Deriving Configuration Interfaces from Feature Models:
A Vision Paper

Quentin Boucher, Gilles Perrouin
PReCISE Research Centre
Faculty of Computer Science
University of Namur, Belgium
{qbo.gpe}@info.fundp.ac.be

Patrick Heymans
PReCISE Research Centre
Faculty of Computer Science
University of Namur, Belgium
INRIA Lille-Nord Europe
Université Lille 1 – LIFL – CNRS, France
phe@info.fundp.ac.be

ABSTRACT
In software product lines, feature models are the de-facto standard for representing variability as well as for configuring products. Yet, configuration relying on feature models faces two issues: i) it assumes knowledge of the underlying formalism, which may not be true for end users and ii) it does not take advantage of advanced user-interface controls, leading to usability and integration problems with other parts of the user interface. To address these issues, our research focuses on the generation of configuration interfaces based on variability models, both from the visual and behavioral perspectives. We tackle visual issues by generating abstract user-interfaces from feature models. Regarding configuration behavior, in particular the configuration sequence, we plan to use feature configuration workflows, variability-aware models that exhibit similar characteristics as of task, user, discourse and business models found in the in the human-computer interaction community. This paper discusses the main challenges and possible solutions to realize our vision.

Keywords
Software Product Lines, Feature Configuration Workflows, Configuration Interfaces

1. INTRODUCTION
Along with the development of e-commerce, mass customization [29] which was formerly performed by software engineers is now realized by product customers through an adequate configuration interface. These configuration applications have permeated a number of markets such as car manufacturers, clothing or computer hardware. Software products are also configurable, service-based applications being one of the most well-known example. The software product line (SPL) community has addressed the design of such configurators [7] by relying on feature models (FMs).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VuMoS ’12, January 25-27, 2012 Leipzig, Germany
Copyright 2012 ACM 978-1-4503-1058-1 ...$10.00.

Indeed, selecting options of a particular product within a given configurator amounts to perform interactive configuration of a feature model where features correspond to options and decisions propagated throughout the configuration interface enabling or disabling specific options according to constraints.

Therefore, several SPL tools such as SPLIT [30] or Pure::Variants [6] offer configuration facilities based on visual representations of FMs, mostly adopting a “tree-based” representation of the features’ hierarchy. However, there are many ways to graphically represent FMs [35, 14] and these ways should be tailored to the usage needs. For example, a preliminary survey [23] has questioned the suitability of the FODA-like notation for editing FMs in practice, resulting in the definition of a textual feature modelling language [8, 10] targeted at SPL engineers and developers. Industrial feedback was promising [21]. Additional evidence is provided by Pleuss et al. [35] while comparing different graphical representations of FMs; tailoring the representations to the task is an important aspect. In this paper, we focus on the configuration needs of end users who will configure their products through an appropriate interface. Thus, it is interesting to look at the human-computer interaction community to further investigate the problem. Configuration interfaces can be thought as plastic user interfaces [45], which adapt themselves due to interactive configuration and can be deployed on a variety of devices. Therefore, for configuration purposes, plasticity may involve omitting the feature hierarchy or break it into smaller parts [24] either to support a deliberate design or to accommodate hardware constraints. This hence discards configurators that are too rigid with respect to the FM’s hierarchy representation. There are also lessons to be learned from the database community, where the generation of form-based interfaces has been addressed [26].

As a result, our vision combines ideas coming from model-based and data-based graphical user interfaces (GUIs) generation with our previous research feature-based configuration [25, 22]. We sketch in this paper the main elements of this vision as well as related research challenges.

The paper is organized as follows. Section 2 sketches our model-based vision for configuration interfaces generation, illustrated through examples. Research challenges to be solved to realize this vision are discussed in Section 3. Related work is surveyed in Section 4. Finally, Section 5 wraps up the paper and presents some on-going and future developments.
2. APPROACH OVERVIEW

As we mentioned above, direct configuration from the FM may not suit every user need. Therefore, our vision is based on the decoupling of the FM and the configuration user interface (UI) by combining separation of concerns [43] and generative techniques [39]. The base process is sketched in Figure 1.

