MASTER EN SCIENCES INFORMATIQUES

Génération d'interfaces de configuration à partir de modèles de variabilité

Lincé, Guillaume

Award date:
2013

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique
Année académique 2012–2013

Génération d'interfaces de configuration à partir de modèles de variabilité

LINCÉ Guillaume

Maître de stage : HEYMANS Patrick

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
HEYMANS Patrick

Co-promoteur : BOUCHER Quentin

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé et mots clés

Résumé : Depuis quelques années, la configuration de produits est devenu un enjeu majeur pour la plupart des entreprises afin de satisfaire au mieux leurs clients et ainsi gagner des parts du marché. Le nombre de configurateurs sur le Web est conséquent et les avantages sont nombreux, allant du gain de temps à la fabrication de produits plus conformes aux exigences du client. Cependant, les configurateurs sont souvent codés de manière ad hoc, les règles de configuration étant éparpillées et mélangées avec le code de l'interface graphique. Des règles sont parfois mal implémentées voire oubliées. Dès lors, des problèmes de flexibilité, d'exactitude et de maintenance apparaissent.

Dans ce mémoire, afin de produire de meilleurs configurateurs, nous proposons une solution basée sur les "Feature Models" où les *features* représentent les options d'un configurateur. Ces modèles sont utilisés à la fois pour exprimer les options et les règles de configuration grâce aux contraintes entre *features* qu'il est possible de définir et comme artefacts permettant la génération d'une interface graphique dans un langage d'implémentation. Un deuxième modèle de "style", proche du CSS, est utilisé pour rendre la génération plus flexible et produire des résultats plus proches des exigences du concepteur. L'architecture d'un tel configurateur est de type Modèle-Vue-Contrôleur afin d'assurer la séparation des préoccupations et ainsi l'exactitude des règles et une maintenance plus aisée.

Mots clés : Configuration, Ligne de produits, Feature Model, Interface graphique, Génération

Abstract : Over the past several years, product configuration has become a major issue for most companies to satisfy their customers and thus gain market share. The number of Web configurators is substantial and advantages are numerous, from time saving to manufacturing of products more in line with customer requirements. However, configurators are developed in an ad hoc manner, configuration rules are scattered and mixed with the GUI code. Therefore, several issues are raised in terms of flexibility, correctness and maintenance.

In this thesis, we propose a solution based on "Feature Models" where *features* represent configuration options. These models are used both for expressing options and configuration rules thanks to the constraints which can be defined between *features* and as artefacts to generate a GUI in an implementation language. A second "beautification" model, close to CSS, is used to make the generation more flexible and produce results closer to the designer requirements. The architecture of such configurator is Model-View-Controller-like to ensure the separation of concerns thus rules correctness and easier maintenance.

Keywords : Configuration, Product Line, Feature Model, Graphical User Interface, Generation

Avant-propos

Le travail présenté dans ce mémoire est le résultat d'un stage effectué à la faculté informatique de l'Université de Namur dans le cadre d'un projet de l'équipe PReCISE spécialisée notamment dans le domaine des lignes de produits et modèles de variabilité. J'ai été suivi par Quentin Boucher et Gilles Perrouin, deux membres de l'équipe et supervisé par le Professeur Patrick Heymans.

Tout d'abord, je souhaiterais remercier Quentin Boucher, mon co-promoteur, pour son suivi régulier tout au long du stage et de la rédaction de ce mémoire. Ses nombreux conseils m'ont énormément aidé. Mes remerciements vont également à Gilles Perrouin qui m'a lui aussi promulgué quelques conseils, notamment sur le langage Acceleo et la transformation de modèles. Merci également à Patrick Heymans qui m'a permis d'effectuer ce stage très intéressant et instructif. Enfin, je tiens à remercier mes parents pour leur soutien durant toutes mes études et en particularité ma mère pour le temps qu'elle a accordé à la lecture de ce mémoire et pour ses conseils en matière d'orthographe.

Sommaire

Table des figures

1 Introduction

1.1 Contexte et problématique

Dans un environnement de plus en plus compétitif et mondialisé, la personnalisation de produits est devenu un enjeu majeur pour la plupart des entreprises afin de satisfaire au mieux leurs clients et de gagner des parts de marché. Les entreprises proposent dès lors des configurateurs afin d'aider le client à décider des caractéristiques qu'il souhaite pour ses produits [40].

L'utilisation de plus en plus importante d'Internet a également permis l'explosion du nombre de configurateurs disponibles sur le Web. Pour preuve, [21] répertoriait, au mois de juillet 2013, 902 entrées réparties dans 16 domaines, allant du domaine de la santé et des produits de beauté jusqu'aux équipements sportifs et autres produits plus exotiques. Le lecteur aura probablement lui-même déjà rencontré et utilisé de tels configurateurs, que ce soit pour configurer son nouveau PC ou sa nouvelle voiture.

Ces configurateurs ne sont pas uniquement des interfaces graphiques permettant à l'utilisateur de configurer ses produits de façon complètement libre. Ils définissent également des règles de configuration, des contraintes. Par exemple, il est possible que le choix de telle option rende une autre option interdite à cause d'un problème de compatibilité.

En théorie, la configuration de produits a plusieurs avantages, allant du gain de temps à la fois pour le client et l'entreprise à une meilleure satisfaction du client lorsque le produit correspond exactement à ses besoins ou souhaits. En effet, si un configurateur fournit une visualisation en temps réel du produit, le client pourra voir à quoi ressemblera le produit final pendant qu'il le conçoit. Il est ainsi directement lié au processus de conception du produit [21].

Cependant, des configurateurs mal conçus, pas fiables peuvent produire l'effet inverse. En effet, si, au final, le produit ne correspond pas à ce que l'utilisateur a demandé, le sentiment de déception sera sans doute encore plus grand que pour un produit "normal", non personnalisé. Il est donc primordial que les règles de configuration soient correctement implémentées par ceux-ci afin d'éviter par exemple, des activations et désactivations d'options non conformes à ces règles.

Malheureusement, après avoir observé plusieurs cas dans l'industrie [40], l'équipe PReCISE de l'Université de Namur a remarqué que cela n'était pas toujours le cas. En effet, ces configurateurs sont implémentés de manière ad hoc. Par exemple, les contraintes impliquant diverses options et leur propagation sont codées à la main et mélangées avec le code de l'interface graphique. Les règles, parfois en grand nombre, sont éparpillées dans le code, dans plusieurs fichiers, ce qui rend la maintenance difficile. De plus, certaines contraintes sont parfois oubliées ou mal gérées, résultant en des erreurs d'affichage pendant la manipulation du configurateur par l'utilisateur et à un résultat final non conforme aux

désirs de ce dernier. L'ajout, la modification ou la suppression de contraintes peuvent également générer de nouvelles erreurs.

Les partenaires de l'équipe dont les configurateurs ont été analysés ont eux-même reconnu ne pas garantir l'exactitude et l'efficacité des opérations de raisonnement [40]. Il est donc nécessaire de proposer des solutions plus fiables, maintenables et flexibles.

Le problème est donc bien réel et la section suivante va introduire la solution que nous proposons.

1.2 Solution proposée

Dans ce mémoire, nous allons proposer une solution qui se base sur 2 principes majeurs : une architecture de configurateur de type Modèle-Vue-Contrôleur permettant la séparation des préoccupations et une approche d'ingénierie dirigée par les modèles pour générer la "Vue", c'est-à-dire les différents "widgets" représentant les options d'un configurateur.

Notre solution se base également sur les *Feature Models* (FMs) qui modélisent la variabilité d'une ligne de produits, ensemble de produits ayant des caractéristiques communes et d'autres variables. En effet, la sélection d'options d'un produit particulier avec un configurateur revient à configurer un FM où les *features* correspondent aux options et où les contraintes, également exprimées dans ce modèle, correspondent à l'activation et la désactivation d'options par rapport à la sélection ou la désélection d'autres options [41]. Ainsi, chaque spécification de produit authorisée par le configurateur correspond à une combinaison valide de *features* du FM [40]. Le FM a donc une double utilité. Premièrement, le raisonnement et la propagation se fera sur base des contraintes définies par le FM. Ainsi, les contraintes sont toutes présentes dans ce modèle, ne sont pas mélangées au code de l'interface graphique ce qui assure la séparation des préoccupations. Deuxièmement, il va servir à générer les éléments de présentation, les options configurables.

Ainsi, si nous revenons sur l'architecture Modèle-Vue-Contrôleur, la "Vue" sera obtenue grâce à une génération des "widgets" à partir d'un FM. Le contrôleur et la partie serveur vont eux avoir pour objectif de s'occuper de la logique cachée. En effet, le contrôleur aura pour tâche de gérer les actions de l'utilisateur et d'effectuer la propagation des contraintes tandis que la partie serveur sera chargée de maintenir une version du FM dont les contraintes sont à tout moment respectées. À chaque requête du contrôleur, suite à une action de l'utilisateur sur l'interface graphique, elle sera chargée de retourner les changements à effectuer en fonction des contraintes exprimées dans ce FM. Afin de proposer une solution la plus efficace possible, nous souhaitons créer un contrôleur et une partie serveur génériques, compatibles avec toutes les interfaces graphiques générées par l'outil.

En conséquence, le gain de temps lors du développement d'un configurateur devrait être conséquent. En effet, l'implémentation d'un configurateur "from scratch" requiert beaucoup de temps, surtout pour gérer la propagation d'un

grand nombre de contraintes. En appliquant la solution proposée, au lieu d'implémenter complètement un configurateur particulier à chaque nouvelle demande d'un client, il est possible de collaborer avec ce dernier pour créer un FM représentant sa ligne de produits. Cela ne requiert pas de grandes connaissances en informatique et ne demande certainement pas autant de temps que l'implémentation complète d'un configurateur. Une fois ce modèle créé, il ne resterait plus qu'à générer la "Vue", c'est-à-dire l'aspect présentation de son interface graphique. De plus, si le client souhaite un jour modifier certaines contraintes ou ajouter de nouveaux éléments, une simple modification du modèle et une éventuelle nouvelle génération seraient suffisantes pour effectuer la mise-à-jour. Encore une fois, le gain de temps serait important.

L'objectif de ce travail est donc de créer un outil de génération d'interfaces utilisateur de configurateur qui gère les différentes constructions définies par le FM et un modèle de "style". Ce deuxième modèle sert à rendre la génération plus flexible en proposant au concepteur différentes alternatives concernant la génération des éléments du FM. En outre, comme l'aspect dynamique d'un configurateur n'est pas à négliger, un deuxième objectif est de créer le contrôleur et la partie serveur de l'architecture. Le premier s'occupera de relayer les changements effectués par l'utilisateur au serveur et de propager les résultats retournés par ce dernier en conséquence des contraintes définies dans le FM. Le second va, à chaque changement, calculer les nouvelles valeurs des éléments du FM en fonction des contraintes exprimées par le modèle et retourner le résultat à propager.

Afin de réaliser cette solution, plusieurs choix doivent être faits sur base de diverses recherches. Ainsi, il faut définir :

1. Le type et le langage de transformation à utiliser.
2. Le ou les modèles à fournir en entrée au programme de transformation.
3. La technologie/le langage cible.

1.3 Structure du mémoire

Tout d'abord, la Partie I sera consacrée à présenter les différents concepts utiles à la compréhension de la solution proposée et qui permettront de fournir une réponse aux trois choix évoqués dans la section précédente. En premier lieu, le Chapitre 2 traitera des configurateurs en détaillant leurs avantages et défauts actuels. En second lieu, le Chapitre 3 abordera l'ingénierie dirigée par les modèles, une approche d'ingénierie basée sur la transformation de modèles permettant de générer au final du code. Dans le Chapitre 4, nous ferons le lien avec les lignes de produits et les FMs qui permettent de les modéliser. Ensuite, le Chapitre 5 présentera deux langages conçus par l'équipe PReCISE : TVL et FCSS. Le premier est un langage de FMs et le second, un langage qui permettra de rendre la génération plus flexible en proposant différentes alternatives pour la génération des divers éléments définis dans le modèle TVL correspondant. Dans le Chapitre 6, nous verrons les technologies cibles étudiées afin de choisir celle qui est la mieux adaptée. Enfin, le Chapitre 7 abordera quelques travaux liés à

la génération d'interfaces graphiques à partir de modèles et à la configuration basée sur les FMs.

La Partie II présentera les différentes contributions de ce mémoire. Premièrement, le Chapitre 8 traitera de l'architecture des configurateurs conçus avec notre approche. Elle est basée sur la séparation des préoccupations. Deuxièmement, nous aborderons la contribution principale qu'est le générateur d'interface graphique dans le Chapitre 9. Troisièmement, dans le Chapitre 10, nous parlerons des deux autres contributions de ce mémoire qui viennent compléter l'architecture d'un configurateur : le contrôleur et la partie serveur.

En dernier lieu, la Partie III évaluera la solution et présentera les conclusions. Deux études de cas seront présentées dans le Chapitre 11. Elles feront office de preuve du concept. Dans le Chapitre 12, nous aborderons, d'une part, les limitations actuelles de la solution proposée et d'autre part, les améliorations à lui apporter. Enfin, les conclusions seront abordées dans le Chapitre 13.

Première partie

Etat de l'art

2 Configurateurs

2.1 Présentation

"Un configurateur de produits est un outil qui supporte le processus de configuration de produits de façon à ce que toutes les règles de conception et de configuration exprimées dans un modèle de configuration de produits sont garanties d'être satisfaites. Le configurateur simplifie le processus de fabrication en assurant que tous les produits commandés peuvent être construits. Les outils configurateurs interactifs peuvent supporter une personnalisation rapide et flexible en fournissant de l'information correcte immédiatement sur les combinaisons d'options disponibles." [52]

Ainsi, un configurateur de produits permet à un utilisateur de personnaliser le produit qu'il souhaite parmi la gamme de produits possibles. Cette gamme de produits est en fait une ligne de produits, c'est-à-dire un ensembles de produits qui possèdent des caractéristiques communes et d'autres variables. Elles seront abordées dans le Chapitre 4. Un configurateur permet donc de sélectionner parmi des listes d'options préconçues, chaque option correspondant à une valeur possible d'une des caractéristiques variables de cette ligne. Ces caractéristiques peuvent aussi bien être physiques, visibles comme par exemple la couleur d'un téléphone portable, qu'intangibles comme le nombre d'années de garantie.

Un configurateur dispose d'une interface graphique, il doit donc répondre aux critères des interfaces utilisateurs en général, comme le critère d'utilisabilité. Comme pour toute application disposant d'une interface utilisateur, la facilité d'utilisation est primordiale. En dehors de ce critère, les principes généraux d'ergonomie d'une interface utilisateur doivent être pris en compte [37]. Parmi ceux-ci, nous pouvons citer la cohérence (interne et externe), l'adaptabilité en fonction du type d'utilisateur (lambda, expert), le feedback, la simplicité et la concision, l'aide à l'utilisateur, etc. L'utilisateur, le consommateur, doit avoir une place prépondérante lors de la conception de ces configurateurs. Par exemple, selon Joseph Pine II : "une des plus grosses erreurs que font les entreprises permettant la personnalisation de masse est de submerger le consommateur avec trop de choix" [21].

Au niveau purement visuel, un configurateur peut prendre la forme d'un simple formulaire sur une seule fenêtre/page ou d'une interface divisée en vues, onglets ou avec menu. Il peut aussi être constitué d'une suite de fenêtres avec possibilité de retour en arrière, passage d'une fenêtre à une autre en fonction de certains choix de l'utilisateur, la séquence de fenêtres étant déterminée par un workflow [41].

En dehors de l'interface graphique, un configurateur répond à une logique cachée. L'utilisateur est rarement libre de sélectionner ce qu'il souhaite, la personnalisation a des limites, il y a des contraintes. Ainsi, il est impératif que cette logique, ces règles de configurations soient bien implémentées.

Les configurateurs sont très répandus et utilisés dans des domaines variés. Pour preuve, [21] répertoriait, au mois de juillet 2013, 902 entrées réparties dans 16 domaines, allant du domaine de la santé et des produits de beauté jusqu'aux équipements sportifs. Les configurateurs sont utilisés aussi bien en "Business to Business" (B2B), le commerce entre entreprises, qu'en "Business to Consumer" (B2C), le commerce entre entreprises et consommateurs. Les utilisateurs des configurateurs peuvent être des employés de l'entreprise ou les consommateurs eux-même.

L'objectif n'est en général pas le même en fonction de son utilisation en B2B ou B2C. En B2C, l'objectif sera de permettre au client de personnaliser le produit selon ses besoins ou désirs. En B2B, par contre, ils sont plutôt utilisés pour supporter les ventes et améliorer l'efficacité de la production [21].

2.2 Avantages et défauts

L'utilisation de configurateurs comporte plusieurs avantages différents selon qu'ils sont utilisés dans un cadre B2B ou B2C [21].

Dans un cadre B2B, nous pouvons citer le fait qu'il est possible de limiter la surproduction étant donné que la production dépend de la demande réelle et non du besoin de garder des produits en stock. Il n'y a dès lors plus besoin de proposer des réductions pour écouler le stock en surplus.

De plus, les entreprises qui produisent des biens complexes vont sans doute passer beaucoup de temps et d'argent pour répondre aux demandes des clients. Une expertise et des enquêtes internes à travers différents départements pourraient être nécessaires pour obtenir des informations. Au contraire, un configurateur rend disponible pour les membres du département des ventes, toute l'information nécessaire pour des offres techniquement correctes et précisément calculées, ce qui leur permet de travailler de manière indépendante. Le service client est fourni avec une meilleure uniformité sur toute la chaîne de distribution, à un moindre coût pour l'entreprise.

Les configurateurs permettent également une amélioration du service client étant donné que le configurateur peut proposer une visualisation "en temps réel" du produit que le client personnalise mais également une évolution du prix en fonction des options sélectionnées ou non. Il y a donc moins de chance que ce dernier soit mécontent du produit final.

Un configurateur permet d'examiner et tester la faisabilité des désirs des consommateurs à une des premières étapes du processus de vente. Une fois qu'une commande est envoyée, les listes de matériaux, les plans de travail et la plupart des documents du projet peuvent être générés automatiquement. Les temps de production et de livraison sont alors raccourcis et moins d'argent est dépensé pour gérer les plaintes et les problèmes de garanties.

Un configurateur assure que les membres du département des ventes et les partenaires de distribution ont toujours accès aux plus récentes informations. Lorsque les réseaux locaux et Internet ne sont pas disponibles immédiatement, le configurateur pourrait être utilisé hors-ligne. Les représentants des ventes qui

voyagent peuvent travailler à partir de leur ordinateur portable et synchronisent par la suite toutes leurs données avec la base de données centrale.

Dans le cadre du B2C, il y a également plusieurs avantages. Par exemple, le fait d'adapter les produits aux besoins spécifiques des consommateurs permet aux entreprises de se distinguer de celles qui font de la production de masse. Les produits personnalisés sont souvent mieux perçus par les consommateurs, plus attractifs et incomparables aux biens produits en masse. Cela permet aux entreprises d'éviter la compétition du marché qui essaie de vendre au prix le plus bas.

De même que pour le B2B, ils permettent de réduire la surproduction.

Les configurateurs permettent également une meilleure connaissance des besoins des utilisateurs en les enregistrant directement et précisément. De nouvelles opportunités peuvent être découvertes rapidement et les produits adaptés en fonction. Les entreprises utilisant la personnalisation massive - dont l'objectif est de fournir des biens et services qui satisfassent les besoins et désirs des clients individuels avec une efficacité comparable à la production de masse (en matière de prix notamment) [21] - mais également de la production de masse conventionnelle peuvent transférer et employer leur connaissance des besoins des utilisateurs.

Ces informations récoltées vont également être en faveur d'une relation durable avec le client puisqu'elles permettent d'adresser les clients individuellement et peuvent être réutilisées pour des commandes futures. Une deuxième configuration de la part du client peut alors être plus rapide que la première. En effet, il suffit de récupérer les options qu'il a déjà sélectionnées par le passé, il y a des chances qu'il fasse globalement les mêmes choix. Cela incite le client à rester loyal à cette entreprise plutôt que de changer d'entreprise et devoir refaire tout le processus de configuration depuis le début.

Enfin, dans le cas d'une interaction directe entre le constructeur et les clients sur le Web, les aspects créatifs de la configuration représente une expérience spéciale pour beaucoup de consommateurs. Le processus va dès lors lier émotionnellement les clients avec un produit en leur permettant de devenir eux-même les concepteurs de ce produit. Le client aura également une perception positive du processus de vente ce qui affecte directement sa satisfaction vis-à-vis du produit acquis.

En dehors de tout cela, le fait que les options et accesssoires sont inclus dans les processus de commandes incite les internautes à les ajouter naturellement au panier alors qu'ils ne l'auraient peut-être pas fait autrement. Dans le cas où la commande concerne des produits nécessitant l'élaboration d'un devis, les clients bénéficient d'un gain de temps considérable puisque le prix est calculé directement par le configurateur. Les employés, n'ayant plus à répondre à autant d'appels téléphoniques, à effectuer ces devis, vont également gagner du temps et pouvoir se concentrer sur d'autres tâches [26].

De plus, grâce aux contraintes implémentées, le configurateur limite lui-même les possibilités de l'utilisateur et ne lui permet pas de commander un produit non réalisable, dont différentes caractéristiques ne sont pas compatibles. Les risques de retour du produit sont donc fortement diminués. De plus, un

produit peut éventuellement être configuré de milliers de façons différentes. Il est donc virtuellement impossible pour l'entreprise de garder la trace de toutes ces combinaisons. Sans configurateur, une entreprise va sans doute mettre en vente certaines configurations "standard" qu'elle pense vendre le mieux. Avec un configurateur, le client a accès directement à toutes les configurations possibles d'un produit [19].

Comme nous l'avons dit précédemment, les configurateurs répondent à certaines contraintes que sont les règles de configuration qui doivent être implémentées correctement. Cependant, à l'heure actuelle, il s'avère que ce n'est pas toujours le cas avec les configurateurs développés "à la main". En effet, comme cela a été abordé dans l'introduction de ce mémoire, certains configurateurs actuels ont leurs règles de configuration "hard-codées" et mélangées avec le code de l'interface graphique. Il en résulte des problèmes pouvant provenir d'oublis ou de mauvaise gestion de contraintes, celles-ci pouvant être en grand nombre et impliquer potentiellement plusieurs éléments de l'interface. La maintenance des configurateurs est dès lors difficile étant donné qu'il est difficile de localiser telles ou telles contraintes perdues dans du code d'interface graphique en général imposant, aussi bien l'ajout que la modification ou la suppression de contraintes devient difficile et peut mener à d'autres erreurs. Le développeur pourrait par exemple introduire des contraintes non compatibles avec celles déjà existantes.

Afin d'illustrer cet état de fait, nous pouvons prendre deux cas industriels observés par l'équipe PReCISE [40]. Dans le premier cas, le configurateur sert à paramétriser le déploiement et le comportement en temps réel de systèmes de communication. Plusieurs problèmes ont été relevés. Premièrement, les opérations de raisonnement consistent simplement en des instructions *if-then-else*. Deuxièmement, de nombreuses portions de codes ont été manuellement dupliquées pour gérer les contraintes et appliquer les mises-à-jour de l'interface graphique. Troisièmement, les opérations de raisonnement sont éparpillées dans différents endroits, différents fichiers. Enfin, les contraintes sont documentées via des commentaires en langage naturel.

Le second cas concerne un outil permettant notamment de configurer des options d'impression. L'équipe a remarqué des incohérences entre la prévisualisation et le résultat final mais également, dans de rares cas, l'impossibilité d'imprimer le document avec l'imprimante sélectionnée. Cela provient du fait que certaines contraintes imposées par les imprimantes ne sont pas implémentées, principalement en raison du temps requis et de la complexité du travail.

Ces deux cas illustrent bien l'intérêt de produire des configurateurs plus fiables, maintenables et flexibles.

Il apparait donc utile de trouver une solution à ce problème, de concevoir des configurateurs dont les règles de configuration sont clairement séparées de la présentation. Pour cela, nous verrons qu'il est possible d'utiliser une approche orientée sur les transformations de modèles, à partir de *feature models* modélisant des lignes de produits. Ces différents concepts sont abordés dans les chapitres suivants.

2.3 Quelques exemples de configurateurs

La Figure 2.1 correspond au configurateur du constructeur automobile Mazda. Cette page est accessible une fois que l'utilisateur a sélectionné le modèle de voiture qu'il souhaite acheter. Il est alors possible de personnaliser dans une certaine mesure sa future voiture. Sur la Figure 2.1, nous voyons que l'utilisateur a choisi une couleur et que le configurateur montre le résultat en temps réel en fonction de ce choix. Ce configurateur est un configurateur à onglets, la page se mettant à jour lorsque l'utilisateur choisit une option spécifique. Il a un comportement dynamique avancé avec des animations, une visualisation en temps réel, etc.

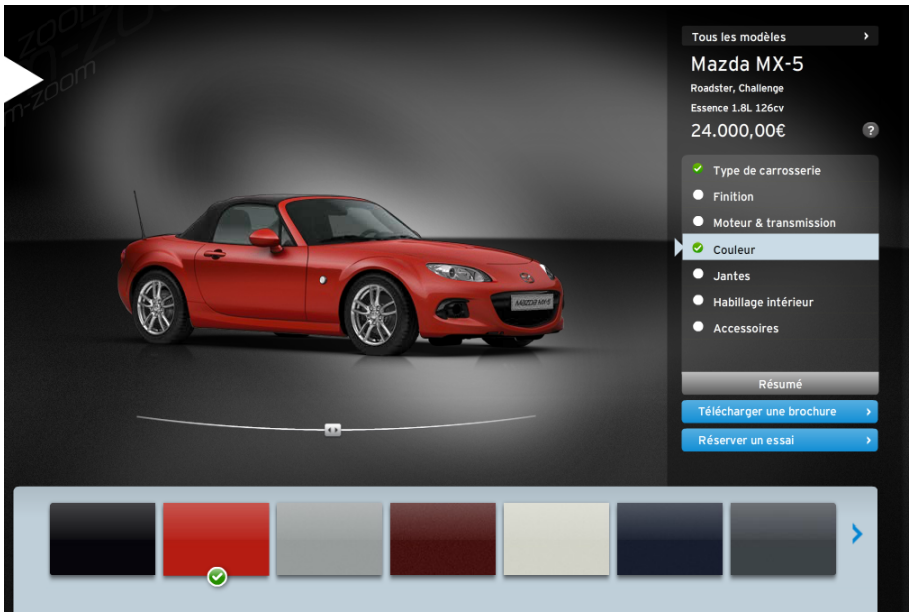

FIGURE 2.1 – Configurateur Mazda (`http://fr.mazda.be/`)

La Figure 2.2 montre une partie d'un configurateur Dell, l'affichage des autres options étant similaire, tandis que la Figure 2.3 en montre un autre provenant du site US. Les deux configurateurs sont visuellements différents puisque le premier prend la forme d'un formulaire sur une seule page tandis que le second est un configurateur à onglets. Dans les deux cas, il apparait lorsque l'utilisateur a sélectionné le modèle d'ordinateur qu'il souhaite acheter. Pour le premier, c'est une page "popup" qui s'affiche tandis que pour le second, l'utilisateur est redirigé vers une autre page. Il peut alors sélectionner parmi différentes listes d'options qui font varier le prix du produit en temps réel.

Ces captures d'écrans montrent des configurateurs visuellements différents. La solution proposée se concentrera sur les configurateurs de type formulaire, plus simples que les autres. Elle devra être améliorée afin de pouvoir générer

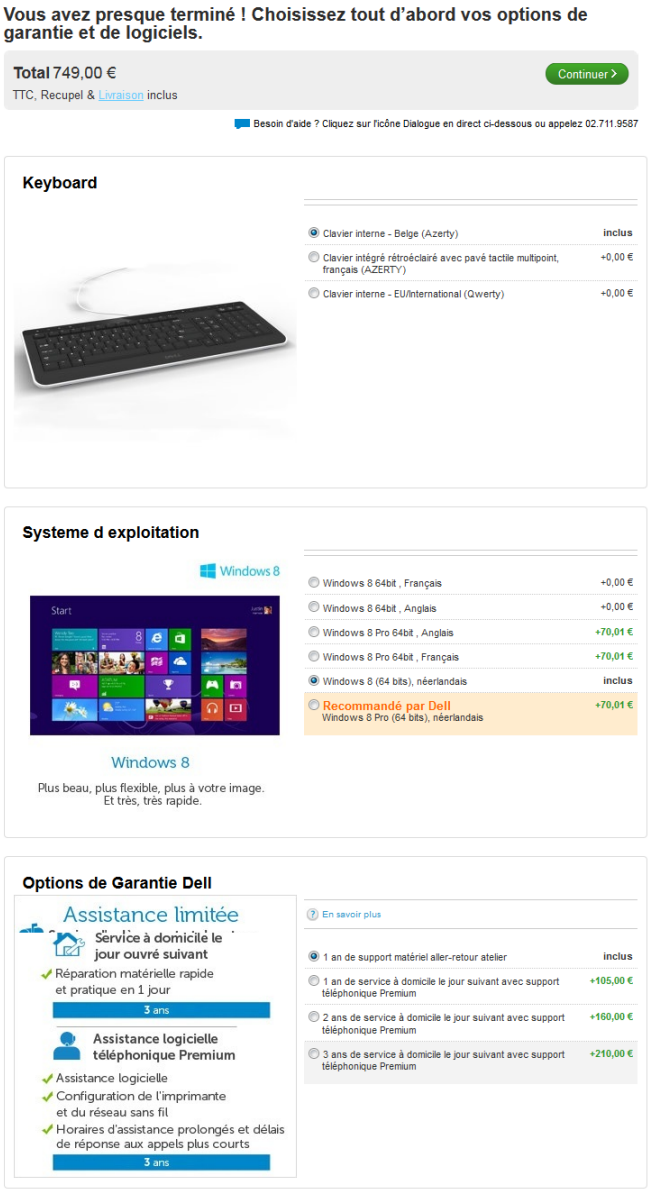

FIGURE 2.2 – Configurateur Dell (`http://www.dell.be/`)

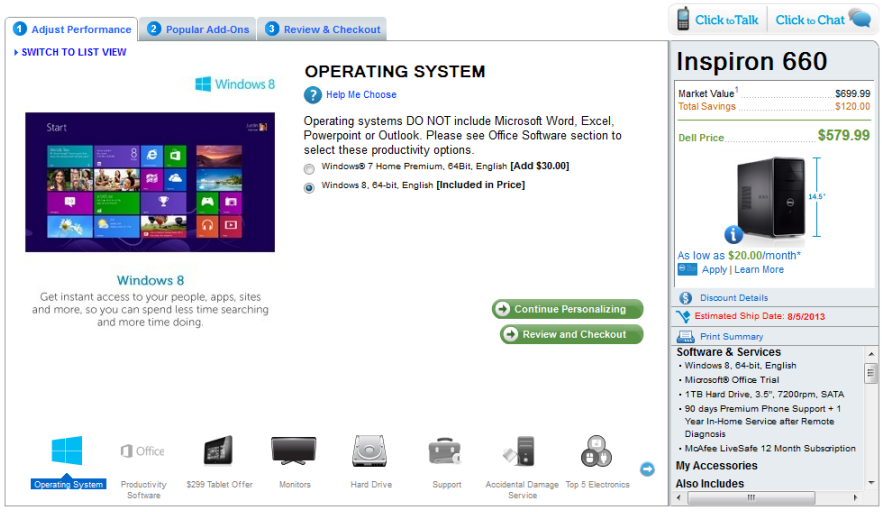

FIGURE 2.3 – Configurateur Dell US (`http://www.dell.com/`)

d'autres types de configurateurs.

3 Ingénierie dirigée par les modèles

Ce chapitre a pour but de présenter l'ingénierie dirigée par les modèles et les transformations de modèles, techniques utilisées pour développer le générateur présenté dans le Chapitre 9.

3.1 Contexte et présentation

L'ingénierie du logiciel, ou génie logiciel, est apparue suite à la crise du logiciel de 1968. La complexité grandissante des systèmes à développer liée au fait que les développeurs travaillaient comme des artisans, sans réelle méthode de travail faisait que les projets n'étaient que très rarement terminés dans les temps, dépassaient le budget alloué voire étaient abandonnés [51]. Les impacts en terme d'argent et de temps étaient donc conséquents. Plusieurs études ont été réalisées dont celle du Standish Group [1] datant de 1995 qui montre cet état de fait [83]. Avec l'ingénierie sont apparus les modèles permettant de représenter les différents aspects d'un système en faisant abstraction d'autres aspects. Dans [45], un modèle est défini comme "une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé". Différents langages de modélisation sont actuellement très répandus, parmi eux nous pouvons notamment citer UML [70] [71], qui comprend un ensemble de modèles (diagrammes de classes, de cas d'utilisations, d'activités, de séquences, etc.), BPMN [66], les modèles entités-associations [80], etc.

Néanmoins, la complexité des logiciels n'a cessé d'augmenter depuis la crise, la puissance du matériel informatique est incomparable à celle du matériel existant pendant la crise, Internet est apparu, le nombre d'utilisateurs a explosé et la plupart ne sont pas des experts, l'interactivité avec ces utilisateurs est de plus en plus importante, les systèmes ne fonctionnent plus seuls mais sont interconnectés, ils sont interopérables, etc. Encore aujourd'hui, de nombreux projets ne sont pas terminés à temps, sont hors-délais ou sont abandonnés [51].

Dans cette ingénierie du logiciel traditionnelle, les modèles sont utilisés comme moyen de communication entre les différents intervenants dans le cycle de vie d'un logiciel. Ils permettent de décrire le domaine d'application, de séparer les différentes préoccupations en modélisant des aspects différents du système, aux nouveaux venus dans un projet de comprendre le système sans devoir analyser tout le code source, de rendre la maintenance d'un système plus aisée, etc.

L'Ingénierie Dirigée par les Modèles (IDM), plus connue sous le nom de MDE pour "Model Driven Engineering", est une approche d'ingénierie du logiciel qui, comme son nom l'indique, est centrée sur les modèles, utilisés non seulement

1. `http://blog.standishgroup.com/`

comme outils de communication et d'information mais également comme artefacts de premier plan, permettant au final la génération de code [45]. Dans ce type d'approche, ce sont les modèles qui sont le fil conducteur du processus de développement. De ce fait, ce type d'ingénierie requiert des modèles, non plus semi-formels, parfois imprécis voire ambigus, mais bien des modèles écrits dans des langages clairement définis par des méta-modèles et qui doivent s'y conformer. Ces modèles peuvent alors être manipulés par des machines. Selon [45], un méta-modèle est "un modèle qui définit le langage d'expression d'un modèle, c'est-à-dire le langage de modélisation". Ainsi, les deux principes fondamentaux de l'IDM sont la méta-modélisation et la transformation de modèles que nous aborderons plus en détail par la suite.

Un processus typique d'IDM correspond en général à une suite de transformations de modèles afin de produire des modèles de moins en moins abstraits et de plus en plus proches de la plateforme et de la technologie cible jusqu'à arriver à un modèle assez précis pour, idéalement, permettre la génération de code à partir de celui-ci. Idéalement car ce genre de scénario où il est possible de générer tout le code final à partir de modèles est assez utopique en pratique. En général, c'est le code répétitif, simple qui pourra être généré tandis que le code requiérant réflexion, créativité de la part du développeur ne pourra pas être automatisé. Néanmoins, cela réduit le travail souvent pénible et peu gratifiant et permet aux développeurs de se concentrer sur ce qui est plus intéressant, qui propose un challenge à ces derniers [50].

D'autres scénarios de transformations sont également possibles comme la rétro-ingénierie qui correspond au processus inverse de celui expliqué dans le paragraphe précédent ou encore la mise-à-jour de modèles (ajout de nouveaux éléments, modifications, suppressions).

Avec l'acceptation du concept de méta-modèle, de nombreux méta-modèles ont proliféré ce qui a poussé l'Object Management Group (OMG) à définir un cadre général pour l'IDM : l'architecture dirigée par les modèles.

3.2 Architecture dirigée par les modèles

Afin de promulguer les bonnes pratiques de modélisation et exploiter pleinement les avantages des modèles, l'organisme de standardisation Object Management Group (OMG) a défini l'architecture dirigée par les modèles, en anglais la "Model Driven Architecture" (MDA). Cela a également apporté un cadre général pour la description des méta-modèles, qui avaient commencé à proliférer dans divers domaines une fois accepté le concept de méta-modèle comme langage de description de modèles. Étant donné que l'IDM met largement les modèles en avant, il est logique que ce soit également un modèle qui permette de définir les méta-modèles, il s'agit du méta-méta-modèle [45]. L'OMG a spécifié le standard Meta-Object Facility (MOF [68]). Étant donné qu'il s'agit d'un modèle, il doit également être défini à partir d'un langage de modélisation. En fait, il est capable de s'auto-décrire. Si ce n'était pas le cas, un modèle de plus haut niveau serait nécessaire et ainsi de suite. Dans [45], un méta-méta-modèle est défini

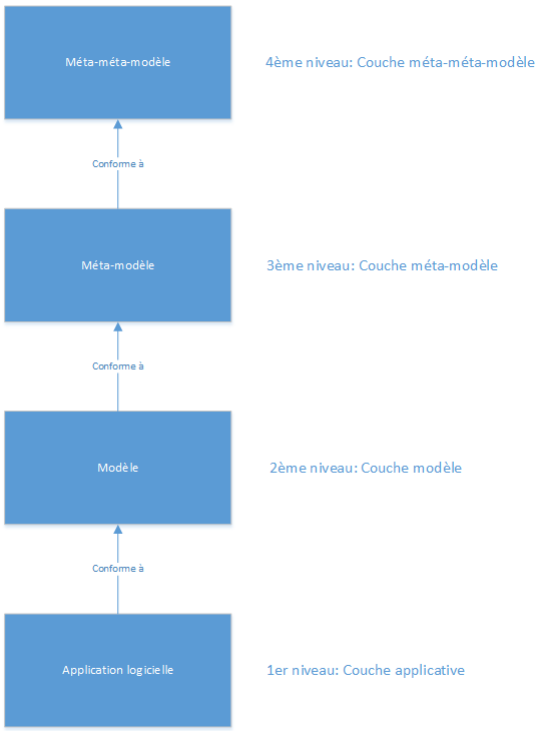

FIGURE 3.1 – Architecture 4 couches

comme "un modèle qui décrit un langage de métamodélisation c.-à-d. les éléments de modélisation nécessaires à la définition des langages de modélisation. Il a de plus la capacité de se décrire lui-même".

L'approche MDA et l'IDM en général se basent sur une architecture de modélisation en 4 couches présentée à la Figure 3.1 qui est aussi décrite sous forme de pyramide dans la littérature [45].

La première couche, en bas, représente le code source, les programmes ou encore les "objets du monde réel". Ceux-ci doivent être conformes aux modèles définis dans la couche supérieure. Ainsi, idéalement, un programme Java aura une structure de classes conforme au diagramme de classes UML défini en amont. La troisième couche est la couche des méta-modèles. Les modèles de la couche inférieure doivent être conformes à leur méta-modèle. Pour garder le même exemple, le diagramme de classes UML modélisant le domaine du programme doit être conforme à son méta-modèle, lui-même exprimé en diagramme de classes UML. Enfin, tous les méta-modèles de la troisième couche doivent être conformes à un même méta-méta-modèle. En d'autres mots, un modèle de la couche 2 spécifie l'application, ou les objets du monde réel, en 1ère couche, un méta-modèle de la couche 3 spécifie une série de modèles de la couche 2 et le

méta-méta-modèle en couche 4 spécifie les méta-modèles de la couche 3. Un méta-méta-modèle spécifie donc un langage pour les méta-modèles de la couche inférieure mais il doit également pouvoir se spécifier lui-même. Pour rappel, le méta-méta-modèle qu'a défini l'OMG pour la MDA s'appelle le MOF. Cette architecture avec MOF au sommet est une architecture fermée et stricte. Fermée car le méta-méta-modèle MOF est conforme à lui-même, il est défini par lui-même, il n'a pas besoin d'être défini par un modèle d'une couche supérieure, encore plus abstraite. Stricte car chaque élément d'un modèle de chaque couche est conforme à un élément d'un modèle de la couche supérieure [3].

Le processus général MDA consiste à partir d'un "Computation Independent Model" (CIM) qui est un modèle métier indépendant de toute considération informatique, spécifiant des concepts du domaine d'application. Une fois qu'il est créé, il est transformé en un modèle indépendant de la plateforme sur laquelle devra tourner la solution finale, le "Platform Independent Model" (PIM). Ce modèle est ensuite transformé en un modèle cette fois spécifique à la plateforme visée, le "Platform Specific Model" (PSM). Les PSMs peuvent être définis grâce à des langages spécifiques à un domaine, en anglais "Domain Specific Languages" (DSLs), ou grâce à des langages généraux comme Java, C, C++, etc. Enfin, le PSM obtenu peut alors servir à la génération du code final. Il est à noter qu'en pratique, l'étape de transformation du PIM en PSM est parfois, voire souvent, ignorée [61]. Dans ce cas, le code est généré directement à partir du PIM.

