Chapter 1
Engineering Configuration Graphical User Interfaces from Variability Models

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

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

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

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

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

G. Perrouin • J.-M. Davril • P. Heymans
PReCISE Research Centre, University of Namur, Rue Grandgagnage 21, 5000, Namur, Belgium
e-mail: gilles.perrouin@unamur.be; jean-marc.davril@unamur.be; patrick.heymans@unamur.be

1.1 Introduction

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

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

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

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

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

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

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

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

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

1.2 Background

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

1.2.1 Feature Modelling

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

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

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

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

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

Table 1.1 Existing textual variability modelling languages

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

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

1.2.2 User Interface Modelling and Generation

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

1.2.2.1 User Interface Description Languages

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

Most approaches propose a partially automated process which uses extra information about the UI stored in models. They are all grouped under the *Model-based User Interface Development* (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"* [42]. Each MBUIDE defines its own set of models to describe the interface. The different MBUIDEs and the associated models have been surveyed by Gomaa et al. [42] and the W3C [87]. Here, we give a summary of User Interface Description Languages (UIDLs) used in MBUID. XML-based UIDLs have also been surveyed by several authors [41, 81]. Such languages can be used to represent the generated GUIs at a more "abstract" level. They are grouped in four categories.

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

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

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

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

[2]See https://www.oasis-open.org/

Table 1.2 Existing user interface description languages

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

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

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

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

1.2.2.2 Feature Models and GUIs

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

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

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

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

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

1.3 The MVP Configurator Pattern

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

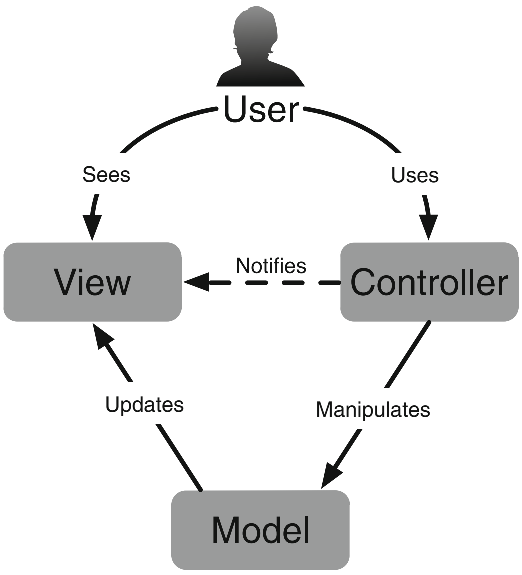


Fig. 1.1 Model-view-controller architecture

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

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

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

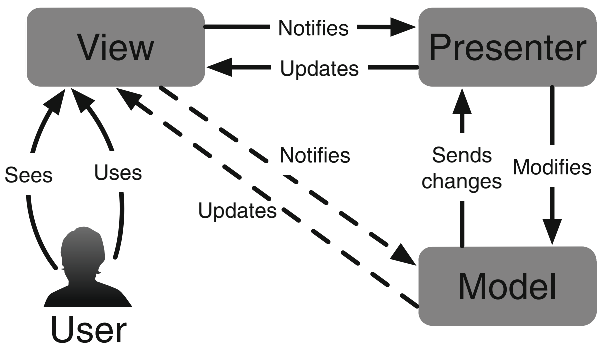


Fig. 1.2 Model-view-presenter architecture

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

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

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

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

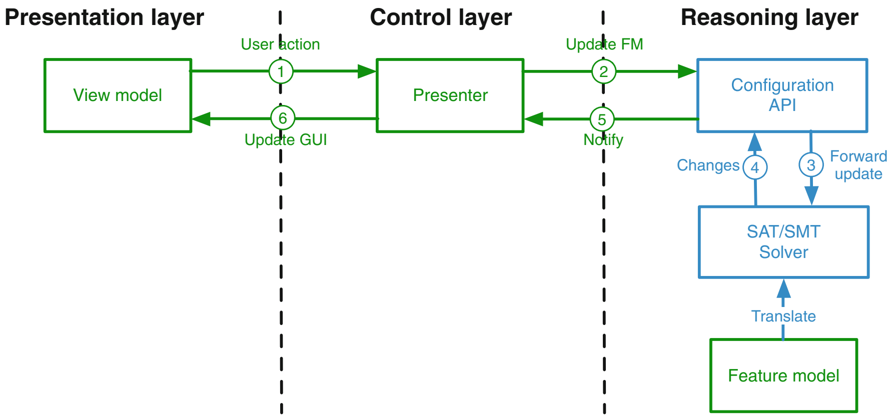


Fig. 1.3 An MVP architecture for configurators

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

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

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

1.4 From Feature Models to MVP Configurators

1.4.1 Illustration

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

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

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

1.4.2 Variability Modelling

1.4.2.1 General Principles and Language

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

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

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

