the `TVL_ID` grammar rule (see line 26 through line 30). We had to redefine the `TVL Long_ID` construct as it covers more elements (e.g., types, constants, etc.) and keywords (*this*, *root*, and *parent*) make no sense for views. An optional refinement can be defined (`ViewExpressionRefinement`) on the referenced TVL construct.

Listing 5.14: View expressions excerpt of the TVDL *Xtext* grammar

```

1 ViewExpression: id=TVL_ID (refinement=ViewExpressionRefinement)?;
2
3 ViewExpressionRefinement:
4   ":" (subtree=SubtreeExpression | attributes=AttributeExpression | group=GROUP | list=
   Common_List);
5
6 SubtreeExpression: keyword=SUBTREE (stopList=Stop_List)? (refinement=
   SubtreeExpressionRefinement)?;
7
8 SubtreeExpressionRefinement: "|" (attributes=AttributeExpression | groups=GroupExpression
   | list=Common_List);
9
10 AttributeExpression: keyword=ATTRIBUTES ( "|" list=Common_List)?;
11
12 GroupExpression: keyword=GROUPS ( "|" list=Common_List)?;
13
14 Stop_List: ("/" stopElements+=List_Element)+;
15
16 Common_List: Exclusion_List | Inclusion_List;
17
18 Exclusion_List: "!" "[" elements+=List_Element (',' (elements+=List_Element))* "]";
19
20 Inclusion_List: "[" elements+=List_Element (',' (elements+=List_Element))* "]";
21
22 List_Element:
23   id=TVL_ID
24   | keyword=(ATTRIBUTES | GROUP | GROUPS);
25
26 TVL_ID:
27   head=[tv1::Feature_Declaration] tail=TVL_IDTail
28   | head=[tv1::Attribute];
29

```

```

30 TVL_IDTail: '.' head=[tv1::FQN] (tail=TVL_IDTail)?;
31
32 terminal GROUP: "group";
33 terminal GROUPS: "groups";
34 terminal SUBTREE: "*";
35 terminal ATTRIBUTES: "attributes";

```

As mentioned at lines 3-4 of Listing 5.13, a `ViewExpressionRefinement` is either a `SubtreeExpression`, an `AttributeExpression`, the `group` keyword, or a `Common_List`.

A `SubtreeExpression` starts with the asterisk character optionally followed by a `Stop_List`. It can be further refined by a `SubtreeExpressionRefinement`, that is an `AttributeExpression`, a `GroupExpression`, or a `Common_List` (line 8).

`AttributeExpression` (line 10) and `GroupExpression` (line 12) are similar. The first one starts with the *attributes* keyword, while the second begins with the *groups* one. They can then both be refined by a `Common_List`.

A `Common_List` is either an `Exclusion_List` or an `Inclusion_List` (see line 16). The only difference is that the exclusion list starts with an exclamation mark. They are both comma-separated lists of (previously introduced) `List_Elements` delimited by square brackets.

The full TVDL grammar is available in Appendix A.2.

5.3.2 Default Infrastructure

As for TVL, we first describe the infrastructure generated by *Xtext* and, in the next section, we explain how it was customized for our specific needs.

The *MWE2* file is quite similar to TVL. Here, we discuss major changes only. In other words, we do not discuss the values of `grammarURI` and `file.extensions` variables, for example, which have been changed to match the TVDL language.

The two major changes relate the registration of the TVL language and the handling of import statements.

In the TVDL grammar, we imported the TVL language via its `nsURI`. However, those languages might not be registered in the current context, i.e., the `nsURI` might not be visible. To avoid such problems, the TVL language has to be registered in the *MWE2* file of TVDL, more specifically inside the `StandaloneSetup` bean:

```

1 bean = StandaloneSetup {
2     ...
3     registerGeneratedEPackage = "be.unamur.tvl.TvlPackage"
4     registerGenModelFile = "platform:/resource/be.unamur.tvl/src-gen/be/unamur/TVL.
        genmodel"
5 }

```

There, the TVL EPackage generated by *Xtext* is registered via its classpath. The Genmodel is also registered. In this case, its platform-relative file path is required. The Genmodel is an Ecore meta-model that contains additional information for code generation.

By importing a TVL file at the beginning of a TVDL model, one expects to get access to its constructs only, not those from other variability models even if they are located in the same folder. Fortunately, this aspect is handled by *Xtext*. We just had to set the following line inside the language part of the Generator component:

```
1  fragment = scoping.ImportURIScopingFragment {}
```

Basically, it states that global scopes will be based on import URIs. It will bind an `ImportUriGlobalScopeProvider` class that handles the construct named `importURI` in the `Import` rule of the TVDL grammar. In our case, the whole import infrastructure generated by *Xtext* was sufficient but it can be customized for more complex situations.

As for TVL, the most visible result, namely the editor, is depicted in Figure 5.5. It contains the TVDL model of our printer software example introduced in Section 4.2. Given that no error is displayed, we can assume that the model is syntactically correct. Contrarily to the TVL editor, the TVDL one does not support code folding (as it makes no sense in TVDL) and syntax highlighting is limited to comments, the *import* keyword, and the name of the imported TVL model. Here too, custom developments were required.

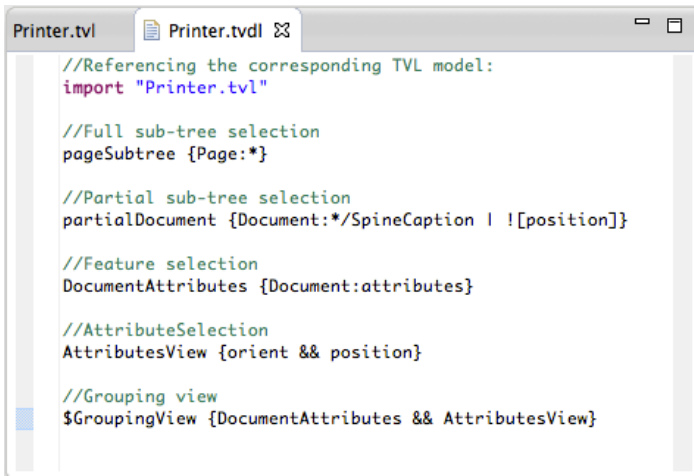


Figure 5.5.: TVDL editor generated by *Xtext*

5.3.3 Custom Developments

Developments were made at two different levels, TVDL model validation and TVDL editor.

Model Validation

For TVDL, we rely on *Xtext* for syntactical checks. However, we still had to define semantic ones. They are all located inside the `TVDLJavaValidator` class of the `be.unamur.validation` package.

For the sake of readability, we present the validation developments in a descending order, from views down to list elements. The first obvious check is the uniqueness of view IDs. It is simply done by comparing a given ID with all other ones in `check_Unique_View_Name(View view)`.

In grouping views, cycles might be defined by the user. The illustrative TVDL model of Listing 5.15 contains an example of such a problem.

Listing 5.15: Illustrative cycle example in a TVDL model

```

1 $GroupA {GroupB}
2
3 $GroupB{GroupC}
4
5 $GroupC{GroupA}
```

To discover them, we explore the descendant views in a recursive way as described the pseudo-code algorithm of Listing 5.16 implemented by the `check_Cyclic_View_Grouping(View_Grouping view)` method.

Listing 5.16: Cycle verification algorithm of the TVDL editor

```

1 CheckCycle(view)
2   for each sub-view of view
3     if the sub-view==view
4       A cycle exists
5     else
6       if the sub-view is a declaration one,
7         No cycle can exist
8       else the sub-view is a grouping one
9         Explore the sub-views of the sub-view itself and compare them to
           view
```

Next comes the verification of TVL constructs referenced in views by their qualified name. The syntax of TVL_IDs ensures that they actually refer to a feature or an attribute. However, it does not guarantee that the qualified name refers to a single feature/attribute. This task is done by the `check_Unique_ViewExpression_ID(ViewExpression expression)` method.

It is sufficient to check that the first component of the qualified name is unique in the TVL model. Indeed, if it is unique, all following elements will be as well given that sibling TVL constructs cannot have the same name.

A view can be refined, unless it refers to an attribute. The `check_NoRefinement_Attributes(ViewExpression expression)` checks that such views do not have a `ViewExpressionRefinement`. For views referencing features and refined by the *group* keyword, a warning is shown to the user if the feature does not contain a group (`check_hasGroup_ViewExpressionRefinement(ViewExpressionRefinement refinement)`).

All other validation methods refer to TVDL `List_Elements`, i.e., elements included either in a list (inclusion or exclusion) or the stop list of a sub-tree refinement. First, we check that the TVL construct referenced by the element is not ambiguous. This task has to be performed only if the element is not contained into the refinement list of a feature. In the latter case, list elements are feature attributes only. They cannot be ambiguous given that sibling attributes cannot have the same name in TVL. For other cases, the `check_Unique_List_Element_ID(List_Element element)` counts the TVL constructs referenced by the element. It must be equal to one. All other checks are trivial and can be summarised as follows:

1. **check_Type_Attribute_List(List_Element element):** Raises an error if an element contained in a list refining an `AttributeRefinementExpression` is not an attribute.
2. **check_Context_List_Element_Keyword(List_Element element):** A list element can also be a keyword. This method checks that 1) such a keyword is not in a sub-tree expression stop list, or 2) is not in an attribute or group expression, 3) the *group* keyword is not used in sub-tree refinements (*groups* keyword should be used instead), and similarly, 4) the *groups* keyword is not used in a view refinement expression.
3. **check_RootFeature_Not_Covered_Exclusion_List(List_Element element):** Checks that a sub-tree exclusion list does not cover the root feature.
4. **check_Unique_Common_List_Element(List_Element element):** Shows a warning if a list contains a duplicated element.
5. **check_Unique_Stop_List_Element(List_Element element):** Plays the same role as the previous one for stop lists of sub-tree expressions.
6. **checkTVDLCoverage_stopList(List_Element element):** Shows a warning if an element in a sub-tree stop list is not in the scope of the sub-tree (e.g., already excluded by other elements) or an error if it is the root of the sub-tree.

7. **check_Covered_Subtree_Exclusion_List(List_Element element):** Warns the user if an element contained in a sub-tree exclusion list is not in the scope of the sub-tree.
8. **check_Conflicts_Exclusion_List(List_Element element):** Displays a warning if an element in an exclusion list is already covered by another element in that list.
9. **check_hasGroup_GroupExpression_Exclusion_List(List_Element element):** Warns the user if a feature contained into an exclusion list does not contain any group.
10. **check_Keyword_ViewExpressionRefinement_List(List_Element element):** Plays the same role as the two previous methods for keywords, i.e., warns the user if the *attributes* keyword is contained in a refinement list of a view expression which does not contain any attribute.

Finally, a coverage analysis algorithm has also been made available. In his thesis, Hubaux provided *sufficient* and *necessary* coverage conditions for views on FMs [Hubaux, 2012]. For TVDL, we implemented the necessary one which states that all the features appear in at least one view. Given that TVL also supports attributes, we had to extend the definition by stating that all attributes should also be covered by at least one view. Furthermore, we also made one implementation choice related to our GUI generation approach. In the TVDL editor, a feature is considered covered if all its contents (attributes and group) and its parent group are covered. For the root feature it is sufficient to check that its contents are covered given that it has no parent group. Our coverage can thus be considered as an intermediary between Hubaux's *sufficient* and *necessary* conditions. The latter states that *"the value of features that do not appear in any view can be inferred from the values on the features that are part of views"*. This implementation is GUI generation-specific and can easily be adapted in the `checkTVDLCoverage(TVDLModel model)` method. It relies on the coverage methods defined in the `be.unamur.utils.TVDLCoverage` package.

Apart from the validation itself, we defined TVDL scoping. As a reminder, it defines which values are available depending on the context. We will not go into details here, the principle is the same as in TVL. We developed four scoping methods. They are all located in the `be.unamur.scoping.TVDLScopeProvider` class. The first one, `scope_TVL_IDTail_head(TVL_IDTail id, EReference ref)` is similar to the method which defines the scope of TVL qualified names. As a reminder, we had to redefine them as some elements (e.g., keywords) made no sense in views. In `scope_View_Grouping_subViews(View_Grouping view, EReference ref)`, we define the scope for a grouping view's sub-views as

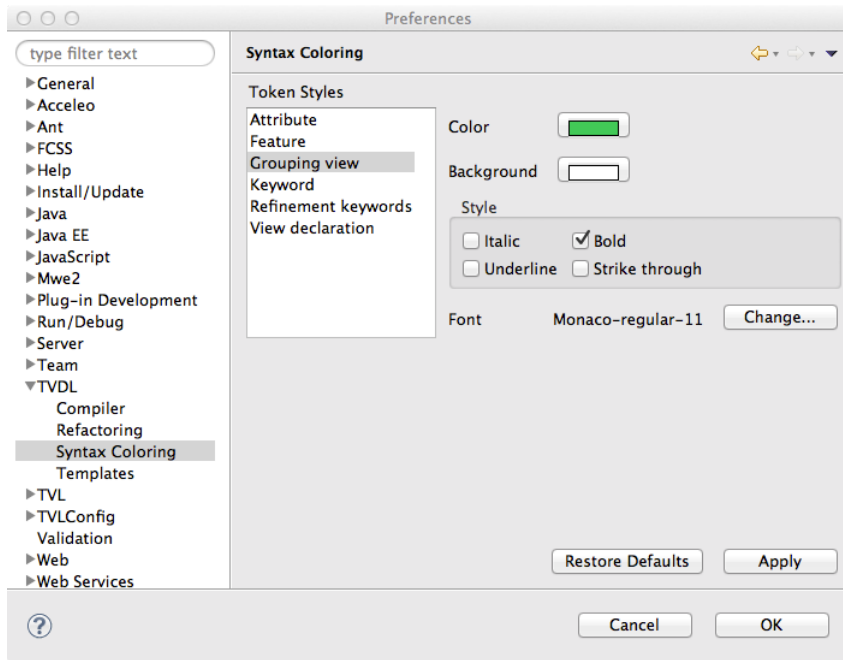


Figure 5.6.: TVDL syntax colouring preferences

the set of all views available in the TVDL model. For the elements of a stop list in a sub-tree expression, only sub-features of the tree should be made available. However, it was not possible to define such a set. The scope returned by `scope_TVL_ID_head(Stop_List list, EReference ref)` is the set of all feature declarations from the TVL model. The parentage is ensured by the validation methods previously introduced. Finally, the scoping of TVL IDs contained in a list is relatively similar to the one for TVL IDs in views presented at the beginning of this paragraph. For the sake of conciseness, it will not be discussed here. Interested readers may refer to the `scope_TVL_ID_head(Common_List list, EReference ref)` method.

As for TVL, the `TVDLScopeProvider` has to be registered into the `be.unamur.TVDLRuntimeModule` class.

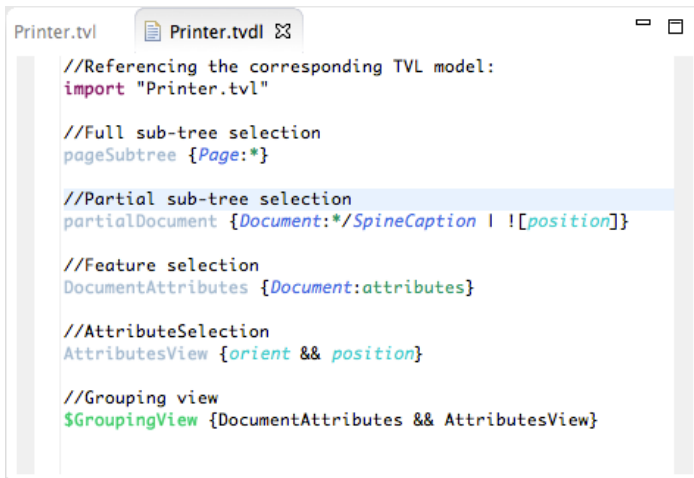
Editor

Two custom developments were made for the TVDL editor: improvement of the syntax highlighting and modification of the outline tree. No quick fix is provided for our view language editor.

With regard to highlighting configuration, we defined six configurations: one for tokens, one for keywords (*group*, *groups* and *attributes*), one for TVL_IDs referencing TVL features, one for the same constructs referencing attributes, and two different ones for grouping and declaration views. The styles defined in the `be.unamur.ui.highlighting.-TVDLHighlightingConfiguration` class are default ones and can be modified by the user in Eclipse preferences as depicted in Figure 5.6.

For TVDL, we defined a token highlighting, as opposed to TVL. The principle is the same as for semantic highlighting: if a given element is found (a keyword in our case), an highlighting configuration is attached to it (the keyword one). It is implemented in the `TVDLTokenHighlightingConfiguration` class and allows to differentiate the three keywords from language tokens. Regarding semantic highlighting, we identified four different elements, the ID of views (declaration and grouping), TVL qualified names for features and for attributes. They are implemented in the `TVDLSemanticHighlightingCalculator` class. The sixth configuration applies to TVDL comments and is handled by default by *Xtext*, as depicted in Figure 5.5.

All syntax highlighting parts are registered inside the `TVDLUIModule`. The result is visible in Figure 5.7. One can, contrarily to Figure 5.5, clearly differentiate grouping from declaration views, keywords, and TVL qualified names referencing features from ones pointing to attributes.



```

Printer.tvl
//Referencing the corresponding TVL model:
import "Printer.tvl"

//Full sub-tree selection
pageSubtree {Page:.*}

//Partial sub-tree selection
partialDocument {Document:*/SpineCaption | ![position]}

//Feature selection
DocumentAttributes {Document:attributes}

//AttributeSelection
AttributesView {orient && position}

//Grouping view
$GroupingView {DocumentAttributes && AttributesView}

```

Figure 5.7.: TVDL editor with custom highlighting

The TVDL editor outline tree was also customized. The motivation behind this is the same as for TVL, improve the readability and the navigability of the

model. Original and customized outline trees can be compared in Figure 5.8. Source code is available in the `TVDLOutlineTreeProvider` class.

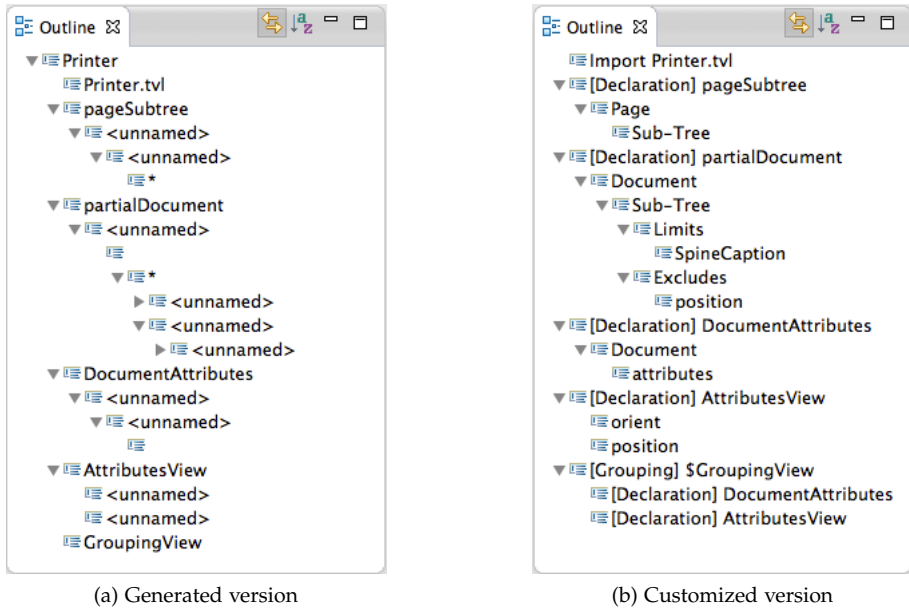


Figure 5.8.: TVDL editor outline view

The editor has been packaged and is available as an Eclipse plug-in at the following URL: <https://staff.info.unamur.be/qbo/Tools/>.

5.4 FCSS EDITOR

Finally, the FCSS editor is the simplest one. The main reason is that a beautification model is essentially a collection of key/value pairs. As a consequence its grammar is not complex (see first sub-section). The advantage of a simple grammar is that *Xtext* handles a large part of the language infrastructure. The number custom developments is thus rather small.

5.4.1 Grammar

Here we show excerpts of the grammar. The complete version is available in Appendix A.3.

The FCSS grammar header is similar to the two other languages except that it imports them both (see Listing 5.17).

Listing 5.17: Header of the FCSS *Xtext* grammar

```

1 grammar be.unamur.FCSS with org.eclipse.xtext.common.Terminals
2
3 import "http://www.unamur.be/TVL" as tvl
4 import "http://www.unamur.be/TVDL" as tvdl
5
6 generate fcss "http://www.unamur.be/FCSS"

```

The key/value pairs of the grammar are not complex and will not be presented. In Listing 5.18, we focus on higher-level grammar rules which are ancestors of those pairs.

Listing 5.18: High level rules of the FCSS *Xtext* grammar

```

1 FCSSModel: tvlImp=Import (tvdlImp=Import)? parts+=Part+;
2
3 Import: 'import' importURI=STRING;
4
5 Part:
6     FCSS_Feature
7     | FCSS_Attribute
8     | FCSS_Global
9     | FCSS_View;
10
11 FCSS_Feature: feature=Feature_ID '{' attributes+=FeatureAttribute+ '}';
12
13 FCSS_Attribute: '#' attribute=Attribute_ID '{' attributes+=AttributeAttribute+ '}';
14
15 FCSS_View: '$' view=[tvdl::View] '{' attributes+=ViewAttribute+ '}';
16
17 FCSS_Global: '.' '{' attributes+=GlobalAttribute+ '}';
18
19 Feature_ID: head=[tvl::Feature_Declaration] ( tail=Feature_IDTail)?;
20
21 Feature_IDTail: '.' head=[tvl::Hierarchical_Feature] ( tail=Feature_IDTail)?;
22
23 Attribute_ID: head=[tvl::Feature_Declaration] tail=Attribute_IDTail
24               | head=[tvl::Attribute];
25 Attribute_IDTail: '.' head=[tvl::Feature_Scope] (tail=Attribute_IDTail)?;

```

An FCSS model is composed of an *import* statement for the TVL model, an optional import of a TVDL model corresponding to the TVL one, and at least one part (line 1). The TVDL import is not mandatory given that views are not always defined on the variability model. Please notice that, similarly to imports in the TVDL grammar, the name (STRING value) of the files is associated to the `importURI` parameter at line 3. In this way, scoping in the imported files is automatically handled by *Xtext*.

The parts of the FCSS model correspond to the different constructs which can be referenced in the TVL and TVDL models, namely features, attributes and views plus the global ones (see line 5 through line 9). They all have a similar

syntax: a starting symbol (except for features) followed by the construct they refer to, i.e., the qualified name of a TVL construct or the ID of a view. A global part does not have such a reference as it applies to several constructs. The key/value pairs are then declared between curly braces.

5.4.2 Default Infrastructure

The FCSS MWE2 file is really similar to the TVDL one. As a reminder, the `importURIScopingFragment` has to be set in order to get the default file scoping provided by *Xtext*. The single difference is that, for FCSS, the TVL and TVDL languages have to be declared into the MWE2 file:

```

1  bean = StandaloneSetup {
2      ...
3      registerGeneratedEPackage = "be.unamur.tvl.TvlPackage"
4      registerGenModelFile = "platform:/resource/be.unamur.tvl/src-gen/be/unamur/TVL.
        genmodel"
5      registerGeneratedEPackage = "be.unamur.tvdl.TvdlPackage"
6      registerGenModelFile = "platform:/resource/be.unamur.tvdl/src-gen/be/unamur/TVDL.
        genmodel"
7  }
```

The FCSS editor visible in Figure 5.9 is one of the (visible) results of running the MWE2 workflow.

5.4.3 Custom Developments

As for the TVDL grammar, FCSS custom developments were less complex than for TVL.

Model Validation

Some validation methods were implemented in the `FCSSJavaValidator` class in order to ensure the correctness of the FCSS model. As for TVDL, we will explain them in a descending order, going from higher to lower level rules.

The first thing checked by the `checkImports(Import imp)` method is that: 1) the first imported file is a TVL model, and 2) the second one (if any) is a TVDL file.

Then come the checks related to the different part types of an FCSS model. For each FCSS model part, we check that it is unique, i.e., no two parts refer to the same TVL or TVDL construct. For features, this task is handled by the `checkUniqueFeature(FCSS_Feature feature)` method, `checkUniqueAttribute(FCSS_Attribute att)` for attributes, and `checkUniqueView(FCSS_View view)` for views. The implementation of the different methods is similar: for a given construct A, get all FCSS parts ref-



```

Printer.tvl  Printer.tvdl  Printer.fcss
//Referencing the corresponding TVL model:
import "Printer.tvl"
//Referencing the corresponding TVDL model (optional):
import "Printer.tvdl"

//Global properties definition section
- {
    optFeature:checkbox;
    view:tab;
}

//View-specific properties section for "DocumentAttributes"
- $DocumentAttributes {
    label: "Properties of the document";
    help: "This tab contains all properties of the document";
    generate: true;
    unavailable: greyed;
}

//View-specific properties section for "GroupingView"
- $GroupingView {
    view>window;
}

//Feature-specific properties section for "Sheet"
- Sheet {
    help:"Properties of a sheet";
    unavailable:hidden;
    //Group properties
    - group {
        label: "Components of the sheet";
        container: true;
    }
}

//Feature-specific properties section for "Hole"
- Hole {
    opt:listbox;
}

//Attribute-specific properties section for "orient"
- #orient {
    label:"Orientation";
    help:"Orientation of the spine caption";
    widget: radiogroup;
}

```

Figure 5.9.: FCSS editor generated by *Xtext*

erencing that type of construct (feature, attribute, or view) and check that it does not refer to A.

As for TVDL, we make sure that the TVL construct referenced by a qualified name is unique. The `checkUniqueFeatureID(FCSS_Feature feature)` and `checkUniqueAttributeID(FCSS_Attribute att)` methods are thus quite similar to those defined for view validation.

A similar check is also available for key/value pairs: we ensure that the same key is not set twice for the same TVL or TVDL construct.

Finally, two very specific validations have been implemented. The first one is related to the *select* property. As a reminder, it will be used to determine the selection widget of a feature in the GUI if the feature's parent group is not rendered. The `checkSelectFeatureRoot(FeatureAttribute attribute)` method warns the user if this property has been set for the root feature. Indeed, it does not make sense to use the *select* property as root features do not have a parent group. The second one refers to the *view* property which determines how views will be rendered (tab or window) into the GUI. Setting this property in a grouping view will guide the generation of widgets for its sub-views but for view declarations, this property is useless. The `checkViewWidgetInView(GlobalAttribute attribute)` method thus checks that the *view* property has not been set for a view declaration.