3.3 Transformations de modèles

Les transformations de modèles font partie intégrante de l'IDM. Il existe différents types de transformations, différents langages dédiés avec des caractéristiques spécifiques et le choix de tel langage plutôt qu'un autre va en général dépendre de ce que les développeurs souhaitent générer et de leurs objectifs.

En toute généralité, les transformations de modèles peuvent être exprimées grâce à des langages généraux comme Java ou C++. Néanmoins, les langages dédiés aux transformations de modèles offrent en général des avantages non négligeables par rapport aux langages généraux. Ils ont des abstractions plus puissantes pour les transformations de modèles par rapport aux langages généraux. Étant donné qu'ils sont créés spécifiquement dans ce but, ils sont plus efficaces car ils permettent d'implémenter des règles de transformation de manière bien plus concise et en cachant le déroulement de la transformation derrière une syntaxe "simple". Ces langages sont plus puissants pour les transformations que les langages généraux car ils ne sont conçus que dans ce but [50]. Un autre avantage de ce genre de langages dédiés est qu'il est possible d'écrire des méta-transformations, des transformations de transformations. En effet, pour certains langages de transformations, un programme de transformation est lui-même un modèle conforme au méta-modèle qui définit le langage de transformation. C'est notamment le cas du langage ATL [55]. La Figure 3.2 illustre cela.

Les transformations de modèles peuvent être classifiées en trois grandes catégories :

1. M2M (Model to Model) : transformations de modèle(s) source(s) en modèle(s) cible(s). L'OMG a défini le standard QVT [67] pour donner un cadre normatif à ce type de transformations. QVT définit un ensemble de langages proposant différents paradigmes des transformations. Ces paradigmes seront présentés par la suite. Le méta-modèle de QVT est conforme au MOF et OCL [72], langage de contraintes permettant de calculer des expressions sur des modèles, est utilisé pour la navigation dans les modèles [45]. Exemples de langages M2M : ATL [55], QVTd [10], QVTo [9],etc.
2. M2T (Model to Text) : transformations de modèle(s) en texte c'est-à-dire en général du code source ou de la documentation. L'OMG a également défini un standard pour ce type de transformations, il s'agit de MOFM2T [65]. Exemples de langages M2T : Acceleo [22], JET [23], etc.
3. T2M (Text to Model) : transformations de texte (code source ou documentation) en modèle(s). Ce type de transformations est sans doute le moins connu et le moins utilisé. Son principal intérêt réside dans la rétro-ingénierie, dans le but de générer un modèle à partir de code ou de documentation. Ce type de transformations n'est pas vraiment intéressant dans le cadre de ce projet et n'est donc pas développé par la suite.

3.3.1 Transformations "Model to Model"

Les transformations M2M peuvent être catégorisées selon différents critères. Selon ceux-ci, les objectifs derrière l'utilisation des transformations peuvent être différents. Dans une approche MDA, c'est ce genre de transformation qui est utilisé pour dériver des modèles à partir d'un modèle initial avant la dernière transformation en code source, qui est, elle, de type M2T.

La Figure 3.2 schématise une telle transformation avec un seul modèle source et un seul modèle cible. Elle est basée sur des figures utilisées dans des travaux tels que [45] et [55].

Les modèles source et cible doivent être conformes à leur propre méta-modèle. Le programme de transformation est réalisé sur base des méta-modèles source et cible. Un modèle source, conforme à son méta-modèle, est passé en entrée au programme de transformation qui produit un modèle cible, conforme à son méta-modèle. Le programme de transformation est lui-même conforme au méta-modèle définissant le langage. Aussi bien ce méta-modèle que les méta-modèles source et cible sont conformes au méta-méta-modèle. Cela a pour conséquence qu'un programme de transformation peut tout à fait servir de modèle source et/ou de modèle cible pour une autre transformation. C'est un des avantages d'utiliser des langages dédiés aux transformations plutôt que des langages généraux. Il est possible de réaliser des méta-transformations, des transformations de transformations.

Nous pouvons placer ces transformations dans plusieurs catégories en fonction de certains critères. Le premier concerne les méta-modèles des modèles sources et cibles :

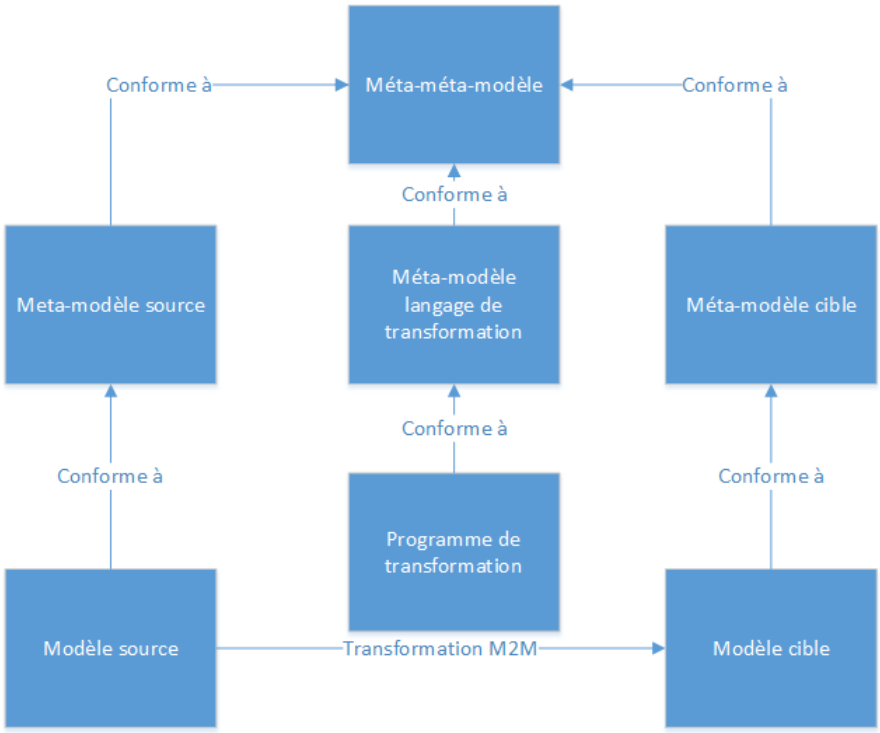

FIGURE 3.2 – Transformation M2M

1. Les transformations endogènes : les modèles source(s) et cible(s) sont tous conformes au même méta-modèle.

   Les transformations endogènes sont en général utilisées pour mettre à jour un modèle et garder une certaine traçabilité de l'évolution d'un logiciel.

   La mise à jour d'un modèle, outre la traçabilité, peut aussi avoir pour objectif de raffiner le modèle, c'est-à-dire d'ajouter par exemple des informations spécifiques à la plateforme visée.

2. Les transformations exogènes : les modèles source(s) et cible(s) sont conformes à des méta-modèles différents.

   Les transformations exogènes consistent donc à générer un modèle cible (ou plusieurs) à partir d'un modèle source différent. La génération d'un diagramme BPMN à partir d'un diagramme d'activités UML en est un exemple.

Nous pouvons également différencier les transformations verticales des transformations horizontales :

1. Les transformations verticales : transformations de modèle(s) source(s) décrit(s) dans un formalisme plus abstrait vers un (ou des) modèle(s) cible(s) décrit(s) dans un formalisme moins abstrait. Une telle transformation implique donc un changement du niveau d'abstraction. Un modèle cible moins abstrait est plus proche de la plateforme d'exécution que le modèle source. C'est ce type de transformations qui a lieu lors du passage du PIM vers le PSM et du PSM vers le code en MDA.
2. Les transformations horizontales : transformations de modèle(s) source(s) vers un (ou des) modèle(s) cible(s) au même niveau d'abstraction.

Combemale [45] illustre ces classes de transformations (endogène/exogène et verticale/horizontale) ainsi que les cas d'utilisations respectifs avec la Figure 3.3. Pour rappel, le PIM est un modèle indépendant de la plateforme cible et le PSM est lui dépendant de cette plateforme. Le passage d'un PIM à un PSM correspond donc bien à une transformation verticale, le second étant moins abstrait que le premier.

Il existe plusieurs paradigmes pour les langages de transformations de modèles [46] :

1. Déclarative : se base sur les "patterns". Cette approche consiste à définir des "patterns" pour ensuite rechercher et transformer chaque élément du (des) modèle(s) source(s), dont la structure est conforme à un même "pattern", en éléments du (des) modèle(s) cible(s). Il s'agit donc d'un "mapping" entre éléments du (des) modèle(s) source(s) et du (des) modèle(s) cible(s). C'est une approche basée sur le description du "quoi" plutôt que du "comment". Le langage QVTd [10] implémente la partie déclarative du standard QVT.
2. Impérative : proche des langages de programmation impératifs. Elle consiste à parcourir le(s) modèle(s) source(s) dans un certain ordre et à générer le(s) modèle(s) cible(s) pendant ce parcours. Cette approche est plus compliquée que la première mais permet de définir toutes les transformations

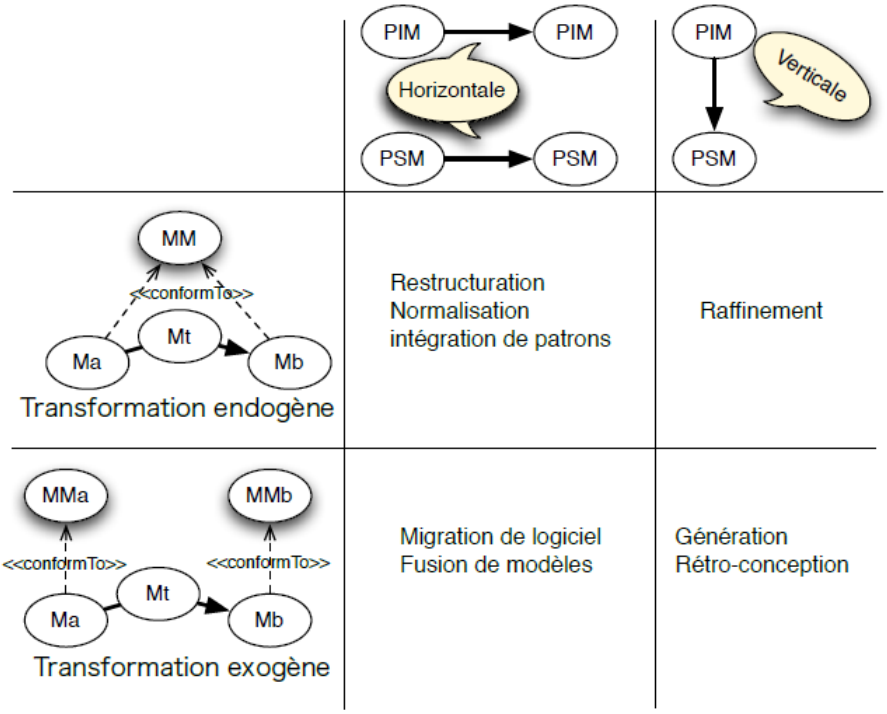

FIGURE 3.3 – Types de transformations et leurs principales utilisations

possibles et imaginables. Le langage QVTo [9] implémente la partie impérative du standard QVT.

3. Hybride : approche qui propose un mélange des approches déclarative et impérative, en proposant des structures déclaratives et d'autres impératives, afin de combiner leurs avantages respectifs. En effet, l'approche déclarative est plus proche de la façon dont les développeurs conçoivent intuitivement une transformation. Cette approche va encoder ces relations et cacher les détails liés à la sélection des éléments sources, l'activation et l'ordre des règles, la gestion de la traçabilité, etc. Des transformations complexes peuvent dès lors être cachées derrière une syntaxe relativement simple. Cependant, il faut parfois utiliser l'approche impérative car certains problèmes sont difficiles à résoudre avec la méthode déclarative [55]. ATL [55] est un exemple de langage hybride.

3.3.2 Transformations "Model to Text"

La Figure 3.4 schématise une transformation M2T avec un seul modèle source en entrée. Elle est dérivée de la Figure 3.2. La seule différence est que le texte généré n'est pas forcémment conforme à un méta-modèle. En effet, rien n'oblige de générer du texte conforme à une syntaxe particulière. Même si cela pourrait être le cas, ce n'est pas illustré sur la Figure 3.4 pour ne pas porter à confusion avec les transformations M2M.

L'explication est la même que pour 3.2 à part le fait que le programme de transformation produit du code ou de la documentation au lieu d'un modèle conforme à un méta-modèle. Il est également théoriquement possible de définir des transformations de transformations, le programme de transformation pouvant servir de modèle source à un autre programme de transformation.

Les transformations M2T peuvent être classées selon deux approches [46] :

1. Approche visiteur
2. Approche basée sur les *templates*

L'approche visiteur est une approche simpliste qui offre un mécanisme de navigation - appelé mécanisme "visiteur" - permettant de traverser la représentation interne d'un modèle et d'écrire le code en tant que flux de texte [46]. Jamda [8] est un exemple d'une telle approche. Il s'agit d'un framework orienté objet composé d'un ensemble de classes pour représenter les modèles UML, d'une API pour manipuler ces modèles et d'un mécanisme "visiteur" pour générer le code [46].

L'approche basée sur les *templates* est beaucoup plus répandue, la majorité des outils MDA supportant cette approche [46]. Un *template* correspond à une règle de transformation. En général, le corps d'un *template* est composé du texte cible et de "méta-code" permettant d'accéder aux éléments du ou des modèles sources et qui seront, à l'exécution, remplacés par les valeurs évaluées.

Dans [46], les règles de transformations sont définies comme étant composées d'une partie gauche et d'une partie droite. La partie gauche concerne le modèle source, la partie droite, le modèle cible. Dans une approche *template*, la partie

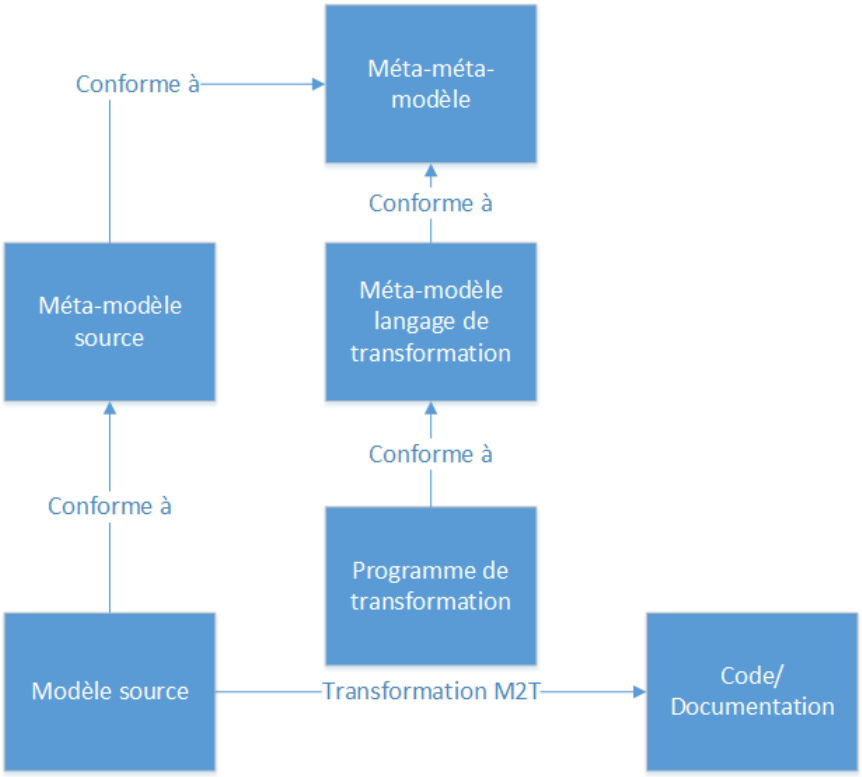

FIGURE 3.4 – Transformation M2T

gauche utilise de la logique exécutable pour accéder au(x) modèle(s) source(s). Celle-ci peut avoir différentes formes. Elle peut être simplement du code Java accédant l'API fournie par la représentation interne du modèle source ou être des requêtes déclaratives en OCL ou XPath par exemple. La partie gauche peut n'être qu'une simple liste de paramètres. La partie droite contient des "patterns" sous forme de chaînes de caractères, le texte généré, et de la logique exécutable.

Les approches *templates* permettent souvent à l'utilisateur de définir la planification interne, c'est-à-dire qu'il est possible d'appeler un *template* à l'intérieur d'un autre.

Comparé à l'approche "visiteur", la structure d'un *template* est plus proche du code à générer. Les *templates* se prêtent au développement itératif car ils peuvent être dérivés d'exemples. Étant donné que les approches *template* produisent du texte, les "patterns" qu'ils contiennent sont non typés et peuvent produire des fragments de code incorrect aussi bien du point de vue syntaxique que sémantique. D'un autre côté, les *templates* textuels sont indépendants du langage cible ce qui simplifie la génération de divers artefacts textuels comme la documentation.

La sous-section suivante présente le langage Acceleo M2T qui utilise l'approche *template*. Comme nous le verrons plus tard, c'est le langage qui a été choisi pour développer le générateur.

3.3.3 Acceleo

Un exemple de langage utilisant l'approche basée sur les *templates* est le langage Acceleo [22] qui est une implémentation du standard MOF Model to Text Transfomation Language (MOFM2T aussi appelé MOFMTL). Outre le fait d'être basé sur un standard, ce langage a aussi pour avantage d'être soutenu par Eclipse qui propose un plugin pour celui-ci comme pour ATL et QVT pour les transformations Model to Model.

Un générateur Acceleo est composé de *templates*. Les *templates* représentent des règles de transformation. Un *template* peut prendre plusieurs éléments de différents méta-modèles en argument. En fait, la définition des *templates* se fait sur base de types définis par des méta-modèles. À l'exécution, ce sont des éléments des modèles conformes à ces types qui seront passés en argument au *template*. Son corps est composé de texte immutable, le code ou la documentation généré(e), et d'expressions OCL. Lorsqu'elles sont appliquées à un modèle précis, ces expressions sont remplacées par le résultat de leur évaluation. Acceleo utilise le langage de contraintes OCL pour naviguer dans les modèles. Une autre construction importante du langage Acceleo sont les *queries*. Elles correspondent à des méthodes qui peuvent prendre des arguments en entrée et retournent un résultat. Elles peuvent être définies récursivement et peuvent également appeler des méthodes Java en jouant le rôle "Java Service Wrapper". Un projet Acceleo peut donc contenir des classes Java dont les méthodes pourront être appelées par le code Acceleo.

Un *template* peut appeler d'autres *templates*, y compris lui-même, le mécanisme de ces appels est déterministe et défini par le développeur. Un appel se

fait sur un élément dont le type est celui du premier argument dans la définition du *template*. Une *query* peut en appeler d'autres et peut être appelée par un *template*, l'inverse n'étant pas vrai.

Les *templates* et *queries* peuvent être regroupés dans des modules, des fichiers d'extension ".mtl". Un module contient une balise "module" qui définit son nom et la liste des méta-modèles utiles pour les *templates* et *queries* du module. Le Code 3.1 montre cette balise.

```
1 [module generateClass('http://www.eclipse.org/emf/2002/Ecore')]
```

Code 3.1 – Exemple de balise module

Comme nous pouvons le voir avec cet exemple, les méta-modèles sont identifiés par une URI. Dans ce cas-ci, les *templates* et *queries* du module sont définis sur des types fournis par le méta-modèle Ecore.

Ce module pourrait par exemple contenir la *query* "getNameClass" (Code 3.2) et le *template* "generateClassName" (Code 3.3), tous deux prenant un argument de type "EClass" défini par le méta-modèle Ecore. La *query* retourne le nom d'un élément de ce type avec la première lettre en majuscule. Si celui-ci n'est pas défini, elle retourne "Not defined". Le *template* prend un élément de ce même type en argument et appelle la *query* "getNameClass" pour générer le nom de la classe ou "Not defined". Comme nous pouvons le voir, une partie du code généré est immutable ("Class name :") tandis que l'autre est une expression qui sera évaluée à l'exécution ("[aClass.getNameClass()/]").

```
1 [query public getNameClass(aClass:EClass):
2 String =
3 if(not aClass.name.oclIsUndefined())
4 then aClass.name.toUpperFirst()
5 else 'Not defined'
6 endif
7 /]
```

Code 3.2 – Exemple de query

```
1 [template public generateClassName(aClass:EClass)]
2 Class name: [aClass.getNameClass()/]
3 [/template]
```

Code 3.3 – Exemple de template

Un module peut également référencer d'autres modules grâce à la balise "import".

De base, un programme Acceleo ne peut prendre qu'un seul modèle source en entrée, sous format XML Metadata Interchange (XMI [69]), un format d'échange de modèles standard représentant les modèles via une notation XML. Il est néanmoins possible de modifier le code Java généré lors de la première exécution de la génération depuis le module "main" et ainsi faire en sorte que le programme accepte plusieurs modèles sources et sous d'autres extensions que ".xmi".

Le moteur Acceleo gère la traçabilité de tous les éléments impliqués dans la génération d'un fichier. Ce système permet, par exemple, de déterminer les éléments des modèles sources qui ont été utilisés pour générer un morceau spécifique de texte et la partie du générateur qui a été impliquée.

Nous avons vu que les langages de transformations de modèles peuvent être classifiés selon divers critères. De plus, l'environnement Eclipse englobe un ensemble d'outils permettant de travailler dans une approche d'IDM et de transformations de modèles. De fait, un langage supporté par Eclipse est sans doute le plus intéressant. Grâce à cela, il sera possible de choisir le langage le plus intéressant au vu des objectifs fixés et d'autres aspects abordés par la suite.

4 Lignes de produits et feature models

Une ligne de produits est un ensemble de produits possédant une série de caractéristiques communes. Les lignes de produits existent depuis longtemps dans de nombreux domaines. Par exemple, les fabricants automobiles peuvent créer différentes variantes uniques d'un modèle de voiture en utilisant un ensemble de parties soigneusement conçues et d'une fabrique spécifiquement conçue pour configurer et assembler ces parties [4].

En informatique, nous avons les lignes de produits logiciels. Une Ligne de Produits Logiciels (LPL), ou "Software Product Line" (SPL), est un ensemble de systèmes logiciels partageant un ensemble de propriétés communes, satisfaisant des besoins spécifiques pour un domaine particulier, plutôt que pour un système particulier, et développée sur base d'un ensemble de composants clefs à l'aide de méthodes, outils et techniques de génie logiciel [20]. Ces méthodes, outils et techniques sont repris sous le nom d'Ingénierie des Lignes de Produits Logiciels (ILPL), ou "Software Product Line Engineering" (SPLE).

La caractéristique qui distingue l'ILPL de précédents efforts consiste à la réutilisation prédictive plutôt que la réutilisation opportuniste de logiciel. Plutôt que de placer des composants logiciels génériques dans une librairie en espérant qu'il y aura des opportunités futures de réutilisation, les lignes de produits logiciels ne commandent la création d'artefacts logiciels que lorsque la réutilisation est prédite pour un ou plusieurs produits d'une ligne de produits bien définie [4].

L'ILPL est une combinaison de l'ingénierie de domaine et de l'ingénierie d'application classique comme l'illustre la Figure 4.1, reprise de [1]. L'ingénierie de domaine est utilisée pour développer un ensemble d'artefacts réutilisables. Ces artefacts sont définis à chaque étape du processus. L'ensemble de produits qui peuvent être dérivés de ces artefacts est appelé le "portefeuille" [53].

Les artefacts sont ensuite réutilisés durant le développement de produits, c'est-à-dire l'ingénierie d'application. Cette ingénierie exploite les caractéristiques partagées du "portefeuille" en configurant les artefacts en question. Le processus de configuration est l'activité durant laquelle l'utilisateur décide quels composants doivent être sélectionnés pour correspondre aux exigences du client pour son produit particulier [53].

En résumé, nous avons donc une première phase qui consiste à définir la ligne de produits avec toutes ces caractéristiques communes et variables, à développer celles-ci. Ensuite, pour chaque nouveau client, en fonction de ses exigences, un produit particulier sera conçu en configurant les différents artefacts représentant les caractéristiques variables de la ligne.

Nous pouvons prendre pour exemple un ensemble de systèmes de sondage. En ingénierie d'application classique, l'équipe de développement créerait un système précis, permettant notamment de créer tels types de questions et tels types de réponses en fonction des exigences de son client. Ils rassembleraient les com-

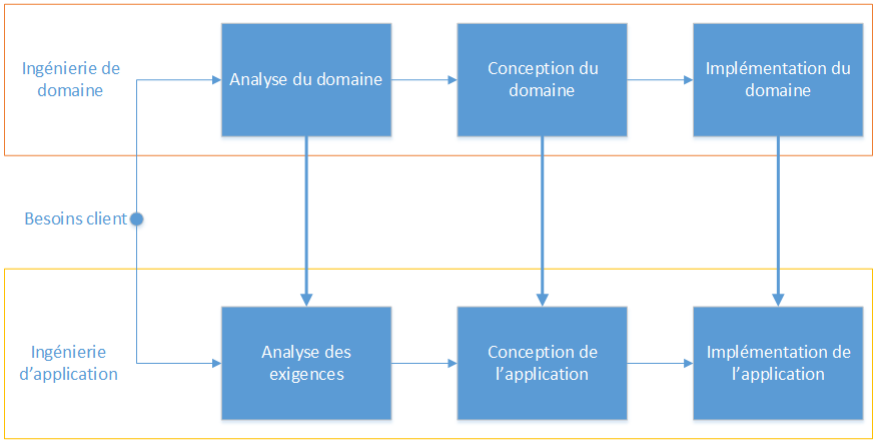

FIGURE 4.1 – Ingénierie des Lignes de Produits Logiciels

posants, les fonctionnalités dans des librairies en espérant que certaines puissent être réutilisées si un autre client demande également un système de sondage ou similaire. En ILPL, plutôt que de réaliser un système particulier, il s'agit de réaliser une fabrique de systèmes de sondage avec différents composants/artefacts. Parmi ceux-ci, nous avons des composants communs, que tous les systèmes de ce type possèdent, et les composants variables, présents ou non dans un système particulier. Par exemple, un premier client sera satisfait avec un système de sondage permettant uniquement de créer des questions à choix multiple tandis qu'un autre demandera la possibilité de créer des questions ouvertes. Dans ce cas, la réutilisation des composants est prévue et non simplement espérée.

Tous les produits d'une ligne disposent donc d'un tronc commun de propriétés/composants et chaque produit est différencié par des propriétés variables. La planification, l'organisation et la construction du "portefeuille" requièrent une documentation rigoureuse de la variabilité de la ligne [53]. Ainsi, une des façons de modéliser la variabilité des lignes de produits est d'utiliser les "Feature Models" (FMs). Un FM est composé de *features* qui représentent les propriétés, les caractéristiques de la ligne. Elles représentent les points de variation qui correspondent aux différents composants/artefacts qu'il sera possible de configurer pour un système spécifique. Une *feature* est définie par [56] comme "un aspect, une qualité ou une caractéristique proéminente ou distinctive, visible pour l'utilisateur, d'un système logiciel ou système". Dans le cas d'une ligne de voitures, la couleur et les roues sont des *features* probables.

Un membre d'une ligne de produits peut être spécifié par une configuration de *features* du FM, c'est-à-dire par un ensemble de *features* que le membre possède. Une configuration précise est permise par le FM si elle ne viole pas les contraintes imposées par le modèle [2]. Le FM va donc délimiter l'ensemble des produits valides de la ligne de produits.

Comme l'explique [44], les FMs sont des graphes dirigés acycliques, en général des arbres, où les noeuds sont les *features* et les arcs représentent la décomposition hiérarchique des *features*. Un lien de décomposition signifie que si une *feature* fait partie de la configuration d'un produit alors certaines de ces *features* filles doivent aussi en faire partie. Quant à savoir le nombre exact et quelles *features* doivent faire partie de la configuration, cela va dépendre du type de lien de décomposition [44]. Dans la notation de base, trois types de décomposition sont possibles : *and*, *or* et *xor*. La première signifie que toutes les *features* de la décomposition doivent faire partie de la configuration si la *feature* parente en fait partie, la seconde signifie qu'au moins une *feature* doit faire partie de la configuration et la dernière qu'une et une seule *feature* doit en faire partie. Les *features* peuvent également être marquées comme optionnelles. Certaines notations permettent de définir les décompositions avec des cardinalités plus précises que les trois décompositions basiques [2].

La Figure 4.2 est un FM fictif d'une ligne d'ordinateurs. Ce FM sera repris plusieurs fois dans ce document comme exemple. Il a été réalisé avec l'outil FeatureIDE [85], outil intégré à l'environnement Eclipse.

La *feature* racine *Ordinateur* possède une décomposition *and* où toutes les *features* doivent faire partie du produit sauf les *features* optionnelles *Batterie* et *GarantieEtendue*. La plupart des *features* filles de Ordinateur ont une décomposition *xor* où une seule *feature* doit faire partie du produit sauf la *feature* *MemoireVive* qui possède une décomposition *or* où au moins une *feature* doit faire partie du produit.

Dans un FM, nous retrouvons diverses contraintes qui peuvent être divisées en trois catégories :

1. Les contraintes de descendance : si une *feature* fait partie de la configuration d'un produit, alors sa *feature* parente et les *features* ascendantes en font également partie.
2. Les contraintes de groupe : la cardinalité d'un groupe de *features* est une forme de contrainte. Par exemple, un groupe *xor* oblige qu'il y ait une et une seule *feature* du groupe dans la configuration si la *feature* parente en fait partie.
3. Les contraintes "cross-tree" : toutes les autres contraintes sont de ce type. Il est par exemple possible de définir des contraintes d'implication entre *features* appartenant à des branches différentes du FM. Dans la Figure 4.2, elles sont définies en dessous de la hiérarchie des *features*.

Dans la communauté scientifique, les FMs sont de fait le standard pour représenter la variabilité d'une ligne de produit. Par contre, cela n'est pas le cas dans l'industrie. Selon [44], il y a deux raisons probables qui expliquent cet état de fait.

Une des raisons serait sans doute leur manque de concision et leur représentation peu naturelle des lignes de produits, pas toujours réaliste. En effet, suite à des expériences variées en industrie, il est apparu qu'une des constructions les plus efficaces pour rendre les FMs plus concis et intuitifs sont les attributs de *features*. Similairement aux attributs de classes dans les diagrammes de classes

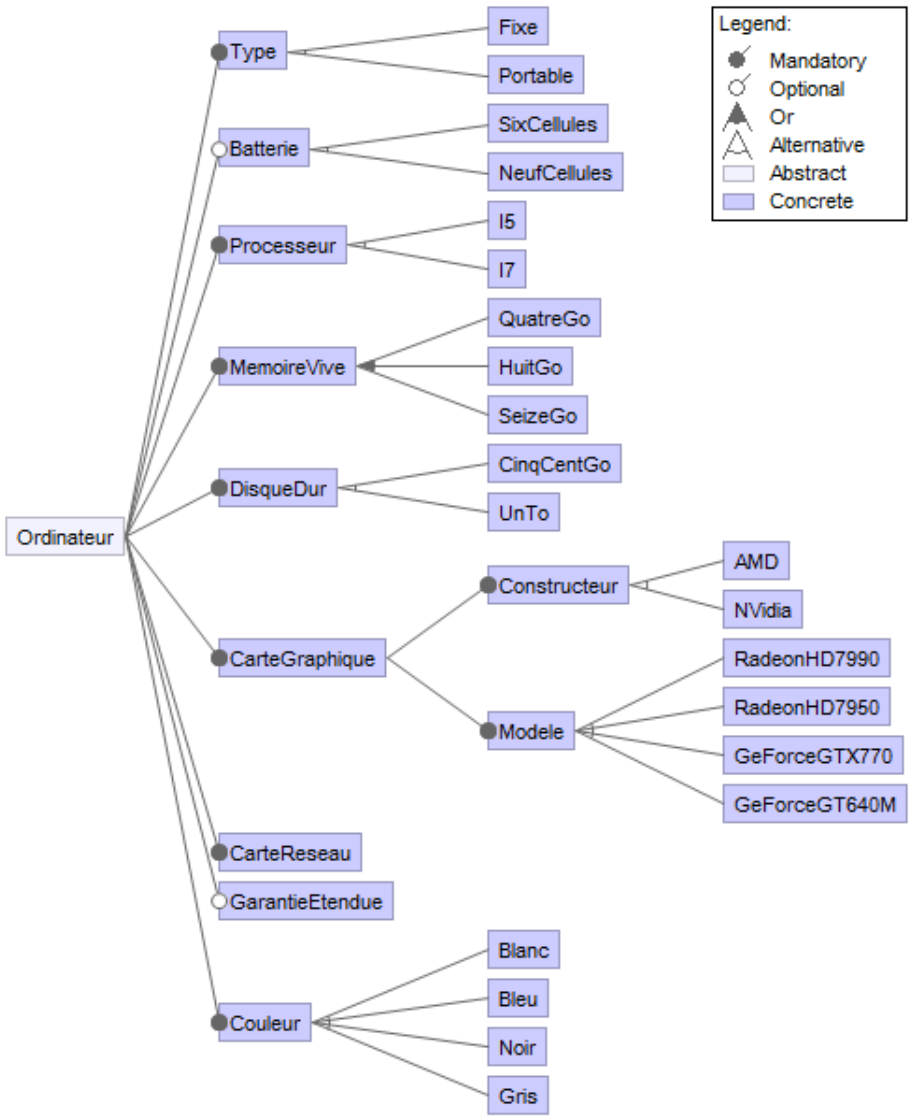

FIGURE 4.2 – Feature Model - Ligne d'ordinateurs

UML, les attributs sont des paramètres typés des *features*. L'utilisation d'attributs va en général permettre de diminuer drastiquement le nombre de *features* ainsi que de rendre le modèle plus naturel et facile à comprendre. Par exemple, sans l'utilisation d'attributs, nous sommes obligés de modéliser les choix alternatifs, qui ne sont pas eux-même décomposés, par des décompositions *xor*, comme c'est par exemple le cas de la *feature Couleur* du FM présenté. La même information pourrait être présentée avec plus de concision grâce à un attribut de type énumération attaché à la *feature Ordinateur*. La sémantique est néanmoins la même, seule la notation diffère. Malgré cela, la sémantique des attributs de *features* est souvent mal comprise et beaucoup d'outils n'offrent pas de support pour les attributs, comme c'est le cas de FeatureIDE, utilisé pour créer l'exemple de FM.

Une autre raison probable au manque d'utilisation industrielle des FMs est la nature graphique de leur syntaxe. Bien que des langages textuels existent, la plupart des langages de FMs sont basés sur la notation FODA [56] qui utilise des graphes avec des noeuds et arcs dans un espace en 2D, comme c'est le cas pour la Figure 4.2. Premièrement, les attributs de *features* sont textuels par nature et ne s'intègrent pas bien à cette représentation. Deuxièmement, les contraintes sont souvent placées sous forme d'annotations et exprimées sous forme d'opérateurs booléens. Si les attributs et contraintes se voyaient attribuer une syntaxe graphique, celle-ci ne ferait qu'alourdir le FM. De plus, en travaillant avec des ingénieurs, l'équipe PReCISE a remarqué qu'une syntaxe graphique représente également une barrière psychologique dans le sens où le fait de devoir dessiner des modèles est vu comme fastidieux et lourd. Il y a également des problèmes au niveau des outils permettant de créer des FMs graphiques qui sont généralement des prototypes de recherche et inférieurs aux outils supportant les langages textuels [44].

L'équipe PReCISE a conçu un nouveau langage textuel pour écrire des FMs, appelé TVL, qui comble les lacunes des langages habituels. Le chapitre suivant est consacré, dans sa première partie, à décrire ce langage et à expliquer en quoi il surmonte ces problèmes.

5 Langages TVL et FCSS

Ce chapitre s'intéresse aux deux DSLs développés par l'équipe PReCISE : TVL et FCSS. Le premier, qui signifie "Text-based Variability Language" [43] [44], est un langage de FMs, qui permet donc d'exprimer la variabilité de lignes de produits. Le second, acronyme de "Feature Cascading Style Sheet", ajoute des informations complémentaires qui seront utiles lors de la génération de l'interface graphique d'un configurateur.

Ces deux langages ont été définis grâce au framework Xtext [24]. Ce framework, intégré à Eclipse, permet de définir des DSLs ainsi que l'infrastructure les accompagnant, parseurs et compilateurs entre autres, et tire avantage du fait qu'il soit lié à d'autres outils d'Eclipse. Parmi ces avantages, il permet de générer différentes syntaxes concrètes pour un même langage. En effet, il est possible d'avoir une notation textuelle de ce langage avec l'Eclipse Modeling Framework (EMF [82]) et une notation graphique avec le Graphical Modeling Framework (GMF [18]) [24]. Une fois qu'une grammaire Xtext est définie, un environnement de développement Eclipse peut être généré avec tous les bénéfices que cela implique, allant de la coloration syntaxique à l'auto-complétion. Outre cet éditeur, un méta-modèle, conforme à Ecore [17], du langage est également généré. Ecore est une implémentation de Essential MOF (EMOF [68]) [3], un sous-ensemble de MOF. Il s'agit donc du méta-méta-modèle dans l'architecture d'Eclipse. Grâce à la connaissance de ces méta-modèles TVL et FCSS, il sera donc possible de définir un programme de transformation, qui à l'exécution, à partir de modèles TVL et FCSS conformes, produira un ou plusieurs modèle(s) ou du code en fonction du type de transformation.

5.1 TVL

TVL est un langage textuel permettant de créer des FMs, développé par l'équipe de PReCISE, qui pallie aux problèmes énoncés dans le chapitre précédent concernant les autres langages de FMs. Ainsi, TVL propose des mécanismes de structuration améliorant la lisibilité des modèles et gère différents types d'attributs. Nous verrons, entre autres, ces différentes constructions par la suite.

L'objectif principal derrière le développement de TVL était de fournir aux ingénieurs un langage lisible avec une syntaxe riche facilitant la modélisation et permettant de créer des modèles naturels, mais aussi avec une sémantique formelle. Une telle sémantique permet d'éviter les ambiguités, permet également que TVL soit utilisé comme format d'échange d'information et pour d'autres processus automatisés. En effet, étant donné que le langage TVL a été conçu avec Xtext, un méta-modèle Ecore du langage est disponible et il est donc possible de l'utiliser dans une approche d'ingénierie dirigée par les modèles, comme présenté dans le Chapitre 3. Le langage TVL est accompagné d'un éditeur capable notamment de vérifier sa syntaxe et, grâce à un solveur interne, que les

contraintes définies soient correctes vis-à-vis des autres. Un autre objectif était d'avoir un langage léger, peu verbeux, au contraire des langages de balises basés sur XML. La syntaxe de TVL se rapproche plus d'un langage comme le C [44].

Cette section a pour but de présenter la syntaxe du langage tout en l'illustrant grâce au FM TVL (Code 5.1) représentant la variabilité d'une ligne d'ordinateurs. Cet exemple, totalement fictif, n'a pas pour but d'être totalement réaliste ou exhaustif dans sa représentation d'une ligne d'ordinateurs, notamment au niveau des modèles des différents composants, il n'est là que pour illustrer le propos. Il s'agit d'une version plus complète et écrite en TVL du FM (voir Figure 4.2) présenté dans le chapitre précédent. La sémantique n'est présentée qu'avec concision, le document [44] explique à la fois la syntaxe et la sémantique de manière plus détaillée et formelle.

```
1 //Déclaration de types prédéfinis
2 int annees in [1..3];
3 enum taille in {TreizePouces, QuinzePouces, DixSeptPouces, DixNeufPouces};
4
5 struct ports{
6   int nbrePortsUSB;
7   int nbrePortsHDMI;
8 }
9
10 //Feature model
11 root Ordinateur {
12   Portable -> Batterie;
13   Fixe -> !Batterie;
14   group allof {
15     Type group [1..1] {Fixe, Portable},
16     opt Batterie group oneof {SixCellules, NeufCellules},
17     Processeur group oneof {I5,I7},
18     MemoireVive group someof {QuatreGo, HuitGo, SeizeGo},
19     DisqueDur{
20       group oneof {CinqCentGo,UnTo}
21       bool ssd;
22     },
23     CarteGraphique{
24       group allof{
25         Constructeur group oneof {AMD, NVidia},
26         Modele group oneof {
27           RadeonHD7990,
28           RadeonHD7950,
29           GeForceGTX770,
30           GeForceGT640M
31         }
32       }
33     },
34     CarteReseau {bool bluetooth;},
35     opt GarantieEtendue{
36       annees anneesGarantie is 1;
37     }
38   }
39   enum couleur in {noir, blanc, gris, bleu};
40   taille tailleEcran in {QuinzePouces,DixSeptPouces,DixNeufPouces};
41   bool paveNumerique;
42   ports lesPorts {
```

```
43      nbrePortsUSB in [2..4]; nbrePortsUSB is 2;
44      nbrePortsHDMI in [0..2]; nbrePortsHDMI is 1;
45    }
46 }
47
48 //Extension de feature
49 CarteGraphique{
50   AMD -> RadeonHD7990 || RadeonHD7950;
51   NVidia -> GeForceGTX770 || GeForceGT640M;
52   Portable -> NVidia && GeForceGT640M;
53   Fixe -> !GeForceGT640M;
54 }
```

Code 5.1 – FM TVL Ordinateur

Il existe plusieurs éléments importants : les *features*, les attributs, les groupes de décomposition de *features* et les contraintes.