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

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

`TVL` offers two mechanisms that can help engineers structure large models. The first is the `include` statement, which takes as parameter a file path. As expected, an *include* statement will include the contents of the referenced file at this point. Includes are in fact preprocessing directives and do not have any meaning beyond the fact that they are replaced by the referenced file. Modellers can thus structure the FM according to their preferences. The second structuring mechanism, hinted at before, is that features can be defined in one place and be extended later in the code. Basically, a feature block may be repeated to add constraints and attributes to the feature. These mechanisms allow modellers to organise the FM according to their preferences and can be used to implement separation of concerns [83]. This way the engineer can specify the structure of the FM upfront, without detailing the features. Feature attributes and constraints can be specified in the second part of the file, or in other files using the *include* statement. The only restriction is that the hierarchy of a feature can only be defined at one place (i.e., the *group* keyword can only be used once for each feature).

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

`TVL` $_2$

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

A string attribute is defined using the *string* keyword. Similarly to other attribute types, an ID is then given to the attribute. The naming convention is the same, the attribute ID has to start with a lower case letter. For example, "`string myString`" is a valid attribute declaration. It is also possible to define string constants in `TVL`$_2$.

In the original `TVL` syntax, each feature can be configured (at most) once. Like most existing languages, ours lacks a construct that allows to duplicate a sub-tree of the FM to configure a product. `TVL`$_2$ now supports so-called feature cardinalities. Their semantics is defined elsewhere [61] and will not be addressed here. Syntactically, feature cardinalities are represented in a similar way to group cardinalities, with bounds between brackets. The cardinality directly follows the name of a feature. If it is not defined, the [1..1] cardinality is assumed. Furthermore, the root feature cannot have a cardinality, i.e., it still has to be unique. Bounds can be either an integer value or a constant, or the asterisk character. Here, the asterisk character means that the number of feature instances is unlimited.

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

1.4.2.2 Addressing Plan Example

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

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

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

```

  // Use types
  int useTypes;
  int futureUseTypes;

  // Locations
  int locations;
  int futureLocations;

  group someof {
    Subnet [0..*] {
      shared UseType useType;
      shared Location location;
      string subnetName;
      string subnetPrefix;
    },
    UseType [0..*] {
      string useTypeName;
      string useTypePrefix;
    },
    Location [0..*] {
      string locationName;
      string locationPrefix;
    },
    Host [0..*] {
      string hostName;
      string loopback;
      group someof {
        Interface [0..*] {
          string index;
          string macAddress;
          real delay;
          shared Subnet subnet;
          string ipAddress;

          group allof {
            ConnectedInterface [0..*] {
              shared Interface connectedInterface;
            }
          }
        },
        RoutingTableEntry [0..*] {
          string destination;
          int metric;
          shared Interface sendingInterface;
          shared Interface nextHop;
        }
      }
    }
  }
}
```

1.4.2.3 Widget Selection

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

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

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

1.4.3 View Definition

1.4.3.1 General Principles and Language

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

Table 1.3 Graphical widgets mappings

Category	Construct	Widget	HTML example
Groups	*and* *(optional features)*	Check button	☐
		List box	True ⇕
		Radio box	● True ○ False
	or	List box	▭
		Check box	☐ ☐
	xor	List box	⇕
		Radio box	● ○
	cardinality	Check box	☐ ☐
Attribute types	*integer*	Text box	0 ⇕
		Slider	5
	real	Text box	0,0 ⇕
		Slider	4.9
	Boolean	Check button	☐
		List box	True ⇕
		Radio box	● True ○ False
	enumeration	List box	⇕
		Radio box	● ○
Features & Attributes	*feature/attribute*	Label	Feature label
		Image	

