HAL
archives-ouvertes.fr

Une étude sémantique du langage QML

Jacques Malenfant

▶ To cite this version:

Jacques Malenfant. Une étude sémantique du langage QML. [Rapport de recherche] RR-4497, INRIA. 2002. inria-00072091

HAL Id: inria-00072091
https://hal.inria.fr/inria-00072091

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Une étude sémantique du langage QML

Jacques Malenfant

N° 4497

Juillet 2002

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4497--FR

Une étude sémantique du langage QML

Jacques Malenfant*

Thème 1 — Réseaux et systèmes
Projet Triskell

Rapport de recherche n° 4497 — Juillet 2002 — 56 pages

Résumé : Le langage QML (*Quality of service Modeling Language*) a été proposé par Frølund et Koistinen pour traiter de manière systématique et déclarative de la qualité de service des composants logiciels. Dans ce rapport, nous présentons le résultat d'une analyse sémantique formelle du langage QML selon l'approche dénotationnelle. De cette sémantique, nous tirons des enseignements dans deux directions : la modélisation par objets des contrats dans le but d'une intégration fluide à UML et une implantation en Java pour une utilisation concrète dans ce langage. Si QML propose des concepts intéressants pour la définition d'ontologies et de types de données pour traiter des types de contrats, des contrats et des niveaux ou valeurs de qualité de service, le sous-langage de contraintes sur les dimensions et la liaison avec les interfaces restent peu approfondis et inaboutis.

Mots-clés : qualité de service, programmation contractuelle, composants logiciels, sémantique dénotationnelle, UML, OCL.

* également Université de Bretagne sud, laboratoire VALORIA.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

A semantic account of QML

Abstract: The QML language (*Quality of service Modeling Language*) has been proposed by Frølund and Koistinen in order to deal with the quality of service of software components in a systematic and declarative manner. In this report, we present the result of a formal analysis of QML using a denotational semantics approach. From this semantics, we derive lessons in two directions : the object-oriented modeling of contracts towards a seamless integration to UML and a Java implementation pointing to concrete uses of the language. If QML offers interesting concepts to define ontologies et data types to deal with contract types, contracts and levels of quality of service, the constraint sublanguage on dimensions as well as the link with interfaces are still underdeveloped and did not come off yet.

Key-words: quality of service, design by contract, software components, denotational semantics, UML, OCL.

1 Introduction

Frølund et Koistinen [FK98] ont proposé et défini un langage de modélisation de la qualité de service appelé QML de manière à promouvoir l'expression déclarative des propriétés qualitatives et quantitatives de qualité de service dans les systèmes répartis. Ce langage se veut généraliste. En ce sens, il vise d'abord à définir des types de contrats qui proposent une ontologie générale du domaine par la définition et la désignation des différentes dimensions, niveaux et valeurs de la qualité de service. Cette organisation du vocabulaire de la modélisation faite, QML permet également de poser des contraintes sur les différentes dimensions identifiés par des contrats puis de lier ces contrats à des interfaces.

2 Description du langage QML

Il ne s'agit pas ici de décrire exhaustivement QML. Le lecteur se reportera aux travaux de Frølund et Koistinen [FK98] pour avoir une idée plus complète du langage et acquérir les notions essentielles à la compréhension de notre étude. Dans ce rapport, nous nous contentons d'expliquer les éléments utiles à l'étude que nous avons mené.

La figure 1 présente ainsi la syntaxe concrète de QML. Un programme QML (`program`) est essentiellement une série de déclarations (`decls`) de types de contrats (`conTypeDecl`), de contrats (`conDecl`) et de profils (`profileDecl`) auxquels sont associés des noms permettant d'en parler dans la suite du programme. Pour les besoins de notre expérience, nous avons introduit des expressions (`test`) permettant de vérifier si une ou des valeurs de qualité de service mesurées satisfont ou non un contrat.

Un type de contrats (`conType`) est constitué d'une séquence de déclarations de dimensions (`dimensions`), lesquelles sont formées d'un identificateur de dimension et d'un type de dimension (`dimType`). Un type de dimension est lui, constitué d'un sorte de dimension (`dimSort`) suivi ou non d'une unité de mesure (`unit`). Une unité peut être un pourcentage, ou encore une unité nommée (par exemple, «`sec`») ou encore une unité composée (par exemple, «`m/sec`»). QML prévoit cinq sortes de dimensions. Les quatres premières permettent de définir des niveaux de qualité de service à l'aide d'identificateurs. Il s'agit donc essentiellement de se donner les moyens d'une modélisation *qualitative* de la qualité de service.

Ces identificateurs peuvent être utilisés individuellement pour définir des valeurs de qualité de service, avec ou sans relation d'ordre (`order`) entre elles, ou encore comme ensembles de valeurs, encore une fois avec ou sans relation d'ordre entre ces valeurs nommées individuelles. Dans le premier cas, les dimensions de types énumération (`enum`) et énumération avec relation d'ordre (`ordered-enum`) sont utilisées, alors que dans le second cas, ce sont plutôt les dimensions de types ensemble (`set`) ou ensemble partiellement ordonné (`poset`) qui sont utilisées. Une dernière sorte de dimension est la dimension numérique (`numeric`), servant bien entendu à la modélisation quantitative de la qualité de service. Une information importante donnée par une déclaration de dimension est la sémantique ou signification de la notion de valeur préférable ou meilleure valeur (`relSem`). En présence d'une relation d'ordre, il est nécessaire de préciser si les valeurs préférables dans une dimension sont les petites ou les grandes valeurs relativement à la relation d'ordre définie.[1]

Un contrat pour sa part est défini (`conExp`) par rapport à son type de contrat, désigné directement par son identificateur dans les déclarations simples et indirectement par le contrat étendu dans les contrats définis par raffinement d'un contrat pré-existant. Outre cela, une définition de contrat (`contractDef`) est constituée d'une série de contraintes (`constraints`) sur les dimensions déclarées dans le type de contrat. Une contrainte simple est constituée d'un opérateur de

[1] Pour un temps de réponse, il est clair que les valeurs les plus petites sont les plus «intéressantes», mais c'est certainement l'inverse pour un temps entre les pannes.

```
program        ::= decls test
decls          ::= decls ind-decl | ind-decl
ind-decl       ::= decl ;
decl           ::= conTypeDecl | conDecl | profileDecl
conTypeDecl    ::= TYPE IDENT = conType
conType        ::= CONTRACT { dimensions } ;
dimensions     ::= dimensions dimension | dimension
dimension      ::= dimName : dimType ;
dimName        ::= IDENT
dimType        ::= dimSort | dimSort unit
items          ::= items , IDENT | IDENT
dimSort        ::= ENUM { items } | relSem ENUM { items } WITH order
                 | SET { items } | relSem SET { items }
                 | relSem SET { items } WITH order | relSem NUMERIC
relations      ::= relations , relation | relation
relation       ::= IDENT < IDENT
order          ::= ORDER { relations }
unit           ::= unit / base-unit | base-unit
base-unit      ::= % | IDENT
relSem         ::= DECREASING | INCREASING
conDecl        ::= IDENT = conExp
conExp         ::= IDENT contractDef
                 | IDENT REFINED BY { constraints }
contractDef    ::= CONTRACT { constraints }
constraints    ::= constraints ind-constraint | ind-constraint
ind-constraint ::= constraint ;
constraint     ::= dimName constraintOp dimValue | dimName { aspects }
constraintOp   ::= == | >= | <= | > | <
dimValue       ::= literal unit | literal
literal        ::= IDENT | { items } | NUMBER
aspects        ::= aspects ind-aspect | ind-aspect
ind-aspect     ::= aspect ;
aspect         ::= PERCENTILE NUMBER constraintOp dimValue
                 | MEAN constraintOp dimValue
                 | VARIANCE constraintOp dimValue
                 | FREQUENCY freqRange constraintOp NUMBER %
freqRange      ::= dimValue | lRangeLimit dimValue , dimValue rRangeLimit
lRangeLimit    ::= [ | (
rRangeLimit    ::= ] | )
profileDecl    ::= IDENT FOR IDENT = profileExp
profileExp     ::= profile | IDENT REFINED BY { requisites }
profile        ::= PROFILE { requisites }
requisites     ::= requisites ind-requisite | ind-requisite
ind-requisite  ::= requisite ;
requisite      ::= REQUIRE contractList | FROM entityList REQUIRE contractList
contractList   ::= contractList , contractElem | contractElem
contractElem   ::= IDENT | conExp
entityList     ::= entityList , entity | entity
entity         ::= IDENT | IDENT . IDENT | RESULT OF IDENT
```

FIG. 1 – Syntaxe concrète du langage QML

```
type Reliability = contract {
  failureMasking : decreasing set { omission, lostResponse, noExecution,
                              response, responseValue, stateTransition } ;
  serverFailure : enum { halt, initialState, rolledBack } ;
  operationSemantics : decreasing enum { atLeastOnce, atMostOnce, once }
                  with order { once < atLeastOnce, once < atMostOnce } ;
  rebindingPolicy : decreasing enum { rebind, noRebind }
                  with order { noRebind < rebind } ;
  dataPolicy : decreasing enum { valid, invalid }
                  with order { valid < invalid } ;
  numberOfFailures : decreasing numeric no / year ;
  availability : increasing numeric  ;
} ;

systemReliability = Reliability contract {
  failureMasking < { noExecution, response } ;
  serverFailure == initialState ;
  operationSemantics <= { atMostOnce } ;
  rebindingPolicy < rebind ;
  dataPolicy < invalid ;
  numberOfFailures < 10 no / year ;
  availability > 0.8 ;
} ;
```

FIG. 2 – Exemple de contrat en QML

contraintes (`constraintOp`), qui est en fait un opérateur de comparaison classique, et d'une valeur cible de la contrainte (`dimValue`). Cependant, QML permet aussi de contraindre statistiquement une dimension (`aspect`), en posant des contraintes sur la moyenne des valeurs, leur variance, leur distribution de fréquences ou les valeurs aux percentiles.

Les profils servent à associer des contrats aux interfaces. Plus précisément, une déclaration de profil (`profileDecl`) comporte un identificateur du profil, l'identificateur de l'interface et une série d'expressions (`profileExp`) requérant l'observation de contrats par des entités de l'interface désignée (`requisites`). La clause `require` lie par défaut toutes les entités de l'interface à une liste de contrats, alors que la clause `from ... require ...` identifie spécifiquement une liste d'entités de l'interface auxquelles seront imposées la liste de contrats désignés dans la clause. Les entités sont soit des identificateurs simples, soit des identificateurs en notation pointée, soit des identificateurs qualifiés par la forme `result of` indiquant que c'est au résultat de l'entité désignée par l'identificateur qu'est imposée la liste de contrats.

La figure 2 propose un exemple suivant la syntaxe concrète de QML. Il comporte deux déclarations, l'un d'un type de contrat `Reliability` et l'autre d'un contrat `systemReliability` de type `Reliability`. Le type de contrat `Reliability` déclare sept dimensions, nommées respectivement `failureMasking`, `serverFailure`, `operationSemantics`, `rebindingPolicy`, `dataPolicy`, `numberOfFailures` et `availability`. La dimension `failureMasking` est définie par des ensembles des valeurs nommées parmi `omission`, `lostResponse`, `noExecution`, etc., dont la sémantique (`decreasing`) interprète les plus petits ensembles de valeurs comme les valeurs préférables dans cette dimension. La dimension `serverFailure` est définie par des valeurs nommées individuelles sans ordre entre elles. Les dimensions `operationSemantics`, `rebindingPolicy` et `dataPolicy` sont également définies par des valeurs nommées entre lesquelles sont définis des ordres partiels. Enfin, les dimensions `numberOfFailures` et `availability` sont des dimensions numériques dont la première a une unité (nombre de pannes par an).

Le contrat `systemReliability` pose des contraintes sur les sept dimensions précédentes. Notons que pour la dimension `failureMasking`, c'est l'inclusion ensembliste qui joue le rôle de relation d'ordre, et que pour la dimension `serverFailure`, seul l'opérateur d'égalité est admis dans la contrainte car il n'y a pas de relation d'ordre définie entre les valeurs de cette dimension.

3 Analyse préliminaire du langage

La première question à se poser en débutant cette étude est celle de l'objectif de la sémantique à construire. Deux alternatives viennent immédiatement à l'esprit. La première est de produire une sémantique de conformité entre contrats. Les auteurs de QML ont en effet accordé une grande importance à cet aspect et ont proposé un ensemble de règles (de type déduction naturelle) permettant de raisonner structurellement sur la conformité entre types de contrats, entre contrats et entre profils. Bien que séduisante, cette approche a le double désavantage d'être plus complexe et moins proche de l'objectif immédiat d'extension de la programmation contractuelle (à la Eiffel [Mey97]) que nous visons. Elle sera cependant utile dans un second temps.

La seconde alternative, que nous avons choisie dans un premier temps, consiste à produire une sémantique dite de vérification, dont l'objectif est de vérifier à l'exécution que les valeurs mesurées selon chaque dimension de qualité de service satisfont bien les contraintes posées par le contrat selon ces différentes dimensions. Ce choix stratégique implique de représenter les dimensions dans les contrats comme des fonctions qui prennent une valeur mesurée (ou un ensemble de valeurs comme nous le verrons) et qui retournent vrai ou faux selon que cette valeur satisfait la contrainte ou non. Un contrat devient donc une entité liant des noms de dimensions à des fonctions que nous appellerons par la suite *fonctions de vérification*. Un contrat est donc structurellement similaire à un environnement classique dans les langages de programmation, c'est-à-dire un ensemble de liaisons identificateurs/valeurs.

Ce choix de représentation pour les contrats établi, il faut en tirer les conséquences au niveau des types de contrats. Quelles sont les informations nécessaires pour définir la fonction de vérification d'une dimension ? Si la contrainte est une contrainte simple, le type de contrat définit les valeurs admissibles et les unités imposées, de même que l'ordre partiel entre les valeurs, s'il y en a un, et la sémantique à associer à cet ordre partiel, c'est-à-dire si les grandes valeurs (`increasing`) ou les petites valeurs (`decreasing`) sont préférables. Du côté du contrat, la contrainte nous indique l'opérateur de contrainte imposé par l'utilisateur et la valeur cible (borne supérieur, borne inférieure ou valeur imposée).

Il apparaît donc assez clairement que le gros de l'information utile pour la vérification est à la disposition du type de contrat et non du contrat. Pour produire la fonction de vérification, nous avons donc le choix entre accéder aux informations définies dans le type de contrat ou encore s'en remettre à ce dernier pour *générer* la fonction de vérification en lui passant l'opérateur et la valeur cible de la contrainte. Nous avons choisi la seconde solution, ce qui entraîne que le type de dimension déclaré dans le type de contrat va être représenté par une fonction dite de *génération*, qui va prendre en argument l'opérateur et la valeur cible de la contrainte et retourner une fonction de vérification qui sera liée à la dimension dans le contrat. On note que le type de contrat est donc essentiellement un ensemble de liaisons entre des noms de dimensions et des fonctions de génération, ce qui encore une fois structurellement similaire à un environnement classique.

