Un profil UML pour la modélisation multiniveau

Frédéric Mallet, Charles André, François Lagarde

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Un profil UML pour la modélisation multiniveau

Frédéric Mallet — Charles André — François Lagarde

N° 7287

Mai 2000

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--7287--FR+ENG

Un profil UML pour la modélisation multiniveau

Frédéric Mallet [*] , Charles André [*] , François Lagarde [†]

Thème COM — Systèmes communicants
Projet Aoste

Rapport de recherche n° 7287 — Mai 2000 — 33 pages

Résumé : Construire un profil UML peut s'avérer fastidieux et sujet à erreurs. La nécessité de commencer par un modéle métier, indépendant d'une technologie d'implantation, est généralement reconnue. Malgré tout, l'adéquation de l'implantation doit étre vérifiée. Nous proposons un procédé automatique pour transformer le modèle métier en un profil UML. L'utilisation du *paradigme multiniveau* réduit la *complexité accidentelle* tout en produisant un profil fidèle au modèle métier. L'intérêt de ce paradigme pour la définition de profils est évalué et son utilisation est illustrée sur un extrait du modèle de temps de Marte récemment adopté par l'OMG.

Mots-clés : modélisation multiniveau, profil UML, MARTE

Ce rapport a été publié dans la revue TSI –29/2010. Ingénierie dirigée par les modèles - pp. 391–419 http://tsi.revuesonline.com

L'origine de ces travaux se trouve dans la thèse de doctorat de François Lagarde [13], financée par le CEA-List, sous la direction conjointe de François Terrier et Charles André.

[*] Université Nice Sophia Antipolis
[†] SAGEM R& D

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

A UML profile for multilevel modeling

Abstract: Building a UML profile is tedious and error-prone. There is no precise methodology to guide the process. Best practices recommend gathering concepts in a technology-independent *domain view* before implementation. Still, the adequacy of the implementation should be verified. This paper proposes to transform automatically a domain model into a profile-based implementation. To reduce *accidental complexity* in the domain model and fully benefit from advanced profiling features in the generated profile, our process relies on the *multilevel paradigm*. The value of this paradigm for the definition of UML profiles is assessed and applied to a subset of the Marte time model.

Key-words: multilevel modeling, UML profile, MARTE

1 Introduction

Construire un système informatique passe généralement (ou du moins devrait passer) par la définition d'une description neutre du problème, *i.e.* la construction d'un *modèle de domaine* ou *modèle métier*. Cela est recommandé aussi bien pour les approches traditionnelles que pour les approches dirigées par les modèles. En effet, une description neutre, indépendante de la technologie d'implantation, permet une participation active des experts du domaine même s'ils ne sont pas informaticiens ou familiers avec la technologie utilisée pour construire la solution. En revanche, si la technologie utilisée pour spécifier le modèle métier est trop éloignée de celle utilisée pour la réalisation, il se pose alors le problème non trivial de garantir que l'implantation réalise effectivement la spécification.

UML (*Unified Modeling Language*) [19] avec ses extensions telles que SysML [24, 20] est une technologie de modélisation souvent retenue pour construire le modèle métier. UML propose un ensemble varié (peut-être même trop varié) de constructions qui couvrent beaucoup d'aspects d'un projet, de la spécification à l'implantation et au déploiement du système. Il permet à la fois une description des aspects structurels aussi bien que fonctionnels/comportementaux.

Pour qu'un langage soit efficace pour la construction du modèle métier, il est primordial que les primitives offertes capturent le plus naturellement possible les idées de l'expert du domaine. Les contraintes techniques intrinsèques au langage ne doivent donc pas altérer les concepts. Or, UML s'appuie sur le mécanisme d'instanciation du paradigme orienté objet et aborde avec difficulté la modélisation d'un concept qui nécessite plus d'un niveau [3, 4]. Les langages objets qui renoncent à la dualité classe/objet se basent, par exemple, sur des prototypes [5, 15], ou des exemplars [14]. Ces langages abandonnent le typage fort au profit d'une programmation dynamique et nécessitent un système de contraintes pour assurer les relations de sous-typage.

Nous nous plaçons résolument dans une approche par typage fort et nous nous proposons de pallier le manque d'UML dans le domaine de la modélisation multiniveau. À notre connaissance, il n'existe qu'une implantation du paradigme multiniveau pour le langage Java [12] et une pour Nivel [2]. Des discussions sont en cours sur la pertinence d'une modification du métamodèle d'UML mais aucune extension légère d'UML ne semble avoir été envisagée pour l'heure. Nous proposons d'utiliser le mécanisme de profil UML. Certes celui-ci est contestable à bien des égards comme son histoire à travers les différentes versions d'UML le montre. Il reste cependant un moyen pragmatique pour étendre simplement UML, adapter les outils de modélisation existants, déployer rapidement à grande échelle et à moindre coût les extensions proposées.

Pour illustrer notre propos et montrer l'intérêt pratique de notre profil UML pour la modélisation multiniveau, nous prenons comme cas d'étude une partie du modèle de temps de Marte [21]. Le modèle de temps de Marte (*Modeling and Analysis of Real-Time and Embedded systems*) a pour objectif d'étendre UML, qui est essentiellement atemporel, avec des mécanismes évolués pour décrire de façon précise le temps. Le modèle métier de Marte contient intrinsèquement des aspects multiniveaux qui sont à l'origine de lourdeurs dans l'implantation sous forme de profil. Ces lourdeurs peuvent sembler injustifiées et nuisent à la

compréhension du profil. La conséquence est qu'il devient alors presque impossible de vérifier l'adéquation entre le profil et le modèle métier. L'introduction explicite des multiniveaux dans le modèle métier le rend plus simple. En effet, la même information est modélisée avec moins d'éléments de modèles et chaque concept est représenté par une seule classe alors qu'il était avant dispersé dans plusieurs. De ce fait, nous considérons que la complexité originelle du modèle était *accidentelle* [6] car due au procédé de modélisation et non inhérente au modèle. De plus, nous proposons ensuite un procédé systématique pour générer une implantation à partir du modèle métier enrichi de l'information complémentaire sur les niveaux de modélisation. Ces niveaux étaient auparavant implicites dans le modèle et sont alors rendus explicites par les stéréotypes de notre profil pour la modélisation multiniveau. Dans l'esprit de Marte, qui est implanté sous forme de profil UML, notre technologie d'implantation est également basée sur le profilage.

La section 2 rappelle les bases de la modélisation multiniveau. La section 3 présente une partie du modèle de temps de Marte en insistant sur les difficultés, sources possibles de confusions, induites par les aspects multiniveaux dans l'interprétation de ce profil. Le modèle métier est alors modifié en explicitant les différents niveaux. La section 4 introduit notre profil UML pour la modélisation multiniveau. Ce profil est par la suite utilisé pour générer une nouvelle implantation du modèle de temps de façon systématique. Le résultat obtenu est alors comparé au profil officiel adopté par l'OMG (*Object Management Group*).

2 La modélisation multiniveau

De nombreux modèles contiennent des aspects multiniveaux. Il y a plusieurs solutions pour les aborder. La première famille de solutions consiste à utiliser les mécanismes objets *classiques* pour modéliser ces niveaux de façon implicite. Ce type de solution, en général inefficace, est discuté dans la section 2.1. La deuxième famille de solutions consiste à définir des mécanismes appropriés pour traiter le problème explicitement. La section 2.2 commente la solution basée sur les *PowerTypes*, la section 2.3 discute la solution basée sur les examplars et les prototypes. Enfin, la section 2.4 montre les avantages de l'utilisation des *clabjects*.

2.1 Solutions objets classiques

Les langages objets basés sur les classes regroupent dans une même structure, une classe, les propriétés communes à un ensemble d'objets. Par exemple, les propriétés qui concernent un assistant numérique personnel de type PDA (*Personal Digital Assistant*) peuvent se regrouper dans une classe PDA [16]. Ces propriétés incluent, par exemple, le nom du fabricant et la taille de l'écran (cf. Figure 1). Notons que la dépendance (flèche pointillée) entre l'instance et sa classe traduit ici une instanciation. C'est la seule utilisation des dépendances que nous faisons dans cette section, il n'y a donc pas d'ambiguïté. Notons également que l'utilisation de points d'interrogation n'est pas conforme à UML. Il s'agit ici de montrer les faiblesses de la solution envisagée en identifiant l'intention du concepteur même si le modèle ainsi construit est non conforme ou peu satisfaisant.

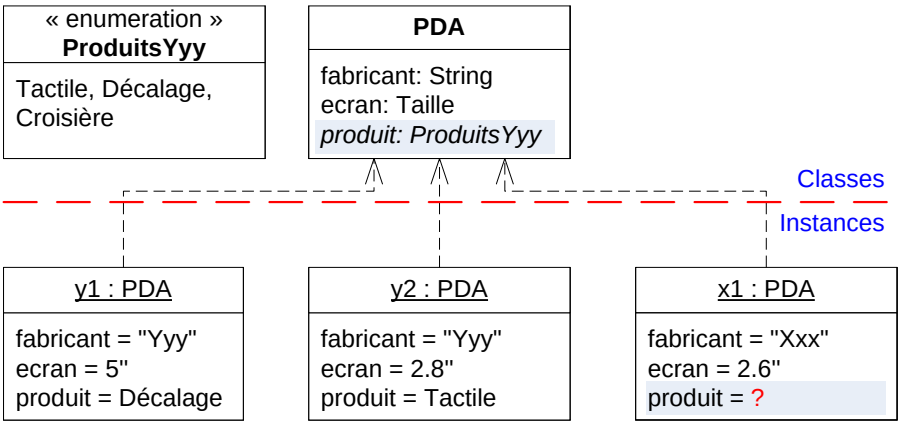

FIGURE 1 – Modèle classe/objet simple

Les propriétés sur fond grisé indiquent les points faibles du modèle que l'on cherche à combler dans les modèles suivants. Enfin, notons que les lignes en pointillés horizontales ne font pas partie du modèle et ont été ajoutées en support à la discussion. Elles séparent les classes des instances et font ainsi ressortir les deux niveaux de modélisation.

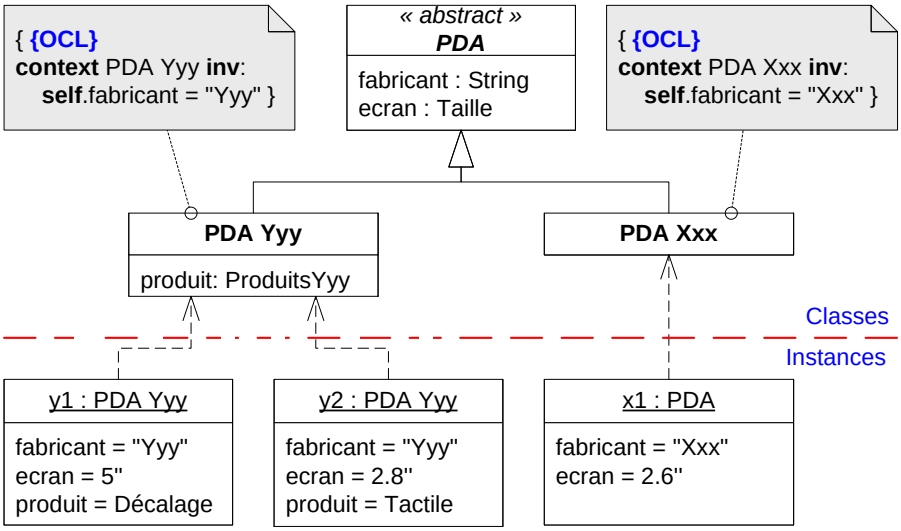

FIGURE 2 – Modèle de PDA avec généralisation

La construction d'un modèle domaine sur des produits appelle souvent à la création de *propriétés de niveau supérieur* qui concernent le modèle du produit. La *généralisation* est une façon de représenter ces niveaux pour distinguer par exemple les propriétés qui sont spécifiques à un modèle ou à un fabricant (cf. Figure 2). Dans l'exemple, on suppose que les PDAs fabriqués par Yyy possèdent une propriété produit de type énuméré qui ne se retrouve pas pour ceux d'Xxx. Cette approche par spécialisation n'est pas satisfaisante car elle doit être combinée avec une contrainte OCL (dans une note sur fond grisé) pour garantir que l'information relative au fabricant ou au modèle est cohérente entre toutes les instances du même fabricant ou du même modèle. En particulier, on doit s'assurer que tous les PDAs de Yyy contiennent effectivement la mention "Yyy" avec la même casse. En plus de requérir l'utilisation d'une contrainte, l'information du fabricant devient largement redondante et est recopiée inutilement dans toutes les instances pour être conforme à la classe abstraite PDA.

