Model Transformation Configuration and Variability Management for User Interface Design

Jean-Sébastien Sottet(✉), Alain Vagner, and Alfonso García Frey

Luxembourg Institute of Science and Technology,
5 av. des Hauts-Fourneaux, Esch/Alzette, Luxembourg
{jean-sebastien.sottet,alain.vagner,alfonso.garcia}@list.lu

Abstract. User Interfaces (UI) design is a complex and multi-faceted problem, owing to the ever increasing variability of the design options and the interaction context (devices, user profiles, and their environment). Moreover, UI design choices stand on users' needs elicitation, which are difficult to evaluate precisely upfront and require iterative design cycles based on trial and error. All this complex variability should be managed efficiently to ensure moderate design costs. In this article, we propose a variability management approach integrated into a UI rapid prototyping process, which involves the combination of Model-Driven Engineering (MDE) and Software Product Lines. Our approach supports the separation of concerns through multi-step partial configuration of UI features, enabling each stakeholder of the UI design process to define the variability on the assets she manages. We have implemented this approach in our existing MDE UI generation Framework.

Keywords: Software engineering · Software product lines · Variability management · Configuration · Model-driven engineering · Model transformations · Human-computer interaction · Model-based user interfaces · Feature models

1 Introduction

In design and development of User Interfaces (UI) it is often required to produce several versions of the same product, including different look and feel and user tasks for different platforms. As stated by [7] interaction design is a complex and multi-faceted problem. When designing interaction, variability is manifold: variability of devices, users, interaction environments, etc. Moreover, user requirements are difficult to evaluate precisely upfront in UI design processes. Therefore, the main UI design processes, such as User-Centred Design [13], implement an iterative design cycle in which a UI variant is produced, tested on end-users, and their feedback is integrated into design artifacts (e.g., part of the UI, requirements, etc.). Since these processes are mostly based on trial and error, some parts of the UI have to be re-developed many times to fit all the different user

P. Desfray et al. (Eds.): MODELSWARD 2015, CCIS 580, pp. 390–404, 2015.
DOI: 10.1007/978-3-319-27869-8_23

requirements. Moreover these processes involve multiple stakeholders with different roles (software developers, UI/User eXperience designers, business analysts, end-users, etc.) that demand a great amount of time to reach consensus. UI variability has thus a significant impact on the design, development and maintenance costs of the UI.

To overcome variability issues in software engineering, researchers have proposed to rely on the paradigm of Software Product Lines (SPLs) [10]. The SPL paradigm allows to manage variability by producing a family of related product configurations (leading to product variants) for a given domain. Indeed, the SPL paradigm proposes the identification of common and variable sets of features, to foster software reuse in the configuration of new products [22].

Model-Driven Engineering (MDE) has already been used to improve the UI design process [27]. According to [4,11] SPL and MDE are complementary and can be combined in a unified process that aggregates the advantages of both methods.

In this paper, we propose an approach to manage UI variability based on MDE and SPL. Our approach relies on model transformations that support the expression of the variability. This approach enables the separation of concerns of the different stakeholders when expressing the UI variability and their design choices (UI configurations). We considered a Multiple Feature Model approach in which each feature model represents a particular concern allowing, if needed, each stakeholder to work independently. We also support the configuration reconciliation to reach a consensus through constraints and dependencies between UI features. We thus proposed a partial and staged configuration process [12] in which we produce partial UI configurations that can be refined by all stakeholders including the users feedback. Finally, we propose to integrate the configuration inside the transformations, modifying existing UI model transformations and integrate them into our model-driven UI design process [27]. We illustrate these concepts with a concrete example of UI variability.

2 Related Work

2.1 Feature Modelling

Feature models (FM) [22] are popular SPL assets that describe both variability and commonalities of a system. They express, through some defined operators, the decomposition of a product related features. The feature diagram notation used in this article is explained in Fig. 1. The E-Shop FM consists of a mandatory feature “catalogue”, two possible payment methods from which one or both could be selected, an exclusive alternative of security levels and an optional search feature. FM constraints can be defined. In this case “credit card” implies a high level of security.