Un problème se pose maintenant pour les dimensions contraintes statistiquement. Pour ces contraintes, nous avons choisi de les représenter par des fonctions qui prennent un ensemble de valeurs mesurées et qui retourne vrai ou faux selon que *toutes* les contraintes sur les aspects statistiques sont vérifiées ou non. Pour créer ces fonctions, les échanges d'informations entre le contrat et le type de contrat sont plus complexes que dans le cas des contraintes simples. Le contrat connaît en effet les aspects statistiques contraints (moyenne, variance, fréquences, percentiles) avec leurs différentes valeurs cibles, plages de valeurs, et opérateurs de contrainte. Plutôt que de passer

toute cette information brute au type de contrat, nous avons choisi de passer une fonction de génération pour les aspects statistiques qui prendra la sémantique de la relation définie au niveau du type de contrat puis un ensemble de valeurs brutes, débarassées de leurs unités déjà vérifiées conformes. La fonction de génération du type de dimension installe alors sa vérification d'unités puis passe la sémantique de la relation d'ordre à la fonction de génération d'aspects statistiquement contraints qui va l'utiliser pour vérifier la conformité des opérateurs de contraintes (selon le cas) puis faire la vérification sur l'ensemble de valeurs mesurées.

4 La sémantique dénotationnelle

Après une brève description de la syntaxe abstraite adoptée, nous expliquons de manière détaillée les domaines sémantiques choisis pour exprimer la sémantique de QML, puis nous passons en revue les différentes fonctions sémantiques. Nous ne présentons pas la sémantique dénotationnelle dans ce rapport. Le lecteur est renvoyé au texte classique et encyclopédique de P.D. Mosses [Mos90] qui expose l'essentiel des techniques utilisées ici.

4.1 Syntaxe abstraite

La figure 3 présente la syntaxe abstraite adoptée pour cette étude sémantique de QML. Pour l'essentiel, il s'agit d'un nettoyage de la sémantique concrète en supprimant les terminaux n'ayant pas d'interprétation sémantique intéressante, et en utilisant les séquences de nœuds d'arbres de syntaxe abstraite pour représenter les constructions de la syntaxe concrète visant à construire des séquences (la nature des éléments variant de cas en cas, bien entendu).

4.2 Domaines sémantiques

La figure 4 détaille les domaines sémantiques utilisés dans la présente étude de QML.[2] Le premier niveau d'environnement créé par un programme QML est l'environnement global qui sert à lier les types de contrats, les contrats et les profils aux identificateurs qui leurs sont attribués dans les déclarations. Mais cinq domaines illustrent encore plus clairement le rôle fondamental de QML dans la création d'«ontologies» de qualité de service et donc d'environnements : **Contract**, **ContractType**, **Set**, **POSet** et **ProfileBindings**.

Contract et **ContractType** introduisent la notion de dimension pour représenter, désigner et organiser la variété des aspects de qualité de service que l'on peut être amené à spécifier : temps de réponse, disponibilité, politique de traitement des erreurs, etc. Comme nous l'avons vu, les contrats, les types de contrats et les profils sont en fait eux-mêmes des environnements, puisqu'ils ont pour principale fonction de lier des noms à des valeurs.

Pour les types de contrats (**ContractType**), les noms de dimensions sont liés à des déclarations qui s'apparentent fort à des types de données, limités en QML à des énumérations et à des ensembles de valeurs nommées ainsi qu'à des valeurs numériques, qualifiées ou non par des unités. Les ensembles de valeurs nommées sont représentées par les valeurs de domaines **Set**, s'il n'y a pas de relation d'ordre définie entre ces valeurs, et par les valeurs du domaine **POSet** sinon.

Pour les contrats (**Contract**), les noms de dimension sont liés à des contraintes imposées sur les valeurs mesurées pour ces dimensions. Enfin, les profils lient des noms (au sens large) d'entités dans

[2] La définition d'un domaine attribue un nom au domaine (par exemple, $\mathbf{V}$), qui est lié à une expression de l'algèbre des domaines [Mos90] (par exemple, $\mathbf{Ide} \oplus \mathbf{Num} \oplus \mathbf{Ide}^*$). Le tableau associe également un nom de variable à chaque domaine (par exemple, v). Dans les équations sémantiques, ce nom de variable est utilisé pour désigner toute valeur du domaine correspondant. Il est éventuellement suivi d'un indice si plusieurs valeurs différentes apparaissent dans une même équation. Cette discipline systématique n'a rien à voir avec la sémantique dénotationelle, mais elle est bougrement utile pour comprendre les équations.

Domaines syntaxiques :

p	$\in$	*Program*	ce	$\in$	*ConExp*
d	$\in$	*Decl*	co	$\in$	*Constraint*
t	$\in$	*Test*	dv	$\in$	*DimValue*
ctd	$\in$	*ConTypeDecl*	lit	$\in$	*Literal*
ct	$\in$	*ConType*	a	$\in$	*Aspect*
dim	$\in$	*Dimension*	f	$\in$	*FreqRange*
dt	$\in$	*DimType*	cp	$\in$	*ConstraintOp*
ds	$\in$	*DimSort*	pd	$\in$	*ProfileDecl*
rel	$\in$	*Relation*	pe	$\in$	*ProfileExp*
o	$\in$	*Order*	r	$\in$	*Requisite*
$unit$	$\in$	*Unit*	cel	$\in$	*ContractElem*
b	$\in$	*BaseUnit*	e	$\in$	*Entity*
rs	$\in$	*RelSem*	num	$\in$	*Number*
cd	$\in$	*ConDecl*	i	$\in$	*Identifier*

p	::=	`(program` d^+ t `)`
d	::=	`(decl-type` ctd `)` | `(decl-contract` cd `)` | `(decl-profile` pd `)`
t	::=	`(test-verify` i i dv `)` | `(test-estimate` i i dv^* `)`
ctd	::=	`(contract-type-decl` i ct `)`
ct	::=	`(contract-type` dim^+ `)`
dim	::=	`(dimension` i dt `)`
dt	::=	`(dimtype-simple` ds `)` | `(dimtype-qualified` ds $unit$ `)`
ds	::=	`(dimsort-enum` i^+ `)` | `(dimsort-enum-with` rs i^+ o `)` | `(dimsort-set` rs i^+ `)` |
		`(dimsort-poset` rs i^+ o `)` | `(dimsort-numeric` rs `)`
rel	::=	`(relation` i i `)`
o	::=	`(order` rel^+ `)`
$unit$	::=	`unit-percent` | `(unit-named` b `)`
b	::=	`(baseunit-composed` b i `)` | `(baseunit-id` i `)`
rs	::=	`decreasing` | `increasing`
cd	::=	`(contract-decl` i ce `)`
ce	::=	`(conexp-simple` i co^+ `)` | `(conexp-refinement` i co^+ `)`
co	::=	`(constraint-simple` i cp dv `)` | `(constraint-statistic` i a^+ `)`
dv	::=	`(dimvalue-qualified` lit $unit$ `)` | `(dimvalue-pure` lit `)`
lit	::=	`(lit-simple` i `)` | `(lit-list` i^+ `)` | `(lit-num` num `)`
a	::=	`(aspect-percentile` num cp dv `)` | `(aspect-mean` cp dv `)` |
		`(aspect-variance` cp dv `)` | `(aspect-frequency` f cp dv `)`
f	::=	`(freq-value` dv `)` | `(freq-pair` dv dv `)`
cp	::=	`equal` | `gtequal` | `ltequal` | `gt` | `lt`
pd	::=	`(profile-decl` i i pe `)`
pe	::=	`(profexp-simple` r^+ `)` | `(profexp-refinement` i r^+ `)`
r	::=	`(require` cel^+ `)` | `(from-require` e^+ cel^+ `)`
cel	::=	`(contelem-id` i `)` | `(contelem-exp` ce `)`
e	::=	`(entity-simple` i `)` | `(entity-dotted` i i `)` | `(entity-result-of` i `)`

FIG. 3 – Domaines syntaxiques pour le langage QML

Domaines sémantiques :

		$\mathbf{O}$	$=$	$\{\bot, \top\}$
ν	$\in$	$\mathbf{Ide}$	$=$	*non-spécifié*
n	$\in$	$\mathbf{Num}$	$=$	*non-spécifié*
τ	$\in$	$\mathbf{T}$	$=$	$\{\bot, \mathbf{true}, \mathbf{false}\}$
υ	$\in$	$\mathbf{V}$	$=$	$\mathbf{Ide} \oplus \mathbf{Num} \oplus \mathbf{Ide}^*$
		$\mathbf{UnitMarker}$	$=$	$\{\bot, \mathbf{none}, \mathbf{percent}, \mathbf{named}\}$
		$\mathbf{NamedUnit}$	$=$	$\mathbf{Ide}^*$
μ	$\in$	$\mathbf{DefinedUnit}$	$=$	$\mathbf{UnitMarker} \otimes \mathbf{NamedUnit}$
δ	$\in$	$\mathbf{DimensionValue}$	$=$	$\mathbf{V} \otimes \mathbf{DefinedUnit}$
$\triangleright$	$\in$	$\mathbf{ConstraintOperator}$	$=$	$\{\bot, \mathbf{equal}, \mathbf{lt}, \mathbf{ltequal}, \mathbf{gt}, \mathbf{gtequal}, \mathbf{aspects}\}$
ψ	$\in$	$\mathbf{DimensionTest}$	$=$	$(\mathbf{V} \oplus \mathbf{V}^*) \to \mathbf{T}$
Ψ	$\in$	$\mathbf{ContractDimension}$	$=$	$(\mathbf{DimensionValue} \oplus \mathbf{DimensionValue}^*) \to \mathbf{T}$
ω	$\in$	$\mathbf{Contract}$	$=$	$\mathbf{Env}$
Ω	$\in$	$\mathbf{TypedContract}$	$=$	$\mathbf{Ide} \otimes \mathbf{Contract}$
σ	$\in$	$\mathbf{Set}$	$=$	$\mathbf{Ide}^*$
φ	$\in$	$\mathbf{OrderRelation}$	$=$	$(\mathbf{Ide} \otimes \mathbf{Ide}) \to (\mathbf{T} \oplus \mathbf{O})$
ς	$\in$	$\mathbf{POSet}$	$=$	$\mathbf{Set} \otimes \mathbf{OrderRelation}$
κ	$\in$	$\mathbf{RelationSemantics}$	$=$	$\{\bot, \mathbf{none}, \mathbf{increasing}, \mathbf{decreasing}\}$
α	$\in$	$\mathbf{AspectTest}$	$=$	$\mathbf{RelationSemantics} \to \mathbf{DimensionTest}$
ξ	$\in$	$\mathbf{DimensionSort}$	$=$	$\mathbf{ConstraintOperator} \to$
				$((\mathbf{V} \to \mathbf{DimensionTest})$
				$\oplus(\mathbf{AspectTest} \to \mathbf{DimensionTest}))$
ϕ	$\in$	$\mathbf{DimensionType}$	$=$	$\mathbf{ConstraintOperator} \to$
				$((\mathbf{DimensionValue} \to \mathbf{ContractDimension})$
				$\oplus(\mathbf{AspectTest} \to \mathbf{ContractDimension}))$
Φ	$\in$	$\mathbf{ContractType}$	$=$	$\mathbf{Env}$
		$\mathbf{EntityMarker}$	$=$	$\{\bot, \mathbf{direct}, \mathbf{dotted}, \mathbf{resultof}\}$
ι	$\in$	$\mathbf{InterfaceEntity}$	$=$	$\mathbf{EntityMarker} \times \mathbf{Ide} \times \mathbf{Ide}$
π	$\in$	$\mathbf{ProfileBindings}$	$=$	$\mathbf{InterfaceEntity} \to (\mathbf{Contract}^* \otimes \mathbf{T})$
Δ	$\in$	$\mathbf{DV}$	$=$	$\mathbf{TypedContract} \oplus \mathbf{ContractType} \oplus \mathbf{DimensionType}$
				$\oplus\mathbf{ProfileBindings} \oplus \mathbf{ContractDimension}$
ρ	$\in$	$\mathbf{Env}$	$=$	$\mathbf{Ide} \to (\mathbf{DV} \oplus \mathbf{T})$

FIG. 4 – Domaines sémantiques pour le langage QML

des interfaces à des listes de contrats qui s'appliquent à ces entités. Le domaine **ProfileBindings** est ainsi structurellement similaire à un environnement puisqu'il lie des entités d'interface à des listes de contrats qui s'y appliquent. Ces valeurs sont produites par le traitement des profils.

Il est particulièrement instructif de se rendre compte que les trois principaux éléments définis dans les programmes QML sont en fait représentables sémantiquement par des valeurs qui sont structurellement similaires au domaine classique des environnements, c'est-à-dire des fonctions qui prennent un identificateur et qui retourne une valeur associée à cet identificateur par l'environnement.

Cette partie de QML, aussi instructive qu'elle soit, n'est pas très compliquée. Nous verrons plus loin que le traitement de la syntaxe abstraite dans ces cas consiste simplement à évaluer les valeurs et les lier à l'identificateur donné dans les différents environnements. La principale difficulté viendra des variantes à offrir (un profil prend des entités d'interface et retourne des listes de contrats, or une entité d'interface n'est pas simplement un identificateur comme c'est le cas dans les environnements classiques).

Notons que le domaine environnement introduit dans le tableau est le domaine standard des langages sans effets de bord (sans mémoire modifiable donc). Le domaine **DV** est un domaine dit

«caractéristique»[3] en ce qu'il indique le contenu des environnements dans le langage. En définissant **DV** comme la somme des domaines **TypedContract**, **ContractType**, **DimensionType**, **ProfileBindings** et **ContractDimension**, on indique la richesse des valeurs pouvant être liées à des noms dans le langage QML. Un tel domaine somme peut être vu comme introduisant une certaine «confusion» de types inappropriée en pratique (différents types de valeurs ne se retrouvant jamais dans les mêmes envronnements, pourquoi fusionner tout en un seul domaine ?), mais cette «confusion» sert ici à faire ressortir l'homogénéité sémantique de ces différentes valeurs.[4]

Par ailleurs, **ProfileBindings** apparaît dans **DV** parce que les profils sont liés à des noms par les déclarations de profils. Cependant, ces mêmes **ProfileBindings** n'ont pas été fusionnés aux environnements à cause de la complexité introduite par les entités d'interface qui ne sont pas de simples identificateurs. On n'a donc pas une vue complète de la richesse des environnements de QML par cette définition. Peut-être faudrait-il songer à faire cette fusion pour rendre parfaitement compte de cette richesse, même au prix d'une complexification du domaine **Env**.

Trois domaines dominent par ailleurs la présente sémantique de QML lorsqu'on s'intéresse plus particulièrement à la mécanique du langage : les valeurs de dimension (**DimensionValue**), les dimensions de contrats (**ContractDimension**) et les types de dimension (**DimensionType**).