L'utilisation d'attributs de classe (statiques) dans les classes PDA Yyy et PDA Xxx limite la redondance (cf. Figure 3).

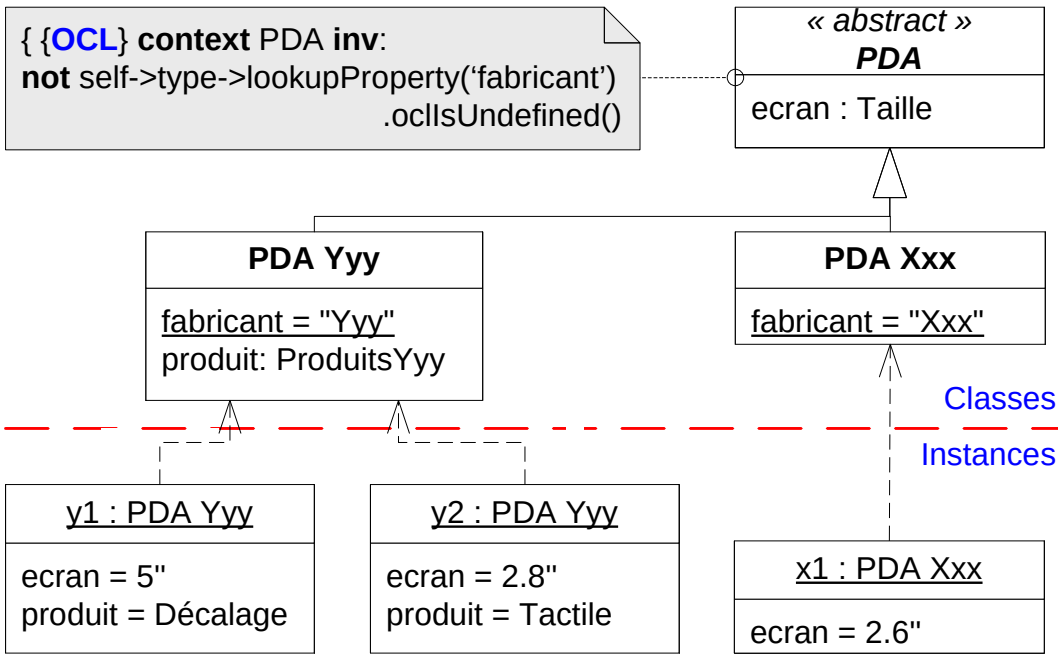

FIGURE 3 – Modèle de PDA avec attributs statiques

Cependant, il n'est plus possible de garantir que tous les PDAs aient effectivement une propriété fabricant. En admettant une capacité d'introspection pour OCL, on pourrait envisager de construire la contrainte de la Figure 3 qui établit que toutes les instances de PDA,

y compris les instances de sous-classes puisque PDA est une classe abstraite, doivent avoir une propriété nommée fabricant. Ce n'est évidemment pas la bonne façon de procéder, et les solutions proposées ensuite traitent le problème plus simplement.

2.2 L'utilisation de PowerTypes

Le patron de conception *PowerType* [18, 7, 11, 19] a été élaboré pour un tel scénario où coexistent dans le même modèle des informations sur un produit et sur le type du produit. Les *PowerTypes* ont été inspirés des *PowerSets*, l'ensemble des parties d'un ensemble. Un *PowerSet* est un ensemble a dont chaque élément est un sous-ensemble d'un autre ensemble b. De façon analogue, un *PowerType* est une classe dont chaque instance est une sous-classe d'une autre classe. L'instance et la sous-classe en question représentent la même entité bien que ce soient des éléments de modèle de niveau différent. La Figure 4 applique ce patron. L'information sur le fabricant est modélisée par le *powertype* FabricantPDA, et la classe PDA y fait référence. Cette classe est elle-même spécialisée en une sous-classe PDA Yyy qui représente spécifiquement les PDAs fabriqués par Yyy. La sous-classe PDA Xxx représente les PDAs fabriqués par Xxx.

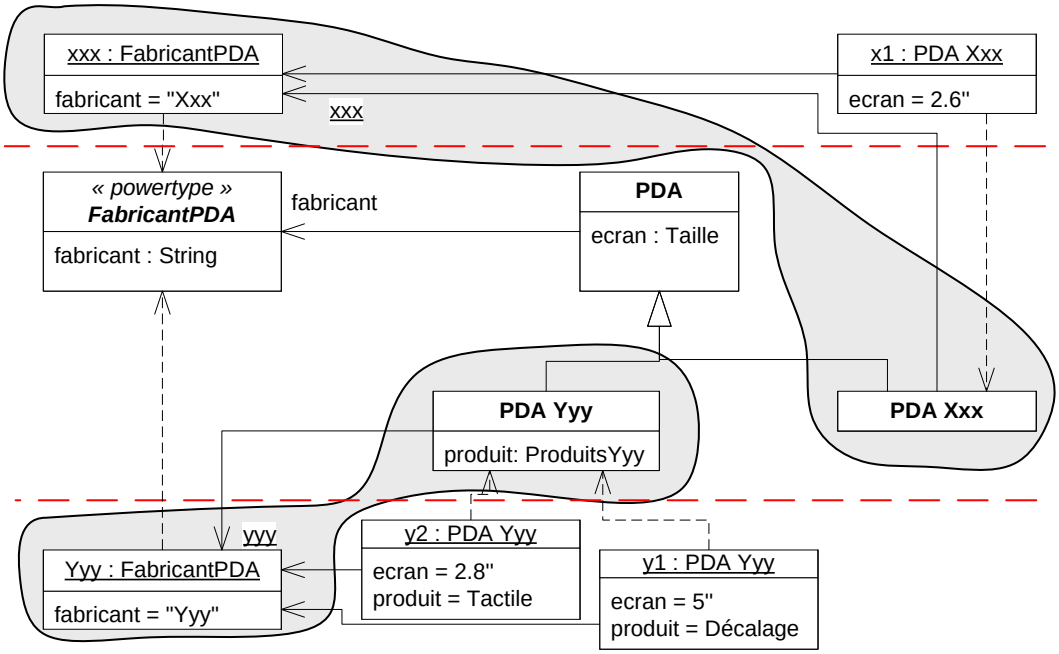

FIGURE 4 – Illustration des PowerTypes

FabricantPDA est un *PowerType* car chacune de ses instances (*e.g.* Yyy) représente la même entité qu'une sous-classe de la classe PDA (*e.g.* PDA Yyy). En fait, on peut établir une partition des PDAs en prenant comme critère le fabricant et cette partition peut être matérialisée par une spécialisation. On pourrait établir une autre partition selon d'autres critères, *e.g.* le système d'exploitation sur lequel ils tournent. Chacun des critères identifiés peut être matérialisé par un *PowerType* (*e.g.* FabricantPDA). Une instance de FabricantPDA est alors aussi le sous-ensemble fabriqué par Yyy de l'ensemble des PDAs, c'est-à-dire exactement ce que représente la classe PDA Yyy. Ce problème de classification multiple est très fréquent. Utiliser un *PowerType* est une façon de traiter ce problème. Ce modèle présente l'avantage de contenir toutes les informations et d'être relativement flexible. On peut par exemple ajouter facilement d'autres critères. La contrepartie est qu'il devient rapidement complexe et éparpille l'information dans plusieurs classes et instances. En particulier, l'information de typage sur les PDAs fabriqués par Yyy est répartie entre la classe PDA Yyy et l'instance Yyy de type FabricantPDA. Une difficulté supplémentaire vient du fait que le même objet physique est représenté par des éléments de modèle de niveaux différents (une classe et une instance).

2.3 Les examplars et les prototypes

Les partisans des prototypes [5] limitent le problème en ne considérant que le niveau des instances. Il n'y a plus d'information de typage. Le mécanisme est fondé sur la constatation qu'il est plus facile d'appréhender un objet concret et de le comparer à d'autres objets concrets que d'abstraire dans une classe les propriétés intrinsèques des objets. La Figure 5 illustre l'utilisation de ce mécanisme pour aboutir à un modèle nettement plus simple, avec peu de redondance.

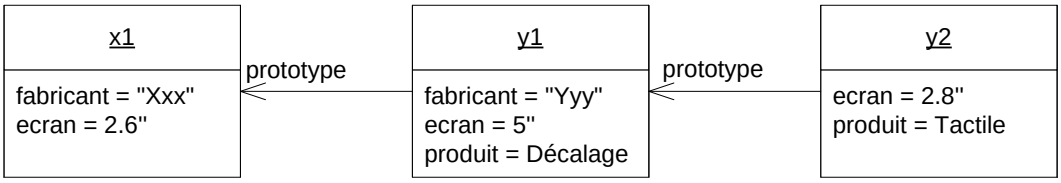

FIGURE 5 – Illustration des prototypes (examplars)

L'approche consiste à choisir un exemplaire représentatif (*e.g.* x1) et à énumérer ses propriétés. S'il existe des objets qui diffèrent de ce *prototype*, on construit alors un autre exemplaire *représentatif par différence* avec le prototype d'origine et on marque la relation de prototypage. Sur la figure, on utilise une flèche qui pointe vers le prototype. Par exemple, la flèche entre l'objet y1 et x1 montre que l'objet y1 est basé sur x1 pour l'essentiel et donc reprend ses propriétés. Il diffère cependant en changeant la valeur de propriétés existantes (*i.e.* fabricant et ecran) et en ajoutant une nouvelle propriété (*i.e.* produit). L'objet y2 lui est

plus proche du y1 que du x1 car il a le même fabricant et il a également une propriété produit, même si une valeur différente est associée à cette propriété. Dans cette approche, aucune classe n'est construite pour identifier la liste des propriétés que doivent avoir les objets en question (leur type). L'approche est très simple et très flexible, mais aucune vérification de type n'est possible. Sans vouloir entrer dans le débat typage fort ou faible, notre approche s'inscrit dans un cadre de typage fort et nous cherchons une façon élégante de modéliser ce problème avec des types.

2.4 L'utilisation de clabjects

L'approche multiniveau recommande d'aplatir les niveaux. Elle se fait par l'intermédiaire d'un *clabject* [3] qui représente indifféremment une classe ou un objet. Un clabject contient des *champs* (*fields*) qui unifient les notions de méta-attribut, d'attribut et de *slot*. Les champs sont associés à un *potentiel* (*potency*), un entier qui représente le nombre d'instanciations nécessaires pour arriver à une valeur. A chaque instanciation, le potentiel est décrémenté. Lorsqu'il arrive à zéro, le champ devient un *slot* et une valeur est fournie. Le *potentiel* rend possible une instanciation d'ordre supérieur dans laquelle les attributs traversent plusieurs niveaux d'instanciation avant d'obtenir une valeur effective. Un champ de potentiel égal à 1 est équivalent à un attribut. Un champ de potentiel égal à 2 est équivalent à un méta-attribut. Notons que tous les champs d'une même classe n'ont pas nécessairement le même potentiel.