Next comes the scoping. As for TVDL, we implemented TVL qualified names scoping in `scope_Feature_IDTail_head(Feature_IDTail id, EReference ref)` and `scope_Attribute_IDTail_head(Attribute_IDTail id, EReference ref)` methods of the `FCSSScopeProvider` class. For the *default* property of TVL groups we defined the scope as the set of group's sub-features (see `scope_GroupAttribute_defaultSubFeature(GroupAttribute att, EReference ref)`). As a reminder, this property determines which sub-features will be selected by default in the GUI. This is a temporary solution until a configuration-specific language is designed.

Editor

Contrarily to the two other editors, for FCSS we had to remove some default syntax highlighting rules. As depicted in Figure 5.9, most values of key/-value pairs are visually similar to keys. For example, the `optFeature` global property and its value, `checkbox`, have the same font. This hinders model understanding. In the `FCSSTokenHighlightingConfiguration` class, we filtered out the values and associated them to a distinct category which can be configured independently as depicted in Figure 5.10 (Property value token).

Besides token highlighting, we defined semantic ones for views, features and attributes in the `FCSSSemanticHighlightingCalculator` class. The views category is split into two: grouping and declaration ones. In order

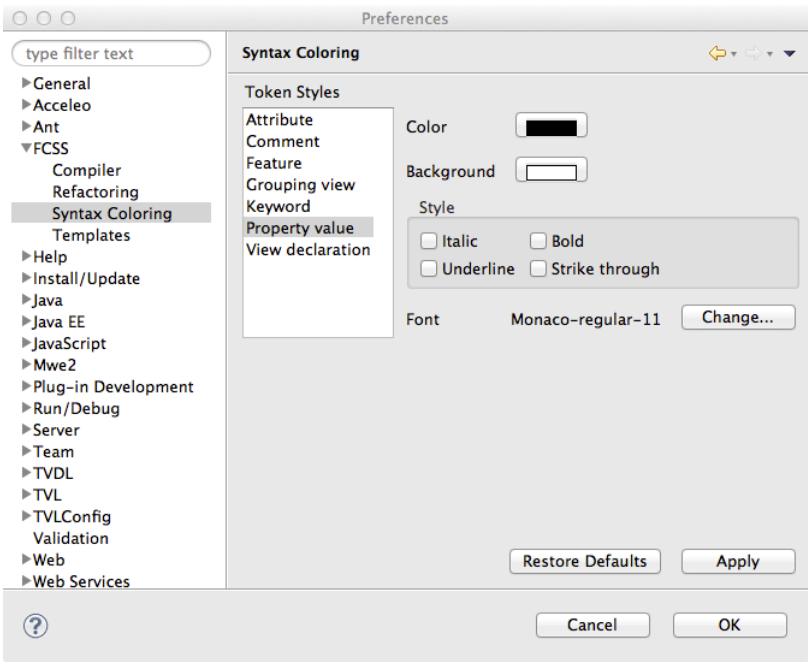



Figure 5.10.: FCSS syntax colouring preferences

to preserve visual coherence with the two other editors, default highlighting configurations have the same values (font weight, color, etc.). Users may change those values according to their preferences in the highlighting configuration window (see Figure 5.10). The result of our customizations is visible in Figure 5.11.

For the outline view, we applied the same principles as for TVL and TVDL in the `FCSSOutlineTreeProvider` class. The original and modified versions are visible in Figure 5.12. There, it can be seen that the default version misses a lot of elements. In FCSS, the hierarchy is not deep, contrarily to TVL. The main reason is that, according to the FCSS grammar, a beautification model is relatively flat.

The editor has been packaged and is available as an Eclipse plug-in at the following URL: <https://staff.info.unamur.be/qbo/Tools/>.



```

Printer.tvl  Printer.tvdl  Printer.fcsc
//Referencing the corresponding TVL model:
import "Printer.tvl"
//Referencing the corresponding TVDL model (optional):
import "Printer.tvdl"

//Global properties definition section
- {
    optFeature:checkbox;
    view:tab;
}

//View-specific properties section for "DocumentAttributes"
- $DocumentAttributes {
    label: "Properties of the document";
    help: "This tab contains all properties of the document";
    generate: true;
    unavailable: greyed;
}

//View-specific properties section for "GroupingView"
- $GroupingView {
    view>window;
}

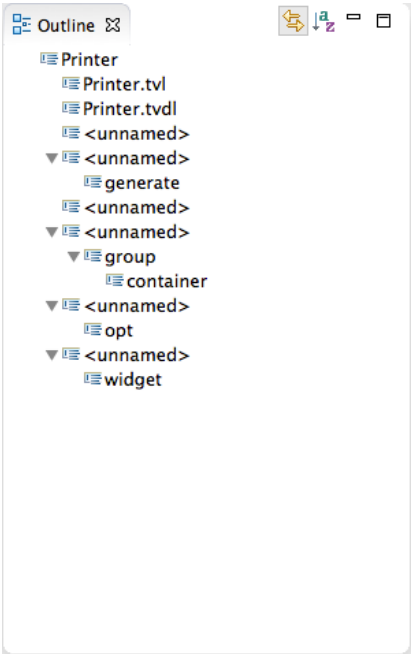
//Feature-specific properties section for "Sheet"
- Sheet {
    help:"Properties of a sheet";
    unavailable:hidden;
    //Group properties
    - group {
        label: "Components of the sheet";
        container: true;
    }
}

//Feature-specific properties section for "Hole"
- Hole {
    opt:listbox;
}

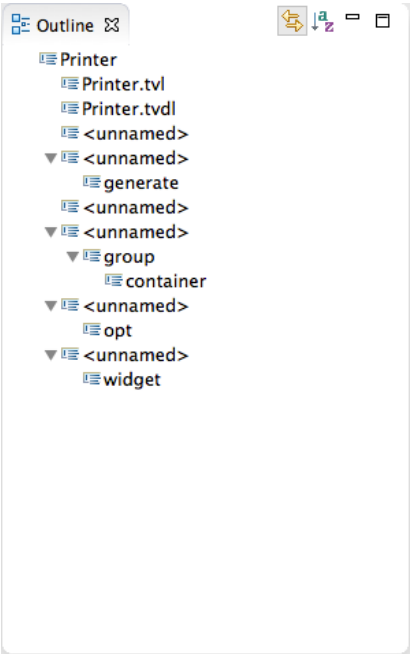
//Attribute-specific properties section for "orient"
- #orient {
    label:"Orientation";
    help:"Orientation of the spine caption";
    widget: radiogroup;
}

```

Figure 5.11.: FCSS editor with custom highlighting



(a) Generated version



(b) Customized version

Figure 5.12.: FCSS editor outline view

AUTOMATION

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 Chapter 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.

We thus had to skip the UIDL model in our MDE transformation chain to prefer a direct generation approach. For this proof of concept, we chose the *HTML5* language [W3C, 2014a], the latest version of the *HTML* standard. This version of the *HTML* language is completely defined since December 17, 2012 but has not yet been standardized by the W3C. This should be done in the last quarter of 2014. As previously mentioned, a lot of configuration interfaces are Web-based, as illustrated by Cyledge’s configurators database [Cyledge, 2013]. 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 [UsiXML Consortium, 2012]. In addition to the *HTML* target language for the static part of configuration GUIs, the presenter (see Chapter 3) is developed in JavaScript, its natural complement. In the following, we detail the *HTML* generator as well as how we implemented a generic JavaScript presenter relating the GUI with the reasoning components.

6.1 HTML INTERFACE GENERATION

As mentioned in Chapter 5, the various language editors developed in *Xtext* come with an EMF meta-model. Provided we have an EMF model as input and text (*HTML* code) as output, an M2T approach was a natural solution (see Chapter 2). More specifically, we used *Acceleo*, an implementation of the MOFM2T standard [Obeo, 2014]. Hereafter, we give an overview of the

Acceleo generator architecture. It can be decomposed into two distinct parts, queries and templates, which will be discussed in separate sections. Finally, the handling of feature instances by the GUI generator will be introduced in the last section.

6.1.1 Architectural Overview

As depicted in Figure 6.1, our implementation of model transformations takes the three models (TVL, TVDL, and FCSS) as input. Actually, *Acceleo* requires EMF model instances as input given that it supports model-to-text transformations. Fortunately, *Xtext*, which uses that kind of model to represent the AST, comes with some facilities to retrieve it based on file paths. The transformations are performed using *Acceleo* queries and templates whose architecture is depicted in Figure 6.2.



Figure 6.1.: Generation process with *Acceleo*

Two component types can be distinguished, namely *queries* and *templates*. Queries are depicted with dotted lines while templates are represented with plain ones. The different queries and templates will be discussed in the following sections.

As their name implies, queries are called by the templates to query the different models, i.e., TVL, TVDL and FCSS. For this reason, most of them are located in the `utils` package (bottom of Figure 6.2). `Models Importer` (top of Figure 6.2) is the exception as it is located in the `main` package. By default, *Acceleo* can import a single model. The `Models Importer` query provides a workaround to this limitation whose implementation will be detailed in the following section. In the `utils` package, queries are defined for each of the three different models, FCSS queries for the beautification model, TVDL queries for views, and TVL feature queries and TVL attribute queries for TVL. We chose to split TVL queries into two files in order to separate

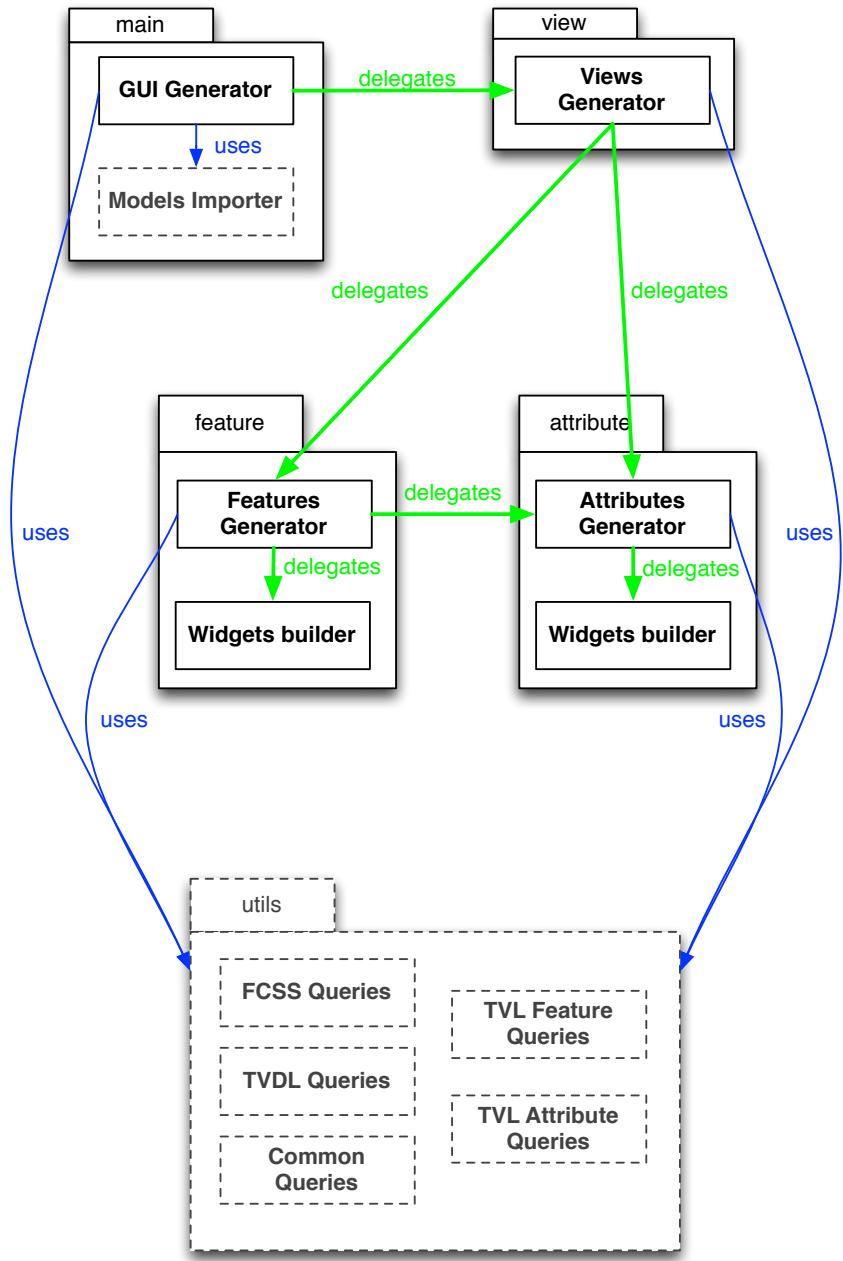


Figure 6.2.: Abstract architecture of our *Acceleo* solution

these two distinct concerns. Finally, the `Common` queries provide information which depends on several models (two or three of them). The other queries will be discussed in the following section.

On the other hand, templates are used to navigate through the TVL model and generate the target code, *HTML5* in our case. The *main* package is the entry point of our *Acceleo* generator. There, the *Generate Configurator* file delegates the code generation to the *view* package. The latter will in turn, depending on the view type (see Chapter 4), delegate the generation of the different TVL components covered by the view (i.e., feature or attribute) to the appropriate packages. In *feature* and *attribute* packages, the *Features/Attributes Generator* files retrieve the information about the construct to generate into the different models before delegating the generation of the corresponding *HTML5* code to the *Widgets builder* files. Widget builders simply create the widgets and do not require access to utilities given that all the information is provided by the caller (i.e., *Features/Attributes Generator*).

While templates are specific to the *HTML5* language, queries could be reused for any target language of model-to-text transformations. They could even be reused for other kinds of transformations, e.g., model-to-model ones provided that they use a similar language (i.e., OCL-like languages [Object Management Group, 2012]). This means that once a UIDL will have been selected, only templates will have to be rewritten. Furthermore, queries are far more complicated to implement than templates. In templates, we simply navigate through the TVL model in a depth-first algorithm and write *HTML5* code while in queries we have to search for information at several places. Rewriting templates for another target language is easy since it is sufficient to replace *HTML5* code excerpts.

In the following, we discuss the different components of this architecture. We start with queries, continue with templates and finish with the handling of feature instances.

6.1.2 Queries

In the *main* package, we have a single query file, `javaServices.mtl`, containing two queries. As discussed earlier, *Acceleo* can, by default, import a single model. However, in our generative approach, we have three input models, namely TVL, TVDL and FCSS. The queries thus provide a two-step workaround to get access to all models.

Given the model imported by *Acceleo* (preferably the TVL one but our approach works with all of them), the first query retrieves its fully qualified file path. In the second query, all models are retrieved based on that file path. We assume that the TVDL and FCSS files are located at the same place and have the same name (except for the file extension) as the file given as parameter.

This constraint might seem strong but our (prototype) solution can be easily adapted. Furthermore, we noticed that this naming convention is a consistent pattern in our industrial case studies. Listing 6.1 contains the second query.

Listing 6.1: Query retrieving the three models

```
[query public getAllModels(genericFilePath:String): OrderedSet(EObject) = invoke('
    TVLtoHTML5.main.ModelsGetter', 'getAllModels(java.lang.String)', Sequence{
        genericFilePath})
/]
```

In *Acceleo*, queries as well as templates are enclosed between square brackets ([/]). Queries start with the query keyword and have a visibility (private, protected or public). They also have a name (getAllModels in our example), parameters between parentheses (genericFilePath:String) and a return type preceded by a colon (OrderedSet(EObject)). Based on a file path (genericFilePath), our getAllModels query thus returns a list (OrderedSet) of EObjects. This list contains 3 elements, a Model (for TVL), a TVDLModel, and an FCSSModel. They are always returned in that order and all have EObject as super type. Then comes the implementation of the query itself, starting with the equals symbol. Two variants exist. A query can either call a Java method or be implemented in an OCL-like language. Here, the getAllModels query invokes (invoke keyword) a Java service. The other category will be introduced later in this section. An invoke query takes three parameters, the first two being strings. The first one is the fully qualified name of the class (TVLtoHTML5.main.ModelsGetter) containing the method to call. The latter is the second parameter (getAllModels(java.lang.String)). Notice that the fully qualified name of parameters types have to be provided. Finally, the parameters of the Java method are provided in the third parameter of the invoke construct. They are contained in a list, and must respect the order of the parameters of the Java method. In our example, the sequence contains a single element, namely genericFilePath, given that the getAllModels method of the TVLtoHTML5.main.ModelsGetter class requires a single parameter. The return type of the Java method has to be the same (or at least compatible) as the one of the query. Our Java method returns a HashSet of EObjects which is compatible with the OrderedSet of the query. They are then used by the templates to generate the *HTML* code, as discussed in the following section.

All other queries are located in the *utils* package, and split into five different *.mtl* files corresponding to the categories introduced in Figure 6.2. Each of those files is an *Acceleo* module.

TVL being the cornerstone of our approach, we start with queries related to that language. They are located in two different files, namely *tvFeatureQueries.mtl* and *tvAttributeQueries.mtl*.

Twenty-five feature-related queries exist but we will not go into implementation detail. Here, we just explain the role of the different queries and give an intuition of their implementation, if required.

- **getBodyItems** This query returns the list of feature body items relevant for GUI generation, i.e., attributes and groups. It invokes the `getFeatureBodyItems` Java method of the editor described in Chapter 5.2. The result is filtered to reject TVL data and constraints which are not relevant in the static part of the GUI.
- **getContainingFeature** Two versions of this query exist, with different parameters. The first one takes a hierarchical feature as parameter while the second one requires a group. They both invoke Java methods (i.e., `getContainingFeature`) of the TVL editor and return the feature containing the parameter feature/group. The result cannot be *null* as hierarchical features and groups are always declared inside another feature. Only the root feature has no parent feature but is not a valid parameter for this method.
- **getContainingGroup** Given a feature declaration, this method returns the group where it is declared or *null* if it is the root. As for previous queries, this one invokes a Java method, namely `getContainingGroup`.
- **getGroup** This query returns the group declared inside the feature declaration given as parameter. If it contains no group, *null* is returned. This query relies on the `getBodyItems` one.
- **getOptFeature** The query returns an optional feature of the group given as parameter. If the group contains several optional features, one of them is chosen randomly. Conversely, if no such feature exists, *null* is returned. It is implemented in Listing 6.2.

Listing 6.2: Query retrieving an opt feature from a group

```
[query public getOptFeature(group:tv1::Feature_Group): Sub_Feature=
  if (group.sub_features->exists(sub:Sub_Feature | sub.resolve().isOpt()))
    then group.sub_features->any(sub:Sub_Feature | sub.resolve().isOpt())
  else null
endif
/]
```

The declaration is the same as for *invoke* queries, only the implementation varies. Here, in the *if* condition, we check that, in the set of group's sub-features, at least one is optional. If it is the case, one of them is returned using the *any* query. Otherwise, *null* is returned.

- **isOpt** This query checks whether a feature declaration is optional or not. It returns *false* for the root feature. For hierarchical features, it returns *true* if and only if it is declared optional or has a [0..1] cardinality.
- **isClonable** Checks if several instances of a feature can exist. The result is *false* for the root feature as it is not clonable. For other features, we check the maximum number of instances. If it is greater than one, *true* is returned.
- **isEmpty** Checks if a feature declaration contains a group and/or attributes. Its implementation is based on the `getBodyItems` query.
- **isJustGroup** This query returns *true* if a feature declaration contains a single element which is a group. It is also based on the `getBodyItems` query. We check that the latter contains a single element of type group.
- **getFeatureCardinality** Returns the minimum and maximum number of instances for the hierarchical feature given as parameter. If no cardinality is defined in the TVL model, minimum and maximum values both equal to one.
- **getBasicCardinalityBounds** Transforms a given cardinality into a sequence of two integers, the first element being the minimum.
- **resolve** Sub-features of a group can either be hierarchical or shared ones. For the latter, this query returns the hierarchical feature they refer to. If the sub-feature is a hierarchical one, it is directly returned by the query.
- **getBounds** Returns a sequence of two integer values corresponding to the cardinality of a group. Syntactical sugars (i.e., *allOf*, *someOf*, *oneOf*) are also transformed into integer values.
- **getType** Retrieves the type of a feature group as a string value. Available values are *and*, *or*, *xor*, and *card*. Its implementation is based on the previous query and checks the minimum and maximum values to determine the type of the group given as parameter.
- **getLongID** Builds the dot-separated fully qualified name of a feature declaration. It is built recursively by appending the parameter feature name to the one computed for its parent.
- **getShortID** Returns the name of the feature declaration given as parameter.
- **isChild** Checks if a feature declaration is declared inside a group, both given as parameters. For this purpose, we check that the set of group's sub-features contains the feature.

- **isDescendant** This query plays a similar role to the previous one but it explores all group's sub-features recursively. Consequently, it returns *true* if and only if the parameter group is an ancestor of the parameter feature declaration.
- **containsGroup** This query takes two parameters, a feature declaration and a list of groups. It returns *true* if at least one of the groups contained in the list is declared by the feature declaration or one of its descendants. If the feature declaration has no group, *false* is returned and no descendant has to be explored.
- **containsFeature** Returns *true* if a feature declaration (second parameter) is a direct or indirect child of another one given as first parameter.
- **isInTheList** Two variants of this query exist, both with two parameters, the second one being a list of EObjects (i.e., TVL constructs). The first version searches for a feature declaration in that list while the second looks for a group. The feature declaration/group is the first parameter of the query. It returns a Boolean value.
- **getRootFeature** Returns the root feature of the TVL model given a feature declaration.
- **getAllSubFeatures** Returns a sequence (list) of hierarchical features, the sub-features of the feature declaration given as parameter. If the latter has no group, the empty list is returned.

Queries have also been defined for TVL attributes. They are located in the *tvAttributeQueries.mtl* file. Their role is to provide accessors and navigators for attribute-related information in the TVL model. The twenty queries are mentioned hereunder.

- **getBaseAttributeType** Returns the type of the TVL attribute given as parameter. As a reminder, attributes can have a predefined (*int*, *real*, etc.) or user-defined type. If the type is a predefined one, it is returned. If it is a user-defined one, the query retrieves its definition and resolves its type. The type is returned as a string value
- **getRecordFieldType** Plays the same role as the previous query for attributes of type record. The returned value is also a string.
- **getAttributeCardinality** Returns the number of instances of an attribute as a sequence of two integers corresponding to minimum and maximum bounds, respectively, the empty sequence if the attribute cannot be instantiated.

- **isOpt** Checks if an attribute is optional, i.e., has a [0..1] cardinality. The returned value is a Boolean.
- **getBaseAttributeDefaultValue** Returns a string containing the default value of an attribute if it is defined, the empty string otherwise. A default value is defined for an attribute if an *is* expression exists for that attribute.
- **getRecordFieldDefaultValue** The role and behaviour of this query is the same as for the previous one except that it applies to record fields.
- **getBaseAttributeDomain** Retrieves the domain of the integer, real or Boolean attribute given as parameter. This domain is generally defined with an *in* expression. The result is a sequence containing the domain values if they are defined, null otherwise.
- **getEnumAttributeDomain** Plays the same role as the previous query for enumerated attributes.
- **getRecordFieldDomain** Similar to **getBaseAttributeDomain** for integer, real, or Boolean record fields.
- **getEnumRecordFieldDomain** Plays the same role as **getEnumAttributeDomain** for enumerated record fields.
- **getAttributeLongID** Returns a string containing the dot-separated fully qualified name of the attribute given as parameter. As for feature long IDs, it is built recursively.
- **getRecordFieldLongID** Requires two parameters, a record field and its containing structure attribute to generate the fully qualified name of the first one. Indeed, in TVL, structure attributes must refer to user-defined types which contain record fields. As a consequence, if several attributes refer to the user-defined type, the query has to know to which structure attribute it is attached. The result of this query is a string containing the fully qualified name of the field including its parent structure attribute's name.
- **getAttributeShortID** Returns a string containing the name of the attribute given as parameter.
- **getRecordFieldShortID** Returns a string containing the name of a record field. It is composed of two parts separated by a dot. The first one is the name of the structure attribute (second parameter of the query) containing the record field (first parameter) whose name is located in the second part.

- **isChildAttribute** Returns a Boolean value indicating whether an attribute is declared inside a feature, both given as parameters. The `getBodyItems` feature query is used to retrieve all feature's body items and check if it contains the attribute.
- **isDescendant** Plays the same role as the previous query except that it explores the feature's descendants. While `isChildAttribute` checks the direct children, this one explores all descendants and stops as soon as the attribute has been found.
- **getContainingFeature** Searches for the feature declaration where the attribute given as parameter is declared. For this purpose, the query uses the `getContainingFeature` Java method of the TVL editor.
- **isClonable** Returns a Boolean value indicating whether or not an attribute can be instantiated several times. For this purpose, it uses the `getAttributeCardinality` query defined earlier.

Next come queries related to the TVDL model. The idea is the same, facilitate the access to the information contained in the view model. However, they will be presented in a different manner given that they can be grouped into categories of similar queries.

1. **Coverage queries.** Three different queries returning coverage have been defined. `getModelCoverage` returns the coverage of the whole model given as parameter. The next one, `getViewCoverage`, requires a view as parameter and returns its coverage. Finally, `getViewExpressionCoverage` returns the coverage of a view expression. The three queries return a sequence of EObjects, the latter being feature declarations, feature groups, attributes and sub-attributes. They are all implemented by invoking Java methods.
2. **Refinement queries.** As explained in Section 4.2, view expressions can be refined in TVDL. The `isRefined` query checks if it is the case for the expression given as parameter. Then, if the answer is positive, one might be interested in knowing the type of refinement. Four of them exist, namely sub-tree, attributes, groups, and lists. A query exists for each of them. All of them call the `isRefined` query to ensure that the view expression is refined. The `isSubtree` query checks if the refinement corresponds to a sub-tree. The result is a Boolean value. If the answer is positive, the sub-tree type can be found with `isSubtreeStructure`, `isSubtreeAttributes`, or `isSubtreeGroups` queries corresponding to the different sub-tree refinement types defined in Section 4.2. It is also possible to get the stop list of a sub-tree expression thanks to the `getStopList` query. It returns a sequence of list

elements (see Section 4.2) and uses the `getStopList` Java method of the TVDL editor. The `isAttributes`, `isGroup`, and `isList` queries play the same role (and are implemented in a similar way) as `isSubtree` for the other refinement categories.