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

To break out of the FM hierarchy, we propose to use views on them. Views are *“a simplified version of an FM that has been tailored for a specific stakeholder, role, task, or, to generalize, a particular combination of these elements, which we call* *a concern. Views facilitate configuration in that they only focus on those parts of the FM that are relevant for a given concern. Using multiple views is thus a way to achieve separation of concerns in FMs"* [50].

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

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

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

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

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

The first view expression type, full sub-tree, is defined using the asterisk character. A so-called *stop list* can be defined to determine the branches of the FM which are not covered by the view expression. The branch is thus pruned before the stop list element, i.e., it is not included. A stop list is composed of stop elements which are a TVL feature name or its (fully) qualified name preceded by the slash character. A stop list is composed of at least one stop element.

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

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

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

1.4.3.2 Addressing Plan Example

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

Listing 1.2 `TVDL` model for the addressing plan example

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

1.4.4 Widget Selection

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

1.4.5 Beautification

1.4.5.1 General Principles and Language

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

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

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

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

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

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

Global Properties

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

The first property dedicated to features is simply called *feature* and determines how they are rendered in the GUI. Available values are *text* and *image*. Those values speak for themselves. The *optFeature* property determines how optional features have to be rendered. Three values exist, *checkbox*, *listbox*, and *radiogroup*. With a check box, the optional feature is selected if (and only if) it is checked. The 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*. In TVDL, we allow to not cover a group if all its sub-features are covered. As a consequence, the group is not rendered in the configuration GUI. Given that all its sub-features are depicted, we propose to use a selection widget in front of all of them, similarly to optional features. In this way, the user is still able to select group's sub-features and the group cardinality will be verified by the solver (the *presenter* in our architecture). The group is scattered all over the configuration GUI but it is still possible to select its sub-features while sticking to its cardinality.

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

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

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

View Properties

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

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

The other category of view-specific properties is similar to the global properties. Indeed, properties falling in this category will influence the rendering of constructs covered by the view. For this reason, the proposed properties are exactly the same as global ones presented earlier. The 14 properties will not be recalled here for the sake of conciseness. However, we would like to draw the attention to one of them, *view*. As a reminder, this property allows to define the widget corresponding to views. Setting this property will have an influence on the views contained in the view corresponding to the view-specific definition section, not on that view itself. The *view* property thus only makes sense for *grouping views*. In our opinion, all views declared at the same level should be depicted by the same widget. This explains why we did not propose a *widget* property in the first category. However, if needed, this property could be easily added.

Feature Properties

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

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

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

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

Attribute Properties

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

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

1.4.5.2 Addressing Plan Example

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

Listing 1.3 FCSS model for the addressing plan example

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

```
#AddressPlan.networkPrefix {
    label: "Numero client";
}
#AddressPlan.strategy {
    label: "Strategy";
}
#AddressPlan.useTypes {
    label: "Number of use types";
}
#AddressPlan.futureUseTpes {
    label: "Number of future use types for expansion";
}
#AddressPlan.locations {
    label: "Number of locations";
}
#AddressPlan.futureLocations {
    label: "Number of future locations for expansion";
}

Subnet {
    label: "Subnet";
}
#Subnet.useType {
    label: "Use type";
}
#Subnet.location {
    label: "Location";
}
#Subnet.subnetName {
    label: "Name";
}
#Subnet.subnetPrefix {
    label: "Prefix";
}
```