Comme le montre l'exemple, le FM en lui-même, comprenant l'arbre des *features*, ne commence pas forcément directement, il peut être précédé de la déclaration de divers éléments. Ces éléments, allant de la ligne 2 à la ligne 8, seront abordés par la suite.

5.1.1 Features et groupes

La *feature* racine, précédée du mot réservé *root* est unique pour tout modèle TVL. Il s'agit ici de la *feature Ordinateur*, décomposée en huit *sous-features* par un groupe *allof* (ligne 14) qui correspond à l'opérateur booleen *and*. Une décomposition *allof* signifie que toutes les *features* du groupe sont obligatoires, à l'exception des éventuelles *features* optionnelles reconnaissables grâce au mot réservé *opt*, ce qui est le cas notamment de la *feature Batterie* à la ligne 16. Les autre types de décomposition définissables en TVL, *oneof* et *someof*, correspondent respectivement à des décompositions *xor* et *or*. Dans le premier cas, une seule *feature* doit être sélectionnée, dans le second, au moins une doit l'être. Ces mots ont été choisis pour rendre le langage accessible aux personnes non familiarisées avec la logique booléenne d'une décomposition. Comme l'exemple l'illustre, les décompositions sont précédées du mot clé *group*.

N'importe quelle *feature* faisant partie d'un groupe peut également être elle-même décomposée en *sous-features*. C'est le cas de la *feature Type* à la ligne 15. Cette décomposition montre qu'en dehors des décompositions déjà abordées, il est possible de donner des cardinalités à celles-ci. Pour définir les cardinalités, le concepteur peut utiliser des nombres naturels, des constantes ou l'astérisque (*). Dans le cas présenté, *group [1..1]* est équivalent à *group oneof*. Pour les autres décompositions possibles, *group [1..*]* correspond à *group someof* et un *group allof* peut être remplacé par *group [*..*]* ou encore *group [6..6]* où 6 est le nombre de *sous-features* de la décomposition. Toutes autres cardinalités sont évidemment possibles, tant que celles-ci sont comprises entre 0 et le nombre de *features* du groupe de décomposition, les groupes *oneof*, *someof* et *allof* n'étant au final que des cas particuliers de cardinalités, appelés sucres syntaxiques. Pour revenir aux *features* optionnelles, à savoir les *features Batterie* et *GarantieEten-*

due dans l'exemple, elles ont pour conséquence de diminuer la borne inférieure d'une décomposition, de 1 par *feature* optionnelle. Ainsi le groupe *allof* à la ligne 14 qui serait normalement de cardinalité [8..8] étant donné qu'il y a huit *sous-features* a en réalité une cardinalité de [6..8]. Si le groupe avait été de type *oneof*, la cardinalité, au lieu d'être [1..1] aurait été [0..1], une borne inférieure à 0 n'ayant pas de sens. Dans le cas d'un groupe de cardinalité *someof*, elle passerait de [1..8] à [0..8].

En dehors du groupe de *sous-features* d'une *feature*, le corps de celle-ci, délimité par des accolades, peut contenir des attributs et des contraintes. Si le corps contient uniquement un groupe, les accolades ne sont pas obligatoires, le corps est délimité par le groupe en lui même et ses propres accolades, comme c'est le cas pour la *feature Type*, toujours à la ligne 15.

Dans le but d'obtenir un document le plus lisible possible, deux procédés de structuration ont été mis en place dans le langage. Premièrement, les *features* peuvent être étendues, c'est-à-dire, que leur contenu peut être réparti entre leur corps dans la hiérarchie de base et leurs extensions, définies en dehors de cette hiérarchie. Dans l'exemple, la *feature CarteGraphique* est déclarée à la ligne 23 et étendue à la ligne 49. Une *feature* peut avoir autant d'extensions que souhaité. Il est à noter que rien n'empêche d'étendre la *feature* racine, toute *feature*, à tout niveau, peut être étendue. Un deuxième procédé, qui n'est pas montré dans l'exemple, est la possibilité d'inclure du contenu provenant d'un autre fichier dans un FM via la ligne *include(chemin_ du_ fichier) ;*.

Une dernière caractéristique de TVL à noter est qu'une *feature* ne peut avoir plusieurs groupes de décompositions. Cette contrainte est valable pour le corps même de la *feature* et ses extensions. Par exemple, il n'est pas possible de définir un groupe *oneof* dans le corps de la *feature* et un groupe *someof* dans une de ses extensions.

5.1.2 Attributs

Comme précisé précédemment, les *features*, en dehors des groupes de *sous-features*, peuvent notamment contenir des attributs qui correspondent à des propriétés de ces *features*. Ils permettent, comme énoncé dans le Chapitre 4, de concevoir des FMs plus concis et qui sont des représentations plus réalistes de lignes de produits. En TVL, il est possible de définir cinq types d'attributs : quatre types d'attributs de base et un type d'attribut structuré. Parmi les attributs de base, nous retrouvons les entiers précédés du mot clé *int*, les réels (*real*), les booléens (*bool*) et les énumérations (*enum*).

Il est possible de donner une valeur par défaut à un attribut ou encore de limiter son domaine de valeurs. Lorsque le concepteur souhaite indiquer une valeur par défaut, il doit faire suivre la déclaration de l'attribut par le mot clé *is* et la valeur en question. En ce qui concerne la limitation du domaine de valeurs, il faut ajouter le mot clé *in* et placer les valeurs possibles entre crochets à la suite de la définition de l'attribut pour une limitation grâce aux bornes inférieure et supérieure ou entre accolades pour une limitation par énumération des valeurs. Dans la structure *lesPorts* à la ligne 42, les attributs entiers de la structure

prennent chacun une valeur par défaut et leur domaine de valeurs est limité. Le concepteur aurait également pu énumérer les trois valeurs possibles comme ceci : *nbrePortsUSB in {2,3,4}*. La limitation de domaine de valeurs pour une énumération ne peut se faire que par énumération des valeurs admises. Dans l'exemple, nous avons un type prédéfini *taille* qui définit une liste de tailles possibles (ligne 3). Dans le FM, l'attribut *tailleEcran* est de ce type et son domaine de valeurs est réduit (ligne 40). Ainsi, les écrans de 13 pouces ne sont pas disponibles. Nous reviendrons un peu plus loin sur les types prédéfinis.

Le fait de pouvoir attribuer une valeur par défaut ou délimiter le domaine de valeurs implique qu'il est nécessaire d'effectuer des vérifications lors de la conception. Par exemple, il est évident qu'un entier dont le domaine de valeurs est compris entre 2 et 4 ne peut avoir 1 comme valeur par défaut. Ces vérifications sont effectuées par l'éditeur TVL et non par la solution qui sera présentée dans la suite de ce document.

Comme nous l'avons abordé précédemment via l'exemple des tailles d'écrans, il est également possible de définir ses propres types d'attributs par dérivation des types de base mais également de définir des types structurés. Cette définition d'un type peut n'être qu'un simple renommage. Par exemple, le type *annees* défini à la ligne 2 revient à renommer le type *entier* en type *annees*. Seulement, ce type est également dérivé puisque le domaine de valeurs du type *annees* est défini et limite les valeurs possibles qu'un attribut de ce type peut prendre. Nous pouvons ensuite, à l'intérieur d'une *feature*, définir un attribut du type *annees* comme ceci : *annees anneesGarantie is 1;* où nous en profitons pour lui donner une valeur par défaut. Cette possibilité permet au concepteur d'à la fois donner une valeur par défaut à un attribut tout en limitant son domaine de valeurs, ce qui n'est pas possible pour un attribut directement déclaré dans une *feature*. Par exemple, l'éditeur TVL ne permet pas d'écrire directement *int anneesGarantie in [1..3] is 1* (ligne 36). TVL permet également de définir des constantes entières, réelles ou booléennes qui peuvent être utilisées pour définir des valeurs par défaut pour les attributs des types correspondants. Comme indiqué précédemment, les constantes entières peuvent également être utilisées pour définir les bornes inférieure et supérieure des cardinalités des groupes de features.

Les attributs structurés sont définis grâce au mot clé *struct*. Ces attributs structurés doivent toujours être définis en dehors de la hiérarchie de *features* du FM. Il s'agit, comme précédemment et comme pour le langage C par exemple, de définir un type d'attribut. Le concepteur pourra ensuite définir autant d'instances de ce type qu'il le souhaite dans le FM. Un attribut structuré est composé d'une liste d'attributs de base. L'exemple ci-dessus propose un type d'attribut structuré *ports*, à la ligne 5, qui contient deux attributs de base indiquant respectivement le nombre de ports USB et de ports HDMI que l'utilisateur souhaite sur son ordinateur configurable. Tout attribut déclaré de ce type possèdera ces deux attributs entiers. Les attributs composants d'une structure peuvent, tout comme les attributs de base, avoir une valeur par défaut ou un domaine limité, comme illustré aux lignes 43 et 44 de l'exemple.

Les similarités que présente TVL par rapport au langage C (déclaration

d'attributs similaire à la déclaration de variables, déclaration de structures également similaire) ont pour but de rendre le langage assez intuitif pour tout informaticien.

5.1.3 Identifiants des éléments

Même si l'exemple ne le montre pas, il est possible, en TVL, de définir deux (ou plus) *features* portant le même nom (aussi appelé *short-id* dans le jargon TVL) tant qu'elles n'ont pas la même *feature* parente. Cela est également vrai pour les attributs, il peut y avoir deux (ou plus) attributs du même nom tant qu'ils ne font pas partie du corps d'une même *feature*.

Afin de distinguer deux *features* ayant le même nom mais se trouvant en des endroits différents, l'éditeur TVL utilise des *long-ids* qui permettent de les identifier. Un *long-id* correspond à un nom pleinement qualifié ("fully qualified names") ou partiellement qualifié ("partially qualified names"). Un *long-id* correspond à la concaténation des noms des *features*, séparés par des points, se trouvant sur le chemin menant de la *feature* racine à l'élément concerné. Dans l'exemple, un *long-id* de *Constructeur* (ligne 25) est *Ordinateur.CarteGraphique.Constructeur* et celui de l'attribut *anneesGarantie* (ligne 36) est *Ordinateur.GarantieEtendue.anneesGarantie*. Dans ce cas, ce sont tous deux des noms pleinement qualifiés. Leur version partiellement qualifiée est respectivement *CarteGraphique.Constructeur* et *GarantieEtendue.anneesGarantie*. Ces noms peuvent être considérés comme des *long-ids* des éléments si ils les identifient ce qui est le cas ici.

5.1.4 Contraintes

Tout comme d'autres langages de FM, TVL permet de définir des contraintes autres que les contraintes de groupes ou celles liées aux relations mères/filles. Ces contraintes peuvent impliquer à la fois des *features* et des attributs. Un exemple de contrainte qui peut être définie est l'implication :

feature1 -> feature2

Une telle contrainte signifie que si l'utilisateur choisit la *feature1*, la *feature2* devient obligatoire. Cette contrainte est évidemment simpliste et il est possible d'utiliser les opérateurs logiques habituels (and, or, xor, &&, ||, !, ->, <->, etc.) ainsi que les opérateurs mathématiques (min, max, >, <, ==, !=, >=, etc.) afin de définir des contraintes plus complexes. Les contraintes doivent être définies à l'intérieur d'une *feature* ou d'une extension de *feature*. Il est également possible de définir des contraintes *IfIn* et IfOut. Une contrainte *IfIn*, placée à l'intérieur d'une *feature*, est applicable si cette *feature* a été sélectionnée par l'utilisateur, c'est-à-dire qu'elle fait partie de la configuration tandis qu'une contrainte *IfOut* est applicable si la *feature* ne fait pas partie de la configuration.

Dans l'exemple, nous avons quelques contraintes aux lignes 12-13 ainsi qu'aux lignes 50 à 53, définies à l'intérieur d'une extension de *feature*.

Grâce au langage TVL, il est donc possible de modéliser la variabilité d'une ligne de produits et, à partir de ce modèle, générer un configurateur permettant à l'utilisateur de configurer le produit qu'il souhaite.

Malgré tout, il n'est pas possible de générer des configurateurs réellement satisfaisants du point de vue visuel uniquement à partir de modèles TVL. Des règles de transformation prenant une construction TVL et la transformant en un "widget" bien précis ne permettraient qu'une génération trop rigide, sans paramétrisation. Par exemple, le générateur devrait prendre les *short-ids* des différents éléments TVL comme labels. Or, les *features* et attributs des FMs peuvent avoir des noms incompréhensibles pour les utilisateurs lambda, parce qu'il s'agit d'acronymes ou de noms internes à l'entreprise par exemple. Un deuxième langage permettant de personnaliser le résultat visuel souhaité est dès lors nécessaire. C'est avec cet objectif en tête que le langage FCSS a été développé.

5.2 FCSS

Le langage FCSS [39] n'est pas un langage de FM mais vient en complément du langage TVL dans l'optique de la génération d'interfaces graphiques. Comme son nom le laisse supposer, il a un rôle similaire au CSS dans le développement d'un site Web en HTML, c'est-à-dire, un rôle "d'embellissement", de feuille de style, de l'interface. Il est tout de même différent dans le sens où l'objectif principal derrière sa mise en place est d'apporter de la flexibilité dans la génération. Comme il sera expliqué dans le Chapitre 9, il permet de contourner les règles de "mappings" par défaut en laissant le concepteur définir lui-même les "mappings" qu'il désire. Un modèle FCSS vient en complément d'un modèle TVL et est donc toujours lié à celui-ci.

Le modèle FCSS présenté (Code 5.2) est un modèle possible lié au FM TVL (Code 5.1) présenté dans la section précédente.

```
1 import "exempleOrdinateur.tvl"
2
3 .{
4   andGroup: textbox;
5   orGroup: checkbox;
6   xorGroup: radiogroup;
7   cardGroup: checkbox;
8   feature: text;
9   optFeature: checkbox;
10  intAttribute: textbox;
11  boolAttribute: checkbox;
12  enumAttribute: listbox;
13  groupContainer: true;
14 }
15
16 Ordinateur{label: "Configurez votre PC";}
17
18 MemoireVive{label: "RAM";}
19 QuatreGo{label: "4 Go";}
20 HuitGo{label: "8 Go";}
```

```
21 SeizeGo{label: "16 Go";}
22
23 Batterie{
24   opt: checkbox;
25   group{
26     label:"Nombre de cellules: ";
27     widget: radiogroup;
28     help: "Plus une batterie a de cellules plus
29     l'autonomie est grande";
30   }
31 }
32 SixCellules{label: "6";}
33 NeufCellules{label: "9";}
34
35 Type{
36   label: "PC fixe ou portable?";
37   widget: text;
38   generate: true;
39 }
40
41 DisqueDur{
42   label: "Disque dur";
43   help: "Un disque dur SSD est plus rapide qu'un disque dur normal";
44   group{
45     widget: listbox;
46     default: UnTo;
47     label: "Capacite du disque dur";
48     container:true;
49   };
50 }
51 CinqCentGo{label: "500 Go";}
52 UnTo{label: "1 To";}
53
54 CarteReseau{label: "Carte reseau";}
55
56 GarantieEtendue{
57   label: "Etendue de garantie";
58   help: "La garantie de base est de 1 an, cochez cette case
59   pour pouvoir l'etendre.";
60 }
61
62 #ssd{label: "Disque dur SSD";}
63
64 #anneesGaranties{label: "Nombre d'annees d'extension de garantie: ";}
65
66 #couleur{
67   label: "Couleur";
68   widget: radiogroup;
69 }
70
71 #tailleEcran{
72   label: "Taille de l'ecran";
73   help: "Les ecrans 13 pouces sont momentanement indisponibles.";
74 }
75
76 #paveNumerique{label: "Clavier avec pave numerique";}
77
```

```
78#bluetooth{label: "Bluetooth integre";}
79
80#lesPorts{label: "Ports";}
81#nbrePortsUSB{label: "Nombre de ports USB: ";}
82#nbrePortsHDMI{label: "Nombre de ports HDMI: ";}
```

Code 5.2 – FCSS Ordinateur

À la première ligne, nous pouvons voir *import "exempleOrdinateur.tvl"* qui indique à quel fichier TVL (donc à quel modèle) ce modèle FCSS est lié.

Un modèle FCSS peut contenir une partie globale et des parties spécifiques aux *features* et attributs. En réalité, il doit contenir au moins une partie, qu'elle soit spécifique à un élément ou que ce soit la partie globale.

5.2.1 Partie globale

La partie globale, qui commence par le symbôle '.' et est délimitée par des accolades, définit un ensemble d'attributs FCSS qui s'appliquent à tous les éléments TVL correspondants. Ainsi, l'attribut global *andGroup* s'applique à tous les groupes de *features* de cardinalité *allof*, l'attribut FCSS *intAttribute* s'applique à tous les attributs TVL entier, etc. Il en va de même pour chaque attribut global. Dans l'exemple, la partie globale s'étend donc de la ligne 3 à la ligne 14.

La liste des attributs globaux est la suivante :

1. *andGroup* : s'applique aux groupes de cardinalité *allof*. La seule valeur possible est "textbox" qui signifie que le contenu des *features* doit être encadré.
2. *orGroup* : s'applique aux groupes de cardinalité *someof*. Les valeurs possibles sont "checkbox" qui consiste à représenter les *features* par un groupe de "checkbox", suivies par les noms de chaque *feature* et "listbox" qui correspond ici à une liste de valeurs, les *features*, à choix multiple.
3. *xorGroup* : s'applique aux groupes de cardinalité *oneof*. Les valeurs possibles sont "listbox", une liste de valeurs pour laquelle un seul choix est autorisé, et "radiogroup" qui correspond à un groupe de radios placées devant chaque nom de *feature* du groupe.
4. *cardGroup* : s'applique aux groupes dont la cardinalité est différente de *allof*, *someof* et *oneof*. La seule valeur proposée est "checkbox", un groupe de "checkbox" placées devant les noms de *features* du groupe.
5. *feature* : s'applique aux *features* en elles-mêmes. Les valeurs possibles sont "text" qui correspond au label affiché pour la *feature*, c'est-à-dire son nom ou le label qui peut être spécifié dans la partie spécifique de celle-ci (voir plus loin) et "image" qui remplace le texte.
6. *optFeature* : s'applique aux *features* optionnelles. Les trois valeurs proposées sont "checkbox", "radiogroup" et "listbox". Dans les deux premiers cas, le *widget* en question est placé devant le label de la *feature* et s'il est coché, la *feature* est sélectionnée. Dans le dernier cas, la "listbox" permet de sélectionner ou non la *feature*.

7. *intAttribute* : s'applique aux attributs de type entier. La seule valeur actuellement disponible est "textbox" qui correspond à un champ de texte, prévu pour recevoir des nombres entiers comme entrée.
8. *realAttribute* : s'applique aux attributs de type réel. La seule valeur actuellement disponible est "textbox" qui correspond à un champ de texte, prévu pour recevoir des nombres réels comme entrée.
9. *boolAttribute* : s'applique aux attributs de type booléen. Deux valeurs sont possibles, il s'agit de "checkbox" et "listbox" qui contient les deux valeurs que peut prendre un attribut booléen.
10. *enumAttribute* : s'applique aux attributs de type énumération. Les valeurs possibles sont "listbox", une liste contenant les différentes valeurs de l'énumération et "radiogroup" qui consiste en un groupe de radios placées devant chaque valeur de l'énumération.
11. *groupContainer* : s'applique à un groupe de décomposition. Il permet d'indiquer si les groupes doivent être encadrés et peut donc prendre les valeurs "true" ou "false".

Comme nous pouvons le voir, pour certains éléments, les valeurs possibles se limitent à une seule valeur. La raison est que le langage FCSS est encore en cours de développement. Des ajouts viendront l'étoffer dans le futur. Certains sont proposés dans la Section 12.2 de ce document.

5.2.2 Parties spécifiques

En dehors de ces attributs globaux, il est possible de définir une partie spécifique à n'importe quelle *feature* ou attribut. Ces parties spécifiques sont définies grâce au nom (le *short-id*) ou au *long-id* de l'élément TVL concerné par chaque partie et sont délimitées par des accolades. Dans le cas des attributs TVL, ces définitions commencent par le caractère réservé '#'. Il existe des attributs FCSS communs aux *features* et attributs et d'autres spécifiques à chacun de ces types d'éléments TVL. Dans les attributs spécifiques aux *features*, nous retrouvons également des attributs FCSS qui concernent les groupes de décomposition.

Les attributs communs sont les suivants :

1. *generate* : indique si l'élément TVL doit être présent sur l'interface graphique. Cet attribut peut donc prendre deux valeurs : "true" ou "false". À la ligne 38 du modèle présenté, la *feature Type* a l'attribut *generate* à "true".
2. *label* : indique le label à afficher à la place du *short-id* de l'élément. Il n'est pas difficile d'imaginer plusieurs situations pour lesquelles cet attribut est utile. Par exemple, il est possible que le nom des *features* soient des noms internes à l'entreprise ou des acronymes, incompréhensibles pour l'utilisateur. Un autre exemple d'utilisation serait la traduction du nom de la *feature* ou de l'attribut dans une autre langue. La valeur de cet attribut est une chaîne de caractères. Dans l'exemple, l'attribut "label"

est utilisé pour plusieurs *features*, notamment aux lignes 16 et 18 à 21, et attributs, notamment à la ligne 62.

3. *help* : indique le texte d'aide. Un exemple d'aide est présenté dans l'extrait à la ligne 43. Les disques durs SSD n'étant pas encore tout à fait répandus, il est peut-être utile de renseigner l'utilisateur sur leur intérêt principal. La valeur de cet attribut est une chaîne de caractères. D'autres exemples d'aides sont donnés aux lignes 58 et 73.

Les attributs spécifiques aux *features* sont les suivants :

1. *widget* : indique la représentation graphique de la *feature* elle-même. Les valeurs possibles sont "text" qui indique que le label de la *feature* sera utilisé pour la représenter et "image" qui remplace le texte initialement prévu. À la ligne 37 de l'exemple, le concepteur a choisi de représenter la *feature Type* par du texte .
2. *opt* : indique la représentation graphique du caractère optionnel de la *feature* elle-même. Cet attribut ne s'applique donc qu'aux *features* optionnelles et ne doit pas être pris en compte si la *feature* en question ne l'est pas. Les valeurs possibles sont les mêmes que pour l'attribut global *optFeature*, à savoir "checkbox", "radiogroup" et "listbox". Si une *feature* a "checkbox" comme valeur de *opt*, cela signifie simplement que son label est précédé ou suivi d'une "checkbox", pas que cette dernière remplace le label. Il en va de même pour la valeur "radiogroup" ou "listbox" qui contiendrait les valeurs "Yes" et "No" avec cette dernière comme valeur par défaut. La ligne 24 de l'exemple indique que le caractère optionnel de la *feature Batterie* doit être représenté par une "checkbox".
3. *group* : indique la partie qui s'applique au groupe de *sous-features*. Elle est comprise entre accolades, comme aux lignes 25 à 30 du modèle présenté.

Parmi les attributs spécifiques aux *features*, nous retrouvons donc quelques attributs permettant de personnaliser le groupe de décomposition de la *feature* en question :

1. *widget* : indique la représentation graphique du groupe de *sous-features*. Les valeurs possibles sont "textbox" qui indique simplement que les contenus des *features* doivent être encadrés, "checkbox" qui consiste à placer des checkbox devant chaque *feature* du groupe, "listbox" qui reprend les différents labels des *features* du groupe si celles-ci sont vides, ou qui prend les valeurs "Yes" et "No" devant chaque feature si elles ne le sont pas, et enfin "radiogroup" qui consiste en un groupe de radios placés devant les labels des *features* du groupe. Il est à noter que les 3 dernières valeurs n'empêchent pas que le contenu d'une *feature* soit encadré, si celle-ci n'est pas vide.
2. *label* : indique le label du groupe de *sous-features*.
3. *help* : indique le texte d'aide.

4. *container* : indique si le groupe doit être encadré et peut donc prendre les valeurs "true" ou "false".
5. *default* : indique la *feature* sélectionnée par défaut dans le groupe de *sous-features* de la *feature* concernée. La valeur doit être le nom d'une *feature* faisant partie du groupe.

Le groupe définit aux lignes 44 à 49 de l'exemple présente 4 de ces attributs.

Enfin, il n'existe, au moment de la rédaction de ce document, qu'un seul attribut FCSS spécifique aux attributs TVL :

1. *widget* : indique la représentation graphique de l'attribut. Les valeurs possibles sont "textbox", "checkbox", "listbox" et "radiogroup". À la ligne 41 de l'extrait, l'énumération *couleur* devra être représentée par un "radiogroup" au contraire des autres énumérations qui seront représentées par des "listbox" comme indiqué dans la partie globale.

L'attribut FCSS *widget* de la partie spécifique de l'attribut TVL *couleur* montre qu'il est nécessaire de faire un choix de représentation concernant cette énumération. Intuitivement, la partie spécifique d'un élément TVL prendra toujours le pas sur ce qui est spécifié dans la partie globale qui concerne notamment l'élément en question. Ainsi, les attributs FCSS spécifiques aux *features* et attributs TVL ont priorité sur les attributs FCSS globaux. Par exemple, l'attribut spécifique FCSS *widget* d'un attribut TVL, appliqué à un attribut entier, aura priorité sur l'attribut FCSS global *intAttribute* qui s'applique à tous les attributs de type entier. Nous y reviendrons dans le Chapitre 9.

6 Technologies cibles

Étant donné que la solution proposée comporte un générateur d'interfaces graphiques, et maintenant que nous savons quels seront les modèles sources, il est nécessaire d'étudier les différentes technologies ou langages cibles possibles. Cela peut aller du Java Swing au HTML, en passant par Qt ou C# entre autres. Néanmoins, ces langages ne sont pas l'objet de ce chapitre même si HTML5 sera également abordé dans la deuxième section.

La première section se concentre sur les "User Interface Description Languages" (UIDLs). Ce sont des langages de description d'interfaces utilisateurs, ils ont des avantages qui font qu'ils représentent une option intéressante.

6.1 Les "User Interface Description Languages"

Nous allons d'abord présenter ce qu'est un UIDL pour ensuite définir les critères qui nous intéressent vis-à-vis de l'objectif fixé. Ensuite, chaque UIDL sera présenté et les différents critères seront évalués. Enfin, un tableau récapitulatif sera présenté ainsi qu'une conclusion expliquant le choix effectué.

6.1.1 Présentation

Un "User Interface Description Language" (UIDL) est un langage de haut niveau qui décrit différents aspects d'une interface graphique en cours de développement [86]. En général, il permettra de décrire les caractéristiques intéressantes d'une interface utilisateur par rapport au reste d'une application interactive en vue d'être utilisé durant certaines étapes du cycle de vie du développement de cette interface. Comme pour tout langage, un UIDL doit avoir une syntaxe et une sémantique définies. Un tel langage peut être vu comme un moyen de spécifier une interface utilisateur indépendamment d'un langage cible de programmation ou de balise qui servirait à implémenter cette interface [86].

Avec la prolifération actuelle des plateformes (PCs, Smartphones, Tablettes, etc.), l'intérêt pour les UIDLs parait évident. L'utilisation d'un UIDL pour décrire une interface utilisateur tout en restant indépendant de toute plateforme comporte en effet plusieurs avantages. Premièrement, cela permet de définir une seule fois l'interface utilisateur pour toutes les plateformes visées. Au final, nous avons une description générique qui sera bien plus facilement maintenable et un générateur pour chaque technologie afin de traduire la description de l'interface utilisateur dans cette technologie. Avec l'arrivée de nouvelles plateformes, la description générique ne devra pas (ou peu) être modifiée et il "suffirait" de créer un nouveau générateur pour la nouvelle plateforme. Deuxièmement, cela peut avoir un impact positif sur le marché. En effet, nous pouvons imaginer que les développeurs soient en général frileux lorsqu'il s'agit d'implémenter un projet (notamment l'UI) dans une nouvelle technologie n'ayant pas encore fait

ses preuves (et ne sachant pas si elle va durer), ayant peur que le travail n'aura servi à rien si celle-ci n'a pas de succès. Ce risque serait réduit par l'utilisation des UIDLs puisque seul le générateur serait à développer tout en gardant la description de l'interface. De plus, la logique du générateur ne changerait pas d'une technologie à une autre même si l'implémentation différait. Le développement d'un nouveau générateur serait dès lors plus rapide que la première fois.

Les UIDLs présentés ici se basent sur XML ce qui comporte deux avantages principaux. Premièrement, la notation XML est facile à comprendre et à utiliser, même pour les non programmeurs. Deuxièmement, XML étant une notation permettant de définir des langages et non un langage en lui-même, il est possible de faire évoluer les langages en définissant de nouvelles balises lors de l'apparition de nouvelles technologiques par exemple [86].

Cette section ne présente volontairement qu'une petite portion des UIDLs disponibles en fonction de leur intérêt vis-à-vis de l'objectif fixé. Tous ces UIDLs utilisent une notation XML.

6.1.2 Critères

Dans cette sous-section, nous allons voir quels sont les critères intéressants du point de vue de la solution visée. Pour rappel, nous souhaitons pouvoir, à partir d'un FM, générer une description d'interface graphique abstraite, indépendante de la plateforme et du langage de programmation. Pour cela, nous avons donc besoin d'un UIDL, ou en tout cas, un méta-modèle d'interface utilisateur abstraite défini par ce langage. Une fois la description abstraite générée, nous souhaitons pouvoir générer une ou plusieurs interfaces utilisateurs finales dans divers langages d'implémentation. Il est donc nécessaire que des outils, prenant en entrée un modèle abstrait conforme au méta-modèle défini par l'UIDL et générant du code source, qui supportent l'UIDL soient disponibles.

En fonction de cela, nous proposons la liste de critères suivante :

1. Indépendance vis-à-vis des plateformes : si le langage ne propose pas des descriptions indépendantes de la plateforme, il est tout aussi intéressant de générer directement l'interface voulue dans un langage d'implémentation.
2. Disponibilité d'outils : existe-t-il des outils permettant de générer des interfaces finales à partir des descriptions d'interfaces générées par la solution ?
3. Disponibilité du ou des méta-modèles : étant donné que nous avons une approche orientée sur les transformations de modèles, il est obligatoire d'avoir connaissance des méta-modèles cibles. Idéalement, ceux-ci devraient être disponibles sous un format exploitable par un outil (Ecore, XML Schema, etc.), pas seulement montrés sous forme d'images. Dans le cas où seule une image d'un méta-modèle est accessible, il serait possible de reconstruire ce dernier à partir de celle-ci. Cependant, il faut tout de même être prudent dans ce cas de figure étant donné qu'il est possible que l'image ne reprenne qu'une partie du méta-modèle, rendant inexploitable celui créé à partir de celle-ci. L'intérêt d'avoir accès au(x) méta-modèle(s) est de pouvoir générer un modèle intermédiaire défini par l'UIDL à partir des modèles TVL et

FCSS pour ensuite utiliser les outils disponibles pour générer les interfaces finales dans diverses technologies cibles.

4. Accessibilité : les informations accessibles sur le langage. L'évaluation de ce critère est assez subjective, elle dépend surtout des différents articles qui parlent de cet UIDL, de la présence d'un site ou d'une page officielle, etc.
5. Standard : le langage est-il une spécification d'un organisme de standardisation ou a-t-il été soumis pour standardisation ?
6. Indépendance vis-à-vis des modes d'interactions : permet-il de décrire des interfaces visuelles, sonores, etc. ? Ce critère n'est pas important actuellement mais pourrait avoir son importance dans le futur.

Certains langages basés sur XML sont assez proches des UIDLs et pourraient être considérés comme tels mais leur objectif est totalement différent de celui visé par ce projet. Ils ont été écartés. Concernant l'accessibilité des informations sur le langage, celle-ci est en général limitée pour les langages conçus par des entreprises vendant des logiciels comme IBM ou Microsoft. De plus, il parait difficile d'avoir accès à des informations tels que les méta-modèles de ces langages. Ils ont donc également été mis à l'écart.

La liste ci-dessous présente quelques langages qui ont été écartés :

1. eXtensible User interface Language (XUL [12]) est un langage de Mozilla pour la description d'interfaces Web, utilisé notamment pour Firefox et ses nombreux plugins. En plus d'être un langage développé par une entreprise, il a été abandonné en 2011 par la société pour la version de Firefox sur Android en raison de performances insuffisantes selon leurs tests. Cette nouvelle a été annoncée par Jonathan Nightingale sur un groupe Google et transmise par différents sites Web dont [27].
2. Macromedia Flex Markup Language (MXML [6]) est un langage propriétaire d'Adobe. MXML est utilisé pour créer des applications Internet riches (Rich Internet Application (RIA)) et des applications interactives [86].
3. VoiceXML [31] est un langage qui se préoccupe des interfaces utilisateurs basées sur la reconnaissance vocale. Il s'agit d'un standard du World Wide Web Consortium (W3C).
4. InkML [30] est un langage qui vise à représenter les données d'encre numérique entrées via un stylo électronique à diverses fins (reconnaissance de l'écriture, vérification d'une signature,etc.). Il s'agit également d'un standard W3C.
5. eXtensible Interaction Scenario Language (XISL [57]) est un langage prévu pour le développement des systèmes Web avec des interfaces utilisateurs multi-modales. Le fait qu'il soit limité aux interfaces multi-modales est la raison pour laquelle il a été écarté.
6. Web Service eXperience Language (WSXL [25]) a été conçu pour représenter des données, la présentation et le contrôle. Il a été écarté car il s'agit d'un langage centré sur les services Web.

Cette section a pour source principale [86] qui répertorie une grande partie des UIDLs disponibles. Certains UIDLs présentés dans cet ouvrage ne sont pas vraiment des UIDLs tels que présentés dans la section précédente, ils ont été ignorés. D'autres l'ont également été pour les raisons précédemment évoquées.

Les sous-sections suivante classifient les UIDLs selon plusieurs catégories.

6.1.3 Langages basés sur le "Cameleon Reference Framework"

Le "Cameleon Reference Framework" [42] est un framework dont l'objectif est de servir de référence pour caractériser les modèles, méthodes et processus impliqués dans le développement d'interfaces utilisateurs ciblant plusieurs plateformes et différents contextes d'utilisations. Le contexte d'utilisation dépend des utilisateurs ciblés ou qui utilisent l'application, des plateformes (hardware et software) visées, c'est-à-dire des dispositifs qui peuvent être utilisés pour interagir avec le système et enfin de l'environnement physique dans lequel les interactions peuvent avoir lieu. Différents modèles permettent de représenter ces caractéristiques du contexte d'utilisation.

Ce framework structure le cycle de développement des interfaces utilisateurs en 4 niveaux d'abstractions :

1. Tâches et Concepts : décrit l'interface utilisateur en fonction des tâches que l'utilisateur peut accomplir et des concepts du domaine d'application manipulés par ces tâches.
2. Interface Utilisateur Abstraite, en anglais "Abstract User Interface" (AUI) : expression du rendu des concepts et des fonctions indépendamment des interacteurs disponibles sur la plateforme cible. L'AUI est indépendante de la plateforme cible et des modalités d'interaction.
3. Interface Utilisateur Concrète, en anglais "Concrete User Interface" (CUI) : expression de ce même rendu mais cette fois en étant dépendant des "widget"/objets d'interactions disponibles sur la plateforme cible. La CUI montre le "look-and-feel" de l'interface finale mais n'est encore qu'une maquette, non exécutable à part éventuellement dans l'environnement de développement.
4. Interface Utilisateur Finale, en anglais "Final User Interface" (FUI) : la FUI est générée à partir de la CUI. Il s'agit de l'interface utilisateur finale, codée dans un langage spécifique (Java Swing/AWT, HTML, etc.), qui peut ensuite être interprétée ou compilée.

À chaque niveau d'abstraction peut correspondre un ou plusieurs modèles. Par exemple, une AUI peut être composée d'un modèle de présentation, qui décrit les différents "widgets", leur emplacement, et d'un modèle de dialogue, décrivant son comportement.

Les auteurs définissent trois types de relations/transformations entre les différents niveaux :

1. Réification : passage d'un modèle plus abstrait à un modèle moins abstrait (ingénierie "directe").

2. Abstraction : passage d'un modèle moins abstrait à un autre plus abstrait (rétro-ingénierie).
3. Traduction : migration d'un modèle d'un certain niveau vers un autre modèle du même niveau. Utilisée pour passer d'un contexte d'utilisation à un autre.

Les modèles AUI et CUI peuvent donc être obtenus grâce à des transformations verticales que nous avons abordées dans le Chapitre 3. Ainsi, le framework est intéressant également de par le fait que ces modèles se situent à différents niveaux d'abstractions et peuvent être obtenus par transformations successives comme pour les modèles CIM, PIM et PSM de l'approche MDA. Ainsi, le modèle des tâches et concepts correspond au CIM, l'AUI au PIM, la CUI au PSM et enfin la FUI au code final.

Le processus de base consiste à concevoir manuellement un modèle des tâches et concepts pour ensuite le transformer en AUI qui elle-même est transformée en CUI puis en FUI. Néanmoins, le framework laisse la possibilité au développeur de commencer le processus à un autre niveau. Il est par exemple possible de concevoir une AUI sans avoir un modèle des tâches et concepts. Les autres niveaux peuvent également être utilisés comme points d'entrées. Ainsi, le framework est plus flexible que s'il ne permettait que des approches complètement "top-down" ou "bottom-up".

Ce framework est globalement accepté dans la communauté UI et certains UIDLs se basent dessus pour proposer leurs modèles et processus de conception d'interfaces utilisateurs, d'autres, sans y faire explicitement référence sont assez proches de celui-ci. Cette catégorie regroupe donc ces UIDLs.

Software Engineering for Embedded Systems using a Component-Oriented Approach (SeescoaXML)

Software Engineering for Embedded Systems using a Component-Oriented Approach (SeescoaXML [60]) consiste en une suite de modèles et d'un mécanisme permettant de produire automatiquement des FUIs pour différentes plateformes qui pourraient être équipées de différents appareils d'entrée/sortie fournissant des modalités variées (par exemple, un joystick). Ce système est dépendant du contexte étant donné qu'il est d'abord exprimé d'une manière indépendante de la modalité puis connecté à une spécialisation pour chaque plateforme spécifique.

La sensibilité au contexte de l'interface utilisateur se concentre ici uniquement sur les variations des plateformes et non sur les variations au niveau de l'utilisateur ou de l'environnement. Le contexte d'utilisation est donc ici réduit à un seul des trois critères définis par le framework.

Une AUI contient les spécifications pour les différents mécanismes de rendu (aspect présentation) et leur comportement (aspect dialogue). Cette AUI est écrite avec un UIDL à notation XML et est ensuite transformée en des spécifications liées à une plateforme grâce à des transformations XSLT. Ces nouvelles spécifications sont alors connectées à une description haut niveau des appareils d'entrée/sortie.

Pour produire des interfaces utilisateurs sensibles au contexte, une traduction a lieu au niveau abstrait avant de continuer dans le framework pour chaque configuration spécifique (correspondant à une plateforme). L'approche n'utilise pas les modèles de concepts et de tâches. Le point d'entrée de cette approche d'ingénierie se situe donc au niveau des interfaces utilisateurs abstraites (AUIs). Dans une version étendue de Seescoa, l'AUI est obtenue à partie d'un modèle de tâches qui est progressivement transformé en un arbre de priorités [86].

Aucun outil supportant Seescoa n'est disponible, il en va de même pour les méta-modèles. Le site officiel [14] ne donne accès qu'à diverses publications et à des informations générales sur le projet.

Model-based lAnguage foR Interactive Applications XML (MARIA XML)

Model-based lAnguage foR Interactive Applications XML (MARIA XML [73]) permet de définir des modèles de tâches et de domaines et des AUIs, descriptions d'interfaces utilisateurs indépendantes de toute plateforme, de toute modalité d'interaction (par exemple des clics, par la voix, via des capteurs qui suivent le mouvement des yeux, etc.) et de tout langage d'implémentation. Il est également possible de définir des CUIs qui, elles, sont dépendantes de la plateforme visée (PC de bureau, mobile, vocale, multimodale pour PCs de bureau et multimodale pour dispositifs mobiles) mais restent indépendantes du langage d'implémentation. Le langage supporte les comportements dynamiques, les événements, les applications Internet riches, les interfaces utilisateurs ciblant plusieurs plateformes et en particulier celles basées sur les services Web [86].

Pour chaque plateforme, MARIA XML propose un méta-modèle de CUI correspondant qui est un raffinement de l'AUI générale. Ainsi, un modèle CUI spécifie comment tel élément abstrait d'un modèle AUI peut être représenté sur la plateforme ciblée. Par exemple, un "Single Choice interactor" peut être implémenté via un groupe de radios ou une "listbox" pour une application bureau tandis qu'il pourrait être représenté par une liste de messages vocaux pour chaque option dans le cas d'une plateforme vocale.

MARIA XML est le successeur du langage Tool for Design and Development of Multi-platform Applications (TERESA). MARIA XML est un des langages à avoir été soumis pour être standardisé par le W3C [32].