Les valeurs de dimensions, du domaine **DimensionValue**, apparaissent dans les contrats lorsque ceux-ci mentionnent des bornes dans les contraintes sur les dimensions. Elles sont formées de valeurs brutes ou pures, du domaine **V**, et d'unités. En QML, les différents types de valeurs brutes sont les noms (identificateurs) pour les énumérations et les listes de noms (d'identificateurs) pour les ensembles (avec ou sans relation d'ordre entre les valeurs individuelles) ainsi que les nombres pour les dimensions numériques (muni de leur relation d'ordre standard).

L'objectif que nous avons choisi est de produire des contrats qui sémantiquement soient des ensembles de fonctions de vérification des valeurs mesurées. Cette approche rend les contrats QML proches des contrats à la Eiffel. Les contrats sont donc représentés par des environnements qui vont lier chaque nom de dimension à une fonction prenant une valeur mesurée et retournant vrai ou faux selon que cette valeur satisfait cette dimension du contrat ou non. Cependant, certaines dimensions sont contraintes «statistiquement», sur la moyenne ou la variance des valeurs mesurées. Dans un premier temps nous avons choisi dans ce cas de créer des fonctions qui prennnent une liste de valeurs mesurées et produisent des estimateurs des statistiques contraintes avant de les vérifier par rapport aux valeurs cibles exigées par le contrat. Toutes les fonctions de vérification sont des valeurs des domaines **ContractDimension** et **DimensionTest**. Les fonctions du domaine **ContractDimension** prennent en argument des valeurs mesurées avec leurs unités (si nécessaire), elles vérifient la concordance des unités, puis appellent leur fonction complémentaire du domaine **DimensionTest** qui se contente de comparer les valeurs brutes du domaine **V** (mesurée versus déclarée dans la contrainte).

Le domaine **DimensionType** est défini de telle manière à produire (ou générer) des valeurs du domaine **ContractDimension**, c'est-à-dire des fonctions de vérification. En fait, les types de contrats définissent les noms de dimensions, et pour chacune d'elle les valeurs admissibles, les relations d'ordre et la sémantique qui doit leur être associé. Ces informations sont nécessaires pour construire correctement la fonction de vérification de la dimension sur examen de la contrainte posée dans le contrat. Une fonction du domaine **DimensionType** est donc un générateur de fonctions du domaine **ContractDimension**. Elle prend en paramètre l'opérateur de la contrainte ainsi que la valeur de la borne introduite dans la contrainte et retourne la fonction de vérification. Comme pour le domaine **DimensionTest** par rapport à **ContractDimension**, il existe un

[3]En sémantique dénotationnelle, il est de pratique courante d'utiliser les mêmes noms d'une sémantique à l'autre de manière à faciliter la comparaison de la puissance d'expression des langages étudiés. Ainsi, le domaine **DV**, pour «denotable values», comprend les valeurs pouvant être désignées par des identificateurs dans le langage.

[4]La sémantique proposée plus loin ne s'attarde pas à vérifier effectivement que les différents types d'environnements sont homogènes en termes des valeurs qu'ils contiennent, mais en pratique cette vérification de type devrait faire partie de la sémantique statique du langage.

$\langle \ldots \rangle$	construction de liste
$l \downarrow k$	kth membre de la liste l (partant de 1)
$\#l$	longueur de la liste l
$l \S t$	concaténation des listes l et t
$l \dagger k$	retrait des k premiers membres de la liste l
$x \mid_{\mathrm{D}}$	projection de la valeur x d'un domaine somme quelconque dans sa composante **D**
$in\mathbf{D}(x)$	injection de la valeur x dans le domaine somme **D**
$on_i(x)$	projection de la valeur x d'un domaine produit quelconque dans son $i$ème composante

FIG. 5 – Résumé de la notation utilisée

domaine secondaire **DimensionSort** générateur de la fonction de vérification complémentaire du domaine **DimensionTest** ne considérant que les valeurs brutes (sans unités).

4.3 Fonctions sémantiques

Dans cette sous-section, nous allons passer en revue les principales fonctions sémantiques, en débutant par les fonctions assurant les grands aiguillages de haut niveau entre les domaines syntaxiques, puis en se concentrant sur les trois principaux aspects que sont les types de contrats, les contrats et les profils.

4.3.1 Grands aiguillages

Les grands aiguillages de la sémantique s'organisent autour d'un groupe de trois fonctions sémantiques traitant les programmes et les trois grands types de déclaration que contiennent les programmes QML : déclaration de types de contrats, déclaration de contrats et déclarations de profils.

L'objectif général d'un programme QML est de définir et nommer types de contrats, contrats et profils. La fonction sémantique sur les programmes, $\mathcal{P}$, a donc pour principal résultat un environnement qui va contenir les liaisons entre noms et les grandes sortes d'entités définies. Par ailleurs, la sémantique des contrats étant de type «vérification» de valeurs (i.e. une ou plusieurs valeurs satisfont-elles ou non le contrat), il est utile en pratique d'utiliser ces définitions. Nous avons donc ajouté au langage deux expressions «`test-verify`» et «`test-estimate`» qui permettent de tester des contrats, et donc de retourner une valeur (vrai ou faux) selon que la ou les valeurs satisfont le contrat ou non. La fonction $\mathcal{P}$ commence donc par évaluer les déclarations, puis elle retourne l'environnement obtenu et le résultat de l'expression de vérification :

$$\mathcal{P} :: [\![Program]\!] \rightarrow \mathbf{Env} \otimes \mathbf{T}$$

$$\mathcal{P}[\![\texttt{(program } d^* \ t\texttt{)}]\!] = \ \mathbf{let}\ \rho = \mathcal{DE}^*[\![d^*]\!]\rho_\emptyset\ \mathbf{in}\ (\rho, \mathcal{T}[\![t]\!]\rho)$$

Selon la syntaxe abstraite, un programme est constitué d'une séquence de déclarations suivie d'expressions de vérification. La séquence de déclarations est traitée par la fonction sémantique $\mathcal{DE}^*$ dont le rôle est de traverser la liste de déclarations et de les faire traiter les unes après les autres par la fonction sémantique $\mathcal{DE}$ qui produit dans chaque une liaison d'un nom à un élément (type de contrat, contrat ou profil), lesquelles sont accumulées dans un environnement par combinaison. Lors de l'appel initial via la fonction $\mathcal{P}$, un environnement vide noté $\rho_\emptyset$ est passé en argument, qui va être peuplé de liaisons ici :

$$\mathcal{DE}^* :: [\![Decl^*]\!] \otimes \mathbf{Env} \rightarrow \mathbf{Env}$$

$$\mathcal{DE}^*[\![d^*]\!]\rho = \ \mathbf{if}\ d^* = \langle\rangle\ \mathbf{then}\ \rho\ \mathbf{else}\ \mathcal{DE}^*[\![d^* \dagger 1]\!](\mathcal{DE}[\![d^* \downarrow_1]\!]\rho)\ \mathbf{endif}$$

La fonction $\mathcal{DE}$ fait l'aiguillage entre les trois grands cas de déclarations : les déclarations de types de contrats, les déclarations de contrats et les déclarations de profils. Dans chaque cas, on appelle la fonction sémantique appropriée, en passant l'environnement courant. L'environnement courant est utile dans le cas des contrats pour retrouver les types de contrats auxquels il se rattachent, et dans le cas des profils pour retrouver les contrats auxquels sont astreints les différents éléments d'interface. Dans le cas des contrats, il est également utile de retrouver les définitions de contrats étendus par des expressions de raffinement de contrats.

$\mathcal{DE} :: [\![Decl]\!] \otimes \mathbf{Env} \rightarrow \mathbf{Env}$

$\mathcal{DE}[\![\texttt{(type } ctd \texttt{)}]\!]\rho = \mathcal{CTD}[\![ctd]\!]\rho$

$\mathcal{DE}[\![\texttt{(contract } cd \texttt{)}]\!]\rho = \mathcal{CD}[\![cd]\!]\rho$

$\mathcal{DE}[\![\texttt{(profile } pd \texttt{)}]\!]\rho = \mathcal{PD}[\![pd]\!]\rho$

La fonction $\mathcal{T}$ traite les deux sortes de tests possibles. Dans le premier cas, il s'agit de tester une valeur de dimension mesurée contre la valeur cible de la contrainte définie par le contrat. Syntaxiquement, il faut donc donner le nom du contrat, le nom de la dimension et la valeur mesurée. Avec le nom du contrat, on retrouve dans l'environnement courant le contrat, puis au sein de ce contrat le nom de dimension permet de retrouver cette dimension. Une dimension simple étant représentée par une fonction prenant une valeur de dimension et retournant vrai ou faux selon que cette valeur vérifie le contrat ou non (**ContractDimension**), il suffit de passer la valeur de dimension (d'abord traduite de la syntaxe vers le domaine sémantique **DimensionValue**) à la fonction récupérée. Pour les tests avec estimation, il s'agit de vérifier une dimension dont le contrat est défini par des aspects statistiques descriptives (moyenne, variance, ...). L'idée proposée ici est de passer une liste de valeurs mesurées (**DimensionValue***), à partir de laquelle des estimateurs des différents aspects pourront être calculés et vérifiés par rapport au contrat.[5]

$\mathcal{T} :: [\![Test]\!] \otimes \mathbf{Env} \rightarrow \mathbf{T}$

$\mathcal{T}[\![\texttt{(test-verify } i\ i_1\ dv \texttt{)}]\!]\rho =$
$\quad \mathbf{let*}\ \omega = on_2(bound(\mathcal{I}[\![i]\!], \rho) \mid_{\mathbf{TypedContract}})$
$\quad \mathbf{and}\ \Psi = bound(\mathcal{I}[\![i_1]\!], \omega) \mid_{\mathbf{ContractDimension}}$
$\quad \mathbf{in}\ \Psi(in(\mathbf{DimensionValue} \oplus \mathbf{DimensionValue}^*)(\mathcal{DV}[\![dv]\!]))$

$\mathcal{T}[\![\texttt{(test-estimate } i\ i_1\ dv^* \texttt{)}]\!]\rho =$
$\quad \mathbf{let*}\ \omega = on_2(bound(\mathcal{I}[\![i]\!], \rho) \mid_{\mathbf{TypedContract}})$
$\quad \mathbf{and}\ \Psi = bound(\mathcal{I}[\![i_1]\!], \omega) \mid_{\mathbf{ContractDimension}}$
$\quad \mathbf{in}\ \Psi(in(\mathbf{DimensionValue} \oplus \mathbf{DimensionValue}^*)(\mathcal{DV}^*[\![dv^*]\!]))$

4.3.2 Les types de contrats

Au niveau le plus élevée, les déclarations de types de contrats sont traitées d'abord en calculant le type de contrat lui-même (**ContractType**), puis en liant l'identificateur à cette valeur dans l'environnement courant :

$\mathcal{CTD} :: [\![ConTypeDecl]\!] \rightarrow \mathbf{Env} \rightarrow \mathbf{Env}$

$\mathcal{CTD}[\![\texttt{(contract-type-decl } i\ ct \texttt{)}]\!]\rho =$
$\quad \mathbf{let}\ \Phi = \mathcal{CT}[\![ct]\!]$
$\quad \mathbf{in}\ combine(binding(\mathcal{I}[\![i]\!], in\mathbf{DV}(\Phi)), \rho)$

Deux compositions d'environnements sont utilisées classiquement en sémantique dénotationnelle : *combine* et *overlay*. L'utilisation de *combine* indique ici qu'il ne peut y avoir deux éléments liés au même nom dans l'environnement.

[5] Une optimisation évidente, et surtout une sémantique plus appropriée, consisterait à accepter les valeurs une par une mais à les conserver en mémoire pour la vérification aux appels subséquents. Pour cela, il faudrait une sémantique avec mémoire plus proche d'un langage de programmation impératif.

Un type de contrat est constitué d'une séquence de déclarations de dimensions. Il s'agit donc de traiter les dimensions pour créer un type de contrat (**ContractType**), c'est-à-dire un environnement liant noms de dimension aux fonctions de génération (**DimensionType**) dénotant les types de dimensions. Notons que l'environnement initialement passé à la fonction $\mathcal{D}^*$ est l'environnement vide $\rho_\emptyset$:

$\mathcal{CT} :: [\![ConType]\!] \rightarrow \textbf{ContractType}$

$\mathcal{CT}[\![\texttt{(contract-type}\ dim^*\ \texttt{)}]\!] = \mathcal{D}^*[\![dim^*]\!]\rho_\emptyset$

Le traitement des séquences de déclarations de dimensions est similaire à celui des séquences de déclarations, c'est-à-dire qu'il s'agit de traiter chaque déclaration pour produire une liaison pour chacune, l'ensemble de ces liaisons étant combinées pour former un environnement, le type de contrat.

$\mathcal{D}^* :: [\![Dimension^*]\!] \otimes \textbf{ContractType} \rightarrow \textbf{ContractType}$

$\mathcal{D}^*[\![dim^*]\!]\rho = \ \textbf{if}\ dim^* = \langle\rangle\ \textbf{then}\ \rho\ \textbf{else}\ \mathcal{D}^*[\![dim^* \dagger 1]\!](\mathcal{D}[\![dim^* \downarrow_1]\!]\rho)\ \textbf{endif}$

Une dimension est déclarée par un nom et un type de dimension. Le traitement d'une dimension consiste donc d'abord à évaluer la déclaration de type de dimension, pour donner une valeur du domaine sémantique **DimensionType**, laquelle sera liée à l'identificateur donné (par *binding*) puis composée avec les autres liaisons (par *combine*).

$\mathcal{D} :: [\![Dimension]\!] \otimes \textbf{ContractType} \rightarrow \textbf{ContractType}$

$\mathcal{D}[\![\texttt{(dimension}\ i\ dt\ \texttt{)}]\!]\rho = \ combine(binding(\mathcal{I}[\![i]\!], in\textbf{DV}(\mathcal{DT}[\![dt]\!])), \rho)$

Il existe deux types de dimensions : les types de dimensions avec unités définies et ceux que nous appelons «purs», c'est-à-dire sans unité. Compte tenu de notre décision de représenter un contrat par un environnement de fonctions testant des valeurs mesurées par rapport à des expressions de dimensions, un type de dimension est représenté par une fonction qui, étant données les informations fournies par le contrat, va générer la fonction de vérification correspondante. Nous allons voir un peu plus loin qu'une dimension dans un contrat est définie par une contrainte formée d'un opérateur de contrainte et d'une valeur, comme par exemple `timeToResponse < 10 msec`, sauf pour les dimensions contraintes statistiquement où il s'agit plutôt d'aspects comme la moyenne et la variance des valeurs sur cette dimension. Pour un type de dimension, il s'agit donc d'évaluer les sortes de dimensions puis de produire une fonction qui va accepter un opérateur de contrainte (par exemple '<') puis une valeur de dimension (par exemple `10 msec`) et vérifier que la valeur de la contrainte et la valeur mesurée ont bien la même unité que celle définie par le type de dimension puis, si c'est le cas, passer la valeur (sans unité) à la fonction produite par la sorte de dimension (voir tout de suite après). Si l'opérateur de contrainte «spécial» `aspects` est passé, on sait qu'on a affaire à une dimension contrainte statistiquement et alors la fonction va prendre plutôt une liste de valeurs de dimension mesurées.