1.4.6 *Putting It All Together*

1.4.6.1 General Principle

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

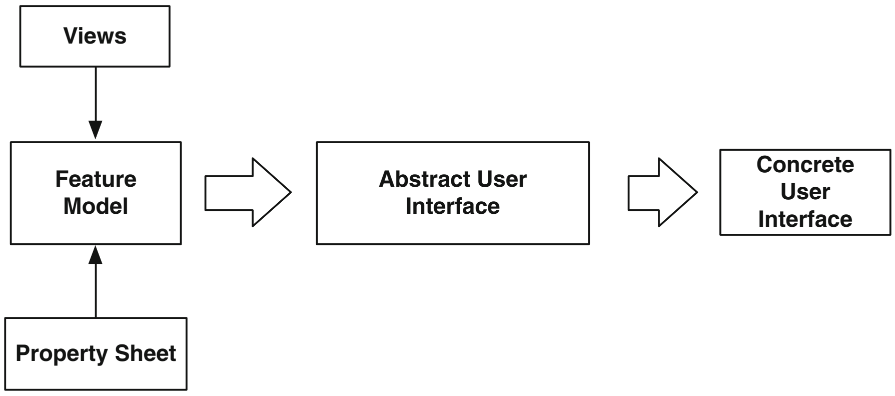


Fig. 1.4 Interface generation process

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

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

1.4.6.2 Addressing Plan Example

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

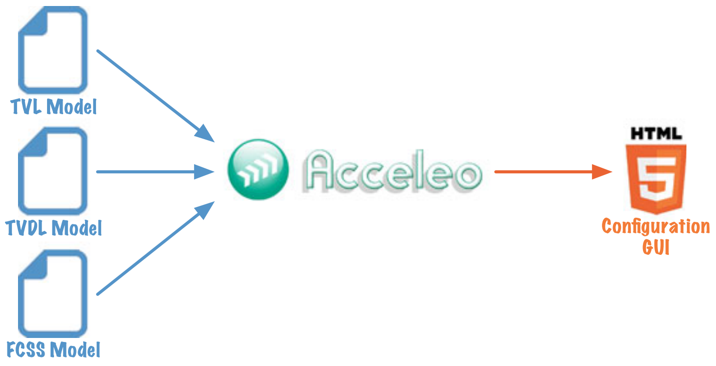
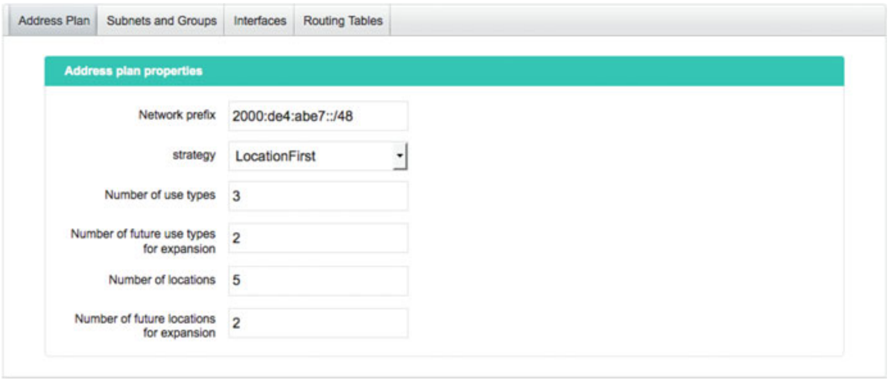