3. **Utils.** Here, we present queries returning information common to most (if not all) views or view expressions. The first one, `contains`, requires two parameters, a sequence of list elements corresponding to a sub-tree stop list, and a feature declaration. It returns the *true* Boolean value if the list contains the feature. For this purpose, it invokes the `covers` Java method. Then, the `resolveID` query resolves the TVL construct given as parameter, i.e., it resolves qualified names. The returned element is either a feature declaration or an attribute. `getID` returns the dot-separated fully qualified name of a view. The latter can be contained by grouping views. For this reason, a list of its ancestors is also required to generate the qualified name. The empty list is used if the view is not contained by any other view. A second implementation, similar to the first one, exists for view expressions. In this case, the `ancestors` list contains at least one element, the view where the expression is declared. Then comes the `isContained` query which checks whether a view is contained in any other view. It returns a Boolean value.

Queries have also been defined for the last language, FCSS. As for TVDL, we will introduce them using categories. We identified two of them. The first one contains properties which can be located at different places while the second one covers properties which are defined at a single pre-defined place. They are all located in the *fcssQueries.mtl* file.

1. **Search at different levels.** In Section 4.3, we have seen that some properties can be defined at different levels. For example, this is the case for the widget of a feature which can be defined at three different levels, namely feature-specific ones, in its containing view(s), or at a global level. As a reminder, more specific properties have priority over others. For such properties, queries have all been implemented using the same pattern. A single public query exists, `getFeatureWidget`. It first calls a private query which checks if a specific property has been defined. In our example, it is `getFeatureSpecificWidget` which checks if 1) a specific entry exists for the given feature, and, if it is the case, 2) searches for the *widget* property. If it exists, then its associated (string) value is returned. If not, the empty string is returned. In the first case, the public query returns the value, in the second case it calls a query which checks if the property has been defined at the view level. It is called `getFeatureViewWidget` for our example. Again, it checks if a view-specific entry exists for the view containing the feature. If

it is the case, it searches for the *feature* property and returns its value if it is found, the empty string otherwise. If the property is neither found at the feature- nor view-level, the public query will finally call a global level one which will check if the given property has been defined at the global level. The behaviour of the latter is the same as other ones (check if a global part exists, and, if it is the case, search for the property), and returns a string value if a value is found or the empty string if it is not the case. As a consequence, the public query will return an empty string if nothing is found. This value has to be handled by templates discussed in the next section. In our example, the last query is called `getFeatureGlobalWidget`. This pattern was used to determine optional feature widgets (`getOptFeatureWidget`), select feature widgets (`getFeatureSelectWidget`), group widgets (`getGroupWidget`), base attribute widgets (`getBaseAttributeWidget`), whether a container widget has to be displayed for a group (`isGroupContainer`), and a display strategy for unavailable contents, views, groups or attributes (`getFeatureUnavailableStrategy`, `getViewUnavailableStrategy`, `getGroupUnavailableStrategy`, and `getAttributeUnavailableStrategy`).

2. **Search at a single place.** Oppositely, some properties are defined at a single level. For them, it is sufficient to search for the given property and return its value if it exists. If it is not found, the empty string is returned in order to be consistent with the previous category of queries. The queries in this group allow to get the following information: get views widgets (`getGlobalViewWidget` and `getGroupingViewWidget`), to check whether or not a view has to be generated (`isGenerateView`), to retrieve view, feature, group, and attribute labels (`getViewLabel`, `getFeatureLabel`, `getGroupLabel`, and `getAttributeLabel`), to get the help text for those same constructs (`getViewHelp`, `getFeatureHelp`, `getGroupHelp`, and `getAttributeHelp`), and to get the default feature of a group (`getGroupDefaultValue` and `isGroupDefault`).

Even if all models have been covered by queries, a fifth module contained in the *commonQueries.mtl* file exists. As its name suggests, the file contains a few queries which require information from several (if not all) models.

1. **Check if an element has to be generated.** Several queries, effectively five, exist to determine whether or not a construct has to be rendered in the configuration GUI. They are all named `toGenerate`, only parameters change. We describe them in descending order. Views come first. A view has to be generated if its *toGenerate* property is set to *true* or it has contents which have to be generated. Generally, that contents

are a (set of) view expression(s). View expressions represent the second level. A view expression has to be generated if its coverage is not empty. For sub-tree structures, we also have to check that the sub-tree has to be generated. Determining whether or not a feature has to be rendered given a sub-tree is a bit more complex. If it is not covered by the sub-tree, then the answer is negative. Otherwise, if the feature is in the sub-tree stop list, then it does not have to be generated. If a feature passes both tests, we have to check if it has contents to generate. It is the case if it contains attributes and/or a group which has to be generated. The last `toGenerate` query takes care of groups. A group does not have to be generated in the context of a sub-tree expression if it is not in its coverage list.

2. **Determine whether an element is available.** In our configuration GUIs, we can set a flag indicating whether or not an element is available (generally displayed) when the Web page is initialised. Basically a feature is available if all its ancestors are. This is checked by the `isIndependentFeatureOnlyAvailable`. Furthermore, the contents of a feature are available if the feature itself is available and it is selected (see `isIndependentFeatureAvailable`). The availability of attributes and groups is the same as their containing feature, as implemented in `isIndependentAttributeAvailable` and `isIndependentGroupAvailable`.

A few other common queries are also defined in *commonQueries.mtl* but, given that they are very specific utils, we will not explain them here.

6.1.3 Templates

As described in the introduction of this chapter, templates are used to navigate through the three different models. Here, we discuss in turn the templates defined in the four packages of Figure 6.2.

1. Main package

This is the entry point of the generation approach. It contains a single template named *generateConfigurator*. Its source code is visible in Listing 6.3. We will use this example to describe the different components of an *Acceleo* template. First, as for queries, a template must be declared. The declaration begins with the `template` keyword and is followed by its visibility, name and parameters between parentheses. For example, at line 1 of our example, a public template called `generateConfigurator` is declared. It requires a single parameter named `theFeatureModel` of type `Model` in the TVL language. It ends

at line 44. In the body of a template, one can distinguish constructs from two languages. The first is *Acceleo* while the second is the target language, *HTML5*. On the one hand, *Acceleo* expressions are written between square brackets, meaning that they have to be evaluated. On the other hand, *HTML5* will be written as-is in the output file.

In our example, the output is a file named `Configurator.html` encoded in UTF-8 as declared at line 3. There, the second parameter (`false`) means that the result of the generation does not have to be appended to the contents of the file (if any). The file is closed at line 43, i.e., all *HTML5* code generated between line 3 and line 43 will be saved in `Configurator.html`.

Then come the `let` constructs which declare *Acceleo* variables for the current template (see lines 4-9). Even if it is not illustrated in our example, variables can be declared anywhere in the template. All `let` declarations have to be closed (lines 37-42). The first end expression closes the last variable declared. For example, `rootFeature` is declared at line 9 and closed at line 37. Variables can be assigned constant values or the result of a query. For example, at line 5, the result of the `getAllModels` query defined in Listing 6.1 is assigned to the `models` variable.

The *HTML5* code is generated from line 10 to line 36. There, most lines of code simply print *HTML* code like the head and body of the Web page. Lines 22, 27, 28 and 31 are exceptions. The three firsts are query calls while the last one is a template call. Two call mechanisms are available for queries. A query is either called as it has been declared (i.e., its name followed by all parameters between parentheses) or in a Java-like manner for methods of an object by calling the query on its first parameter. For example, at line 22, the `getFeatureLabel` query requires two parameters (see previous section). Here we chose to call it on its first parameter, namely `fcssModel`. Line 31 is an example of template call. It takes care of the generation of the configuration form and will be discussed in next sub-sections. A template can call public templates contained in any module imported by its own module (i.e., its container). A module contains several templates and is contained in a *.mtl* file with the same name. Oppositely, queries can call other queries but not templates.

Listing 6.3: Main template

```

1  [template public generateConfigurator(theFeatureModel:tv1::Model)]
2  [comment @main/]
3  [file ('Configurator.html',false,'UTF-8')]

```

```

4  [let genericfilepath:String = theFeatureModel.eResource().getFilepath()]
5  [let models:OrderedSet(EObject) = getAllModels(genericfilepath)]
6  [let featureModel:tv1::Model=models->at(1)]
7  [let tvdlModel:tvdl::TVDLModel=models->at(2)]
8  [let fcssModel:fcss::FCSSModel=models->at(3)]
9  [let rootFeature:tv1::Root_Feature=featureModel.model->selectByType(Root_Feature).
   oclAsSet()->asSequence()->first()]
10 <!DOCTYPE html>
11 <html>
12   <head>
13     <meta charset="utf-8" />
14     <link rel="stylesheet" href="style.css" />
15     <script type="text/javascript" src="jquery.js"></script>
16     <script type="text/javascript" src="tristate.js"></script>
17     <script type="text/javascript" src="controller.js"></script>
18     <script type="text/javascript" src="fancySliding.js"></script>
19     <script type="text/javascript" src="clonesHandling.js"></script>
20     <title>
21       [comment: The title of the web page is the name of the root feature/]
22       Configurator [fcssModel.getFeatureLabel(rootFeature)/]
23     </title>
24   </head>
25   <body>
26     [comment: The title of the web page is the name of the root feature/]
27     <h1 title="[fcssModel.getFeatureHelp(rootFeature)/]">
28       [fcssModel.getFeatureLabel(rootFeature)/]
29     </h1>
30     <form id="configuratorForm">
31       [generateTVDLModel(tvdlModel,fcssModel)/]
32     </form>
33     [comment: A div reserved for displaying messages/]
34     <div id="messages" class="messages"></div>
35   </body>
36 </html>
37 [/let]
38 [/let]
39 [/let]
40 [/let]
41 [/let]
42 [/let]
43 [/file]
44 [/template]

```

2. View package

This package contains a single module named *generateViews* which itself contains six templates, one public and five privates.

The public template is the single entry point of the module and is named *generateTVDLModel*. Based on a TVDL and an FCSS model, it dispatches the generation of view widgets to the corresponding template. However, at the moment it is useless as a single widget has been im-

plemented for views, namely tabs. In the future, we intend to propose other widgets like, e.g., windows.

Building the tabs is handled by the *buildTabs* template. It takes four parameters. The first one is the list of views to generate. The second is a Boolean value indicating whether the views are direct children of the TVDL model or not (contained in grouping views). This value is used by queries to check if a view has to be generated. The third parameter is a sequence of views, ancestors of the first parameter. If the second parameter is set to true, this sequence will obviously be empty as first level views do not have ancestors. Finally, as for most (if not all) templates, the FCSS model is given as parameter. The template generates two *HTML divs*. The second one contains code to navigate through the different tabs while the first one contains the code corresponding to all views given as first parameter of the template. For this purpose, we iterate through the list and, if a view has to be generated, generate the code for its contents by calling the *buildView* template. Notice that for views represented as windows, we will have to call that same template as the contents of a view should be generated independently of the representation of the view itself.

Given a TVDL view, the list of its ancestor views and the indispensable FCSS model, the *buildView* template generates the *HTML* code related to a view. Each view is contained in its own *div* whose ID is the dot-separated name of the view and its ancestors, and help text is provided by the FCSS model. A label is then defined for the view based on the *getViewLabel* query. Then, we distinguish between view declarations and grouping views. In the first case, we iterate through the view expressions. For each of them, we create an *HTML div* if the *toGenerate* query returns a positive answer. Its ID is the ID of the containing view's *div* to which a view part number has been appended. The generation of the view expression contents are then delegated to the *generateViewExpression* template.

The *generateViewExpression* template dispatches the generation of the view expression given as parameter to the corresponding template. For this purpose, it also requires the list of ancestor views of the view expression and the FCSS model. In this case, the list contains at least one view as a view expression must be declared inside a view. There are two possible cases: either the expression refers to a TVL feature or to an attribute. In the second case, we call the *generateIndependentAttributes* template contained in the *attribute* package. In the other case (i.e., the view expression refers to a TVL feature), we dispatch the generation depending on the type of the view expression. Four different cases exist. The first one covers features which are not refined or refined by a list

(inclusion or exclusion). In that case, the coverage is a list whose first element is the feature followed by (some of) its body items and the *generateIndependentList* template is called (*feature* package). The view expression can also be a sub-tree one. Given that several refinements exist for sub-tree expressions, they are handled in an independent template named *generateSubtreeExpression* and discussed in the following paragraph. Third, the expression can also be an attribute one. In that case, the coverage is a list of attributes (and possibly sub-attributes) whose handling is delegated to *generateIndependentAttributes* template (*attribute* package). Finally, it can also be a group expression. In that case, the list of groups is handled by the *generateIndependentGroups* from the *generateFeature* module.

The *generateSubtreeExpression* template takes four parameters: a TVL feature (the root of the sub-tree), a TVDL sub-tree expression, the list of ancestor views for the sub-tree expression, and the FCSS model. Sub-tree expressions preserving the structure of the FM are directly handled by the *generateIndependentSubtree* template from the *generateFeature* module. For other ones, we distinguish three cases similarly to what has been done in *generateViewExpression*. A sub-tree expression refined by an attribute one is handled by the *generateIndependentAttributes* as it covers attributes only. If it is a group refinement, the coverage list contains groups only which will be handled by the *generateIndependentGroups* template from the *feature* package. Finally, sub-tree expressions refined by a list have a coverage list with the following pattern: a TVL feature followed by some of its body items. The list contains at least one instance of this pattern. This case is handled by the *generateIndependentList* template. The latter was also used for non-refined expressions in *generateViewExpression* which have the same list pattern.

Finally, *dispatchViewGrouping* is similar to *generateTVDLModel* except that it handles grouping views. Its role is to dispatch the generation of the correct widget for the grouping view. At the moment it refers only to the *buildTabs* template for the reasons mentioned earlier.

3. Feature package

In the first version of our generator, TVDL and the *view* package did not exist. In that version, the hierarchy of the FM was directly rendered in the configuration GUI. Different traversal strategies exist for trees like depth-first or breadth-first. Among them, we chose the depth-first one as it allows to visually group a feature together with its contents (i.e., sub-features and attributes). Representing the whole FM on a single Web page is acceptable for small examples. For larger and most real

ones, limits are quickly reached: long Web pages, tree structure visible in the GUI, lost spaces due to the page setup, etc. For all these reasons, in the second version of the generator, views were added. One of the impacts is the *view* package previously introduced. But major changes were located in the *feature* package. Indeed, the whole tree had to be explored in the first version while views contain parts of it. In order to avoid confusion, we present only the second version, the first one being a specific case of it.

This package contains two modules, namely *generateFeature* and *buildFeatureWidget*. In the second, templates simply print *HTML* code based on the information provided by their callers. We followed the *buildXXX* naming convention for all of them. They will not be discussed here. In the *generateFeature* module, templates can be categorized along several dimensions. The first one we chose is their visibility (i.e., public *vs.* private). The private category being larger, we defined two sub-categories, namely dispatchers and generators. The different roles will be explained in their specific paragraphs.

Three public templates were implemented. As depicted by the single *delegates* arrow between *view* and *feature* packages in Figure 6.2, they are all called from view templates and have already been mentioned in the corresponding section. Here, we use the order in which they have been previously introduced. The *generateIndependentList* template generates the contents of lists of features and their respective (partial) contents. For this purpose, it requires 1) a feature, 2) a coverage list, 3) the view expression with that coverage list, 4) the list of ancestor views, 5) the short ID of the feature (first parameter), and 6) the FCSS model. The first parameter is either a feature contained in the coverage list or an ancestor of one of those features. In the first case, we simply iterate through the coverage list to find a feature and its contents and delegate their *HTML* rendering to the *generateIndependentList* private template discussed later. The second case is used to keep track of the FM hierarchy everywhere in the configuration GUI. As previously mentioned, we use *HTML divs* to represent views, features, etc. The role of this alternative behaviour is to build the hierarchies of features, ancestors of features contained in the coverage list. Keeping track of this hierarchy is required, amongst other things, to know if ancestor features are selected. This point will be further discussed in Section 6.2. Second, the *generateIndependentGroups* template handles the generation of a group view expression. Its parameters are 1) a TVL feature, 2) a coverage list exclusively composed of feature groups, 3) the view expression with that coverage list, 4) the list of ancestor views of the third parameter, 5) the short ID of the first parameter, and 6) the FCSS model. The role of

the first parameter is the same: keep track of the hierarchy. It means that, in the *generateViews* module, the template will always be called with the root feature as first parameter as it is the starting point of the TVL model hierarchy. Oppositely, for recursive calls, the feature will be the ancestor of at least one of the covered groups (second parameter). The template's behaviour is the same; it first builds the ancestors' hierarchy before delegating its widget generation to the private *dispatchGroup* template. Finally, the *generateIndependentSubtree* template applies a similar mechanism to generate all widgets corresponding to a sub-tree view expression. The feature hierarchy is built from the FM root feature down to the sub-tree root feature before delegating the sub-tree generation to the *generateSubtree* template.

In private templates, dispatchers retrieve information from the TVL and FCSS models in order to dispatch widget generation to the correct generator template. The *dispatchOpt* template checks, for a given feature, which widget has to be used to represent its optionality and delegates its generation to the corresponding template in the *buildFeatureWidget* module. The behaviour of *dispatchSelect* is similar and is not discussed here. At the moment, the *dispatchFeature* template always delegates feature widget generation to the *buildFeatureLabel*. In the future, our generator should support image widgets for features. Dispatchers also exist for groups. *dispatchGroup* retrieves the type of a group and delegates it to the corresponding dispatcher. Three group-type dispatchers exist: *dispatchOr*, *dispatchXor*, *dispatchCard*. They all retrieve information from the FCSS model in order to dispatch the group to its corresponding generator template. No dispatch template exists for *and*-decompositions given that there is only one widget for them.

All other templates are generators. Their role is to generate *HTML* code and go through coverage lists and the TVL model. Most of them have three parameters in common: the FCSS model, a view expression, and the list of ancestor views of that view expression. We start with the *generateSubtree* template which builds an *HTML div* for the sub-tree root containing a label created by the *buildLabel* template and the sub-tree contents delegated to the *generateContent* template. The latter creates a content-specific *div* which contains all feature body items. We iterate through the items and delegate their generation either to *generateAttribute* or to *dispatchGroup*, depending on their type.

Most other templates contained in the *generateFeature* module are group-related. *And*-decompositions are handled by *generateAnd*. There, a *div* is created for the group. It optionally contains a *div* for the group label if it has been defined in the FCSS model. The group sub-features are then handled by the *generateAndSubFeatures* template. The latter creates

an enclosing *div* for the group. It also iterates through the sub-features, creates a label for each of them and delegates their contents generation to *generateContent* if required. Contents have to be generated if and only if the view expression given as parameter is a sub-tree one which covers the feature's contents. Other view expressions are handled in other templates like, e.g., *generateListElement* for refinement lists. In *generateListbox*, the generation is decomposed into two steps. First, a select box containing an option for each of the group's sub-features is created in the label *div*. Second, in the contents *div*, a *div* is created for each feature. They all contain a *label div* whose contents are generated by *dispatchFeature* and a *contents div* delegated to *generateContent*. *generateCheckbox* and *generateRadio* are similar. Similarly to *generateAnd*, they create a *div* for the group optionally containing a label. Then, in the contents *div*, a widget (check box or radio button) is created for each sub-feature. Furthermore, the contents generation is delegated to *generateContent* if required. Generation conditions are the same as for and sub-features.

Finally, the two remaining templates handle coverage lists composed of a feature and its contents. *generateListElement* generates a feature declaration and its contents contained in such a list. The label is built by the *buildLabel* template and the contents are handled by *generateListFeatureContent*. In that template, we iterate through the list and delegate attribute and group generation to *generateAttribute* and *dispatchGroup*, respectively. Then, if group's sub-features have to be rendered, widget generation is delegated to the *generateIndependentList* template discussed earlier.

4. Attribute package

The attribute-related package is composed of two modules, namely *generateAttribute* and *buildAttributeWidget*. As for features, the second one will not be addressed here since it simply prints *HTML* code based on the information provided by calling templates. Here, we focus on the *generateAttribute* module. Templates contained in that module are far less complex than the previous one. Indeed, most attributes are leaves of the FM and do not need to be explored. The only exception to this are structure attributes.

Two different public templates exist for attributes. *generateIndependentAttributes* generates the GUI corresponding to a list of attributes. As for features, we keep track of the entire FM hierarchy by creating a *div* for each ancestor feature of each attribute contained in the list. Once this hierarchy has been created, the generation of the attribute widget

is delegated to the *generateAttribute* template. In the first version of our generator, the latter was the single entry point (i.e., public template) of its module. Depending on the attribute given as parameter (base or structure), it simply delegates generation to the corresponding template (*generateBase* or *generateStruct*).

The *generateBase* template creates a *div* for the attribute containing its label (built by *generateLabel*) and its contents (built by *generateAttributeContent*). The latter template simply checks the type of the attribute given as parameter and dispatches its generation to the corresponding template. *generateIntAttribute* takes care of integer attributes. It retrieves information from the FCSS model and builds the correct widget by calling either *dispatchDefaultNumberWidget* or *buildNumberSlider*. The behaviour of *generateRealAttribute* is similar except that the step for text input is a real value. In *generateBoolAttribute*, the task is delegated to *buildBoolCheckbox*, *buildBoolCombobox*, or *buildBoolRadio* templates. *generateEnumAttribute* delegates enumerated attributes generation either to *buildEnumCombobox* or to *buildEnumRadioGroup*. If no information has been defined in the FCSS model, the *generateEnumDefaultWidget* template is called. There, we generate a radio group for enumeration with three values or less, and a combo box for other ones. In *generateStringAttribute*, a text box is built by *buildStringTextbox*.

The handling of structure attributes by *generateStruct* is similar to the one of base attributes. A *div* is created for each attribute. It contains a *div* for its label and its contents generation is delegated to *generateStructAttributeContent*. In that template, we iterate through all sub-attributes of the structure. Given that it is, at the moment, not possible to customize sub-attributes in the FCSS model, they are directly handled by *generateNotClonableStructAttributeContent* except for labels which are handled by *generateLabel*. As a consequence, number (i.e., integer and real), Boolean, enumeration, and string attributes will always be rendered by the same widgets, regardless of their containing structure attribute.

6.1.4 Handling Feature Instances

In Section 4.1, we introduced TVL₂ which adds, inter alia, feature instances. Feature instances are not supported by the generator introduced in the previous sections. But our industrial partners have demonstrated a keen interest in that kind of variability construct meeting a real need in their day-to-day work. A third version of the generator has been developed to take them into account.

At first, our intent was to extend the existing generator. That solution quickly showed its limitations. Indeed, Web pages generated by the second version of the generator are static. That means that rendered *HTML* pages contain all widgets for all TVDL views generally covering the whole TVL model. While this solution is acceptable for a simple FM, it is not adapted to models containing feature instances. For such features, the Web page would contain as many duplicated blocks of *HTML* code as its upper bound in order to be sure to be able to display the required number of clones. There are three main problems with that approach. First, the number of instances can be unlimited (using the asterisk character as upper bound). Rendering an infinite number of blocks of *HTML* code is impossible. The only solution would therefore be to set an arbitrarily high upper bound. Though this is possible in some contexts, it is not the case for all of them. Then, even if a high upper bound is set, the size of the *HTML* page would be huge. In our *Rexel* case (discussed in Chapter 7), we set the unlimited upper bounds to three in order to check the size of the generated file. Even if that number is very small for the unlimited bound, the generated file had a size larger than 20MB. This is definitely not acceptable for a Web page. Finally, the *Acceleo* implementation also has its own limits. In the same *Rexel* example, the tool was not able to generate more than three instances of the same feature, mainly due to the nesting of such features which is not supported by the heap underlying *Acceleo* implementation.

The chosen alternative is to handle dynamic Web pages. A feature instance is added (removed) from the *HTML* configurator according to user inputs through a Web service. The details about it will not be discussed here. All we can say about it is that it stores a parametrizable instance of each clonable feature and injects it on request at the right place in the *HTML* file. Here, we just discuss how the generator handles such features by giving a hint of their *Acceleo*'s implementation.

generateFeature is the only module impacted by feature instance support. Some templates like *generateContent* or *generateListFeatureContent* had to be duplicated in order to distinguish instanciable (e.g., *generateClonableContent*) from non-instanciable (e.g., *generateNotClonableContent*). The major change in such templates is the addition of a list box to select a feature instance. Such widgets also had to be added in templates were the FM hierarchy (actually a hierarchy of *HTML divs*) of a given TVL construct had to be built. If a feature on the path between the root feature and that construct is instanciable, a selector is created in its *div*. This allows to select the correct instance of the construct's ancestor. In the future, we intend to offer several selection mechanisms for feature instances in the FCSS model.

In this third version of the generator, we generate as many instances of the corresponding block of *HTML* code as the lower bound. If that bound is equal

to zero, a single instance is generated. Once the generation is completed, the *Parser* Java class parses the Web page to extract, parametrize, and store those blocks of *HTML* code on the Web server. Finding such blocks is made possible by adding recognizable tags at generation time. The latter are removed by the *Parser* Java class as well as block of *HTML* code representing instanciable features with a lower bound equal to zero. We do not give details about the Java class here. Interested readers can refer to Appendix B.1 for the source code.

6.2 PRESENTER

In this section, we discuss the behaviour of the configuration GUI implemented using JavaScript. Contrarily to the *HTML*, this one is not generated. Instead, it has been made as generic as possible. For this purpose, we use specific and recognizable classes for *HTML* representing TVL constructs. Those classes are added by the generator presented in the previous section but will be explained here in order to avoid confusion. The JavaScript is thus independent from the FM and has to be coded once for every variability model imaginable.

This section is sub-divided into three subsections corresponding to the different steps of the Web page's life cycle. Those are the initialisation, followed by the configuration of the product itself, and finalisation tasks.

6.2.1 Initialisation

The first step in the life cycle of the Web configurator is its initialisation. Our first intent was to get the values of the *HTML* widgets as soon as the page is loaded in order to set the corresponding values in the TVL solver. Our purpose was to synchronize the GUI and the solver. A first prototype was built. Unfortunately TVL does not provide the mechanisms to handle such *default* values. And the *default* property for TVL groups in the FCSS model cannot substitute them. Indeed, we quickly realised that those default values could lead to invalid configurations. For example, default sub-features of two different groups might be mutually exclusive. In such cases, initialisation of the configuration GUI was made impossible. Default TVL values being out of the scope of this thesis, we do not initialise the solver as the *HTML* page is loaded. Instead, checks are delegated to the solver as soon as the user inputs a choice. This point should be addressed in the future.