Features composing a FM depict different parts of a system without any clear separation. The absence of feature types makes these models popular as there are no limits for the expression of design artifacts. But at the same time, [8] have

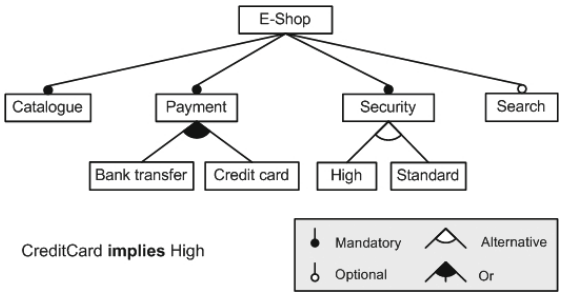

Fig. 1. Feature Model from an E-shop. Source: Wikimedia commons.

demonstrated that depicting information in a single FM leads to feature redundancies due to the tree structure. As a result, separation of variability concerns into multiple FM seems to be crucial for understanding [18] and manipulating [1] the many different faces of variability. Each of these FM focuses on a viewpoint on variability which makes easier to handle variability for each stakeholder.

2.2 SPL Configuration

The configuration process is an important task of SPL management: producing a particular product variant based on a selection of features to fit the customers' needs. In this context, a configuration is a specific combination of FM features such as hierarchical decomposition, operators (Or, optional, etc.) and constraints of FM.

As stated by [29], when designers and developers configure a system according to requirements, the enforcement of FM constraints can limit them in their design choices. Moreover, the separation of the variability in multiple FMs is also a source of complexity due to many dependencies across FMs. The fusion of all FMs into one for configuration purposes seems to solve this issue but results in a large FM that mixes different facets: this may lead to invalid configurations and thus inefficient products. Some solutions exist to overcome these problems. The work by [23] proposes an implementation of a configuration composition system defining a step-by step configuration [12] using partial configurations [5]. Thus, some portion of a FM can be configured independently, without considering all the constraints (coming from other configurations) at configuration time. Then, constraints amongst configurations may be solved by implementing consistency transformations such as in [1].

2.3 Model-Driven User Interfaces Variability

Model-Driven UI calls for specific models and abstraction. These models address the flow of user interfaces, the domain elements manipulated during the interaction, the models of expected UI quality, the layout, the graphical rendering, etc. In addition, each model corresponds to a standard level of abstraction as

identified in the CAMELEON Reference Framework (CRF) [9]. The CRF aims at providing a unified view on modelling and adaptation of UIs. In the CRF, each level of abstraction is a potential source of variability. Modifying a model of a specific level of abstraction corresponds to a specific adaptation of the UI.

For instance in the CRF, the most abstract model, the task model, depicts the interaction between the user and the features offered by the software. Adding or removing a task results in modifying the software features. Considering this, we can assume that there is a direct link between classical feature modelling and task modelling such as presented in [21]. In this work, a task model is derived from an initial FM. However the authors did not go any further in describing the variability related to interaction and UI (graphical components, behavior, etc.).

In [14], the authors present an integrated vision of functional and interaction concerns into a single FM. This approach is certainly going a step further by representing variability at the different abstraction levels of the CRF. However, this approach has several drawbacks. On the one hand, this approach derives functional variability only from the task model, limiting the functional variability of the software. On the other hand, all the UI variations are mixed into a single all encompassing FM which blurs the various aspects for comprehension and configuration [18].

Finally, Martinez et al. [19] presented an initial experience on the usage of multiple FMs for web systems. This work showed the feasibility of using multiple FMs and the possibility to define a process around it. It implements FMs for a web system, interaction scenario, a user model (user impairments), and device. However, this approach does not consider the very peculiarity of UI design models and their variability.

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

3 UI-SPL Approach

Model-driven UI design is a multi-stakeholder process [15,17] where each model – representing a particular subdomain of UI engineering– is manipulated by specific stakeholders. For instance, the choice of graphical widgets to be used (e.g. radio button, drop-down list, etc.) is done by a graphical designer, sometimes in collaboration with the usability expert and/or the client. Our model-driven UI design approach [27] relies on a revised version of the CRF framework (Fig. 2).