Dans les deux cas, la fonction retournée teste d'abord si l'opérateur de contrainte est l'opérateur spécial `aspects`. Si c'est le cas, on va retourner une fonction qui prend un aspect α puis un argument projeté dans les séquences de valeurs de dimensions. On vérifie ensuite la concordance des unités, et si elle est correcte, on appelle la fonction ξ de la sorte de dimension en passant l'opérateur de contrainte, l'aspect α et la séquence de valeurs (brutes, puisque les unités sont devenues inutiles car vérifiées conformes). Si l'opérateur de contrainte n'est pas **aspects**, alors, il suffit de retourner une fonction qui acceptera une valeur projetée dans les valeurs de dimensions (unitaire), et qui va vérifier la concordance des unités puis appeler la fonction ξ.

$\mathcal{DT} :: [\![DimType]\!] \rightarrow \textbf{DimensionType}$

$\mathcal{DT}[\![\texttt{(dimtype-simple}\ ds\ \texttt{)}]\!] =$
$\quad \textbf{let}\ \xi = \mathcal{DS}[\![ds]\!]$
$\quad \textbf{in}\ \lambda\triangleright.\ \textbf{if}\ \triangleright = \textbf{aspects}$
$\qquad\qquad \textbf{then}\ \ in\textbf{ResDT}($
$\qquad\qquad\qquad \lambda\alpha.\lambda\eta.\ \ \textbf{let}\ \delta^* = \eta\ |_{\textbf{DimensionValue*}}$
$\qquad\qquad\qquad\qquad \textbf{in}\ \ \textbf{if}\ all\text{-}true(map(\lambda\delta.on_2(\delta) = (\textbf{none}, \langle\rangle), \delta^*))$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \xi(\triangleright)\ |_{\textbf{FromDSAspectTestCons}}\ (\alpha)(in(\mathbf{V}\oplus\mathbf{V}^*)(map(\lambda\delta.on_1(\delta),\delta^*)))\\
\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \bot_{\mathbf{T}}\\
\qquad\qquad\qquad\qquad\qquad \textbf{endif}\ \ )\\
\qquad \textbf{else}\ \ in\textbf{ResDT}(\\
\qquad\qquad \lambda\delta.\ \ \textbf{if}\ on_2(\delta) = (\textbf{none}, \langle\rangle)\\
\qquad\qquad\qquad \textbf{then}\ \lambda\eta.\ \ \textbf{let}\ \delta_1 = \eta\ |_{\textbf{DimensionValue}}\\
\qquad\qquad\qquad\qquad \textbf{in}\ \ \textbf{if}\ on_2(\delta_1) = on_2(\delta)\\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ \xi(\triangleright)\ |_{\textbf{FromDSCons}}\ (in(\mathbf{V}\oplus\mathbf{V}^*)(on_1(\delta)))(on_1(\delta_1))\\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \bot_{\mathbf{T}}\\
\qquad\qquad\qquad\qquad\quad \textbf{endif}\\
\qquad\qquad\qquad \textbf{else}\ \bot_{\textbf{ContractDimension}}\\
\qquad\qquad\qquad \textbf{endif}\ \ )\\
\qquad \textbf{endif}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{DT}[\![\texttt{(dimtype-qualified}\ ds\ unit\ \texttt{)}]\!] =\\
\quad \textbf{let}\ \xi = \mathcal{DS}[\![ds]\!]\\
\quad \textbf{in}\ \lambda\triangleright.\ \ \textbf{if}\ \triangleright = \textbf{aspects}\\
\qquad\quad \textbf{then}\ \ in\textbf{ResDT}(\\
\qquad\qquad \lambda\alpha.\lambda\eta.\ \ \textbf{let}\ \delta^* = \eta\ |_{\textbf{DimensionValue*}}\\
\qquad\qquad\qquad \textbf{and}\ \mu = \mathcal{U}[\![unit]\!]\\
\qquad\qquad\qquad \textbf{in}\ \ \textbf{if}\ foldl(\lambda(acc,\delta).(acc)\ \wedge\ (on_2(\delta) = \mu), \textbf{true}, \delta^*)\\
\qquad\qquad\qquad\quad \textbf{then}\ \xi(\triangleright)\ |_{\textbf{FromDSAspectTestCons}}\ (\alpha)(in(\mathbf{V}\oplus\mathbf{V}^*)(map(\lambda\delta.on_1(\delta),\delta^*)))\\
\qquad\qquad\qquad\quad \textbf{else}\ \bot_{\mathbf{T}}\\
\qquad\qquad\qquad\quad \textbf{endif}\ \ )\\
\qquad\quad \textbf{else}\ \ in\textbf{ResDT}(\\
\qquad\qquad \lambda\delta.\ \ \textbf{if}\ on_2(\delta) = \mathcal{U}[\![unit]\!]\\
\qquad\qquad\quad \textbf{then}\ \lambda\eta.\ \ \textbf{let}\ \delta_1 = \eta\ |_{\textbf{DimensionValue}}\\
\qquad\qquad\qquad \textbf{in}\ \ \textbf{if}\ on_2(\delta_1) = on_2(\delta)\\
\qquad\qquad\qquad\quad \textbf{then}\ \xi(\triangleright)\ |_{\textbf{FromDSCons}}\ (in(\mathbf{V}\oplus\mathbf{V}^*)(on_1(\delta)))(on_1(\delta_1))\\
\qquad\qquad\qquad\quad \textbf{else}\ \bot_{\mathbf{T}}\\
\qquad\qquad\qquad\quad \textbf{endif}\\
\qquad\qquad\quad \textbf{else}\ \bot_{\textbf{ContractDimension}}\\
\qquad\qquad\quad \textbf{endif}\ \ )\\
\qquad\quad \textbf{endif}
\end{array}
$$

Les fonctions sémantiques $\mathcal{U}$ et $\mathcal{BU}$ traduisent sémantiquement les unités définies syntaxiquement. Elles sont compréhensibles sans plus d'explications.

$\mathcal{U} :: [\![Unit]\!] \rightarrow \textbf{DefinedUnit}$

$\mathcal{U}[\![\texttt{unit-percent}]\!] =\ (\textbf{percent}, \langle\rangle)$

$\mathcal{U}[\![\texttt{(unit-named}\ b\ \texttt{)}]\!] =\ (\textbf{named}, \mathcal{BU}[\![b]\!])$

$\mathcal{BU} :: [\![BaseUnit]\!] \rightarrow \textbf{NamedUnit}$

$\mathcal{BU}[\![\texttt{(baseunit-composed}\ b\ i\ \texttt{)}]\!] =\ \mathcal{BU}[\![b]\!]\S\langle\mathcal{I}[\![i]\!]\rangle$

$\mathcal{BU}[\![\texttt{(baseunit-id}\ i\ \texttt{)}]\!] =\ \langle\mathcal{I}[\![i]\!]\rangle$

Les sortes de dimension définissent le type de valeurs manipulées dans chaque dimension d'un type de contrat. Cinq grandes sortes apparaissent dans QML : les énumérations, avec ou sans ordre entre les éléments, les ensembles, avec ou sans ordre entre les éléments, et enfin les dimensions numériques. Sauf pour les énumérations sans ordre entre éléments, toutes les sortes de dimensions définissent une sémantique relationnelle disant lesquelles des valeurs grandes ou petites sont les plus intéressantes. Avec ces indications, la fonction $\mathcal{DS}$ produit pour chaque dimension une fonction

prenant un opérateur de contrainte et une valeur cible (posés sur la dimension par le contrat) et retourne une fonction acceptant une valeur mesurée et vérifiant si cette valeur satisfait ou non le contrat selon cette dimension.

Des contraintes sémantiques s'appliquent sur la concordance entre la sémantique relationnelle et les opérateurs de contraintes admissibles ; elles sont vérifiées systématiquement. Dans le cas des énumérations sans ordre, le seul opérateur de contrainte admis est l'égalité. Dans la mesure où la valeur donnée dans la contrainte est bien dans l'énumération, le test se limite à vérifier l'égalité de la valeur mesurée avec celle de la contrainte.

$$
\begin{array}{l}
\mathcal{DS} :: [\![DimSort]\!] \rightarrow \mathbf{DimensionSort} \\
\mathcal{DS}[\![\texttt{(dimsort-enum } i^* \texttt{)}]\!] = \\
\quad \mathbf{let}\ \sigma = \mathcal{I}^*[\![i^*]\!] \\
\quad \mathbf{in}\ \lambda \triangleright . in\mathbf{ResDS}(\lambda \varepsilon.\ \mathbf{let}\ \upsilon = \varepsilon \mid_{\mathbf{V}} \\
\qquad\qquad\qquad\qquad \mathbf{in}\ \ \mathbf{if}\,(\triangleright = \mathbf{equal})\ \wedge\ (\upsilon \mid_{\mathbf{Ide}}\ \in \sigma) \\
\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \lambda \upsilon_1 . \upsilon = \upsilon_1 \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \perp_{\mathbf{ContractDimension}} \\
\qquad\qquad\qquad\qquad\quad \mathbf{endif}\ \ )
\end{array}
$$

Pour les énumérations ordonnées, il faut calculer la sémantique de relation *rs*, l'ensemble des valeurs admises par l'énumération i^* et la relation d'ordre o déclarés. Ensuite, la fonction retournée prend en paramètre l'opérateur de contrainte puis la valeur cible de la contrainte et retourne, en fonction de l'opérateur de contrainte et de la sémantique de relation une fonction de vérification. Pour les comparaisons d'inégalité, on s'en remet à la relation d'ordre définie sur la dimension.

$$
\begin{array}{l}
\mathcal{DS}[\![\texttt{(dimsort-enum-with } rs\ i^*\ o \texttt{)}]\!] = \\
\quad \mathbf{let*}\ \sigma = \mathcal{I}^*[\![i^*]\!] \\
\quad \mathbf{and}\ \kappa = \mathcal{RS}[\![rs]\!] \\
\quad \mathbf{and}\ \varphi = \mathcal{O}[\![o]\!]\sigma \\
\quad \mathbf{in}\ \lambda \triangleright . in\mathbf{ResDS}(\lambda \varepsilon.\ \mathbf{let}\ \upsilon = \varepsilon \mid_{\mathbf{V}} \\
\qquad\qquad\qquad\qquad \mathbf{in}\ \ \mathbf{if}\,(\upsilon \mid_{\mathbf{Ide}}\ \in \sigma)\ \wedge\ (compatible(\triangleright, \kappa)) \\
\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \ \mathbf{if}\,\triangleright = \mathbf{equal} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \lambda \upsilon_1 . \upsilon = \upsilon_1 \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{elseif}\ (\kappa = \mathbf{increasing})\ \wedge\ (\triangleright = \mathbf{gt}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \lambda \upsilon_1 . \upsilon \mid_{\mathbf{Ide}} \prec_{\varphi}\ \upsilon_1 \mid_{\mathbf{Ide}} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{elseif}\ (\kappa = \mathbf{increasing})\ \wedge\ (\triangleright = \mathbf{gtequal}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \lambda \upsilon_1 . (\upsilon = \upsilon_1)\ \vee\ (\upsilon \mid_{\mathbf{Ide}} \prec_{\varphi}\ \upsilon_1 \mid_{\mathbf{Ide}}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{elseif}\ (\kappa = \mathbf{decreasing})\ \wedge\ (\triangleright = \mathbf{lt}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \lambda \upsilon_1 . \upsilon_1 \mid_{\mathbf{Ide}} \prec_{\varphi}\ \upsilon \mid_{\mathbf{Ide}} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{elseif}\ (\kappa = \mathbf{decreasing})\ \wedge\ (\triangleright = \mathbf{ltequal}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \lambda \upsilon_1 . (\upsilon = \upsilon_1)\ \vee\ (\upsilon_1 \mid_{\mathbf{Ide}} \prec_{\varphi}\ \upsilon \mid_{\mathbf{Ide}}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \perp_{\mathbf{T}} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{endif} \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \perp_{\mathbf{ContractDimension}} \\
\qquad\qquad\qquad\qquad\quad \mathbf{endif}\ \ )
\end{array}
$$

Pour les ensembles définis sans ordre entre les valeurs, le schéma suivi est similaire au précédent, à ceci près que les comparaisons font appel à l'ordre standard d'inclusion des ensembles.

$$
\begin{array}{l}
\mathcal{DS}[\![\texttt{(dimsort-set } rs\ i^* \texttt{)}]\!] = \\
\quad \mathbf{let}\ \sigma = \mathcal{I}^*[\![i^*]\!] \\
\quad \mathbf{and}\ \kappa = \mathcal{RS}[\![rs]\!] \\
\quad \mathbf{in}\ \lambda \triangleright . in\mathbf{ResDS}(\lambda \varepsilon.\ \mathbf{let}\ \upsilon = \varepsilon \mid_{\mathbf{V}} \\
\qquad\qquad\qquad\qquad \mathbf{in}\ \ \mathbf{if}\,(\upsilon \mid_{\mathbf{Ide*}} \subseteq \sigma)\ \wedge\ (compatible(\triangleright, \kappa)) \\
\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \ \mathbf{let}\ \sigma_1 = \upsilon \mid_{\mathbf{Ide*}} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ \mathbf{if}\,\triangleright = \mathbf{equal}
\end{array}
$$

$$
\begin{array}{l}
\quad\quad\quad\quad\quad\quad \textbf{then}\ \lambda v_1.\sigma_1 = v_1\ |_{\textbf{Ide*}} \\
\quad\quad\quad\quad\quad\quad \textbf{elseif}\ (\kappa = \textbf{increasing})\ \wedge\ (\triangleright = \textbf{gt}) \\
\quad\quad\quad\quad\quad\quad \textbf{then}\ \lambda v_1.\sigma_1 \subset v_1\ |_{\textbf{Ide*}} \\
\quad\quad\quad\quad\quad\quad \textbf{elseif}\ (\kappa = \textbf{increasing})\ \wedge\ (\triangleright = \textbf{gtequal}) \\
\quad\quad\quad\quad\quad\quad \textbf{then}\ \lambda v_1.\sigma_1 \subseteq v_1\ |_{\textbf{Ide*}} \\
\quad\quad\quad\quad\quad\quad \textbf{elseif}\ (\kappa = \textbf{decreasing})\ \wedge\ (\triangleright = \textbf{ltequal}) \\
\quad\quad\quad\quad\quad\quad \textbf{then}\ \lambda v_1.v_1\ |_{\textbf{Ide*}} \subseteq \sigma_1 \\
\quad\quad\quad\quad\quad\quad \textbf{elseif}\ (\kappa = \textbf{decreasing})\ \wedge\ (\triangleright = \textbf{lt}) \\
\quad\quad\quad\quad\quad\quad \textbf{then}\ \lambda v_1.v_1\ |_{\textbf{Ide*}} \subset \sigma_1 \\
\quad\quad\quad\quad\quad\quad \textbf{else}\ \perp_{\textbf{T}} \\
\quad\quad\quad\quad\quad\quad \textbf{endif} \\
\quad\quad\quad \textbf{else}\ \perp_{\textbf{ContractDimension}} \\
\quad\quad\quad \textbf{endif}\ \ )
\end{array}
$$