6.2.2 Configuration

It is the user's responsibility to set her choices using *HTML* widgets in the Web page. Our first task in the configuration step is thus to detect such inputs. For this purpose, we could either use *HTML event attributes*¹ (e.g., *oninput*, *onchange*, etc.) or JQuery event handler functions², JQuery being a feature-rich JavaScript library [JQuery, 2014]. In order to reduce the impact on the *Acceleo* generator, we chose the second solution which does not require to modify the *HTML* file as soon as the signature of a JavaScript function is changed. In our presenter, we thus have listeners for each type of widget representing a TVL construct. For example, Listing 6.4 contains the listener of list boxes representing feature groups. There, as soon as a change event is detected in the document (i.e., the *HTML* page) on list box widgets (*select*) which have the *group* class, the *handleSelectGroup* function is called. Its second parameter is a Boolean value indicating the origin of the change (*false* for the user and *true* for the solver) given that the handling will slightly differ. In the *Acceleo* generator, a class name corresponding to the TVL construct it represents is added to each *HTML* widget. Available values are *group*, *opt*, *independent*, *clones*, *attribute*, *number*, *boolean*, *enumeration*, and *string*. All listeners are implemented similarly to the one presented in Listing 6.4 and call an *handleXXX* function.

Listing 6.4: Event listener for TVL groups represented as list boxes

```
1 $(document).on("change", "select.group", function(){  
2     handleSelectGroup($(this),false);  
3 });
```

Change handling functions depend on two parameters: the impacted widget and the origin of the change (user *vs.* solver). Here, we do not give detailed information for each of those functions. Instead we explain the general principle which is common to most widgets, notwithstanding the TVL construct they represent. The first task is to determine the change in the widget. While it is simple for some *HTML* constructs like check boxes, others require to keep track of previously selected values (e.g., list boxes or radio groups). For this second case, we made use of the data storage facility offered by *JQuery*. We store the previously selected value(s) in a key/value construct where the key is the widget and the value, the previously selected value(s). From this information, we extract two variables: the removed values and the added ones. The data pair is updated at the end of the handling function in order to reflect more accurately the current state of the Web page.

¹ See http://www.w3schools.com/tags/ref_eventattributes.asp

² See <http://api.jquery.com/category/events/>

From here a distinction has to be made between solver and user choices. In the first case one can be sure that changes are valid ones and can be automatically propagated. For features which are not selected anymore (resp. newly selected), their contents have to be marked as unavailable (resp. available). In our approach, the availability of an *HTML* element is based on the *unavailable/available* classes. The tricky part is that using TVDL the contents of a feature can be split across different views (i.e., parts of the *HTML* file), duplicated, etc. We do not provide details about the implementation of our search algorithm as it is really technical. Basically, we rely on *HTML* IDs and classes to determine if a construct represents the contents of a feature. For example, the naming convention for the ID of a *div* representing the contents of a feature *X* ends with *X_content*. Those contents can be scattered through the Web page. Given that a TVL construct can be depicted in several views, all its instances have to be updated in the *HTML* file. For this purpose we also use the ID-classes mechanism. If the change handled by the function has been made by the user, we cannot be sure that it is valid. In that case we have to contact the Web server which hosts the TVL solver. Its answers can be classified into two categories with different impacts. Either it is an invalid change and the JavaScript warns the user before resetting the Web page to its previous (valid) state. Or the change is valid and the solver returns a list of impacted features/attributes together with their values. The Web page is updated according to that list (see next paragraph) and the handling function can perform the same tasks as the one described earlier in this paragraph (availability of the contents, consistency of *HTML* constructs representing the given TVL element, etc.). The behaviour of the `handleInputCloneNumber` function is different. There, if the number of clones is increased, the Web service containing parametrizable clones instances is called and returns the *HTML* code to inject in the *HTML* page. Oppositely, the *HTML* code for clones instances is removed if their number is decreased.

Communication with the server is done through Ajax POST requests with data in JSON format. The returned result is an array containing propagated values if the requested change is a valid one, *null* otherwise. We built a wrapping function on top of it. It takes all removed values and new ones as parameters and iteratively unsets and sets the corresponding TVL constructs in the solver on the Web server. At the end, the result of the propagation is returned and forwarded to functions which update the configuration GUI accordingly. Such a wrapping function avoids duplicating code in the different handlers.

Propagating the solver results is done by iterating on its result list. For each TVL construct whose value has been propagated, we search for an *HTML* widget representing it. Some constructs are not represented in the *HTML* configurator as they do not require to be configured (e.g., empty mandatory features

contained in an *and*-decomposition) and do not have to be visually updated. In other cases, the TVL construct is represented by at least one *HTML* widget. We search for one of those widgets, update its value, and call the corresponding listener (as described in the first paragraph of this *Configuration* section) with the Boolean parameter set to *true*. If other widgets represent the TVL construct, their value will be updated by the handler function as mentioned in the third paragraph of this section. The *unassign* value is a special one for a TVL construct. Basically, it means that it was previously set by the solver and is no longer propagated due to the last choice. In that case, we chose to manually reassign the value (i.e., like if it was a user choice) previously propagated in the GUI. Our intent is to avoid GUIs where widget values appear and disappear all the time. In this way, we reduce the number of visual changes for the user. At the implementation level, if the *unassign* value is found, the value of the corresponding *HTML* widget is left unchanged and added to a queue of TVL constructs which have to be reassigned in the solver. They are sent to the Web server as soon as all elements of the currently handled propagation list have been rendered in the GUI. We opted for such a strategy in order to avoid intertwining (and sometimes conflicting) propagation lists.

Figure 6.3 provides a simplified view of that JavaScript workflow.

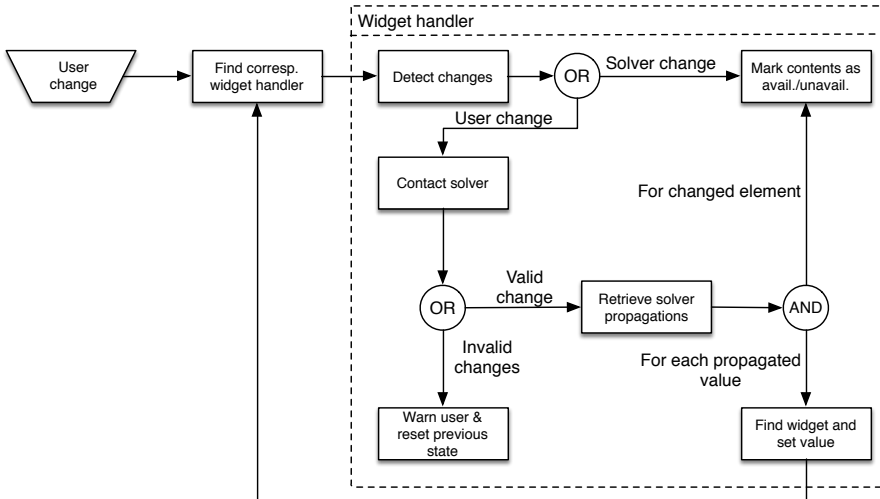


Figure 6.3.: Simplified workflow of the JavaScript presenter

6.2.3 Finalisation

Finalisation of the configuration process really depends on the user's requirements. In some cases she will wish to have a button to save her configuration locally or, for on-line stores, to send it to the seller. In other cases, the configuration will be sent to an e-mail address. Due to this large number of use cases and their specificity, we did not implement any JavaScript finalisation function. As a consequence, the configuration is lost as soon as the Web page is closed. Instead, we prefer to build user-specific finalisation functions on demand.

Generally speaking, the overall behaviour of such functions is the same. On user request (button clicked, etc.), a finalisation function is called. The latter will take a picture of the current configuration state, get it in a given format (human readable, XML, etc.) and display or send it to the required location. The picture of the current state can be extracted either by going through the *HTML*, finding widgets representing TVL constructs, and retrieving their value, or by contacting the solver to get its current state and formatting the answer in the required format.

6.3 SUMMARY

In this chapter, we gave technical details about our prototype solution. As a summary, our architecture is based on the model-view-presenter pattern. The view is generated using a model-to-text approach with *Acceleo*. The source code can be decomposed into two parts: *queries* and *templates*. The queries allow to navigate through the different models (i.e., TVL, TVDL and FCSS) and are used by the templates. The latter iterate through views of the TVDL model to generate *HTML* source code. The generated code contains the widgets corresponding to all views except for feature instances which are dynamically added and removed. The presenter has been implemented in JavaScript and is the central point between the GUI and the solver. It is generic and can be re-used for any configurator. Linking the presenter and the *HTML* Web page is done through feature qualified names and *HTML* classes.

Part III

EVALUATION & CONCLUSIONS

EVALUATION

We performed two evaluations. The first is the evaluation of the TVL language based on small controlled experiments with five professionals from four different companies (Section 7.1). In the second evaluation, we apply our languages (TVL, TVDL and FCSS) and tools to the prototyping of a configurator for an industrial partner (Section 7.2). An extended version of the first evaluation is available in [Hubaux et al., 2010a].

7.1 EVALUATION OF TVL

7.1.1 *Evaluation Criteria*

The criteria that we used to measure the quality of TVL are inspired and adapted from [Holtz and Rasdorf, 1988, Pratt, 1984]. Originally, these criteria were established to evaluate the quality of programming languages. We have selected programming language criteria because prevailing frameworks meant to evaluate modelling languages like [Green, 1989, Moody, 2009] are essentially dedicated to graphical rather than textual notations. Additionally, TVL can be seen as a declarative constraint language whose constructs are tailored to variability modelling. Finally, TVL is integrated in development environments like Eclipse or advanced text editors like Emacs or vim. TVL is thus likely to be assimilated to a programming language by developers. We outline the quality criteria relevant to our study below.

Clarity of notation

The meaning of constructs should be unambiguous and easy to read for non-experts.

Simplicity of notation

The number of different concepts should be minimum. The rules for their combinations should be as simple and regular as possible.

Conciseness of notation

The constructs should not be unnecessarily verbose.

Modularisation	The language should support the decomposition of the model into several modules.
Expressiveness	The concepts covered by the language should be sufficient to express the problems it addresses. Proper syntactic sugar should also be provided to avoid convoluted expressions.
Ease and cost of model portability	The language should be platform independent.
Ease and cost of model creation	The elaboration of a solution should not be overly human resource-expensive.
Ease and cost of model translation	The language should be reasonably easy to translate into other languages.
Learning experience	The learning curve of the language should be reasonable.

7.1.2 Cases

The evaluation of TVL has been carried out with five participants coming from four distinct companies. This subsection describes the areas of expertise of these companies and the motivations of the participants to evaluate TVL. Table 7.1 summarises the profiles of the five participants involved in the evaluation along with the company they work for and a short description of the project chosen to evaluate TVL. For each participant, we collected his position, years of experience in software engineering, his fields of expertise, the modelling and programming languages he used for the last 5 years, his experience with SPLE and FMs, and the number of years he actively worked on the selected project. Note that for the experience with languages, SPLE and FMs, the results are punctuated with the frequency of use, i.e., *intensive/regular/occasional/evaluation*.

Hereunder we give a brief description of the four different cases and the participants:

1. PloneMeeting

PloneGov [PloneGov, 2010] is an international Open Source (OS) initiative coordinating the development of secure, collaborative and evolutive eGovernment Web applications. PloneGov gathers hundreds of public organizations worldwide. This context yields a significant diversity in, for example, languages, regulations, third-party systems or cultures, which, in turn, is the source of ubiquitous variability in the applica-

Table 7.1.: Profiles of the five participants

Criteria	PhoneMeeting	PRISMAprepare	CPU calculation	OSGeneric
Company	<i>GezTern</i>	<i>OSL Namur S.A.</i>	<i>NXP Semiconductors</i>	<i>Virage Logic</i>
#Employees	1	70	28 000 (worldwide)	700 (worldwide)
Location	Belgium	Belgium	The Netherlands	The Netherlands
Type of software	Open source	Proprietary	Proprietary	Proprietary
Project kickoff date	January 2007	May 2008	June 2009	2004
Project maturity level	Production	Production	Development	Production
Model version	1.7 build 564	4.2.1	July 6, 2009	September 30, 2009
Position	Freelance	Product Line Manager	Senior Scientist	Senior Software Architect
Years of experience in SE	12+ years	31+ years	20+ years	20+ years
Fields of expertise	WA, CMS	PM	PM, SPI, SR	ES, RT
Modelling languages	UML (regular)	None	UML (occasional), DSCT (evaluation)	UML (regular)
Programming languages	Python (intensive), Java-script (regular)	C++ (occasional)	C (occasional), Prolog (regular), Java (evaluation)	C (occasional), C++ (occasional), Visual Basic (occasional), Python (occasional)
Experience with SPLE	2 years (evaluation)	1 year (occasional)	3 years (intensive)	4 years (intensive)
Experience with FDs	2 years (evaluation)	1 year (occasional)	3 years (intensive)	4 years (occasional)
Project participation	3 years	2 years	1 year	6 years

Legend

CMS - Content Management System, **ES** - Embedded System, **McS** - Multi-core System, **PM** - Project Management, **RPC** - Remote Procedure Call, **RT** - Real Time, **SE** - Software Engineering, **SPI** - Software Process Improvement, **SR** - Software Reliability, **WA** - Web Applications

tions. We focus here on PloneMeeting, PloneGov's meeting management project written in Python.

PloneMeeting was re-engineered with *appy.gen*. In a nutshell, *appy.gen* enables the automated generation of full-blown Plone applications. A major challenge was to extend *appy.gen* to explicitly capture variation points and provide systematic variability management. We collaborated with the developers to design a FM representing the configuration options of PloneMeeting.

We interacted with several developers during the collaboration, but only the main developer provided the quantitative and qualitative feedback as part of the evaluation. The initial motivation of the developer to engage in the evaluation was to assess the opportunity of using FMs for code generation. He has not used FMs to that end so far because, in his words, graphical editing functionalities (typically point-and-click) offered by feature modelling tools are cumbersome and counter-productive. The textual representation of FMs is therefore more in line with his development practice than its graphical counterpart.

2. PRISMAprepare

Océ Software Laboratories S.A. (OSL) [Océ Software Laboratories, 2010], is specialized in document management for professional printers. One of their main product lines is *Océ PRISMA*, a family of products covering job creation, submission and delivery via preparation and accounting. A job can be basically defined as a document to print. Our collaboration focuses on one sub-line called *PRISMAprepare*, an all-in-one tool that guides document preparation including the customisation of printing options and the preview of documents.

In *PRISMAprepare*, mismatches between the preview and the actual output could occur, and, in rare cases, documents may not even be printable on the selected printer. The reason was that incompatibilities between the document's specificities and the selected printer were not always completely detected. For example, a prepared document could require to staple all sheets together while the target printer could not staple more than 20 pages together, or does not support stapling at all. The root cause was that only some constraints imposed by the printers were implemented in the source code, mostly for time and complexity reasons. Consequently, OSL decided to rely on FMs to formally represent the variability of their tool.

Feedback was provided by the *product line manager* of *PRISMAprepare*. OSL was evaluating different modelling alternatives to express the variability of its new SPL and generate the configuration GUI. The problem

of most tools is that they impose their own configuration interface—commonly a tree exposed in a file explorer style. Their motivation for evaluating TVL was that it is independent from any front-end, which *a priori* makes it easier to generate domain specific interfaces.

3. CPU calculation

NXP Semiconductors [NXP Semiconductors, 2010] is an international provider of Integrated Circuit (IC). ICs are used in a wide range of applications like automotive, infotainment or navigation systems. ICs typically embed several components like CPU, memory, input and output ports. Due to constraints imposed by hardware, a valid configuration is determined by the combination of hardware and the features that can run on it.

In this paper, we focus on the FM that models the variability of the video processing part of their product line and the impact it has on the CPU load and other physical constraints. The FM is meant to support the customer during the selection of features while ensuring that no hardware constraint can be violated, e.g., excessive clock speed required by the features. Besides ensuring the verification of constraints, the FM also allows the user to strike an optimal price/performance balance, where the price/performance ratio is computed from attributes attached to features.

The evaluation was performed by the developer that created the FM. The variability used in this example was previously modelled using pure::variants [Beuche, 2008]. Regarding calculations, the Prolog language was used for defining the constraints. The major problem was that the time needed to implement the calculation over attributes was excessive compared to the time needed to design the whole FM. The interest in considering an alternative language lied in the expressive power that textual languages could have over existing graphical solutions in capturing the constraints based on calculation of feature attributes.

4. OSGeneric

Virage Logic [Virage Logic, 2010] is a supplier of configurable hardware and software to a broad variety of customer such as the Dolby Laboratories, Microsoft or AMD. The hardware and software intellectual property (IP) delivered by Virage Logic offers a high degree of variability that allows its customers to create a specific variant for the manufacturing of highly tailored systems on chip (SoC). OSGeneric (Operating System Generic) is a product line of operating systems used on SoCs. The produced operating systems can include both proprietary and free

software. Every SoC can embed a great variety of hardware and contain several processors from different manufacturers.

The evaluation was performed by two participants: the lead software architect of OSGeneric and the software development manager. Their product line was previously modelled with pure::variants. The participants were considering other techniques in modelling the variability. Their motivation for evaluating TVL lied in using a language that (1) is more suited for engineers with a C/C++ background, (2) has a lower learning curve than pure::variants and (3) makes use of standard editors.

7.1.3 Research Protocol

In this experiment, TVL was evaluated through interviews with the five participants of the four companies. Interviews were conducted independently from each other, except for Virage Logic where the two participants were interviewed together. Two researchers were in charge of the interviews, the synthesis of the results and their analysis. For each interview, we followed the protocol presented in Figure 7.1.

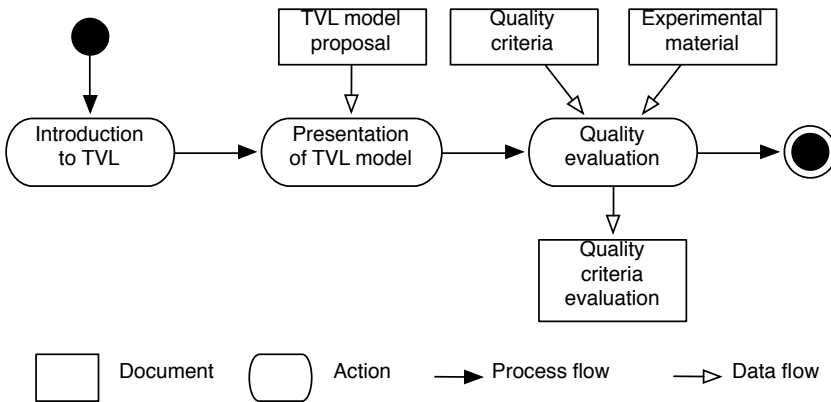


Figure 7.1.: Interview protocol

The protocol starts with a short introduction to TVL (*circa* 20 minutes) that aims at giving the participants an overview of the language. At this stage, the participants are not exposed to details of the language. The goal of the second step is to provide the participants with a real TVL model. To keep the effort of the participants moderate, the appointed researchers designed, for each company and prior to the interviews, TVL models that respectively cor-

respond to the configuration menus of PloneMeeting and PRIMSAprepare, and the FMs of the CPU calculation and OSGeneric. The presentation of the TVL model was limited to 30 minutes to keep the participants focused on the understanding of the model and avoid untimely discussions about the quality of the language. During that step, the participants are exposed to more details of the language and discover how their product line can be modelled using TVL. Alternative design decisions are also discussed to demonstrate the expressiveness of TVL.

During the third step, the participants fill out the evaluation form presented in Table 7.2. The evaluation scale proposed to the participants is: **+** the participant is strongly satisfied; **+** the participant is rather satisfied; **○** the participant is neither satisfied nor dissatisfied; **-** the participant is rather dissatisfied; **-** the participant is completely dissatisfied; *N/A* the participant is not able to evaluate the criterion.

The results of the evaluation are then discussed during the fourth step. The qualitative information collected during this last phase is obtained by asking, for each criteria, the rationale that lead the participant to give his mark. On average, these two last steps lasted two hours in total.

7.1.4 Analysis of TVL

Table 7.2 synthesises the evaluation of TVL performed by the participants of GeezTeem, OSL, NXP and Virage Logic. Note that we kept the evaluations of the two Virage Logic participants separate, has indicated by the two columns under OSGeneric.

To facilitate the explanation, we group the criterion into five categories: *notation*, *modularisation*, *expressiveness*, *ease* and *cost*, and *learning experience*. Note that the collaborations with OSL, NXP and VirageLogic are protected by non-disclosure agreements. Therefore, specific details of the models are not disclosed.

NOTATION [c1-c3]. The participants unanimously appreciated the notation and the advantages of text in facilitating editing (creating, modifying and copy/pasting model elements). The NXP and VirageLogic participants liked the compactness of attributes and constraints and the fact that attributes were explicitly part of the language rather than an add-on to a graphical notation.

The GeezTeem participant appreciated the ability of the language to express constraints very concisely. He reported that *appy.gen*, his Web site generator, offered two major ways of specifying constraints. First, guards could be used to make the value of an attribute depend on the value of another

Table 7.2.: Results of the evaluation of TVL

Criterion		PloneMeeting	PRISMA prepare	CPU calculation	OSGeneric	
C1	Clarity of notation	+	+	○	+	+
C2	Simplicity of notation	+	+	+	+	+
C3	Conciseness of notation	+	+	+	+	+
C4	Modularisation	○	+	+	+	+
C5	Expressiveness	-	○	+	+	+
C6	Ease and cost of model portability	+	+	+	+	+
C7	Ease and cost of model creation	+	+	+	○	○
C8	Ease and cost of model translation	+	+	+	○	+
C9	Learning experience	+	+	+	+	+

attribute. Secondly, Python methods could be used to express arbitrary constraints. These mechanisms could rapidly lead to convoluted constraints that are hard to maintain and understand. Additionally, developers struggled to maintain these constraints across Web pages. The participant reported that at least 90% of the constraints (both within and across pages) implemented in classical *appy.gen* applications could be more efficiently expressed in TVL.

The OSL participant was particularly satisfied to see that TVL is not based on XML. He reported that their previous attempts to create XML-based languages were not very satisfactory because of the difficulty to write, read and maintain them. He also reported that the model represented in the language is much more compact than anything he could have produced with existing graphical representations.

The NXP participant was concerned about the scalability of the nested structure, i.e., the tree-shaped model, offered by TVL. He also reported that people used to graphical notations who already knew FMs might prefer classical decomposition operators (*and*, *or*, *xor*) rather than their TVL counterparts (*allOf*, *someOf*, *oneOf*). Finally, the participants from NXP and Virage Logic were confused by the fact that the *->* symbol can always replace *requires* but not the other way around. In their opinion, a language should not offer more than one means to express the same thing.

One of the Virage Logic participants reported that attributes might be hard to discern in large models. He suggested to declare them in an Interface

Description Language (IDL) style by prefixing the attribute declaration with the `attribute` keyword.

MODULARISATION [c4]. The ability to define a feature at one place and extend it further in the code was seen as an undeniable advantage as it allows to distribute the FM among developers. The Virage Logic participants both discussed the difference between the TVL `include` mechanism and an *import* mechanism that would allow to specify exactly what parts of an external TVL model can be imported but also what parts of a model can be exported. In their opinion, it would improve FM modularisation and module reuse since developers are already used to import mechanisms.

Apart from the `include` mechanism, TVL does not support *model* specialisation and abstraction (as opposed to constructs, e.g., class and specialisation). In contrast, the developer of *appy.gen* considered them as fundamental. Their absence is one of the reasons that lead them to drop UML tools. Along the same lines, the OSL participant argued that the `include` should be augmented to allow *macro* definitions. By *macro*, the participant meant parametrized models similar to parametrized types, e.g., Java generics. A typical use case of that mechanism would be to handle common variability modelling patterns.

EXPRESSIVENESS [c5]. The GeezTeem participant expressed that TVL is sufficiently expressive to model variability in most cases. However, he identified several constructs missed by TVL that would be needed to model Plone-Meeting. First, TVL does not offer *validators*. In his terms, a validator is a general-purpose constraint mechanism that can constrain the formatting of a field. For instance, validators are used to specify the elements that populate a select list, to check that an email address is properly formatted or that a string is not typed in where the system expects an integer. Secondly, he made intensive use of the specialisation and abstraction mechanisms available in Python, which have no TVL equivalents. These mechanisms are typically used to refine already existing variation points (e.g., add an attribute to a meeting item) or to specify abstract variation points that have to be instantiated and extended when generating the configuration menu. Thirdly, multiplicities are used to specify the number of instances, i.e., clones, of a given element. Cloning was a fundamental aspect of *appy.gen* as many elements can be cloned and configured differently in Plone applications. Those have been added in TVL₂. Besides offering more attributes types, *appy.gen* also allows developers to add parameters to attributes, e.g., to specify whether a field can be edited or requires specific read/write permissions. Type parameters are mandatory in *appy.gen* to support complete code generation. Finally, in order to be able to display Web pages in different languages, *i18n* labels

are attached to elements. *i18n* stands for internationalisation and is part of Plone's built-in translation management service. Translations are stored in key/value pairs. A key is a label in the code identifying a translatable string; the value is its translation. For instance, the `meeting_item_i18n` element will be mapped to *Meeting Item* (English) and *Point de discussion* (French). In most cases, several labels are attached to an element (e.g., a human-readable name and a description). That information is now included in FCSS models.

The OSL participant also pointed out some missing constructs in TVL. First, default values which are useful in their projects for things like page orientation or paper dimensions. Secondly, feature cloning was missing. In PRISMA-prepare, a document is normally composed of multiple *sheets*, where sheets can be configured differently and independently from one another. Thirdly, optionality of attributes should be available. For instance, in TVL, the *binding margin* of a page was specified as an attribute determining its size. If the document does not have to be bound, the *binding margin* attribute should not be available for selection. As for feature instances, attribute ones are now supported in TVL₂.

The NXP and VirageLogic participants also recognized that feature cloning and default features were missing in the language. Additionally, they missed the specification of error, warning and information messages directly within the TVL model. These messages are not simple comments attached to features but rather have to be considered as guidance provided to the user that is based on the current state of the configuration. For instance, in the NXP case, if the selected video codec consumes most of the CPU resources, the configurator should issue a warning advising the user to select another CPU or select a codec that is less resource-demanding. Since they are a needed input for a configurator, they argued that a corresponding construct should exist in TVL.

EASE AND COST [c6-c8]. The OSL participant reported that improvements in terms of *model creation* were, in his words, very impressive compared to the graphical notation that was used initially [Czarnecki et al., 2005]. And since TVL is formally defined, he did not foresee major obstacles to its translation into other formalisms.

The NXP and VirageLogic participants reported that, no matter how good the language is, the process of model creation is intrinsically very complex. This means that the cost of model creation is always high for real models. Nevertheless, they observed that the mechanisms offered by TVL facilitate the transition from variability elicitation to a formal specification, hence the neutral score.

LEARNING EXPERIENCE [c9]. All the participants agreed that the learning curve for software engineers with a good knowledge of programming languages was rather gentle. Software engineers who use only graphical models might need a little more time to feel comfortable with the syntax. In fact, the NXP and VirageLogic participants believed that people in their teams well versed in programming languages would give a \oplus whereas those used to modelling languages would give a \bigcirc , hence their average $+$ score.