It consists of two base metamodels, the Domain metamodel -representing the domain elements manipulated by the application as provided by classical domain analyst- and the Interaction Flow Model (IFM) [20]. From these models we derive the Concrete User Interface model (CUI) which depicts the application "pages" and their content (i.e., widgets) as well as the navigation between pages. The CUI metamodel aims at being independent of the final implementation of any graphical element. Finally, the obtained CUI model is transformed

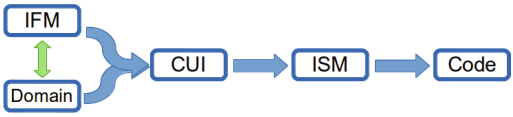

Fig. 2. Model-Driven UI design process.

into an Implementation Specific Model (ISM) that takes into account platform details (here platform refers to UI tool-kits such as HTML/JQuery, Android GUI, etc.). A last model-to-text transformation generates the code according to the ISM. This separation allows for separate evolution of CUI metamodel and implementation specific metamodel and code generation.

We propose a multiple feature models (multi-FM) approach (see Sect. 3.1) to describe the various facets of UI variability (e.g., UI layout, graphical elements, etc.). In a second time, (see Sect. 3.2) we introduce our specific view on configuration on this multi-FM and its implementation in our model-driven UI design approach (see Sect. 4.3).

3.1 Multi-FM Approach

Classical FM approaches combine different functional features [21]. In the specific context of UI design, we propose to rely on a similar approach for managing variability of each UI design concern. UI variability is thus decomposed into FMs (Fig. 3).

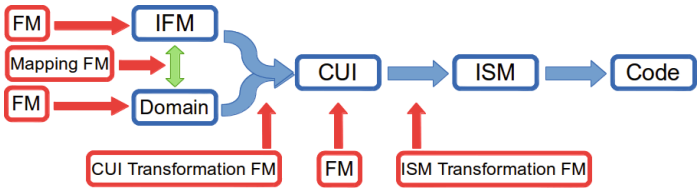

Fig. 3. Variability models (FM) coverage on our UI modeling framework.

Each of these FMs is related either to a model, a metamodel, a mapping or a transformation depending on the nature of the information it conveys.

– Models: Three FMs in Fig. 3 manage variations at the model level (IFM, Domain and CUI). The FM responsible for the Domain configuration can be used to express alternatives of a same concept, e.g., using or not the address, age or photo of a given class "Person". The IFM variability can express for instance the possible navigation alternatives to be activated or not. For instance on a shopping website, shortcuts providing quickly access to a given product can be configured. The CUI FM represents alternative representations of a widget: a panel (i.e., portion of UI displayed on a screen) can become a full window (i.e., displayed has full-screen) on mobile phones.

- Mappings: The variability of the mapping between IFM and Domain can be managed by a mapping FM. Interaction flow elements (UI states) can involve different concepts of the domain model.
- Transformations: Variability can be expressed also at the level of transformations. The variability that a transformation could convey (i.e., multiple output alternatives) can be expressed with FMs. The transformations impacted are (1) between IFM, Domain and CUI, (2) between CUI and ISM. The variability of (1) expresses the possible UI design choices: how an IFM state selection can be represented: a simple list, an indexed list, a tile list, etc. The variability in (2) depends on the target ISM and configures the final representation to be provided to end-users. For instance, a CUI simple list can be represented using, as output of the transformation, the following HTML markup alternatives: "<select>" or "<ul>".

By scoping the FM to a specific concern, our approach allows to focus only on the variations related to the underlying concern. In our approach, the different FM enrich the existing UI design process accompanying, step by step, the design of models and its variability. However, all the concerns are interrelated in the generation process: the IFM, Domain and CUI are in relation together (derivation traceability, explicit mapping, etc.). Thus, the FMs are interrelated as well leading to dependencies and sometimes overlaps between them. As such, the configuration of the transformation IFM-Domain to CUI is overlapping the FM of the CUI. If the UI design stakeholders are following an automated approach, they will rather use the transformation FM. The FM responsible for the CUI transformation configuration (i.e., elements to be displayed) is directly dependent on the mappings between domain elements and IFM. For instance, if the picture (Domain) is not mapped to a selection state (IFM), the generated CUI list would not contain this picture.