Fig. 1.5 Generation process with *Acceleo*



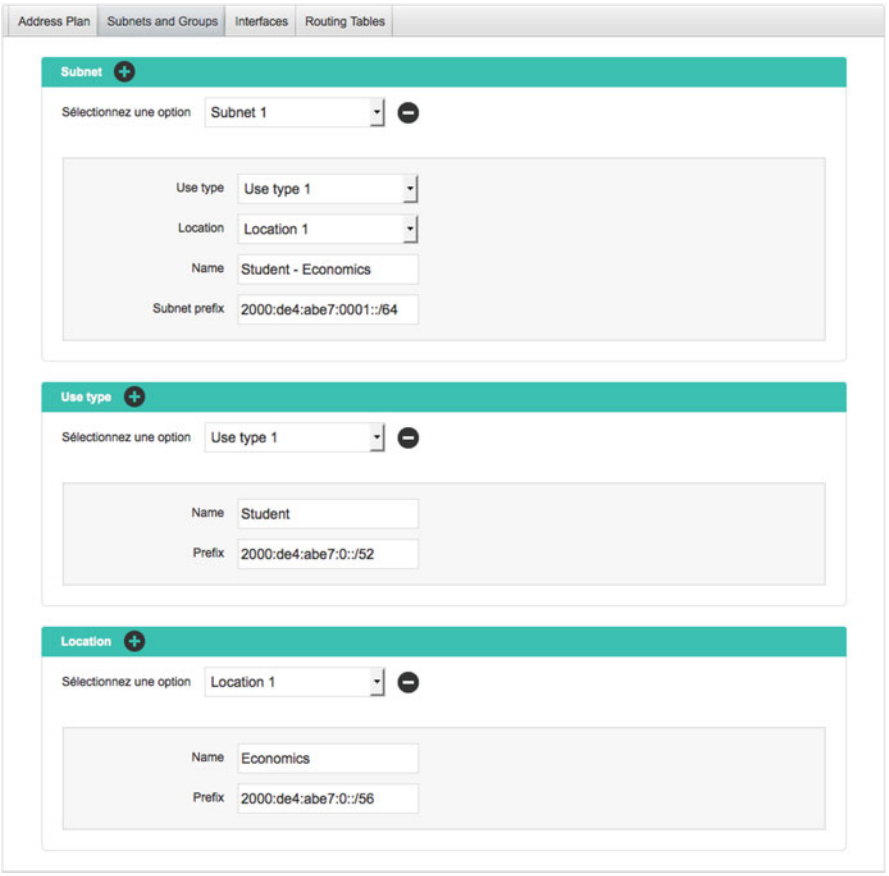
Network prefix	2000:de4:abe7::/48
strategy	LocationFirst
Number of use types	3
Number of future use types for expansion	2
Number of locations	5
Number of future locations for expansion	2

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

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

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

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



Subnet

Sélectionnez une option	Subnet 1
Use type	Use type 1
Location	Location 1
Name	Student - Economics
Subnet prefix	2000:de4:abe7:0001::/64

Use type

Sélectionnez une option	Use type 1
Name	Student
Prefix	2000:de4:abe7:0::/52

Location

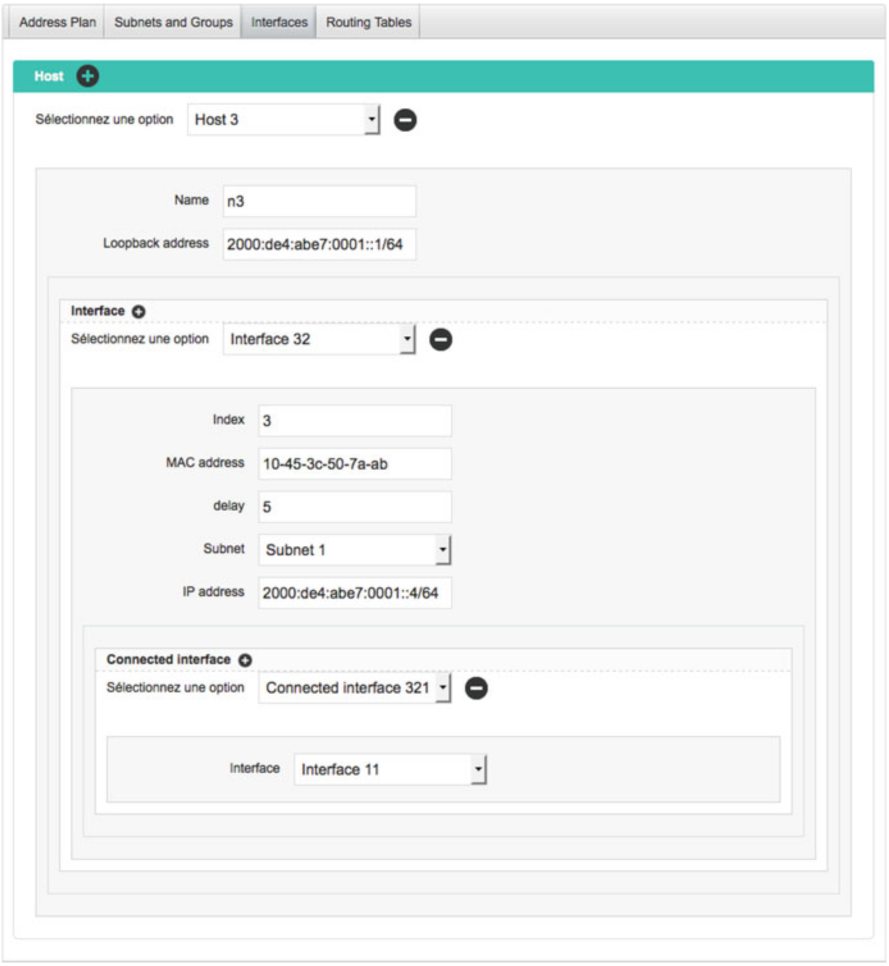
Sélectionnez une option	Location 1
Name	Economics
Prefix	2000:de4:abe7:0::/56