Pour les ensembles avec un ordre défini sur les éléments, l'ordre entre les valeurs (sous-ensembles) est défini par l'inclusion ensembliste étendu à l'ordre sur les éléments ($\sqsubseteq_\varsigma$), selon la définition de QML, c'est-à-dire soit l'ensemble de valeurs mesurées est sous-ensemble de l'ensemble cible de la contrainte ou encore l'ensemble cible contient des valeurs «majorant» (au sens de la relation d'ordre de la sémantique relationnelle associée) toutes les valeurs de l'ensemble de valeurs mesurées.

$$
\begin{array}{l}
\mathcal{DS}[\![\texttt{(dimsort-poset}\ rs\ i^*\ o\ \texttt{)}]\!] = \\
\quad \textbf{let*}\ \sigma = \mathcal{I}^*[\![i^*]\!] \\
\quad \textbf{and}\ \varphi = \mathcal{O}[\![o]\!]\sigma \\
\quad \textbf{and}\ \varsigma = (\sigma, \varphi) \\
\quad \textbf{and}\ \kappa = \mathcal{RS}[\![rs]\!] \\
\quad \textbf{in}\ \lambda\triangleright.in\textbf{ResDS}(\lambda\varepsilon.\ \ \textbf{let}\ v = \varepsilon\ |_{\textbf{V}} \\
\quad\quad\quad\quad \textbf{in}\ \ \textbf{if}\ (v\ |_{\textbf{Ide*}} \subseteq \sigma)\ \wedge\ (compatible(\triangleright, \kappa)) \\
\quad\quad\quad\quad\quad \textbf{then}\ \ \textbf{let}\ \sigma_1 = v\ |_{\textbf{Ide*}} \\
\quad\quad\quad\quad\quad\quad \textbf{in}\ \ \textbf{if}\ \triangleright = \textbf{equal} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{then}\ \lambda v_1.\sigma_1 = v_1\ |_{\textbf{Ide*}} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{elseif}\ (\kappa = \textbf{increasing})\ \wedge\ (\triangleright = \textbf{gt}) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{then}\ \lambda v_1.\ \ \textbf{let}\ \sigma_2 = v_1\ |_{\textbf{Ide*}} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in}\ (\neg(\sigma_1 = \sigma_2))\ \wedge\ (\sigma_1\ \sqsubseteq_\varsigma\ \sigma_2) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{elseif}\ (\kappa = \textbf{increasing})\ \wedge\ (\triangleright = \textbf{gtequal}) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{then}\ \lambda v_1.\ \ \textbf{let}\ \sigma_2 = v_1\ |_{\textbf{Ide*}} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in}\ \sigma_1\ \sqsubseteq_\varsigma\ \sigma_2 \\
\quad\quad\quad\quad\quad\quad\quad \textbf{elseif}\ (\kappa = \textbf{decreasing})\ \wedge\ (\triangleright = \textbf{ltequal}) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{then}\ \lambda v_1.\ \ \textbf{let}\ \sigma_2 = v_1\ |_{\textbf{Ide*}} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in}\ \sigma_2\ \sqsubseteq_\varsigma\ \sigma_1 \\
\quad\quad\quad\quad\quad\quad\quad \textbf{elseif}\ (\kappa = \textbf{decreasing})\ \wedge\ (\triangleright = \textbf{lt}) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{then}\ \lambda v_1.\ \ \textbf{let}\ \sigma_2 = v_1\ |_{\textbf{Ide*}} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in}\ (\neg(\sigma_1 = \sigma_2))\ \wedge\ (\sigma_2\ \sqsubseteq_\varsigma\ \sigma_1) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{else}\ \perp_{\textbf{T}} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{endif} \\
\quad\quad\quad\quad\quad \textbf{else}\ \perp_{\textbf{ContractDimension}} \\
\quad\quad\quad\quad\quad \textbf{endif}\ \ )
\end{array}
$$

Pour les dimensions numériques finalement, le schéma est encore similaire aux précédents, mais cette fois-ci en utilisant l'ordre standard sur les nombres.

$$
\begin{array}{l}
\mathcal{DS}[\![\texttt{(dimsort-numeric}\ rs\ \texttt{)}]\!] = \\
\quad \textbf{let}\ \kappa = \mathcal{RS}[\![rs]\!] \\
\quad \textbf{in}\ \lambda\triangleright.\ \ \textbf{if}\ \triangleright = \textbf{aspects}
\end{array}
$$

$$
\begin{array}{l}
\textbf{then } in\textbf{ResDS}(\lambda\alpha.\alpha(\kappa)) \\
\textbf{else } in\textbf{ResDS}(\lambda\varepsilon.\ \textbf{let } \upsilon = \varepsilon\mid_{\mathbf{V}} \\
\qquad \textbf{in } \textbf{if}\,(is\textbf{Num}\,?(\upsilon)) \wedge ((\kappa = \textbf{increasing}) \vee (\kappa = \textbf{decreasing})) \\
\qquad\quad \textbf{then } \textbf{if} \rhd = \textbf{equal} \\
\qquad\qquad \textbf{then } \lambda\upsilon_1.\upsilon\mid_{\mathbf{Num}} = \upsilon_1\mid_{\mathbf{Num}} \\
\qquad\qquad \textbf{elseif } (\kappa = \textbf{increasing}) \wedge (\rhd = \textbf{gt}) \\
\qquad\qquad \textbf{then } \lambda\upsilon_1.\upsilon\mid_{\mathbf{Num}} < \upsilon_1\mid_{\mathbf{Num}} \\
\qquad\qquad \textbf{elseif } (\kappa = \textbf{increasing}) \wedge (\rhd = \textbf{gtequal}) \\
\qquad\qquad \textbf{then } \lambda\upsilon_1.(\upsilon\mid_{\mathbf{Num}} = \upsilon_1\mid_{\mathbf{Num}}) \vee (\upsilon\mid_{\mathbf{Num}} < \upsilon_1\mid_{\mathbf{Num}}) \\
\qquad\qquad \textbf{elseif } (\kappa = \textbf{decreasing}) \wedge (\rhd = \textbf{ltequal}) \\
\qquad\qquad \textbf{then } \lambda\upsilon_1.(\upsilon\mid_{\mathbf{Num}} = \upsilon_1\mid_{\mathbf{Num}}) \vee (\upsilon_1\mid_{\mathbf{Num}} < \upsilon\mid_{\mathbf{Num}}) \\
\qquad\qquad \textbf{elseif } (\kappa = \textbf{decreasing}) \wedge (\rhd = \textbf{lt}) \\
\qquad\qquad \textbf{then } \lambda\upsilon_1.\upsilon_1\mid_{\mathbf{Num}} < \upsilon\mid_{\mathbf{Num}} \\
\qquad\qquad \textbf{else } \bot_{\mathbf{T}} \\
\qquad\qquad \textbf{endif} \\
\qquad\quad \textbf{else } \bot_{\mathbf{ContractDimension}} \\
\qquad\quad \textbf{endif}\ \ ) \\
\textbf{endif}
\end{array}
$$

Le traitement des sémantiques de relations est une simple traduction de domaine syntaxique vers domaine sémantique :

$\mathcal{RS} :: [\![RelSem]\!] \rightarrow \textbf{RelationSemantics}$

$\mathcal{RS}[\![\texttt{decreasing}]\!] = \textbf{decreasing}$

$\mathcal{RS}[\![\texttt{increasing}]\!] = \textbf{increasing}$

La sémantique de relation doit être compatible avec l'opérateur de contrainte de la dimension, selon les préceptes de QML. En effet, s'il n'y a pas de relation d'ordre et donc de sémantique de cette relation, l'opérateur de contrainte ne peut qu'être l'opérateur égal. Pour une sémantique de relation *increasing*, les opérateurs de contraintes admissibles sont égal, supérieur ou encore supérieur ou égal. Pour une sémantique de relation *decreasing*, les opérateurs admissibles sont égal, inférieur ou encore inférieur ou égal. La fonction *compatible* vérifie la compatibilité de l'opérateur de contrainte avec une certain sémantique de relation :

$$
\begin{array}{l}
compatible(\rhd, \kappa) = \\
\quad ((\kappa = \textbf{none}) \wedge (\rhd = \textbf{equal})) \vee \\
\quad ((\kappa = \textbf{increasing}) \wedge ((\rhd = \textbf{equal}) \vee (\rhd = \textbf{gt}) \vee (\rhd = \textbf{gtequal}))) \vee \\
\quad ((\kappa = \textbf{decreasing}) \wedge ((\rhd = \textbf{equal}) \vee (\rhd = \textbf{lt}) \vee (\rhd = \textbf{ltequal})))
\end{array}
$$

Une relation d'ordre est définie par une séquence de paires (x, y) indiquant que x est en relation avec y. La relation d'ordre produite est représentée par une fonction (**OrderRelation**) prenant deux identificateurs (**Ide**) et retournant vrai si le premier est en relation avec le second. La fonction $\mathcal{O}$ traite donc la séquence de relation pour produire une telle fonction (à partir de la relation vide représentée par $\varphi_\emptyset$), puis en prend la fermeture transitive.

$\mathcal{O} :: [\![Order]\!] \otimes \textbf{Set} \rightarrow \textbf{OrderRelation}$

$$
\begin{array}{l}
\mathcal{O}[\![\texttt{(order }\kappa\texttt{)}]\!]\sigma = \\
\quad \textbf{let } \varphi = \mathcal{REL}^*[\![\kappa]\!]\varphi_\emptyset \\
\quad \textbf{in } transitive\text{-}closure(\varphi, \sigma)
\end{array}
$$

$\mathcal{REL}^* :: [\![Relation^*]\!] \rightarrow \textbf{OrderRelation} \rightarrow \textbf{OrderRelation}$

$$
\begin{array}{l}
\mathcal{REL}^*[\![\kappa]\!]\varphi = \\
\quad \textbf{if } \kappa = \langle\rangle \\
\quad \textbf{then } \varphi
\end{array}
$$

else $\mathcal{REL}^*[\![\kappa \dagger 1]\!](\mathcal{REL}[\![\kappa{\downarrow_1}]\!]\varphi)$
endif

Chaque paire donne lieu à la création d'une fonction qui prend deux identificateurs et testent si ces identificateurs correspondent aux deux membres de la paire. Si c'est le cas, on retourne vrai, sinon, on retourne indéfini.

$\mathcal{REL} :: [\![Relation]\!] \to \textbf{OrderRelation} \to \textbf{OrderRelation}$

$\mathcal{REL}[\![\texttt{(relation } i\ i_1 \texttt{)}]\!]\varphi =$
$\quad$ **let** $\nu = \mathcal{I}[\![i]\!]$
$\quad$ **and** $\nu_1 = \mathcal{I}[\![i_1]\!]$
$\quad$ **in** $\lambda(\nu_2, \nu_3).$ **if** $(\nu_2 = \nu) \wedge (\nu_3 = \nu_1)$
$\qquad\qquad$ **then** $in\mathbf{T} \oplus \mathbf{O}(\mathbf{true})$
$\qquad\qquad$ **else** **if** $(\nu_3 = \nu) \wedge (\nu_2 = \nu_1)$
$\qquad\qquad\qquad$ **then** $in\mathbf{T} \oplus \mathbf{O}(\mathbf{false})$
$\qquad\qquad\qquad$ **else** $\varphi(\nu_2, \nu_3)$
$\qquad\qquad\qquad$ **endif**
$\qquad\qquad$ **endif**

La fonction $\prec_\varphi$ pour une relation d'ordre φ donnée s'applique entre deux valeurs nommées (identificateurs) et retourne faux si la relation d'ordre n'est pas définie sur une paire d'identificateurs donnée et vrai par contre si les deux identificateurs sont en relation par φ.

4.3.3 Les déclarations de contrats

La déclaration d'un contrat doit mener à la liaison dans l'environnement courant de l'identificateur de contrat à la valeur de contrat elle-même (**TypedContract**). La fonction $\mathcal{CD}$ évalue donc cette valeur de contrat avant de la lier à l'identificateur donné puis de combiner cette nouvelle liaison avec l'environnement courant. On note encore ici l'utilisation de *combine* interdisant d'avoir deux éléments liés au même nom dans l'environnement.

$\mathcal{CD} :: [\![ConDecl]\!] \to \textbf{Env} \to \textbf{Env}$

$\mathcal{CD}[\![\texttt{(contract-decl } i\ ce \texttt{)}]\!]\rho =$
$\quad$ **let** $\omega = \mathcal{CE}[\![ce]\!]\rho$
$\quad$ **in** $combine(binding(\mathcal{I}[\![i]\!], in\mathbf{DV}(\omega)), \rho)$

Un contrat en QML est typé, dans la mesure où il est défini par rapport à une déclaration de type de contrat. Sémantiquement, on représente un contrat typé comme une paire (identificateur de type, contrat). Le contrat lui-même est un environnement liant chaque identificateur de dimension à une fonction de vérification. Il y a deux formes d'expressions de contrats. Le contrat simple est défini à l'aide de l'identificateur du type de contrat et d'une séquence de contraintes. L'expression de raffinement est composée de l'identificateur du contrat à raffiner et d'une séquence de contraintes.

Dans le cas du contrat simple, il suffit de retourner le contrat typé formé du nom de type donné et de l'évaluation des contraintes. Pour le contrat raffiné, il faut utiliser le type du contrat dont l'identificateur est donné, en le recherchant dans l'environnement, et composer les contraintes du contrat à raffiner avec celles ajoutées par l'expression de raffinement.

$\mathcal{CE} :: [\![ConExp]\!] \to \textbf{Env} \to \textbf{TypedContract}$

$\mathcal{CE}[\![\texttt{(conexp-simple } i\ co^* \texttt{)}]\!]\rho =$
$\quad (\mathcal{I}[\![i]\!], \mathcal{C}^*[\![co^*]\!](bound(\mathcal{I}[\![i]\!], \rho) \mid_{\textbf{ContractType}}, \rho_\emptyset, \rho))$

$\mathcal{CE}[\![\texttt{(conexp-refinement } i\ co^* \texttt{)}]\!]\rho =$
$\quad$ **let** $(\nu, \omega) = bound(\mathcal{I}[\![i]\!], \rho) \mid_{\textbf{TypedContract}}$
$\quad$ **in** $(\nu, combine(\mathcal{C}^*[\![co^*]\!](bound(\nu, \rho) \mid_{\textbf{ContractType}}, \omega, \rho), \omega))$

L'évaluation d'une séquence de contraintes vise à retourner un environnement (**Contract**) liant nom de dimension à fonction de vérification. La fonction sémantique $\mathcal{C}^*$ suit donc le schéma déjà utilisé à plusieurs reprises pour traiter chaque contrainte et combiner les liaisons produites au sein d'un environnement retourné comme résultat.