3.2 Configuration: The Specific Case of Rapid Prototyping

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

[1] http://balsamiq.com.

their tasks. In fact, through these iterations they elicit the product configuration that best fits their needs.

This user-centred process is composed of many iterations that test only a portion of the UI. If we can test the global interaction experience with the end-users, the designer may also need to focus on one particular UI part/concern in isolation. In previous work [26,27], we have tested the global usability of generated UI prototypes. We will focus here on the configuration of specific UI parts. In order to perform this, we stand on partial configurations [5]. For instance, configuring content of a specific page or a specific interaction state.

The partial configuration does not take into account the external feature[2] constraints which enter in the partial configuration (e.g., requires, implies, excludes, etc.). In other words, if another configuration has a feature which requires/excludes a feature of the partial configuration being set we simply do not consider it. For the features required or excluded by the partial configuration being set, we do not involve the end-users/designer but rather compute the consequences: i.e., provide a configuration for these external features. The constraints that are expressed inside the partial configuration are still to be considered as in traditional approaches.

We designed a generation process of UI prototypes from partial configurations and their evaluation with end-users. We divided this process into 4 steps from the identification of UI parts to be tested to the corresponding prototype generation:

1. **Selection of the Model Parts to be Configured:** in order to target the critical portion(s) of the application to be tested with end-users, the selection should be done on the input design models (e.g., IFM and Domain). Then, we deduce the impacted FMs and provide for each of them the relevant subset.
2. **Combination of the Various FMs Subsets:** the subset of FMs previously selected have to be aggregated (see FMs aggregation [1]) together in order to produce a unified configuration.
3. **Establishment and Use of Configuration UI:** the configuration UI can be derived from the obtained FM. We have developed a FM to CUI transformation that is used to generate the configuration UI.
4. **Executing "Variability Aware" Transformations:** we have finally developed transformation rules that take into account variability rules (that we will call later variability rules) related to the idea of [25]. The variability rules allow for a smarter management of FM in transformations. Secondly, these transformations should also support, with some default heuristics, elements that have no configuration.

In order to illustrate the process above let's introduce a simple application example (Fig. 4). We have realized this application model with our application modeling tool AME [16,17]. The application[3] proposes to search actors that play in a film, whose name is given by the user; then the user is able to select one actor to see more detailed information (birth date, photos, etc.). This simple

[2] External features refer to features coming from another partial configuration.
[3] This example is supported by the Neo4J tutorial: Movie DB, www.neo4j.com.

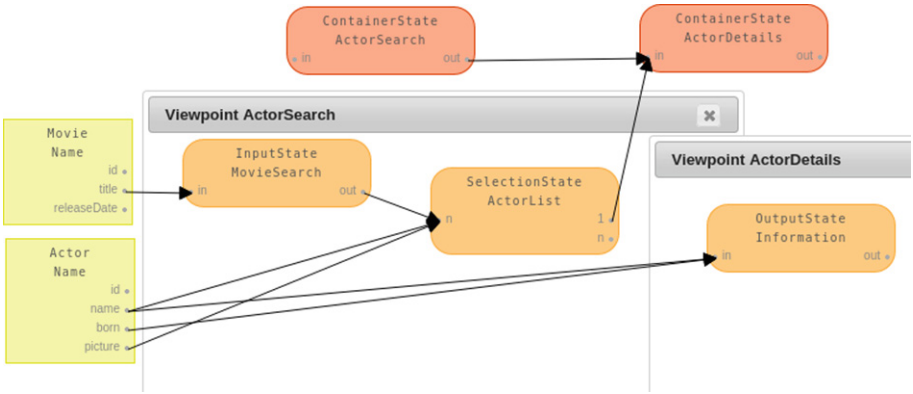

Fig. 4. Model of the application realized with our modeling tool AME [16,17].

application is composed by two interaction flow states: one for searching by film and displaying the list of actors and one for displaying information about the selected actor.

In this example, the UI designer wants to focus on a specific element to test and enhance with end-users: test the most appropriated way to select the actor[4]. Thus, the designer could propose several variations for the actor selection. The actor selection is supported by different types of lists as depicted by the CUI metamodel (tile list, radio-button, etc.) and the information that this list should convey (actors photo, name, etc.).