Our approach relies on the notion of Abstract User Interface (AUI) [9]. According to Coutaz et al. [9], an AUI is “a canonical expression of the rendering of the domain concepts and functions in a way that is independent of the interactors available on the targets”. In other words, an AUI is a language and target platform independent description of the UI, which allows considering mappings from the feature model, configuration process and its interaction with the UI in an unique and reusable manner. This AUI can be directly generated from the FM with the possibility to use Views [24] or Feature Configuration Workflows [22] to tweak configuration interface decomposition and sequencing logic. The layout of the elements composing the UI can be guided by a Property sheet containing beautifying information. Once created, the AUI can be then transformed in a Concrete User Interface (CUI). CUI's look and feel could also be driven by information contained in the Property sheet. Depending on the required sophistication level of the interface, different combinations of views, configuration workflows and property sheets are envisioned, as explained in the subsequent paragraphs.

Thus, this architecture exhibits the following benefits:

- **Seamless integration with existing approaches.** The explicit distinction between variability and interface modelling aspects ease the connection of the approach with state of the art research. For example, at the feature modelling level, we can take advantage of modularity techniques [1, 25, 24, 3] but also automated analyses [5, 30]. Regarding interfaces, the approach can be integrated with existing frameworks and CUI generation tools [9].
- **Inconsistent configuration analysis and evolution.** Since the configuration logic is not implemented directly in the CUI, it is easy to use the analysis tools above to detect conflicting situations. Also, evolution is managed at the abstract level and can be enacted via reapplication of the process.
- **Platform Independence** With the development of different devices, it is not rare that the same configuration interface needs to be deployed on various computing environments (OS, Screen, etc). Thanks to its MDA-like architecture [46], we can reuse configuration logic across languages and platforms. Some parts of the framework can be made even more generic, if we devise transformation patterns such as the ones sketched in [39].

In the following, we detail how the various models can be employed for the generation of UIs of growing complexity.

2.1 Form-like Interfaces

Forms “are a structured means of displaying and collecting information for further processing” [38]. They have become a natural part of a wide variety of applications and websites. In the database domain, generating forms from data models is a common task. There, forms to query or insert information into a database are generated from data models, using their meta-data for example [20]. FMs being interpreted as class models with a containment hierarchy [15], a similar approach could be applied in SPL engineering.

Forms are the simplest kind of configuration interfaces targeted by our approach. Thus, using only an FM one would be able to generate a basic form. To make this transformation possible, a set of rules for mapping FM constructs to UI widgets is required. For example, TVL bool attributes could be translated to checkboxes, enum attributes to drop-down lists or list boxes, depending on the numbers of available values, etc. Even if this translation is technically feasible, the result would be rough as it is relies only on information contained in TVL models (and FMs in general) which is rather technical. For example, using feature and attribute names
as label for the input fields might not be expressive enough to understand their meaning. To tackle this problem we propose to add a so-called Property Sheet. This document is meant to store display information to beautify the UI, by attaching a display name, help text, etc. to features and attributes.

Based on the FM and the associated Property sheet, an AUI can be defined for the form. 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 concrete UI which is the implementation of the UI in a given language for a specific platform. For example, Figure 2 sketches a concrete UI corresponding to a form for the following TVL model representing the variability of a motherboard:

```plaintext
root Motherboard {
    group oneOf {
        Asus {ifIn: parent.socket==LGA1156;},
        Aopen {ifIn: parent.socket==ASB1;}
    }
    enum socket in {LGA1156 , ASB1};
    int price in [0..500];
    struct dimension {
        int height;
        int width;
    }
}
```

There, labels for the different input fields, like Manufacturer, come from the Property sheet. Feature decomposition and enumerated attributes both have been represented using drop-down lists. Integers have been implemented in two different manners: the dimension structure and its width and height attributes with a stepper field, and the price with two fields associated to Min and Max values. Such a representation of the price attribute has to be defined using the Property sheet. One can also notice that the Socket option is greyed as the choice of this attribute has been determined by the Manufacturer selection. Integration of the UI with solvers will be discussed in Section 3. Question marks on the right side of the input fields correspond to help information. This information has to be defined in the Property sheet.

### 2.2 View-based Interfaces

Forms are suitable for relatively small FMs but once their size increases, managing all the variability in a single form becomes a non trivial task. The user might be overwhelmed by all this information displayed in a single monolithic screen. To manage this complexity, we propose to divide FMs depending on the different concerns they include using multi-view FMs [25].