$\mathcal{C}^* :: [\![Constraint^*]\!] \to \mathbf{ContractType} \to \mathbf{Contract} \to \mathbf{Env} \to \mathbf{Contract}$

$\mathcal{C}^*[\![co^*]\!]\Phi\omega\rho =$
- **if** $co^* = \langle\rangle$
- **then** ω
- **else** $\mathcal{C}^*[\![co^* \dagger 1]\!](\Phi, \mathcal{C}[\![co^* \downarrow_1]\!]\Phi\omega\rho, \rho)$
- **endif**

Le traitement des contraintes individuelles s'appuient sur la définition des dimensions dans les types de contrats pour générer les fonctions de vérification : il suffit de récupérer la fonction de génération sur la dimension telle qu'elle est définie dans le type de contrats et de l'appliquer à l'opérateur et à la valeur cible qui apparaissent dans la contrainte. Le résultat de cette génération est lié à l'identificateur de dimension dans le contrat.

$\mathcal{C} :: [\![Constraint]\!] \to \mathbf{ContractType} \to \mathbf{Contract} \to \mathbf{Env} \to \mathbf{Contract}$

$\mathcal{C}[\![\texttt{(constraint-simple } i\ cp\ dv\ \texttt{)}]\!]\Phi\omega\rho =$
- **let∗** $\delta = \mathcal{DV}[\![dv]\!]$
- **and** $\Delta = bound(\mathcal{I}[\![i]\!], \Phi)$
- **and** $\Delta_1 = in\mathbf{DV}(\Delta\ |_{\mathbf{DimensionType}}\ (\mathcal{CO}[\![cp]\!])\ |_{\mathbf{FromDTCons}}\ (\delta))$
- **in** $combine(\omega, binding(\mathcal{I}[\![i]\!], \Delta_1))$

$\mathcal{C}[\![\texttt{(constraint-statistic } i\ a\ \texttt{)}]\!]\Phi\omega\rho =$
- **let∗** $\Delta = bound(\mathcal{I}[\![i]\!], \Phi)$
- **and** $\Delta_1 = in\mathbf{DV}(\Delta\ |_{\mathbf{DimensionType}}\ (\mathbf{aspects})\ |_{\mathbf{FromDTAspectTestCons}}\ (\mathcal{A}^*[\![a^*]\!]\rho))$
- **in** $combine(\omega, binding(\mathcal{I}[\![i]\!], \Delta_1))$

La principale difficulté vient du traitement des contraintes statistiques. En effet, le type de contrat ne connaissant pas la contrainte statistique, il n'est pas possible de générer correctement par le type de contrat les fonctions de vérification de ces contraintes. Il faut donc inverser les rôles ici. Les dimensions statistiquement contraintes génèrent une fonction qui va prendre une liste de valeurs mesurées et vérifier tous les aspects statistiques mentionnées dans la contrainte. La fonction $\mathcal{A}^*$ évalue en séquence chaque aspect et combine les résultats en une seule fonction vérifiant tous les aspects. Dans ce cas, les types de dimensions prennent en paramètre la fonction de vérification obtenue et l'emballent dans une fonction qui va d'abord vérifier la concordance des unités pour les valeurs mesurées avant d'appliquer cette fonction de test.

$\mathcal{A}^* :: [\![Aspect^*]\!] \to \mathbf{Env} \to \mathbf{AspectTest}$

$\mathcal{A}^*[\![a^*]\!]\rho =$
- **if** $a^* = \langle\rangle$
- **then** $\lambda\kappa.\lambda\varepsilon.\mathbf{true}$
- **else** **let** $\alpha = \mathcal{A}[\![a^* \downarrow_1]\!]\rho$
  - **and** $\alpha_1 = \mathcal{A}^*[\![a^* \dagger 1]\!]\rho$
  - **in** $\lambda\kappa.$ **let** $\psi = \alpha(\kappa)$
    - **and** $\psi_1 = \alpha_1(\kappa)$
    - **in** $\lambda\varepsilon.(\psi(\varepsilon))\ \wedge\ (\psi_1(\varepsilon))$
- **endif**

Individuellement, les aspects calculent l'estimateur à partir de l'échantillon de valeurs passées en paramètres et applique la vérification approprié sur le résultat. Pour l'instant, seuls les aspects moyenne et variance sont traités dans la sémantique.

$\mathcal{A} :: [\![Aspect]\!] \to \mathbf{Env} \to \mathbf{AspectTest}$

$$
\begin{array}{l}
\mathcal{A}[\![\texttt{(aspect-mean}\ cp\ dv\ \texttt{)}]\!]\rho = \\
\quad \mathbf{let} \triangleright = \mathcal{CO}[\![cp]\!] \\
\quad \mathbf{and}\ \delta = \mathcal{DV}[\![dv]\!] \\
\quad \mathbf{in}\ \ \mathbf{if}\ is\mathbf{Num}\,?(on_1(\delta)) \\
\qquad \mathbf{then}\ \lambda\kappa.\ \ \mathbf{if}\ compatible(\kappa, \triangleright) \\
\qquad\qquad\qquad \mathbf{then}\ \lambda\varepsilon.\ \ \mathbf{let}\ n = compute\text{-}mean(\varepsilon\ |_{\mathbf{V*}}) \\
\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \ \mathbf{if} \triangleright = \mathbf{equal} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ n = on_1(\delta)\ |_{\mathbf{Num}} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{elseif} \triangleright = \mathbf{gt} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ n > on_1(\delta)\ |_{\mathbf{Num}} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{elseif} \triangleright = \mathbf{gtequal} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ (n = on_1(\delta)\ |_{\mathbf{Num}})\ \vee\ (n > on_1(\delta)\ |_{\mathbf{Num}}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{elseif} \triangleright = \mathbf{lt} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ n < on_1(\delta)\ |_{\mathbf{Num}} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{elseif} \triangleright = \mathbf{ltequal} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ (n = on_1(\delta)\ |_{\mathbf{Num}})\ \vee\ (n < on_1(\delta)\ |_{\mathbf{Num}}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \bot_{\mathbf{T}} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{endif} \\
\qquad\qquad\qquad \mathbf{else}\ \bot_{\mathbf{DimensionTest}} \\
\qquad\qquad\qquad \mathbf{endif} \\
\qquad \mathbf{else}\ \bot_{\mathbf{AspectTest}} \\
\qquad \mathbf{endif}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{A}[\![\texttt{(aspect-variance}\ cp\ dv\ \texttt{)}]\!]\rho = \\
\quad \mathbf{let} \triangleright = \mathcal{CO}[\![cp]\!] \\
\quad \mathbf{and}\ \delta = \mathcal{DV}[\![dv]\!] \\
\quad \mathbf{in}\ \ \mathbf{if}\ is\mathbf{Num}\,?(on_1(\delta)) \\
\qquad \mathbf{then}\ \lambda\kappa.\lambda\varepsilon.\ \ \mathbf{let}\ n = compute\text{-}variance(\varepsilon\ |_{\mathbf{V*}}) \\
\qquad\qquad\qquad \mathbf{in}\ \ \mathbf{if} \triangleright = \mathbf{equal} \\
\qquad\qquad\qquad\quad \mathbf{then}\ n = on_1(\delta)\ |_{\mathbf{Num}} \\
\qquad\qquad\qquad\quad \mathbf{elseif} \triangleright = \mathbf{gt} \\
\qquad\qquad\qquad\quad \mathbf{then}\ n > on_1(\delta)\ |_{\mathbf{Num}} \\
\qquad\qquad\qquad\quad \mathbf{elseif} \triangleright = \mathbf{gtequal} \\
\qquad\qquad\qquad\quad \mathbf{then}\ (n = on_1(\delta)\ |_{\mathbf{Num}})\ \vee\ (n > on_1(\delta)\ |_{\mathbf{Num}}) \\
\qquad\qquad\qquad\quad \mathbf{elseif} \triangleright = \mathbf{lt} \\
\qquad\qquad\qquad\quad \mathbf{then}\ n < on_1(\delta)\ |_{\mathbf{Num}} \\
\qquad\qquad\qquad\quad \mathbf{elseif} \triangleright = \mathbf{ltequal} \\
\qquad\qquad\qquad\quad \mathbf{then}\ (n = on_1(\delta)\ |_{\mathbf{Num}})\ \vee\ (n < on_1(\delta)\ |_{\mathbf{Num}}) \\
\qquad\qquad\qquad\quad \mathbf{else}\ \bot_{\mathbf{T}} \\
\qquad\qquad\qquad\quad \mathbf{endif} \\
\qquad \mathbf{else}\ \bot_{\mathbf{AspectTest}} \\
\qquad \mathbf{endif}
\end{array}
$$

Le traitement des valeurs de dimensions, des litéraux et des opérateurs de contraintes est trivial ; nous laissons au lecteur le soin de lire les définitions des fonctions sémantiques correspondantes.

$$\mathcal{DV}^* :: [\![DimValue^*]\!] \rightarrow \mathbf{DimensionValue}^*$$
$$\mathcal{DV}^*[\![dv^*]\!] = \ \mathbf{if}\ dv^* = \langle\rangle\ \mathbf{then}\ \langle\rangle\ \mathbf{else}\ \langle \mathcal{DV}[\![dv^* \downarrow_1]\!]\rangle \S \mathcal{DV}^*[\![dv^* \dagger 1]\!]\ \mathbf{endif}$$

$$\mathcal{DV} :: [\![DimValue]\!] \rightarrow \mathbf{DimensionValue}$$
$$\mathcal{DV}[\![\texttt{(dimvalue-pure}\ lit\ \texttt{)}]\!] = (\mathcal{L}[\![lit]\!], (\mathbf{none}, \langle\rangle))$$

$$\mathcal{DV}[\![\texttt{(dimvalue-qualified}\ lit\ \mu\ \texttt{)}]\!] = (\mathcal{L}[\![lit]\!], \mathcal{U}[\![unit]\!])$$

$\mathcal{L} :: [\![Literal]\!] \rightarrow \mathbf{V}$

$\mathcal{L}[\![(\texttt{lit-simple}\ i\ )]\!] = in\mathbf{V}(\mathcal{I}[\![i]\!])$

$\mathcal{L}[\![(\texttt{lit-list}\ i^*\ )]\!] = in\mathbf{V}(\mathcal{I}^*[\![i^*]\!])$

$\mathcal{L}[\![(\texttt{lit-num}\ num\ )]\!] = in\mathbf{V}(\mathcal{N}[\![num]\!])$

$\mathcal{CO} :: [\![ConstraintOp]\!] \rightarrow \mathbf{ConstraintOperator}$

$\mathcal{CO}[\![\texttt{equal}]\!] = \mathbf{equal}$

$\mathcal{CO}[\![\texttt{gtequal}]\!] = \mathbf{gtequal}$

$\mathcal{CO}[\![\texttt{ltequal}]\!] = \mathbf{ltequal}$

$\mathcal{CO}[\![\texttt{gt}]\!] = \mathbf{gt}$

$\mathcal{CO}[\![\texttt{lt}]\!] = \mathbf{lt}$

4.3.4 Les déclarations de profils

Comme dans les cas des déclarations de types de contrats et de contrats, le traitement des déclarations de profils consiste à évaluer le profil puis à le lier à l'identificateur donné dans l'environnement. On note encore une fois ici l'utilisation de *combine* interdisant d'avoir deux éléments liés au même nom dans l'environnement.

$\mathcal{PD} :: [\![ProfileDecl]\!] \rightarrow \mathbf{Env} \rightarrow \mathbf{Env}$

$\mathcal{PD}[\![(\texttt{profile-decl}\ i\ pe\ )]\!]\rho =$
  $\mathbf{let}\ \pi = \mathcal{PE}[\![pe]\!]\rho$
  $\mathbf{in}\ combine(binding(\mathcal{I}[\![i]\!], in\mathbf{DV}(\pi)), \rho)$

Un profil contient des liaisons d'éléments d'interface à des contrats. Les éléments d'interface comportent les noms simples (identificateurs), les noms en notation pointée (deux identificateurs liés par un opérateur «point») et le résultat d'une opération de nom donné. Pour les contrats, il peut s'agir soit de noms de contrats préalablement définis ou de nouvelles expressions de contrats. Une expression de profil contient donc deux formes d'énoncés : l'imposition d'une séquence de contrats (`require`) ou l'imposition d'une séquence de contrats à une séquence d'éléments d'interface (`from-require`). Lorsqu'un profil requiert une liste de contrats sans préciser les éléments d'interface, les contrats s'appliquent par défaut à tous les éléments d'interface.

Un profil est donc représenté par une variante d'environnement qui va lier des entités d'interface à des listes de contrats.

$\mathcal{PE} :: [\![ProfileExp]\!] \rightarrow \mathbf{Env} \rightarrow \mathbf{ProfileBindings}$

$\mathcal{PE}[\![(\texttt{profexp-simple}\ r^*\ )]\!]\rho =$
  $\mathcal{R}^*[\![r^*]\!]\rho$

$\mathcal{PE}[\![(\texttt{profexp-refinement}\ i\ r^*\ )]\!]\rho =$
  $\mathbf{let}\ \pi = bound(\mathcal{I}[\![i]\!], \rho)$
  $\mathbf{and}\ \pi_1 = \mathcal{R}^*[\![r^*]\!]\rho$
  $\mathbf{in}\ \lambda\iota.\ \mathbf{case}\ tmp\ = \pi(\iota)\ \mathbf{of}$
    $is\mathbf{T}\ ?\Rightarrow\ \pi_1(\iota),$
    $is\mathbf{ProfileBindings}\ ?\Rightarrow$
      $\mathbf{case}\ tmp_1\ = \pi_1(\iota)\ \mathbf{of}$
      $is\mathbf{T}\ ?\Rightarrow\ tmp,$
      $is\mathbf{ProfileBindings}\ ?\Rightarrow\ in(\mathbf{Contract^*} \oplus \mathbf{T})(tmp\ |_{\mathbf{Contract^*}}\ \S tmp_1\ |_{\mathbf{Contract^*}})$

endcase
endcase

Une clause `require` déclare une liste de contrats devant être observés par toutes les entité d'interface. Elle produit donc une liaison «balai» qui retourne tous ces contrats peu importe l'entité d'interface reçue en paramètre. La clause `from-require` déclare une liste de contrats à être observés par une liste d'entités d'interface. Elle va donc produire une fonction qui va vérifier si l'entité reçue en paramètre fait bien partie de la liste déclarée avant de retourner la liste de contrats. Par ailleurs, le fait de requérir des contrats pour toutes les entités par défaut nous oblige à accumuler les contrats imposés par différentes déclarations au sein d'un même profil (voire d'un profil à raffiner et de l'expression de raffinement). La composition des fonctions doit donc en tenir compte et faire les concaténations de listes de contrats au besoin.