The rapid prototyping configuration occurs in the transformation between IFM, Domain and CUI Models (Fig. 3). This configuration stands on the combination of two FMs (as described in step 2). Firstly, the "CUI transformation FM" that depicts the variability of the widgets to be generated (see Fig. 5 upper part CUI decomposition). Secondly, the "Mapping FM" (mapping between IFM and domain) which depicts the configuration of the information conveyed by the list of actors (see Fig. 5 right frame Mapping FM). We also added a constraint: to be efficient a tile list requires to show some pictures (Tile List implies picture).

As a summary, the FMs of the Fig. 5 correspond to the UI designer specific viewpoint when he/she starts to configure the actors selection.

4 Implementation

In the following section we will illustrate the process described in Sect. 3.2. As such we introduce the management of design and feature models and the selection of the part of FM from which we will derive (generate) the partial configuration. Finally, we introduce the transformation rules tuned for supporting partial configuration.

[4] Here, we focus on a widget but other factors are to be adjusted by the interface designer like style, layout, etc.

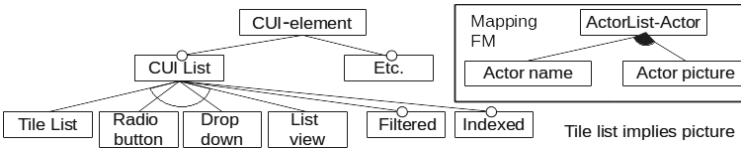

Fig. 5. CUI Transformation and Mapping IFM-Domain FM for excerpt List configuration.

4.1 Feature Models: Selection and Reconciliation

The selection of the part to be configured is done through the design models. On such models the designer is able to select the critical part of the UI. For instance, it can be a crucial part of the application or a problematic situation (e.g., a particularly long interaction with many steps, a page with many information to fill in, etc.). In our simple example, the "ActorList" selection state, related to the concept it manipulates, is central to the application: it is the main entry point for the users.

Once the part of the input design model is selected, the relation with the FM still has to be found. The model transformation gives us a first link to the target metamodel elements. In fact, the transformation is linking elements from the input models (previously selected in the IFM and Domain models) to elements from the target metamodel. Through the transformation scope (e.g., CUI List) we are able to scope the variability of the target metamodel[5]. The IFM states "selection" are transformed into various types of lists. As a result we have selected the portion of CUI transformation FM corresponding to List. We have to do the same for the domain concept mapped to the "selection" state: as shown on Fig. 4 the selection state actor list is mapping to the attributes of concept actor name and picture. The selection can be implemented using the slicing operator defined in Familiar [2].

Once the FM have been selected and rightly scoped (i.e., we obtained FM portions) we have to aggregate them together. We can use the insert operator of Familiar using the following Familiar [2] expression see Listing 1.1.

Listing 1.1. Familiar insertion operator for building complete partial configuration.

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

4.2 Generation of the Configuration UI

Once the aggregated FM is produced for the specific point that needs to be configured, it is necessary to build a configuration interface. As the FM are selected and composed on demand, we should significantly improve the production time of it by using UI automatic generation. Indeed, this kind of UI can be

[5] We designed the transformation FMs in accordance to the transformations themselves.

automatically generated thanks to simple heuristics, see for instance [6,24]. We implement some of these heuristics in order to produce from a FM a configuration CUI. For instance in Listing 1.2 we generate a label and a check-box (Input of type 'checkox') for each optional feature (using the isInOptionalGroup()). In the same idea, mandatory features are just shown with a label (no user action is required). For the groups (XOR, OR, AND), we generate specific elements: XOR are decomposed into a drop-down list where only one feature can be selected, OR as a checkbox group. Finally, for each level of the FM hierarchy we generate a container (panel or a tab) that would help separating the features.