A view is “a simplified representation 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” [25] and corresponds to a subset of the features of the FM. Several views allow to divide the FM into smaller, more manageable parts.

Developers now have to define Views corresponding to the FM (see Figure 1). Hubaux et al. identified two ways of defining views: through extensional definitions, i.e. by enumerating all features appearing in a view, or through intentional definitions, i.e. by providing a language that takes advantage of the tree structure and avoids lengthy enumerations [25]. Once views 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, their display order, etc.

The illustrative TVL model of previous section restricted the configuration of a computer to its motherboard. But in actual configuration, other concerns like CPU, graphics or accessories could also be included in the FM. To avoid displaying all the information in the same window one could define four views corresponding to the different concerns of a computer configuration. Figure 3 sketches an interface illustrating this separation of concern. There, the different views have been implemented by four tabs but one could think about different implementation approaches which could be influenced by information contained in the Property sheet.

### 2.3 Workflow-based Interfaces

Views might not be sufficient in some cases where more complex constructs like ordered or user-determined sequences of views are required. We identified two distinct families of UIs belonging to this category: wizards and eclipse-like applications. A wizard is a UI where the user is guided through a succession of screens, eventually influenced by her
choices. This kind of UI is generally used at compilation time. Eclipse-like interfaces are UIs where the user determines her task ordering by selecting the different options available in the interface (toolbar, menubar, etc.). Contrarily to wizards, this kind of UI is generally used to manage configuration at runtime.

To describe the behaviour of those high-level UIs, we suggest to use Feature Configuration Workflows (FCWs) proposed by Hubaux et al. [22]. There, the authors extend their work on views by proposing to use workflows to drive their configuration. Their research was inspired by the concept of multi-level staged configuration of Czarnecki et al. [13]. 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. An FCW is thus a combination of views on the FM, workflow and the mapping between them. Up to now, FCWs focused on distributed configuration among several stakeholders but one might easily adapt it to other purposes like the dynamic behaviour of a UI in our case.

While the link between wizards and FCWs is direct (a wizard is a predefined sequence of screens where user’s choices will determine the path between a start and an end), it requires more explanation for eclipse-like UIs. Even if the use of such complex UIs seems non-linear, we claim it can still be represented using workflows. Figure 4 is an example of such workflow expressed in the YAWL formalism [44]. There, T* labels correspond to tasks and C1 label refers to a condition. Our complex UI is composed of a home screen (T1) where the user has three options (T21, T31, T4). Once she has chosen one, she enters a “sub-workflow” which can be composed of a single task (T4), a sequence of tasks (T21-T22) or more complex (T31, T32, T33, T34). Once tasks of the selected path are complete, task T5 is reached. It corresponds to a “dummy” join task. Then comes condition C1 which is used to determine if the configuration is finished. If the condition is not satisfied, the workflow goes back to task T1, so creating a loop. On the other side, a positive answer would mean that the configuration is finished. This condition could either be automatic (the workflow loops until there is no more variability to be configured) or manual (the user clicks on a button to indicate that she has completed her configuration task).

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

3. CHALLENGES

As previously mentioned, Section 2 is an overview of the proposed approach to be implemented. In this section, we will discuss some of the remaining challenges and leads to explore.

User Interface Description Language. Different languages, so called User Interface Description Languages (UIDLs), have been proposed in the literature to express UIs. They consist of languages which describe various aspects of an UI in an abstract manner, like e.g. AIOs. UsiXML [46], UIML [2] or XUL [49] are some examples of such languages. Each UIDL has its own characteristics like supported platforms, target languages, device-independence or available tools. Guerrero-Garcia and González-Calleros [19], and Souchon and Vanderdonckt [41] surveyed some of those languages based on their criteria. We will also have to conduct a similar survey to select the most appropriate UIDL according to our own requirements.