$\mathcal{R}^* :: [\![Requisite^*]\!] \rightarrow \mathbf{Env} \rightarrow \mathbf{ProfileBindings}$

$\mathcal{R}^*[\![r^*]\!]\rho =$
  if $r^* = \langle\rangle$
  then $\lambda\iota.in(\mathbf{Contract^*} \oplus \mathbf{T})(\mathbf{true})$
  else **let** $\pi = \mathcal{R}[\![r^* \downarrow_1]\!]\rho$
    and $\pi_1 = \mathcal{R}^*[\![r^* \dagger 1]\!]\rho$
    in $\lambda\iota.$ **case** $tmp = \pi(\iota)$ **of**
      $is\mathbf{T}\ ? \Rightarrow \pi_1(\iota),$
      $is\mathbf{ProfileBindings}\ ? \Rightarrow$
        case $tmp_1 = \pi_1(\iota)$ **of**
        $is\mathbf{T}\ ? \Rightarrow tmp,$
        $is\mathbf{ProfileBindings}\ ? \Rightarrow in(\mathbf{Contract^*} \oplus \mathbf{T})(tmp\ |_{\mathbf{Contract^*}}\ \S tmp_1\ |_{\mathbf{Contract^*}})$
        endcase
      endcase
  endif

Pour le traitement des clauses individuelles, pour la clause `require`, les éléments de contrats sont évalués et la fonction **ProfileBindings** retournée va prendre en argument une entité d'interface et retourner la liste de contrats évaluées, peu importe l'élément d'interface reçu (ce qui revient à imposer ces contrats à tous les éléments d'interface). Pour la clause `from-require`, la liste d'éléments d'interface est évaluée, puis la liste d'éléments de contrats, puis une fonction **ProfileBindings** est produite en s'inspirant cette fois de la composition de type *combine* dans les environnements, c'est-à-dire qu'un élément d'interface ne peut apparaître deux fois dans une même clause, ce qui paraît une contrainte sémantique judicieuse :

$\mathcal{R} :: [\![Requisite]\!] \rightarrow \mathbf{Env} \rightarrow \mathbf{ProfileBindings}$

$\mathcal{R}[\![$`(require` cel^* `)`$]\!]\rho =$
  let $\omega^* = \mathcal{CEL}^*[\![cel^*]\!]\rho$
  in $\lambda\iota.in(\mathbf{Contract^*} \oplus \mathbf{T})(\omega^*)$

$\mathcal{R}[\![$`(from-require` $e^*\ cel^*$ `)`$]\!]\rho =$
  let $\omega^* = \mathcal{CEL}^*[\![cel^*]\!]\rho$
  in $foldl(\lambda(\pi,\iota).\lambda\iota_1.$ **case** $tmp = \pi(\iota_1)$ **of**
    $is\mathbf{T}\ ? \Rightarrow \lambda\tau.$ **if** $(\tau) \wedge (tmp\ |_{\mathbf{T}})$
      then ω^*
      else tmp
      endif ,
    $is\mathbf{ProfileBindings}\ ? \Rightarrow \lambda\tau.$ **if** τ
      then $in(\mathbf{Contract^*} \oplus \mathbf{T})(\mathbf{false})$
      else tmp
      endif
    endcase $(\iota_1 = \iota), \lambda\iota.in(\mathbf{Contract^*} \oplus \mathbf{T})(\mathbf{true}), \mathcal{E}^*[\![e^*]\!])$

Le traitement des éléments de contrats consiste à retrouver dans l'environnement les contrats à partir de leur nom, ou encore à les créer à partir de l'expression de contrat donnée, puis à retourner la liste de contrats résultante.

$\mathcal{CEL}^* :: [\![ContractElem^*]\!] \rightarrow \mathbf{Env} \rightarrow \mathbf{Contract}^+$

$$\begin{array}{l}
\mathcal{CEL}^*[\![cel^*]\!]\rho = \\
\quad \mathbf{if}\ cel^* = \langle\rangle \\
\quad \mathbf{then}\ \langle\rangle \\
\quad \mathbf{else}\ \langle \mathcal{CEL}[\![cel^* \downarrow_1]\!]\rho \rangle \S \mathcal{CEL}^*[\![cel^* \dagger 1]\!]\rho \\
\quad \mathbf{endif}
\end{array}$$

$\mathcal{CEL} :: [\![ContractElem]\!] \rightarrow \mathbf{Env} \rightarrow \mathbf{Contract}$

$$\begin{array}{l}
\mathcal{CEL}[\![\texttt{(contelem-id}\ i\ \texttt{)}]\!]\rho = \\
\quad \mathbf{let}\ tmp = bound(\mathcal{I}[\![i]\!], \rho) \\
\quad \mathbf{in}\ \ \mathbf{if}\ is\mathbf{DV}\ ?(tmp) \\
\qquad \mathbf{then}\ \ \mathbf{let}\ \Delta = tmp\ |_{\mathbf{DV}} \\
\qquad\qquad \mathbf{in}\ \ \mathbf{if}\ is\mathbf{TypedContract}\ ?(\Delta) \\
\qquad\qquad\quad \mathbf{then}\ on_2(\Delta\ |_{\mathbf{TypedContract}}) \\
\qquad\qquad\quad \mathbf{else}\ \bot_{\mathbf{Contract}} \\
\qquad\qquad\quad \mathbf{endif} \\
\qquad \mathbf{else}\ \bot_{\mathbf{Contract}} \\
\qquad \mathbf{endif}
\end{array}$$

$\mathcal{CEL}[\![\texttt{(contelem-exp}\ ce\ \texttt{)}]\!]\rho = \mathcal{CE}[\![ce]\!]\rho$

Le traitement des entités d'interface est trivial (résultat de la faiblesse de QML concernant la liaison avec les interfaces justement).

$\mathcal{E}^* :: [\![Entity^*]\!] \rightarrow \mathbf{InterfaceEntity}^*$

$\mathcal{E}^*[\![e^*]\!] = \ \mathbf{if}\ e^* = \langle\rangle\ \mathbf{then}\ \langle\rangle\ \mathbf{else}\ \langle \mathcal{E}[\![e^* \downarrow_1]\!] \rangle \S \mathcal{E}^*[\![e^* \dagger 1]\!]\ \mathbf{endif}$

$\mathcal{E} :: [\![Entity]\!] \rightarrow \mathbf{InterfaceEntity}$

$\mathcal{E}[\![\texttt{(entity-simple}\ i\ \texttt{)}]\!] = (\mathbf{direct}, \mathcal{I}[\![i]\!], \bot_{\mathbf{Ide}})$

$\mathcal{E}[\![\texttt{(entity-dotted}\ i\ i_1\ \texttt{)}]\!] = (\mathbf{dotted}, \mathcal{I}[\![i]\!], \mathcal{I}[\![i_1]\!])$

$\mathcal{E}[\![\texttt{(entity-result-of}\ i\ \texttt{)}]\!] = (\mathbf{resultof}, \mathcal{I}[\![i]\!], \bot_{\mathbf{Ide}})$

5 Modélisation objet

Dans cette section, nous proposons une première modélisation objet des contrats à la QML en nous inspirant de la sémantique dénotationnelle précédente. Cette modélisation est réalisée en UML.

5.1 Remarques préliminaires

Notre objectif est de transposer en termes d'objets la sémantique formelle de vérification présentée dans la section précédente. Le passage de la sémantique dénotationnelle à un modèle objet est tout sauf direct. En effet, le λ-calcul, ossature de la partie calculatoire de la sémantique dénotationnelle, tire une grande partie de sa puissance d'une utilisation systématique des fonctions d'ordre supérieur, beaucoup moins usitée dans l'approche objet. *A contrario*, la modélisation objet

s'appuie sur l'héritage, le polymorphisme et, éventuellement, la surcharge des méthodes en fonction des types et du nombre de leurs arguments.

Ainsi, tirer une modélisation objet d'une sémantique dénotationnelle demande de définir une hiérarchie d'héritage pour représenter les données, de répartir les fonctions utilisant de la sélection sur le type des données en méthodes sur les classes correspondantes, et enfin d'utiliser la composition de méthodes par super ou par envoi de messages là où la sémantique dénotationnelle compose des fonctions. Plus spécifiquement, les approximations suivantes sont fort utiles pour passer des domaines aux objets :

- modélisation des domaines sommes par une hiérachie d'héritage où le domaine somme est superclasse de ses composants, eux-mêmes modélisés par des classes représentant les étiquettes de ces derniers,
- modélisation des fonctions comme entités de plein droit par des méthodes sur des classes,
- modélisation des domaines produits par des classes faisant l'agrégation des composants par un nombre de variables d'instance correspondant au nombre de composants du domaine produit,
- introduction de l'héritage là où cette relation existe entre les domaines.

Pour ce qui concerne les domaines sémantiques que nous avons définis pour QML (figure 4), les domaines standards, comme **Ide**, **Num** et **T**, sont directement modélisés par les types chaînes de caractères (`String`), nombres (ici `Real`) et booléens (`Boolean`), que nous suppposerons connus du modèle. Les domaines finis **ConstraintOperator** et **RelationSemantics** se prêtent à une modélisation en type énumération. Le domaine somme **V** représentant les valeurs exprimables donnera lieu à la création d'une classe abstraite pour la somme et de sous-classes concrètes pour les sommandes. Les domaines produits **DimensionValue** et **POSet** vont se voir appliquer la correspondance avec deux classes dont les variables d'instance contiendront les membres des produits.

Par contre, les domaines servant uniquement à marquer des entités, **UnitMarker** et **Entity-Marker**, ne seront plus nécessaires dans la mesure où une bonne modélisation objet doit permettre de les remplacer par l'héritage et la liaison tardive des appels aux méthodes. C'est ainsi que les unités seront représentées par une classe abstraite sous-classée en trois classes concrètes.

Plus généralement, les domaines représentant des fonctions dans notre sémantique doivent abandonner leur statut d'entités de plein droit manipulables par la sémantique, pour être plus simplement représentés par des méthodes sur des classes. Il faudra donc définir des classes déclarant ces méthodes, comme par exemple des classes pour les sortes de dimension, de contraintes et d'aspects. De plus, les fonctions sémantiques font apparaître des traitements différenciés selon les sortes de dimension, les types de contraintes et les types d'aspects statistiques contraints. Par exemple, la fonction sémantique $\mathcal{DS}$ faisant apparaître une sélection sur les types de dimension via la syntaxe abstraite, une hiérarchie de types de dimension sera introduite pour définir ces traitements comme des méthodes spécifiques à chacune des sortes de dimension. Cette voie sera également suivie pour les types de contraintes (voir la fonction sémantique $\mathcal{C}$) et les types d'aspects (voir la fonction sémantique $\mathcal{A}$).

La liaison des contrats aux interfaces par les profils demeurant inaboutie en QML, nous laissons de côté cet aspect du langage dans la modélisation. Nous repoussons une meilleure intégration des contrats aux interfaces aux travaux en perspectives.

Finalement, il restera la question de la représentation des domaines environnements, **Contract** et **ContractType**, sur lesquels nous revenons plus en détails un peu plus loin.

5.2 Le modèle

Compte tenu des éléments d'analyse précédents, nous proposons une modélisation objet de QML en UML [SA98] inspirée de notre sémantique dénotationnelle. Ce modèle est largement

pourvu de contrats fonctionnels, c'est-à-dire invariants de classe, pré- et post-conditions, en OCL [WK99].

5.2.1 Unités de mesure et de valeurs exprimables

Les unités de mesure de QML peuvent être de trois natures différentes : absence d'unité, un pourcentage ou encore une unité nommée. Une unité nommé peut avoir un seul nom ou encore être constituée de plusieurs noms interprétés comme une unité composée, du genre km/s ou encore m/s/s. Les domaines **UnitMarker** et **DefinedUnit** de notre sémantique dénotationnelle servent à distinguer ces trois grandes cas, et dans le cas des unités nommées à représenter la séquence de noms sous la forme d'une liste d'identificateurs.

La représentation objet correspondante consiste simplement à définir une classe abstraite `Unit`, avec trois sous-classes `NamedUnit`, `PercentUnit` et `NoUnit`, représentant respectivement les unités nommées, les pourcentages et l'absence d'unité. Les diagrammes de ces classes apparaissent à la figure 6. La classe abstraite `Unit` définit une seule méthode d'instance, `allWithThisUnit`, qui prend une séquence de valeurs de dimension (voir ci-après) et vérifie si toutes ces valeurs ont pour unité la valeur d'unité réceptrice du message[6] :

```
Unit::allWithThisUnit(dvals : Sequence(DimensionValue)) : Boolean
post : result = dvals->forall(d | d.getUnit() = self)
```

La classe `NamedUnit` déclare une variable d'instance qui contiendra la séquence des noms d'unités. Les trois classes concrètes implantent les méthodes `equals` pour vérifier l'égalité des valeurs d'unité.

```
NamedUnit
invariant : names->notEmpty

NamedUnit::NamedUnit(lst : Sequence(String))
pre  : lst->notEmpty
post : names = lst

NamedUnit::equals(u : Unit) : Boolean
post : u = null implies result = false
post : u <> null and u.oclIsKindOf(NamedUnit) implies
          result = (self.names = u.oclAsType(NamedUnit).names)
```

Pour les classes `PercentUnit` et `NoUnit`, on peut soit les mettre en œuvre comme des classes singleton ou encore s'assurer que la méthode `equals` assimile toutes les instances à une seule et même valeur ainsi (celle de `NoUnit` est similaire) :

```
PercentUnit::equals(u : Unit) : Boolean
post : result = u.oclIsKindOf(PercentUnit)
```

Valeurs exprimables

Les valeurs exprimables dénotées par le domaine somme $\mathbf{V} = \mathbf{Ide} \oplus \mathbf{Ide}^* \oplus \mathbf{Num}$ sont représentées par une classe abstraite `V` et trois sous-classes `IdeOfV`, `SeqIdeOfV` et `NumOfV` représentent les trois opérandes de la somme. Ces trois classes définissent des méthodes `equals`. La classe `IdeOfV` déclare une variable d'instance `ide` pour contenir la chaîne de caractères représentant sa valeur. La classe `SeqIdeOfV` déclare une variable d'instance `ides` qui va contenir une séquence de

[6] Nous déclarons ici comme type de l'argument `Sequence(DimensionValue)`, or `Sequence` est un type du «métalangage» OCL. L'intérêt de ce mélange du langage et du métalangage est de pouvoir utiliser l'opération `forall` définie en OCL dans nos pré- et postconditions. Aucun moyen de médiation entre langage et métalangage n'étant prévu à notre connaissance (voir [WK99]), cette solution, bien que pas totalement satisfaisante, nous donne le moyen de spécifier plus complètement notre modèle.