Listing 1.2. Configuration rule for optional feature.

```
rule FeatureOptionalToCheckBox {
  from
    source : MM!Feature (source.isInOptionalGroup())
  to
    target : MM2!Input (
      id<- 'checkbox_'+source.id,
      type <- 'checkbox',
      content <- source.name ),
    t2: MM2!Label(
      content <- source.name,
      "for" <- target,
      id <- 'checkbox-label_'+source.id )
}
```

As such, we have generated a configuration (see Fig. 6) for the partial aggregated FMs of Fig. 5. We have thus generated a drop-down list for the selection of list type, a group of check-boxes for list options (filtered, indexed) and finally another group of check-boxes for each mapped concept (name and picture).

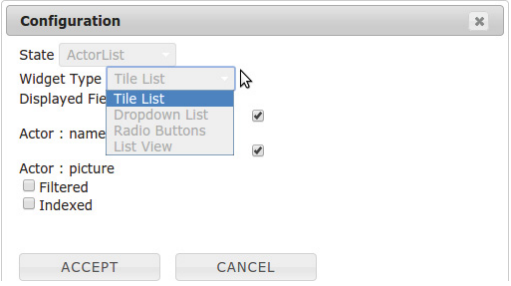

Fig. 6. Configuration Interface for a given State (here "ActorList").

The configuration UI is filling a configuration model conforms to a configuration metamodel (a generic configuration metamodel dedicated to our UI design process). It sets for the selected interaction state - IFM model - (e.g., ActorList) the value of "WidgetType" to be targeted and the potential options (for a list: filtered, indexed, etc.).

4.3 Transformations

The implementation of our approach relies on an existing system that derives a UI from IFM and domain models using successive model transformations [28].

This initial system was not taking into account the configuration of the variability. We added the possibility of configuring the system using a specific UI dedicated to specific configurations.

In addition, as mentioned in the Sect. 3.2, we should not freeze the UI generation until a complete configuration is provided: we rely on partial configuration. As a result, we have kept our initial tool behavior: it should generate an executable UI even if no explicit configuration is provided for all elements. A default transformation (see Listing 1.3) is executed if it has no configuration (i.getConfiguration.oclIsUndefined()): it will, by default, transform a selection state into a "ListView". The "getConfiguration" helper uses the explicit link between the input models (IFM/Domain) and the configuration model. It is based on a reference of an IFM state previously selected (see Fig. 6 where the reference to the IFM state model is given - State ActorList).

Listing 1.3. Excerpt of the default transformation used if no configuration is defined.

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

We modified our initial tool by adding the configuration as a parameter of the transformation. This has no impact on the other transformations allowing us to reuse the rest of the transformation chain up to the application generation: CUI to ISM and ISM to Code. For each type of widget to be generated a rule is produced (see the Listing 1.4 for a configuration of a tile list). Each particular attribute of the widget (i.e., indexed and filtered) is also dependent on the configuration thus introducing additional conditional expressions (ListFilters and ListDividers conditions in Listing 1.4).

Listing 1.4. Excerpt of Selection to Tile List in CUI transformation including configuration helpers.

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

```
        filterRevealedList <- false),
    divider : CUI!Divider(
      autodivider <- true )
}
```

4.4 Example of Product Configuration

In order to perform a configuration, the designer uses the interface provided in Fig. 6. He/she first selects the input state model element (e.g., selection state "ActorList") and the type of widget (e.g., among the type of lists) using the drop-down menu. In addition he/she can select/de-select the domain elements manipulated by this state (configuration of the mapping FM), for instance removing/adding the Actor name and Actor picture that could be displayed by the list. If the selected type of list is "Tile List", it needed an attribute of type image (e.g., the actor picture). As a result, the actor picture will be selected by default and not de-selectable.

Once the designer has finished and saves the configuration (i.e., as a configuration model), he/she can choose to launch the generation process. The transformation chain will generate the configured UI part within the rest of the application. End-users are able to evaluate the configured part inside the whole application interaction process. This task has to be repeated for each identified configurations. Figure 7 shows the result of two UIs generated from two different list configurations. A video summarizing the edition of models and the realization of configurations, including the automatic generation of the prototypes, is available at http://youtu.be/78t11o0jatU.