More detailed findings on the questionnaire can be found in [Hubaux et al., 2010a].

7.1.5 Threats to Validity

The evaluation was performed with four product lines and five participants, providing a diversity of domains and profiles. Yet, their diversity did not have a significant influence on the results since we observed a substantial overlap between their comments. Therefore, we believe that the results are valid for wide a range of organizations and products [Yin, 2002].

The TVL models were prepared in advance by the two researchers and later checked by the participants. Consequently, the expertise of the researchers might have influenced the models and the evaluation of the participants. In order to assess this potential bias more precisely, we will have to compare models designed by participants to models designed by the two researchers. However, TVL is arguably simpler than most programming languages and the modelling task was felt to be rather straightforward. As a consequence, we do not expect this to be a problem for our evaluation. Furthermore, when the participants questioned the design decisions, alternative solutions were discussed based on the constructs available in the language—even those not disclosed during the presentation.

The limited hands-on experience with TVL might have negatively influenced the evaluation of the expressiveness, notation and modularisation of the language, and positively impacted the evaluation of the ease and cost and learning experience. That situation resembles the setting of an out-of-box experience [Sangwan and Hian, 2004]. This gives valuable insight as to how software engineers perceive TVL after a one-hour training and how fast they can reach a good understanding of the fundamental concepts.

A more specific problem was the unavailability of proper documentation and the limited access granted to the codebase in the case of OSL, NXP and Virage Logic. This made the modelling of those cases more difficult.

In the case of OSL, the development team is still in the SPL adoption phase. This could be a threat as the participant has only been exposed to FMs for reviewing. Therefore, he might have focused on comparing the textual and graphical approaches rather than evaluating the language itself. Along the

same lines, the PloneMeeting participant was already reluctant to use graphical FMs and might have evaluated the textual approach rather than TVL itself. In any case, we believe that the feedback received was more shaped by the expectations and requirements of the participants than by the preference for a textual representation over a graphical one.

7.2 EVALUATION OF LANGUAGES AND TOOLS

Rexel [Rexel, 2014] is a French-based worldwide distributor of products and services in the energy area, mainly focused on the electrical supplies. Its customers are professionals in the residential, commercial as well as industrial sectors. On its Web site, the company states that it operates in 37 countries across most continents. Their role is to supply their customers with necessary resources to build electrical systems. For example, they sell wires, plugs, etc. but also have an expertise in various domains such as heating, ventilation and cooling systems.

In a competitive sector like that of electrical supplies, Rexel differentiates itself by offering customer services with an added value for the customer. By this, we mean that they offer, for example, consultancy for large projects or software tools for electricians. The required service depends on a lot of parameters such as the size of the project, its complexity, available resources, etc. The goal of Rexel is to facilitate its customers' work by avoiding waste of time, order errors, etc.

Our collaboration with Rexel focuses on software tools they offer to their customers in order to place orders, especially for electrical panels. Such tools allow customers to describe the domain (e.g., the building, its dimensions, its specificities), the requirements (e.g., number of plugs or lights) and the constraints (e.g., local regulations). Based on that information, the feasibility of the project is assessed, errors reported, and optimisations proposed. Rexel wishes to increase the evolvability of its tools. Indeed, a lot of variability exists in the domain. The main example is the regulations which vary over time as well as depending on the location. In current versions of the tools, managing this variability requires a lot of work and is error-prone since one has to modify constraints across the whole source code. Problems faced by Rexel can thus be considered as similar to the ones of OSL mentioned in the previous section. We evaluated our approach on the electrical panels case together with people at Rexel.

7.2.1 *Models*

In our approach, the first task is to represent the variability of the configurable product. In this case, we had several sources to undertake that task.

First, we used the existing tool developed by Rexel. They helped us to extract the variation points, i.e., features and attributes in the TVL model. Scenarios were also defined in order to extract constraints between those variation points. However, such an approach does not ensure the elicitation of all cross-tree constraints. The result depends on the quality and accuracy of the scenarios. Ideally, this task should be automated in order to automatically explore all use cases. Reverse engineering existing tools is not covered here but is discussed in another PhD thesis [Abbasi, 2014] and introduced in [Boucher et al., 2012a]. Regulation documents supplied by Rexel also provided a reliable source of information. There, all legal and regulatory constraints could be found and added to the TVL model. For technical ones, we got input from skilled persons at Rexel who have developed a certain knowledge of the field, either due to their contribution in the development of the existing tool, or their technical background in the electrical field. Note that, during the workshops at Rexel, recommendation was mentioned but is not introduced here as it is not part of our approach.

The TVL model built with those three sources is visible in Listing 7.1. There, constraints have been removed in order to keep the code as compact as possible. Most feature and attribute names are in French but this should not be an obstacle to the validation of the approach. The reason is that Rexel wishes, as a first step, to evaluate the approach on a well mastered and controlled local product. The FM has 42 features and a maximum depth of 7 levels. The number of attributes is 56, broken down as follows: 3 Boolean, 10 strings, 15 enumerations and 28 integers. All enumerations have been declared using user-defined attribute types at the beginning of the TVL model (lines 1-10). The root feature is decomposed into four sub-features. Three of them, namely *Projet* (lines 14-17), *Client* (lines 18-24) and *Techniques* (lines 25-27) contain general context information about the project while *Abstract* (lines 28-160) contains all relevant information for the configuration itself. That feature has a single sub-feature, *Logement* (lines 30-158) which describes an accommodation. The *Abstract* feature had to be introduced for the TVDL model. Rexel wished to separately display the information of the project from the one related to the electrical-related concepts but it was initially not possible with our view definition language. An accommodation is composed of several *Pieces* (i.e., rooms at lines 32-61) and *Circuits* (lines 62-107) as well as a *Tableau* (i.e., electrical panel at lines 108-156). Each room has a type (line 33), a surface (line 34) and contains several *Equipements* (i.e., equipments from line 36 to line 59). The contents of *Piece*'s siblings contain technical information related to the electrical field which will not be addressed here for simplicity.

Listing 7.1: TVL model (excl. constraints) for the Rexel case

```

1  enum pieces in {Sejour, Cuisine, Chambre, Couloir, WC, Extérieur,
2      SalleDeBain};
3  enum marques in {Legrand, Hager, Schneider};
4  enum interrupteursDifferentiels in {AC, A, ASi};
5  enum disjoncteurs in {AC, A, ASi};
6  enum disjoncteursDifferentiels in {AC, ASi};
7  enum fusibles in {A, ASi};
8  enum poles in {BiPolaire, QuadriPolaire};
9  enum technologies in {Automatique, AVis};
10 enum sectionType in {F1_5, F2_5, F6, F10, F16};
11
12 enum departements in {Ain, Aisne, Allier, AlpesDeHauteProvence,
13     HautesAlpes, AlpesMaritimes, Ardeche, Ardennes, Ariege, Aube, Aude,
14     Aveyron, BouchesDuRhône, Calvados, Cantal, Charente,
15     CharenteMaritime, Cher, Correze, CorseDuSud, HauteCorse, CoteDOr,
16     CotesDArmor, Creuse, Dordogne, Doubs, Drome, Eure, EureEtLoir,
17     Finistere, Gard, HauteGaronne, Gers, Gironde, Herault,
18     IleEtVilaine, Indre, IndreEtLoire, Isere, Jura, Landes, LoirEtCher,
19     Loire, HauteLoire, LoireAtlantique, Loiret, Lot, LotEtGaronne,
20     Lozere, MaineEtLoire, Manche, Marne, HauteMarne, Mayenne,
21     MeurtheEtMoselle, Meuse, Morbihan, Moselle, Nièvre, Nord, Oise,
22     Orne, PasDeCalais, PuyDeDome, PyreneesAtlantiques, HautesPyrenees,
23     PyreneesOrientales, BasRhin, HautRhin, Rhone, HauteSaone,
24     SaoneEtLoire, Sarthe, Savoie, HauteSavoie, Paris, SeineMaritime,
25     SeineEtMarne, Yvelines, DeuxSevres, Somme, Tarn, TarnEtGaronne,
26     Var, Vaucluse, Vendee, Vienne, HauteVienne, Vosges, Yonne,
27     TerritoireDeBelfort, Essonne, HautsDeSeine, SeineSaintDenis,
28     ValDeMarne, ValDOise, Guadeloupe, Martinique, Guyane, LaReunion,
29     Mayotte};
30
31 root ProjetTuring {
32     group allOf {
33         Projet {
34             string nomProjet;
35             string dateProjet;
36         },
37         Client {
38             string numeroClient;
39             string nomClient;
40             string adresseClient;
41             string villeClient;
42             departements departementClient;
43         },
44         Techniques {
45             int surfaceLogement;
46         },
47         Abstract {
48             group allof {
49                 Logement {
50                     group allof {
51                         Piece [1..*] {
52                             pieces typePiece;
53                             int surfacePiece;
54                         }
55                     }
56                 }
57             }
58         }
59     }
60 }

```

```

36      Equipement [1..*] {
37          group oneof {
38              PC16A,
39              PC32A,
40              EclairagePlafond ,
41              EclairageApplique ,
42              PlaqueDeCuisson ,
43              LaveLinge ,
44              LaveVaisselle ,
45              SecheLinge ,
46              Congelateur ,
47              Four ,
48              ChauffeEau {
49                  int puissanceChauffeEau ;
50              },
51              VoletRoulant ,
52              VMC,
53              Chauffage {
54                  bool filPilote ;
55                  int puissanceChauffage ;
56              },
57              Autre
58          }
59      }
60  },
61  Circuit [1..*] {
62      string name ;
63      sectionType section ;
64      int intensite ;
65      group someOf {
66          ProtectionCircuit {
67              int largeurProtectionCircuit in [0..24] ;
68              marques marqueProtectionCircuit ;
69              string gammeProtectionCircuit ;
70              group oneof {
71                  Disjoncteur {
72                      int calibreDisjoncteur in {2, 6, 10, 16, 20, 25, 32} ;
73                      disjoncteurs typeDisjoncteur ;
74                  },
75                  DisjoncteurDifferentiel {
76                      int calibreDisjoncteurDifferentiel in {10, 16, 20, 25, 32} ;
77                      disjoncteursDifferentiels typeDisjoncteurDifferentiel ;
78                  },
79                  Fusible {
80                      int calibreFusible in {10, 16, 20, 25} ;
81                      fusibles typeFusible ;
82                  }
83              }
84          },
85      },
86      opt Commande {
87          int nombreContactsCommande in {1, 2} ;
88          marques marqueCommande ;

```



```

142     }
143   }
144 }
145 },
146 Rehausse {
147   int hauteurRehausse;
148   int largeurRehausse;
149 },
150 Porte {
151   int hauteurPorte;
152   int largeurPorte;
153   bool porteOpaque;
154 }
155 }
156 }
157 }
158 }
159 }
160 }
161 }
162 }

```

The views were iteratively defined together with our contact persons in the company. The result TVDL model is visible in Listing 7.2. At line 1, the TVL model previously introduced is imported before defining the five different views. The first one, `WelcomeTab` (line 3) contains all information related to the project currently configured. It covers data such as the name of the project, the contact information of the client and the surface of the accommodation. In the second view, `LogementTab` (line 5), Roxel wishes to select all information related to rooms, namely their attributes as well as the contained equipments. The third view at line 7 (`CircuitTab`) contains all TVL constructs in the sub-tree with `Circuit` as root, i.e., all information related to electrical circuits. The fourth view, `ElementTab` (line 9), covers `Coffret`'s sub-features, namely `RepartitionVerticale` and `Rangee`. However, the `Circuit` sub-tree is excluded given that it is already covered in the third view. Finally, `TableauTab` (line 11) covers all the attributes of `Tableau`'s sub-features, i.e., it contains all information for `Rehausse` and `Porte`, and technical information about the `Coffret`.

Listing 7.2: TVDL model for the Roxel case

```

1 import "rexel_demo.tvl"
2
3 WelcomeTab {Projet && Client && Techniques}
4
5 LogementTab {Piece:[ attributes ] && Equipement:* }
6
7 CircuitTab { Circuit:* }
8

```



```

9 | ElementTab { RepartitionVerticale && Rangee:*/Circuit}
10
11 | TableauTab { Coffret:[ attributes ] && Rehausse:[ attributes ] && Porte:[
    attributes ]}

```

Finally, the FCSS model contains only labels for features, attributes and views with the exception of a global part where default widgets are defined for *xor*- and *or*-decompositions. Due to space constraints, only the beginning of the FCSS model is visible in Listing 7.3. All other entries are similar to those depicted in the code excerpt. In the global part (from line 4 to line 7), the default widget for *xor*-decompositions (resp. *or*-decompositions) is set to listbox (resp. checkbox). Labels are then defined for views (e.g., line 8), features (e.g., line 26) and attributes (e.g., line 29).

Listing 7.3: FCSS model for the Rexel case

```

1 | import "rexel_demo.tvl"
2 | import "rexel_demo.tvdl"
3
4 | .{
5 |     xorGroup: listbox;
6 |     orGroup: checkbox;
7 | }
8 | $WelcomeTab {
9 |     label: "Accueil";
10 | }
11 | $LogementTab {
12 |     label: "Logement";
13 | }
14 | $CircuitTab {
15 |     label: "Circuits";
16 | }
17 | $ElementTab {
18 |     label: "Elements";
19 | }
20 | $TableauTab {
21 |     label: "Tableau";
22 | }
23 | $ProduitTab {
24 |     label: "Produits";
25 | }
26 | Projet {
27 |     label: "Donnees du projet";
28 | }
29 | #Projet.nomProjet {
30 |     label: "Nom du projet";
31 | }
32 | #Projet.dateProjet {
33 |     label: "Date";
34 | }
35 | Client {

```

```

36         label: "Donnees du client";
37     }
38     #Client.numeroClient {
39         label: "Numero client";
40     }
41     #Client.nomClient {
42         label: "Nom du client";
43     }
44     #Client.adresseClient {
45         label: "Adresse";
46     }
47     #Client.villeClient {
48         label: "Ville";
49     }
50     #Client.departementClient {
51         label: "Departement";
52     }
53     ...

```

7.2.2 Generated Configurator

The *HTML* page generated by our *Acceleo* tool is depicted in Figures 7.2 to 7.6. Each figure represents the same *HTML* file with a different tab selected. For that configurator we just developed a custom *CSS* layout, the content of the page being automatically rendered by our generator.

Figure 7.2 represents the *WelcomeTab* view of the *TVDL* model available in Listing 7.2. Each tab can be decomposed into two parts. The first one covers the contents of the *TVDL* view. It is contained in the green box labelled ① in Figure 7.2. The second part, labelled ②, contains the links to navigate through the different views represented by tabs. There, the labels correspond to the ones defined at lines 8-25 of the *FCSS* model (see Listing 7.3). Clicking on the *Circuits* tab will bring us to Figure 7.4. That information is also used in the head title of ①. The content of that same part of the GUI is itself decomposed into three boxes, A, B and C. Each of those boxes represent a *view part* defined at line 5 of the *TVDL* model (Listing 7.2). A represents the *Projet* part which is composed of the *Projet* feature and its attributes, namely *nomProjet* and *dateProjet* of type string. The label of the feature is defined at line 26. Lines 29-34 contain the same information for attributes. B and C are similar to A except that the first one contains an *enum* attribute (i.e., *departementClient* at line 23 of Listing 7.1) represented by a list box.

The *Logement* tab (Figure 7.3) is decomposed into two parts, *Piece* and *Equipement*, both representing duplicable features. The number inputs at the far right of the part labels represent the number of instances of the corresponding feature, 3 for *Piece* and 4 for *Equipement*. As a reminder, the *HTML* code corresponding to each instance is dynamically added to the Web

The screenshot shows the 'Accueil' tab of the HTML configurator for Rexel. The interface is organized into three main sections, each with a header and a list of attributes:

- Donnees du projet** (Section A):
 - Nom du projet: Thesis example (String attribute)
 - Date: 11/02/2014
- Donnees du client** (Section B):
 - Numero client: 20130921
 - Nom du client: Rexel
 - Adresse: Boulevard du Fort de Vaux
 - Ville: Paris
 - Departement: Paris (Enum attribute)
- Donnees techniques** (Section C):
 - Surface totale du logement: 125

At the bottom, there is a navigation bar with tabs: Accueil, Logement, Circuits, Elements, and Tableau. The 'Accueil' tab is currently selected.

Figure 7.2.: *Accueil* tab of the *HTML* configurator for Rexel

page. In the first part, each line corresponds to an instance of the *Piece* feature which contains two attributes, an *enum* (the type of the room) and an integer (its surface). The first room is a *Chambre* (i.e., bedroom) with a surface of 11 square meters. Displaying all instances at once is not the default behaviour of our generator. For this purpose, customisation has been added in CSS file. *Equipement* being a sub-feature of *Piece*, the user must be able to select the *Piece* in which a given *Equipement* has to be added. The list box above the *Equipement* box is used for that purpose. In Figure 7.3, the first room of type *Chambre* is selected and it contains four equipments. Two other values are available in the clone list box, namely “[WC] *Piece* 2” and “[Exterieur] *Piece* 3”. Selecting one of those options will change the contents of the *Equipement* box accordingly.

The *Circuits* tab (Figure 7.4) contains a single part, the sub-tree with *Circuit* feature as root. That feature being duplicable, an input for the number of instances as well as an instance selector have been made available similarly to what has been done for *Piece* in Figure 7.3. Each circuit contains three attributes. What is new in this screen compared to the previous one is the presence of a feature group. That TVL construct is contained in a different coloured box. An *and*-decomposition is depicted for *Circuit* in Figure 7.4. The *Protection* circuit sub-feature is mandatory and its contents are displayed on the right, in their own box. It in turn contains attributes

The screenshot shows the 'Logement' tab of the HTML configurator for Rexel. The interface is divided into two main sections: 'Piece' and 'Equipement'. The 'Piece' section has a header bar with 'Piece' on the left and 'Number of instances' set to 3 on the right. Below this, there are three rows of attributes: 'Chambre' (Enum attribute) with value 11 (Integer attribute), 'WC' (m2) with value 2, and 'Exterieur' (m2) with value 120. An 'Instance selector' dropdown shows '[Chambre] Piece 1'. The 'Equipement' section has a header bar with 'Equipement' on the left and 'Number of instances' set to 4 on the right. Below this, there are four rows of attributes: 'Prise de courant 16A' (three times) and 'Eclairage Applique'. At the bottom, there is a navigation bar with tabs: 'Accueil', 'Logement' (active), 'Circuits', 'Elements', and 'Tableau'.

Figure 7.3.: *Logement* tab of the *HTML* configurator for Rexel

and a group. In that case, it is a *xor*-decomposition represented by a list box. Changing its value will modify the content of the underneath box. In Figure 7.4, *Calibre* and *Intensite* represent the attributes of the *Disjoncteur* feature. Notice that the *Type* attribute is an enumeration one represented by radio buttons contrarily to previous ones which are represented by list boxes. This is due to the limited number of values (3) for that attribute. Finally, the *Commande* feature is an optional one and has been excluded as depicted by the **X** symbol in its check box. Selecting that feature would display its contents in a similar way than *Protection* circuit.

In the *Elements* tab (Figure 7.5), the three states of check boxes representing features are visible, **X** for rejected, **V** for selected and nothing for undecided. In the last case, the user has not chosen a value for the feature. As a consequence its contents are not displayed like, e.g., *Repartition verticale*. In *Repartition horizontale*, two new types of attributes are displayed. The first one, an integer, is represented by a list box given that it is limited to a few values (see *Largeur*). The second one is a Boolean attribute and is rendered by a check box, just like optional features. The box labelled **A** contains an *or*-decomposition in which *Parafoudre* has been selected and *Interrupteur Differentiel* rejected. Given that in such a feature group at least one sub-feature has to be selected, trying to change the value of *Parafoudre* is forbidden as shown by the error message in box **B**. The error

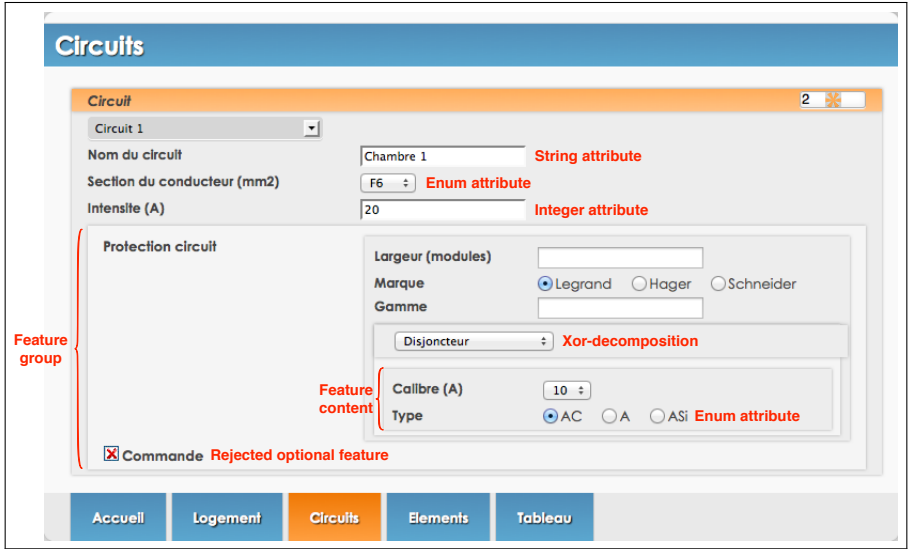


Figure 7.4.: *Circuits* tab of the *HTML* configurator for Rexel

states that Parafoudre cannot be rejected. This message is displayed by the presenter thanks to the result it gets from the solver.

Finally, Figure 7.6 contains all information related to an electric panel. It is decomposed into three components, corresponding to the different parts of the *TableauTab* view (line 11 of Listing 7.2). All its constructs have already been discussed and do not need further explanations.

7.2.3 *Feedback from Rexel*

Globally, Rexel was pleased with the generated *HTML* interface even if they did not use the full power of the FCSS model. We could thus conclude that the default behaviour of our generator matches the expectations of our partners in this industrial case study. The ease and speed with which interfaces could be generated allowed us to easily interact with people without variability modelling background. The TVL and TVDL models changed a lot over time and all required changes were supported by the proposed languages. The developers even challenged us and were not able to find weak points for TVDL.

However, Rexel missed three things in the generated configuration Web page. First, they would like an additional tab summarizing the products to order. This point has already been addressed in Section 6.2.3. Finalisation

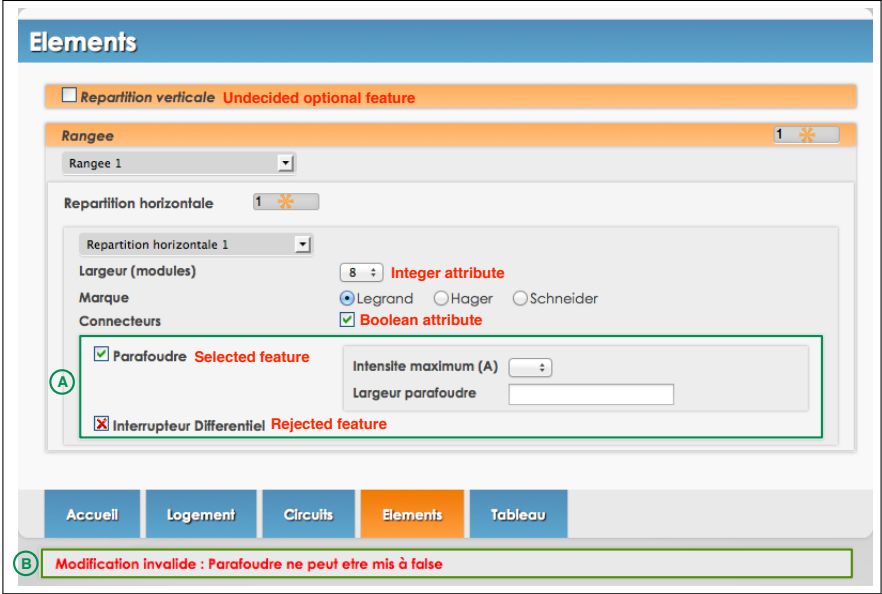


Figure 7.5.: *Elements* tab of the *HTML* configurator for Rexel

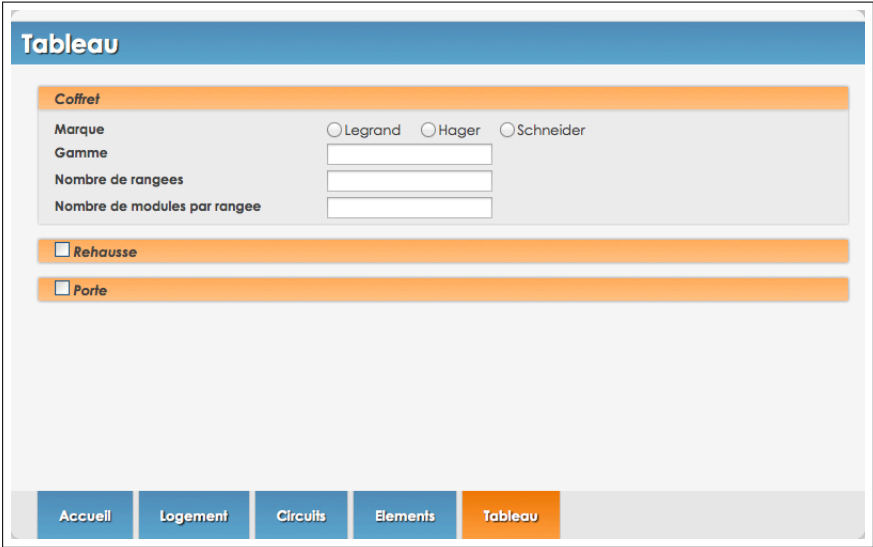


Figure 7.6.: *Tableau* tab of the *HTML* configurator for Rexel

being case-specific, we decided to not handle it in our generator. Instead, it should be developed based on Rexel requirements. A possible implementation would be a Web service which, for a given configuration, returns a list of products to order with all required details.

Rexel also wished a much finer-grained handling for feature instances. In the generated interfaces, 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. But so far, we have decided to write the corresponding code directly in the *HTML* file in order to get validation from Rexel. Figure 7.7 contains the proposed solution, a *delete* button for each feature instance, whether it is a *Piece* or an *Equipement*. Ideally, a button to create a new instance should also be added after the current last one.