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

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

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

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



Sélectionnez une option: Host 3

Name: n3
Loopback address: 2000:de4:abe7:0001::1/64

Interface
Sélectionnez une option: Interface 32

Index: 3
MAC address: 10-45-3c-50-7a-ab
delay: 5
Subnet: Subnet 1
IP address: 2000:de4:abe7:0001::4/64

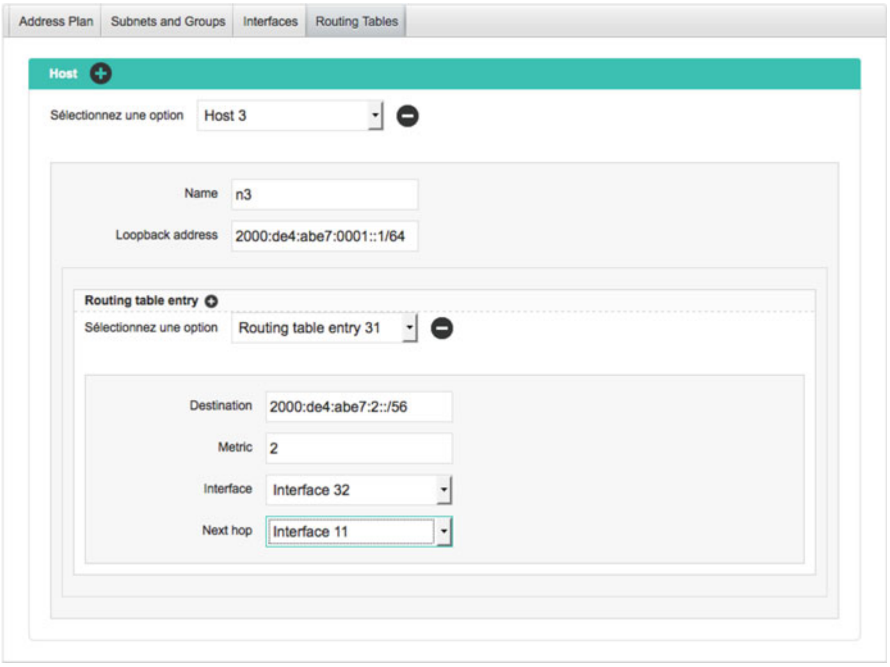
Connected interface
Sélectionnez une option: Connected interface 321

Interface: Interface 11

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

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

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



Sélectionnez une option: Host 3

Name: n3

Loopback address: 2000:de4:abe7:0001::1/64

Routing table entry

Sélectionnez une option: Routing table entry 31

Destination: 2000:de4:abe7:2::/56

Metric: 2

Interface: Interface 32

Next hop: Interface 11

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

1.5 Lessons Learned

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

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

[3]Unfortunately, these cases could not be reported here for confidentiality reasons.

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

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

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

Completeness of TVL. In the biggest TVL model we had to produce so far, we count four duplicable features. The same comment applies to string attributes added in TVL $_2$ and used ten times in the same case study, that is 17,9% of the attributes. Generally speaking, TVL offered the required expressiveness. Shared features also proved relevant, even if they are currently not supported by the underlying solver.