La Figure 6 illustre l'utilisation de ce mécanisme sur le même exemple et met en œuvre trois niveaux de modélisation. Le *clabject* PDA contient un champ fabricant garantissant, par typage, que cette propriété est commune à tous les PDAs. Il contient également un champ ecran de potentiel 2. Tous les PDAs ont des informations sur la taille de leur écran mais il faudra attendre deux instanciations pour avoir la valeur effective. Au deuxième niveau apparaissent les clabjects spécifiques à un fabricant. Le champ fabricant obtient une valeur et le potentiel du champ ecran est décrémenté. Pour les PDAs fabriqués par Yyy, un nouveau champ produit (potentiel par défaut 1) apparaît pour modéliser une information spécifique à Yyy. Au dernier niveau, tous les champs ont un potentiel zéro et obtiennent une valeur.

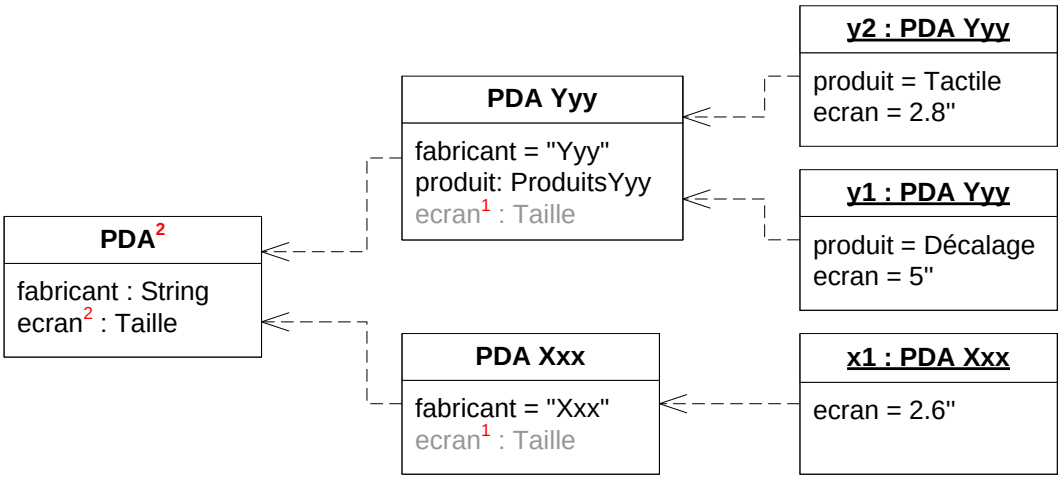

FIGURE 6 – Illustration des clabjects

La relation d'instanciation *profonde* est également représentée par une dépendance. La flèche est dirigée depuis le clabject de niveau inférieur vers le clabject de niveau immédiatement supérieur. Tous les champs présents dans le clabject supérieur et qui ont un potentiel supérieur à 0 se retrouvent dans le clabject de niveau inférieur avec une valeur de potentiel décrémentée. De nouveaux champs peuvent également être introduits, avec un potentiel quelconque. Ce modèle est à la fois flexible, compact et fidèle. On peut malgré tout regretter que le potentiel soit une constante. En faire une variable pourrait apporter plus de flexibilité et en particulier éviterait une décision prématurée sur le nombre d'instanciations requises. Par exemple, le potentiel du champ ecran pourrait être $\star$ pour signifier qu'il doit y avoir une description de l'écran mais sans préciser le nombre d'instanciations nécessaires pour obtenir une valeur.

3 Le modèle de temps de Marte

3.1 Le profil de l'OMG

La Figure 7 est un extrait du profil de temps de Marte [1]. Il ne s'agit pas ici d'étudier ce profil en détail mais plutôt de le prendre comme cas d'étude non trivial pour montrer comment l'approche présentée aurait pu faciliter sa conception. Pour cela, nous avons choisi un sous-ensemble représentatif, suffisant pour mettre en évidence les problèmes liés à la présence de plusieurs niveaux de façon inhérente au domaine. La figure contient trois parties. La partie inférieure gauche est une description simplifiée du profil de temps de Marte. La partie droite montre une partie des bibliothèques associées au profil. La partie supérieure présente les métaclasses UML sur lesquelles reposent le sous-ensemble choisi et nécessaire à la discussion. Cette dernière partie est extraite de UML::Classes::Kernel.

Les annotations (A) et (B) ainsi que les dépendances annotées (A) et (C) ne font pas parties des spécifications mais ont été ajoutées pour faciliter la compréhension. Le cœur du profil est fait de deux stéréotypes (ClockType et Clock). Ces stéréotypes fournissent des mécanismes pour créer de nouvelles horloges et rassembler dans une même structure (ClockType) les propriétés communes caractérisant plusieurs horloges. Le profil de temps, et plus généralement l'ensemble des profils de Marte, font un usage avancé des stéréotypes. En effet, les propriétés des stéréotypes ne se limitent pas à des types primitifs, mais utilisent des métaclasses et même d'autres stéréotypes.

Un premier point sensible concerne le stéréotype ClockType et son attribut resolAttr. Il sert à qualifier la résolution d'une horloge. Le type de cet attribut est Property. L'utilisation d'une métaclasse plutôt qu'un type primitif (*e.g.* Integer) offre une liberté pour le choix du type. Un lecteur averti pourrait imaginer le recours à une classe paramétrée (*template*) pour différer le choix du type. Cependant la conception du profil doit répondre au besoin d'annoter un modèle existant pour identifier une propriété particulière qui, selon l'expert du domaine, joue le rôle de résolution (qui pourrait avoir pour nom granularité ou encore quantum). De même le type de cette propriété peut être très variable et peut inclure, des nombres rationnels, des nombres entiers ou même simplement une énumération (*e.g.* fine, moyenne,

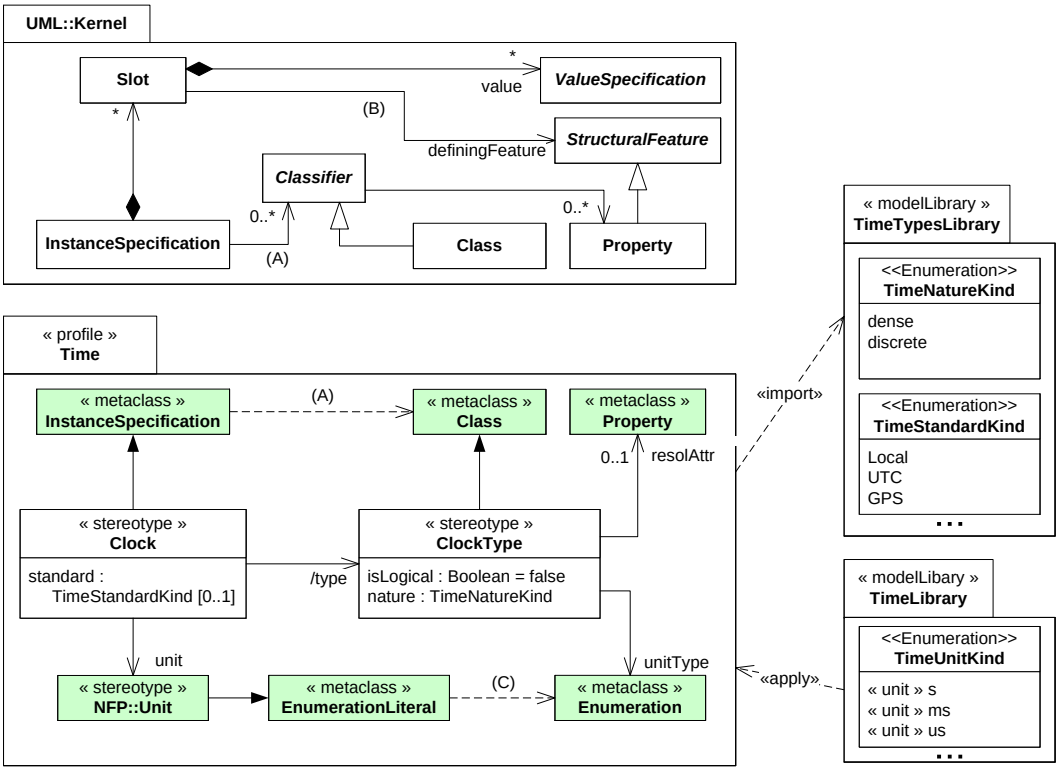

FIGURE 7 – Extrait du profil de temps de Marte

grossière). Faute d'un système ontologique plus perfectionné, l'expert de domaine a ainsi un moyen d'identifier et de marquer, avec l'attribut resolAttr, la propriété qu'il estime avoir la sémantique d'une résolution d'horloge, et ce quels que soient son type et son nom effectif. Ce mécanisme a été utilisé à plusieurs reprises dans Marte. Son utilité et son exploitation avec des outils ont été illustrées sur un exemple mettant en œuvre la transformation de modèles de systèmes d'exploitations temps réel [23].

Un autre exemple qui va servir pour notre argumentation est celui de l'attribut unit du stéréotype Clock. Cet attribut est de type NFP::unit, c'est-à-dire un autre stéréotype de Marte qui étend la métaclasse EnumerationLiteral. N'importe quel EnumerationLiteral ne convient pas pour représenter une unité mais seulement un littéral stéréotypé par NFP::unit. Ce stéréotype apporte une information concernant les facteurs de conversion vers d'autres unités. Sans cette information, il peut être difficile voire impossible d'établir un lien avec les autres unités et les autres horloges du système. Typer les attributs d'un stéréotype par d'autres stéréotypes est un mécanisme très peu utilisé, en particulier parce que parmi les outils de modélisation UML qui permettent la construction de profils peu le supportent. Lorsque ce mécanisme n'est pas supporté, il faut avoir recours à des contraintes OCL pour obtenir un effet similaire. Nous avons préféré une approche par typage fort à une approche par contraintes. Notre choix semble d'autant plus justifié que peu d'outils vérifient la conformité des contraintes OCL sur les modèles UML. Le critère essentiel reste donc la simplicité et lisibilité du modèle de domaine, indépendamment des outils disponibles pour l'implantation.

3.2 Les aspects multiniveaux

La Figure 7 fait apparaître clairement au moins deux sources de confusions entre les niveaux de modélisation : l'utilisation de bibliothèques modèle (*modelLibrary*) et l'utilisation du patron *PowerType* [18], également connu sous les noms *Item/Descripteur* [7] et *Type/Objet* [11]. Dans la suite, nous utiliserons indifféremment les trois terminologies selon le contexte.

On pourrait en effet penser que les niveaux sont clairement identifiés en fonction des éléments de modèles utilisés. Les métaclasses (*e.g.* InstanceSpecification, Class, Enumeration, EnumerationLiteral) identifient un niveau *meta*, les classes ou types primitifs identifient un niveau *modèle*. Une étude attentive du métamodèle UML montre qu'il n'en est rien. Toutes les métaclasses ne représentent pas des concepts de même niveau. Class est un type pour InstanceSpecification et Enumeration est un type pour EnumerationLiteral, donc Class et Enumeration sont de niveau supérieur. C'est un exemple classique d'utilisation du patron *Type/Objet*. Les relations (A) et (B) de la Figure 7, bien qu'étant de simples associations, représentent en fait une relation de typage. Le lien (A), entre une instance (InstanceSpecification) et son classeur (Classifier), détermine la nature du lien qu'il y a entre une horloge (Clock) et son type (ClockType). L'association entre ces deux stéréotypes et l'attribut dérivé type ne sont pas obligatoires car l'association est déjà présente dans le métamodèle UML mais leur présence éclaire l'utilisateur sur l'intention réelle des concepteurs de Marte. Il s'agit visiblement d'un artifice pour mettre en place ce patron très courant avec un langage n'offrant que deux niveaux (classes/objets).