The screenshot shows a web application interface for a 'Logement' (Housing) tab. The interface is divided into two main sections: 'Piette' and 'Equipement'. The 'Piette' section contains three rows of data: 'Chambre' (11 m2), 'WC' (2 m2), and 'Exterieur' (120 m2). Each row has a delete button (X) on the right. The 'Equipement' section contains four rows of data: 'Prise de courant 16A', 'Prise de courant 16A', 'Prise de courant 16A', and 'Prise de courant 32A'. Each row has a delete button (X) on the right. At the bottom, there is a navigation bar with buttons: 'Accueil', 'Logement', 'Circuits', 'Elements', 'Tableau', and 'Bon de commande'.

Figure 7.7.: Finer-grained handling of feature instances in the *Logement* tab

Rexel also required to be able to assign Equipements defined into the Logement tab to Circuits defined in the view of the same name. Theoretically, this request is supported by TVL through the *shared* feature construct. As a reminder, such features can have several parents. In this case, each Equipement would have a Piece and a Circuit as parents. 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 case study allowed us to get accurate requirements for shared constructs.

The generator should be modified accordingly. In the meantime, a solution has been manually developed. It is visible in Figure 7.8. There, the Equipements sur le circuit list box contains all Equipements defined in all Pieces of the Logement tab.

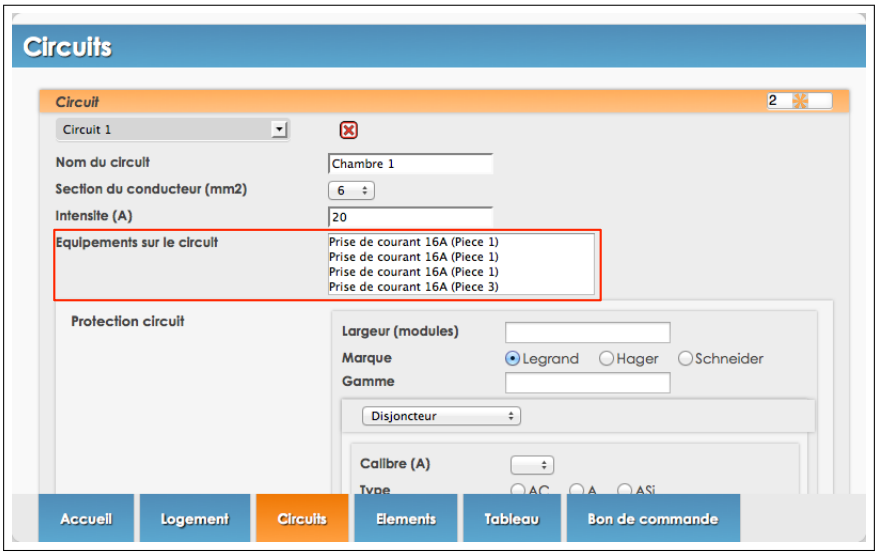


Figure 7.8.: Shared feature in the *Circuits* tab

7.2.4 Lessons Learned

Here, we report our findings about the approach, including the solver, the TVL, TVDL and FCSS languages, the presenter or the generated configurator.

COMPLETENESS OF TVL. This Rexel case study confirmed some outcomes of our previous evaluation of the TVL language (see Section 7.1). Among them, we can mention feature instances. Such constructs were added in TVL₂ based on the previous evaluation and have proved helpful in this case too. In the Rexel TVL model, we count four duplicable features. The same comment applies to string attributes added in TVL₂ and used ten times in our 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. As previously mentioned, the view definition language has been challenged. It turned out that it supports all views re-

quired by Rexel with one exception. To deal with this weakness, an abstract feature was added right under the root feature (see line 28 of Listing 7.1). 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. Even though there are grounds for criticism. First, according to Rexel, it should be possible to define the position of a label, before or after the TVL construct with which it is associated, as illustrated by the m^2 label for each Piece in Figure 7.7. 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 Rexel. Finally, colours could also be defined for elements to be rendered in the GUI. Our partner company also mentioned that it should be possible to generate the same interface in several languages with different FCSS models. For this purpose, they suggest to 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 iterative work with Rexel, the generated interfaces provided valuable input to initiate discussion. Changing the variability in the TVL model 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 electricians. As expected, Rexel does not envision to reuse the generated configuration GUI as-is in its commercial products. There are several reasons for this, including the graphical charter of the company, legacy tools, etc. As mentioned at the beginning of this dissertation, generating ready-to-use configuration GUIs is not our goal, mainly because it is hardly possible.

PROPAGATION STRATEGIES. In the electrical panel example, 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. Thanks to the Rexel case, we identified 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 ren-

dered 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 was applied in our industrial case study. 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. This approach is adapted to this specific case study. Rexel has drawn our attention that it might not be the case for other configurators. 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.

7.2.5 *Threats to validity*

The approach was applied on a single case study. It is therefore difficult to make general statements about the quality of our work. Yet, it shows that it fits a randomly chosen tool, partially proving its generic nature. Furthermore, the approach was proposed to users with different profiles and they were also satisfied by the configuration GUIs tailored to their needs.

At the beginning, the approach was introduced with a basic version of the solver. This could be a threat as the participants might have evaluated the ergonomics of the *HTML* Web page rather than the whole approach. We believe that our iterative approach with Rexel where a full-fledged solver has been added in a latter stage tackles that threat. Introducing the two major aspects of the approach at different different times avoids information overload and confusion. In latter stages, both components were jointly assessed.

The TVL, TVDL and FCSS models were written by the author of this thesis who also designed the generator. Our interlocutors thus have a limited hands-on experience of the three languages, their editors and the generator. Yet, the models were presented to them and edited during workshops and meetings. It did not prevent them to assess our languages by requesting changes and improvements in the proposed models.

The approach was evaluated on an *HTML* generator. Consequently, we cannot draw general conclusions for other target languages. However, it gives us a glimpse of the industrial interest about such an approach. Only time will tell which target languages are relevant for industrials.

As for some case studies in Section 7.1, we did not get access to the code base of the existing tool for electrical panels. This made the modelling of the TVL model more difficult. Furthermore, the electrical domain was partially unknown to us.

7.3 FURTHER EVALUATIONS

The two evaluations that we already performed allowed us to validate and identify improvements for our languages and tools. However, ideally, further evaluations should be performed to pursue the work beyond this thesis. In particular, an evaluation of the overall approach we propose would be a interesting complement to what we have already achieved.

The reviews for two vision papers related to our approach [Boucher et al., 2012a, Boucher et al., 2012d] were positive. Among them, we can mention *“an interesting [...] approach to improve reliability in existing configurators”*, *“it is a real and very common problem”*, or *“this is an important topic that still has not been addressed satisfactorily in industry, despite several research initiatives in this field”*. Furthermore, several master students from the University of Namur worked on it. This sharing of ideas during internships or lectures such as “Advanced topics in software engineering and information systems” (INFOM435¹) provide an even stronger base for our approach. Such informal comments cannot, however, substitute for a formal evaluation. In the following paragraphs, we sketch of a possible way in which such an evaluation should be performed.

The first task would be to define a scenario for the experiment, ideally based on a business case. That scenario would be composed of at least two steps. In the first one, a comprehensive description of the problem would be given, either in a document or by organising meetings and workshops together with the client, or ourselves if the provided scenario is an illustrative one. In other steps, the requirements of the initial problems should evolve. Such changes might impact the variability of the represented product, its constraints, etc. The purpose of such changes would be to assess the effort required to change the configuration GUI.

The second activity would be to select a group of experimenters. In most cases, researchers enrol master students from their university. Alternatives exist. Possible solutions are to get in touch with Web designers schools or with professional Web developers. In our case, the first one seems more appropriate as hiring professionals would be expensive. But finding a representative number of student Web developers can be tricky too as the evaluation might require a lot of time. For this purpose, long term co-operation with such schools might be envisioned.

In the third step, available developers would randomly be split into two groups. They would both get the same information about the Web configurator to develop, the requirements, etc. based on the information defined in the first step. Each developer of the first group would build her Web configurator from scratch. On the other hand, each member of the second group

¹ <http://directory.unamur.be/teaching/courses/INFOM435>

would use the approach proposed in this thesis as a starting point, based on a preliminary introduction to it and the supporting tools. The generated configurator would then probably have to be customized. At the end of the first step, they would be asked to answer evaluation questions about their development process, the experienced difficulties. Examples of such questions are provided in the following paragraph. The second group would probably get additional questions specific to our approach. Then, all developers would have to iterate through the different scenarios defined in the first step. The purpose of requirements changes would be to compare the evolvability of generated interfaces with coded ones. At the end of each iteration, questions related to the changes would be asked to developers.

The questions are not yet set. But we could group them into two categories: the quality of the generated Web page and the development process. For the first category, we can rely on external and existing resources such as the client, scenarios with predefined outputs, and standards such as [W3C, 2013]. Examples of development-related questions are provided hereunder:

1. *What was the percentage of custom developments?*
2. *How long did it take to develop a) the Web page, b) the controller?*
3. *How long did it take to modify the Web configurator according to requirement changes?*
4. *How hard was it to locate the changes required by the modified requirements?*
5. *What was the major difficulty?*
6. *For the second group, which constructs/information are missing in the TVL, TVDL, and FCSS models?*
7. *For the second group, what is the learning curve for the different languages and groups?*
8. ...

This list of questions will have to be extended in order to have a comprehensive comparison between our approach and a more traditional one.

CONCLUSIONS

8.1 SUMMARY OF CONTRIBUTIONS

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 thesis, we propose 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. The language has been evaluated on four industrial case studies. That evaluation has led to the definition of TVL₂ which now supports, inter alia, feature instances and string attributes. *Xtext* has been used to develop a TVL editor which provides syntax highlighting, auto-completion, scoping, etc. and comes with a meta-model of the language which can be used to generate graphical configurators.

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 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 for TVL, an editor has been developed for TVDL. That editor comes with a meta-model of the language.

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. As for feature and view modelling languages, an editor has been developed for FCSS.

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, model queries, can be massively reused.

The tools and languages were evaluated together within a multinational company on one of their existing products. That company, namely Rexel, is currently re-engineering its electrical panel configuration tool. Our approach and the generator were used iteratively to demonstrate and evaluate the ca-

pabilities 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 experience with Rexel demonstrated the utility of the approach and allowed to identify various improvement opportunities.

8.2 LIMITATIONS

In this section, we present the limitations of the proposed approach, languages, the prototype generator and the future work that can solve or minimize them.

SINGLE PLATFORM GENERATION. Our transformations only support *HTML* generation. This limitation is partly due to immaturity, unavailability and complexity of existing UIDLs. Future work will explore the possibility of domain-specific UIDLs (see below).

However, we think that *HTML* is a good candidate given that most Web configurators are developed in that language. Alternatives are, for example, Flash Web sites. Furthermore, as previously mentioned, moving from a model-to-text transformation to a model-to-model transformation should not be complicated given that the most complex part of the *HTML* generator could be re-used.

LACK OF DYNAMIC ASPECTS. Generated *HTML* configurators are static ones. By this we mean that configuration widgets corresponding to all feature model constructs are stored in the Web page at loading time, even those which are not available. Some of them might never become available, making the file size unnecessarily large. Instead, unavailable contents should be dynamically added to the Web page as soon as it becomes available. It has already been done for feature instances. A next version of the prototype could be built using a similar technique.

We think that this threat is rather limited in our case given that the generated *HTML* files are used as a springboard for discussion. We do not intend, at the moment, to make them available out of the box to final users or customers on the Internet.

EVALUATION. Ideally, the proposed approach should have been evaluated in a more systematic way. In this thesis, the main evaluation is based on an industrial case study. Although the results for the Rexel case are encouraging, we cannot claim that the approach will be suitable in all circumstances. That case allowed us to find some limitations of the languages and the prototype

generator. Other case studies could lead us to discover needs of improvement.

To tackle that limitation, we propose to evaluate the approach and the associated languages on several industrial cases. If possible, that assessment should be conducted in a more guided way in order to be able to compare the outputs from the different cases in a more systematic way.

QUALITY OF THE GENERATED CONFIGURATORS. As raised by Rexel, generated configurators cannot always be directly put in the user's hand. They are graphical user interfaces which require some customization efforts. While the FCSS language alleviates this problem, our primary goal was to generate functionally correct configurators, being aware that the ultimate layout and beautification operations will be left to designers.

Nichols and Faurling already mentioned that automatically generating fully-functional user interfaces has been successfully applied in limited domains [Nichols and Faulring, 2005]. Among them, we can cite remote controls [Nichols et al., 2002] and dialogue box design [Kim and Foley, 1993]. Whether configuration engineering can reach that level of maturity is still an open question.

8.3 PERSPECTIVES

8.3.1 *Reverse-engineering*

In this thesis, we focus on the development of new configurators. We believe that our approach could be used together with reverse-engineering techniques in order to migrate and update existing *ad-hoc* configurators. That point has already been addressed elsewhere for Web configurators [Abbasi et al., 2014]. There, Abbasi *et al.* propose the supervised and semi-automatic reverse-engineering process depicted in Figure 8.1. The user starts with defining *variability data extraction patterns* (vde patterns) for a given Web page in an *HTML*-like language (①). For a given Web page and a given vde pattern, the Web Wrapper extracts data which correspond to the pattern and save it in an XML format (②). Some user configurations are also simulated in order to extract dynamic contents like, e.g., cross-tree constraints (③). The contents extracted in the second and third steps can then be edited (④) and are transformed into a TVL feature model in step (⑤). Actually, several FMs can exist, e.g., one for each configuration step. They are merged by FAMILIAR (⑥) a tool-supported language to merge incomplete FMs into a single FM [Acher et al., 2013].

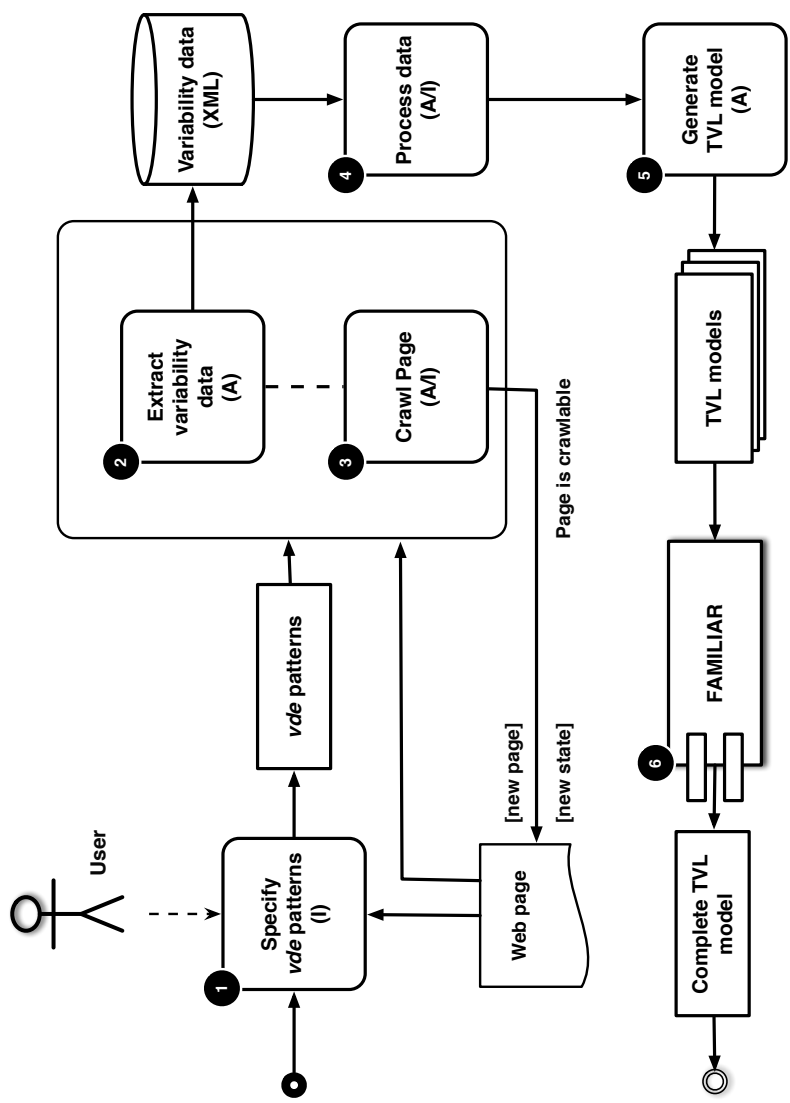


Figure 8.1.: Reverse-engineering process for Web configurators (taken from [Abbasi et al., 2014])

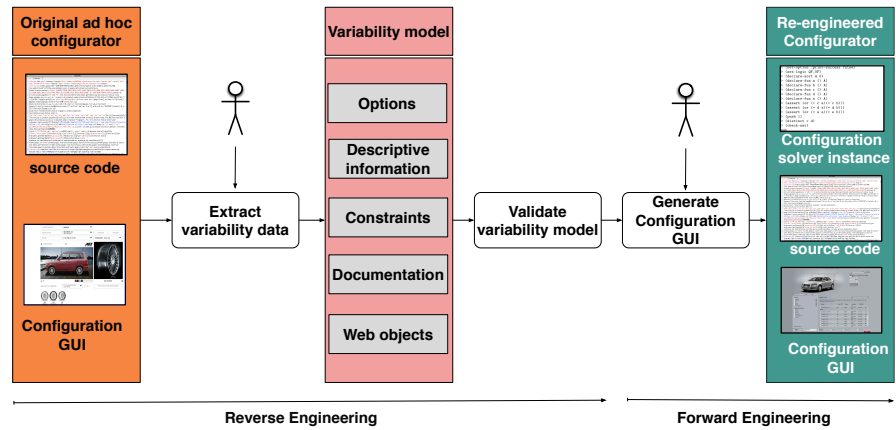


Figure 8.2.: Re-engineering process for configurators

Figure 8.2 presents the full re-engineering process, the contribution of this thesis being on the right side. Interested readers might refer to [Boucher et al., 2012a] for more detailed information.

8.3.2 Multiple Targets

We envision two solutions to target multiple output languages. The critical point is to have a 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. That work is in its exploratory phase in our research laboratory. We do not discuss the content of our custom UIDL here as the current version, based on our work, is still under development. At the moment, a prototype generator produces simple Swing configurators.

8.3.3 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. As we have seen with the Audi example in Chapter 1 (see Figure 1.1), 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 [Hubaux et al., 2009] or multi-step SPL configuration [White et al., 2014] 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.

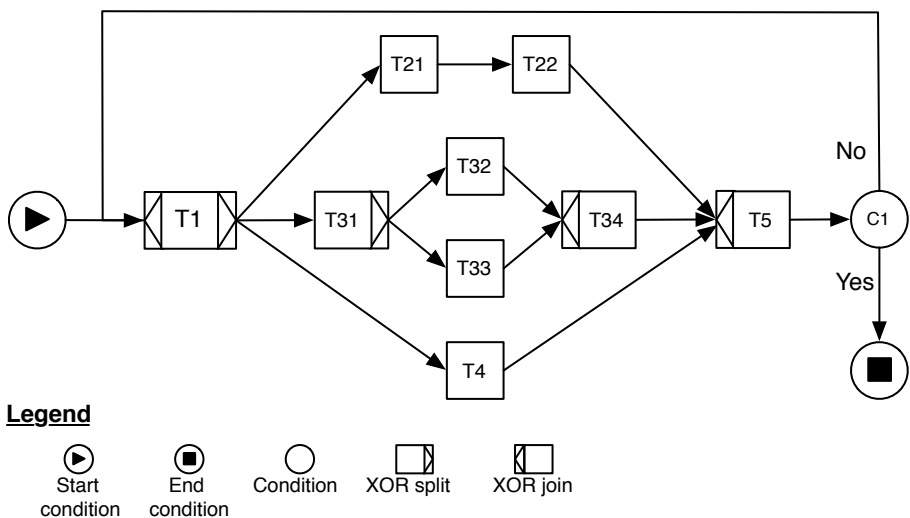


Figure 8.3.: Illustrative YAWL workflow for a configuration GUI

Figure 8.3 is an example of such workflow expressed in the YAWL formalism [ter Hofstede et al., 2010]. There, T* labels correspond to tasks and the C1 label refers to a condition. Our complex GUI is composed of a home screen (T1) where the user has three options (T21, T31, T4). Once she has chosen one, she enters a sub-workflow which can be composed of a single task (T4), a sequence of tasks (T21-T22) or more complex patterns (T31, T32, T33, T34). Once the tasks of the selected path are complete, task T5 is reached. It corresponds to a dummy join task. Then comes condition C1 which is used to determine if the configuration is finished. If the condition is not satisfied, the workflow goes back to task T1, so creating a loop. On the other side, a positive answer would mean that the configuration is finished. This condition could either be automatic (the workflow loops until there is no more variabil-

ity to be configured) or manual (the user clicks on a button to indicate that she has completed her configuration task).

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.

8.3.4 Workflow Configuration

In [Boucher et al., 2012b], we also proposed to combine workflows with our approach for the workflows of the ISO/IEC 29110 standard [ISO/IEC, 2010]. The solution is an extension of another one devoted to configurable workflows [Gottschalk et al., 2008] and the *Synergia* tool suite [La Rosa et al., 2007]. That first solution was introduced in [Boucher et al., 2012c].

Our research took place within the NAPLES project¹ which aimed at providing an affordable approach to software lifecycle management. In that context, our focus was on the operationalization of the workflows from the ISO/IEC 29110 standard with the *Bonita* [Bonita, 2014] workflow engine.

Such workflows being configurable, we proposed to use variability models to configure them. The workflow configuration approach can be decomposed into four steps:

1. **Variability identification.** The user has to extract the variability from the workflows' definition. *Bonita* does not support configurable (i.e., optional) tasks. As a consequence, we proposed to tag workflow transitions with conditions on variable values. A transition should thus be made available for each variable value.
2. **Questionnaire creation.** For each variation point identified in the first step, a question has to be created in the workflow configuration questionnaire. *Bonita* comes with a questionnaire editor. However, it is a simple drag-and-drop tool where it is not possible to define constraints between the different questions/answers. As a consequence, we proposed to use our GUI generation approach based on TVL.
3. **Workflow & FM connection.** A link has to be defined between the workflow variables and the features from the FM. We proposed to use a database which, for each answer given by the user (i.e., feature selected), will store the corresponding variable value (for the workflow). At runtime, *Bonita* will access that database to check conditions on transitions.

¹ See <http://www.cetic.be/NAPLES>, 1162

4. **Configuration and workflow running.** Finally, the user has to answer questions from the questionnaire and the workflow will be run accordingly using the conditions on transitions and database values.

Figure 8.4 depicts the project planning workflow taken from the ISO/IEC 29110 standard [ISO/IEC, 2010]. It contains eight variables. Four of them are related to the workflow configuration through the questionnaire, namely two to the `GetStatementWorkReview` task and the two others to `GetProjectPlanReview`. The value of the other variables is determined by tasks of the workflow, namely `getparallelworkflow` and `Plan Review` by Customer.

The TVL FM for the questionnaire is visible in Listing 8.1. There, only SWR feature and its sub-features, i.e., `SWR_Immediate` and `SWR_Meeting`, are useful for the workflow depicted in Figure 8.4. Other features correspond to other workflows not presented here.

Listing 8.1: TVL model for the project planning workflow questionnaire

```

1  root Tune {
2      PPR_Meeting -> IMRT;
3      SWR_Meeting -> IMRT;
4      RSMC_No -> RATC_Yes;
5      group allof {
6          SWR group oneof {SWR_Immediate, SWR_Meeting},
7          PPR group oneof {PPR_Immediate, PPR_Meeting},
8          opt IMRT group oneof {Specific, Standard},
9          RAT group oneof {QC, Redmine},
10         opt RAW group oneof {OSL},
11         RSMC group oneof {RSMC_No, RSMC_Yes},
12         RATC group oneof {RATC_No, RATC_Yes},
13         PW group oneof {PW_No, PW_Yes}
14     }
15 }
```

Finally, the questionnaire depicted in Figure 8.5 is generated by our approach with additional data contained in an FCSS model not presented here. In that case, the controller had to be extended to store the user answers in a database in order to be able to run the *Bonita* workflow.

That prototype was developed for workflows from ISO/IEC standards but it could easily be applied to configure e-commerce Web sites, for example.