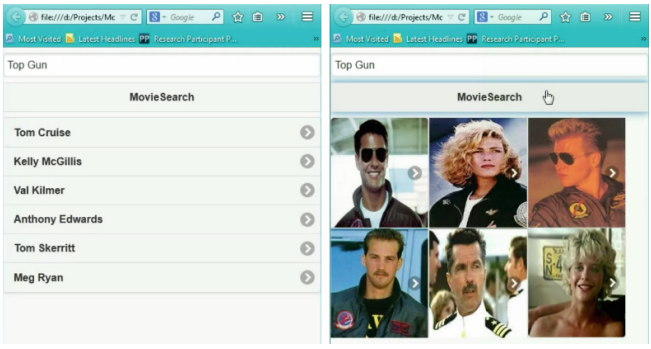

Fig. 7. At left: List View configuration for actor selection displaying actor names. At right: Tile List configuration with actor images.

Once a convincing configuration is positively evaluated by the end-users, it is stored to be used for the final product. The final product configuration should be done by composing all the relevant configurations, using for instance the approach by [23].

5 Conclusions and Perspectives

We have considered in this article the issue related to UI variability. UI variability has numerous facets (e.g., graphical design, development, usability, etc.) due to the diversity of stakeholder profiles (including end-users). Moreover, in UI design, one encounters frequently the difficulty to align the products with fuzzily defined user requirements. This complexity can lead to an inefficient UI design process, which has an impact on the UI design costs.

Therefore, we proposed an approach to manage UI variability based on MDE and SPL, integrating SPL management into our current MDE UI design process. Whereas traditional approaches focus on the elicitation of product line from FMs only, we rather claim that FMs are supporting the design of a product line completing the existing design choices (i.e., the one made by expert stakeholders using their design models). In our approach, UI design stakeholders can express the variability on the models, mappings and transformations by defining multiple FMs related to each of these assets.

In order to build a viable product, the stakeholders have to confront their viewpoints when configuring products. The proposed approach is based on iterative-steps and partial configurations that can be refined by all stakeholders including the end-users. As a result, we have worked on the selection and aggregation of FM subsets in order to provide a unified frame for building a partial configuration. Then, in order to support such dynamic definition of partial configuration, we have to automatically derive a configuration UI based on the sliced and aggregated FMs.

Partial configurations are used to parameterize the transformations: it specifies some design choices. Default transformation heuristics are then used to complete the partial configuration. A partial configuration should be independent from other partial configurations (i.e., independent from external constraints) but we should derive all the implications (i.e., required or disabled features) of the current partial configuration in order to generate a proper product.

More particularly, in the context of rapid prototyping, the UI designer can focus on a specific point, using partial configurations and test it with end-users. We implemented this approach in our existing UI model transformations which are parameterized by partial configurations and based on heuristics to generate default UI elements for the features that are not configured.

As future work, we have to further explore the merging of partial configurations in order to produce a final product configuration according (e.g., [23]) to the global product constraints. When trying to solve the global constraints from each individual partial configurations, we can reach a point where the product expected by the end-users enforces some of theses constraints. Thus, we will have to provide some collaborative environment to actually help in reaching consensus and maybe make compromises amongst the partial configurations. As such, we plan to further exploit our collaborative infrastructure [15] for partial configurations reconciliation.

Finally, we discovered that the variability is manifold and multi-dimensional in the design of UIs and that FMs are of several types. We think that building

a taxonomy of these different FM could help us in understanding more precisely UI variability and could lead us to a better reuse of variability assets across projects and domains.

Acknowledgements. This work has been supported by the FNR CORE Project MoDEL C12/IS/3977071 and AFR grant agreement 7859308.