Les mêmes contraintes ont motivé la création de l'association (B) liant indirectement la métaclasse Slot à son type Property (sous-classe de StructuralFeature). Cette association est exploitée dans Marte pour l'attribut resolAttr du stéréotype ClockType.

Ce patron apparaît une troisième fois avec le lien (C) entre un EnumerationLiteral et son type Enumeration. Notons que ce lien est implicite en UML. C'est pour cela que nous l'avons représenté par une simple dépendance en pointillés. Ce lien indique que l'unité d'une horloge (unit de type NFP::unit) est un littéral choisi nécessairement parmi la liste de littéraux définis par l'énumération unitType associée au type de l'horloge (ClockType).

Une deuxième source de confusion vient de l'utilisation de bibliothèques modèle qui échappent volontairement aux niveaux de classification. Les bibliothèques peuvent, en effet, être utilisées aussi bien dans les méta-métamodèles (comme le MOF), que les métamodèles, les profils ou les modèles utilisateurs. Leurs utilisations sont relativement faciles à comprendre lorsqu'il s'agit de types primitifs simples tels que le type Boolean, défini dans la bibliothèque standard UML. C'est également le cas pour les énumérations définies dans la bibliothèque TimeTypesLibrary de Marte. Dans certaines situations, le concepteur sait que les types définis ne peuvent pas traverser les niveaux et sont restreints à un seul niveau. Il en est ainsi pour la bibliothèque TimeLibrary dont les énumérations (*e.g.* le type TimeUnitKind) ne peuvent être utilisées qu'à un niveau inférieur au profil puisque cette bibliothèque applique le profil. Une utilisation dans le profil lui-même donnerait lieu à des cycles de définition.

3.3 L'utilisation du profil de temps

La Figure 8 présente un exemple d'application du profil de temps pour la création de deux types d'horloge. La partie gauche montre l'usage conventionnel des profils UML. Le type Chronometric représente des horloges discrètes et chronométriques, ces dernières étant liées au temps physique. Ces horloges ne sont pas parfaites et peuvent donc avoir des défauts (*e.g.* jigue, déphasage) par rapport au temps physique *idéal*. La propriété resolution de type Real caractérise un des défauts, à savoir la précision avec laquelle l'horloge est capable de mesurer le temps, c'est-à-dire sa résolution. L'utilisateur et les outils qui manipulent ce modèle sont informés du rôle joué par cette propriété grâce à l'application du stéréotype ClockType et de la valeur de l'attribut resolAttr. Le deuxième type d'horloge (Cycle) représente des horloges logiques qui mesurent le temps en nombre de cycles de processeur ou de bus. Ces horloges sont dites logiques car elles ne sont pas liées au temps physique. Pour les horloges logiques la propriété résolution n'a pas nécessairement de sens, d'où la multiplicité 0..1 de l'attribut resolution.

L'horloge chronométrique cc1 complète la spécification donnée par son type en choisissant explicitement une unité (s). Cette unité est choisie parmi les littéraux définis dans l'énumération TimeUnitKind identifiée dans Chronometric comme étant le unitType. cc1 spécifie également un standard (*e.g.* UTC) et une résolution effective (dans le *slot* réservé à resolution). De même, l'horloge p1 de type Cycle choisit une unité (ici processorCycle) et la liste des unités disponibles est donnée par l'énumération CycleUnitKind. Chaque nouveau ClockType peut associer une nouvelle liste d'unités en fonction des besoins. Les horloges de ce type en

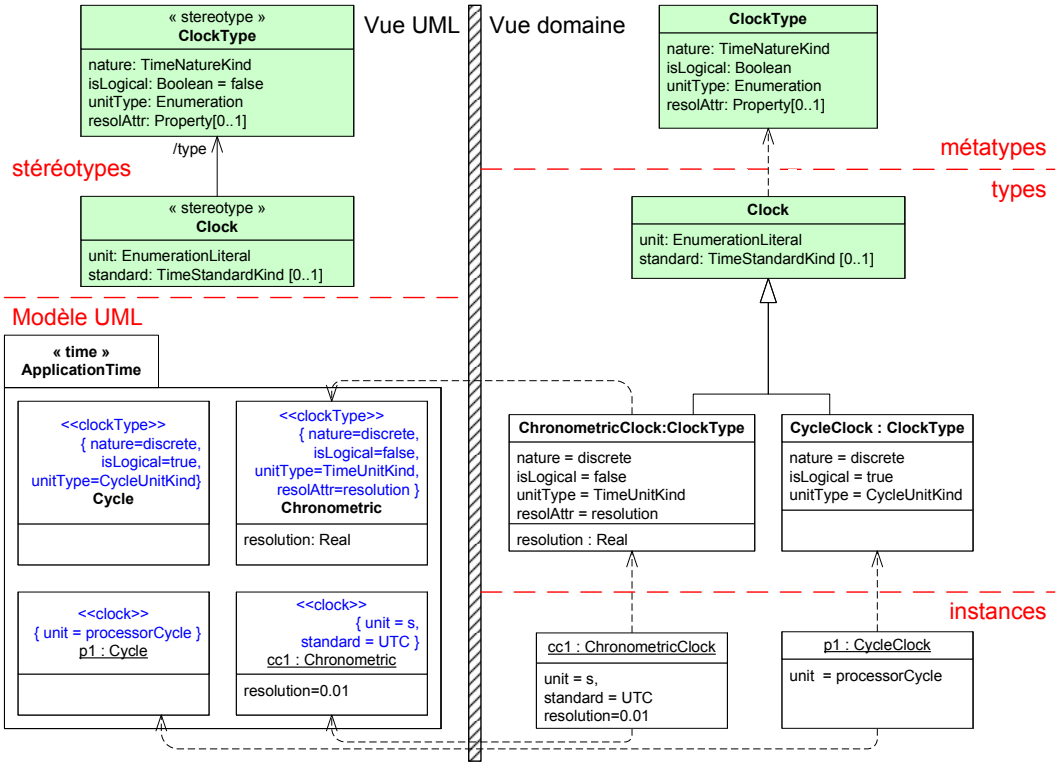

FIGURE 8 – Exemples d'horloges avec leur type

sélectionnent une dans la liste proposée. Les horloges de même type doivent nécessairement utiliser des unités compatibles.

La partie de droite est une représentation conceptuelle qui met en évidence les niveaux de modélisation effectifs. La notation utilisée est inspirée de la notation des *clabjects* [3]. Sur ce modèle le potentiel n'est pas explicite. En revanche, certains éléments de modèle, comme ChronometricClock et CycleClock contiennent deux compartiments pour différencier les champs de potentiel 0, qui portent une valeur, des champs de potentiel 1 qui sont de simples déclarations. La notation avec les deux-points précise la classe qui déclare les champs de potentiel 0 (ici ClockType). La différence majeure avec des attributs statiques est que le typage garantit ici la présence des champs nature, isLogical et unitType.

Dans la vue UML, les horloges et leurs types sont représentés au même niveau, comme des stéréotypes. Cependant, ClockType est un *descripteur* pour un ensemble de Clock (selon le patron *Item/Descripteur*). Les deux concepts appartiennent donc à des niveaux logiques différents. Sur la vue domaine, les niveaux logiques sont mis en évidence par la relation instance/type (flèche verticale en pointillés).

Cette stratégie de conception découlant du patron Type/Objet conduit à un éparpillement de l'information sur les horloges. Une partie de l'information est donnée dans la classe, une autre dans les méta-attributs de stéréotypes qui s'appliquent soit à une classe soit à une instance. Le reste (*e.g.* la valeur de la résolution) est donné dans les *slots* des instances. L'information est progressivement raffinée de la classe vers les instances. C'est vrai en particulier pour la résolution et l'unité. D'autres propriétés ne caractérisent qu'un seul niveau soit le niveau classe, soit le niveau instance. Au niveau classe, c'est le cas de l'attribut nature et au niveau instance, c'est le cas de l'attribut standard.

La complexité de ce mécanisme est essentiellement due aux limitations d'UML qui ne propose qu'une modélisation sur deux niveaux. Il s'accompagne d'une complexité qui peut être considérée comme *accidentelle* [6] et qui aurait pu être réduite en utilisant des mécanismes de modélisation multiniveau.

4 Un profil pour la modélisation multiniveau

4.1 Les principes

Cette section présente notre implantation des mécanismes pour supporter une modélisation multiniveau. Son application est illustrée sur le modèle de temps de Marte. Cependant, la démarche est générale et peut être utilisée de façon systématique pour engendrer une implantation dans UML d'un modèle de domaine quelconque. L'approche s'appuie sur un processus de génération à trois étapes. La première étape est la spécification du modèle métier (cf. section 4.3). Elle se fait par un profil UML dédié pour annoter le domaine avec les informations liées aux niveaux. Cela consiste essentiellement à caractériser les champs par un potentiel. Au cours de la seconde étape (cf. section 4.4), une implantation du domaine est engendrée automatiquement sous la forme d'un profil UML. La génération de l'implantation réduit la distance avec la spécification et limite la perte ou l'altération des concepts. Le profil

devient le moyen pour mettre en pratique une modélisation multiniveau. La troisième étape consiste à appliquer le profil généré sur le modèle cible (cf. section 4.5). Une assistance par un outil lors de cette dernière étape simplifie nettement la procédure. Dans la suite de la section, chacune des étapes est détaillée et illustrée sur une partie du modèle de temps de Marte dont le modèle multiniveau est discuté à la section 4.2. La dernière section discute les extensions à ce profil.

4.2 Modélisation explicite multiniveau

La carence de moyens pratiques pour suivre une modélisation multiniveau en UML a rendu l'implantation du modèle métier de Marte difficile. Son résultat, en l'état actuel, peut paraître inutilement complexe. Dans le modèle métier, (cf. Figure 9b), le concept d'horloge apparaît sous la forme d'une classe unique. Or, dans l'implantation (cf. Figure 7) sous forme de profil UML, deux stéréotypes Clock et ClockType ont été nécessaires. En utilisant les mécanismes de modélisation multiniveau, le passage du modèle de domaine à l'implantation devient systématique. La Figure 9 illustre la progression intellectuelle pour passer au concept d'horloge introduit dans Marte à une modélisation multiniveau.

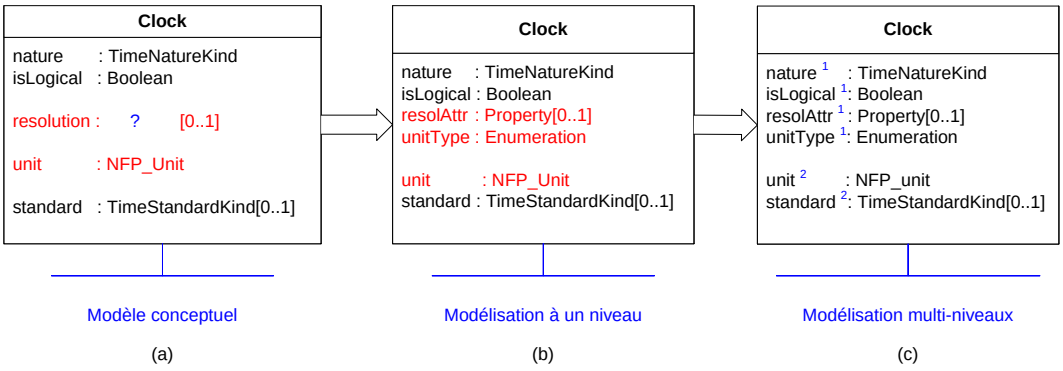

FIGURE 9 – Passage au modèle d'horloge multiniveau

La première étape consiste à définir les propriétés essentielles associées au concept d'horloge, ou Clock (cf. Figure 9a). Cette représentation est seulement partielle ici, par rapport au modèle adopté par l'OMG. Pour simplifier la discussion, un sous-ensemble minimal a été choisi pour illustrer les différents aspects de notre proposition. Comme l'explique la section 3.1, il est difficile de choisir un type pour modéliser la résolution et la modélisation de l'unité par un simple littéral n'est pas complètement satisfaisante pour assurer la compatibilité entre les horloges d'un même domaine.