MARIA XML permet de décrire, non seulement les aspects de présentation mais aussi le comportement interactif grâce à divers modèles ("Data Model", "Event Model", "Dialog Model", etc.).

MARIA XML est supporté par un outil qui s'appelle MARIAE (MARIA Environment) Tool [1]. Il exploite les différents modèles MARIA XML pour la conception et le développement d'applications interactives basées sur les services Web pour différents types de plateformes (bureau, mobile, vocale, multimodale pour PCs de bureau et multimodale pour dispositifs mobiles). Lorsque l'utilisateur crée un nouveau projet, il peut créer soit une interface abstraite (AUI) soit

1. `http://giove.isti.cnr.it/tools/MARIAE/home`

une interface concrète (CUI) pour les différentes plateformes. Il est possible au final de générer du code HTML ou JSP.

En théorie, cet outil fonctionne, des vidéos montrant quelques projets sont disponibles sur le site officiel du projet. Néanmoins, après quelques tests, il semblerait qu'il y ait quelques problèmes. Par exemple, l'outil ne reconnait pas les fichiers d'extension ".meprj" alors qu'il s'agit des fichiers des projets, il n'arrive pas à les ouvrir bien qu'il indique que le projet soit chargé. Il en va de même lorsque l'utilisateur souhaite créer un modèle AUI. Ceux-ci, comme pour les autres modèles, ont une extension ".xml". Une fois le modèle créé, si l'utilisateur souhaite l'ouvrir, l'outil lui indique qu'il ne reconnait pas le type du fichier. L'outil a été testé dans ses versions 1.5.0 et 1.5.3 (la dernière en date). Les versions antérieures n'ont peut-être pas ce problème.

D'autres outils, moins intéressants pour nous, sont également disponibles. Par exemple, l'outil Reverse Maria [2], permet de créer une interface concrète de bureau à partir d'une page Web par rétro-ingénierie. Ces outils n'ont pas été testés.

En ce qui concerne les méta-modèles, ceux-ci sont visibles dans différents travaux sur cet UIDL mais sont de simples images. Ils ne sont pas accessibles dans un format directement exploitable.

USer Interface eXtensible Markup Language (UsiXML)

USer Interface eXtensible Markup Language (UsiXML [59]) est un langage structuré selon les différents niveaux d'abstraction du framework. Ainsi, il propose cinq types de modèles : modèle des tâches, modèle de domaine, AUI, CUI et FUI. Le développement de l'interface utilisateur se fait par transformations successives.

UsiXML est en théorie supporté par une série d'outils. Une partie sont présentés dans [59] :

1. GraphiXML : un éditeur textuel et graphique de modèles UsiXML qui permet également de générer du code XHTML 1.0 ou Java Swing.
2. VisiXML : un éditeur graphique de modèles CUI UsiXML.
3. SketchiXML : un éditeur permettant de réaliser des esquisses de CUI.
4. IdealXML : un éditeur de modèles de tâches et domaine, d'AUI et de relations inter-modèles.
5. KnowiXML : un éditeur de modèles de tâches et d'AUI. Il permet également de transformer un modèle de tâches en AUI.
6. ReversiXML : un outil de rétro-ingénierie. Il permet de générer une AUI ou CUI à partir de code HTML 4.0.
7. TransformiXML : un outil de transformations qui prend en entrée et produit n'importe quel type de modèle UsiXML.

2. `http://giove.isti.cnr.it/tools/ReverseMARIA/home`

Ces outils sont censés être disponibles sur l'ancien site officiel d'UsiXML [3] mais, en réalité ce n'est pas le cas. Le nouveau site officiel [4] ne donne pas non plus accès à ces outils.

Deux autres outils de rendus ont également été implémentés :

1. RenderXML [5] : un moteur de rendu d'interfaces qui permet de transformer des interfaces décrites avec UsiXML en des interfaces finales sur de multiples plateformes. Cet outil n'est malheureusement pas téléchargeable et donc non accessible.
2. InterpiXML : un autre moteur de rendu permettant de générer des interfaces graphiques à partir de modèles UsiXML (de CUI plus précisémment). Des démonstrations sont trouvables sur Internet mais cet outil n'est pas téléchargeable non plus.

Au final, ces outils, pourtant décrits dans des articles scientifiques et paraissant tout à fait intéressants dans le cadre de ce projet, ne sont pas mis à disposition des chercheurs.

Au niveau de la disponibilité des méta-modèles, ceux-ci sont à la fois montrés dans différents articles mais également disponibles dans un format exploitable, en l'occurence, en Ecore [11].

Comme MARIA XML, UsiXML a été soumis au W3C [32].

6.1.4 UIML et ses dérivés

Cette sous-section concerne UIML et un langage qui est dérivé de ce dernier : DISL.

User Interface Markup Language (UIML)

User Interface Markup Language (UIML [35]) est un UIDL spécifié par OASIS [16], un consortium qui s'occupe du développement et de l'adoption de standards dans le domaine de l'e-business et des services Web. L'objectif d'UIML est de fournir une représentation canonique de n'importe quelle interface utilisateur qui peut ensuite être traduite dans des langages d'implémentation [64]. UIML n'est qu'une pièce d'un puzzle qui doit être utilisé avec d'autres technologies, par exemple, un langage de modélisation de tâches d'interfaces utilisateurs, des algorithmes de transformations, etc.

UIML permet de décrire l'apparence, l'interaction et la connexion de l'interface utilisateur avec la logique de l'application. Il y a quatre concepts clés [64] :

1. UIML est un méta-langage, un peu comme XML. Bien qu'il soit lui-même défini par un schema XML, il définit un petit ensemble de balises générales. Par exemple, la balise <part> représente une "partie" d'une interface, la

3. http://www.usixml.org/
4. http://www.usixml.eu/
5. http://sourceforge.net/projects/renderxml/

balise <property>, une "propriété" d'une "partie". Ces balises sont indépendantes de la modalité, de la plateforme et du langage d'implémentation. Pour utiliser UIML, il est nécessaire de définir un vocabulaire un peu comme un DTD va définir le vocabulaire d'un langage basé sur XML. Ce vocabulaire spécifie un ensemble de "classes" de "parties", par exemple une classe "Bouton", et des "propriétés" de ces "classes". En fonction de leur besoins, les développeurs conçoivent le vocabulaire nécessaire.

2. UIML sépare les éléments d'une interface utilisateur. Cette séparation identifie quelles parties composent l'interface, le style de présentation de chaque partie, le contenu de chaque partie (du texte, des images, du son) et le lien entre le contenu et des ressources externes, le comportement des parties exprimées par un ensemble de règles avec des conditions et actions appliquées à des "parties", la connexion de l'interface utilisateur avec le monde extérieur et la définition du vocabulaire des "classes".
3. UIML voit l'interface utilisateur comme un arbre de parties d'interface qui change durant son cycle de vie. L'arbre initial représente l'interface utilisateur telle qu'elle est au début de l'utilisation. L'arbre initial des parties peut ensuite changer dynamiquement avec l'ajout ou la suppression de parties. Par exemple, si l'utilisateur ouvre une nouvelle fenêtre, cela peut revenir, au niveau UIML, à l'ajout d'un sous-arbre de parties. UIML fournit des éléments pour décrire la structure de l'arbre initial et pour la modifier dynamiquement.
4. UIML permet aux parties d'interfaces et aux arbres des parties d'être rassemblés dans des *templates*. Ces *templates* peuvent être alors réutilisés dans d'autres conceptions d'interfaces. Ainsi, la notion de réutilisabilité est intégrée à UIML.

Des outils générateurs sont disponibles afin de transformer des documents UIML en interfaces utilisateurs finales :

1. UIML.net : outil de génération de code pour la plateforme .Net. Le projet semble être à l'arrêt.
2. JUIML : outil de génération de code Java. Il s'agit d'une version beta qui n'a, de plus, pas été mis à jour depuis 2009.
3. PyUIML (Python UIML XML Parser) : outil de génération de code Python ou HTML. D'autres langages sont prévus. Cet outil n'en est qu'à sa version 0.0.2, la dernière mise-à-jour datant de février 2013.

Transformation-based Integrated Development Environment (TIDE) est un IDE qui permet d'écrire du code UIML pour ensuite générer l'interface graphique finale dans un langage donné, à savoir HTML ou Java. Il a été développé avec l'idée que le développeur qui crée une interface utilisateur dans un langage abstrait comme UIML pour ensuite être transformée dans un ou plusieurs langages concrets va devoir faire plusieurs essais avant d'obtenir ce qu'il souhaite. Le développeur conçoit en UIML ce qu'il pense être approprié lorsque ce sera rendu dans un langage concret puis fait des changements en fonction du résultat réel. Cet IDE est donc conçu pour aider le développeur dans ce processus et va

montrer trois choses : l'interface abstraite en code source UIML (sous forme de texte ou d'arbre), le résultat du rendu dans le langage et la relation entre les deux éléments. La sélection d'un élément du code UIML a pour effet de mettre en évidence cet élément dans l'interface graphique générée et inversémment [35].

L'outil n'a pas été retrouvé. D'après [35], il se trouve sur le site d'Harmonia mais l'adresse exacte n'est pas précisée. Le méta-modèle d'UIML est uniquement disponible sous forme d'image.

Dialog and Interface Specification Language (DISL)

Dialog and Interface Specification Language (DISL [63]) peut être vu comme un sous-ensemble d'UIML qui étend le langage afin de permettre des descriptions, génériques et indépendantes des modalités, du comportement d'une interface utilisateur. Pour le dialogue avec l'interface, il utilise une spécification orientée états. Ainsi, le comportement est représenté comme un graphe avec des opérations sur des éléments de l'interface utilisateur qui provoquent des changements d'états.

Au niveau des "widgets" génériques, ceux-ci sont introduits dans le but de séparer la présentation de la structure et du comportement, c'est-à-dire, de séparer les propriétés spécifiques aux appareils, utilisateurs et modalités de la présentation indépendante de la modalité. Chaque "widget" générique peut être assigné à un type particulier de fonctionnalité qu'il effectue (champ variable, champ de texte, commande, etc.). Ensuite, un moteur de rendu DISL peut utiliser cette information pour créer des composants d'interface approprié à la modalité d'interaction (vocale, graphique) liée au "widget".

Aucun outil ni méta-modèle n'est apparemment disponible pour ce langage.

6.1.5 Langages visant les applications Web

Les UIDLs spécifiques aux applications Web restent intéressants étant donné que les configurateurs Web sont très nombreux.

XForms

Spécification du W3C, XForms [87] est un langage qui, au départ, était destiné à être intégré à des documents HTML-XHTML dans le but de créer des formulaires avec séparation de la présentation et des données au contraire des formulaires HTML. Le but était d'améliorer les aspects réutilisation et indépendance vis-à-vis de la machine. Désormais, XForms peut être intégré à tout langage balisé compatible [86].

XForms sépare les formulaires XHTML en trois modèles [86] :

1. Modèle XForms : permet de définir un formulaire et contient tous les éléments qui définissent le modèle. Il peut y avoir autant d'éléments que l'on souhaite dans un document.
2. Données d'instance : permet d'envoyer des données collectées sous format XML. Ces données sont connectées avec des éléments de formulaires.

3. Interface utilisateur : il s'agit d'une AUI qui définit des concepts tels que "output", "input", "submit", "select", "alert", "label", etc.

Les contrôles d'interfaces utilisateurs définis par XForms sont génériques et indépendants de la plateforme et des modalités d'interactions [87].

XForms n'est pas forcémment considéré comme un UIDL mais le fait qu'il permette de définir des interfaces abstraites peut faire de lui un langage potentiellement intéressant. XForms est supporté par un grand nombre d'outils, répertoriés dans [29]. Au niveau méta-modèle, en plus d'une image de ce dernier reprise par [86], le schéma XML [6] de XForms est également accessible.

eXtensible user-Interface Markup Language (XICL)

eXtensible user-Interface Markup Language (XICL [47]) permet de développer des composants d'interfaces utilisateurs ("User Interface Components") pour des applications tournant sur navigateurs. Il permet de créer de nouveaux composants à partir de composants HTML et d'autres composants XICL. Une description XICL est traduite en code Dynamic HTML (DHMTL) qui comprend les technologies côté client recommandées par le W3C (notamment HTML, CSS, Javascript).

Un document XICL est composé d'une description de l'interface en terme d'éléments HTML et XICL et de plusieurs composants abstraits que sont les propriétés, les méthodes, les événements et la structure.

Aucun outil n'est disponible pour ce langage. Une image d'un méta-modèle XICL est reprise dans [86] mais celle-ci semble grandement simplifier ce dernier.

eXstensible user-Interface Markup Language (XIML)

eXstensible user-Interface Markup Language (XIML [77]) est un langage permettant de représenter des données d'interactions pour les sites Web et applications Web [34]. XIML peut représenter des éléments abstraits et concrets.

Le langage est composé principalement de quatre types de composants : modèles, éléments, attributs et relations entre éléments.

XIML distingue deux types de modèles : le modèle interface et les composants du modèle. Les seconds sont des sous-modèles du modèle interface, le tout formant un document XIML. Les composants de modèle, à savoir les composants tâche, domaine, utilisateur, présentation, dialogue, plateforme, préférences et le modèle général, contiennent des informations spécifiques à une dimension de l'interface.

Un élément est une information qui décrit un composant. Par exemple, un élément du composant "presentation" est une unité d'information décrivant l'apparence de l'interface utilisateur. Un élément de présentation peut être décomposé en éléments. C'est le cas d'un élément fenêtre contenant une table qui contient elle-même des lignes, etc.

6. `http://www.w3.org/MarkUp/Forms/2002/XForms-Schema.xsd`

Un attribut représente une unité d'information déclarative lié à un modèle d'interface, un composant du modèle ou un élément. Un attribut a notamment une liste de valeurs possibles, une valeur par défaut et un type.

Une relation est un lien entre éléments et/ou modèles. L'élément référencé est spécifié grâce à un attribut de référence qui contient l'identifiant de cet élément.

Le modèle de présentation est composé d'une série d'éléments qui correspondent aux "widgets" de l'interface utilisateur et les attributs représentent leurs caractéristiques comme la taille, la couleur, etc. Au niveau présentation, les relations sont des liens entre labels et les "widgets" que ces labels décrivent.

Il n'y a pas vraiment d'outils téléchargeables pour XIML. Par contre, [34] permet à l'utilisateur de créer son site web avec XIML pour peu qu'il soit enregistré sur le site. Le méta-modèle de XIML est disponible uniquement sous forme d'image, il a d'ailleurs été repris dans [86].

6.1.6 Autres langages

Cette sous-section présente les UIDLs qui ne rentraient pas dans les catégories précédentes.

Generalized Interface Markup Language (GIML)

Generalized Interface Markup Language (GIML) est un langage de description d'interfaces utilisateurs utilisé dans le cadre du projet Generalized Interface Tool Kit (GITK [58]). GIML sépare la présentation des fonctionnalités. GIML décrit les fonctionnalités de dialogues tandis que l'interface utilisateur (la présentation) est dérivée de fichiers XSL qui viennent de profils utilisateurs et systèmes. Cette information est fusionnée avec les descriptions fonctionnelles en utilisant XSLT [7] pour former une description d'interface finale [86].

GIML n'utilise pas des termes habituels tels que "push button" (qui n'aurait aucun sens pour une interface vocale) ou "scrollbar" mais plutôt des termes comme "action", "data-entry/value- choice/single/limited" qui sont plus neutres, abstraits. L'objectif est d'utiliser les "patterns" d'interface [8] dans le futur. Néanmoins, le projet semble être arrêté, la dernière modification du site datant de 2004 [5].

En résumé, l'interface graphique finale est obtenue après transformations successives en partant de la description GIML et en lui appliquant des fichiers XSLT.

Seule la documentation de GITK est téléchargeable, pas l'outil en question [5]. Aucun méta-modèle de GIML n'est disponible.

7. http://www.w3schools.com/xsl/
8. http://ui-patterns.com/

Multiple Device Markup Language (MDML)

Multiple Device Markup Language (MDML [54]) est un langage basé sur XUL. Il permet de spécifier les concepts de navigation, de structuration et de composants d'une interface graphique générique. Un système de règles, composé de fichiers de règles XML et d'un moteur de règles, est également défini pour transformer les spécifications pour différents appareils en fonction de leurs caractéristiques telles que la mémoire, les capacités d'affichage et la représentation interne des données.

Bien qu'une représentation générique peut être utilisée pour tous les appareils, cette approche est parfois considérée comme trop simpliste et donc les règles produites sont différentes en fonction du type d'appareil visé. Quatre profils sont gérés : l'application bureau ("Desktop"), l'application Web ("Web"), l'application mobile ("Mobile") et l'application avec reconnaissance vocale ("Voice").

Au niveau de l'architecture, nous avons donc un moteur de règles qui va analyser le fichier de règles XML et un moteur d'affichage qui parcourt et analyse le fichier de présentation MDML. Ensuite, un gestionnaire qui produit un autre fichier XML et un générateur s'occupent de générer le code de l'interface finale. Toutes ces parties sont des applications Java. MDML est donc un framework complet.

Au niveau des outils, ceux-ci sont donc intégrés framework. Néanmoins, [15] ne le rend pas disponible. Au niveau du méta-modèle, le schema XSD est normalement téléchargeable. Cependant, le lien de téléchargement ne fonctionne plus [15].

Simple Unified Natural Markup Language (SunML)

Simple Unified Natural Markup Language (SunML [74]) est un langage permettant de spécifier des interfaces utilisateurs concrètes qui peuvent être transformées pour différents dispositifs tels que des PCs, PDAs ou des dispositifs à reconnaissance vocale. L'innovation principale de ce langage est la possibilité de spécifier des composants dynamiquement.

SunML utilise des concepts d'interfaces utilisateurs tels que "Element" qui contient des informations donnés par l'utilisateur au système, "Field", qui permet de récupérer de telles informations, "Link" qui référence l'exécution de quelque chose en particulier, "List" correspond à une liste d'élements avec comme objectif de les présenter comme une liste de valeurs ou d'actions, "Dialog" est une liste qui regroupe des "widgets" dans le but de créer un dialogue entre eux.

Bien que ces concepts soient limités, il est possible de composer les "widgets" pour en créer de nouveaux, plus complexes. Par exemple, une liste d'"éléments" peut être traduite par une "combobox" et une liste de "liens" peut être traduite par un menu [86].

SunML est supporté par un IDE nommé "Adaptable & Mergeable User INterface" (AMUSINg) qui permet de concevoir des modèles SunML et de générer des interfaces finales en Java Swing [74]. Cependant, les recherches effectuées

n'ont pas permis d'y avoir accès. Une image du méta-modèle est accessible.

TADEUS-XML

TADEUS-XML [62] est le langage qui met en pratique l'approche basé sur les modèles du même nom [86]. Un document TADEUS-XML décrit une interface utilisateur en deux parties : une partie présentation et une partie fonctionnalités appelé l'"abstract interaction model". TADEUS-XML propose plusieurs niveaux d'abstraction. Le premier est le "XML-based Interaction Model" composé de "User Interface Objects" (UIOs) qui ont chacun leurs propres attributs précisant leur comportement et qui sont structurés hiérarchiquement. Grâce au "XML-based Device Definition", il est possible d'obtenir un second modèle dépendant de la plateforme cible à partir du premier modèle. Ce second modèle reste un modèle abstrait mais intègre certaines contraintes spécifiques à la plateforme cible (il s'agit d'un "mapping" entre UIO abstraits et UIO concrets). Enfin, le "XSL-based Model Description" peut être dérivé du second modèle en se basant sur la connaissance que le concepteur a des UIOs disponibles pour telle ou telle spécification. Une interface utilisateur finale est générée à partir de ce dernier modèle.

Lors de l'écriture de [81], TADEUS-XML était encore en développement et les outils supportant ce langage pas encore disponibles. Après recherches, aucun outil ne semble disponible même à l'heure actuelle ce qui est également le cas du ou des méta-modèles du langage.

6.1.7 Récapitulatif

Le tableau à la Figure 6.1 récapitule les résultats obtenus pour les différents UIDLs présentés vis-à-vis des critères mentionnés en début de section.

Parmi les UIDLs étudiés, UsiXML parait le plus adapté pour atteindre nos objectifs. Le fait d'être basé sur un framework reconnu mais surtout de fournir des méta-modèles exploitables sont de gros avantages par rapport à d'autres UIDLs. Malheureusement, le manque d'outils réellement disponibles lui fait cruellement défaut. Bien que des outils sont en théorie implémentés, nous n'y avons pas accès.

Ensuite viennent MariaXML et UIML, deux spécifications, respectivement du W3C et d'OASIS, qui font partie des UIDLs dont les informations sont les plus nombreuses. Ils sont également intéressants en théorie mais le fait que les méta-modèles soient disponibles uniquement sous forme d'images et que les outils soit ne fonctionnent pas correctement soit ne sont pas disponibles les rend finalement inintéressants vis-à-vis de nos objectifs.

Les autres UIDLs, en plus d'avoir les mêmes défauts que ces derniers en ont également d'autres comme la difficulté d'accéder à l'information ou simplement la faible quantité d'informations. Ils ne constituent dès lors pas des solutions envisageables non plus.

D'un autre côté, nous avons le cas de XForms. Celui-ci remplit tous les critères présentés. Néanmoins, étant donné que XForms, comme son nom l'indique,

UIDLs	Indépendance plateformes	Disponiblité d'outils	Disponibilité des méta-modèles	Accessibilité des informations	Standard	Indépendance modalités
SeescoaXML	Oui	Non	Non	Moyenne	Non	Oui
MARIA XML	Oui	Oui***	Oui*	Elevée	Oui	Oui
UsiXML	Oui	Non**	Oui	Elevée	Oui	Oui
UIML	Oui	Non**	Oui*	Elevée	Oui	Oui
DISL	Oui	Non	Non	Moyenne	Non	Oui
Xforms	Oui	Oui	Oui	Elevée	Oui	Oui
XICL	Oui	Non	Oui*	Faible	Non	/
XIML	Oui	Oui	Oui*	Elevée	Non	/
GIML	Oui	Non**	Non	Faible	Non	Oui
MDML	Oui	Non**	Non	Moyenne	Non	Oui
SunML	Oui	Non**	Oui	Moyenne	Non	Oui
TADEUS-XML	Oui	Non	Non	Faible	Non	Oui

(*) : les méta-modèles ne sont disponibles que sous forme d'images.
(**) : des outils ont "théoriquement" été implémentés mais ils ne sont pas disponibles ou ont été abandonnés.
(***) : les tests de l'outil ont révélé des problèmes qui ont empêché de vérifier son bon fonctionnement.

FIGURE 6.1 – Tableau récapitulatif des UIDLs

se concentre sur la description de formulaires, ce n'est pas non plus une bonne solution sur le long terme.

Une solution concrète au problème posé étant souhaitable, l'utilisation d'un UIDL, à l'heure actuelle, n'est finalement pas réellement intéressante. En effet, pour obtenir une interface graphique finale, il faudrait non seulement créer un générateur produisant une description d'interface, sans doute une AUI dans le cas d'UsiXML, à partir d'un modèle TVL mais également un second générateur qui servirait à concevoir l'interface graphique concrète, implémentée dans un langage spécifique.

Suite à ces observations, il apparait donc nécessaire de se tourner vers une technologie cible spécifique, à savoir HTML5.

6.2 HTML5

Bien que les UIDLs possèdent des avantages certains, le fait que certains ne soient pas supportés par des outils et que d'autres ne donnent pas accès à leur méta-modèles est un handicap non négligeable. Étant donné que la solution à apporter n'est pas une solution abstraite mais bien une solution fournissant des résultats utilisables par des utilisateurs lambda, il est nécessaire de trouver une technologie alternative.

Pour cela, nous allons nous tourner vers HTML5 [33] [13], la dernière version en date du standard HTML qui n'est lui-même pas encore un standard W3C. Bien que la spécification d'HTML5 soit complète depuis le 17 décembre 2012, il ne devrait devenir un standard W3C qu'en 2014 [28].

HTML5 a été conçu par le Web Hypertext Application Technology Working Group (WHATWG [9]) en réponse à la lenteur du développement des standards Web suivis par le W3C et leur décision d'abandonner la technologie HTML au profit des technologies basées sur XML, notamment XHTML 2.0 [84]. Depuis 2006, les deux groupes coopèrent au développement d'HTML5 [33].

HTML5 apporte un grand nombre de nouveautés par rapport aux anciennes versions d'HTML et supprime certaines constructions jugées inutiles. Ainsi, HTML5 propose une série d'APIs pour la lecture de vidéos et de sons grâce aux nouvelles balises *video* et *audio* permettant de remplacer le plugin Adobe Flash Player [10], pas toujours stable. D'autres APIs existent, permettant notamment la géolocalisation, le "drag and drop", de dessin 2D avec la balise *canvas*, le stockage de données sur le client, la création de Threads appelés "Web Workers", etc.

En dehors de cela, ce qui est sans doute plus intéressant si nous souhaitons créer des configurateurs Web, ce sont les nouvelles balises et attributs. Outre les balises prévues pour les APIs de vidéos ou dessins par exemple, HTML5 propose également une série de balises dites "sémantiques" : les balises *article*, *header*, *footer*, *section*, *nav*, *aside* et *hgroup*. Dans les anciennes versions d'HTML, la

9. `http://www.whatwg.org/`
10. `http://get.adobe.com/fr/flashplayer/`

structuration des pages se faisait grâce aux balises *div* utilisées à toutes les sauces sans que le navigateur "sache" ce que signifient ces balises, elles ne disent rien sur leur contenu. Avec les nouvelles balises sémantiques, les navigateurs savent qu'une balise *nav* représente un ensemble de liens de navigation, donc un menu, une balise *header* représente le bloc d'en-tête, etc. Ces balises définissent clairement leur contenu et les navigateurs peuvent donc les traiter comme il se doit. Elles sont également plus faciles à comprendre pour les développeurs, ils s'y retrouveront plus facilement lorsqu'ils devront effectuer des modifications sur ses pages Web.

D'autres balises et attributs sont également ajoutés, permettant de créer de nouveaux "widgets" tels que les *sliders*, les *datalist* qui proposent une série de valeurs possibles au fur et à mesure que l'utilisateur entre des caractères dans le champ prévu, des champs prévus pour les dates, les URLs, les emails, des numéros de téléphones, des champs de recherches, etc. Au niveau des attributs, plusieurs sont ajoutés à la balise *input* qui représente un champ d'un formulaire. Ceux-ci permettent notamment de définir des valeurs minimale et maximale qui sont alors vérifiées directement par le navigateur, sans requérir du Javascript, d'activer le mécanisme d'auto-complétion, etc. HTML5 réduit donc l'utilisation de langages de scripts grâce aux nouvelles balises et attributs, le contrôle et la validation des champs d'un formulaire étant grandement simplifiée.

Au niveau des inconvénients, nous avons le problème majeur auquel sont confrontés les développeurs Web concernant le rendu des pages Web sur les différents navigateurs. En effet, ils doivent faire attention à ce que ces pages soit rendues correctement, au moins sur les navigateurs les plus connus tels que Internet Explorer, Mozilla Firefox, Google Chrome, etc. La difficulté vient du fait que chaque navigateur interprète différemment les pages HTML.

Lié à ce problème, l'inconvénient principal d'HTML5 est qu'aucun navigateur ne gère toutes les constructions et APIs que le langage a apporté. Néanmoins, tous les constructeurs des navigateurs majeurs, à savoir, Safari, Chrome, Firefox, Opera et Internet Explorer, continuent à améliorer la compatibilité de ceux-ci de version en version en gérant de plus en plus de constructions du langage [7].

7 Travaux liés

Ce chapitre est basé sur les recherches de l'équipe PReCISE concernant la majorité des travaux présentés.

La génération d'interfaces homme-machine à partir de modèles est un sujet qui a pris de l'importance depuis quelques années. Nous avons d'ailleurs vu, dans la Section 6.1 concernant les UIDLs, qu'il existe de nombreuses solutions basées sur la modélisation d'interfaces graphiques et la génération de telles interfaces à partir des modèles. Ces UIDLs sont en général supportés par un framework ou des outils qui effectuent des transformations de modèles sur les modèles de description pour arriver au final au code d'implémentation. Ces approches sont connues sous le nom de "Model-Based User Interface Development" (MBUID).

Par contre, ces approches n'abordent pas les problèmes spécifiques à la génération de configurateurs, à savoir l'intégration de mécanismes de raisonnement pour contrôler et propager les choix de l'utilisateur en fonction des règles de configuration cachées. Les interactions de l'utilisateur doivent se refléter sur l'interface du configurateur, par exemple en grisant des options lors de la sélection ou la désélection d'une autre option. De plus, nous pouvons profiter du modèle de variabilité (le FM) pour utiliser des solveurs efficaces pour gérer le processus de configuration.

En ce qui concerne la configuration de produits/systèmes grâce aux FMs, la plupart des outils permettent de créer et de configurer des FMs sous forme d'arbre. C'est notamment le cas de FeatureIDE, utilisé pour réaliser le FM présenté à la Figure 4.2, pure : :variants [38] et Feature Modeling Plug-in [36]. Ces outils possèdent une interface graphique permettant à l'utilisateur de sélectionner ou désélectionner des *features* avec une propagation des contraintes automatique.

La Figure 7.1 illustre le type d'interface graphique de configuration de FM que proposent ces outils avec l'exemple de FeatureIDE. FeatureIDE permet de créer autant de configurations que souhaité. Nous remarquons que des options ont été grisées suite à la propagation des contraintes. L'utilisation de cet outil ne peut se faire qu'intégré à l'environnement Eclipse.

Néanmoins, ce type d'interface n'est pas vraiment adapté aux utilisateurs lambda, habitués à manipuler des interfaces graphiques composées de différents "widgets" ("radios", "checkbox", "listbox", etc.), des fenêtres, onglets, etc. De plus, elles ont également le défaut lié à leur représentation graphique du FM, le fait de ne pas gérer les attributs de *features*, comme expliqué dans le Chapitre 4.

La suite d'outils AHEAD, proposé par [49], permet par contre de générer des interfaces graphiques Java assez simples contenant divers "widgets" comme des "checkbox" ou des boutons "radio" tout en utilisant des annotations que fournit la syntaxe GUIDSL. Parmi ces annotations, "disp" permet de spécifier le label d'une *feature* à afficher, "help" de fournir un texte d'aide, "hidden" de

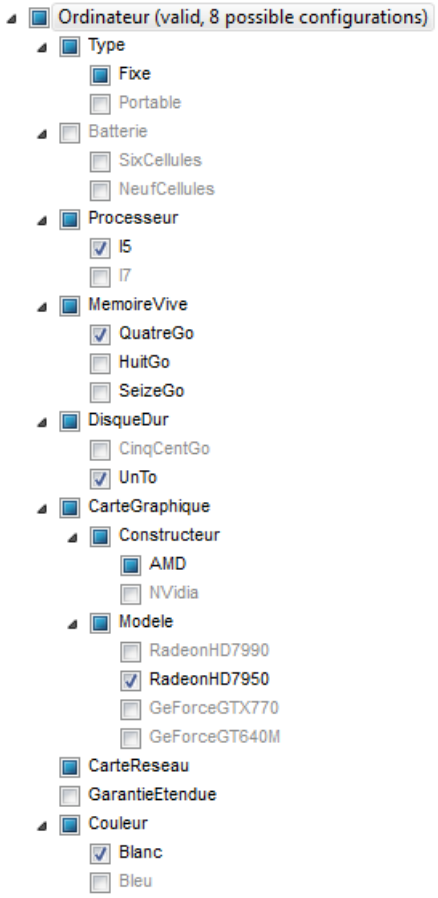

FIGURE 7.1 – Configuration Ordinateur avec FeatureIDE

cacher une *feature*, "tab" de définir un nouvel onglet dont la racine est la *feature* en question, etc. Les annotations de GUIDSL sont donc assez similaires à ce que propose le langage FCSS présenté dans la Section 5.2, elles ont d'ailleurs servi d'inspiration pour la conception du langage [41]. De plus, les interfaces générées restent assez basiques.

Malheureusement, ces configurateurs ne peuvent pas être mis à disposition des clients étant donné qu'ils sont dépendants de l'outil en question, ils ne peuvent être inclus à une interface graphique existante comme une page Web d'une entreprise.

Plusieurs auteurs ont également combiné le MBUID avec les FMs. Parmi ceux-ci, nous pouvons citer Pleuss et al. [75] [76] qui dérivent automatiquement une interface graphique correspondant à un produit configuré spécifique. Une telle interface graphique affiche les différentes caractéristiques du produit en question, l'utilisateur pouvant ensuite faire son choix sur le produit qu'il souhaite parmi ceux affichés. Leurs objectifs ne sont pas les mêmes que les nôtres. Tandis qu'ils visent à générer des interfaces graphiques affichant les caractéristiques de produits dérivés de la ligne de produits, nous visons à générer des interfaces graphiques de configurateurs permettant aux clients de personnaliser leurs produits selon leurs besoins.

Schlee et Vanderdonckt [79] ont également travaillé sur la génération d'interfaces graphiques à partir de FMs. La variabilité de l'interface est modélisée avec un FM utilisé pour dériver cette interface. Leur travail est assez proche de la solution présentée dans ce mémoire mais il semble abandonné depuis 2004. De plus, ils ne s'occupent pas de certains problèmes tels que la personnalisation de la génération et la gestion du raisonnement.

Deuxième partie

Contributions

8 Architecture

8.1 Présentation

Ce chapitre présente l'architecture de la solution proposée au problème abordé dans les chapitres précédents. Pour rappel, les configurateurs actuels sont pour la plupart codés manuellement. Les contraintes régissant les différents éléments sont souvent mélangées au code de l'interface graphique. La maintenance de ceux-ci est alors rendue difficile et certaines règles sont parfois tout simplement oubliées. La solution exposée dans ce chapitre est basée sur la séparation des préoccupations, approche bien connue en ingénierie du logiciel mais en pratique rarement appliquée en ce qui concerne les configurateurs actuels.

Tout d'abord, il est important de bien faire la différence entre le générateur, qui ne produit que la partie statique d'un configurateur final et un configurateur en lui-même qui dispose d'une partie statique et de composants gérant son comportement, la partie dynamique. Les composants gérant la dynamique du configurateur, quant-à-eux, ne sont pas générés et sont donc communs à tous les configurateurs créés via la méthode décrite dans le chapitre suivant.

L'architecture voulue pour un configurateur créé à partir d'un modèle TVL est une variante d'une architecture Modèle-Vue-Contrôleur [78]. Cette approche a été proposée par Boucher et al. [40]. La Figure 8.1 illustre cette architecture.

Les cases foncées correspondent aux contributions de ce mémoire, les cases plus claires correspondent à des éléments existants ou considérés comme tels ou encore simplement donnés dans le cas du FM. L'idée derrière cette architecture est de bien séparer les préoccupations, la première étant l'aspect visuel du configurateur et la seconde étant la gestion du raisonnement sur ce configurateur à partir des contraintes définies dans le FM.

8.2 Couches

L'architecture est donc composée de trois couches :

1. La vue est ici l'interface graphique générée, une représentation visuelle de la variabilité modélisée par le FM. Elle fait appel aux fonctions du contrôleur pour la gestion des événements et est mise à jour par celui-ci.
2. Le contrôleur a pour but de faire le lien entre la vue et la partie serveur qui traite le modèle. Il gère les événements générés par les manipulations de l'utilisateur, transmet les changements à la partie serveur, reçoit les nouvelles valeurs et met à jour la vue.
3. Le modèle est le FM TVL, chargé et traduit par un solveur SAT en une contrainte en forme normale conjonctive donc en problème de satisfiabilité à résoudre. La couche modèle contient également une API qui permet d'accéder au solveur afin de modifier l'état courant du modèle, en ajoutant,

modifiant ou retirant des contraintes, ou simplement de récupérer cet état courant, représenté par la liste des *features* qui doivent être sélectionnées.

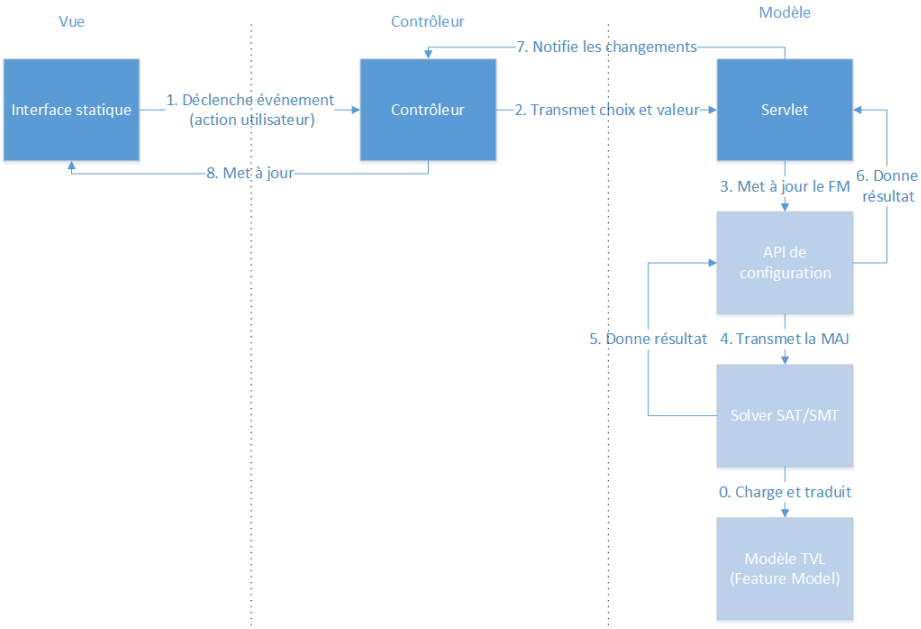

FIGURE 8.1 – L'architecture type MVC

La principale différence de cette architecture par rapport à une architecture MVC classique est le fait que le FM n'est pas directement lié à la vue, en dehors du fait qu'il est utilisé pour générer celle-ci. Cela s'explique par le fait que l'interaction avec le configurateur peut induire des mises-à-jour de l'interface graphique pour lesquelles un comportement spécifique doit être donné. Hors, ce comportement peut être en grande partie généralisé ce qui fait que le contrôleur peut être réutilisé pour chaque interface graphique générée.

8.3 Fonctionnement

Les flèches numérotées (1) à (8) correspondent à la suite d'actions de haut niveau qui se déroulent lorsque l'utilisateur effectue une action sur le configurateur qui déclenche un événement.

La flèche numérotée (0) ne participe pas directement à cette séquence d'actions. En fait, avant que toute action soit gérée dynamiquement par le configurateur, il faut que le côté serveur soit prêt. Pour cela, il doit charger le modèle TVL en question afin de connaître les éléments de ce modèle ainsi que les contraintes qui les régissent. Le solveur SAT traduit ce modèle en une contrainte sous forme normale conjonctive.

Lorsque le solveur a chargé le modèle, le configurateur est prêt à être utilisé. Une action utilisateur (1) provoque un événement qui est géré par une fonction du contrôleur. Celui-ci va alors traiter cet évènement en récupérant l'élément concerné et en le transmettant, ainsi que sa valeur, à la servlet (2) qui est le point d'entrée de la partie serveur contenant le modèle. La servlet, interfacée à l'API de configuration, va ensuite transmettre la mise-à-jour à celle-ci (3) qui va elle-même la transmettre au solveur SAT (4). Le solveur va alors résoudre la nouvelle contrainte obtenue après le changement, ce qui correspond à l'état courant du modèle, et fournir les nouvelles valeurs des différents éléments propagés à l'API (5). L'API va transmettre ce résultat à la servlet (6) qui elle-même notifie le contrôleur (7). Celui-ci a alors pour tâche de propager ces changements à l'interface graphique (8).

Les deux chapitres suivants sont consacrés à la description des différents éléments qui constituent les contributions de ce mémoire, à savoir le générateur d'interface graphique ainsi que les composants contrôleur et serveur communs à tous les configurateurs générés par l'outil en question.

9 Générateur

Maintenant que l'architecture des configurateurs est définie, nous allons voir comment cette architecture est construite. Ce chapitre est consacré à la description du générateur d'interface utilisateur. Ce générateur constitue la contribution principale de ce mémoire. D'autres contributions, moins apportantes, seront présentées au chapitre suivant.

9.1 Remarques préliminaires

Pour rappel, le générateur est codé en Acceleo, une implémentation du standard MOFM2T pour la transformation de modèles en texte, et le code généré est du code HTML5.

La Figure 9.1 schématise le processus de génération. Comme nous le voyons, le générateur prend donc un modèle TVL et un modèle FCSS en entrée et produit un document HTML5 en sortie.