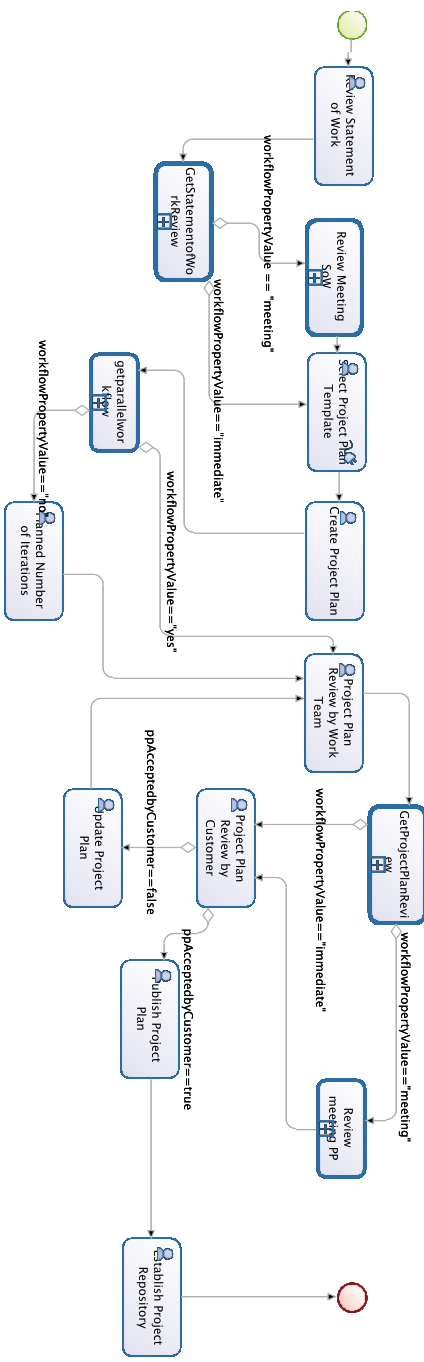


Figure 8.4.: Project planning workflow

Tune Development Method

Project : Naples

Release : JLDP 2012

Statement of Work Review	immediate
Project Plan Review	immediate
Internal Meeting Record Template	meeting specific
Requirement Analysis Tool	HP QC
Requirement Analysis Workflow	none
Requirement Send Mail to Customer	no
Requirement Assign Task to Customer	no
Parallel Workflow	yes

Next

Figure 8.5.: Questionnaire for the project planning workflow

Part IV

APPENDIXES



LANGUAGE GRAMMARS

A.1 TVL GRAMMAR

```
1 grammar be.unamur.TVL with org.eclipse.xtext.common.Terminals
2
3 generate tvl "http://www.unamur.be/TVL"
4
5 Model:
6     model+=ModelElement*;
7
8 ModelElement:
9     Type
10    | Constant
11    | Root_Feature
12    | Feature_Extension;
13
14 /* TYPE SECTION */
15 Type:
16     SimpleType
17     | Record;
18
19 SimpleType:
20     type='int' name=ID ('in' domain=Set_Expression)? ';'
21     | type='real' name=ID ('in' domain=Set_Expression)? ';'
22     | type='enum' name=ID 'in' enumDomain=Enum_Expression ';'
23     | type='bool' name=ID ';'
24     | type='string' '<' size=INT '>' name=ID ';'
25
26 Record:
27     type='struct' name=ID '{' fields+=Record_Field+ '}';
28
29
30 Record_Field:
31     type='int' name=ID ('in' domain=Set_Expression)? ';'
32     | type='real' name=ID ('in' domain=Set_Expression)? ';'
33     | type='enum' name=ID 'in' enumDomain=Enum_Expression ';'
34     | type='string' name=ID ';'
35     | type='bool' name=ID ';'
36     | typeref=[SimpleType|ID] name=ID ';'
37
```

```

38 /* CONSTANT SECTION */
39 Constant:
40     'const' type='int' name=ID value=Integer ';'
41     | 'const' type='real' name=ID value=Real ';'
42     | 'const' type='bool' name=ID (value='true' | value='false') ';'
43     | 'const' type='string' name=ID value=STRING ';'
44     ;
45
46 /* ID SECTION */
47 Common_Feature: Feature_Declaration | Feature_Extension;
48
49 Feature_Declaration: Root_Feature | Hierarchical_Feature;
50
51 Short_ID:
52     Feature_Declaration
53     | Attribute
54     | Constant
55     | Enum_Element
56     ;
57
58 Short_IDTail:
59     Feature_Declaration
60     | Attribute
61     | Shared_Feature
62     | Sub_Attribute
63     ;
64
65 Common_Short_ID: Short_ID | Short_IDTail;
66
67 Long_ID:
68     keyword=Shortcut tail=Long_IDTail
69     | head=[Short_ID] (tail=(Long_IDTail))?;
70
71 Long_IDTail:
72     '.' head=[Short_IDTail] (tail=Long_IDTail)?;
73
74
75 Shortcut:
76     'parent'
77     | 'this'
78     | 'root';
79
80
81 /* FEATURE SECTION */
82
83 Root_Feature: 'root' name=ID body=Feature_Content;
84
85 Feature_Extension: ref=Long_ID body=Feature_Content;
86
87 Feature_Content:
88     '{' bodyItems+=Feature_Body_Item+ '}'
89     | group=Feature_Group;
90

```

```

91 Feature_Body_Item:
92     Data
93     | Constraint
94     | Attribute
95     | Feature_Group;
96
97 Feature_Group:
98     'group' cardinality=Cardinality '[' sub_features+=Sub_Feature (' ' sub_features+=
99     Sub_Feature)* ']';
100
101 Hierarchical_Feature:
102     optional?=('opt')? name=ID cardinality=BasicCardinality? body=Feature_Content?;
103
104 Shared_Feature:
105     'shared' ref=Long_ID;
106
107 Sub_Feature:
108     Hierarchical_Feature
109     | Shared_Feature;
110
111 Cardinality:
112     value=('oneof' | 'oneOf')
113     | value=('someof' | 'someOf')
114     | value=('allof' | 'allOf')
115     | '[' min=Bound '..' max=Bound ']';
116
117 BasicCardinality:
118     '[' min=Bound '..' max=Bound ']';
119
120 Bound:
121     boundRef=[Constant]
122     | boundInt=INT
123     | boundAst='*';
124
125 Feature_Scope: Attribute | Hierarchical_Feature;
126
127 FQN: Feature_Declaration | Attribute;
128
129 /* ATTRIBUTE SECTION */
130 Attribute:
131     Base_Attribute
132     | {Structure_Attribute} type=[Record] name=ID cardinality=BasicCardinality? '['
133     sub_attributes+=Sub_Attribute+ ']';
134
135 Base_Attribute:
136     type='int' name=ID cardinality=BasicCardinality? attr_body=Attribute_Body? ';'
137     | type='real' name=ID cardinality=BasicCardinality? attr_body=Attribute_Body? ';'
138     | type='bool' name=ID cardinality=BasicCardinality? attr_body=Attribute_Body? ';'
139     | type='string' name=ID cardinality=BasicCardinality? attr_body=Attribute_Body? ';'
140     | type='enum' name=ID cardinality=BasicCardinality? 'in' domain=Enum_Expression ('is'
141     attr_value=Expression | ',' attr_condition=Attribute_Conditional)? ';'
142     | predefined_type=[SimpleType | ID] name=ID cardinality=BasicCardinality? attr_body=
143     Attribute_Body? ';'

```

```

140      ;
141
142  Attribute_Body:
143      'is' attr_value=Expression
144      | 'in' attr_value_set=Set_Expression(',' attr_condition=Attribute_Conditional)?
145      | ',' attr_condition=Attribute_Conditional;
146
147  Attribute_Conditional:
148      (' ifin ':' | ' ifIn ':' ) (' is' ifin_condition=Expression12 | 'in' ifin_condition_set=
149          Set_Expression) (',' (' ifout ':' | ' ifOut ':' ) (' is' ifout_condition=Expression12 | 'in'
150              ifout_condition_set=Set_Expression))?
151      | (' ifout ':' | ' ifOut ':' ) (' is' ifout_condition=Expression12 | 'in' ifout_condition_set=
152          Set_Expression);
153
154  Sub_Attribute:
155      sub_id=[Record_Field] attr_body=Attribute_Body ',';
156
157  Enum_Expression: '{' list=Enum_List '}';
158
159  Enum_List: enums+=Enum_Element (',' enums+=Enum_Element)*;
160
161  Enum_Element: name=ID;
162
163  /* Expression SECTION */
164
165  Expression12 returns ComplexExpression:
166      Expression11 =>({If.left=current} '?' right+=Expression12 ':' right+=Expression11)*;
167
168  Expression11 returns ComplexExpression:
169      Expression10 =>({LeftImplication.left=current} '<-' right=Expression10)*;
170
171  Expression10 returns ComplexExpression:
172      Expression9 =>({RightImplication.left=current} '->' right=Expression9)*;
173
174  Expression9 returns ComplexExpression:
175      Expression8 =>({BiImplication.left=current} '<->' right=Expression8)*;
176
177  Expression8 returns ComplexExpression:
178      Expression7 =>({Or.left=current} '|' right=Expression7)*;
179
180  Expression7 returns ComplexExpression:
181      Expression6 =>({And.left=current} '&&' right=Expression6)*;
182
183  Expression6 returns ComplexExpression:
184      Expression5 =>((((Equality.left=current} '==' | {Inequality.left=current} '!=') right=
185          Expression5)* | ({In.left=current} 'in') right=(Set_Expression)))?;
186
187  Expression5 returns ComplexExpression:
188      Expression4 =>(((Less.left=current} '<' =>({Lessequal.left=current} '=<') | {Greater.left=current}
189          '>' =>({Greaterequal.left=current} '=>')) right=Expression4)*;
190
191  Expression4 returns ComplexExpression:

```

```

188     Expression3 =>(((Plus.left=current)+' ' | {Minus.left=current}-') right=Expression3)*;
189
190 Expression3 returns ComplexExpression:
191     Expression2 =>(((Multiplication.left=current)*' ' | {Division.left=current}/') right=
192         Expression2)*;
193
194 Expression2 returns ComplexExpression:
195     Expression =>({Excludes.left=current}excludes' | {Requires.left=current}requires')
196         right=Expression)*;
197
198 Expression:
199     value='true'
200     | value='false'
201     | value=Integer
202     | value=Real
203     | ref=Long_ID
204     | op='!' expression=Expression
205     | op='-' expression=Expression
206     | op='(' expression=Expression12 ')'
207     | op='abs' '(' expression=Expression12 ')'
208     | op='sum' '(' (expression_list=Expression_List | child=Children_ID | values=Values_Set)
209         ')'
210     | op='mul' '(' (expression_list=Expression_List | child=Children_ID) ')'
211     | op='min' '(' (expression_list=Expression_List | child=Children_ID | values=Values_Set)
212         ')'
213     | op='max' '(' (expression_list=Expression_List | child=Children_ID | values=Values_Set)
214         ')'
215     | op='count' '(' (children='children' | children='selectedchildren' | instances=
216         Filtered_Instances_Set) ')'
217     | op='avg' '(' (expression_list=Expression_List | child=Children_ID | values=Values_Set)
218         ')'
219     | op='and' '(' (expression_list=Expression_List | child=Children_ID | values=Values_Set)
220         ')'
221     | op='or' '(' (expression_list=Expression_List | child=Children_ID | values=Values_Set)
222         ')'
223     | op='xor' '(' (expression_list=Expression_List | child=Children_ID | values=Values_Set)
224         ');
225
226 Expression_List:
227     expressions+=Expression12 ('' expressions+=Expression12)*;
228
229 Set_Expression:
230     '[' list=Expression_List ']'
231     | '[' min=Set_Expression_Bound '..' max=Set_Expression_Bound ']' ;
232
233 Set_Expression_Bound:
234     Integer
235     | Real
236     | '*';
237
238
239
240

```



```

231 Values_Set:
232     values=Mapped_Values ('.filter' '(' expression=Expression12 ')')?
233 ;
234 Mapped_Values:
235     instances_set=Filtered_Instances_Set '.map' '(' expression=Expression12 ')'
236 ;
237
238 Filtered_Instances_Set:
239     instances_set=Terminal_Instances_Set ('. filter' '(' expression=Expression12 ')')?
240 ;
241
242 Terminal_Instances_Set:
243     instances=[Feature_Declaration] ;
244
245
246 Temporary_Variable:
247     'var' ' name=ID
248 ;
249
250 Children_ID:
251     child='selectedchildren.' ref=Long_ID
252 | child='children.' ref=Long_ID;
253
254 /* CONSTRAINTS SECTION */
255 Constraint:
256     condition=('ifin ':'|' ifIn :) expression=Expression12';'
257 | condition=('ifout ':'|' ifOut :) expression=Expression12';'
258 | expression=Expression12';';
259
260 /* DATA SECTION */
261 Data:
262     'data' '{' pairs+=Data_Pair+ '}';
263
264 Data_Pair:
265     key=STRING value=STRING;
266
267 Integer:
268     ('-')? INT;
269
270 Real:
271     Integer '.' INT+;
272
273 terminal STRING :
274     ''' _.*_ ''' ;

```

A.2 TVDL GRAMMAR

```

1 grammar be.unamur.TVDL with org.eclipse.xtext.common.Terminals
2
3 import "http://www.unamur.be/TVL" as tvl
4
5 generate tvdl "http://www.unamur.be/TVDL"
6
7 TVDLModel:
8     tvl=Import
9     views+=View*;
10
11 Import: 'import' importURI=STRING;
12
13 View: View_Grouping | View_Declaration;
14
15 View_Grouping: '$' name=ID '{' subViews+=[View] ('&&' subViews+=[View])* '}';
16
17 View_Declaration: name=ID '{' expressions+=ViewExpression ('&&' expressions+=
18     ViewExpression)* '}';
19
20 ViewExpression: id=TVL_ID (refinement=ViewExpressionRefinement)?;
21
22 ViewExpressionRefinement:
23     ":" (subtree=SubtreeExpression | attributes=AttributeExpression | group=GROUP | list=
24     Common_List);
25
26 SubtreeExpression: keyword=SUBTREE (stopList=Stop_List)? (refinement=
27     SubtreeExpressionRefinement)?;
28
29 SubtreeExpressionRefinement: "|" (attributes=AttributeExpression | groups=GroupExpression
30     | list=Common_List);
31
32 AttributeExpression: keyword=ATTRIBUTES ( "|" list=Common_List)?;
33
34 GroupExpression: keyword=GROUPS ( "|" list=Common_List)?;
35
36 Stop_List: ("/" stopElements+=List_Element)+;
37
38 Common_List: Exclusion_List | Inclusion_List;
39
40 Exclusion_List: "!" "[" elements+=List_Element (',' (elements+=List_Element))* "]";
41
42 Inclusion_List: "[" elements+=List_Element (',' (elements+=List_Element))* "]";
43
44 List_Element:
45     id=TVL_ID
46     | keyword=(ATTRIBUTES | GROUP | GROUPS);
47
48 TVL_ID:
49     head=[tvl::Feature_Declaration] tail=TVL_IDTail
50     | head=[tvl::Attribute];

```

```
47 TVL_IDTail: '.' head=[tv1::FQN] (tail=TVL_IDTail)?;
48
49
50 terminal GROUP: "group";
51 terminal GROUPS: "groups";
52 terminal SUBTREE: "*";
53 terminal ATTRIBUTES: "attributes";
```

A.3 FCSS GRAMMAR

```

1 grammar be.unamur.FCSS with org.eclipse.xtext.common.Terminals
2
3 import "http://www.unamur.be/TVL" as tvl
4 import "http://www.unamur.be/TVDL" as tvdl
5
6 generate fcss "http://www.unamur.be/FCSS"
7
8 FCSSModel:
9     tvlImp=Import (tvdlImp=Import)? parts+=Part+;
10
11 Import: 'import' importURI=STRING;
12
13
14 Part:
15     FCSS_Feature
16     | FCSS_Attribute
17     | FCSS_Global
18     | FCSS_View;
19
20 FCSS_Feature: feature=Feature_ID '{' attributes+=FeatureAttribute+ '}';
21
22 FCSS_Attribute: '#' attribute=Attribute_ID '{' attributes+=AttributeAttribute+ '}';
23
24 FCSS_View: '$' view=[tvdl::View] '{' attributes+=ViewAttribute+ '}';
25
26 FCSS_Global: '.' '{' attributes+=GlobalAttribute+ '}';
27
28 Feature_ID: head=[tvl::Feature_Declaration] ( tail=Feature_IDTail)?;
29
30 Feature_IDTail: '.' head=[tvl::Hierarchical_Feature] ( tail=Feature_IDTail)?;
31
32 Attribute_ID: head=[tvl::Feature_Declaration] tail=Attribute_IDTail
33             | head=[tvl::Attribute];
34 Attribute_IDTail: '.' head=[tvl::Feature_Scope] (tail=Attribute_IDTail)?;
35
36 GlobalAttribute:
37     globalType='andGroup' ':' globalValue=('textbox') ':'
38     | globalType='orGroup' ':' globalValue=('listbox' | 'checkbox') ':'
39     | globalType='xorGroup' ':' globalValue=('listbox' | 'radiogroup') ':'
40     | globalType='cardGroup' ':' globalValue=('checkbox') ':'
41     | globalType='feature' ':' globalValue=('text' | 'image') ':'
42     | globalType='optFeature' ':' globalValue=('checkbox' | 'listbox' | 'radiogroup') ':'
43     | globalType='selectFeature' ':' globalValue=('checkbox' | 'listbox' | 'radiogroup') ':'
44     | globalType='intAttribute' ':' globalValue=('textbox') ':'
45     | globalType='realAttribute' ':' globalValue=('textbox') ':'
46     | globalType='boolAttribute' ':' globalValue=('checkbox' | 'listbox') ':'
47     | globalType='enumAttribute' ':' globalValue=('listbox' | 'radiogroup') ':'
48     | globalType='groupContainer' ':' globalValue=('true' | 'false') ':'
49     | globalType='unavailableContent' ':' globalValue=('hidden' | 'greyed' | 'none')
50     | globalType='view' ':' globalValue=('tab') ':' ;

```

```

51
52 FeatureAttribute:
53     featureType='widget' ':' featureValue=('text' | 'image') ';'
54     | featureType='opt' ':' featureValue=('checkbox' | 'listbox' | 'radiogroup') ';'
55     | featureType='select' ':' featureValue=('checkbox' | 'listbox' | 'radiogroup') ';'
56     | featureType='group' ':' groupAttributes+=GroupAttribute+ ';'
57     | CommonAttribute;
58
59 AttributeAttribute:
60     attributeType='widget' ':' attributeValue=('textbox' | 'checkbox' | 'listbox' | '
        radiogroup') ';'
61     | CommonAttribute;
62
63 ViewAttribute:
64     viewType='widget' ':' viewValue=('tab') ';'
65     | viewType=generate ':' viewValue=('true' | 'false') ';'
66     | CommonAttribute
67     | GlobalAttribute;
68
69 CommonAttribute:
70     commonType='label' ':' commonValue=STRING ';'
71     | commonType='help' ':' commonValue=STRING ';'
72     | commonType='unavailable' ':' commonValue=('hidden' | 'greyed' | 'none');
73
74 GroupAttribute:
75     groupType='widget' ':' groupValue=('textbox' | 'listbox' | 'checkbox' | 'radiogroup')
        ';'
76     | groupType='container' ':' groupValue=('true' | 'false') ';'
77     | groupType='default' ':' defaultSubFeature=[tv1::Hierarchical_Feature] ';'
78     | CommonAttribute;
79
80 terminal STRING :
81     '"' _.* _ '"' ;

```

PROTOTYPE GENERATOR

```

1 package parserHtmlToJS;
2
3 import java.io . File ;
4 import java.io . FileWriter ;
5 import java.io . IOException ;
6 import java.io . InputStream ;
7 import java.io . UnsupportedEncodingException ;
8 import java . util . List ;
9 import java . util . Properties ;
10
11 import org.apache.http.HttpResponse;
12 import org.apache.http.client . HttpClient ;
13 import org.apache.http.client . methods . HttpPost ;
14 import org.apache.http.entity . StringEntity ;
15 import org.apache.http.impl.client . DefaultHttpClient ;
16 import org.apache.http.message . BasicHeader ;
17 import org.apache.http.protocol . HTTP ;
18 import org.json.simple . JSONArray ;
19 import org.json.simple . JSONObject ;
20 import org.jsoup . Jsoup ;
21 import org.jsoup.nodes . Attribute ;
22 import org.jsoup.nodes . Document ;
23 import org.jsoup.nodes . Element ;
24 import org.jsoup.nodes . Node ;
25 import org.jsoup.nodes . TextNode ;
26
27
28 public class Parser {
29
30     private static String htmlFilePath; //The absolute path to the HTML file
31
32     private static String targetHtmlFilePath; //The target path for the HTML file
33
34     private static String serverAddress; //Address of the server where you upload the json
35         file
36     /**

```

```

37      * Remove unnecessary clones from the initial html configurator (when the minimum
38      * cardinality is zero)
39      * and create a copy of each clone in the json format on the server
40      */
41      public static void main (String[] args){
42          //Load the parser properties file
43          Properties props = new Properties();
44          InputStream stream = Parser.class.getClassLoader().getResourceAsStream("config.
45              properties");
46          try {
47              props.load(stream);
48          } catch (IOException e) {
49              e.printStackTrace();
50          }
51          htmlFilePath = props.getProperty( "htmlFilePath" ); // Retrieve the absolute path to
52              the HTML file
53          targetHtmlFilePath = props.getProperty( "targetHtmlFilePath" ); // Retrieve the target
54              path for the HTML file
55          serverAddress = props.getProperty( "serverAddress" ); //Retrieve the address of the
56              server where you upload the json file
57
58          Document domObject = getTheHtmlDomObjectFromHtmlFile();
59
60          JSONObject obj = transformTheHtmlDomObjectToJsonFormat(domObject);
61
62          modifyTheHtmlFileWithTheNewHtmlDomObject(domObject);
63
64          sendTheJsonObjectToTheServer( obj );
65
66          System.out.println("Parsing:_done");
67      }
68  /**
69   * Send the Json Object to the server that contains the clones
70   * @param obj          The JSONObject that represents all the clones of the configurator
71   *                      html file
72   */
73  private static void sendTheJsonObjectToTheServer(JSONObject obj) {
74      //Create the post request
75      HttpClient httpclient = new DefaultHttpClient();
76      HttpResponse response = null;
77      HttpPost httpPost = new HttpPost(serverAddress);
78
79      // Initialize the request body with the json object
80      StringEntity se = null;
81      try {
82          se = new StringEntity( obj.toString() );
83      } catch (UnsupportedEncodingException e) {
84          e.printStackTrace();
85      }

```

```

84         se.setContentEncoding(new BasicHeader(HTTP.CONTENT_TYPE, "application/json"))
85         ;
86         httppost.setEntity(se);
87         //Execute the post request
88         try {
89             response = httpclient.execute(httppost);
90         } catch (IOException e) {
91             e.printStackTrace();
92         }
93         // Retrieve the status of the response from the server
94         int statusCode = response.getStatusLine().getStatusCode();
95     }
96
97     /**
98     * Modify the html file with the html dom object without unnecessary clones
99     * @param domObject      The Document dom object that represents the new html page
100    *                        of the configurator
101    */
102    private static void modifyTheHtmlFileWithTheNewHtmlDomObject(Document
103        domObject) {
104        try {
105            //Create the new html file at the location of the old file
106            FileWriter file = new FileWriter(targetHtmlFilePath);
107            file.write(domObject.toString());
108            file.flush();
109            file.close();
110        } catch (IOException e) {
111            e.printStackTrace();
112        }
113    }
114
115    /**
116    * Remove unnecessary clones of the object when the minimum cardinality is zero
117    * and create a json object for each clones encountered in the Document which represents
118    * the initial html
119    * @param domObject      The Document dom object that represents the initial html
120    *                        page of the configurator
121    * @return JSONObject    A json object that contains a copy of all the clones found in
122    *                        the initial html
123    */
124    private static JSONObject transformTheHtmlDomObjectToJsonFormat(Document
125        domObject) {
126        //Warning: absolutely use Node object to preserve the structure of the html document
127        //Retrieve the first relevant nodes in the hierarchy to start the parsing
128        List<Node> racines = domObject.childNodes(1).childNodes(2).childNodes(3).childNodes
129            (1).childNodes();
130        //Create a json array that will contain all the json clones
131        JSONArray list = new JSONArray();
132        //Apply the parsing to each child node
133        for ( int i = 0 ; i < racines.size() ; i++ ) {
134            parsing( racines.get(i), list , true);
135        }
136    }

```



```

129     }
130     JSONObject obj = new JSONObject();
131     //Put the array of json clones in the result object
132     obj.put("clones", list);
133     return obj;
134 }
135
136 /**
137  * Get the html file and transform it into Document object
138  * @return Document      The Document dom object that represents the initial html
                           page of the configurator
139  */
140 private static Document getTheHtmlDomObjectFromHtmlFile() {
141     Document racine = null;
142     try {
143         //Get the html file at the 'htmlFilePath' location
144         File input = new File(htmlFilePath);
145         //Transform the file into a Document object
146         racine = Jsoup.parse(input, "UTF-8");
147     } catch (IOException e) {
148         e.printStackTrace();
149     }
150     return racine;
151 }
152
153 /**
154  * Handle a node of the dom hierarchy by applying parsing to its child nodes in accordance
                           with the different cases
155  * @param node            The Node to parse
156  * @param list            The json array containing all the clones already registered
                           from the dom hierarchy
157  * @param firstClone      A boolean indicating if it is necessary or not to record the
                           next clone met
158  */
159 private static void parsing(Node node, JSONArray list, boolean firstClone) {
160     int i = 0;
161     //If the node has a class attribute
162     if ( node.hasAttr("class") ) {
163         //If the node contains clonable content
164         if ( ((Element)node).hasClass("clonable") && node.childNodes().size()>=5 ) {
165             //This is the first time we encounter this clone, we must create a json object
166             if ( firstClone ) {
167                 //Parse the first clone and create a json object that represents it into
                           the json array
168                 list.add(parsingClone( node.childNodes(3), list, true, 0, true));
169                 firstClone = false;
170                 //If this is not the first time that the clone is encountered and the
                           minimum cardinality is > 0
171             } else if ( !node.childNodes(3).hasAttr("delete") ) {
172                 //Just parse the clone
173                 parsing( node.childNodes(3), list, firstClone );
174             }
175             //Remove the unnecessary 'label' attribute

```

```

176         node.childNode( 3 ).removeAttr("label");
177         // If the minimum cardinality = 0
178         if ( node.childNode(3).hasAttr("delete") ) {
179             //Remove the first clone
180             node.childNode(2).remove();
181             node.childNode(2).remove();
182             i = 2;
183         } else {
184             i = 4;
185         }
186     }
187 }
188 // Retrieve the children of the node
189 List<Node> nodes = node.childNodes();
190
191 //Apply the parsing to each child node not yet visited
192 for (int j = i ; j < nodes.size() ; j++) {
193     parsing( nodes.get(j), list , firstClone );
194 }
195 }
196
197 /**
198  * Handle a node of the dom hierarchy by applying parsing to its child nodes in accordance
199  * with the different cases
200  * and create an object in json format which represents the node
201  * @param node           The Node to parse
202  * @param list           The json array containing all the clones already registered
203  *                        from the dom hierarchy
204  * @param newClone       A boolean indicating if it begins to generate a new clone or
205  *                        not
206  * @param indice         An integer indicating the number of parameters in the first id
207  *                        of the clonable content
208  * @param firstClone     A boolean indicating if it is necessary or not to record the
209  *                        next clone met
210  * @return JSONObject    A json object that contains a copy clonable content found in
211  *                        the initial html
212  */
213 private static JSONObject parsingClone(Node node, JSONArray list, boolean newClone,
214 int indice, boolean firstClone ) {
215     JSONObject obj = new JSONObject();
216     // If the node isn't a TextNode
217     if ( !node.nodeName().equals("#text") ) {
218         //Put the tag name of the html node to the current json object
219         obj.put("kind", node.nodeName());
220
221         //Put all the attributes of the html node to the current json object
222         indice = convertAndAddAttributes( obj, node.attributes().asList(), newClone,
223             indice );
224
225         int i = 0;
226         // If the node has a class attribute
227         if ( node.hasAttr("class") ) {
228             // If the node contains clonable content