Ainsi, le modèle est raffiné (cf. Figure 9b). L'attribut resolution est typé par la métaclasse Property. L'unité est décomposée en deux notions, l'unité elle-même (unit) et son type unitType qui définit l'ensemble des unités compatibles et autorisées. La métaclasse Enumeration est

choisie pour unitType à cause du lien sémantique qu'il y a dans UML entre EnumerationLiteral (et donc NFP_unit) et Enumeration.

Enfin, les niveaux sont rendus explicites par l'ajout d'un potentiel aux champs. Ainsi, le potentiel détermine le nombre d'instanciations nécessaires pour associer une valeur effective à l'attribut. Le cas d'usage de la Figure 8 montre qu'il y a trois niveaux (*instance*=0, *type*=1, *métatype*=2), ce qui justifie un potentiel de 2 pour les champs unit et standard qui prennent leur valeur au niveau *instance*. nature, isLogical et unitType prennent leur valeur au niveau *type* et ont donc logiquement un potentiel de 1. Enfin, resolAttr prend également un potentiel de 1 car sa valeur est également donnée au niveau *type*. Sa valeur n'est en effet pas la résolution elle-même (attribuée au niveau *instance*), mais la propriété qui a la sémantique d'une résolution.

Le résultat (Figure 9c) est très concis tout en conservant la même expressivité que le profil initial (Figure 7). Il n'existe pas à notre connaissance de méthode systématique pour déterminer les niveaux tout comme il n'existe pas de méthode systématique pour déterminer les classes à utiliser, les propriétés à faire apparaître et le choix des types à utiliser. Notre argument est que les niveaux existent dans la plupart des modèles et qu'il faut chercher à les identifier explicitement. Une fois ces niveaux identifiés, on peut soit essayer de les modéliser avec une des approches mentionnées à la section 2, soit choisir de modéliser explicitement ces niveaux. Notre profil présenté dans la section 4.3 permet cette modélisation explicite si on choisit UML comme langage de modélisation.

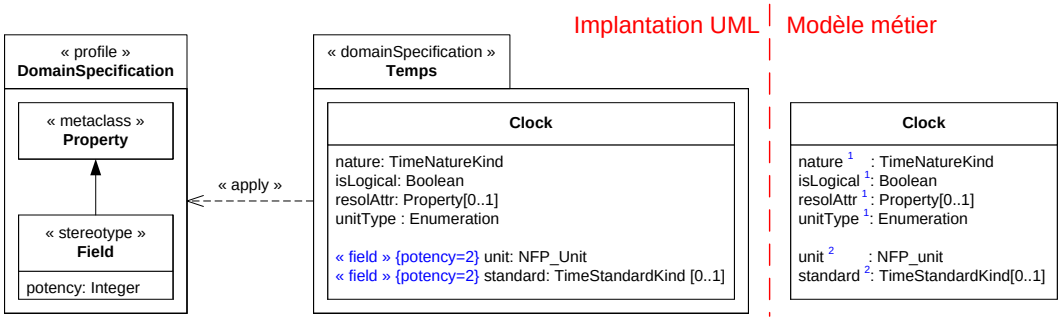

FIGURE 10 – Le profil domainSpecification appliqué au modèle de temps

4.3 La spécification du modèle métier

Notre profil pour la modélisation multiniveau (DomainSpecification) est présenté à la Figure 10. Il contient un seul stéréotype Field qui étend le concept de propriété UML (Property) et introduit la notion de champ associé à un potentiel. Les extensions possibles à ce profil sont discutées à la section 5. Les propriétés qui ne sont pas stéréotypées sont considérées comme des champs de potentiel 1. Le modèle obtenu à la Figure 9 peut désormais être modélisé en UML. Il constitue la partie droite de la Figure 10.

4.4 La génération automatique d'une implantation UML

Le modèle de domaine précédemment spécifié est automatiquement transformé en une implantation, c'est-à-dire sous forme de profil. L'information donnée par les potentiels est utilisée pour guider la génération. Le résultat, présenté à la Figure 11, est un profil UML spécifique pour le domaine considéré.

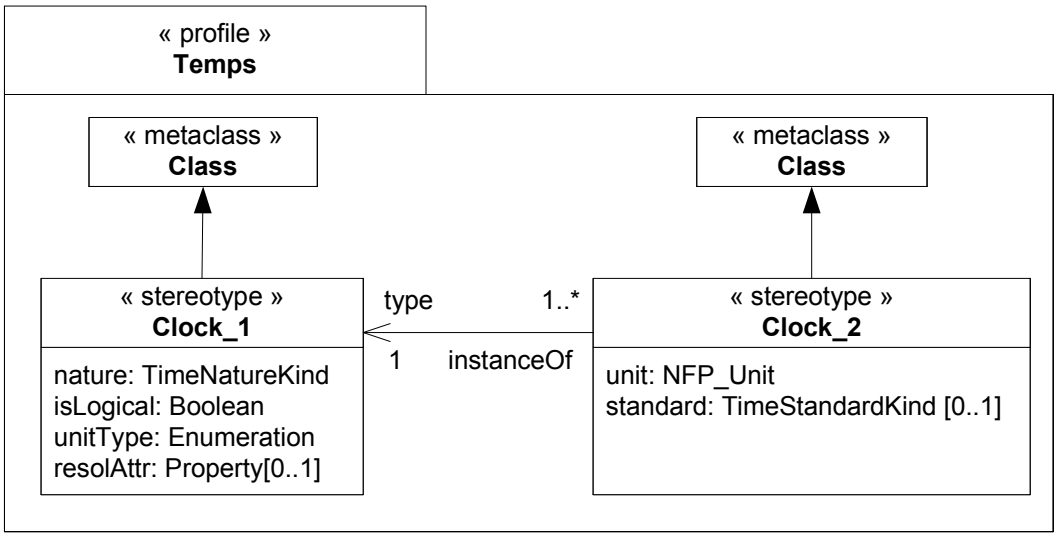

FIGURE 11 – Le profil de temps généré automatiquement

L'algorithme utilisé pour la génération fonctionne en une passe. Chaque classe donne lieu à autant de stéréotypes qu'il y a de niveaux d'instanciation. Le nom du stéréotype généré est celui du *clabject* initial suffixé par le niveau de modélisation concerné. Dans notre exemple, Clock fait apparaître deux niveaux d'instanciation car il contient des champs de potentiel 1 et des champs de potentiel 2. Deux stéréotypes sont alors générés. Le stéréotype Clock_2 contient des attributs qui représentent les champs de potentiel 2. Le stéréotype Clock_1 contient des attributs qui représentent les champs de potentiel 1. Tous les stéréotypes générés étendent par défaut la métaclasse Class d'UML. Le stéréotype Clock_1 représente le premier niveau d'instanciation et est l'équivalent de l'ancien stéréotype ClockType. Le stéréotype Clock_2 représente le deuxième niveau d'instanciation et donc l'ancien stéréotype Clock. Une association entre les deux stéréotypes est ajoutée pour maintenir le lien type/instance entre les éléments de modèles qui représentent un même concept.

On peut noter certaines différences mineures. En particulier, certaines associations (comme resolAttr) de la Figure 7 sont représentées ici comme des attributs pour simplifier la figure. Il est bien évident que si l'outil de modélisation permet de faire la distinction entre une

association et un attribut lors de la construction du domaine métier, alors cette distinction est reportée telle qu'elle lors de la génération du profil.

4.5 L'application du profil généré sur le modèle utilisateur

La Figure 12 illustre l'utilisation du profil généré en reprenant le même exemple que celui de la Figure 8.

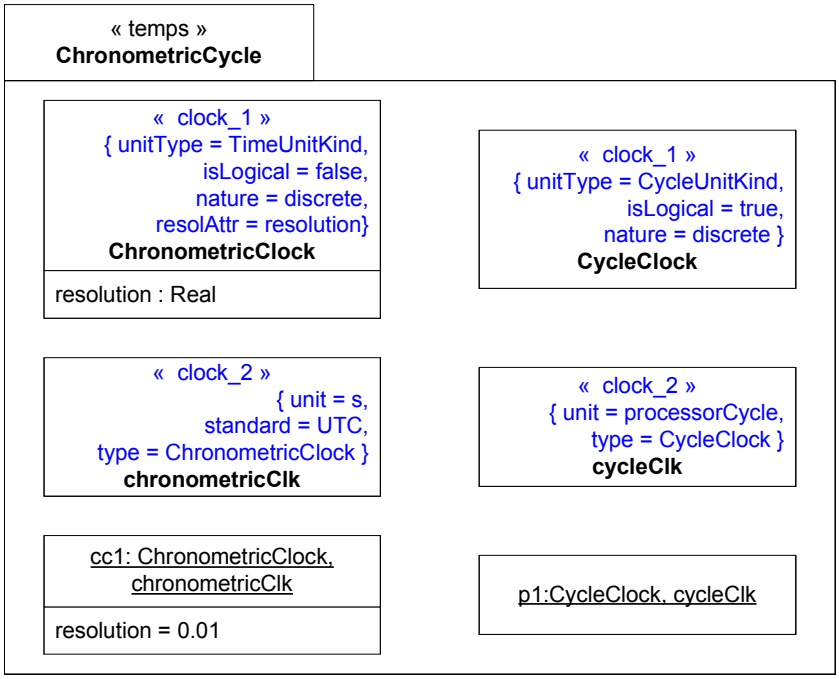

FIGURE 12 – La création d'horloges avec le profil généré

La modélisation de l'horloge logique Cycle s'effectue au travers de deux classes : CycleClock et cycleClk. La première stéréotypée par clock_1, la deuxième par clock_2. Chaque classe donne une valeur aux concepts d'un niveau spécifique. Le méta-attribut type du stéréotype clock_2 établit un lien vers CycleClock. Ce lien est indispensable pour savoir que cycleClk est effectivement une horloge logique discrète et dont l'ensemble des unités valides est défini par l'énumération CycleUnitKind.

La structure de ce modèle est très similaire à celui de la Figure 8. Il y a cependant des différences dues à la génération automatique et qui méritent d'être soulignées. La première différence évidente est qu'avec le modèle de l'OMG, les horloges (*e.g.* cycleClk) sont des instances alors qu'elles sont des classes ici. Le profil générique nécessite la création d'une

instance séparée (*e.g.* p1) avec deux types. Il en est de même pour l'horloge ChronometricClk et la valeur de la résolution est donnée dans l'instance cc1. Les autres différences et leurs conséquences sont discutées plus en détail dans la section suivante.

4.6 Des extensions possibles

Le profil proposé ici est minimaliste pour rendre le discours plus simple. Un profil plus complet et une transformation plus paramétrable ont été proposés par ailleurs [16]. Cependant les choix de génération proposés sont sujets aux pratiques de modélisation observées au sein de chaque équipe. L'idée est donc de plaider pour la fin d'une conception manuelle des profils en faveur d'approches génératives. Ainsi, si une équipe décide par la suite de générer un environnement spécifique au domaine plutôt que de personnaliser UML, le travail effectué sur le modèle métier peut être réutilisé et l'information sur les multiniveaux reste valable. De plus, rendre la ou les implantations automatiques autorise une comparaison des procédés de génération avec des métriques objectives.

La Figure 13 montre une extension simple possible avec un exemple d'utilisation. Il s'agit ici d'implanter explicitement le concept de *clabject* par un nouveau stéréotype qui vient d'ajouter au stéréotype Field. Le stéréotype Clabject a trois propriétés : potency, levelNames et extension.