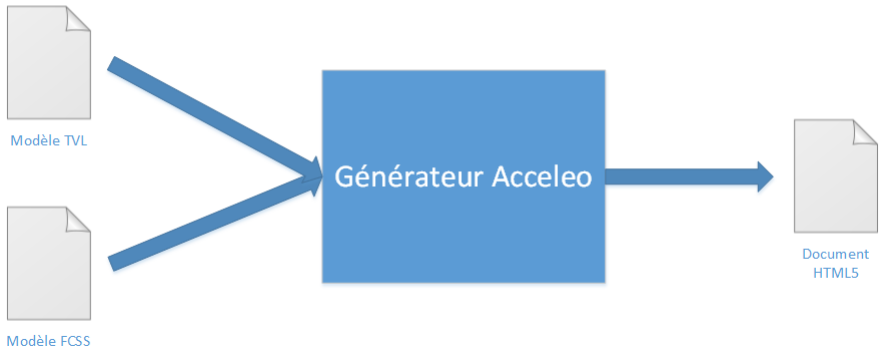

FIGURE 9.1 – Génération de l'interface statique

Ce chapitre a pour but d'expliquer le principe de la génération en Acceleo qui permet donc d'obtenir la partie "vue" de l'architecture d'un configurateur (voir Figure 8.1).

Nous allons tout d'abord plonger dans ce que l'on pourrait qualifier de "tronc commun". Il s'agit des différentes choses à spécifier ou à effectuer avant de pouvoir commencer la génération en elle-même.

Avant toute chose, le fichier de sortie, dans lequel sera écrit le code HTML5 généré, est défini. Le Code 9.1 montre comment le fichier de sortie est défini dans le langage Acceleo.

```
1 [file ('Configurator.html', false, 'UTF-8')]
```

```
2 ...
3 [/file]
```

Code 9.1 – Définition du fichier de sortie

Le nom par défaut du fichier de sortie est donc "Configurator.html" mais rien n'empêche de modifier ce nom, cela n'a pas d'effet sur le fonctionnement global du configurateur final. Le deuxième argument indique si le texte généré doit écraser le texte éventuellement déjà présent ou s'il doit être écrit à la suite. Dans le cas présent, le texte généré écrase le texte déjà présent. Le troisième argument indique que les caractères seront encodés en UTF-8. Toute la génération se fait à l'intérieur de ces balises.

Avant de commencer la génération en elle-même, il est nécessaire de charger les deux modèles, TVL et FCSS. De base, un programme Acceleo ne peut pas prendre plus d'un modèle en entrée. Pour pallier à ce problème, plusieurs solutions sont possibles. Une d'entre elles, mise en pratique, consiste à charger les deux modèles ensemble au début de l'exécution du programme, avant toute autre action. Pour cela, le générateur utilise une *query* Acceleo qui permet d'appeler une méthode Java. Cette *query* appelle la méthode en question qui charge les deux modèles et retourne une liste contenant les références à ces deux modèles. De là, les deux modèles sont accessibles depuis le code Acceleo et peuvent être utilisés pour la génération. De ce fait, le modèle TVL passé en entrée au programme Acceleo ne sert qu'à connaître le chemin du fichier TVL en question et il est impératif que le fichier FCSS ait le même nom que ce dernier et se trouve dans le même dossier.

La *query* "getBothModels" (Code 9.2) permet d'appeler le service Java qui retourne les références aux deux modèles.

```
1 [query public getBothModels(filePathTVL:String,filePathFCSS:String):
2 OrderedSet(EObject) = invoke('TVLtoHTML5.main.ModelsGetter',
3   'getBothModels(java.lang.String,java.lang.String)',
4   Sequence{filePathTVL,filePathFCSS}) /]
```

Code 9.2 – Query getBothModels

Le premier argument de l'*invoke* donne la classe contenant la méthode Java, le second donne le nom de la méthode Java et le type des paramètres attendus et le dernier donne la séquence des arguments à passer à la méthode qui sont les mêmes que ceux fournis en paramètre à la *query*. Dans notre cas, la classe Java s'appelle "ModelsGetter" et se trouve dans le package "TVLtoHTML5.main", la méthode en question s'appelle "getBothModels" et elle prend deux arguments de type "String" qui sont les chemins des fichiers TVL et FCSS donnés en argument de la *query*. Le type de retour est un ensemble ordonné d'*EObject*, un type de variables défini par Ecore, qui va contenir les références aux deux modèles.

Si la méthode Java chargeant les deux modèles ne trouve pas le modèle FCSS, parce qu'il n'existe pas ou ne se trouve pas dans le même dossier que le fichier TVL, celle-ci appelle une autre méthode qui crée un fichier FCSS avec un contenu "par défaut". En effet, la génération ne peut avoir lieu si un modèle

FCSS n'est pas chargé, fusse-t-il inutile. Le générateur doit donc posséder une référence à un modèle FCSS même si celui-ci est fictif dans le cas où le concepteur n'en a pas défini. Le modèle FCSS en question (Code 9.3) ne contient que la ligne spécifiant le fichier TVL auquel il fait référence ainsi que la partie globale contenant uniquement le comportement par défaut adopté pour les groupes de *features oneof*, l'éditeur FCSS n'acceptant pas un modèle FCSS vide.

```
1 import "fichierTVL.tvl"
2 .{xorGroup:listbox;}
```

Code 9.3 – Modèle FCSS par défaut

Une fois le fichier de sortie défini et les deux modèles chargés, le squelette de la page HTML5 est défini. Il est composé des balises habituelles, la balise <!DOCTYPE html> permettant de définir le type de document, les balises <html></html> délimitant le document, les balises <head></head> délimitant l'en-tête et les balises <body></body> comprenant le contenu réel du document. Les balises <head></head> contiennent diverses informations telles que le titre de la page, des liens vers des feuilles de style mais surtout vers des scripts permettant la gestion dynamique de la page. Le Code 9.4 permet donc de faire le lien entre l'interface statique et le contrôleur et, pour la première ligne, de fournir un lien vers une feuille de style que le concepteur est libre de définir comme il l'entend.

```
1<link rel="stylesheet" href="style.css" />
2<script type="text/javascript" src="controller.js"></script>
```

Code 9.4 – Lien avec feuille de style et contrôleur

Ensuite, les balises <body></body> contiennent un titre qui correspond au label de la *feature* racine et les balises définissant le formulaire dans lequel seront générés tous les widgets correspondant aux éléments TVL définis dans le modèle TVL. Pour lancer la génération, qui s'effectue en cascade, en suivant la hiérarchie des *features*, il faut d'abord appeler le *template* de génération d'une *feature* sur la *feature* racine.

Étant donné que le code Acceleo est composé d'une séries de *templates* qui traduisent des règles de transformations, il est donc nécessaire de définir ces règles. À chaque élément TVL doit correspondre du code dans le langage cible. Ce code est composé de texte immutable et de variables qui vont prendre leurs valeurs pendant le parcours des modèles TVL et FCSS grâce au langage OCL.

Avant de parler de la gestion des éléments TVL et FCSS, nous allons d'abord voir l'architecture abstraite du générateur.

9.2 Architecture abstraite du générateur

La Figure 9.2 présente l'architecture abstraite du générateur. Elle illustre les liens qui existe entre les différents éléments TVL à gérer.

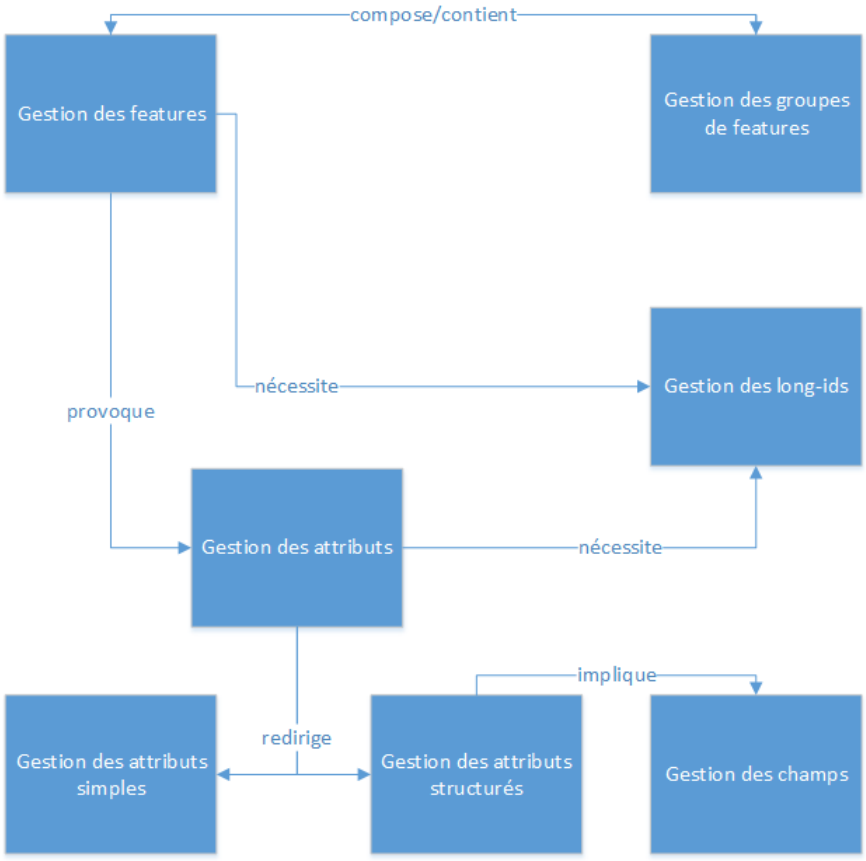

FIGURE 9.2 – Architecture abstraite du générateur

Comme nous le verrons par la suite, la génération s'effectue en cascade. Lorsque le générateur rencontre une *feature*, il doit générer son contenu qui est, en toute généralité, composé d'attributs et d'un groupe de *sous-features*. Ainsi, la gestion des *features* implique la gestion des groupes de *features* et inversément. La gestion d'une *feature* implique également la gestion de ses attributs. Les attributs sont de différents types, simples ou structurés. En fonction de cela, la gestion de ceux-ci va en partie différer. La gestion des attributs structurés implique évidemment la gestion des attributs qui les composent. De plus, les *features* et attributs sont identifiés par leur *long-id*. Lorsqu'ils sont générés, leur *long-id* doit également l'être.

Les sections suivantes sont consacrées à expliquer comment les différents éléments qui composent un modèle TVL sont gérés, avec et sans prise en compte du modèle FCSS. Ainsi, nous verrons comment les *features*, les groupes et les attributs sont gérés. Chaque section est divisée en deux parties. La première

explique les principes de la gestion des constructions en question, indépendamment du langage d'implémentation, et la seconde montre en partie comment ceux-ci sont implémentés en Acceleo. Nous aborderons également la question des *long-ids* ainsi que les modifications qu'il a fallu effectuer pour obtenir une application "standalone", capable de s'exécuter en dehors de l'environnement Eclipse dans lequel elle a été développée.

En fin de chapitre, nous verrons l'architecture concrète du générateur, liée au langage Acceleo.

9.3 Gestion des features

9.3.1 Principes

Dans la présentation du langage TVL, nous avons vu qu'il existait plusieurs "types" de *features* : la *feature* racine, les *features* hiérarchiques et les extensions de *features*. Ces dernières ne sont en réalité pas des *features* à part entière mais uniquement des extensions des corps principaux des *features*, elles doivent donc être intégrées à la génération du corps d'une *feature*.

Toutes les *features* d'un modèle TVL sont gérées de la même manière car il n'y a pas de différence fondamentale entre elles. En effet, la *feature* racine n'est en réalité pas différente d'une *feature* hierarchique, elle possède un corps principal, elle peut posséder des extensions, elle peut posséder un groupe de *sous-features*, tout comme n'importe quelle *feature* hiérarchique. Au niveau des différences, outre le fait que, comme son nom l'indique, la *feature* racine n'a pas de *feature* parente, elle a obligatoirement un corps principal, sinon le modèle serait vide, ce qui n'est pas le cas des *features* hiérarchiques. Ces dernières peuvent être vides.

Afin de générer le contenu d'une *feature*, il est nécessaire d'obtenir tous les éléments de son corps. Étant donné qu'une *feature* peut avoir plusieurs extensions, il faut non seulement récupérer les éléments du corps principal de la *feature* mais également les éléments se trouvant dans les extensions. Il faut donc commencer par récupérer l'ensemble des extensions qui référencent la *feature* en question. Ensuite, il faut récupérer les éléments du corps principal de la *feature*, les éléments des différentes extensions et les rassembler en un ensemble d'éléments qui forment le contenu de la *feature*.

Cet ensemble contient typiquement plusieurs types d'éléments : des contraintes, des attributs simples et/ou structurés et un groupe de *features*. Il faut donc boucler sur ceux-ci et les gérer différemment en fonction de leur type. Par contre, les contraintes doivent être ignorées puisqu'elles sont des éléments définissant le comportement et ne doivent pas être générées dans l'interface utilisateur par respect du principe de séparation des préoccupations.

La génération des éléments d'une *feature* n'est pas suffisante. Il faut également générer la *feature* en elle-même, c'est-à-dire le "widget" qui va la représenter. À ce niveau, il faut considérer deux cas : la *feature* est vide ou elle ne l'est pas. Une *feature* est considérée comme vide si son corps et ses éventuelles extensions ne contiennent rien ou, éventuellement, uniquement des contraintes.

Les deux cas présentent plusieurs similarités. En effet, dans les deux cas, il faut générer le "widget" la représentant, c'est-à-dire le label ou une image en fonction de ce qui est spécifié dans le modèle FCSS. Dans le cas où c'est une image, il faut que le nom du fichier corresponde au *long-id* de la *feature* représentée. Si rien n'est spécifié, le comportement par défaut consiste à considérer le *short-id* de la *feature* comme le label à générer. L'outil doit aussi générer le "widget" de sélection de la *feature*. Le "widget" de sélection généré dépend de la cardinalité du groupe dans lequel se trouve la *feature*. Tout ce qui concerne les groupes de *features* sera abordé plus en détail dans la Section 9.4.

Au niveau de sa fonction, il faut faire la différence entre les *features* optionnelles et les autres. Pour une *feature* optionnelle, le "widget" sert non seulement à changer la valeur de la *feature* (sélectionnée ou non) mais également à afficher/dissimuler son contenu. En effet, le contenu généré d'une *feature* optionnelle est caché par défaut. Lorsque l'utilisateur sélectionne la *feature*, son contenu est affiché. S'il la désélectionne, le contenu est de nouveau dissimulé.

Il est également nécessaire de vérifier l'attribut spécifique FCSS "help" afin de générer l'éventuel texte d'aide.

La différence entre le cas où la *feature* est vide et le cas où elle ne l'est pas se situe simplement au niveau du fait que le contenu d'une *feature* non vide est délimité, encadré afin que l'utilisateur voit l'ensemble des "widgets" qui constituent cette *feature*.

Avant de générer tout cela, il est nécessaire de vérifier si le modèle FCSS contient l'attribut spécifique "generate" à la *feature* en question. S'il est spécifié et que sa valeur est à "false", rien n'est généré, dans les autres cas, la génération a lieu.

Toute la génération s'effectue en cascade en commançant par la *feature* racine puis en parcourant la hiérarchie des *features* du FM en profondeur d'abord.

9.3.2 Implémentation en Acceleo

C'est le module "generateConfigurator.mtl" qui appelle la première fois le *template* de génération du contenu d'une *feature* pour la *feature* racine. Ce *template* se trouve, comme tous les autres visant à traiter les *features* et groupes dans le module "generateFeature.mtl". Ensuite, toute la génération s'effectue en cascade en suivant la hiérarchie des *features* du FM.

Nous allons voir le *template* "generateFieldSetFeature" (Code 9.5) afin d'expliquer comment se passe la génération d'une *feature* en Acceleo. Le *template* en question est celui qui génère un "fieldset" représentant une *feature* et délimitant son contenu. Ce *template* s'applique aux *features* non vides.

```
1 [template private generateFieldSetFeature(
2 aHierarchicalFeature:tVL::Hierarchical_Feature,
3 fcssModel:fCSS::FCSSModel) post(trim())]
4   [comment: check if the feature must be generated/]
5   [if(aHierarchicalFeature.checkGenerateFeature(fcssModel)='true')]
6
7   <fieldset
8   id="[aHierarchicalFeature.generateLongIdFeature()/]-fieldset"
```

```
 9  [comment: to not display elements inside an
10  optional feature by default/]
11  [aHierarchicalFeature.eContainer().eContainer().
12  eContainer(tVL::Common_Feature).
13  manageDisplayElementFeature(fcssModel)/]
14  >
15  <legend
16  [aHierarchicalFeature.manageFeatureHelp(fcssModel)/]
17  >
18  [comment: if hierarchical-feature is optionnal/]
19  [if (aHierarchicalFeature.optional)]
20    [aHierarchicalFeature.manageOptionalFeatureWidget(fcssModel)/]
21  [else]
22    [comment: if the feature is not optional, generates the
23    selection widget: checkbox for group someof,
24    radio for group oneof/]
25    [aHierarchicalFeature.generateSelectionWidgetFeature
26    (fcssModel)/]
27  [/if]
28
29  [aHierarchicalFeature.manageFeatureWidget(fcssModel)/]
30  </legend>
31    [comment: generate the content/]
32    [aHierarchicalFeature.generateFeature(fcssModel)/]
33  </fieldset>
34
35  [/if]
36 [/template]
```

Code 9.5 – Template generateFieldSetFeature

Le *template* prend deux paramètres, le premier indique qu'il s'applique à tout élément de type "Hierarchical_Feature" ce qui exclut la *feature* racine. En effet, le contenu de la *feature* racine correspond au document HTML5 entier et son label ou son image la représentant sera le titre de la page. Le second paramètre est un élément de type "FCSSModel" qui correspondra à la référence au modèle FCSS lors de la génération.

Avant toute chose, le *template* vérifie si la *feature* doit vraiment être générée (ligne 5). La *query* "checkGenerateFeature" retourne "false" si l'attribut spécifique "generate" du FCSS vaut "false" pour cette *feature*, "true" dans les autres cas.

Ensuite, le fieldset en lui-même est généré. L'attribut "id" de la balise HTML5 permettant d'identifier le widget, celui-ci contiendra la concaténation du *long-id* de la *feature* avec la chaîne de caractère "-fieldset" (ligne 8). Il ne peut être identifié uniquement par le *long-id* de la *feature* car le widget représentant l'éventuel groupe de *sous-features*, s'il existe, est identifié par celui-ci. Aux lignes 11 à 13, le *template* "manageDisplayElementFeature" est appelé. Il va générer le code permettant de cacher le "fieldset" si la *feature* parente est optionnelle.

À la ligne 16, le *template* fait appel au *template* "manageFeatureHelp" qui va générer le code permettant d'afficher un texte d'aide lorsque l'utilisateur passera sa souris sur le label/l'image de la *feature* qui se trouve dans la balise "legend".

Les lignes 19 à 27 sont consacrées à la génération du "widget" de sélection. Les *templates* appelés sont différents en fonction du caractère optionnel ou non de la *feature* pour la raison évoquée dans la sous-section précédente.

Enfin, à la ligne 32, le *template* permettant de générer le contenu de la *feature* est appelé. Celui-ci fait lui-même appel à la *query* "getBodyItems" permettant de récupérer les différents éléments que la *feature* possède. Pour rappel, celle-ci a besoin de connaître les différentes extensions de la *feature* pour récupérer tous ces éléments. Elle fait donc appel à la *query* "getAFeaturesExtensions" (Code 9.6) permettant de récupérer toutes les extensions d'une *feature*. Une fois ces extensions obtenues, une autre *query* se chargera de récupérer le contenu de chaque extension qui sera concaténé avec le contenu du corps principal de la *feature*.

```
1 [query public getAFeaturesExtensions(aFeature:tVL
2 ::Feature_Declaration):
3 Set(tVL::Feature_Extension) =
4 aFeature.getModel().model->selectByType(tVL::Feature_Extension)->
5 select(anExtension:tVL::Feature_Extension |
6 anExtension.getExtensionRef(anExtension.ref.tail)=aFeature)
7 /]
```

Code 9.6 – Query getAFeaturesExtensions

Cette *query* utilise la *query* "getModel" qui, à partir de n'importe quelle *feature*, permet de récupérer l'élément "Model" qui est la racine du modèle TVL. À partir de là, la query se charge de sélectionner les éléments de type "Feature_Extension" pour ensuite ne garder que les extensions qui référencent la *feature*. Elle retourne un ensemble d'éléments "Feature_Extension".

9.4 Gestion des groupes de features

9.4.1 Principes

Parmi les éléments faisant partie du contenu d'une *feature*, nous avons donc les groupes de *(sous-)features*. La première chose à faire lorsqu'un tel groupe est rencontré est de générer l'éventuel label du groupe, s'il est spécifié dans le modèle FCSS. Ensuite, il est nécessaire de vérifier les *features* du groupe. Plus précisémment, il faut vérifier si les *features* sont toutes vides, ce qui signifie que ces *sous-features* doivent être traitées comme un ensemble de valeurs étant donné qu'il n'y a rien à générer au delà, ce sont des feuilles de l'arbre du FM TVL. Pour rappel, une *feature* est considérée comme vide si elle ne possède pas d'attribut ni de groupe de *sous-features*. Une *feature* qui ne contient que des contraintes, que ce soit dans son corps principal ou dans des extensions, est également considérée comme vide.

Dans le cas où toutes les *features* du groupe sont vides, par défaut, la représentation graphique du groupe de *features* va dépendre de la cardinalité de celui-ci. Cependant, il faut également prendre en compte ce qui est spécifié dans le modèle FCSS. En effet, il faudra d'abord vérifier ce qui est spécifié pour ce groupe spécifique. Si rien n'est indiqué dans cette partie, il est alors nécessaire

de vérifier ce qui est spécifié dans la partie globale du modèle pour les groupes de même cardinalité que ce groupe ou plutôt de même type car un groupe de cardinalité [2..3] et un groupe de cardinalité [2..4] sont considérés comme des groupes "card". De même, si rien n'est spécifié non plus dans la partie globale, alors la représentation graphique par défaut est choisie. Ce comportement est valable pour tout ce qui est personnalisable dans la génération, que ce soit les "widgets" des attributs et groupes de *features*, les labels, etc.

Pour revenir sur les "types" de groupes, nous en considérons quatre :

1. *oneof* : comprend les groupes *oneof* explicites et les groupes de cardinalité [1..1].

2. *someof* : comprend les groupes *someof* explicites et les groupes de cardinalité [1..*].

3. *allof* : comprend les groupes *allof* explicites et les groupes de cardinalité [*..*] ou encore [*nbFeatures..nbFeatures*] où *nbFeatures* est le nombre de *features* du groupe.

4. *card* : comprend tous les autres groupes ([2..3] par exemple).

Les "mappings" par défaut des groupes de *features* vides sont les suivants : les groupes de type *oneof* sont représentés par des "listbox", les groupes de type *someof* sont représentés par des "checkbox" tout comme les groupes *card* et les groupes de type *allof* sont représentés par des "textbox" qui encadrent leur contenu si les *features* ont du contenu configurable. Comme ce n'est pas le cas pour les *features* vides, celles-ci ne sont pas générées sauf si elles sont optionnelles. Dans ce cas, par défaut, un "widget" de type "checkbox" est généré en plus du label de la *feature* optionnelle. En effet, dans le cas d'un groupe de ce type, toutes les *features* sont considérées comme obligatoires. De ce fait, l'utilisateur n'a en réalité pas de choix, il est donc inutile qu'elles apparaissent à l'écran et surtout faux de laisser l'utilisateur les choisir si elles n'ont pas de contenu configurable. Dans ce type de groupe, seules les *features* optionnelles peuvent être sélectionnées ou non par l'utilisateur d'où le choix de la représentation.

La Figure 9.3 récapitule les mappings par défaut des groupes de *features* vides. Pour le groupe *allof*, seule la deuxième *feature* est optionnelle donc générée tandis que les autres *features* ne le sont pas. Pour le groupe de type *card*, sa cardinalité exacte est [1..2] d'où le message d'aide destiné à l'utilisateur au dessus des *checkbox*.

Type de groupe	Représentation par défaut	Balise HTML5	Illustration
oneof	Listbox	<select> <option>...</option>...</select>	FeatureA
someof	Checkbox	<input type=checkbox .../>	FeatureA FeatureB FeatureC
allof	Checkbox (pour feature optionnelle)	[if feature.opt] <input type=checkbox .../>	OptFeatureB
card	Checkbox	<input type=checkbox .../>	Sélectionnez entre 1 et 2 features FeatureA FeatureB FeatureC

FIGURE 9.3 – Mappings par défaut des groupes de features vides

Avec le modèle FCSS, la sélection de la bonne représentation est donc plus complexe puisqu'elle nécessite de vérifier la partie spécifique à la *feature* contenant le groupe de *sous-features*. Il faut noter que l'éditeur FCSS doit vérifier si l'attribut "widget" a une valeur qui peut s'appliquer au groupe en question. Ainsi, un groupe de type *oneof* peut être représenté par un "radiogroup", une "listbox" ou un groupe de "checkbox". Pour les deux premières représentations, le navigateur Web vérifie d'office que la contrainte de cardinalité est respectée étant donné le type de "widgets" tandis que pour la dernière représentation, l'API vérifiera de toute façon que la contrainte est respectée. Pour un groupe *someof* ou *card*, seules les "checkbox" et "listbox" (à choix multiple) sont des représentations compatibles. Enfin, un groupe *allof* ne peut être représenté que par des "textbox" étant donné que les *features* sont toutes obligatoires. Comme nous l'avons déjà dit, dans le cas de groupes de *features* vides, seules les *features* explicitement mentionnées comme optionnelles sont alors générées avec le "widget" mentionné pour celles-ci, à savoir "checkbox", "listbox" ou "radiogroup". Si rien n'est spécifié, il faut vérifier l'attribut de la partie globale qui s'applique au type de groupe : *xorGroup* si c'est un groupe *oneof*, *orGroup* si c'est un groupe *someof*, *andGroup* si c'est un groupe *allof* et *cardGroup* si c'est un groupe *card*. Enfin, les représentations par défaut peuvent être appliquées si l'attribut FCSS global en question n'est pas spécifié. Dans le cas d'un groupe *card*, il apparait utile de générer un texte d'aide informant l'utilisateur du nombre de *features* qu'il doit/peut sélectionner comme le montre la Figure 9.3.

Pour illustrer tout ce qui a été dit précédemment, nous allons prendre la portion de Code 9.7 TVL.

```
1 root FeatureRoot group allof{
2   FeatureAvecGroupeCard group [2..3]{
3     FeatureA,
4     FeatureB,
5     FeatureC,
6     FeatureD
7   }
8 }
```

Code 9.7 – Exemple groupe card

Ensuite, nous associons le Code 9.8 FCSS en spécifiant que le groupe de la *feature* "FeatureAvecGroupeCard" doit être représenté par une "listbox".

```
1 import "exempleFeatureCard.tvl"
2
3 .{
4   cardGroup: checkbox;
5 }
6
7 FeatureAvecGroupeCard{
8   group{
9     label: "Groupe Card";
10    widget: listbox;
11  }
12 }
```

Code 9.8 – FCSS Exemple groupe card

Le résultat est présenté à la Figure 9.4. Comme nous pouvons le voir, le groupe est bien représenté par une "listbox" comme indiqué à la ligne 10 et non par un groupe de "checkbox" spécifié par l'attribut "cardGroup" à la ligne 4. Il y a également un texte qui informe l'utilisateur sur le nombre de *features* qu'il peut sélectionner. De plus, le label du groupe spécifié à la ligne 9 a bien été généré.

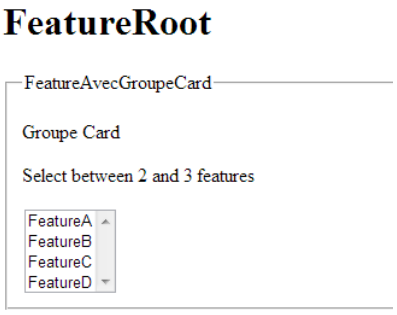

FIGURE 9.4 – Génération de groupe card avec attribut FCSS spécifique

Si nous reprenons le même code FCSS mais en supprimant la ligne 10, le comportement défini par "cardGroup" à la ligne 4 doit s'appliquer. Le nouveau résultat est présenté à la Figure 9.5. Le groupe est maintenant représenté par un groupe de "checkbox", toujours avec le texte d'information.

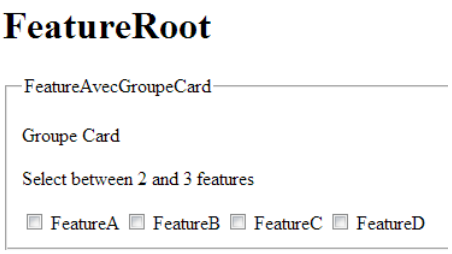

FIGURE 9.5 – Génération de groupe card avec attribut FCSS global

Enfin, si nous retirons également l'attribut global "cardGroup" du modèle FCSS, c'est le comportement par défaut qui va être choisi. Accidentellement, le "widget" par défaut est également le groupe de checkbox et donc le résultat est identique à celui de la figure 9.5.

Lorsque toutes les *features* ne sont pas vides, ce qui inclut la situation ou une partie seulement l'est, les *features* sont générées les unes après les autres avec leur éventuel contenu comme expliqué dans la Section 9.3. Comme vu dans cette section, les *features* sont gérées différemment en fonction de leur contenu (vide ou non) et de leur caractère optionnel ou non. De plus, le "widget" de sélection varie en fonction du type de groupe dont les *features* font partie. Ainsi, le "widget" de sélection d'une *feature* générée est une "checkbox" si la *feature* fait partie d'un groupe *someof* ou *card* et une "radio" si la *feature* fait partie d'un groupe *oneof*.

Si le groupe est un groupe *allof* alors les *features* vides ne doivent pas être générées et les *features* non vides sont encadrées ("textbox") mais ne doivent pas être précédées d'un "widget" de sélection, à part pour les *features* optionnelles, dont le "widget" par défaut est la "checkbox". En effet, un groupe *allof* signifie que toutes les *features* sont obligatoires, à part celles qui sont explicitement déclarées comme optionnelles. L'utilisateur n'a donc rien à configurer si la *feature* obligatoire est vide et il ne doit pas non plus la sélectionner puisqu'elle l'est d'office. Il est donc inutile qu'elle apparaisse dans l'interface graphique du configurateur. Pour rappel, le "widget" de sélection d'une *feature* optionnelle sert également à afficher/cacher le contenu de cette *feature*, cela afin de bien les distinguer des autres *features*.

La Figure 9.6 illustre les "mappings" par défaut pour les groupes de *features* non vides. En raison du manque d'espace, les images ne montrent pas l'encadrement auteur des *features* non vides. Néanmoins, il est bien généré comme nous le verrons dans la suite de cette section.

Type de groupe	Représentation par défaut	Balise HTML5	Illustration
oneof	Radiogroup	<fieldset> <input type=radio .../> </fieldset>	○ FeatureA ○ FeatureB ○ FeatureC
someof	Checkbox	<fieldset> <input type=checkbox .../> </fieldset>	☐ FeatureA ☐ FeatureB ☐ FeatureC
allof	Textbox (+checkbox pour feature optionnelle)	<fieldset> [if feature.opt] <input type=checkbox .../> <fieldset>	FeatureA ☐ OptFeatureB FeatureC
card	Checkbox	<fieldset> <input type=checkbox .../> </fieldset>	Sélectionnez entre 1 et 2 features ☐ FeatureA ☐ FeatureB ☐ FeatureC

FIGURE 9.6 – Mappings par défaut des groupes de features non vides

Pour finir cette section, nous allons voir deux exemples de modèles TVL qui illustrent les différences de traitement des types de groupes et des *features* (vides/non vides, optionnelles/obligatoires). Le premier exemple (Code 9.9) montre une *feature* qui possède un groupe *oneof* dans lequel se trouve quatre *features* : la première a du contenu et n'est pas optionnelle, la seconde est vide et non optionnelle, la troisième n'est pas vide mais est optionnelle et la dernière est vide et optionnelle.

```
1 root FeatureRoot group allof{
2   MixFeatures group oneof{
3     FeatureA{int a;},
4     FeatureB,
5     opt FeatureC{bool b;},
6     opt FeatureD
7   }
8 }
```

Code 9.9 – Exemple groupe features hétérogènes

Comme la Figure 9.7 le montre, toutes les *features* sont ici générées mais différemment selon leurs caractéristiques. Les "widgets" de sélection sont des radios, les *features* vides sont représentées par un simple label et la *feature* optionnelle a son contenu caché au départ. Si le groupe avait été de type *someof* ou card, les radios auraient simplement été remplacées par des "checkbox".

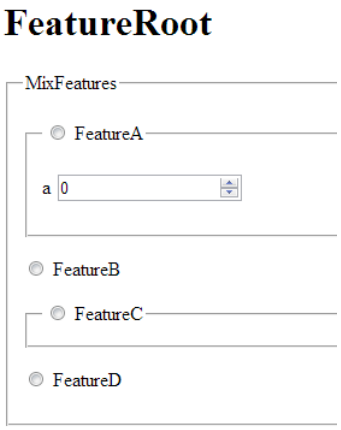

FIGURE 9.7 – Génération d'un groupe oneof avec divers types de features

Le deuxième exemple est identique au premier à ceci près que le groupe est de cardinalité *allof*. Le résultat présenté à la Figure 9.8 est néanmoins assez différent. Premièrement, la *feature FeatureB* n'est pas générée car elle est vide et n'est pas optionnelle. Deuxièmement, seules les *features* optionnelles ont un *widget* de sélection et il s'agit de "checkbox" étant donné que les *features* font partie d'un groupe *allof*.

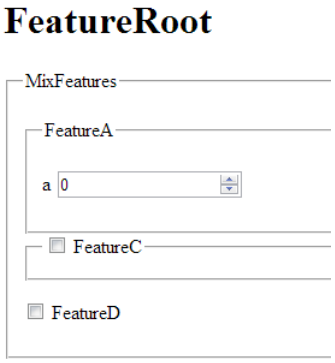

FIGURE 9.8 – Génération d'un groupe allof avec divers types de features

Grâce au FCSS, nous pouvons également spécifier la *feature* par défaut d'un groupe. Cette *feature* est donc cochée/sélectionnée lorsqu'elle est rencontrée pendant la génération.

9.4.2 Implémentation en Acceleo

Nous illustrons en partie le raisonnement décrit ci-dessus avec le *template* "checkBeforeGenFeatureGroupWithVoidSubFeatures" (Code 9.10) qui vérifie la représentation graphique à donner au groupe avant d'appeler le *template* ad-hoc.

```
1 [template private checkBeforeGenFeatureGroupWithVoidSubFeatures
2 (aFeatureGroup:tVL::Feature_Group, fcssModel:fCSS::FCSSModel)]
3   [let graphicalRepresentation:String = aFeatureGroup.
4   getContainingFeatureDeclaration().
5   getFeatureGroupGraphicalRepresentation(aFeatureGroup, fcssModel)]
6     [if(graphicalRepresentation = 'textbox')]
7       [aFeatureGroup.generateGroupVoidOptionalFeaturesAsCheckboxes
8       (fcssModel)/]
9     [elseif(graphicalRepresentation = 'checkbox')]
10      [aFeatureGroup.generateGroupVoidFeaturesAsCheckboxes
11      (fcssModel)/]
12    [elseif(graphicalRepresentation = 'radiogroup')]
13      [aFeatureGroup.generateGroupVoidFeaturesAsRadio(fcssModel)/]
14    [elseif(graphicalRepresentation = 'listbox')]
15      [comment: in the case of the listbox representation,
16      it is a multiple selection listbox if the feature group is a
17      someof, allof or card group/]
18      [if(aFeatureGroup.getFeatureGroupType() = 'oneof')]
19        [aFeatureGroup.generateGroupVoidFeaturesAsListbox
20        (fcssModel, false)/]
21      [else]
22        [aFeatureGroup.generateGroupVoidFeaturesAsListbox
23        (fcssModel, true)/]
24      [/if]
25    [/if]
26  [/let]
27 [/template]
```

Code 9.10 – Template checkBeforeGenFeatureGroupWithVoidSubFeatures

Aux lignes 3 à 5, le *template* fait appel à la *query* "getFeatureGroupGraphicalRepresentation" (Code 9.11) pour récupérer la représentation graphique à appliquer au groupe. Il place cette valeur dans une variable finale grâce à la balise "let". Ensuite, en fonction de la valeur, il appelle le *template* qui génère cette représentation.

Aux lignes 18 à 24, le *template* vérifie le type du groupe grâce à la *query* "getFeatureGroupType". En fonction du type, il appelle le même *template* "generateGroupVoidFeaturesAsListbox" mais avec deux valeurs différentes du deuxième argument. Cela permet au *template* de savoir s'il doit générer une "listbox" à choix unique (groupe *oneof*) ou multiple (autres groupes).

```
1 [query public getFeatureGroupGraphicalRepresentation
2 (aFeature:tVL::Feature_Declaration,
3 aFeatureGroup:tVL::Feature_Group,
4 fcssModel:fCSS::FCSSModel):
5 String =
6 if (not aFeature.getGroupWidget(fcssModel).oclIsUndefined())
7 then aFeature.getGroupWidget(fcssModel)
8 else
9   if (not fcssModel.getGlobalPartFCSSModel().
```

```
10   oclIsUndefined())
11   then
12     if (aFeatureGroup.getFeatureGroupType().
13     equalsIgnoreCase('oneof')
14     and not fcssModel.getGlobalPartFCSSModel().getXorGroup().
15     oclIsUndefined())
16     then fcssModel.getGlobalPartFCSSModel().getXorGroup()
17     else
18       if (aFeatureGroup.getFeatureGroupType().
19       equalsIgnoreCase('someof')
20       and not fcssModel.getGlobalPartFCSSModel().getOrGroup().
21       oclIsUndefined())
22       then fcssModel.getGlobalPartFCSSModel().getOrGroup()
23       else
24         if (aFeatureGroup.getFeatureGroupType().
25         equalsIgnoreCase('allof')
26         and not fcssModel.getGlobalPartFCSSModel().getAndGroup().
27         oclIsUndefined())
28         then fcssModel.getGlobalPartFCSSModel().getAndGroup()
29         else
30           if (aFeatureGroup.getFeatureGroupType().
31           equalsIgnoreCase('card')
32           and not fcssModel.getGlobalPartFCSSModel().getCardGroup().
33           oclIsUndefined())
34           then fcssModel.getGlobalPartFCSSModel().getCardGroup()
35           else aFeatureGroup.
36           getDefaultGraphicalRepresentationFeatureGroup()
37           endif
38         endif
39       endif
40     endif
41   else aFeatureGroup.
42   getDefaultGraphicalRepresentationFeatureGroup()
43   endif
44 endif
45 /]
```

Code 9.11 – Query getFeatureGroupGraphicalRepresentation

La *query* "getFeatureGroupGraphicalRepresentation" permet donc de récupérer la représentation graphique à appliquer au groupe en cours de traitement. Elle fait d'abord appel à la *query* "getGroupWidget" (lignes 6 et 7) qui récupère la valeur de l'attribut FCSS "widget" du groupe en question, si elle existe. Si celui-ci n'est pas défini, elle vérifie la partie globale (lignes 9 et 10). Pour savoir quel attribut global s'applique au groupe, elle doit appeler la *query* "getFeatureGroupType" (lignes 12, 18, 24 et 30). Si l'attribut en question n'est pas spécifié, elle appelle la *query* "getDefaultGraphicalRepresentationFeatureGroup" qui, en fonction du type de groupe, retourne la représentation graphique à appliquer par défaut (lignes 35-36 et 41-42).

Cette *query* retourne une chaîne de caractères correspondant à la représentation graphique que doit avoir le groupe en question.

9.5 Gestion des attributs

9.5.1 Principes

Lorsque l'élément à générer est un attribut, il faut commencer par vérifier s'il s'agit d'un attribut simple ou structuré, quelques différences les séparant. Nous commencerons par voir comment sont gérés les quatre types d'attributs simples que sont les entiers, les réels, les booléens et les énumérations. Nous verrons ensuite la gestion des attributs structurés et de leurs champs.

En ce qui concerne les attributs simples, une fois le label de l'attribut à traiter généré, il faut vérifier son type. Une fois qu'il est connu, il faut contrôler le modèle FCSS avant d'appliquer la règle de transformation ad hoc et ainsi générer le bon "widget". Ainsi, comme pour les groupes de *features*, il est d'abord nécessaire d'examiner ce qui est écrit dans la partie spécifique à l'attribut à générer. Si rien n'est spécifié, il faut vérifier ce que contient la partie globale pour le type d'attribut en question. Si rien n'est spécifié là non plus, la règle de transformation par défaut est appliquée.

Par défaut, les entiers sont représentés par ce que nous appelons des "textbox" en toute généralité et qui prennent la forme d'"inputs" de type "number" en HTML5, les réels sont représentés de la même manière mais avec une petite spécificité : l'attribut "step" qui définit le saut lorsque l'utilisateur clique sur une des deux flèches du "widget" vaut 0.1 au lieu de 1 par défaut pour les entiers. Le saut est différent pour que l'utilisateur puisse lui-aussi faire la différence entre un attribut entier et un réel lorsqu'il manipule un "widget" représentant un attribut d'un de ces deux types. Ces deux types d'attributs devant être représentés graphiquement de manière très similaire, ils sont tous les deux gérés de la même façon afin d'éviter de la redondance. Les booléens sont représentés par des "checkbox" et les énumérations sont représentées par des groupes de "radios" si le nombre de valeurs possibles est inférieur ou égal à trois ou par des "listbox" dans le cas contraire. Le choix du chiffre '3' comme valeur permettant de choisir l'une ou l'autre représentation par défaut est totalement arbitraire, il pourrait facilement être modifié si nécessaire.