AIO mapping. Once the UIDL has been selected, a mapping between FM constructs and AIOs will have to be defined. It includes low-level as well as “pattern” mappings. By low-level we mean direct mappings between FM constructs and AIOs like checkboxes for Boolean attributes, for example. Most mappings will not be one-to-one but one-to-many based either on properties of the FM constructs or on user preferences. An enumerated attribute can, for example, be represented either by a list box or a drop-down list, depending on the number of elements in the enumeration. Additionally, some patterns in FMs might also be associated to specific AIOs. For example, an or-decomposed feature whose sub-features are leaf nodes could be mapped to a list box allowing selection of multiple values. For this task, we can rely on guidelines commonly accepted in the human-computer interaction (HCI) community [48].

Objects placement. Another aspect which we will have to take into consideration is the placement of the identified AIOs on the screen. Following the HCI literature, this task is non-trivial and there have been successes in limited domains such as dialogue box design and remote controls [32]. An alternative solution would be to provide the UI designer with a set of automatically generated atomic AIOs which she can place in the layout structure she has defined, so ensuring the clarity of the layout. Schramm et al., for example, implemented this second option [40].

Property sheet. A language influencing the beautification of the UI will have to be defined. Our first intention is to extend TVL with some UI-specific tags. We have to evaluate the feasibility of this approach, especially the integration of TVL annotations with views and FCWs. Thanks to the structuring mechanism of TVL (include construct), beautifying annotations could be defined either in the same file as the FM or in a specific file, so not interfering with the reading of the FM. Supported tags should at least include label...
(the label of an UI element), \textit{group} (to define a group of features logically linked), \textit{group element} (to attach a feature to a group), \textit{ordering} (to define the order of the features), \textit{hide} (to not display a feature in the UI) and \textit{default} (to define the default value in the UI) constructs. View definition instructions could also be part of those constructs.

\textbf{Views.} In their research on views, Hubaux et al. focused on three main aspects: \textit{view specification}, \textit{view coverage} and \textit{view visualisation} [25]. There are two ways of specifying views: extensional definition (enumerate each feature appearing in a view) and intentional definition (take advantage of the tree structure) using a subset of XPath\footnote{http://www.w3.org/TR/xpath/}. In our case, we could implement the extensional definition using TVL tags, as mentioned in previous challenge. A drag-and-drop tool is another solution which could be explored. The second aspect, view coverage, should also be verified for Uls as it guarantees that all configuration questions be eventually answered. The conditions defined by the authors should be sufficient for our UI generation approach and require no modification. Finally, the three view visualisations presented in [25], such as displaying only features that are relevant for the view are of little interest in UI generation. However, we might go through the same process to identify different ways of displaying views in a UI (tabs, etc.).

\textbf{Feature Configuration Workflows.} As previously mentioned, an FCW is a workflow such that tasks are associated to FM-views. A view will have to be configured during the execution of its associated task [22]. Thanks to inter-views links, choices made in a task will automatically be propagated to other parts of the FM. Even if the work of the authors focused on the configuration of an FM split among several stakeholders, it should only require slight changes to support the dynamic aspect of Uls. FCWs in the context of UI generation would even be less restrictive. In FCWs, a variability point cannot be left undecided unless one can ensure that it will be configured later in the workflow. Otherwise, deadlock could appear in the configuration process [11]. This property is still valid for wizard generation (except if default values are available for all variation points) but not for other Uls. Indeed, the workflow will either loop until there is no variability left ("automatic" condition of Section 2.3) or signal to the user that she still has some variability left when she validates her configuration ("manual" condition of Section 2.3). Constraints related to parallel configuration will also be relaxed in UI generation. In FCWs, conflicts might appear during the reification of FM-views which were configured in parallel tasks. We do not expect to run up against the same problem in UI where tasks should not be conducted in parallel. We will thus have to analyse FCWs in further detail to check if some other constraints can be relaxed or, contrarily, should be added.

\textbf{Solver Integration.} A crucial part of the approach which has not been mentioned yet is the link between the generated UI and a solver, this last being the core component of variability configuration. This involves a two-way communication. Each time the user makes a choice in the UI it has to be forwarded to the solver which will compute a new set of valid configurations. This set must then be sent back to the UI in order to update options available to the user. Being able to give an explanation for automatically selected/disabled UI options would also be a nice-to-have. The solution should be as generic as possible (i.e. require slight modification) to support different solvers (SAT, CSP, BDD, SMT,...), depending on the targeted configuration task.