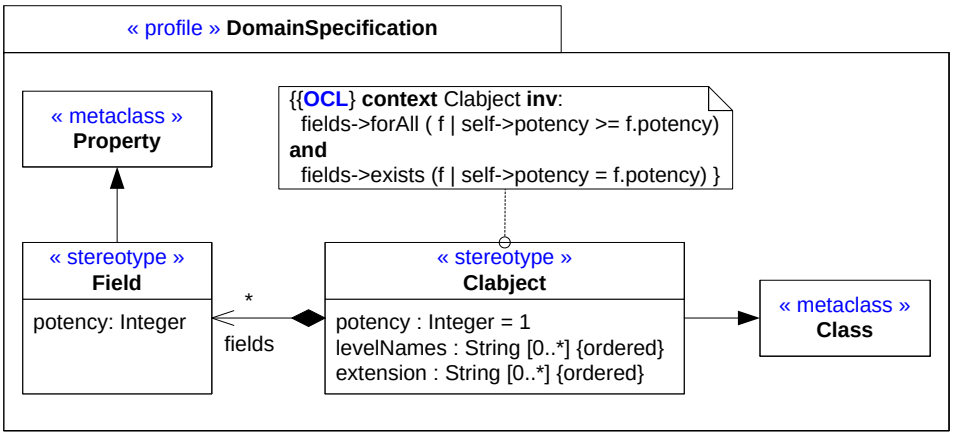

FIGURE 13 – Le stereotype Clabject

Dans la version simplifiée du profil de modélisation multiniveau (cf. Figure 10), toutes les classes du modèle étaient considérées implicitement comme des clabjects et transformées selon l'algorithme de génération. Dorénavant, seules les classes stéréotypées « clabject » seront transformées. Les autres seront reproduites telles quelles.

La propriété potency définit le potentiel d'un Clabject. Lorsque le potentiel n'est pas précisé, il est considéré par défaut comme égal à 1. Ce potentiel permet de connaître le nombre

maximum d'instanciations nécessaires. Une règle OCL impose que le potentiel d'un *clabject* est égal au maximum des potentiels des champs de ce *clabject*.

La propriété levelNames définit les règles de nommage des stéréotypes générés à partir du *clabject*. Dans la version simplifiée, le nom du *clabject* était suffixé par un numéro relatif à la profondeur d'instanciation. Ces identificateurs sont remplacés par les noms donnés sous forme d'un tableau de chaînes de caractères dans la propriété levelNames.

Enfin, la propriété extension définit les métaclasses à étendre par chacun des stéréotypes générés. Utiliser la métaclasse Class est toujours possible, c'est le choix par défaut. Cependant, le stéréotype représente plus précisément le concept du domaine métier si la métaclasse étendue est mieux ciblée. Par exemple, Marte a introduit la notion de TimedEvent qui modélise des événements associés à des horloges. Même s'il est possible d'étendre la métaclasse Class pour construire un TimedEvent, il est plus judicieux d'étendre la métaclasse TimeEvent définie dans UML.

La Figure 14 reprend l'exemple de la Figure 10 et illustre l'utilisation du stéréotype Clabject. La partie gauche est le modèle métier annoté. La partie droite est le profil qui résulte de la transformation.

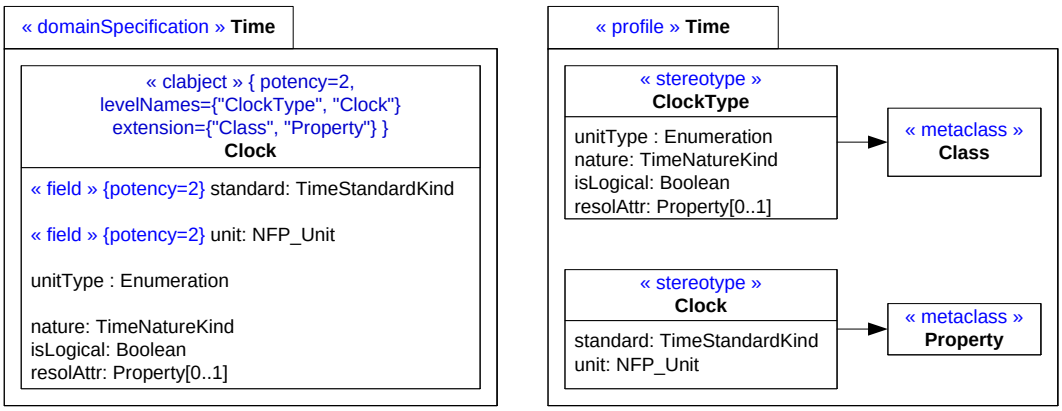

FIGURE 14 – Exemple d'utilisation du stereotype Clabject

5 Discussions

5.1 Une comparaison des flots de conception

La Figure 15 compare les flots de conception entre l'approche de génération préconisée et celle suivie par les concepteurs de Marte.

Les deux approches sont très similaires et procèdent en deux phases. La première phase est la modélisation du modèle métier. Notre proposition demande de rendre explicite les

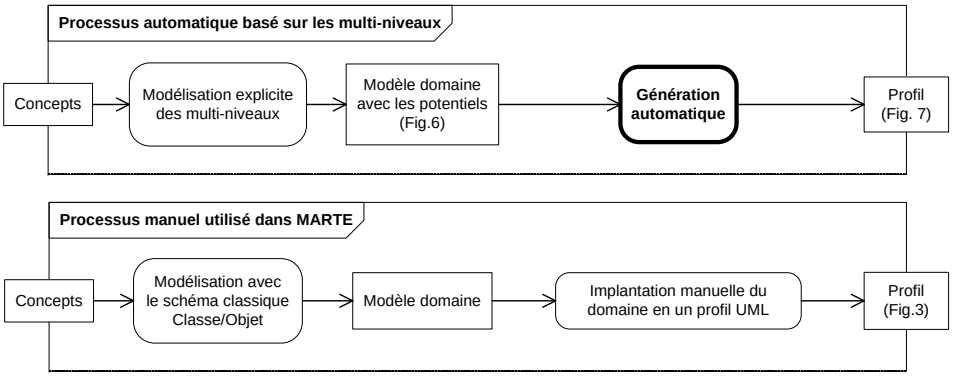

FIGURE 15 – Comparaison des flots de conception

différents niveaux de modélisation. C'est un travail qui s'inscrit dans les actions usuelles d'une conception de systèmes logiciels. Il s'agit entre autres d'identifier et d'appliquer des patrons de conceptions. Dès que le patron Type/Objet s'applique, il est traduit par des potentiels.

La deuxième phase est délicate et essentielle. Elle consiste à implanter les concepts domaines sur les concepts déjà disponibles en UML. La section 3.2 discute en détail tous les pièges à éviter et les choix faits dans Marte. Le procédé de génération que nous proposons implante ces choix et les rend explicites dans le moteur de transformation. Par ailleurs, rendre l'implantation automatique rend le processus plus sûr et assure qu'aucun concept n'a été ignoré ou ajouté involontairement. Enfin, la maintenance du modèle est facilitée car il n'y a plus qu'un seul modèle à maintenir, le modèle métier, l'implantation étant générée à nouveau chaque fois que nécessaire.

5.2 Les travaux connexes

Outre l'implantation des mécanismes de modélisation multiniveau dans UML, une contribution de notre travail est d'étudier les possibilités d'automatisation de la réalisation en UML d'un langage spécifique de domaine (DSL). L'intérêt attendu est la réutilisation de modeleurs UML, qui sont devenus matures, et pour lesquels une génération d'ingénieurs est formée ou du moins sensibilisée. Ceci devrait améliorer la réutilisation de modèles et réduire les coûts de conception.

Dans les travaux sur l'implantation de DSL, deux communautés se confrontent. La communauté des profileurs et celle des métamodeleurs. Les solutions dites légères passent par la création de profils alors que la création d'outils plus aboutis passe par la métamodélisation. Ce travail focalise sur la création de profils mais, la problématique de modélisation multiniveau est tout autant d'actualité pour la communauté des métamodeleurs. En effet, la cible de génération qui est actuellement un profil pourrait aussi bien être un métamodèle.

Dans ce travail, nous nous sommes concentrés sur la relation « instanceOf ». Cette relation n'est certainement pas exclusive et doit être combinée avec d'autres relations tout aussi importantes comme les associations, dépendances, composition ou encore les relations de conformité. En revanche, cette relation a spécifiquement inspiré de nombreux travaux, en particulier sur la définition de patrons de conception. Il nous semble évident qu'elle joue un rôle prédominent dans l'activité de modélisation, surtout avec les approches orientées objet ou orientées composant.

Dans la communauté des profileurs et malgré le nombre important de profils créés, il y a très peu de travaux qui se posent la question de mesurer la validité de profils par rapport au domaine visé. Il nous semble que c'est pourtant un aspect essentiel. Des progrès encourageants ont été faits depuis que l'activité de profilage proprement dite est précédée de la construction d'un modèle métier [8]. Uun travail plus récent [22] décrit les étapes nécessaires à la création d'un profil cohérent en évitant les pièges classiques. Il donne également des conseils précieux pour choisir les métaclasses à étendre.

La section 2 discute les nombreux mécanismes de modélisation multiniveau. Nous complétons ici cette discussion par la description brève de travaux connexes.

Les *powertype*s contournent le problème de la modélisation multiniveau en utilisant une association pour traduire la relation de typage. Des travaux ont déjà établi un parallèle entre les *powertypes* et les mécanismes de profilage [9]. La section 2 décrit en détail la raison pour laquelle nous préférons utiliser la modélisation multiniveau plutôt que les powertypes. Notre approche est cependant similaire puisque nous faisons un lien avec les mécanismes de profilage.

Atkinson et Kühne ont étudié les fondements de la modélisation multiniveau et ont proposé une intégration au métamodèle de UML1.x [3]. Les changements entre UML 1 et UML 2 demandent une adaptation de leur approche. Notre proposition va dans ce sens mais considère qu'il est inutile d'envisager une refonte lourde du métamodèle et qu'une approche par profilage convient. Ils ont aussi proposé une implantation dans le langage de programmation Java [12]. D'autres travaux ont intégré les mêmes concepts dans un langage de métamodélisation appelé Nivel [2] qui dispose d'une sémantique formelle. L'utilisation d'UML est en amont de l'utilisation de langages de programmation et notre approche est donc complémentaire de la leur sur ce point.

Globalement, notre approche se différencie dans son objectif. Nous avons évalué l'intérêt de la modélisation multiniveau pour guider la génération automatique de profils UML à partir du modèle métier et nous avons montré que l'intégration avec UML2 est réalisable sans aucune modification du métamodèle d'UML.

5.3 Généralisation de l'approche

Pour illustrer l'aspect général de l'approche, nous l'illustrons sur un nouvel exemple indépendant du profil Marte. Cet exemple met en œuvre un nombre de niveaux supérieur. Chaque niveau est justifié par les exigences issues du domaine métier. L'approche présentée à la section précédente est appliquée de façon systématique.

Cet exemple modélise des mesures physiques. La Figure 16 montre notre spécification en *clabjects*. Les concepts et le vocabulaire correspondants sont rassemblés dans un document produit par le Comité commun pour les guides en métrologie [10]. Une mesure physique porte sur une *grandeur* physique. Le résultat s'exprime par une *valeur* numérique et une *unité* associée à la grandeur mesurée. Une mesure fait implicitement référence à un *système de grandeurs*. Ce système choisit un ensemble de grandeurs de base à partir desquelles d'autres grandeurs peuvent être dérivées. Dans notre exemple les grandeurs de base sont la *longueur* et le *temps*, la grandeur dérivée est la *vitesse*. Pour un système donné, à chaque grandeur de base est associée une *unité de base*. Une grandeur dérivée est caractérisée par une *équation aux grandeurs* qui la lie aux grandeurs de base.