```

```

221         if ( ((Element)node).hasClass("clonable") && node.childNodes().size()>=5 ) {
222             //This is the first time we encounter this clone, we must create a json
                object
223             if ( firstClone ) {
224                 //Parse the first clone and create a json object that represents
                    it into the json array
225                 list.add(parsingClone( node.childNodes(3), list, true, 0, true ));
226                 firstClone = false;
227             }
228             //Remove the unnecessary 'label' attribute
229             node.childNodes( 3 ).removeAttr("label");
230             // If the minimum cardinality = 0
231             if ( node.childNodes(3).hasClass("delete") ) {
232                 //Remove the first clone
233                 node.childNodes(2).remove();
234                 node.childNodes(2).remove();
235                 i = 3;
236             }
237         }
238     }
239     // Retrieve the children of the node
240     List<Node> nodes = node.childNodes();
241     // Create and add the children nodes to the current json object
242     JSONArray content = null;
243     if ( nodes.size() > 0 ) {
244         content = new JSONArray();
245     }
246     // Apply the parsing to each child node not yet visited
247     for ( int j = 0 ; j < nodes.size() ; j++ ) {
248         if ( j != i || i != 3 ) {
249             content.add(parsingClone( nodes.get(j), list, false, indice,
                firstClone ));
250         }
251     }
252     obj.put("content", content);
253
254     // If the node is a TextNode
255     } else {
256         // Create a json object representing a TextNode
257         obj.put("kind", "text");
258         obj.put("champ", ((TextNode)node).text());
259     }
260     return obj;
261 }
262
263 /**
264  * Convert the attributes contained in the html node in json format
265  * and add them to the current object json
266  * @param obj A JSONObject representing the current json object to which we
267             add the attributes
268  * @param attr A List of Attribute that must be handled

```

```

269      * @param newClone      A boolean indicating if it begins to generate a new clone or
                             not
270      * @param indice        An integer indicating the number of parameters in the first id
                             of the clonable content
271      * @return int          An integer indicating the number of parameters in the first
                             id of the clonable content
272      */
273      private static int convertAndAddAttributes(JSONObject obj, List<Attribute> attrs,
          boolean newClone, int indice) {
274          //For each attribute
275          for ( int i = 0; i < attrs.size(); i++ ) {
276              //If the attribute is the ID
277              if ( attrs.get(i).getKey().equals("id") ) {
278                  //If we begin to generate a new clone
279                  if ( newClone ) {
280                      //Split the id
281                      String str[] = attrs.get(i).getValue().split("-");
282                      String shortId = str[0];
283                      //Removes all numbers with the following pattern ( shortId: name(-name)*
                      // )
284                      for( int j = 1; j < str.length-1; j++ ){
285                          if ( j % 2 == 0 ) {
286                              shortId = shortId + "-" + str[j];
287                          }
288                      }
289                      obj.put("id", shortId );
290                      // initialize the number of parameters
291                      indice = str.length/2;
292                  } else {
293                      //If it's not a new clone, retrieve the extension of the first id because
                      // all id have the following pattern: firID+extension
294                      String strTemp[] = attrs.get(i).getValue().split("-[0-9]+", indice+1);
295                      String smallId = strTemp[indice];
296
297                      obj.put("id", smallId );
298                  }
299              }
300              //For all other attributes except 'delete' attribute
301              else if ( ! attrs.get(i).getKey().equals("delete") ) {
302                  //Parameterize and add the attribute to the current object json
303                  obj.put(attrs.get(i).getKey(), adaptAttribute( attrs.get(i).getValue(),
                      indice ) );
304              }
305          }
306          return indice;
307      }
308  }
309
310  /**
311   * Adapt an attribute by replacing the numbers with parameters
312   * @param value      A String representing an attribute before the adaptation
313   * @param indice      An integer indicating the number of parameters in the first id
                       of the clonable content

```

```

314      * @return String          A String representing an attribute after the adaptation with
                                   parameters
315      */
316      private static String addaptAttribute(String value, int indice) {
317
318          String strCase[];
319          // Split the initial attribute
320          String strTemp[] = value.split ("-[0-9]+-", indice );
321          String newVal = strTemp[0];
322          //Reconstruct the attribute with parameters in place of numbers
323          for ( int i = 1; i < strTemp.length; i++ ) {
324              if ( i < strTemp.length-1 ) {
325                  newVal = newVal + "-#-" + strTemp[i];
326              } else {
327                  strCase = strTemp[i].split ("-[0-9]+", 2);
328                  newVal = newVal + "-#-" + strCase[0]+"-#" + strCase[1];
329              }
330          }
331          return newVal;
332      }
333  }
334  }

```

Part V

BIBLIOGRAPHY

BIBLIOGRAPHY

- [Jet, 2007] (2007). Jet. Last consulted: February 2014.
- [TEF, 2013] (2013). Textual Editing Framework (TEF). Last consulted: October 2013.
- [Abbasi, 2014] Abbasi, E. K. (2014). *Reverse Engineering Web Configurators*. PhD thesis, University of Namur, PReCISE Research Centre.
- [Abbasi et al., 2014] Abbasi, E. K., Acher, M., Heymans, P., and Cleve, A. (2014). Reverse engineering web configurators. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'14)*, Antwerp, Belgique. IEEE.
- [Abbasi et al., 2013] Abbasi, E. K., Hubaux, A., Acher, M., Boucher, Q., and Heymans, P. (2013). The anatomy of a sales configurator: An empirical study of 111 cases. In Salinesi, C., Norrie, M. C., and Pastor, O., editors, *Proceedings of the 25th International Conference on Advanced Information Systems Engineering (CAiSE'13)*, volume 7908, pages 162–177. Springer.
- [Abele et al., 2010] Abele, A., Papadopoulos, Y., Servat, D., Törnngren, M., and Weber, M. (2010). The CVM framework – a prototype tool for compositional variability management. In [Benavides et al., 2010b], pages 101–106.
- [Acher et al., 2012] Acher, M., Collet, P., Lahire, P., and France, R. (2012). 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. to appear.
- [Acher et al., 2013] Acher, M., Collet, P., Lahire, P., and France, R. B. (2013). FAMILIAR: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6):657–681.
- [Ali et al., 2002] Ali, M., Pérez-Quinones, M. A., Abrams, M., and Shell, E. (2002). Building multi-platform user interfaces with UIML. In Kolski, C. and Vanderdonckt, J., editors, *Computer-Aided Design of User Interfaces III*, pages 255–266. Springer Netherlands.
- [Antkiewicz and Czarnecki, 2004] Antkiewicz, M. and Czarnecki, K. (2004). FeaturePlugin: Feature modeling plug-in for eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*.

- [ANTLR, 2013] ANTLR (2013). ANTLR. Last consulted: October 2013.
- [Bak et al., 2010] Bak, K., Czarnecki, K., and Wasowski, A. (2010). Feature and meta-models in Clafer: Mixed, specialized, and coupled. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10)*, pages 102–122.
- [Batory and Geraci, 1996] Batory, D. and Geraci, B. J. (1996). Validating component compositions in software system generators. In *Proceedings 4th International Conference on Software Reuse (ICSR'96)*, pages 72–81.
- [Batory, 2005] Batory, D. S. (2005). Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC'05)*, pages 7–20.
- [Benavides et al., 2010a] Benavides, D., Batory, D. S., and Grünbacher, P., editors (2010a). *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria*, volume 37 of ICB-Research Report. Universität Duisburg-Essen.
- [Benavides et al., 2005] Benavides, D., Martín-Arroyo, P. T., and Cortés, A. R. (2005). Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05)*, pages 491–503.
- [Benavides et al., 2010b] Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010b). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636.
- [Benavides et al., 2007] Benavides, D., Segura, S., Trinidad, P., and Cortés, A. R. (2007). 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)*, pages 129–134.
- [Beuche, 2008] Beuche, D. (2008). Modeling and building software product lines with pure: :variants. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, page 358, Washington, DC, USA. IEEE Computer Society.
- [Blouin et al., 2011] Blouin, A., Morin, B., Nain, G., Beaudoux, O., Albers, P., and Jézéquel, J.-M. (2011). 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)*, pages 85–94. ACM.

- [Blumendorf et al., 2010] Blumendorf, M., Lehmann, G., and Albayrak, S. (2010). 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)*, pages 9–18. ACM.
- [Bonita, 2014] Bonita (2014). <http://www.bonitasoft.com>.
- [Botterweck et al., 2009] Botterweck, G., Janota, M., and Schneeweiss, D. (2009). A design of a configurable feature model configurator. In *Proceedings of the 3rd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'09)*, pages 165–168.
- [Boucher et al., 2012a] Boucher, Q., Abbasi, E. K., Hubaux, A., Perrouin, G., Acher, M., and Heymans, P. (2012a). Towards more reliable configurators: A re-engineering perspective. In *Proceedings of the 3rd Product Line Approaches in Software Engineering (PLEASE'12)*, pages 29–32.
- [Boucher et al., 2010] Boucher, Q., Classen, A., Faber, P., and Heymans, P. (2010). Introducing TVL, a text-based feature modelling language. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, pages 159–162. Universität Duisburg-Essen.
- [Boucher et al., 2012b] Boucher, Q., Flamand, J., Thunissen, M., and Deprez, J.-C. (2012b). Vers des workflows configurables et standardisés : Premières expériences. Actes de la Journée Lignes de Produits (JLDP'12).
- [Boucher et al., 2012c] Boucher, Q., Perrouin, G., Deprez, J.-C., and Heymans, P. (2012c). Towards configurable ISO 29110-compliant software development processes for very small entities. In *Proceedings of the 19th European System, Software & Service Process Improvement & Innovation Conference (EUROSPI'12)*, pages 169–180. Springer.
- [Boucher et al., 2012d] Boucher, Q., Perrouin, G., and Heymans, P. (2012d). Deriving configuration interfaces from feature models: A vision paper. In *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'12)*, pages 37–44. ACM.
- [Burbeck, 1992] Burbeck, S. (1992). Applications programming in smalltalk-80: How to use model-view-controller (MVC). <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>.
- [Calvary et al., 2003] Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. (2003). A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15:289–308.

- [Chen et al., 2009] Chen, L., Babar, M. A., and Ali, N. (2009). Variability management in software product lines: A systematic review. In *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, pages 81–90.
- [Classen et al., 2011] Classen, A., Boucher, Q., and Heymans, P. (2011). A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76:1130–1143.
- [Classen et al., 2008] Classen, A., Heymans, P., and Schobbens, P.-Y. (2008). What's in a feature: A requirements engineering perspective. In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08)*, pages 16–30.
- [Coutaz, 2010] Coutaz, J. (2010). User interface plasticity: Model driven engineering to the limit! In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'10)*, pages 1–8. ACM.
- [Cyledge, 2013] Cyledge (2013). Cyledge Configurator Database. Last consulted: August 2013.
- [Czarnecki, 2010] Czarnecki, K. (2010). Variability modeling: State of the art and future directions (keynote). In [Benavides et al., 2010b], page 11.
- [Czarnecki and Eisenecker, 2000] Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.
- [Czarnecki and Helsen, 2003] Czarnecki, K. and Helsen, S. (2003). Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*.
- [Czarnecki and Helsen, 2006] Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.
- [Czarnecki et al., 2005] Czarnecki, K., Helsen, S., and Eisenecker, U. W. (2005). Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29.
- [de Sousa and Leite, 2003] de Sousa, L. G. and Leite, J. C. (2003). XICL: A language for the user's interfaces development and its components. In *Proceedings of the Latin American Conference on Human-computer Interaction (CLIHC'03)*, CLIHC '03, pages 191–200, New York, NY, USA. ACM.
- [Devboost, 2013] Devboost (2013). EMFText. Last consulted: October 2013.

- [Eclipse, 2013] Eclipse (2013). Eclipse Modeling Project. Last consulted: October 2013.
- [Efftinge and Zarnekow, 2010] Efftinge, S. and Zarnekow, S. (2010). Goodbye XML - befreiungsakt mit Xtext. <http://jaxenter.de/artikel/Goodbye-XML-Befreiungsakt-mit-Xtext-0>.
- [Eisenstein et al., 2001] Eisenstein, J., Vanderdonckt, J., and Puerta, A. (2001). 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)*, IUI '01, pages 69–76, New York, NY, USA. ACM.
- [Fowler, 2005] Fowler, M. (2005). Language workbenches: The killer-app for domain specific languages?
- [Gabillon et al., 2013] Gabillon, Y., Biri, N., and Otjacques, B. (2013). 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)*, pages 74–81. ACM.
- [Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Boston, MA, USA.
- [García et al., 2009] García, J. G., González-Calleros, J. M., Vanderdonckt, J., and Arteaga, J. M. (2009). A theoretical survey of user interface description languages: Preliminary results. In *Proceedings of the 2009 Latin American Web Congress (La-web 2009)*, pages 36–43.
- [Gomaa et al., 2005] Gomaa, M., Salah, A., and Rahman, S. (2005). Towards a better model based user interface development environment : A comprehensive survey. In *Proceedings of the 38th Midwest Instruction and Computing Symposium (MICS'05)*.
- [Gottschalk et al., 2008] Gottschalk, F., Aalst, W. V. D., Jansen-Vullers, M. H., and Rosa, M. L. (2008). Configurable workflow models. *International Journal of Cooperative Information Systems*, 17(2):177–221.
- [Grechanik et al., 2004] Grechanik, M., Batory, D. S., and Perry, D. E. (2004). Design of large-scale polylingual systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 357–366.
- [Green, 1989] Green, T. (1989). Cognitive dimensions of notations. In Sutcliffe, A. and Macaulay, L., editors, *People and Computers V*, pages 443–460. Cambridge University Press.

- [Greenfield and Short, 2004] Greenfield, J. and Short, K. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, Indianapolis, IN.
- [Griss et al., 1998] Griss, M. L., Favaro, J., and Alessandro, M. d. (1998). Integrating feature modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse (ICSR'98)*, pages 76–85.
- [Helms et al., 2009] Helms, J., Schaefer, R., Luyten, K., Vermeulen, J., Abrams, M., Coyette, A., and Vanderdonckt, J. (2009). Human-centered engineering of interactive systems with the user interface markup language. In *Human-Centered Software Engineering*, pages 139–171. Springer.
- [Heymans et al., 2012] Heymans, P., Boucher, Q., Classen, A., Bourdoux, A., and Demonceau, L. (2012). A code tagging approach to software product line development: An application to satellite communication libraries. *International Journal on Software Tools for Technology Transfer*.
- [Holtz and Rasdorf, 1988] Holtz, N. and Rasdorf, W. (1988). An evaluation of programming languages and language features for engineering software development. *Engineering with Computers*, 3:183–199.
- [Hubaux, 2012] Hubaux, A. (2012). *Feature-based Configuration: Collaborative, Dependable, and Controlled*. PhD thesis, University of Namur, PRECISE Research Centre.
- [Hubaux et al., 2010a] Hubaux, A., Boucher, Q., Hartmann, H., Michel, R., and Heymans, P. (2010a). Evaluating a textual feature modelling language: Four industrial case studies. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10)*, pages 337–356.
- [Hubaux et al., 2009] Hubaux, A., Classen, A., and Heymans, P. (2009). Formal modelling of feature configuration workflows. In *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, pages 221–230.
- [Hubaux et al., 2010b] Hubaux, A., Classen, A., Mendonca, M., and Heymans, P. (2010b). A preliminary review on the application of feature diagrams in practice. In [Benavides et al., 2010b], pages 53–59.
- [Hubaux et al., 2011] Hubaux, A., Heymans, P., Schobbens, P.-Y., Deridder, D., and Abbasi, E. K. (2011). Supporting multiple perspectives in feature-based configuration. *Software and Systems Modeling*, pages 1–23.
- [Hubaux et al., 2013] Hubaux, A., Heymans, P., Schobbens, P.-Y., Deridder, D., and Abbasi, E. K. (2013). Supporting multiple perspectives in feature-based configuration. *Software and System Modeling*, 12(3):641–663.

- [ISO/IEC, 1996] ISO/IEC (1996). ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF.
- [ISO/IEC, 2010] ISO/IEC (2010). ISO/IEC 29110 – Lifecycle Profiles for Very Small Entities (VSEs).
- [Jamda Project, 2014] Jamda Project (2014). Jamda. Last consulted: February 2014.
- [Jin et al., 2014] Jin, D., Qu, X., Cohen, M. B., and Robinson, B. (2014). Configurations everywhere: Implications for testing and debugging in practice. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, ICSE Companion 2014, pages 215–224, New York, NY, USA. ACM.
- [Johnson and Parekh, 2003] Johnson, P. D. and Parekh, J. (2003). Multiple device markup language: A rule approach. Technical report, DePaul University.
- [Jouault et al., 2008] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39.
- [Jouault et al., 2006] Jouault, F., Bézivin, J., and Kurtev, I. (2006). TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative Programming and Component Engineering (GPCE'06)*, GPCE '06, pages 249–254, New York, NY, USA. ACM.
- [jQuery, 2014] JQuery (2014). <http://jquery.com/>. Last consulted: January 2014.
- [Kang et al., 1990] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, SEI, CMU.
- [Kang et al., 1998] Kang, K. C., Kim, S., Lee, J., Kim, K., Kim, G. J., and Shin, E. (1998). FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168.
- [Kästner et al., 2009] Kästner, C., Thüm, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., and Apel, S. (2009). FeatureIDE: A tool framework for feature-oriented software development. In *Proceedings of the 31th International Conference on Software Engineering (ICSE'09)*, pages 311–320.
- [Kent, 2002] Kent, S. (2002). Model driven engineering. In Butler, M., Petre, L., and Sere, K., editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer.

- [Kim and Foley, 1993] Kim, W. C. and Foley, J. D. (1993). Providing high-level control and expert assistance in the user interface presentation design. In *Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 430–437, New York, NY, USA. ACM.
- [Kleppe et al., 2003] Kleppe, A. G., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [Knuth, 1971] Knuth, D. E. (1971). Semantics of context-free languages. *Mathematical Systems Theory*, 5(1):95–96.
- [Kost, 2006] Kost, S. (2006). *Dynamically generated multi-modal application interfaces*. PhD thesis, Dresden University of Technology. <http://d-nb.info/980375045>.
- [La Rosa et al., 2007] La Rosa, M., Lux, J., Seidel, S., Dumas, M., and ter Hofstede, A. (2007). Questionnaire-driven configuration of reference process models. In *Proceedings of the 19th International Conference on Advanced Information Systems Engineering (CAiSE'07)*, pages 424–438. Springer.
- [Limbourg et al., 2005] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and López-Jaquero, V. (2005). UsiXML: A language supporting multi-path development of user interfaces. In Bastide, R., Palanque, P., and Roth, J., editors, *Engineering Human Computer Interaction and Interactive Systems*, volume 3425 of *Lecture Notes in Computer Science*, pages 200–220. Springer Berlin Heidelberg.
- [Mendonca, 2009] Mendonca, M. (2009). *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo.
- [Mendonca et al., 2009] Mendonca, M., Branco, M., and Cowan, D. (2009). S.P.L.O.T.: Software product lines online tools. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA'09)*.
- [Mens and Gorp, 2006] Mens, T. and Gorp, P. V. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142.
- [Merkle, 2010] Merkle, B. (2010). Textual modeling tools: overview and comparison of language workbenches. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA'10)*, pages 139–148. ACM.

- [Michel et al., 2011] Michel, R., Classen, A., Hubaux, A., and Boucher, Q. (2011). A formal semantics for feature cardinalities in feature diagrams. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'11)*, pages 82–89.
- [Moody, 2009] Moody, D. L. (2009). The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35:756–779.
- [Mueller et al., 2004] Mueller, W., Schaefer, R., and Bleul, S. (2004). Interactive multimodal user interfaces for mobile devices. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*, HICSS '04, pages 90286.1–, Washington, DC, USA. IEEE Computer Society.
- [Müller et al., 2001] Müller, A., Forbrig, P., and Cap, C. H. (2001). 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)*, DSV-IS '01, pages 16–27, London, UK, UK. Springer-Verlag.
- [Myers et al., 2000] Myers, B. A., Hudson, S. E., and Pausch, R. F. (2000). Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction*, 7:3–28.
- [Nichols and Faulring, 2005] Nichols, J. and Faulring, A. (2005). Automatic interface generation and future user interface tools. In *Proceedings of the ACM CHI 2005 Workshop: The Future of User Interface Design Tools*.
- [Nichols et al., 2002] Nichols, J., Myers, B. A., Higgins, M., Hughes, J., Harris, T. K., Rosenfeld, R., and Pignol, M. (2002). Generating remote control interfaces for complex appliances. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology (UIST'02)*, pages 161–170.
- [NXP Semiconductors, 2010] NXP Semiconductors (2010). <http://www.nxp.com/>.
- [Obeo, 2014] Obeo (2014). Acceleo. <http://www.eclipse.org/acceleo/>. Last consulted: February 2014.
- [Object Management Group, 2012] Object Management Group (2012). OCL 2.3.1 Specification.
- [Océ Software Laboratories, 2010] Océ Software Laboratories (2010). <http://www.osl.be/>.

- [OMG, 2008] OMG (2008). MOF Model to Text Transformation Language (MOFM2T), 1.0.
- [OMG, 2011] OMG (2011). Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification, Version 1.1.
- [Paterno' et al., 2009] Paterno', F., Santoro, C., and Spano, L. D. (2009). MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction*, 16(4):19:1–19:30.
- [Picard et al., 2003] Picard, E., Fierstone, J., Pinna-Déry, A.-M., and Riveill, M. (2003). Atelier de composition d'IHM et évaluation du modèle de composants. Technical Report Livrable 3, Réseau National des Technologies Logicielles.
- [Pleuss et al., 2010] Pleuss, A., Botterweck, G., and Dhungana, D. (2010). 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)*, pages 69–76.
- [Pleuss et al., 2011] Pleuss, A., Rabiser, R., and Botterweck, G. (2011). Visualization techniques for application in interactive product configuration. In *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC'11)*, page 22.
- [PloneGov, 2010] PloneGov (2010). <http://www.plonegov.org/>.
- [Pohl et al., 2005] Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.
- [Potel, 1996] Potel, M. (1996). MVP: Model-View-Presenter the taligent programming model for c++ and java. *Taligent Inc.*
- [Pratt, 1984] Pratt, T. W. (1984). *Programming Languages : Design and Implementation*. Prentice Hall, second edition edition. 604 pages.
- [Puerta and Eisenstein, 2002] Puerta, A. and Eisenstein, J. (2002). XML: A common representation for interaction data. In *Proceedings of the 7th International Conference on Intelligent User Interfaces (IUI'02)*, IUI '02, pages 214–215, New York, NY, USA. ACM.
- [Puerta and Eisenstein, 2003] Puerta, A. and Eisenstein, J. (2003). Developing a multiple user interface representation framework for industry. In *Multiple User Interfaces: Engineering and Application Framework*, pages 119–148. John Wiley and Sons.

- [pure-systems GmbH, 2006] pure-systems GmbH (2006). Variant management with pure::variants. <http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf>. Technical White Paper.
- [Quinton et al., 2011] Quinton, C., Parra, C. A., Mosser, S., and Duchien, L. (2011). Using multiple feature models to design applications for mobile phones. In *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC'11)*, page 23.
- [QVT Declarative, 2014] QVT Declarative (2014). <http://projects.eclipse.org/projects/modeling.mmt.qvtd>. Last consulted: February 2014.
- [QVT Operational, 2014] QVT Operational (2014). <http://projects.eclipse.org/projects/modeling.mmt.qvt-oml>. Last consulted February 2014.
- [Reenskaug, 1979a] Reenskaug, T. (1979a). Models-Views-Controllers. <http://heim.ifi.uio.no/trygver/1979/mvc-2/1979-12-MVC.pdf>.
- [Reenskaug, 1979b] Reenskaug, T. (1979b). Thing-Model-View-Editor: An example from a planning system. http://folk.uio.no/trygver/2007/MVC_Originals.pdf.
- [Reiser, 2009] Reiser, M.-O. (2009). Core concepts of the compositional variability management framework (cvm). Technical report, Technische Universität Berlin.
- [Rexel, 2014] Rexel (2014). <http://www.rexel.com>.
- [Sangwan and Hian, 2004] Sangwan, S. and Hian, C. K. (2004). User-centered design: marketing implications from initial experience in technology supported products. In Press, I. C. S., editor, *Proceedings of the ASEM Engineering Management Conference*, volume 3, pages 1042– 046.
- [Schlee and Vanderdonckt, 2004] Schlee, M. and Vanderdonckt, J. (2004). Generative programming of guis. In *Proceedings of the 7th International Working Conference on Advanced Visual Interfaces (AVI'04)*, pages 403–406. ACM.
- [Schobbens et al., 2006] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., and Bontemps, Y. (2006). Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479.
- [Souchon and Vanderdonckt, 2003] Souchon, N. and Vanderdonckt, J. (2003). A review of XML-compliant user interface description languages. In Jorge,

- J., Jardim Nunes, N., and Falcão e Cunha, J., editors, *Interactive Systems. Design, Specification, and Verification*, volume 2844 of *Lecture Notes in Computer Science*, pages 377–391. Springer Berlin Heidelberg.
- [Tarr et al., 1999] Tarr, P., Ossher, H., Harrison, W., and Sutton, S. M. J. (1999). N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 107–119.
- [ter Hofstede et al., 2010] ter Hofstede, A. H. M., van der Aalst, W. M. P., Adams, M., and Russell, N. (2010). *Modern Business Process Automation - YAWL and its Support Environment*. Springer.
- [Tun et al., 2009] Tun, T. T., Boucher, Q., Classen, A., Hubaux, A., and Heymans, P. (2009). Relating requirements and feature configurations: A systematic approach. In *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, pages 201–210.
- [UsiXML Consortium, 2012] UsiXML Consortium (2012). User Interface eXtensible Markup Language (UsiXML). Submitted to the W3C Model-Based UI Working Group.
- [van Deursen and Klint, 2002] van Deursen, A. and Klint, P. (2002). Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18.
- [Vanbrabant, 2008] Vanbrabant, R. (2008). *Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress)*. APress.
- [Virage Logic, 2010] Virage Logic (2010). <http://www.viragelogic.com/>.
- [W3C, 2008] W3C (2008). Cascading Style Sheets. <http://www.w3.org/TR/REC-CSS1/>. Last consulted: October 2013.
- [W3C, 2009] W3C (2009). XForms 1.1.
- [W3C, 2010a] W3C (2010a). Model-based UI XG final report. <http://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui-20100504/>.
- [W3C, 2010b] W3C (2010b). XML Path Language (XPath) 2.0.
- [W3C, 2013] W3C (2013). Accessibility evaluation resources. <http://www.w3.org/WAI/eval/>. Last consulted: January 2014.
- [W3C, 2014a] W3C (2014a). HTML5. <http://www.w3.org/TR/html5/>. Last consulted: December 2013.

- [W3C, 2014b] W3C (2014b). MBUI - abstract user interface models. <http://www.w3.org/TR/abstract-ui/>.
- [White et al., 2014] White, J., Galindo, J. A., Saxena, T., Dougherty, B., Benavides, D., and Schmidt, D. C. (2014). Evolving feature model configurations in software product lines. *Journal of Systems and Software*, 87(0):119 – 136.
- [Xtext, 2013] Xtext (2013). <http://www.eclipse.org/Xtext/>. Last consulted: October 2013.
- [Yin, 2002] Yin, R. K. (2002). *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods*. Sage Publications, Inc, 3rd edition.