\textbf{Application Integration.} Similarly, for interactive applications, we will have to propose a mechanism to easily link the generated UI with the functions of the application. Manually mapping UI buttons to application functions is still possible but is a time consuming and error-prone task. Our ultimate goal is thus to also use the list of functions of the application as input of our generation process.

\textbf{ Beautification.} Despite their efficiency, model-based generation of Uls have often suffered from a lack of usability and flexibility [12, 34] of resulting Uls. Mechanisms to improve the quality of generated Uls will thus have to be proposed. This concern is still an open question in the HCI community. An option would be to define tags in the property sheet, additionally to those aimed at beautifying FM-views and FCW-related information. Providing UI templates as input of the UI generation process is another alternative. This would especially allow to comply with existing in-house visual guidelines. This option requires the definition of a linking mechanism between templates and other inputs. Alternative solutions as well as their combination will have to be considered.

\textbf{Round Tripping.} Another consequence of the failure to generate good UIs is that they are generally customized by graphical designers. A new kind of problem might arise when the underlying model (FM in our case) is modified (new configuration rules, new options): the designer will have to customize the UI each time it is generated. To minimize this recurring manual task, we intend to define a way to send enough information back to the configuration UI generator to take designer’s modifications into account each time the UI is generated.

4. RELATED WORK

In the HCI research domain, automation of Uls 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 automatize the generation of interfaces. On the other hand, fully automated approaches generally fail to generate good Uls, except for domain specific applications [31]. Most approaches propose a partially automated process which uses extra information about the UI stored in models. They are all grouped under the \textit{Model-based User Interface Development} (MBUID) denomination. They are generally supported by a MBUID environment (MBUIDE) which is "a suite of software tools that support designing and developing Uls by creating interface models" [17]. Among them we can mention ADEPT [27], Teallach [33], MASTERMIND [42] or GENIUS [26] which also uses the notions of views on entity-relationships models and sequence between them to generate Uls. 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. [17].
Most approaches rely on the same principle: starting from
a task model, domain model and/or user model, an abstract
presentation model is derived, which will be implemented
by a concrete interface model. The task model describes
the task that the user can perform, including sub-tasks, their
goals as well as the procedures used to achieve them. The
domain model is a high level representation of the objects
and their associated functions in a given model. Other mod-
els, like discourse or application models, are implemented
in some approaches. Our approach is related to MBUID in
that it is also based on models and follows a similar process.
But, in our case, the models are different as variability and
workflow models are used as input.
Pleuss et al. combine SPLs and the concepts from the
MBUID domain to integrate automated product derivation
and individual UI design [34]. 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 ap-
application can be fully automatically generated while others
like the visual appearance are also generated automatically,
but can be influenced by the user. While we share similar
views regarding MBUID, our overall goals differ. Pleuss et
al. aims at generating the UI of products derived from the
product line while our interest is on generating the interface
of a configurator allowing end users to derive product ac-
cording to their needs. We are therefore not concerned
with product derivation but rather on the link between feature
model configuration and UIs.
Schlee and Vanderdonckt [39] also combined FMs with
graphical UI generation. Relying on the generative program-
ning paradigm, the authors represent the UI options with an
FM which will be used to generate the corresponding inter-
face. 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 be believe to
be a critical concern for complex UIs.
In most variability-related tools, FMs are represented and
configured using tree-views. We can, for example, men-
tion pure::variants [6], FeatureIDE [28] or Feature Mod-
eling Plug-in [4]. Those tools have a graphical interface
in which users can select/deselect features in a directory-
tree like interface where constraints are automatically prop-
gagated. Several visualization techniques have been proposed
to represent FMs [36], 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 Batory et al. [18].
Simple Java configuration interfaces including checkboxes,
radio buttons, etc. are generated using beautifying anno-
tations supported by the GUIDSL syntax used in the tool
suite. Examples of annotations are disp which corresponds
to the displayed name of a feature, help which stores help
information for a feature, tab which defines a new tab rooted
by the associated feature in the UI, hidden which allows to
hide a feature, etc. We are studying those annotations as
well as their syntax to implement similar tags into TVL.
Automatically selecting AIOs on the basis of data types
has already been solved in the HCI community. For exam-
ple, MECANO [37] selects interactors depending a.o. on the