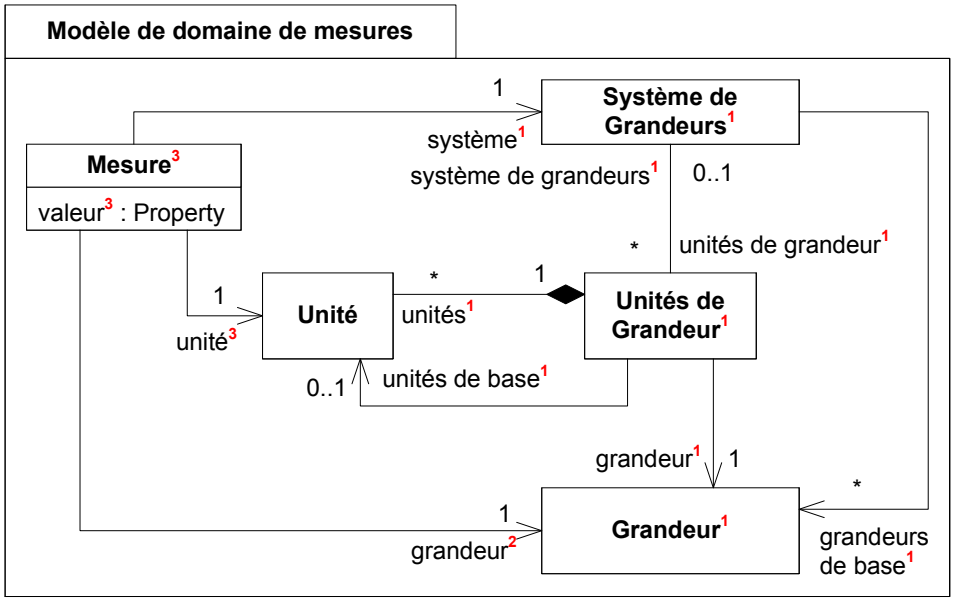

FIGURE 16 – Spécification de Mesure avec les clabjects

Dans notre exemple, la vitesse a pour équation $vitesse = longueur/temps$. Étant donné une grandeur à mesurer, la première chose à faire pour définir une mesure est de choisir un *système de grandeurs*. On spécifie ensuite la grandeur à mesurer par sa *dimension* qui s'obtient à partir de son équation aux grandeurs et des grandeurs de base du système. Dans le cas étudié, la dimension est $L.T^{-1}$. Enfin, on choisit une unité pour la grandeur dérivée. Chaque niveau de décision se traduit par un niveau de modélisation différent. L'utilisation de *clabjects* permet une modélisation explicite de ces niveaux et donc de l'ordre dans lequel doivent se faire les décisions.

Pour simplifier les diagrammes, les *équations de grandeurs* n'ont pas été représentées. L'ordre dans lequel se font les choix détermine le potentiel à appliquer : système a le potentiel 1, grandeur le potentiel 2 et unité le potentiel 3. Notons, que valeur a également le potentiel 3 alors que la valeur effective sera donnée un temps plus tard que l'unité. Cela vient du fait que le type choisi pour le champ valeur est Property, *i.e.* une métaclasse. Cette utilisation de métaclasses a été justifiée dans la section 3.

La Figure 17 contient les quatre instanciations successives de ces clabjects (5 niveaux de modélisation).

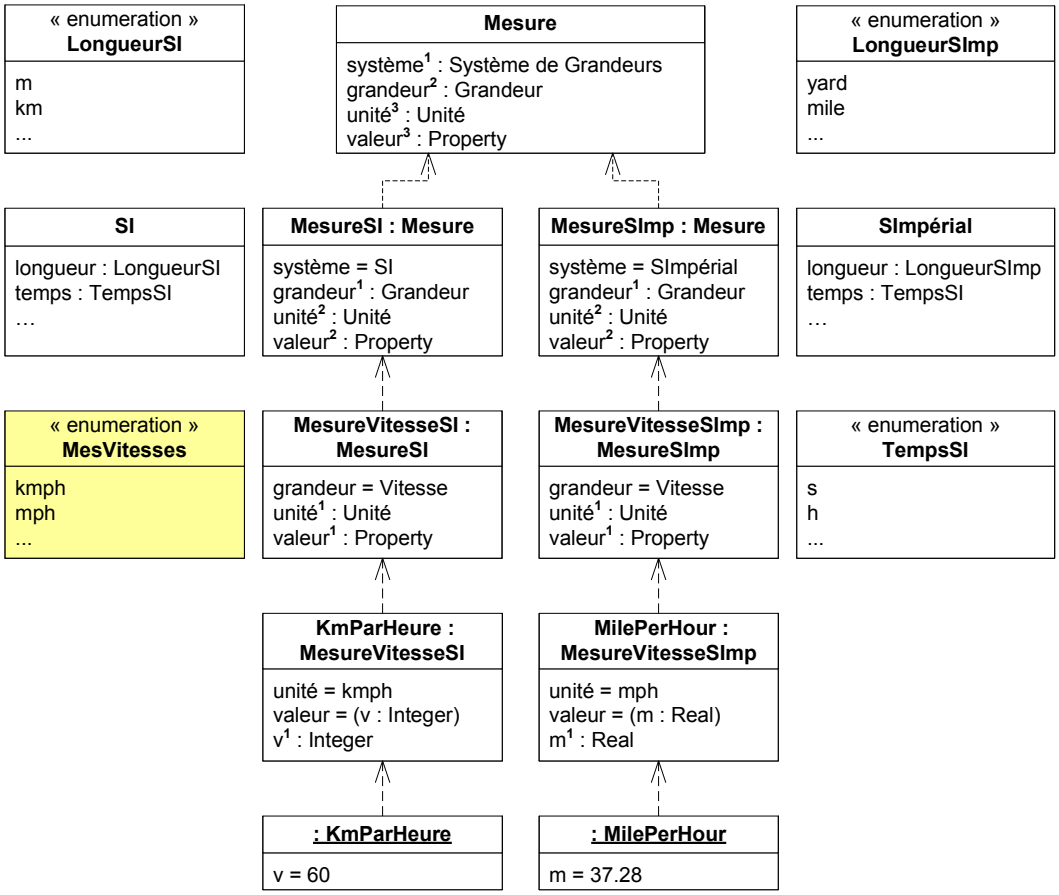

FIGURE 17 – Exemple de systèmes, grandeurs et unités avec les clabjects

Nous avons considéré deux cas : une réalisation dans le cadre strict du *système international* (SI, abréviation officielle de ce système) et une empruntant l'unité de longueur au *système*

impérial (SImp). Des types supplémentaires sont nécessaires pour décrire les systèmes, les grandeurs et les unités. Certains de ces éléments, en particulier ceux relatifs au système SI, peuvent être regroupés dans une bibliothèque. D'autres, comme l'énumération MesVitesses (fond sombre), sont spécifiques à l'exemple.

En suivant notre approche, il convient de construire un modèle UML qui représente ce domaine métier et de l'annoter en appliquant notre profil pour la modélisation multiniveau (cf. Figure 18). Pour alléger les diagrammes, les noms de rôle composés ont été remplacés par leurs acronymes.

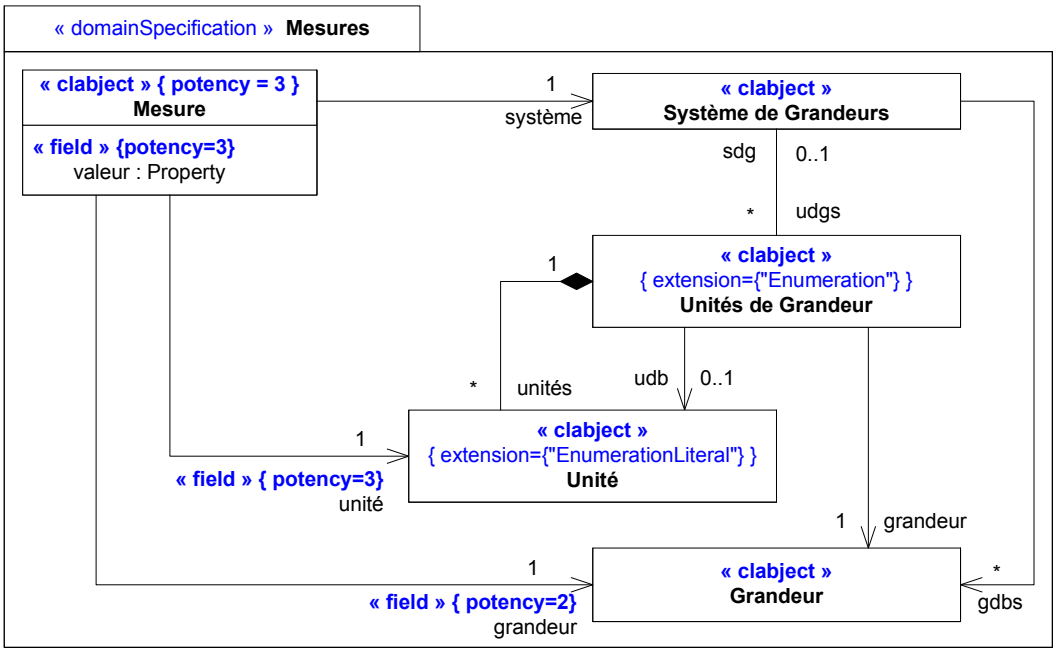

FIGURE 18 – Spécification de Mesure

Notons que les concepts de *Mesure*, *Système de Grandeurs* et *Grandeur* sont représentés par des classes, ce qui est le choix par défaut. La métaclasse Class a été omise sur le diagramme. En revanche, les *Unités de Grandeurs* sont représentées par des énumérations et les unités par des littéraux ; ces métaclasses sont explicites. Ce modèle est alors transformé automatiquement en un profil présenté dans la Figure 19 en appliquant l'algorithme suivant :

```
pour chaque clabject c faire
  si c.potency = 1 alors
    générer un stéréotype s de nom c.name
    chaque champ du clabject c devient un méta-attribut de s
```

```
  sinon
    pour i de 1 à c.potency faire
      si c.levelNames est indéfini alors
        générer un stéréotype s de nom (c.name + '_' + i)
      sinon
        générer un stéréotype s de nom c.levelNames[i-1]
      fin si

      si c.extension est indéfini alors
        s étend la métaclasse Class
      sinon
        s étend la métaclasse c.extension[i-1]
      fin si

      pour chaque champ ch de c faire
        si ch.potency = 1 alors
          ajouter un méta-attribut à s de mêmes nom et type que ch
        sinon
          décrémenter ch.potency
        fin si
      fin pour
    fin pour
  fin si
fin pour
```

En utilisant ce nouveau profil généré, l'exemple de la Figure 17 peut désormais être modélisé entièrement en UML. La Figure 20 contient des éléments d'usage général qui devraient appartenir à des bibliothèques. Notez que le système impérial ne définissant pas la grandeur Temps, nous lui avons fourni une référence sur TempsSI défini dans le cadre du système SI.

La Figure 21 montre la partie du modèle UML directement dépendante de l'application traitée. On construit deux mesures de vitesse exprimées en km/h et en mph. Les unités peuvent être enrichies comme celles de NFP pour faire apparaître les relations de conversion qui permettent en particulier d'exprimer analytiquement la relation : 1 mph = 1.61 km/h. En pratique, l'utilisateur se contentera des classes stéréotypées KmParHeure ou MilePerHour pour modéliser ses mesures de vitesse.

Au-delà de cet article, cet exemple est intéressant pour alimenter les discussions en cours en vue d'améliorer le modèle de représentation des valeurs de grandeurs existant dans SysML et éventuellement introduire les notions de dimension, unités et grandeurs dans UML.

6 Conclusion et perspectives

Cet article propose un procédé automatique pour générer un profil UML à partir du modèle métier. Le modèle métier est réalisé en UML et les différents niveaux de typage sous-jacents au domaine sont rendus explicites à l'aide des *potentiels* [3]. Les potentiels sont exprimés à l'aide d'un nouveau profil UML, appelé DomainSpecification, qui supporte les concepts de *clabject*, *champ* et *potentiel*.