La Figure 9.9 illustre les règles de transformations appliquées par défaut, c'est-à-dire, si aucun élément du modèle FCSS référençant le modèle TVL en question ne dit ce qu'il faut générer pour l'élément TVL en cours de traitement.

Attribut TVL	Représentation par défaut	Balise HTML5	Illustration
Entier (int)	Textbox	<input type=number .../>	unEntier 12
Réel (real)	Textbox	<input type=number step=0.1 .../>	unReel 1.6
Booléen (bool)	Checkbox	<input type=checkbox .../>	unBooleen
Enumération (enum)	Radiogroup ou Listbox	[if size <= 3] <input type=radio .../> [if size > 3] <select> <option>...</option>...</select>	uneEnum val1 val2 val3 autreEnum val1

FIGURE 9.9 – Mappings par défaut des attributs

Le modèle FCSS permet de choisir d'autres représentations graphiques pour les attributs booléens et les énumérations. Les premiers peuvent également être représentés par des paires de "radio" ou des "listbox" avec les valeurs "yes" et "no" tandis que les seconds peuvent être représentés par des groupes de "radios" ou des "listbox" sans avoir à tenir compte de la taille de leur domaine de valeurs. Par contre, pour les nombres entiers et réels, il ne permet pas encore de définir d'autres représentations graphiques tels que les "sliders" par exemple qui seraient des "widgets" appropriés pour ces types d'attributs. Néanmoins, la règle de transformation permettant de générer de tels "widgets" a été définie et doit simplement être intégrée correctement à la génération. L'éditeur FCSS vérifie donc que la valeur de l'attribut "widget" appliqué à l'attribut a une valeur compatible avec le type d'attribut en question. Ainsi, il n'est par exemple pas question de représenter un attribut entier par une "checkbox".

Au delà de la simple transformation des attributs en fonction de leur type, il est également indispensable de gérer deux types de contraintes que le concepteur peut leur appliquer. En effet, comme précisé dans le Chapitre 5 abordant les langages TVL et FCSS, il est possible de donner une valeur initiale ou de limiter le domaine de valeurs d'un attribut. Pour rappel, il n'est possible de donner la valeur par défaut et de limiter le domaine que lorsqu'un type d'attribut a été défini en dehors de toute *feature* et que l'attribut en question est un attribut de ce type. Il n'est pas question ici de vérifier que la valeur par défaut se trouve bien dans le domaine de valeurs défini, dans le cas où le concepteur définirait les deux, ou encore que les valeurs possibles soient conformes au type de l'attribut concerné, cela est contrôlé par l'éditeur du langage TVL, mais bien que l'interface graphique produite vérifie ces contraintes.

Un attribut entier peut se voir attribuer la valeur par défaut '4' et un domaine de valeur [3..9], la gestion des réels est similaire. Le domaine de ces attributs peut également être limité par énumération des valeurs possibles. Cela n'est, par contre, pas géré par le générateur actuel. Cela impliquerait l'ajout de nouvelles règles de transformations car les "widgets" "textbox" ne conviendraient plus. Nous y reviendrons dans la Section 12.1.

En ce qui concerne les énumérations, elles peuvent prendre pour valeur une des valeurs définies lors de la déclaration du type de l'attribut et le domaine peut être réduit en citant les différentes valeurs, parmi la liste originale, que l'attribut

peut effectivement prendre. La gestion de la limitation de domaine se révèle plus complexe que pour les entiers et réels, étant donné que la représentation graphique par défaut des énumérations diffère en fonction du nombre de valeurs possibles. En effet, pour savoir quelle représentation il faut utiliser, il faut vérifier la taille de la liste de valeurs. Hors dans le cas où celle-ci est limitée, il faut vérifier la taille du sous-ensemble de valeurs possibles et non de l'ensemble original.

Les booléens, quant à eux, sont un cas un peu spécifique puisqu'ils ne peuvent prendre, de toute façon, que deux valeurs. Dans ce cas, seule l'attribution d'une valeur par défaut est possible.

Il est à noter qu'étant donné qu'il existe trois types de constantes (entières, réelles et booléennes), il est possible de spécifier les valeurs par défaut et celles limitant le domaine des attributs entiers, réels et booléens avec des constantes. Cette possibilité doit donc également être prise en compte.

Un attribut structuré est géré de façon similaire à une *feature*. En effet, la génération d'un tel attribut implique la génération de son contenu, l'ensemble de ses champs, tout comme la génération d'une *feature* implique la génération des éléments de son corps. De même, l'attribut structuré fournit un cadre à ces champs afin de délimiter son contenu.

Un attribut structuré est toujours de type prédéfini. Lorsque nous écrivons un modèle TVL, nous pouvons déclarer un type d'attribut structuré en dehors de la hiérarchie des *features*. Ensuite, nous pouvons déclarer, dans cette hiérarchie, des attributs structurés de ce type.

Le FM TVL du Code 9.12 va permettre d'illustrer ce qui différencie la génération des attributs structurés de celle des attributs simples. Il présente une *feature* contenant une structure de type "maStructure". Ce type de structure contient six champs dont deux sont de type prédéfini (lignes 4 à 11). Dans la définition de l'attribut structuré "maStruct", différents champs de la structure sont limités ou possèdent une valeur par défaut (lignes 15 à 19).

```
1 enum monEnum in {valeur1, valeur2, valeur3, valeur4};
2 int monEntier in [2..5];
3
4 struct maStructure{
5   monEntier a;
6   int i in [2..3];
7   monEnum e;
8   enum e2 in {val1,val2};
9   real r;
10  bool b;
11 }
12
13 root FeatureRoot group allof{
14   Feature{
15     maStructure maStruct{
16       a is 3; a in [3..4];
17       e1 is valeur3; e1 in {valeur1,valeur3};
18       r is 4.5;
19     }
20   }
```

```
21 }
```

Code 9.12 – Exemple d'attribut structuré

Il est tout d'abord nécessaire de générer le label de l'attribut structuré et le cadre qui contiendra les "widgets" repésentant les différents champs tout en vérifiant les attributs FCSS qui peuvent être associés à l'attribut, tout comme pour les *features*. Il est à noter que l'attribut "widget" spécifique à un attribut structuré ne possède pas de valeur possible pouvant s'appliquer en dehors de "textbox" qui consiste à encadrer le contenu de l'attribut structuré mais qui est de toute façon le comportement appliqué par défaut. Cet attribut FCSS est donc ignoré lors de la génération. Il devra être pris en compte une fois que d'autres représentations graphiques pourront être définies. Ensuite, il faut boucler sur les différents champs du type de structure de l'attribut (lignes 5 à 10 de l'exemple).

En effet, il ne faut pas boucler sur les sous-attributs qui référencent ces champs (lignes 16 à 18) étant donné qu'ils ne sont justement pas ces champs. En effet, un champ particulier peut être référencé par 0 (les champs "b", "e2" et "i"), 1 (le champs "r") ou 2 (les champs "a" et "e1") pour autant que la définition d'un attribut du type structuré contienne au moins un sous-attribut référençant un champ. Cet attribut structuré doit donc avoir au moins un sous-attribut référençant un champ, car la syntaxe de TVL l'oblige, mais il se peut très bien que tous les autres champs ne soient référencés par aucun sous-attribut. Le nombre de sous-attributs n'est donc pas le même que le nombre de champs, sauf coïncidence. Ce que nous appelons sous-attributs ici, ce sont en fait les contraintes de valeur par défaut ou de limitation de domaine de valeurs données à la déclaration d'une structure d'un tel type.

Ensuite, de la même manière que pour les attributs simples, il faut gérer les différents champs en fonction de leur type. Ainsi, les champs entiers et réels sont gérés de la même façon étant donné que la représentation graphique est la même à une différence près, le saut de 0.1 pour les réels au lieu de 1 pour les entiers.

La gestion des différents types de champs étant similaire à celle des attributs simples de mêmes types, nous nous concentrerons ici sur leurs différences. Une première différence est qu'il n'est pas possible de définir un comportement spécifique pour un champ dans le modèle FCSS, il n'y a donc pas lieu de le vérifier. Par contre, il faut contrôler ce qui est spécifié dans la partie globale puisque les types des champs sont les mêmes que les types d'attributs, les attributs globaux peuvent s'appliquer.

Néanmoins, la différence majeure provient du fait qu'il est possible d'avoir plusieurs sous-attributs référençant le même champ comme il est possible de n'en avoir aucun. Pour chaque champ, il est nécessaire de récupérer l'ensemble des sous-attributs le référençant, ensemble qui peut être vide.

La définition des valeurs par défaut et des domaines limités peuvent se faire de plusieurs façons, via un sous-attribut ou dans la définition du type de l'attribut structuré. L'exemple de Code 9.12 montre ces différentes possibilités. Le champ "a" à la ligne 5 est de type "monEntier" qui définit un type d'entier dont le domaine de valeurs est compris entre 2 et 5. Dans la définition de l'attri-

but "maStruct", deux sous-attributs référencent ce champ. Un premier donne la valeur par défaut de "a" et le second restreint encore plus le domaine de valeurs.

Ainsi, lors de l'obtention des valeurs minimales et maximales d'un attribut entier ou réel, il faut d'abord vérifier si un sous-attribut les définit. Si c'est le cas, ces valeurs sont récupérées, sinon, il faut rechercher ce qui est indiqué dans le type prédéfini. Si l'attribut n'est pas de type prédéfini ou si celui-ci ne spécifie pas ses valeurs minimale et maximale, ces valeurs n'existent pas et donc le domaine n'est pas limité. En ce qui concerne le domaine d'une énumération, encore une fois, il peut être défini de plusieurs façons. Il peut être défini par le type prédéfini si l'énumération est de type prédéfini, dans la définition du type structuré sinon et encore dans les sous-attributs. Ainsi, l'énumération "e1" a un domaine de valeurs d'abord défini dans son type (ligne 1) puis restreint via un sous-attribut (ligne 17). Par contre, l'énumération "e2" a son domaine de valeurs défini dans le type structuré (ligne 8).

La valeur par défaut, quant à elle, ne peut être donnée que par un sous-attribut comme c'est le cas à la ligne 16 pour l'entier "a" et à la ligne 17 pour l'énumération "e1".

Les différents cas présentés doivent être pris en compte.

La Figure 9.10 montre le résultat obtenu à la génération à partir du Code 9.12.

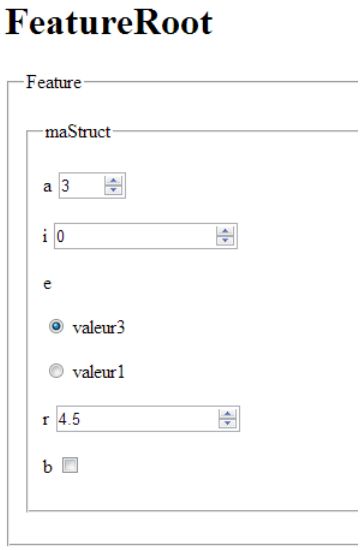

FIGURE 9.10 – Génération d'un attribut structuré

9.5.2 Implémentation en Acceleo

Lorsque l'élément à générer est un attribut, un *template*, placé dans le module "generateAttribute.mtl", va d'abord vérifier si l'attribut est de base ou

structuré avant de rediriger la génération vers le bon *template*. Les *templates* de gestion des attributs simples sont rassemblés dans le module "generateBaseAttribute.mtl tandis que ceux gérant les attributs structurés se trouvent dans le module "generateStructAttribute.mtl".

Après vérification du type et du modèle FCSS, un *template* de génération du "widget" de représentation est appelé. Le *template* "generateNumberTextBox" (Code 9.13) permet de générer un "input" de type "number" pour les attributs entiers et réels.

```
1 [template private generateNumberTextBox(
2 numberAttribute:tVL::Base_Attribute,
3 fcssModel:fCSS::FCSSModel) post(trim())]
4   <input type="number"
5   min="[if(not numberAttribute.getMinDomainAttribute().
6   oclIsUndefined())][numberAttribute.getMinDomainAttribute()/][/if]"
7   max="[if(not numberAttribute.getMaxDomainAttribute().
8   oclIsUndefined())][numberAttribute.getMaxDomainAttribute()/][/if]"
9   value="[numberAttribute.generateDefaultValueNumber(fcssModel)/]"
10  [if(numberAttribute.getAttributeType() = 'real')]step="0.1"[/if]
11  name="[numberAttribute.generateLongIdAttribute()/]"
12  id="[numberAttribute.generateLongIdAttribute()/]"
13
14  [numberAttribute.manageAttributeHelp(fcssModel)/]
15  onblur="manageNumber(this)"
16  />
17 [/template]
```

Code 9.13 – Template generateNumberTextBox

En HTML5, les attributs "min" et "max" des balises "input" permettent de gérer la limitation du domaine de valeurs sans que le développeur ait besoin de recourir à une fonction Javascript comme c'était le cas dans les versions précédentes de HTML. Ceux-ci vont donc contenir les valeurs minimale et maximale du domaine du nombre, si elles existent. Pour cela, le *template* fait appel aux *queries* "getMinDomainAttribute" (lignes 5 et 6) et "getMaxDomainAttribute" (lignes 7 et 8) respectivement. Les attributs de la balise seront vides si ces valeurs n'existent pas.

Pour la valeur par défaut, le *template* fait appel à un autre *template*, "generateDefaultValueNumber" (ligne 9), qui génère la valeur par défaut du nombre, en vérifiant si elle est donnée via une constante ou pas. Si elle n'est pas donnée, il vérifie si la valeur minimale est spécifiée et la prend comme valeur par défaut dans ce cas. Si elle n'est pas spécifiée non plus, la valeur par défaut est '0'.

Les attributs "name" et "id" de la balise se voient attribués le *long-id* de l'attribut comme valeur (lignes 11 et 12), afin que le contrôleur puisse identifier quel élément le "widget" représente.

À la ligne 14, le *template* fait appel au *template* "manageAttributeHelp" qui génère le texte d'aide dont la valeur est placée dans l'attribut de balise "title", attribut HTML5 prévu pour accueillir ce genre de texte.

Enfin, à la ligne 15, le *template* génère le code permettant de faire appel à une fonction Javascript définie dans le contrôleur qui gère ce type de "widget".

L'attribut d'événément "onblur" est activé au moment où le "widget" perd le "focus".

La *query* "getDefaultValueAttribute" (Code 9.14) permet de récupérer la valeur par défaut d'un attribut autre qu'une énumération. En effet, la navigation dans le modèle TVL est différente pour ce type d'attribut et il n'est pas possible d'utiliser une constante comme valeur par défaut d'une énumération.

```
1 [query public getDefaultValueAttribute
2 (anAttribute:tVL::Base_Attribute):
3 String =
4 if (not anAttribute.attr_body.attr_value.ref.oclIsUndefined())
5 then anAttribute.attr_body.attr_value.ref.head.
6 oclAsType(tVL::Constant).value
7 else anAttribute.attr_body.attr_value.value
8 endif
9 /]
```

Code 9.14 – Query getDefaultValueAttribute

Les lignes 4 à 6 servent à vérifier si la valeur par défaut est une constante et à la récupérer si c'est le cas. La ligne 7 retourne la valeur par défaut si elle a été spécifiée par une valeur directe et "undefined" si celle-ci n'existe pas.

Cette *query* retourne donc soit une chaîne de caractère qui correspond à la valeur par défaut soit "undefined".

Les *templates* générant les "widgets" des champs des attributs structurés étant fort semblables, nous ne décrirons pas une nouvelle fois un tel *template*. Par contre, nous pouvons voir que la *query* "getSubAttributesReferencingRecordField" (Code 9.15) permet de récupérer les sous-attributs référençant un même champ.

```
1 [query public getSubAttributesReferencingRecordField
2 (aRecordField:tVL::Record_Field,
3 aStructureAttribute:tVL::Structure_Attribute):
4 Set(tVL::Sub_Attribute) = aStructureAttribute.sub_attributes->
5 select(aSubAttribute:tVL::Sub_Attribute |
6 aSubAttribute.sub_id = aRecordField)
7 /]
```

Code 9.15 – Query getSubAttributesReferencingRecordField

Cette *query* a besoin de deux arguments pour fonctionner (lignes 2 et 3). Elle s'applique au champ (*Record_Field*) en question et doit connaître l'attribut structuré (*Structure_Attribute*) puisque c'est lui qui contient les éventuels sous-attributs référençant le champ. Elle récupère l'ensemble des sous-attributs (ligne 4) avant de le trier pour ne garder que ceux qui référencent le champ en question (lignes 5 et 6).

Elle retourne donc un ensemble de sous-attributs (*Sub_Attribute*).

Une fois que le *template* de génération du "widget" correspondant au champ connaît ces sous-attributs. Il n'a plus qu'à récupérer les éventuelles valeurs minimale, maximale et initiale grâce à d'autres *queries* qui vérifient quel sous-attribut donne quelle information. Elles vérifient également les autres endroits où ces valeurs peuvent être spécifiées comme expliqué précédemment.

9.6 Gestion des long-ids

9.6.1 Principes

Comme nous l'avons déjà vu dans le chapitre abordant les langages TVL et FCSS, les *long-ids* des éléments TVL les identifient. Il faut donc pouvoir les générer afin que le contrôleur puisse reconnaitre de quel élément il s'agit lorsqu'il gère un événement sur un "widget" spécifique. Cela est essentiel pour pouvoir avoir un contrôleur et une partie serveur unique pour tous les configurateurs générés.

Pour illustrer la génération des *long-ids*, nous prendrons un FM TVL (Code 9.16) pour exemple.

```
1 struct structure {
2   int astruct;
3   real rstruct;
4   bool bstruct;
5 }
6
7 root A{
8   group allof{
9     B,
10    C,
11    D{
12      int i;
13      structure s{astruct is 1; bstruct is false;}
14    },
15    E group someof{
16      F,
17      G
18    }
19  }
20 }
21
22 D{
23  real r;
24  group oneof{
25    E,
26    F group allof{
27      G,
28      H
29    }
30  }
31 }
```

Code 9.16 – TVL illustration des long-ids

Comme nous pouvons le voir, certaines *features* ont le même nom - "E" et "F" reviennent deux fois chacun, aux lignes 15 et 25 pour "E" et 16 et 26 pour "F" - ce qui ne constitue pas une erreur puisque le nom d'une *feature* ou d'un attribut ne l'identifie pas. Via cet exemple, nous allons voir quelles sont les différentes situations qui peuvent survenir lors de la génération des *long-ids*.

Commençons par le plus simple : la génération du *long-id* d'une *feature* se trouvant dans la hiérarchie principale. Si nous prenons l'exemple de la *feature*

"G" à la ligne 14, son *long-id* est "A.E.G", la génération se fait de manière récursive en remontant la hiérarchie de *feature*.

La génération du *long-id* d'un attribut qui se trouve lui aussi dans la hiérarchie principale s'effectue de manière similaire. Il faut générer le nom de l'attribut pour ensuite le concaténer avec le *long-id* de la *feature* contenante. Ainsi, le *long-id* de l'attribut "i" à la ligne 12 est "A.D.i". Il en va de même pour un attribut structuré tel que "s" à la ligne 13 dont le *long-id* est "A.D.s".

Par contre, dans le cas d'une structure, il faut également générer le *long-id* de chacun de ses champs, pour les "widgets" qui les représentent.

Il faut donc connaître deux choses lors de la génération du *long-id* d'un champ : le champ lui-même et l'attribut structuré dans lequel il se trouve. En effet, le champ étant défini avec le type de structure et non l'attribut structuré, sa seule connaissance n'est pas suffisante puisque le *long-id* du champ est composé de son nom et du *long-id* de la structure, et non du type qui n'est pas un élément proprement dit. De plus, connaître uniquement l'attribut structuré pour ensuite générer les *long-ids* de chacun des champs n'est pas suffisant non plus puisque rien ne dit que tous ces champs seront bien référencés par des sous-attributs définis dans la structure.

Si nous prenons l'exemple de la structure "s" à la ligne 13, seuls deux champs sont mentionnés, "astruct" et "bstruct". Si nous souhaitons générer le *long-id* du champ "rstruct" de la structure "s", nous sommes obligés de connaître à la fois le nom du champ défini dans le type (ligne 3) et l'attribut structuré. Pour la génération proprement dite, le nom du champ est concaténé avec le *long-id* de l'attribut structuré. Le *long-id* généré du champ "rstruct" est donc "A.D.s.rstruct".

Lorsque la *feature*, l'attribut ou le champ dont nous souhaitons générer le *long-id* se trouve sous une extension de *feature*, directement comme pour la *feature* "E" à la ligne 25 ou indirectement comme pour la *feature* "H" à la ligne 28, il est évidemment nécessaire de prendre cet état de fait en compte. Dans ce cas, la génération du *long-id* se fait en deux temps. Elle va d'abord se faire récursivement jusqu'à arriver à la racine locale, c'est-à-dire l'extension. Une fois arrivé à cette racine locale, il faut trouver la *feature* référencée par l'extension pour ensuite générer le *long-id* de cette *feature* et le concaténer avec ce qui a été généré précédemment.

9.6.2 Implémentation en Acceleo

Le *template* "generateLongIdFeature" (Code 9.17) montre comment s'effectue la génération du *long-id* d'une *feature*.

```
1 [template public generateLongIdFeature
2 (aFeature_Declaration:tVL::Feature_Declaration) post(trim())]
3 [comment: the hierarchical_feature can be below another hierarchical
4 feature or below the root_feature or a feature_extension/]
5 [if (not aFeature_Declaration.eContainer(tVL::Feature_Group).
6 oclIsUndefined())]
7   [if (aFeature_Declaration.eContainer(tVL::Feature_Group).
8   eContainer(tVL::Feature_Content).eContainer().
```

```
 9   oclIsTypeOf(tVL::Feature_Extension))]
10     [aFeature_Declaration.eContainer(tVL::Feature_Group).
11     eContainer(tVL::Feature_Content).
12     eContainer(tVL::Feature_Extension).
13     generateLongIdFeatureExtension()/].
14   [/if]
15   [if (not aFeature_Declaration.eContainer(tVL::Feature_Group).
16   eContainer(tVL::Feature_Content).
17   eContainer(tVL::Feature_Declaration).
18   oclIsTypeOf(tVL::Root_Feature))]
19     [aFeature_Declaration.eContainer(tVL::Feature_Group).
20     eContainer(tVL::Feature_Content).eContainer
21     (tVL::Feature_Declaration).generateLongIdFeature()/].
22   [else]
23     [aFeature_Declaration.eContainer(tVL::Feature_Group).
24     eContainer(tVL::Feature_Content).eContainer
25     (tVL::Feature_Declaration).name/].
26   [/if]
27 [/if][aFeature_Declaration.name/]
28 [/template]
```

Code 9.17 – Template generateLongIdFeature

Une *feature* peut se trouver sous une autre *feature* hiérarchique, sous une extension de *feature* ou directement sous la *feature* racine. Le *template* vérifie d'abord s'il y a une *feature* au dessus de celle en cours de traitement. Si ce n'est pas le cas, c'est que la hiérarchie des *features* a été totalement remontée jusqu'à atteindre la *feature* racine et qu'il ne reste plus qu'à générer le nom de la *feature* (ligne 27). S'il y a une *feature* au dessus, il faut voir de quel type il s'agit.

Si cette *feature* est en fait une extension (lignes 7 à 9) alors, ce *template* appelle celui générant le *long-id* d'une extension (lignes 10 à 13) qui lui-même rappelle le premier cité une fois qu'il connait la *feature* référencée par cette extension.

Si la *feature* est hiérachique, un appel récursif sur cette *feature* a lieu (lignes 19 à 21).

S'il s'agit de la *feature* racine, le *template* génère son nom (lignes 23 à 25).

Le *template* 9.13 montre l'endroit où sont générés ces *long-ids* pour un "widget" représentant un attribut entier ou réel. L'attribut "id" servant d'identifiant de balise en HTML5, il va naturellement contenir le *long-id* de l'élement que la balise/le "widget" représente. Cela sera la même chose pour les *features*, leur *long-id* étant généré et placé comme valeur de l'attribut "id".

Pour certains "widgets", à savoir les "radios" et les "checkbox", l'attribut "name" a également un intérêt particulier. En effet, celui-ci permet de les grouper. Des "widgets" qui ont la même valeur de cet attribut font partie du même groupe. Par exemple, un navigateur reconnaîtra les "radios" d'un même groupe grâce à l'attribut "name" et empêchera l'utilisateur de sélectionner plus d'une "radio" de ce groupe, ce qui est essentiel lorsque nous souhaitons représenter un groupe *oneof* par exemple. Il est également nécessaire que le contrôleur ait accès à cette information lorsqu'il propage les contraintes. De ce fait, pour les "widgets" de type "radio" et "checkbox" représentant les *features* d'un groupe, l'attribut "name" va contenir le *long-id* de la *feature* parente qui identifie le

groupe de *features* (toutes les *features* de ce groupe ont la même *feature* parente) tandis que l'attribut "id" contiendra le *long-id* de l'élément lui-même. Pour les attributs de type énumération ou booléen représentés par des "radios", l'attribut "name" contiendra le *long-id* tandis que l'attribut "id" contiendra ce *long-id* suivie de la valeur représentée (une des valeurs de l'énumération ou "true/false" pour un booleen). Pour les autres "widgets", l'attribut "name" n'a pas de réel intérêt.

9.7 Générateur en tant qu'application standalone

9.7.1 Principes

À ce stade, la solution proposée ne fonctionne que lorsqu'elle est exécutée dans l'environnement Eclipse. Afin d'avoir un outil utilisable en tant qu'application "standalone", il est donc nécessaire d'exporter le projet. Avant cela, certaines modifications du code Java généré lors de la compilation du module principal Acceleo sont nécessaires. En effet, il faut savoir que lors de l'exécution du module Acceleo en question, Eclipse effectue diverses opérations cachées au développeur. Elle enregistre notamment les URI des méta-modèles des modèles impliqués dans la transformation, les méta-modèles TVL et FCSS dans ce cas.

De plus, de base, un générateur Acceleo prend des versions XMI des modèles en entrée, des fichiers d'extensions *.xmi* donc, et non *.tvl*. Même si les versions XMI des modèles TVL et FCSS peuvent être obtenues grâce à un petit projet Java pour ensuite être données en entrée du générateur Acceleo, il serait plus intéressant de pouvoir passer directement le fichier d'extension ".tvl". L'ajout de quelques lignes de code dans la même classe Java générée permet de résoudre ce problème.

Une fois tout cela effectué, le générateur peut dès lors tourner en tant qu'application "standalone". La commande 9.18 permet de lancer une génération via cet outil, dont le nom est "TVLtoHTML5Generator".

```
1 java -jar TVLtoHTML5Generator.jar <chemin fichier TVL>
2 <chemin dossier cible>
```

Code 9.18 – Commande d'exécution du générateur

Les chemins spécifiés peuvent être absolus ou relatifs. L'outil est compatible avec les systèmes d'exploitation Windows, Linux et Mac et requiert au minimum la version 1.6 de Java pour fonctionner.

Ce passage de la version Eclipse à une version "standalone" provoque l'apparition d'exceptions "AcceleoEvaluationException" jusque là cachées par le plugin Acceleo. Elles apparaissent lorsque le générateur tente d'aller chercher quelque chose qui n'existe pas dans les modèles. Le modèle FCSS est le principal responsable de telles exceptions. En effet, il n'est évidemment jamais obligatoire de remplir complètement le modèle FCSS avec chaque attribut global et chaque attribut spécifique de chaque élémént du modèle TVL correspondant. Ainsi, si dans le modèle TVL, nous avons des groupes *oneof*, le générateur vérifiera ce que le modèle FCSS spécifie pour ces groupes. Si rien n'est spécifié une exception

"AcceleoEvaluationException" sera levée car le générateur aura récupéré une valeur "undefined". Heureusement, ce genre d'exception ne gène en rien le processus de génération. Le problème provient du fait que celles-ci sont nombreuses et peuvent induire en erreur le concepteur qui, voyant la quantité faramineuse d'exceptions dans la console, peut penser que la génération ne s'est pas déroulée correctement.

Étant donné que les exceptions font partie du flux d'erreurs, celui-ci a été redirigé vers un fichier texte. À la fin de la génération, ce fichier est supprimé ce qui permet d'éviter une mauvaise interprétation de la part du concepteur.

Cette gestion des erreurs est néanmoins problématique car elle cache toutes les erreurs qui pourraient survenir, nous y reviendrons dans le Chapitre 12 consacré notamment aux limitations de la solution.

9.7.2 Implémentation en Java

Dans la méthode "registerPackages", il est nécessaire d'ajouter le Code 9.19. Ces modifications permettent d'enregistrer les méta-modèles Ecore de TVL (lignes 1 à 5) et FCSS (lignes 7 à 11).

```
1 if (!isInWorkspace(be.ac.fundp.info.tVL.TVLPackage.class)) {
2     resourceSet.getPackageRegistry().put(
3     be.ac.fundp.info.tVL.TVLPackage.eINSTANCE.getNsURI(),
4     be.ac.fundp.info.tVL.TVLPackage.eINSTANCE);
5 }
6
7 if(!isInWorkspace(be.ac.fundp.info.fCSS.FCSSPackage.class)){
8     resourceSet.getPackageRegistry().put(
9     be.ac.fundp.info.fCSS.FCSSPackage.eINSTANCE.getNsURI(),
10    be.ac.fundp.info.fCSS.FCSSPackage.eINSTANCE);
11 }
```

Code 9.19 – Modification registerPackages

Dans la méthode "registerResourceFactories", il faut ajouter le Code 9.20. Les lignes 1 à 6 sont obligatoires pour que l'application fonctionne en "standalone", les suivantes permettent de passer un fichier d'extension ".tvl" en entrée au générateur.

```
1 resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().
2 put("ecore", new EcoreResourceFactoryImpl());
3
4 resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().
5 put(IAcceleoConstants.EMTL_FILE_EXTENSION,
6 new EMtlResourceFactoryImpl());
7
8 new TVLStandaloneSetupGenerated().
9 createInjectorAndDoEMFRegistration();
10
11 resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().
12 put(".tvl", new XMIResourceFactoryImpl());
```

Code 9.20 – Modification registerResourceFactories

9.8 Architecture du générateur Acceleo

La Figure 9.11 synthétise l'architecture des packages et modules du générateur Acceleo présentés dans les sections précédentes ainsi que les liens entre ceux-ci.

Le package "main" est le package contenant notamment le module principal, *generateConfigurator.mtl*, qui est appelé au début de la génération et dont la fonction a été globalement décrite dans la Section 9.1. Il contient également des classes Java, une générée par Acceleo, à savoir *GenerateConfigurator.java* lors de la compilation du module principal, et deux contenant des services Java utiles pour la génération, *ModelsGetter.java* qui permet de récupérer les références aux deux modèles et *SettingsGetter.java* qui contient une méthode retournant le chemin complet du fichier TVL, et appelées via des *queries* Acceleo qui se trouvent dans le dernier fichier du package, *javaServices.mtl*.

Le package "feature" contient un seul module, *generateFeature.mtl*, dans lequel sont définis tous les *templates* en rapport à la gestion des *features* et groupes de *features*.

Le package "attribute" est composé de trois modules. Le premier, *generateAttribute.mtl*, contient un seul *template* qui,en fonction du type d'attribut, simple ou structuré, va rediriger la génération vers les *templates* des modules *generateBaseAttribute.mtl* et *generateStructAttribute.mtl* respectivement.

Le package "fcss" contient un module, *manageFCSS.mtl*, avec des *templates* de gestion des attributs FCSS, notamment en ce qui concerne les labels/images des *features* et attributs, les textes d'aide, etc.

Le package "longId" contient le module *generateLongId.mtl* dans lequel se trouvent les *templates* de génération des *long-ids* pour les différents types d'éléments TVL.

Le package "queries" est composé de différents modules contenant uniquement des *queries* permettant d'accéder à divers éléments des modèles TVL et FCSS mais également à d'autres fonctions visant à limiter le code OCL dans les *templates* de génération. Au niveau des *queries* prévues pour les éléments TVL, il y a deux modules : *AttributeTVLQueries.mtl* qui regroupe les *queries* utiles pour la gestion des attributs, simples et structurés, et *FeatureTVLQueries.mtl* qui contient les *queries* appelées par les *templates* de gestion des *features* et groupes de *features*. Au niveau des éléments FCSS, nous avons trois modules : *GlobalPartFCSSQueries.mtl* contient des *queries* d'accès aux attributs globaux d'un modèle FCSS, *SpecificAttributeFCSSQueries.mtl* regroupe les *queries* d'accès aux attributs FCSS spécifiques à un attribut TVL et *SpecificFeatureFCSSQueries.mtl* regroupe les *queries* d'accès aux attributs FCSS spécifiques à une *feature*.

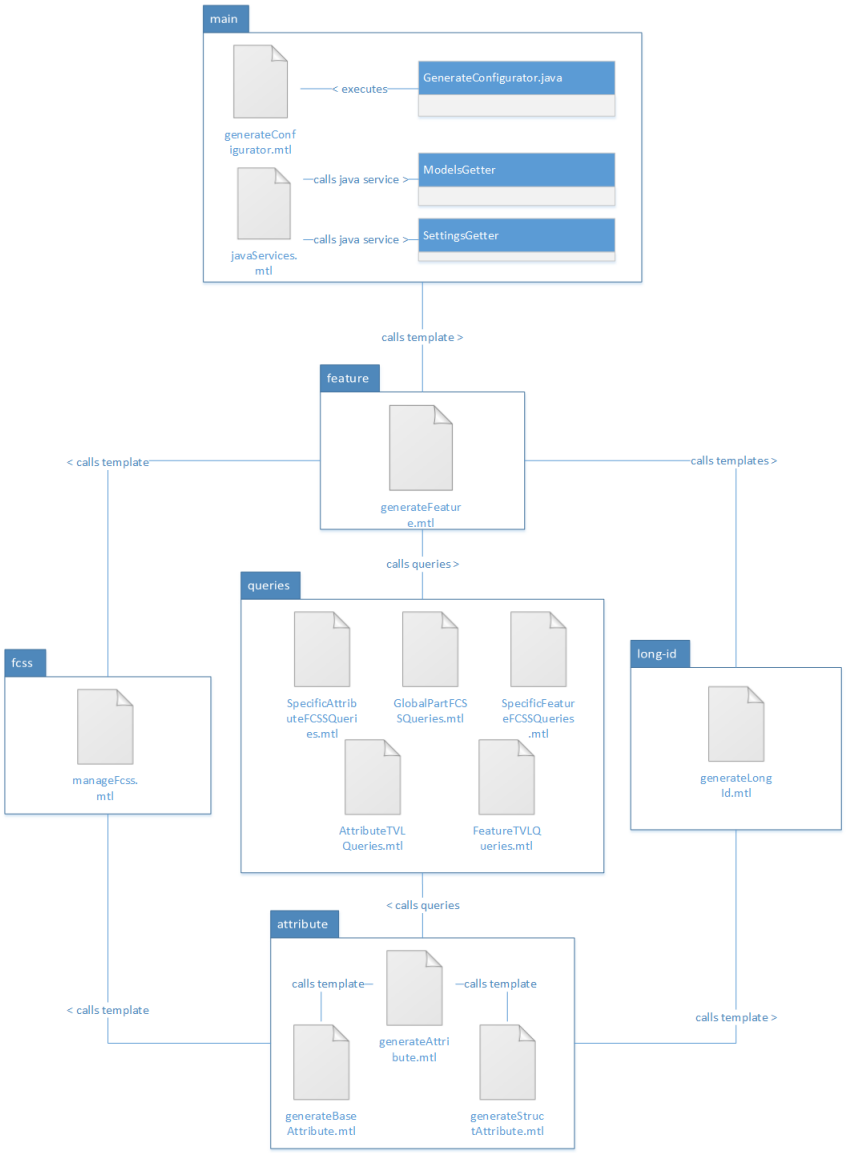

FIGURE 9.11 – Architecture des packages et modules du générateur

10 Autres contributions

10.1 Contrôleur

Pour rappel, le contrôleur effectue le lien entre l'interface et le modèle. Il a pour rôles d'écouter les actions de l'utisateur, de les transmettre à la servlet et, lorsqu'il reçoit le résultat de cette dernière, de mettre à jour la vue. Cette mise-à-jour correspond à la propagation des contraintes spécifiées dans le modèle à la vue (voir Figure 8.1).

Afin de remplir ces rôles, plusieurs fonctions sont définies. Elles sont divisées en trois catégories.

1. Les fonctions de gestion des événements : il en existe une pour chaque type de "widget" représentant les groupes de *features*. Ainsi, une fonction s'occupe de gérer la sélection et désélection de *features* dans les "listbox", une autre s'occupe de la même chose pour les groupes de "radios" et une dernière pour les groupes de "checkbox". Enfin, il y a également une fonction qui gère la sélection/désélection d'une *feature* optionnelle et donc respectivement l'affichage/la dissimulation de son contenu. Cette fonction n'est évidemment utile que lorsque la *feature* optionnelle a effectivement un contenu généré à l'intérieur d'un "fieldset". Des fonctions de gestion des "widgets" représentant visuellement des attributs TVL sont également prévues. En réalité, elles sont en partie implémentées mais, comme nous le verrons par la suite, l'API de configuration ne gère pas les attributs TVL et il n'est donc pas possible de tester ces fonctions. Elles ne sont donc pas complètes, ni liées aux autres fonction détaillées dans les deux points suivants.

2. Une fonction de communication avec la servlet. Cette fonction est appelée par toutes les fonctions de gestion des événements pour faire appel à la servlet afin de transmettre l'identifiant de l'élément modifié et sa nouvelle valeur. L'appel à la servlet prend la forme d'une requête HTTP GET :

   ```
   http://localhost:8080/TVLServer/TVLServlet?
   element="+elementId+"&value="+value+"
   ```

   Les deux variables correspondent donc à l'identifiant de l'élément TVL concerné, c'est-à-dire son *long-id*, et à la nouvelle valeur qui lui est assignée.

   La requête reçoit une liste de chaînes de caractères comme résultat qu'il va falloir traiter un à un. Chaque chaîne de caractères contient l'identifiant de l'élément dont la valeur a changé et qu'il faut propager ainsi que le caractère '!' en première position si la valeur de cet élément est "false", sinon sa nouvelle valeur est "true". Cette même fonction boucle donc sur ces résultats et appelle la dernière fonction qui effectue la propagation.

3. Une fonction de propagation des résultats sur la vue. Cette fonction ne s'occupe que de la propagation d'un résultat à la fois. Elle effectue plusieurs actions. Premièrement, elle doit retrouver le "widget" représentant l'élément identifié par l'identifiant donné. Ensuite, elle doit vérifier de quel type de "widget" il s'agit. Enfin, elle peut propager le résultat en cochant/-sélectionnant/décochant/désélectionnant le "widget".

   Dans le cas où le "widget" représentant l'élément en question est une "checkbox" ou une "radio", il est nécessaire de connaître la nature de l'élément TVL correspondant. En effet, il est possible que la *feature* soit optionnelle ce qui implique que le "widget" représente deux choses : le caractère optionnel de la *feature* et le fait qu'elle fasse partie d'un groupe de décomposition. Si la *feature* est optionnelle, en plus de sélectionner ou désélectionner celle-ci en fonction de la valeur fournie à la fonction, il faut également appeler la fonction qui gère le contenu des *features* optionnelles (l'affiche ou le cache).

   Pour reconnaître un "widget" représentant également le caractère optionnel d'une *feature* des autres widgets représentant simplement le fait que la *feature* fasse partie d'un groupe de *sous-feature*, le générateur ajoute la valeur "opt" à l'attribut "class" de l'"input" en question lorsque le "widget" représente une *feature* optionnelle.

   La fonction de gestion d'une *feature* optionnelle peut donc être appelée dans deux cas différents. Premièrement lorsque l'utilisateur coche lui-même le "widget" représentant le caractère optionnel de la *feature* (voir premier point), deuxièmement, lorsque celui-ci est coché en conséquence de la propagation de contraintes, par exemple lorsque le choix d'une *feature* implique que la *feature* optionnelle en question devienne obligatoire.

Toutes ces fonctions sont "génériques", c'est-à-dire que le contrôleur est compatible avec toutes les interfaces graphiques générées par l'outil précédemment décrit, comme expliqué dans la section sur l'architecture. Elles sont écrites dans le langage Javascript [1] et les appels à la servlet se font grâce à AJAX, pour Asynchronous JavaScript and XML [2]. Ce dernier met à disposition des classes telles que *XMLHttpRequest* (pour les requêtes dans le même domaine) et *XDomainRequest* (pour les requêtes *cross-domain*) dont les instances permettent de communiquer avec un serveur. Une dernière fonction du contrôleur est donc responsable de créer et retourner un objet *XDomainRequest* pour pouvoir envoyer les requêtes à la servlet.