References

1. Acher, M., Collet, P., Lahire, P., France, R.B.: Separation of concerns in feature modeling: support and applications. In: Proceedings of the 11th Conference on Aspect-oriented Software Development (2012)
2. Acher, M., Collet, P., Lahire, P., France, R.B.: Familiar: a domain-specific language for large scale management of feature models. Sci. Comput. Program. **78**(6), 657–681 (2013)
3. Acher, M., Lahire, P., Moisan, S., Rigault, J.P.: Tackling high variability in video surveillance systems through a model transformation approach. In: MISE 2009. ICSE Workshop, pp. 44–49. IEEE (2009)
4. Batory, D., Azanza, M., Saraiva, J.: The objects and arrows of computational design. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 1–20. Springer, Heidelberg (2008)
5. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Inf. Syst. **35**(6), 615–636 (2010)
6. Boucher, Q., Perrouin, G., Heymans, P.: Deriving configuration interfaces from feature models: A vision paper. In: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, pp. 37–44. ACM (2012)
7. Brummermann, H., Keunecke, M., Schmid, K.: Variability issues in the evolution of information system ecosystems. In: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (2011)
8. Bühne, S., Lauenroth, K., Pohl, K.: Why is it not sufficient to model requirements variability with feature models. In: Workshop on Automotive Requirements Engineering (AURE04), at RE04, Japan (2004)
9. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A unifying reference framework for multi-target user interfaces. Interact. Comput. **15**(3), 289–308 (2003)
10. Clements, P., Northrop, L.: Software Product Lines. Addison-Wesley Boston, Boston (2002)
11. Czarnecki, K., Antkiewicz, M., Kim, C.H.P., Lau, S., Pietroszek, K.: Model-driven software product lines. In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 126–127. ACM (2005)
12. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. Softw. Process Improv. Pract. **10**(2), 143–169 (2005)
13. DIS, I.: 9241–210: 2010. Ergonomics of human system interaction-part 210: Human-centred design for interactive systems. International Standardization Organization (ISO). Switzerland (2009)
14. Gabillon, Y., Biri, N., Otjacques, B.: Designing multi-context uis by software product line approach. In: ICHCI 2013 (2013)

15. Garcìa Frey, A., Sottet, J.S., Vagner, A.: A multi-viewpoint approach to support collaborative user interface generation. In: 19th IEEE International Conference on Computer Supported Cooperative Work in Design CSCWD (2015)
16. García Frey, A., Sottet, J.S., Vagner, A.: Ame: an adaptive modelling environment as a collaborative modelling tool. In: Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 189–192. ACM (2014)
17. García Frey, A., Sottet, J.S., Vagner, A.: Towards a multi-stakehoder engineering approach with adaptive modelling environments. In: Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 33–38. ACM (2014)
18. Mannion, M., Savolainen, J., Asikainen, T.: Viewpoint-oriented variability modeling. In: COMPSAC 2009 (2009)
19. Martinez, J., Lopez, C., Ulacia, E., del Hierro, M.: Towards a model-driven product line for web systems. In: 5th Model-Driven Web Engineering Workshop MDWE 2009 (2009)
20. OMG: IFML- interaction flow modeling language, March 2013
21. Pleuss, A., Hauptmann, B., Dhungana, D., Botterweck, G.: User interface engineering for software product lines: the dilemma between automation and usability. In: EICS, pp. 25–34. ACM (2012)
22. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
23. Rosenmüller, M., Siegmund, N.: Automating the configuration of multi software product lines. In: VaMoS, pp. 123–130 (2010)
24. Schlee, M., Vanderdonckt, J.: Generative programming of graphical user interfaces. In: Proceedings of the Working Conference on Advanced Visual Interfaces, pp. 403–406. ACM (2004)
25. Sijtema, M.: Introducing variability rules in atl for managing variability in mde-based product lines. In: Proceedings of MtATL 2010, pp. 39–49 (2010)
26. Sottet, J.S., Calvary, G., Coutaz, J., Favre, J.M.: A model-driven engineering approach for the usability of plastic user interfaces. In: Gulliksen, Jan, Harning, Morton Borup, van der Veer, Gerrit C., Wesson, Janet (eds.) EIS 2007. LNCS, vol. 4940, pp. 140–157. Springer, Heidelberg (2008)
27. Sottet, J.S., Vagner, A.: Genius: generating usable user interfaces (2013). arXiv preprint arXiv:1310.1758
28. Sottet, J.S., Vagner, A.: Defining domain specific transformations in human-computer interfaces development. In: 2nd International Conference on Model-Driven Engineering and Software Developement (2014)
29. White, J., Dougherty, B., Schmidt, D.C., Benavides, D.: Automated reasoning for multi-step feature model configuration problems. In: Proceedings of the 13th International Software Product Line Conference (2009)