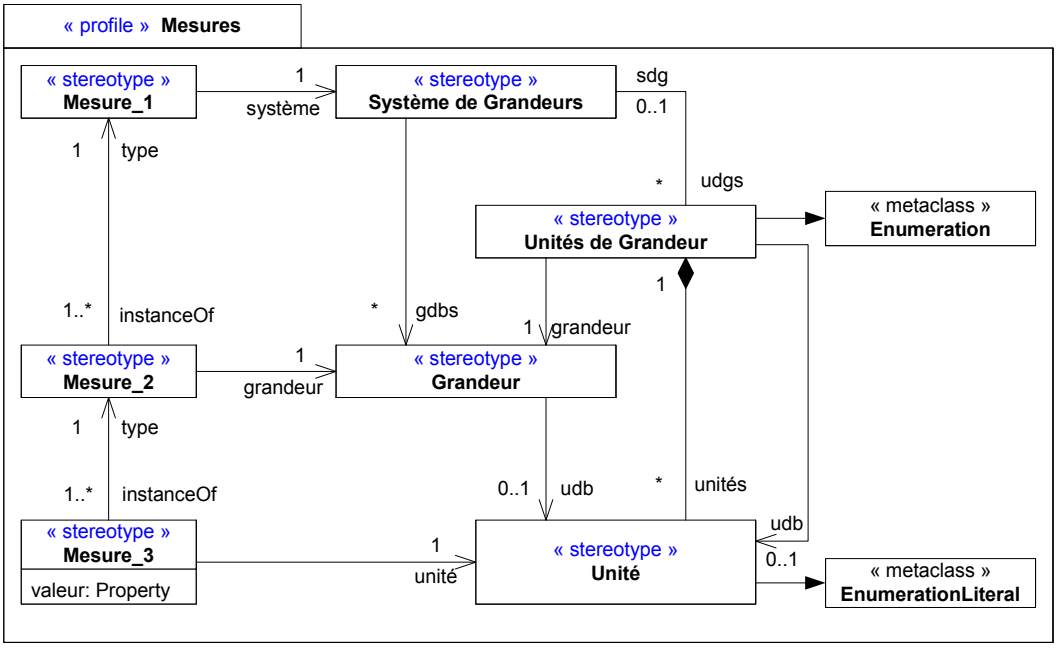

FIGURE 19 – Le profil généré pour Mesure

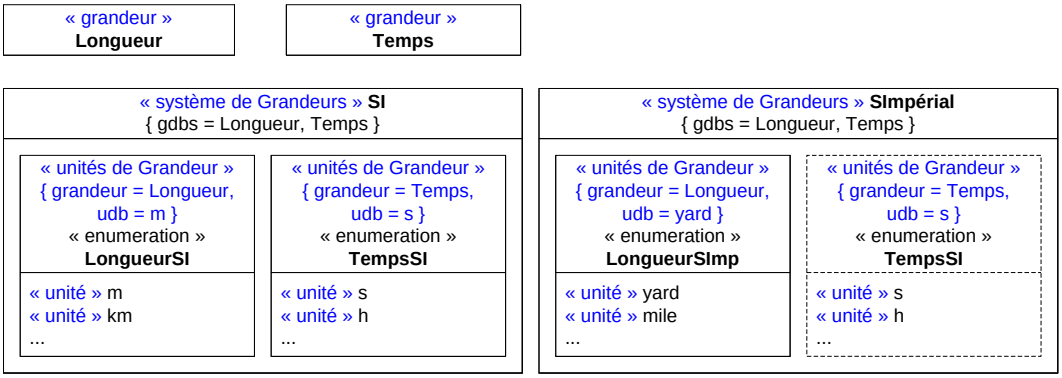

FIGURE 20 – Modèle UML pour deux systèmes de grandeurs

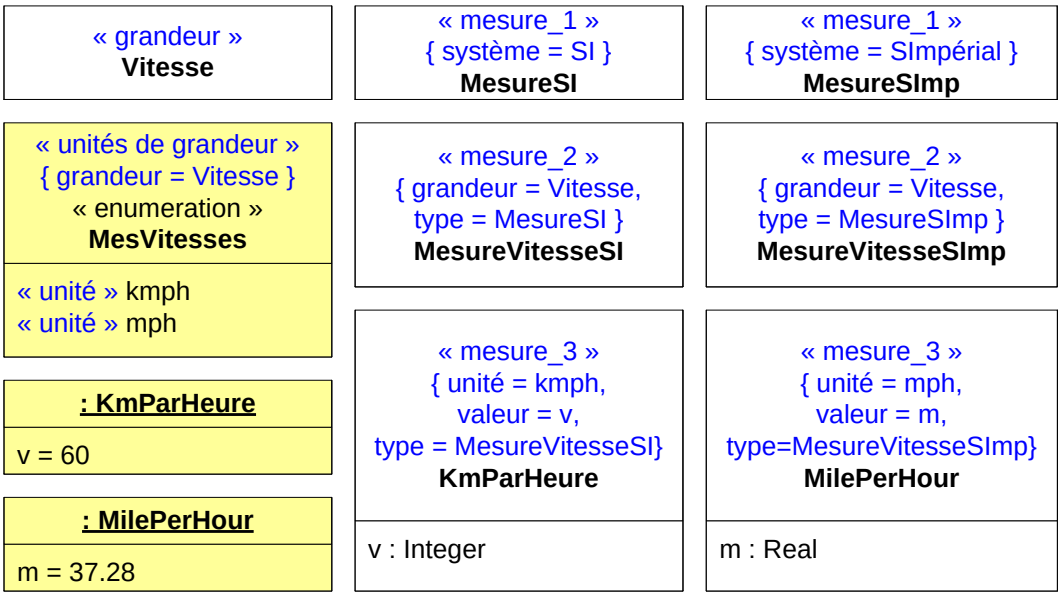

FIGURE 21 – Modèle UML pour deux mesures de vitesse

La génération tient compte des différents niveaux pour produire un profil dans lequel les aspects liés au typage sont garantis par construction. Les stéréotypes générés assurent une implantation de tous les aspects liés à un concept donné, aux différents niveaux mis en évidence. Ils maintiennent également un lien entre les différents niveaux d'un même concept et évitent ainsi une dispersion de l'information.

Le procédé de génération proposé et l'utilisation du profil pour la modélisation multiniveau sont illustrés sur deux exemples. Le premier est un extrait du modèle de temps de Marte récemment adopté par l'OMG. L'utilisation de la modélisation multiniveau rend explicite certains choix des concepteurs du profil. Une assistance par un moteur de transformation rend possible de nouvelles opportunités. Par exemple, l'utilisateur désireux de créer une horloge pourrait être guidé dans sa démarche puisque le lien entre Clock et ClockType est rendu explicite. Ainsi, une interface graphique peut s'appuyer sur le modèle multiniveau pour guider l'utilisateur dans la création des ClockType, puis des Clock et enfin l'établissement de liens avec les éléments de modèle existants. Nous considérons également que cette discussion éclaire sur les intentions des concepteurs de Marte et qu'avec un tel support, il n'est plus utile de présenter le profil lui-même mais que les discussions peuvent être focalisées sur la modélisation du temps.

Le second exemple permet d'exprimer en UML des résultats de mesures physiques. Ces résultats comprennent une valeur numérique et une unité. La modélisation retenue, conforme aux recommandations des organismes internationaux de métrologie [10], introduit quatre

niveaux d'instanciation. La présentation décrit de façon détaillée l'application de notre approche multiniveau pour définir des mesures de vitesse. Cette illustration est une contribution aux discussions en cours pour la révision de SysML et la modélisation des concepts d'*unité*, *grandeur* et *dimension*.

De façon plus large, nous préconisons l'utilisation d'une méthode de génération de profils de préférence à une activité manuelle sujette aux erreurs. Une approche automatique permet d'appliquer des métriques de façon systématique sur de gros projets (comme Marte) et ainsi de comparer différentes approches avec des critères objectifs. Ce travail ne propose pas de métriques mais il est clair que pour quantifier objectivement la complexité accidentelle sur des exemples plus gros que celui présenté ici, l'usage de métriques est nécessaire.

Références

[1] Charles André, Frédéric Mallet, and Robert de Simone. Modeling time(s). In *Proc., 10th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of *LNCS*, pages 559–573. Springer, 2007.

[2] Timo Asikainen and Tomi Männistö. Nivel : a metamodelling language with a formal semantics. *Software and Systems Modeling*, 8(4) :521–549, 2009.

[3] Colin Atkinson and Thomas Kühne. The Essence of Multilevel Metamodeling. *UML - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, 2185 :19–33, October 2001.

[4] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software and System Modeling*, 7(3) :345–359, 2008.

[5] Alan Borning. Classes versus prototypes in object-oriented languages. In *Proc. of the Fall Joint Computer Conference*, pages 36–40. IEEE Computer Society, November 1986.

[6] Frederick Phillips Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4) :10–19, 1987.

[7] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9) :152–159, 1992.

[8] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. An Introduction to UML Profiles. *UML and Model Engineering*, V(2), April 2004.

[9] Brian Henderson-Sellers and Cesar Gonzalez-Perez. Connecting powertypes and stereotypes. *Journal of Object Technology*, 4(7) :83–96, 2005.

[10] JCGM. *Vocabulaire international de métrologie (VIM) – Concepts fondamentaux et généraux et termes associés*. Joint Committee for Guides in Metrology, 2008. JCGM 200.

[11] Ralph Johnson and Bobby Woolf. *Type Object*, volume 3, pages 47–65. ADDISON-WESLEY, October 1997.

[12] Thomas Kühne and Daniel Schreiber. Can programming be liberated from the two-level style? – Multi-level programming with DeepJava. In *OOPSLA'07 : Proc. of the 22nd annual ACM SIGPLAN conf. on Object oriented programming systems and applications*, pages 229–244. ACM, 2007.

[13] François Lagarde. *Contribution à la conception de langages de modélisation spécifiques fondés sur UML*. PhD thesis, Université de Nice-Sophia Antipolis, November 2008.

[14] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An exemplar based smalltalk. In *OOPSLA*, pages 322–330, 1986.

[15] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPLSA'86 : Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223, New York, NY, USA, 1986. ACM.

[16] Frédéric Mallet, François Lagarde, Charles André, Sébastien Gérard, and François Terrier. An automated process for implementing multilevel domain models. In *2nd Int. Conf. on Software Language Engineering, SLE 2009*, Lecture Notes in Computer Sciences, October 2009.

[17] James Odell. *Advanced Object-Oriented Analysis & Design using UML*, volume 12 of *SIGS Reference Library*. Cambridge University Press, 1998.

[18] James Odell. *Power Types*, chapter 3. Volume 12 of *SIGS Reference Library* [17], 1998.

[19] OMG. *Unified Modeling Language, Superstructure, v2.1.2*. Object Management Group, November 2007. formal/2007-11-02.

[20] OMG. *Systems Modeling Language (SysML), v1.1*. Object Management Group, November 2008. formal/2008-11-02.

[21] OMG. *UML Profile for MARTE, v1.0*. Object Management Group, November 2009. formal/2009-11-02.

[22] Bran Selic. A systematic approach to domain-specific language design using uml. In *Tenth IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 2–9. IEEE Computer Society, 2007.

[23] Frédéric Thomas, Jérôme Delatour, François Terrier, and Sébastien Gérard. Towards a framework for explicit platform-based transformations. In *11th IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC'08)*, pages 211–218. IEEE Computer Society, May 2008.

[24] Tim Weilkiens. *Systems Engineering with SysML/UML : Modeling, Analysis, Design*. The MK/OMG Press, Burlington, MA, USA., 2008.

Table des matières

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399