10.2 Servlet

10.2.1 Principes

La servlet HTTP, écrite dans le langage Java, est le point d'entrée de la partie serveur, qui contient également l'API de configuration, de l'architecture

1. `http://www.w3schools.com/js/default.asp`
2. `http://www.w3schools.com/ajax/default.ASP`

d'un configurateur créé grâce à cette approche. C'est à elle que le contrôleur communique les changements provoqués par l'utilisateur et reçoit les résultats à propager.

À l'initialisation, elle va chercher le FM TVL afin que le solveur SAT le charge et le traduise en contraintes sous forme normale conjonctive, tout cela grâce à l'API de configuration. Pour cela, elle dispose d'un fichier "properties" qui contient le chemin absolu du fichier TVL.

Une fois initialisée, elle est prête à recevoir des requêtes HTTP GET et à les traiter grâce à une méthode dédiée. La première chose à faire lorsqu'elle reçoit une requête est d'accepter la requête "cross-domain". Une requête est considérée comme "cross-domain" lorsqu'elle provient d'un domaine différent du domaine où est hébergé la servlet.

Une fois cela fait, elle peut traiter la requête. Elle doit d'abord récupérer les valeurs des paramètres qui lui sont fournis, à savoir "element" et "valeur". Étant donné que la solution actuelle ne gère que les *features*, la valeur est toujours de type booléen. En effet, une *feature* ne peut prendre comme valeur que "true", si elle est sélectionnée, ou "false", si elle ne l'est pas, au contraire d'un attribut entier par exemple.

Ensuite, elle fait appel à l'API pour mettre à jour la contrainte représentant la configuration actuelle du FM avec l'identifiant de l'élément et sa valeur qu'elle a reçue. Elle récupère le résutat que l'API lui retourne qui est une liste de *features* dont les valeurs respectives ont changé et qu'il faut donc propager.

Cette liste est ensuite formatée en liste de chaînes de caractères, composée de l'identifiant de chaque *feature* ainsi que du caractère '!' si sa nouvelle valeur est "false", afin de pouvoir être écrite dans la réponse HTTP. Avant de l'écrire, il est encore nécessaire de concaténer ces chaînes de caractères en une seule en les séparant avec un caractère spécifique, qui ne risque pas de se retrouver dans les chaînes concaténées, à savoir ';'. La chaîne de caractères est alors enfin écrite dans la réponse.

Cette servlet fait partie d'un projet Web J2EE qui contient donc, en plus de celle-ci, l'API de configuration et le solveur ainsi que le fichier "properties" précédemment abordé. Ce projet doit être hébergé sur un serveur applicatif comme Apache Tomcat [3].

10.2.2 Exemple de résultat

Prenons pour exemple le configurateur généré à partir du FM TVL (Code 5.1) représentant une ligne d'Ordinateurs, et présenté dans l'étude de cas de la Section 11.2. Si l'utilisateur sélectionne la *feature Portable*, la servlet fera appel à l'API pour ajouter la nouvelle contrainte "*Portable*=true" et assigner les nouvelles valeurs des *features* qui doivent changer de valeur. Comme il s'agit de la première action de l'utilisateur, toutes les *features* se voient assignées une valeur. Le Code 10.1 présente le résultat retourné par l'API.

```
1 [Ordinateur [I], Processeur [I], DisqueDur [I], Type [I], Fixe [E],
```

3. http://tomcat.apache.org/

```
2 CarteGraphique [I], Modele [I], RadeonHD7990 [E], GeForceGT640M [I],
3 GeForceGTX770 [E], RadeonHD7950 [E], Constructeur [I], AMD [E],
4 NVidia [I], MemoireVive [I], ArtificialParentBatterie [I],
5 Batterie [I], ArtificialParentGarantieEtendue [I], CarteReseau [I]]
```

Code 10.1 – Résultat de l'API

La servlet va ensuite formater ce résultat pour pouvoir le retourner au contrôleur étant donné qu'à l'état actuel, il s'agit d'une liste d'objets représentant les *features* et leur valeur. "[I]" signifie "Included" c'est-à-dire que la *feature* fait partie de la configuration de *features*, sa valeur est donc "true" tandis que "[E]" signifie "Excluded" donc "false". Nous remarquons également la présence de deux *features* artificielles, *ArtificialParentBatterie* et *ArtificialParentGarantieEtendue* qui servent de parentes artificielles aux *features Batterie* et *GarantieEtendue* respectivement, cela afin de respecter la contrainte de cardinalité imposée par le groupe en plus de prendre en compte le caractère optionnel des *features* en question. Le solveur assigne la valeur "true" à la *feature* artificielle afin de vérifier la cardinalité de groupe mais ne sélectionne pas automatiquement la *feature* optionnelle afin de respecter son caractère non-obligatoire.

Le Code 10.2 présente le résultat formaté, le caractère '!' étant ajouté devant le *short-id* de la *feature* lorsque sa valeur est false et le caractère ';' étant utilisé comme séparateur.

```
1 Ordinateur;Processeur;DisqueDur;Type;!Fixe;CarteGraphique;Modele;
2 !RadeonHD7990;GeForceGT640M;!GeForceGTX770;!RadeonHD7950;
3 Constructeur;!AMD;NVidia;MemoireVive;ArtificialParentBatterie;
4 Batterie;ArtificialParentGarantieEtendue;CarteReseau
```

Code 10.2 – Résultat formaté pour le contrôleur

Ce résultat sera retourné au contrôleur qui devra alors boucler sur ces valeurs et en fonction de l'absence ou de la présence du caractère '!', sélectionnera ou désélectionnera le "widget" de la *feature* correspondante.

Troisième partie

Evaluation et conclusions

11 Etudes de cas

Le présent chapitre illustre l'approche et les résultats obtenus via deux études de cas. La première consiste en un exemple fourni à l'équipe de PReCISE par un de leurs partenaires et la seconde reprend l'exemple du FM TVL (Code 5.1) introduit dans la Section 5.1.

La présentation de chacune d'entre elles est divisée en trois parties. La première présente le modèle TVL et le résultat de la génération sans modèle FCSS. La seconde présente le modèle FCSS et le résultat de la génération lorsqu'il est intégré à celle-ci. La troisième présente le résultat suite à la propagation de contraintes sur l'interface utilisateur.

11.1 Première étude : Tune

Cette première étude de cas est fournie par le projet Naples [1]. Ce projet vise à encourager la réutilisation dans les processus de développement, notamment en implémentant des "workflows" standards et en permettant aux clients, en général des petites entreprises, de les configurer en fonction de leurs besoins. Dans ce but, les membres de l'équipe PReCISE ont voulu d'abord implémenter les "workflows" du standard ISO/IEC 29110 [48] dans Bonita [2], un moteur de "workflows". Ce moteur est accompagné d'un outil permettant de créer des formulaires Web utilisés ici pour la configuration de "workflows" via des questions. Cependant, des configurateurs basés sur des questionnaires requièrent également que des contraintes soient définies et s'appliquent aux questions, ce qui n'est pas supporté par Bonita. Nous allons voir ici le résultat qu'il est possible d'obtenir en générant un configurateur à partir de modèles TVL et FCSS.

La Figure 11.1 représente l'interface graphique du configurateur de "workflows" créé avec Bonita.

L'objectif de cette étude est de montrer qu'il est possible d'obtenir un résultat très proche visuellement de cette interface graphique en utilisant la génération de code HTML5 à partir des modèles TVL et FCSS présentés par la suite.

1. `http://www.cetic.be/NAPLES`
2. `http://www.bonitasoft.com/`

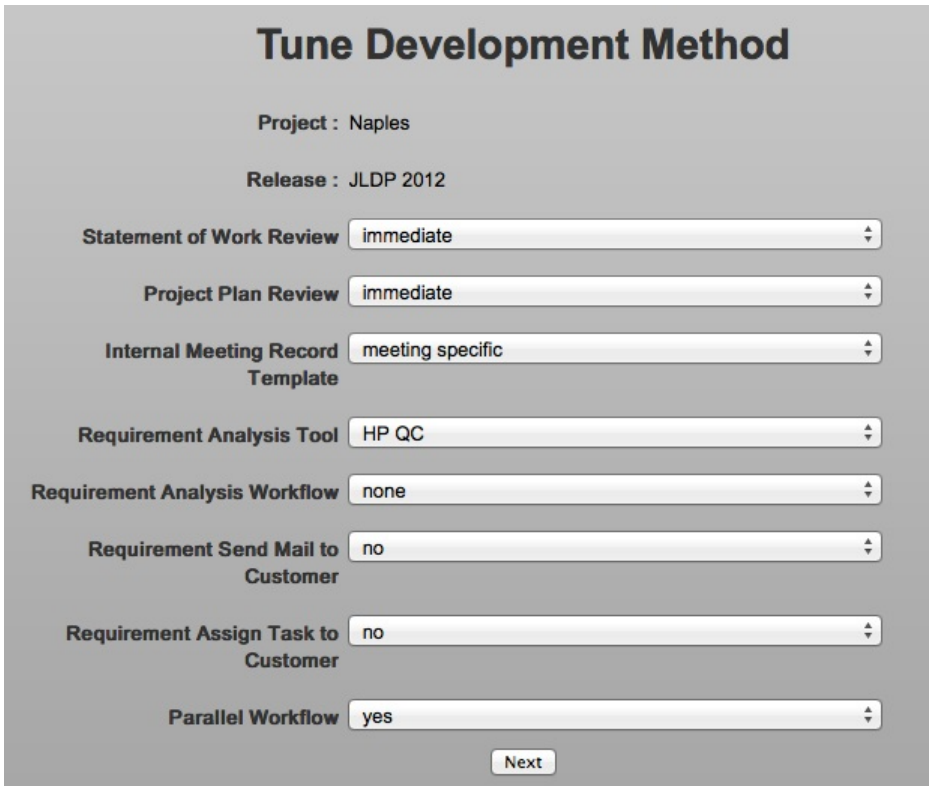

FIGURE 11.1 – GUI créée avec Bonita

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

Code 11.1 – FM TVL Tune

Le FM TVL (Code 11.1) en question est un modèle assez simple présentant la *feature* racine *Tune* possédant un groupe de cardinalité *allof*. Toutes les *features* de ce groupe ont une décomposition *oneof*. Deux *features* sont optionnelles :

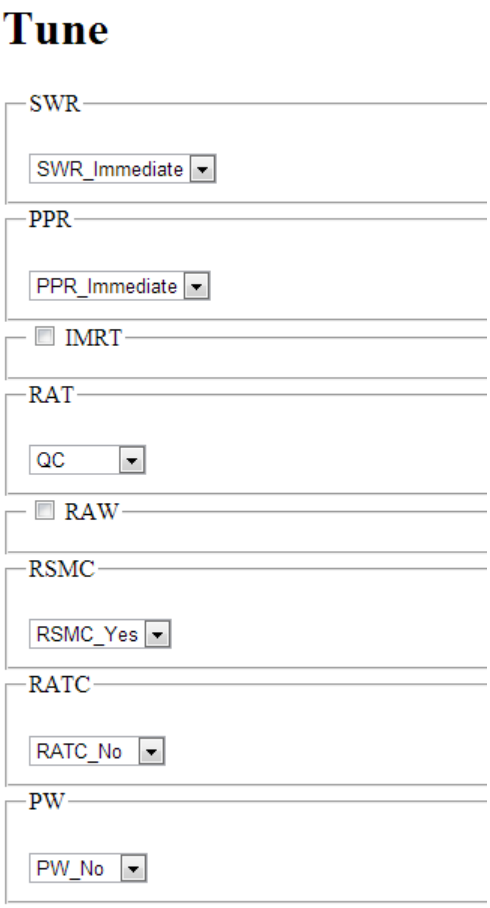

FIGURE 11.2 – Tune - 1ère génération, sans FCSS

IMRT et *RAT* (lignes 8 et 10).

Trois contraintes sont également énoncées. La première (ligne 2) signifie que si la *feature PPR_ Meeting* est sélectionnée, la *feature IMRT* devient obligatoire. La seconde (ligne 3) est similaire, la sélection de la *feature SWR_ Meeting* implique que la *feature IMRT* soit également sélectionnée. La troisième (ligne 4) signifie que si la *feature RSMC* a la valeur *No*, c'est-à-dire que la *feature RSMC_ No* est sélectionnée, alors la *feature RATC* prend la valeur *Yes* représentée par la *feature RATC_ Yes* du groupe.

La génération du code HTML5 uniquement à partir de ce modèle TVL produit le résultat présenté à la Figure 11.2.

Les *features* optionnelles *IMRT* et *RAW* ne sont pas cochées donc leur contenu n'est pas visible.

Globalement, la structure est assez similaire à l'interface graphique souhaitée présentée au début de cette étude de cas. Cependant, les labels des différentes options sont différents et n'ont que peu de sens pour l'utilisateur lambda dans le cas de l'interface générée. L'emplacement des "widgets" par rapport aux labels est également différent. Pour se rapprocher encore plus de celle-ci, il est nécessaire d'écrire un modèle FCSS. Ce modèle est présenté dans le Code 11.2.

```
1 import "tune.tvl"
2
3 Tune{label: "Tune Development Method";}
4 SWR {label:"Statement of work review";}
5 PPR {label:"Project plan review";}
6 IMRT {label:"Internal meeting record template";}
7 RAT {label:"Requirement analysis tool";}
8 RAW {label:"Requirement analysis workflow";}
9 RSMC {label:"Requirement send mail to customer";}
10 RATC {label: "Requirement assign task to customer";}
11 PW {label: "Parallel workflow";}
12 SWR_Immediate{label: "immediate";}
13 SWR_Meeting{label: "meeting";}
14 PPR_Immediate{label: "immediate";}
15 PPR_Meeting{label: "meeting";}
16 Specific{label: "meeting specific";}
17 Standard{label: "meeting standard":}
18 QC{label: "HP QC";}
19 Redmine{label: "Redmine";}
20 OSL{label: "OSL";}
21 RSMC_Yes{label: "yes";}
22 RSMC_No{label: "no"};
23 RATC_No{label: "no"};
24 RATC_Yes{label: "yes";}
25 PW_No{label: "no"};
26 PW_Yes{label: "yes";}
```

Code 11.2 – FCSS Tune

L'élaboration du modèle FCSS, dans le cas présent, a pour objectif de préciser les labels des différentes *features* qui viendront remplacer les acronymes utilisés dans le modèle TVL afin de les rendre compréhensibles pour l'utilisateur.

La Figure 11.3 montre le nouveau résultat obtenu à partir des deux modèles, plus proche de l'interface souhaitée.

Pour coller encore plus à l'interface graphique donnée en exemple, il est bien sûr possible d'ajouter un peu de code CSS, par exemple pour changer la couleur de fond, centrer les différents *widgets* ou encore supprimer les bordures des *fieldsets*. Pour rappel, l'outil génère une ligne permettant au fichier HTML d'être lié à une feuille de style appelée "style.css".

La Figure 11.4 illustre le résultat après application du code CSS, encore un peu plus proche de la Figure 11.1.

Au delà de l'aspect visuel du configurateur, désormais proche de l'interface graphique souhaitée, il est nécessaire de vérifier que le configurateur se comporte correctement vis-à-vis des contraintes, c'est-à-dire qu'il propage correctement celles-ci lors de manipulations de l'utilisateur.

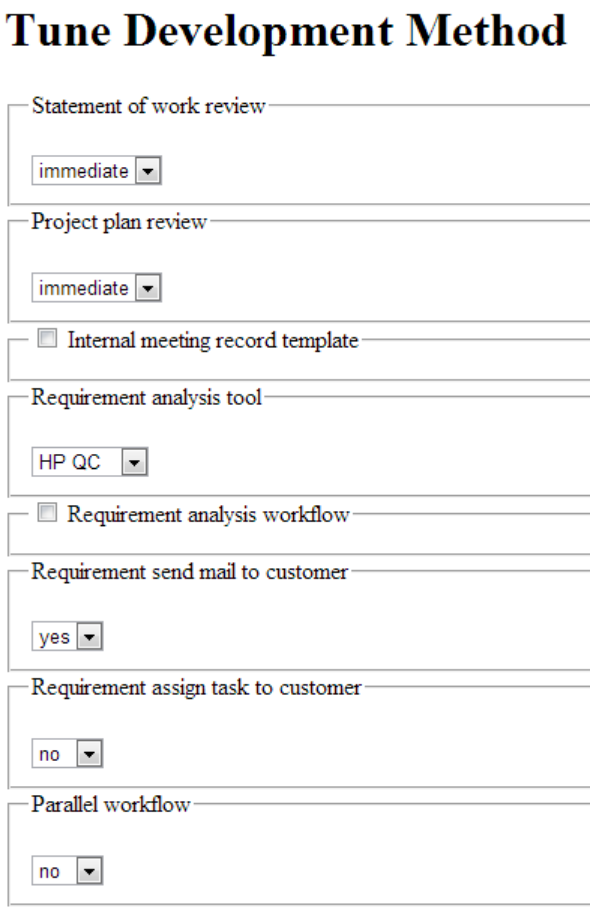

FIGURE 11.3 – Tune - 2ème génération, avec FCSS

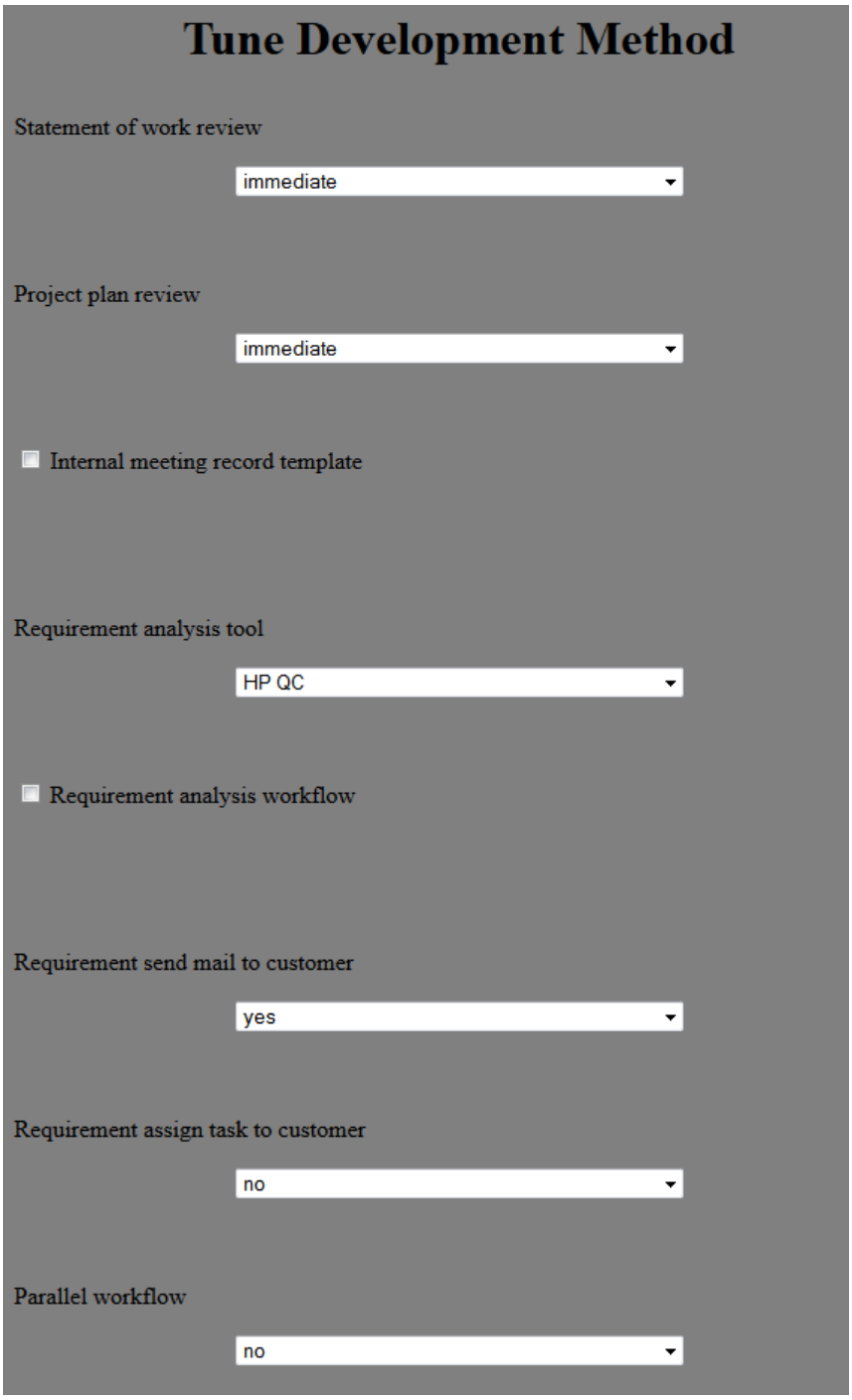

FIGURE 11.4 – Tune - 2ème génération, avec FCSS et CSS

La Figure 11.5 donne l'état du configurateur lorsque l'utilisateur a effectué plusieurs actions ayant provoqué la propagation des contraintes spécifiées.

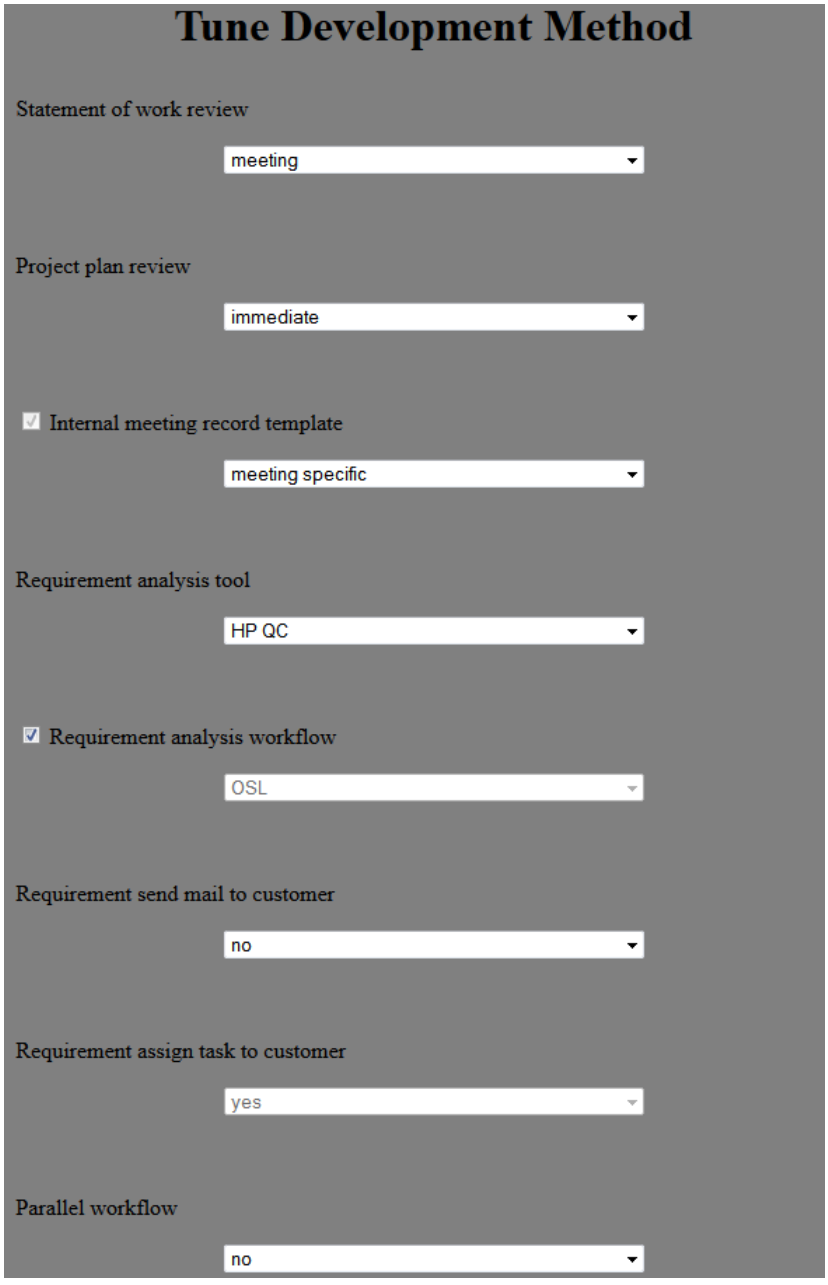

FIGURE 11.5 – Tune - État résultant de la propagation des contraintes

Pour obtenir ce résultat, l'utilisateur a choisi la *feature SWR_ Meeting* dans la liste des *features* de *SWR*. Le configurateur a dès lors propagé la contrainte *SWR_ Meeting -> IMRT* et donc coché la *feature IMRT*. Étant donné que, par défaut, le contenu d'une *feature* optionnelle est caché à l'utilisateur, le fait que celle-ci s'est retrouvée cochée automatiquement a aussi rendu son contenu visible. Ensuite, l'utilisateur a sélectionné la *feature RSMC_ No* ce qui a provoqué la propagation de la contrainte *RSMC_ No -> RATC_ Yes* et donc la sélection de la *feature RATC_ Yes*. Enfin, en cochant la *feature* optionnelle *RAW*, son contenu a été rendu visible. Même si, en dehors de l'affichage du contenu de la *feature*, aucun autre changement n'est visible, tout le processus de vérification des contraintes passant par le contrôleur et toute la partie serveur du configurateur s'est déroulé de la même manière que pour les deux autres actions de l'utilisateur.

Cette étude de cas donne donc un résultat plutôt positif des capacités du générateur même si elle ne montre notamment pas toutes les possibilités de personnalisation de l'interface graphique grâce au langage FCSS. L'étude suivante va montrer d'autres possibilités liées aux deux langages mais également une partie des limites de la solution actuelle.

11.2 Deuxième étude : Configuration d'un ordinateur

La deuxième étude concerne l'exemple que nous avons introduit dans la Section 5.1 sur TVL.

Pour rappel, le modèle TVL 5.1 représente, sommairement, la variabilité d'une ligne d'ordinateurs fictive. Le but ici est d'obtenir un configurateur permettant au final à l'utilisateur de personnaliser, dans une certaine mesure, son futur PC comme le permettent la plupart des constructeurs actuels. Ainsi, l'utilisateur pourra choisir différents composants de l'ordinateur comme le processeur, la carte graphique, le disque dur, etc.

La Figure 11.6 présente le résultat graphique obtenu en ne prenant que ce modèle en compte. Nous remarquons, étant donné la différence de taille du modèle par rapport au précédent cas, que le formulaire est également plus long. Comme nous en parlerons dans le Chapitre 12, cela peut devenir un problème lorsque le modèle est trop imposant. En dehors de cela, les différents "widgets" sont générés selon les règles de transformations par défaut, les labels correspondent aux *short-ids* des éléments et ne sont donc pas spécialement adaptés, pour la plupart même s'ils sont compréhensibles. De plus, nous aimerions fournir une aide à l'utilisateur concernant certains éléments. Par exemple, l'utilisateur lambda ne connait peut-être pas l'avantage principal d'un disque dur SSD par rapport à un disque dur HDD "classique".

Afin de personnaliser cette interface graphique, le modèle FCSS 5.2 est intégré à la génération. La Figure 11.7 présente le nouveau résultat.

Comme nous pouvons le remarquer, la plupart des attributs FCSS ont bien été appliqués, que ce soient les labels ou les "widgets". Nous voyons également bien la priorité des constructions FCSS. Par exemple, bien que l'attribut global

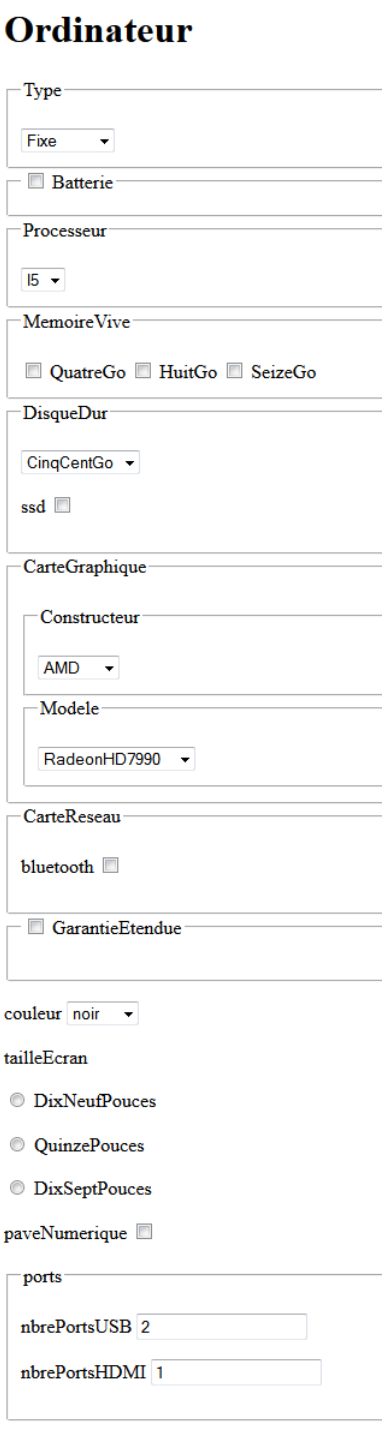

FIGURE 11.6 – Ordinateur - 1ère génération, sans FCSS

FIGURE 11.7 – Ordinateur - 2ème génération, avec FCSS

enumAttribute s'appliquant à toutes les énumérations a pour valeur *listbox* (ligne 12), c'est l'attribut spécifique *widget* à la ligne 68 qui est pris en compte lors de la génération de l'attribut TVL *couleur*.

En ce qui concerne les textes d'aide, ils sont bien générés également même si la Figure 11.7 ne le montre pas. Si l'utilisateur passe sa souris sur le label du *fieldset* représentant la *feature Garantie*, il peut voir le texte d'aide. Il en va de même pour tous les autres textes d'aide, celui sur les disques durs s'affichant lorsque l'utilisateur passe sur le label du *fieldset* tandis que celui sur les tailles d'écrans apparait lorsque l'utilisateur passe sa souris sur la *listbox*. La Figure 11.8 montre à quoi ressemblent ces textes d'aide.

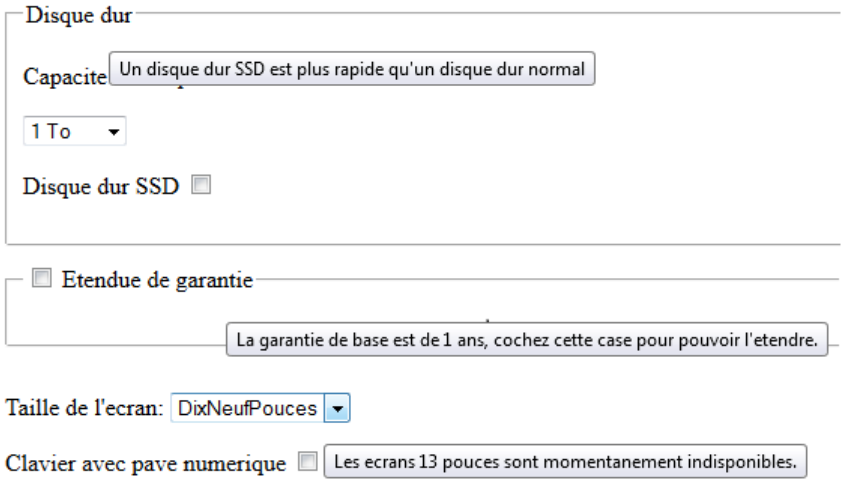

FIGURE 11.8 – Ordinateur - Exemples textes d'aide

Néanmoins, certaines constructions FCSS ne sont pas prises en compte. Le lecteur attentif remarquera que c'est notamment le cas des deux dernières lignes du modèle FCSS qui fournissent des labels aux champs de la structure. En effet, ces lignes s'appliquent à des *sub-attributes* qui correspondent aux "contraintes" qui référencent les champs de la structure (lignes 43 et 44 du modèle TVL 5.1). Or, lors de la génération, celle-ci boucle sur les champs (lignes 6 et 7 du modèle TVL 5.1) et non les *sub-attributes*, comme expliqué dans la Section 9.5 et le FCSS n'est pas vérifié pour ceux-ci. Les attributs spécifiques FCSS ne sont pas applicables aux champs d'un type structuré car le type de ces champs n'est pas un sous-type d'attribut TVL.

Nous remarquons également que les lignes 13 et 48 indiquant que les groupes doivent encadrer leur contenu, cela pour tous les groupes comme l'indique la ligne 13 et pour le groupe spécifique à la ligne 48, ne sont pas prises en compte lors de la génération. Un *fieldset* aurait dû être généré pour les groupes comme c'est le cas pour les *features* non vides.

Ces constructions font partie des quelques constructions non gérées par le générateur dans sa version actuelle.

Comme pour l'étude de cas précédente, le résultat visuel n'est pas forcément tel que le concepteur le souhaite. Ainsi, il peut encore une fois créer une feuille de style pour embellir l'interface graphique. Il peut également apporter quelques modifications mineures au fichier HTML5 afin, par exemple, de modifier les labels qui posent encore problème.

Nous allons maintenant voir le résultat après manipulation de la part de l'utilisateur. Pour cela, nous devons fournir au composant serveur une version sans attribut du modèle TVL. En effet, nous y reviendrons dans le Chapitre 12 mais l'API de configuration, à l'heure de la rédaction de ce mémoire, ne gère pas les attributs TVL autres que ceux de type booléen et ceux-ci peuvent la rendre instable.

La Figure 11.9 présente le résultat après propagation des contraintes et autres changements dûs aux actions de l'utilisateur. Une feuille de style a également été appliquée et les labels indiquant les nombres de ports USB et HDMI ainsi que les valeurs de la "listbox" permettant de choisir la taille de l'écran ont été modifiés.

Pour arriver à cet état, l'utilisateur a dû effectuer plusieurs actions. Premièrement, il a choisi la *feature Portable* ce qui, par propagation de la contrainte *Portable -> Batterie*, provoque la sélection de la *feature Batterie*. Il peut alors choisir le type de batterie qu'il souhaite en sélectionnant une des deux *features* filles de la *feature Batterie*. Ici, il a sélectionné la *feature NeufCellules*. Avec la sélection de la *feature Portable*, la contrainte *Portable -> NVidia && GeForceGT640M* est également propagée ce qui fait que l'utilisateur n'a plus le choix concernant les cartes graphiques, un seul modèle étant un modèle pour portable. Il a ensuite choisi le nombre de Go de RAM (*feature HuitGo*), le processeur (*feature I5*), la couleur (valeur *blanc* de l'attribut *couleur*), etc.

Tout semble s'être déroulé correctement. Malheureusement, si l'utilisateur continue ses manipulations, il remarquera vite que ce n'est pas le cas. En effet, à cause d'une mauvaise gestion de l'état courant du modèle par l'API de configuration, l'utilisateur se retrouve plus ou moins bloqué. Certains "widgets" sont grisés donc bloqués. Certes, il peut encore choisir certains éléments comme la taille de la RAM, la taille de l'écran, l'extension de garantie, etc. Par contre, il ne peut revenir sur ses choix ni sur ce que ceux-ci ont provoqué. Ainsi, même s'il change d'avis et choisit un PC fixe, ce que l'interface permet encore de faire, celle-ci ne se met pas à jour en "dégrisant" certains "widgets" qui devraient l'être comme ceux représentant les différentes cartes graphiques. D'autres possibilités lui sont également refusées.

Un autre problème peut également être aperçu grâce à cet exemple. Cette capture d'écran représente l'interface graphique sous Mozilla Firefox et comme nous pouvons le voir, les "input" de type "number" en HTML5 ne sont pas rendus de la même façon que sous Google Chrome par exemple. En effet, il manque les flèches qui, en plus de diminuer les chances que l'utilisateur se trompe en entrant des données incorrectes (des caractères alphabétiques plutôt que des nombres), vérifient que l'utilisateur ne descend pas en dessous de la valeur mi-

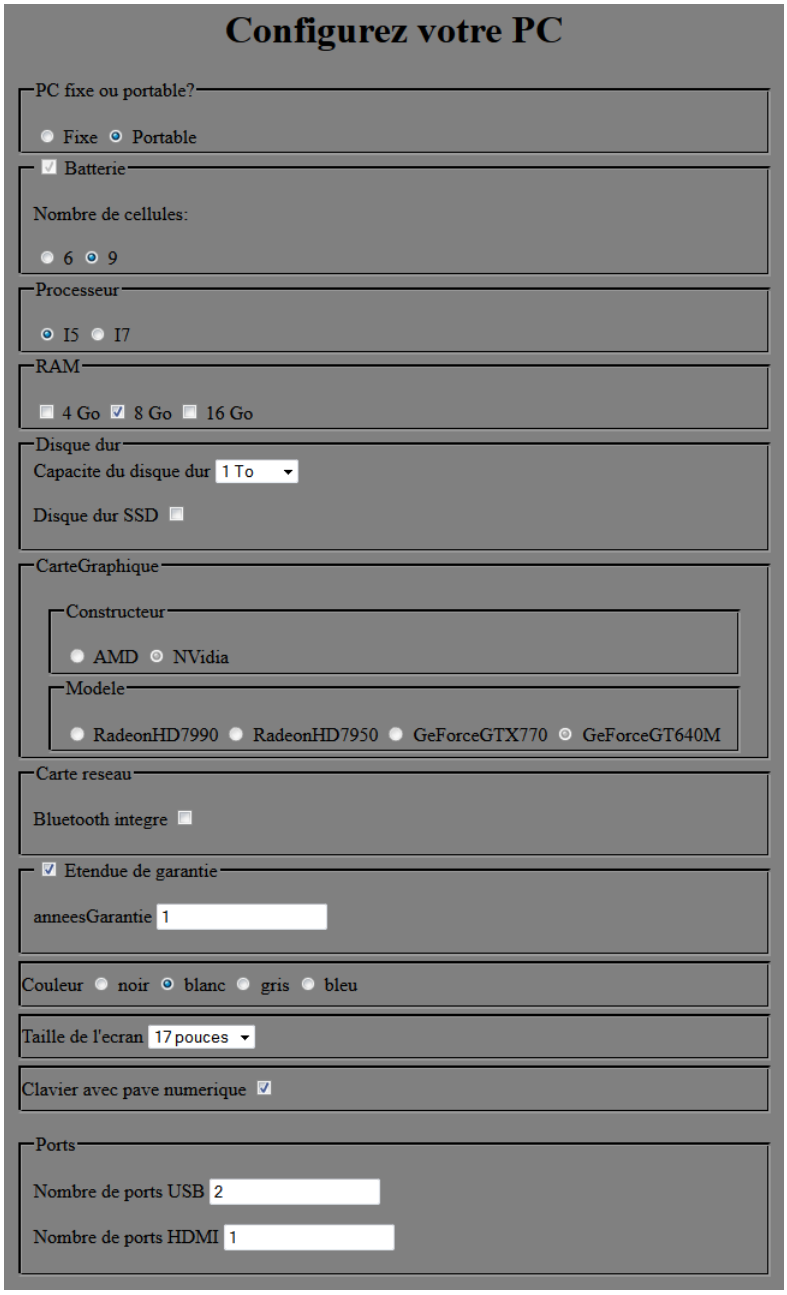

FIGURE 11.9 – Ordinateur - État résultant de la propagation des contraintes

nimale ou monte au dessus de la valeur maximale, si elles ont été spécifiées. Cette construction fait partie des constructions non supportées par Mozilla Firefox. En effet, aucun navigateur n'est actuellement totalement compatible avec HTML5 comme nous l'avons vu dans la Section 6.2.

Avec cette étude, nous avons remarqué quelques limitations de la solution proposée. Celles-ci seront abordées plus en détail dans le prochain chapitre qui proposera également des pistes de solution.

12 Travaux futurs

12.1 Limitations

Malgré des résultats globalement positifs, la solution n'est pas exempte de défauts. Ce chapitre a pour but d'énumérer et d'expliquer ses limitations pour ensuite proposer des éléments de réponses sur la façon dont la solution pourrait être améliorée.

12.1.1 Limitations du générateur

Tout d'abord, plusieurs constructions TVL et/ou FCSS ne sont actuellement pas gérées par le générateur :

1. Pour les groupes de *features* non vides, seules les "checkbox" et "radios" peuvent être générées comme "widget" de sélection. Les "listbox" ne sont pas gérées. Le modèle FCSS n'est également pas pris en compte pour la génération du "widget". Ainsi, si le concepteur souhaite une "listbox" pour un groupe *oneof* de *features* non vides, le générateur ne le prendra pas en compte et générera d'office des "radios". Par contre, les "widget" des groupes de *features* vides sont, eux, gérés comme il se doit et le modèle FCSS est bien pris en compte comme nous l'avons vu via plusieurs exemples.
2. Pour les *features* optionnelles, le "widget" "listbox" qui contiendrait les valeurs "Yes" et "No" avec cette dernière comme valeur par défaut n'est pas géré. Seules les "checkbox" et "radios" le sont.
3. Les attributs entiers et réels dont la valeur est limitée à une liste de valeurs possibles. Imaginons qu'au lieu d'avoir un entier dont le domaine est limité par deux bornes comme [2..10], nous ayons un entier limité par une liste de valeurs qu'il peut prendre, comme pour un attribut de type énumération, 2,5,7,10 par exemple. Dans ce cas, la "textbox" qui est un "input" de type "number" n'est pas un "widget" qui convient puisque la distance (le "step") entre deux valeurs n'est pas forcément la même pour chaque paire de valeurs (différence de 3 entre 2 et 5, de 2 entre 5 et 7 et de nouveau de 3 entre 7 et 10). Une liste de valeurs, représentée soit par une "listbox" soit par un "radiogroup", serait plus appropriée, comme c'est le cas pour les énumérations. Cependant, à l'heure actuelle, aucun autre "widget" ne peut être généré pour les attributs entiers et réels. Le langage FCSS devrait d'ailleurs être également étoffé à ce niveau pour permettre au concepteur de choisir entre ces différents "widgets" en fonction de la situation.
4. L'attribut global FCSS *groupContainer* et l'attribut *container* spécifique à un groupe indiquant si le groupe doit encadrer son contenu n'est pas pris en compte par le générateur.