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

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

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

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

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

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

Source of propagations. Initially, the presenter was able to handle values propagated by the solver in a specific way. In the prototype version, they were greyed out in order to prevent user changes. But this approach was rather restrictive with respect to the results sent back by the solver. For example, if a feature is selected, the propagation set contains its parent which will be greyed out in the configuration GUI. While this behaviour respects the semantics of FMs, it is not adapted to GUIs. In such a case, the user would have to deselect all sub-features to unblock the parent one. Instead, it should be possible to set the parent to false with the unassignment of child features as side effect. We identify three categories of propagation sources: cross-tree, hierarchy, and siblings constraints. The first category should trigger the disabling mechanism (e.g., grey out). The second one has been illustrated by the example earlier in this paragraph. Finally, siblings constraints should be handled differently by the presenter depending on the widget representing the group. For example, *xor*-decompositions rendered as a list box or a radio group are automatically handled by the widget, contrarily to those depicted by a set of check boxes. In the future, the solver should return three propagation sets, differently handled by the presenter.

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

1.6 Perspectives

1.6.1 Multiple Targets

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

1.6.2 Ordering Views

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

To describe those behaviours, feature configuration workflows [47] or multi-step SPL configuration [91] could be used. There, the workflow defines the configuration process and each view on the FM is assigned to a task in the workflow. A view is configured when the corresponding workflow task is executed. A feature configuration workflow is thus a combination of views on the FM, workflow and the mapping between them. Up to now, feature configuration workflows focused on distributed configuration among several stakeholders but one might easily adapt them to other purposes like the dynamic behaviour of a GUI in our case.

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

1.6.3 Re-engineering

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

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

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

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

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

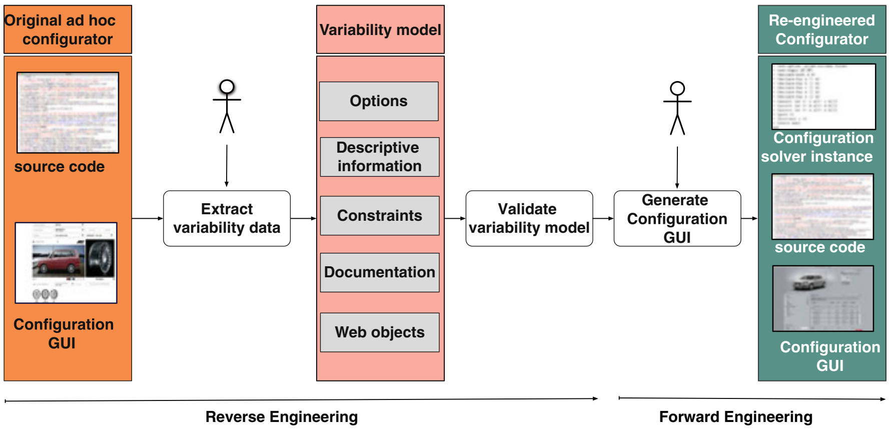


Fig. 1.10 Re-engineering process for configurators

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

1.6.4 Product Selection

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

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

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

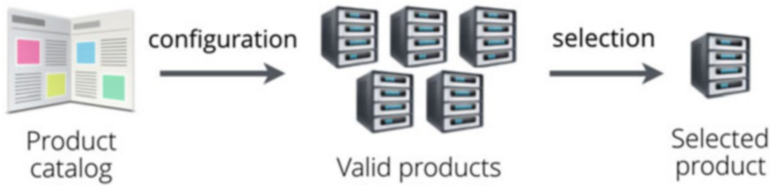


Fig. 1.11 The configuration phase and the selection phase for a catalogue of servers

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

1.6.4.1 Product Comparators

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

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

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

1.6.4.2 Knowledge-Based Recommender Systems

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

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

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

1.6.5 Recommendations

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

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

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

1.6.6 Evalution of Configuration Interfaces

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

1.7 Conclusion

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

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

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

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

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

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

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

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

References

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

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

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

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

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

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