5. Les cardinalités des groupes pourraient ne pas être respectées. En effet, un groupe *oneof* correspond à la cardinalité [1..1]. Or lorsque des "radios" sont générées pour le représenter, aucune n'est cochée automatiquement. Si l'utilisateur ne coche rien dans ce groupe, la contrainte liée à la cardinalité ne sera pas respectée. Le problème est le même pour les groupes *someof* et *card* représentés par des "checkbox". Par contre, si le modèle FCSS spécifie la *feature* à cocher par défaut, le générateur la cochera et la cardinalité sera donc respectée dès le début. De même, si un groupe *oneof* est représenté par une "listbox", la cardinalité [1..1] sera cette fois-ci respectée car la première valeur de la "listbox" est d'office sélectionnée. Seulement, si une *feature* optionnelle se trouve dans ce groupe, la cardinalité passe à [0..1] ce qui signifie que la "listbox" devrait contenir une valeur vide ou une valeur représentant le fait que l'utilisateur ne choisit aucune *feature* du groupe. Or, le générateur ne génère pas cette valeur.
6. Le langage FCSS ne permet pas de spécifier des attributs spécifiques aux *Record_Fields* qui sont les champs des types d'attributs structurés. Il est uniquement possible de définir des attributs FCSS spécifiques aux attributs TVL simples, structurés et les *sous-attributs*), or un *sous-attribut* correspond plus à une contrainte qu'à un réel attribut. D'un autre côté, le type *Record_field* n'est pas un sous-type d'attribut ce qui fait que les attributs FCSS spécifiques aux attributs TVL ne s'appliquent pas à eux. Pour contourner ce problème, il aurait été possible de récupérer ce qui est spécifié pour un sous-attribut référençant le champ mais le générateur ne le fait pas actuellement. Cependant, cela n'est pas une solution totalement valable puisqu'il est possible qu'un champ ne soit référencé par aucun *sous-attribut*.

De plus, le résultat n'est pas toujours satisfaisant visuellement parlant, au niveau du placement des différents "widgets". En effet, la génération est une génération en cascade, qui ne considère pas l'endroit où doit être placé tel ou tel "widget" représentant tel ou tel élément TVL. Ainsi, bien que le langage FCSS apporte un certain degré de configuration au niveau de l'aspect des éléments individuels, il n'est pas possible de choisir comment les placer les uns par rapport aux autres. Tout est généré dans la même page HTML et les éléments sont emboités de la même manière qu'ils le sont dans le modèle TVL. Cela peut vite devenir problématique lorsque la taille du modèle explose. À l'heure actuelle, il s'agit sans doute de la limitation principale du générateur conçu.

Néanmoins, cette limitation n'est nullement une surprise étant donné que l'objectif principal fixé était avant tout de gérer les différentes constructions TVL et FCSS, ce qui est le cas, à quelques exceptions près comme nous l'avons vu précédemment, au contraire de la structuration de ceux-ci dans l'interface.

Une autre limitation du générateur concerne le code HTML5 généré. En effet, celui-ci est en grande partie mal indenté ce qui le rend difficilement lisible par moment. Cela est particulièrement le cas si le FM TVL est conséquent, ce qui implique beaucoup de niveaux d'emboitement. Si ce problème d'indentation est difficilement contournable lors de la génération, il existe néanmoins des outils

permettant d'indenter du code HTML existant. Des outils disponibles en ligne tels que Tabifier [1] ou encore FreeFormatter [2] produisent des résultats plutôt convaincants et "user-friendly".

D'autres limitations liées au langage HTML5 ont déjà été présentées dans la Section 6.2 le concernant. Pour rappel, les navigateurs actuels ne gèrent pas complètement HTML5 et les rendus peuvent varier comme nous l'avons vu dans l'étude de cas de la Section 11.2. La compatibilité des différents navigateurs avec HTML5 est évaluée par [7] qui montre tout de même que les dernières versions des navigateurs ont un degré de compatibilité supérieur à leurs aînées. Ce problème doit donc être réglé par les développeurs de ces navigateurs et n'est pas de notre fait. D'ailleurs, pour les différents exemples présentés, le code HTML5 généré a été validé par le validateur W3C [3]. Bien sûr, ce validateur n'est pas encore "définitif" étant donné qu'HTML5 n'est pas encore un standard W3C.

Enfin, comme il a déjà été abordé dans la Section 9.7, la gestion des erreurs "AcceleoEvaluationException" est problématique. Elles sont gérées grâce à une redirection du flux d'erreurs vers un fichier qui est supprimé à la fin de la génération, cela afin que le concepteur ne soit pas induit en erreur par ces "fausses erreurs". En effet, de telles exceptions n'ont pas d'impact sur la génération en dépit de ce que le concepteur pourrait penser. Néanmoins, cette solution n'est pas totalement satisfaisante. Cela est dû au fait que tout le flux d'erreurs est redirigé vers le fichier et que celui-ci est supprimé. Ainsi, si un autre type d'erreur survenait pendant la génération, sa trace serait perdue et le concepteur ne serait pas au courant de l'erreur. En pratique, cela n'est jamais arrivé mais rien ne permet d'affirmer que ce sera toujours le cas. Dès lors, il est difficile d'évaluer la validité de l'outil.

Une autre solution possible aurait été de laisser le fichier à la fin de la génération, celui-ci pouvant servir de fichier log. Cependant, cette solution n'est pas vraiment satisfaisante non plus, les exceptions étant si nombreuses qu'il est en réalité impossible, ou en tout cas, extrêmement laborieux, de s'y retrouver. En fait, la solution la plus simple serait simplement d'attraper ces exceptions dans la classe Java qui contient le "main" du générateur mais cela ne fonctionne pas.

En réalité, ce problème vient du fait que certaines *queries* ne sont pas assez bien pensées. En effet, ces exceptions proviennent d'éléments "undefined" qui ne sont pas vérifiés. La plupart des *queries* pourraient donc être améliorées. Comme exemple de *query* qui ne provoque jamais de telle exception, nous pouvons prendre la *query* qui récupère la représentation graphique d'un groupe (Code 9.11). En effet, celle-ci ne retournera jamais "undefined" car ce qui peut être "undefined" est à chaque fois contrôlé. Si l'attribut spécifique n'existe pas, la *query* vérifie l'attribut global jusqu'à retourner la représentation par défaut si rien n'est spécifié dans le modèle FCSS concernant le groupe en question. Par contre, la *query* qui récupère la valeur par défaut d'un attribut (Code 9.14) va

1. `http://tools.arantius.com/tabifier`
2. `http://www.freeformatter.com/html-formatter.html`
3. `http://validator.w3.org/`

retourner "undefined" si aucune valeur par défaut n'est donnée. En effet, elle ne fait que vérifier si cette valeur est donnée via une constante. Si c'est le cas, elle retourne la valeur de la constante, sinon, elle retourne l'autre valeur. Seulement, cette dernière n'existe pas si aucune valeur par défaut n'a été spécifiée et est donc "undefined". La vérification est effectuée au niveau du *template* appelant mais l'exception Acceleo a apparemment déjà été lancée.

Ces problèmes au niveau des *queries* sont dûs au fait que les exceptions Acceleo n'apparaissent jamais lorsque le générateur est lancé à partir du module principal et non de la classe Java générée. Elles ne furent donc remarquées qu'à la fin du développement de l'outil. Comme le fait de retourner des élements "undefined" ne semblaient pas poser de problème lors de la génération, le contrôle n'a pas été systématique.

Il faudrait donc faire en sorte que les *queries* ne retournent jamais "undefined" mais des valeurs spécifiques lorsque elles ne trouvent pas les "bonnes" valeurs. Par exemple, pour les "widgets" des attributs, une *query* devrait faire la même chose que la *query* qui récupère la représentation graphique d'un groupe (Code 9.11), c'est-à-dire vérifier la partie spécifique FCSS puis la partie globale et enfin retourner la représentation par défaut si rien n'est spécifié. Dans la version actuelle, ce sont les *templates* qui font ces vérifications. Pour d'autres *queries*, des valeurs spécifiques pourraient être retournées. Si nous reprenons la *query* qui récupère la valeur par défaut d'un attribut (Code 9.14), elle pourrait retourner la valeur minimale si elle est spécifiée et que la valeur par défaut ne l'est pas ou "0" si ni valeur par défaut ni valeur minimale n'est trouvée.

12.1.2 Limitations de la partie dynamique

Les limitations de la partie dynamique sont en grande partie dues au fait que l'API de configuration, considérée comme existante et donc en dehors du cadre de ce mémoire, est incomplète et comporte des erreurs d'implémentation.

Plusieurs problèmes concernant cette API ont été détectés :

1. L'API ne gère pas les attributs TVL autres que les attributs booléens. Elle ne fonctionne que si le modèle TVL ne contient que des *features* et éventuellement des attributs booléens.
2. L'API ne gère pas les *long-ids* ce qui est problématique dans l'optique ou plusieurs *features* peuvent avoir le même nom tant qu'elles ne se trouvent pas dans le même sous-arbre du modèle. Le générateur a d'ailleurs dû être modifié pour ne générer que les noms (*short-ids*) des *features* afin que le résultat soit compatible avec cette API qui ne comprend que ceux-ci.
3. L'API retourne parfois des résultats erronés. Évidemment, cela a un impact sur la propagation des contraintes qu'effectue le contrôleur. L'état résultant de l'interface graphique peut donc lui aussi être incorrect. Par exemple, il va parfois retourner toutes les *features*, comme si elles avaient toutes été modifiées par un changement, ce qui fait que le configurateur se retrouve bloqué car tous les "widgets" sont grisés.

Ces erreurs sont liées à la version de l'API fournie lors du développement du générateur et des autres contributions. Celle-ci était alors en cours de développement et donc incomplète. Entre temps, l'API a été corrigée.

Par conséquent, suite à ces problèmes, le contrôleur est, lui aussi, incomplet. Il ne gère pas les événements provoqués par la modification d'une valeur d'un attribut par exemple. Il est également possible que certains bugs n'apparaissant que dans des situations précises soient encore présents, ces situations ne pouvant survenir avec une gestion de la dynamique aussi limitée.

Un autre problème du contrôleur, qui n'est pas réellement une limitation, concerne la politique à appliquer concernant la propagation des contraintes. En effet, actuellement, celles-ci sont gérées en grisant les *features* qui deviennent obligatoires ou interdites suite à la sélection ou la désélection d'une *feature*. Cependant, ce n'est peut-être pas une bonne politique étant donné qu'elle restreint les actions que l'utilisateur peut effectuer. Actuellement, il n'y a donc pas de politique bien définie ni de moyen de la modifier si le client souhaite un autre traitement que celui par défaut. Dans le futur, il serait sans doute intéressant d'ajouter cet élément dans le langage FCSS, via un attribut global par exemple. Étant donné que c'est le contrôleur qui se charge de la propagation et donc de griser les éléments en question, cela signifie qu'une partie du contrôleur devrait être générée en fonction du comportement que le concepteur souhaite pour la propagation.

Toujours concernant cette propagation, le contrôleur devrait avoir un comportement différent en fonction du type de contrainte qui a provoqué le changement de valeur de certains éléments.

En effet, comme nous l'avons vu dans le Chapitre 4, un FM possède trois types de contraintes : les contraintes de descendance, les contraintes de groupe et les contraintes "cross-tree". Les premières signifient que si une *feature* est sélectionnée, sa parente et ses *features* ascendantes doivent aussi l'être. Les secondes résultent des cardinalités des groupes de *features*. Les dernières sont définies explicitement et impliquent des *features* qui ne sont pas forcément dans le même sous-arbre. Seulement, ce n'est que dans ce dernier cas que le contrôleur devrait griser les "widgets" représentant les *features* propagées.

Pour les contraintes de descendance, le contrôleur grise la *feature* propagée parente de celle sélectionnée alors qu'il ne devrait pas. Néanmoins, il doit tout de même sélectionner la *feature* parente puisqu'elle fait partie des *features* propagées retournées par l'API. Il ne doit simplement pas griser le "widget" étant donné que l'utilisateur peut toujours décider plus tard de ne pas choisir cette *feature* parente (et donc les *features* filles également) et de sélectionner une autre. Lorsque le contrôleur grise le "widget", il empêche l'utilisateur de revenir sur son choix.

Pour les contraintes de groupe, le contrôleur fonctionne correctement, il ne grise pas les *features* appartenant au même groupe de *features* que la *feature* sélectionnée. En effet, si deux *features* "A" et "B" font partie d'un groupe *oneof*, il n'y a aucune raison de griser le "widget" représentant la *feature* "B" si l'utilisateur a sélectionné la *feature* "A". Il est tout à fait possible que l'utilisateur revienne sur son choix. Seulement, il est possible qu'une contrainte "cross-tree"

implique des *features* du même groupe. Dans ce cas, le contrôleur devrait griser ces *features* car ce n'est plus uniquement la contrainte de groupe qui joue mais également la contrainte "cross-tree".

Cependant, avec la version actuelle de l'API, il n'y a aucun moyen de savoir si une *feature* doit être propagée en conséquence d'une contrainte de descendance, de groupe ou "cross-tree". Il faudrait donc modifier l'API mais également la servlet pour qu'elles ajoutent cette information qui se révélerait utile pour le contrôleur, afin d'affiner son fonctionnement.

En attendant, deux solutions temporaires seraient envisageables. La première consiste simplement à remonter la hiérarchie des balises HTML à partir de la balise représentant la *feature* sélectionnée. Cela permettrait de voir si une *feature* est la *feature* parente d'une autre. Pour une *feature* du même groupe, il faudrait, une fois la *feature* parente retrouvée, redescendre la hiérarchie pour retrouver les balises se trouvant au même niveau que celle représentant la *feature* sélectionnée. La deuxième solution implique que l'API gère complètement les *long-id*. En effet, le contrôleur retrouve le "widget" représentant la *feature* à propager grâce à l'attribut "id" de la balise qui correspond au *long-id* de la *feature* représentée. Pour savoir si la *feature* à propager est bien la *feature* parente de la *feature* sélectionnée, il doit pouvoir comparer le *long-id* de la *feature* parente avec le *long-id* de la *feature* sélectionnée tronquée de son *short-id*. Pour prendre un exemple, la *feature* "A.B.C" a pour parente la *feature* "A.B". Pour les *features* d'un même groupe, le raisonnement est similaire puisqu'elles ont la même *feature* parente. Ainsi, les *features* "A.B.C" et "A.B.D" appartiennent au même groupe contenu dans la *feature* "A.B". Au final, les deux solutions se rejoignent mais la seconde implique moins de changements au niveau du code du contrôleur puisqu'il suffit de retrouver directement le bon "widget" à partir de l'attribut "id" plutôt que de naviguer dans la hiérarchie des balises jusqu'à retrouver celle représentant la *feature* parente ou une autre *feature* du groupe.

Le contrôleur doit donc être amélioré avec les ajouts qui seront nécessaires lorsque l'API sera corrigée.

Enfin, nous pouvons également relever quelques points moins importants à l'heure actuelle ou, en tout cas, qui n'entraient pas dans le cadre de ce travail :

1. Aucun mécanisme d'envoi des formulaires n'est encore prévu. Au niveau du générateur, il faudrait au moins générer un bouton pour pouvoir envoyer ce formulaire. Au niveau de l'envoi en lui-même, il pourrait sans doute être effectué par le contrôleur via une fonction commune requérant une légère personnalisation, par exemple, pour l'adresse à laquelle envoyer le formulaire.
2. Si nous comparons les configurateurs générés avec certains configurateurs actuels de voitures par exemple, nous remarquons que ceux-ci sont assez éloignés visuellement. Les seconds sont en général très dynamiques avec des animations intégrées, une visualisation possible du résultat en direct (par exemple, pour la couleur), etc. À ce niveau, les configurateurs générés via les modèles TVL et FCSS peuvent encore être largement améliorés. Cependant, il est très difficile d'avoir un processus complètement automa-

tisable et des peaufinements manuels seront sans doute souvent requis, surtout si le concepteur souhaite un résultat visuel "impressionnant".

3. La question des demandes de plusieurs utilisateurs en parallèle pour un configurateur basé sur un FM n'a pas été posée dans ce mémoire. A. Hubaux propose une solution dans [53] qui aborde la configuration collaborative basée sur les *features*.

12.2 Améliorations possibles

La première amélioration à apporter au niveau du générateur est, logiquement, la gestion des différentes constructions TVL et FCSS qui ne sont actuellement pas gérées. Cela implique quelques modifications de *templates* et l'ajout d'autres mais également des modifications au niveau du contrôleur. Néanmoins, ces changements restent assez mineurs et ne devraient pas demander beaucoup de temps.

En ce qui concerne le placement des différents "widgets" et la structuration de l'interface, un langage de vue, permettant notamment de définir l'emplacement des éléments devra être défini. Il permettrait de définir éventuellement des menus, des onglets, des panels, plusieurs pages, etc. Comme il est spécifié dans la Section 6.1 sur les *UIDLs*, une *AUI* pourrait faire office de modèle de vues. Une autre option qui se présente pour les membres de l'équipe PReCISE est de définir leur propre *AUI*, un *UIDL* tel qu'*UsiXML* étant beaucoup plus complexe que ce qui est nécessaire dans le cadre du projet. D'un autre côté, cela équivaudrait à réinventer la roue. Ces deux opportunités présentent donc chacune leurs avantages et leurs inconvénients. Il sera donc sans doute utile de faire des recherches supplémentaires avant de se décider.

Ensuite, comme nous l'avons remarqué dans la Section 5.2, le langage FCSS est encore en cours de développement et va devoir être étoffé pour traiter de nouveaux éléments. La liste suivante contient une série de propositions afin d'enrichir le langage :

1. Il pourrait éventuellement permettre de définir des couleurs pour les éléments TVL. Celles-ci pourraient être appliquées aux labels ou aux *fieldsets* de *features* et attributs. Toutefois, le langage CSS permet, dans le cadre de configurateurs HTML, de faire cela.
2. Au niveau des attributs entiers et réels, l'ajout du *slider* comme "widget" de représentation serait également un plus. Le générateur possède d'ailleurs déjà un *template* prévu à cet effet qui n'a plus qu'à être correctement intégré dans le processus de génération. Si le domaine de valeurs d'un attribut entier ou réel est limité par énumération des valeurs possibles, le "widget" à générer doit être différent, une *textbox* ne pouvant convenir, ni un *slider* d'ailleurs. Dans ces cas-là, il devrait être possible de spécifier les mêmes "widgets" que ceux utilisés pour les attributs de type énumération, c'est-à-dire, les "listbox" et "radiogroup".
3. Concernant la politique de propagation des contraintes, ce langage pourrait également la stipuler. Cela pourrait prendre la forme d'un attribut

global *constraintPropagation* par exemple, dont les valeurs pourraient être "greyed" qui serait la politique actuelle, "none" qui consisterait simplement à propager sans griser de sorte à ce que l'utilisateur puisse toujours modifier la valeur du "widget". Dans ce cas, à la validation du formulaire, les contraintes devraient être vérifiées étant donné qu'elles pourraient alors être violées. Grâce à cet attribut, si la politique par défaut ne convient pas au client, elle pourrait être personnalisée. On pourrait même éventuellement faire en sorte que la politique de propagation soit différente en fonction du type de "widget" représentant les éléments à propager. Par exemple, un groupe de *features* représenté par un groupe de "checkbox" ne devrait pas être grisé mais une "listbox" représentant un groupe de *features oneof* devrait l'être. Une autre valeur possible serait "help" qui consisterait à ajouter un texte d'aide pour que l'utilisateur sache que la sélection d'un tel élément l'oblige à également sélectionner un autre. Le texte d'aide pourrait également être combiné avec la valeur "greyed" par exemple. À ce moment là, un simple attribut global serait sans doute insuffisant. Un tel attribut FCSS pourrait éventuellement être appliqué à chaque contrainte individuellement mais il faudrait alors que le contrôleur soit au courant des contraintes et connaisse, pour chaque contrainte, la technique de propagation souhaitée. D'autres comportements sont sans doute envisageables, cela requiert d'analyser ce qui se fait actuellement en matière de configurateur à ce niveau.

4. Pour les groupes de type *card*, il faudrait ajouter la valeur "listbox" dans les valeurs possibles de l'attribut global *cardGroup*, comme pour les groupes de type *someof*.

5. Ajouter la valeur "radiogroup" pour l'attribut global correspondant aux attributs TVL booléens.

6. Pour les *features* optionnelles, dans le cas où une telle *feature* ne possède qu'un groupe représenté par une "listbox", le concepteur pourrait demander à ce que cela soit représenté par une valeur vide ou "none" dans la "listbox" comme c'est le cas du "widget" représentant la *feature IMRT* du modèle TVL *Tune* à la Figure 11.1. Les attributs *optFeature* et *opt* pourraient donc prendre une nouvelle valeur qui indiquerait cette représentation. Le générateur pourrait également avoir un comportement par défaut qui prendrait en compte cette situation mais ajouter cette possibilité au FCSS augmenterait le degré de paramétrisation de la génération.

7. Dans les attributs spécifiques communs aux *features* et attributs, nous pourrions ajouter un attribut permettant de choisir où placer le "widget" correspondant à l'élément par rapport à son label (avant ou après). Dans la version actuelle, le "widget" est placé après le label pour les attributs mais avant pour les *features*.

8. Ajouter un attribut *container* spécifique à une *feature* qui permettrait de choisir de générer un *fieldset* encadrant le contenu ou pas. En effet, même si une *feature* n'est pas vide, il est possible qu'elle ne possède qu'un

seul élément. Dans ce cas, il n'est peut-être pas utile que son contenu soit encadré par un *fieldset*.

9. Ajouter un attribut global et un spécifique indiquant si les "radios" et les "checkbox" d'un même groupe doivent se trouver les unes en dessous des autres ou les unes à côté des autres. En fait, il devrait y avoir deux attributs globaux, un pour les "radios" et un pour les "checkbox" et un attribut spécifique commun aux *features* et attributs puisque cela pourrait s'appliquer aussi bien aux groupes de *features* qu'aux attributs tels que les énumérations voire même les booléens ou les entiers et réels une fois que les nouveaux "widgets" seront intégrés pour ceux-ci.
10. Ajouter un attribut qui s'appliquerait aux entiers et réels indiquant le "step" que le concepteur pourrait définir au lieu de 1 et 0,1 par défaut respectivement.
11. Le langage FCSS pourrait intégrer des attributs à appliquer à une valeur d'une énumération. Ces valeurs commencent toujours par une lettre ce qui n'est pas forcémment ce que le concepteur souhaite. Par exemple, la valeur *QuinzePouces* dans Ordinateur.tvl est une valeur de l'énumération *tailleEcran* devrait être affichée "15 pouces" mais ce n'est pas possible. Un attribut *label* pour ces valeurs serait intéressant.
12. Permettre de définir une politique de sélection des *features* et de leur contenu. Par exemple, le concepteur pourrait spécifier que le configurateur oblige l'utilisateur à sélectionner d'abord la *feature* parente avant de sélectionner une *feature* fille ou de modifier la valeur d'un attribut. À ce moment-là, les "widgets" représentant les *features* filles et attributs devraient tous être grisés au départ puis dégrisés dès que l'utilisateur sélectionne la *feature* parente.

Au niveau du langage TVL, même si celui-ci est presque finalisé, un nouveau type d'attribut pourrait être ajouté. Il s'agit d'énumérations pour lesquelles l'utilisateur peut sélectionner plusieurs valeurs. Dans la dernière version de TVL, l'équipe de PReCISE a effectué cet ajout. Les remarques concernant l'API ont également été prises en compte et les attributs, notamment, sont désormais gérés.

À ce niveau de développement de l'outil, il n'y a pas de réelle prise en compte de l'utilisateur et de ses caractéristiques. Différentes règles d'interfaces homme-machine pourraient être prises en compte comme le taux d'information affichée sur une page, dans un menu (pour la diminution de la charge de travail), les groupements des informations "similaires", la séparation des informations sans rapport entre elles, etc [37]. Pour le groupement des informations similaires, la génération des *fieldset* pour les *features* permettent de grouper leur contenu et de bien le distinguer du reste mais cela ne va pas plus loin. Par exemple, nous pourrions coloriser de la même couleur des informations similaires. Les configurateurs générés de cette façon sont prévus uniquement pour les utilisateurs "normaux", sans handicap. Rien n'est prévu pour les utilisateurs malvoyants par exemple, que ce soit au niveau de la taille des labels, de la génération d'une interface avec des sons ou simplement de la compatibilité avec des navigateurs

spécifiquement prévus pour ces utilisateurs. Tout cela serait possible, au moins en partie, grâce à certains ajouts dans le langage FCSS, l'ajout du langage de vue et la gestion des nouveaux éléments par le générateur.

Enfin, sur le long terme, un des objectifs est de pouvoir générer ce même type de configurateur dans d'autres technologies telles que Java Swing, .Net, etc. Dans ce but, et afin de ne pas devoir réeffectuer tout le cheminement, l'*AUI* pourrait également servir de modèle intermédiaire, indépendant de la plateforme et qui est ensuite traduite vers une autre technologie.

13 Conclusions

Dans ce mémoire, nous avons proposé une solution aux problèmes rencontrés avec les configurateurs actuels. Pour rappel, bien qu'ayant de nombreux avantages, ceux-ci ne sont pas toujours correctement implémentés. Ainsi, le mélange des règles de configurations avec le code de l'interface graphique est une mauvaise pratique qu'adoptent souvent les développeurs. En conséquence, des problèmes d'exactitude, de flexibilité et de maintenance apparaissent. Il devient difficile de distinguer les règles de configurations du reste du code, certaines sont oubliées ou mal implémentées. Les modifications, ajouts ou suppressions de règles sont rendues difficiles.

La solution présentée se base sur plusieurs concepts d'ingénierie du logiciel. Premièrement, il apparait nécessaire de bien séparer les préoccupations. Ainsi, la partie présentation de l'interface utilisateur du configurateur doit être séparée de la logique cachée, qui prend la forme d'un ensemble de contraintes. Nous proposons donc de créer des configurateurs dont l'architecture est de type Modèle-Vue-Contrôleur. Afin de représenter les options de configurations, nous utilisons des "Feature Models" (FMs) écrits dans le langage textuel TVL. Ceux-ci permettent non seulement de décrire les différentes options configurables sous formes de *features* et attributs, sans lien avec la technologie cible, mais également d'exprimer les contraintes liant ces éléments et qui correspondent aux règles de configurations à appliquer. Un FM va donc servir de base à la génération de l'interface graphique mais également à la vérification de la logique cachée.

Dans cette architecture, le contrôleur et la partie serveur, qui s'occupe de gérer le FM en vérifiant que la configuration du produit fasse partie, à tout instant, des configurations admises par ce dernier, sont génériques. Chaque interface graphique générée à partir d'un modèle TVL quelconque sera compatible avec ces deux composants.

En ce qui concerne la présentation, celle-ci est obtenue grâce à un générateur. Pour cela, nous avons fait appel aux concepts de l'ingénierie dirigée par les modèles et la transformation de modèles. Dans notre cas, après des recherches effectuées dans le domaine, c'est finalement une transformation Model-to-Text qui a été choisie, c'est-à-dire une transformation qui, à partir d'un ou plusieurs modèle(s) va générer du texte, plus précisémment du code source dans notre cas. Deux modèles sont fournis au générateur : le modèle TVL et un modèle FCSS correspondant. Le second a pour but de rendre la génération plus flexible et produire des interfaces graphiques plus jolies et "user-friendly". En sortie, l'outil produit du code HTML5, futur standard W3C en matière de pages Web. Du fait du choix de cette technologie, nous nous sommes donc restreint à la production de configurateurs Web, ce qui ne constitue pas réellement une importante restriction étant donné la grande quantité de configurateurs disponibles sur le Web.

Nous avons prouvé, via deux études de cas, que la solution était viable même si certaines limitations sont apparentes.

L'objectif était de gérer les différentes constructions des modèles TVL et FCSS. Celui-ci est en majeure partie rempli même si quelques constructions manquent à l'appel ou ne sont gérées que dans certaines situations. Par exemple, nous avons vu que l'attribut "widget" du modèle FCSS était bien appliqué aux groupes de *features* vides mais pas aux groupes de *features* non vides.

D'autres limitations sont également présentes. Nous pouvons notamment citer le fait qu'il n'est pas possible de structurer les éléments générés comme le concepteur le souhaiterait. Cela n'est néanmoins pas une surprise étant donné que cela ne faisait pas partie de l'objectif initial. Un troisième langage, basé sur le concept des AUIs vus dans la Section 6.1, devra être intégré à la génération afin de gérer le placement des éléments et ainsi la rendre encore plus flexible et améliorer le résultat.

L'API de configuration, pas encore au point lorsque ce projet a été implémenté, est désormais corrigée. Le contrôleur doit dès lors subir également quelques modifications afin de propager les changements comme il se doit.

Au final, bien que cette version remplisse l'objectif initial, elle reste perfectible et peut être améliorée. D'autres objectifs viendront s'ajouter aux versions suivantes, notamment la gestion du placement des éléments sur l'interface graphique. Pour l'instant, la génération en cascade produit des configurateurs basiques de type formulaire mais la possibilité de placer les éléments où le concepteur le souhaite permettra de créer des configurateurs plus complexes visuellements comme ceux divisés en vues.

Bibliographie

[1] http://en.wikipedia.org/wiki/Domain_engineering. Last Time Consulted : 24/08/2013.

[2] http://en.wikipedia.org/wiki/Feature_model. Last Time Consulted : 16/08/2013.

[3] http://en.wikipedia.org/wiki/Meta-Object_Facility. Last Time Consulted : 13/08/2013.

[4] http://en.wikipedia.org/wiki/Software_product_line. Last Time Consulted : 12/08/2013.

[5] http://gitk.sourceforge.net/overview.html. Last Time Consulted : 20/07/2013.

[6] http://help.adobe.com/en_US/flex/using/WS2db454920e96a9e51e63e3d11c0bf5f39f-7fff.html. Last Time Consulted : 18/08/2013.

[7] http://html5test.com/results/desktop.html. Last Time Consulted : 26/07/2013.

[8] http://jamda.sourceforge.net/. Last Time Consulted : 11/08/2013.

[9] http://projects.eclipse.org/projects/modeling.mmt.qvt-oml. Last Time Consulted : 13/08/2013.

[10] http://projects.eclipse.org/projects/modeling.mmt.qvtd. Last Time Consulted : 13/08/2013.

[11] http://research.edm.uhasselt.be/~kris/research/projects/StoryBoardML/usixml.html. Last Time Consulted : 18/08/2013.

[12] https://developer.mozilla.org/en-US/docs/XUL. Last Time Consulted : 20/07/2013.

[13] https://developer.mozilla.org/fr/docs/Web/Guide/HTML/HTML5. Last Time Consulted : 20/07/2013.

[14] https://distrinet.cs.kuleuven.be/projects/SEESCOA/. Last Time Consulted : 21/07/2013.

[15] http://shrike.depaul.edu/~pjohnso2/abstract.htm. Last Time Consulted : 19/08/2013.

[16] https://www.oasis-open.org/. Last Time Consulted : 17/08/2013.

[17] http://wiki.eclipse.org/Ecore. Last Time Consulted : 13/08/2013.

[18] http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial. Last Time Consulted : 13/08/2013.

[19] http://www.allconfigurators.com/. Last Time Consulted : 11/08/2013.

[20] http://www.cetic.be/Ligne-de-produits-logiciels. Last Time Consulted : 16/07/2013.

[21] http://www.configurator-database.com. Last Time Consulted : 20/03/2013.

[22] http://www.eclipse.org/acceleo/. Last Time Consulted : 09/08/2013.

[23] http://www.eclipse.org/modeling/m2t/?project=jet#jet. Last Time Consulted : 10/08/2013.

[24] http://www.eclipse.org/Xtext/. Last Time Consulted : 28/07/2013.

[25] http://www.ibm.com/developerworks/webservices/library/ws-adapt/index.html. Last Time Consulted : 19/08/2013.

[26] http://www.jimywoo.fr/solutions-web/configurateur. Last Time Consulted : 11/08/2013.

[27] http://www.journaldunet.com/developpeur/client-web/firefox-pour-android-abandon-de-l-interface-xul-1011.shtml. Last Time Consulted : 24/08/2013.

[28] http://www.w3.org/. Last Time Consulted : 20/07/2013.

[29] http://www.w3.org/community/xformsusers/wiki/XForms_Implementations. Last Time Consulted : 18/08/2013.

[30] http://www.w3.org/TR/InkML/. Last Time Consulted : 18/08/2013.

[31] http://www.w3.org/TR/voicexml21/. Last Time Consulted : 18/08/2013.

[32] http://www.w3.org/wiki/MBUI_Submissions. Last Time Consulted : 16/08/2013.

[33] http://www.w3schools.com/html/html5_intro.asp. Last Time Consulted : 20/07/2013.

[34] http://ximl.com/. Last Time Consulted : 18/08/2013.

[35] Mir Farooq Ali, Manuel A. Pérez-Quiñones, Eric Shell, and Marc Abrams. Building multi-platform user interfaces with uiml. *CoRR*, cs.HC/0111024, 2001.

[36] Michal Antkiewicz and Krzysztof Czarnecki. FeaturePlugin : Feature Modeling Plug-in for Eclipse. In *Proceedings of the Workshop on Eclipse Technology eXchange at OOPSLA'04*, pages 67–72, 2004.

[37] J.M. Christian Bastien and Dominique L. Scapin. Ergonomic criteria for the evaluation of human-computer interfaces. Technical Report RT-0156, INRIA, June 1993.

[38] Danilo Beuche. Modeling and building product lines with pure : :variants. In *Proceedings of the 17th International Software Product Line Conference co-located workshops*, SPLC '13 Workshops, pages 147–149, New York, NY, USA, 2013. ACM.

[39] Quentin Boucher, Ebrahim Khalil Abbasi, Arnaud Hubaux, Gilles Perrouin, Mathieu Acher, and Patrick Heymans. Towards More Reliable Configurators : A Re-engineering Perspective. In *International Workshop on*

Product Line Approaches in Software Engineering (PLEASE), pages 29–32, Zurich, Suisse, June 2012. IEEE.

[40] Quentin Boucher, Gilles Perrouin, Mathieu Acher, and Patrick Heymans. Engineering configuration graphical user interfaces : A model-based perspective. Technical report, PReCISE Research Center, University of Namur, Belgium, 2012.

[41] Quentin Boucher, Gilles Perrouin, and Patrick Heymans. Deriving configuration interfaces from feature models : a vision paper. In *VaMoS*, pages 37–44, 2012.

[42] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computer*, 15(3) :289–308, 2003.

[43] Andreas Classen, Quentin Boucher, Paul Faber, and Patrick Heymans. The TVL specification. Technical Report P-CS-TR SPLBT-00000003, PReCISE Research Center, University of Namur, Namur, Belgium, 2010.

[44] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling : Syntax and semantics of tvl. *Sci. Comput. Program.*, 76(12) :1130–1143, 2011.

[45] Benoit Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle*. Thèse de doctorat, Institut National Polytechnique de Toulouse, Toulouse, France, juillet 2008.

[46] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[47] Lirisnei Gomes de Sousa and Jair C Leite. 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, pages 191–200, New York, NY, USA, 2003. ACM.

[48] International Organization for Standardization. Iso29110 - lifecycle profiles for very small entities. Standard, Geneva, Switzerland, 2011.

[49] Mark Grechanik, Don S. Batory, and Dewayne E. Perry. Design of large-scale polylingual systems. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *ICSE*, pages 357–366. IEEE Computer Society, 2004.

[50] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004.

[51] Naji Habra. Ingénierie du logiciel. University Lecture, 2012.

[52] Gorel Hedin, Lennart Ohlsson, and John McKenna. Product configuration using object oriented grammars. In *8th International Symposium on System Configuration Management (SCM-8)*, Brussels, July 1998. (co-located with ECOOP'98).

[53] A. Hubaux. *Feature-based Configuration : Collaborative, Dependable, and Controlled*. PhD thesis, University of Namur, Belgium, 2012.

[54] Paul D. Johnson and Jinesh Parekh. Multiple device markup language a rule approach. Technical report, DePaul University, 2003.

[55] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl : A model transformation tool. *Sci. Comput. Program.*, 72(1-2) :31–39, 2008.

[56] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[57] Kouichi Katsurada, Yusaku Nakamura, Hirobumi Yamada, and Tsuneo Nitta. Xisl : a language for describing multimodal interaction scenarios. In *Proceedings of the 5th international conference on Multimodal interfaces*, ICMI '03, pages 281–284, New York, NY, USA, 2003. ACM.

[58] Stefan Kost. Dynamically generated multi-modal application interfaces, 2004.

[59] Quentin Limbourg and Jean Vanderdonckt. Usixml : A user interface description language supporting multiple levels of independence. In *ICWE Workshops*, pages 325–338, 2004.

[60] Kris Luyten, Tom Van Laerhoven, Karin Coninx, and Frank Van Reeth. Runtime transformations for modal independent user interface migration. *Interacting with Computers*, 15(3) :329–347, 2003.

[61] Andreas Metzger. A systematic look at model transformations. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-Driven Software Development*, chapter 2, pages 19–33. Springer-Verlag, Berlin/Heidelberg, 2005.

[62] Andreas Müller, Peter Forbrig, and Clemens H. Cap. 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, pages 16–27, London, UK, UK, 2001. Springer-Verlag.

[63] Wolfgang Müller, Robbie Schäfer, and Steffen Bleul. Interactive multimodal user interfaces for mobile devices. In *Tagungsband der HICCS-37, Waikoloa, HI, USA*, Los Alamitos, USA, November 2004. IEEE CS Press.

[64] OASIS. User interface markup language (uiml) specification, March 2004.

[65] OMG. MOF Model to Text Transformation Language, Version 1.0, January 2008.

[66] OMG. Business Process Model and Notation (BPMN), Version 2.0, January 2011.

[67] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011.

[68] OMG. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, August 2011.

[69] OMG. OMG MOF 2 XMI Mapping Specification, Version 2.4.1, August 2011.

[70] OMG. OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1, August 2011.

[71] OMG. OMG Unified Modeling Language (UML), Superstructure, Version 2.4.1, August 2011.

[72] OMG. OMG Object Constraint Language (OCL), Version 2.3.1, January 2012.

[73] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. Maria : A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *Acm Transactions on Computer-Human Interaction*, 16(4) :219–224, 2009.

[74] E. Picard, Jérémy Fierstone, Anne-Marie Pinna-Déry, and Michel Riveill. Atelier de composition d'ihm et évaluation du modèle de composants. Technical Report Livrable 3, RNTL ASPECT, May 2003.

[75] Andreas Pleuss and Goetz Botterweck. Visualization of variability and configuration options. *STTT*, 14(5) :497–510, 2012.

[76] Andreas Pleuss, Goetz Botterweck, and Deepak Dhungana. Integrating automated product derivation and individual user interface design. In David Benavides, Don S. Batory, and Paul Grünbacher, editors, *VaMoS*, volume 37 of *ICB-Research Report*, pages 69–76. Universität Duisburg-Essen, 2010.

[77] Angel R. Puerta and Jacob Eisenstein. Ximl : a common representation for interaction data. In *IUI*, pages 216–217, 2002.

[78] Trygve Reenskaug. Models-views-controllers. Xerox PARC, 1979.

[79] Max Schlee and Jean Vanderdonckt. Generative programming of graphical user interfaces. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '04, pages 403–406, New York, NY, USA, 2004. ACM.

[80] Peter Pin shan Chen. The entity-relationship model : Toward a unified view of data. *ACM Transactions on Database Systems*, 1 :9–36, 1976.

[81] Nathalie Souchon and Jean Vanderdonckt. A review of xml-compliant user interface description languages. In *DSV-IS*, pages 377–391, 2003.

[82] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2 edition, 2009.

[83] The Standish Group. Chaos report, 1995. `http://www.cs.nmt.edu/~cs328/reading/Standish.pdf`.

[84] Philippe Thiran. Web technologies. University Lecture, 2013.

[85] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide : An extensible framework for feature-oriented software development. *Science of Computer Programming*, (0) :–, 2012.

[86] Jean Vanderdonckt, Josefina Guerrero Garcia, and Juan Manuel Gonzáles Calleros. State of the art of user interface description languages. Technical report, Université catholique de Louvain, 2011.

[87] W3C. XForms, Version 1.1, October 2009.