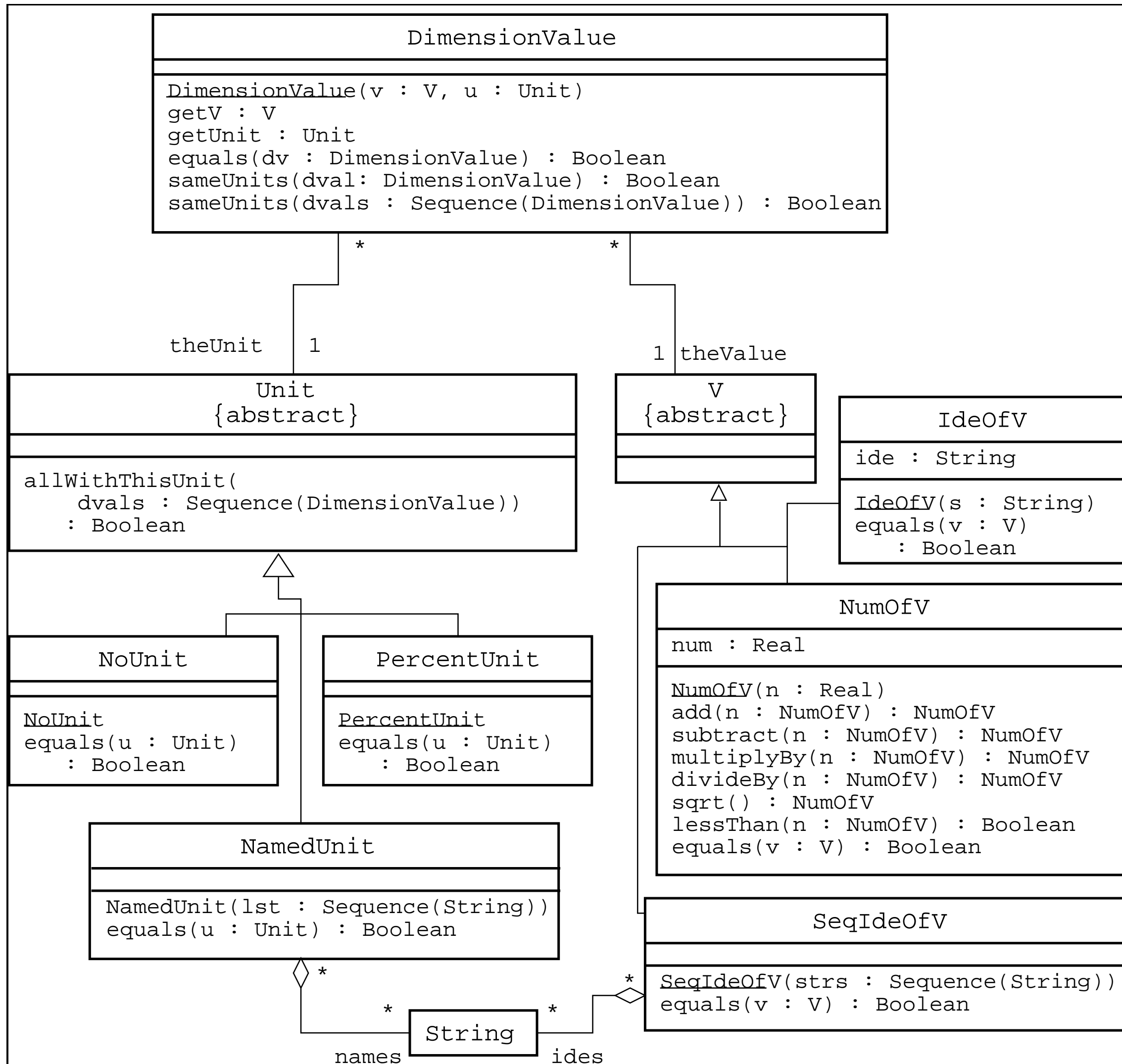

FIG. 6 – Classes de valeurs de dimensions, d'unités de mesure et de valeurs exprimables

noms (chaînes de caractères). L'invariant de la classe `IdeOfV` réclame que que la chaîne `ide` ne soit pas nulle, alors que celui de la classe `SeqIdeOfV` est trivialement vrai.

```
IdeOfV
invariant : ide <> null

IdeOfV::IdeOfV(s : String)
pre  : s <> null
post : ide = s

IdeOfV::equals(v : V) : Boolean
post : v = null implies result = false
post : v <> null and v.oclIsKindOf(self.oclType) implies
          result = (self.ide = v.oclAsType(IdeOfV).ide)

SeqIdeOfV::SeqIdeOfV(strs : Sequence(String))
post : ides = strs
```

```
SeqIdeOfV::equals(v : V) : Boolean
post : v = null implies result = false
post : v <> null and v.oclIsKindOf(self.oclType) implies
          result = (self.ides = v.oclAsType(IdeOfV).ides)
```

La classe `NumOfV`, représentant des nombres, implante des méthodes pour les opérations standards sur les nombres : `add`, `subtract`, `multiplyBy`, `divideBy`, `sqrt`, et `lessThan`. Son invariant est trivialement vrai alors que les contrats sur ses méthodes sont :

```
NumOfV::NumOfV(n : Real)
post : num = n

NumOfV::add(n : NumOfV) : NumOfV
pre  : n <> null
post : result.num - n.num = self.num

NumOfV::subtract(n : NumOfV) : NumOfV
pre  : n <> null
post : result.num + n.num = self.num

NumOfV::multiplyBy(n : NumOfV) : NumOfV
pre  : n <> null
post : result.num = n.num * self.num

NumOfV::divideBy(n : NumOfV) : NumOfV
pre  : n <> null
post : result.num * n.num = self.num

NumOfV::sqrt : NumOfV
pre  : self.num >= 0.0
post : result.num * result.num = self.num

NumOfV::lessThan(NumOfV n) : NumOfV
pre  : n <> null
post : result = (self.num < n.num)

NumOfV::equals(v : V) : Boolean
post : v = null implies result = false
post : v <> null and v.oclIsKindOf(self.oclType) implies
          result = (self.num = v.oclAsType(NumOfV).num)
```

Les valeurs observables sur les dimensions

La classe `DimensionValue` représente les valeurs du domaine sémantique de même nom dans notre sémantique dénotationnelle, composées d'une valeur exprimable (`V`) et d'une unité de mesure (`Unit`). Son invariant impose que sa valeur et son unité ne soient pas nuls :

```
DimensionValue
invariant : theValue <> null and theUnit <> null

DimensionValue::DimensionValue(v : V, u : Unit)
pre  : v <> null and u <> null
post : theValue = v and theUnit = u
```

Les accesseurs `getV` et `getUnit` permettent d'accéder à la valeur et à l'unité :

```
DimensionValue::getV : V
post : result = theValue
```

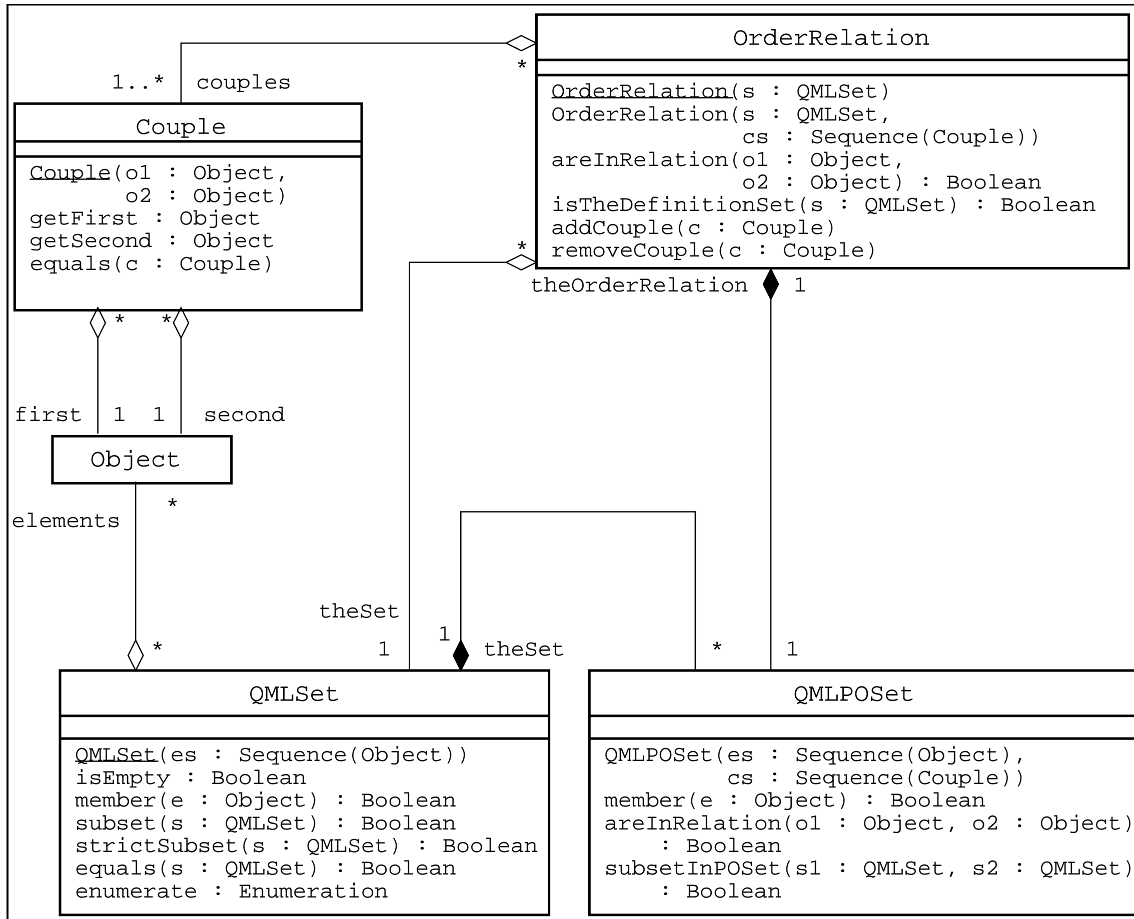

FIG. 7 – Classes d'ensembles

```
DimensionValue::getUnit : Unit
post : result = theUnit
```

La méthode `equals` vérifie l'égalité de deux valeurs de dimension par l'égalité de leurs valeurs et unités :

```
DimensionValue::equals(dval : DimensionValue) : Boolean
post : dval = null implies result = false
post : dval <> null implies result = (self.theValue = dval.theValue and
                                      self.theUnit = dval.theUnit)
```

La méthode `sameUnits(DimensionValue)` se contente de vérifier l'égalité des unités :

```
DimensionValue::sameUnits(dval : DimensionValue) : Boolean
post : dval = null implies result = false
post : dval <> null implies result = (self.theUnit = dval.theUnit)
```

La méthode `sameUnits(Sequence(DimensionValue))`, elle, vérifie que toutes les valeurs de dimension d'une séquence passée en argument ont la même unité que la valeur réceptrice :

```
DimensionValue::sameUnits(dvals : Sequence(DimensionValue)) : Boolean
post : dvals = null implies result = true
post : dvals <> null implies
          result = (dvals->forall(dval | self.theUnit = dval.theUnit))
```

5.2.2 Ensembles, relation d'ordre et ensembles ordonnés

La classe `QMLSet` modélise les ensembles de valeurs. Elle déclare une variable d'instance `elements` qui contiendra les valeurs (le choix d'implantation n'étant pas précisé). Par définition d'un ensemble, les éléments n'y apparaissent qu'une seule fois. Ses principales méthodes sont standards pour les ensembles : `isEmpty` retourne vrai si l'ensemble est vide, `member` vérifie l'appartenance d'un élément à l'ensemble, `subset` et `strictSubset` vérifie respectivement l'inclusion et l'inclusion stricte du receveur dans l'ensemble passé en argument, alors que `equals` vérifie l'égalité de deux ensembles. Par ailleurs, un ensemble peut être énuméré et implante donc une méthode `enumerate` retounant un énumérateur. L'invariant et les contrats sont :

```
QMLSet
invariant : elements->forall(e | elements->count(e) = 1)

QMLSet::QMLSet(es : Sequence(Object))
pre  : es->forall(e | es->count(e) = 1)
post : es->forall(e | self.elements->includes(e))

QMLSet::isEmpty : Boolean
post : result = self.elements->isEmpty

QMLSet::member(e : Object) : Boolean
pre  : element <> null
post : result = self.elements->includes(e)

QMLSet::subset(s : QMLSet) : Boolean
pre  : s <> null
post : result = self.elements->forall(e | s.elements->includes(e))

QMLSet::strictSubset(s : QMLSet) : Boolean
pre  : s <> null
post : result = (self.elements->forall(e | s.elements->includes(e)) and
                 s.elements->exists(e | not self.elements->includes(e)))

QMLSet::equals(s : QMLSet) : Boolean
pre  : s <> null
post : result = (self.subset(s) and s.subset(self))
```

Les relations d'ordre sont définies sur des ensembles de valeurs nommées par un ensemble de couples (a,b) tels que a et b font partie de l'ensemble de définition de la relation et a est en relation avec b. La classe `Couple` est utilisée pour représenter les couples (a,b) nécessaires à la définition des relations d'ordre. L'invariant structurel de cette classe exige simplement que les membres soient non-nuls :

```
Couple
invariant : self.first <> null and self.second <> null
```

Le constructeur répercute au niveau de sa précondition ce qui devra être vrai en terme d'invariant, la postcondition indiquant elle que l'initialisation est faite à partir des arguments :

```
Couple::Couple(o1 : Object, o2 : Object)
pre  : o1 <> null and o2 <> null
post : first = o1 and second = o2
```

Les accesseurs `getFirst` et `getSecond` retournent respectivement le premier et le deuxième membre du couple :

```
Couple::getFirst : Object
post : result = self.first
```

```
Couple::getSecond : Object
post : result = self.second
```

La méthode `equals` définit l'égalité de deux couples comme l'égalité respective de leurs deux membres :

```
Couple::equals(c : Couple) : Boolean
post : o = null implies result = false
post : o <> null implies
          result = (self.first = c.first and self.second = c.second)
```

La classe `OrderRelation` modélise les relations d'ordre. Elles sont constituées d'un ensemble de définition `theSet` instance de la classe `QMLSet` et d'un ensemble de couples de valeurs `couples` dont les valeurs, s'il y en a, appartiennent à l'ensemble de définition. En QML, les couples explicitement spécifiés dans une relation d'ordre ne sont pas les seuls en relation puisque la relation d'ordre doit être comprise comme transitive. La principale méthode définie sur cette classe est `areInRelation`, vérifiant si deux objets sont en relation, l'invariant de la classe et le contrat de cette méthode sont :

```
OrderRelation
invariant : self.couples->forall(c1 |
               self.theSet.member(c1.getFirst()) and
               self.theSet.member(c1.getSecond()) and
               self.couples->forall(c2 |
                  self.areInRelation(c1.getSecond(), c2.getFirst()) implies
                  self.areInRelation(c1.getFirst(), c2.getSecond())))

OrderRelation::areInRelation(o1 : Object, o2 : Object) : Boolean
pre  : o1 <> null and o2 <> null
post : result = (self.couples->exists(c | o1 = c.getFirst() and o2 = c.getSecond())
          or self.couples->exists(c1 |
                self.couples->exists(c2 |
                   o1 = c1.getFirst() and o2 = c2.getSecond()
                   and self.areInRelation(c1.getSecond(), c2.getFirst()))))
```

Deux constructeurs sont prévus, l'un si aucun couple n'est défini et l'autre si des couples initiaux sont définis. Dans les deux cas, nous ne répétons pas en postcondition l'invariant qui doit nécessairement être établi :

```
OrderRelation::OrderRelation(s : QMLSet)
pre  : not s.isEmpty()
post : theSet = s and couples->isEmpty

OrderRelation::OrderRelation(s : QMLSet, cs : Sequence(Couple))
pre  : not s.isEmpty() and cs->notEmpty and
       cs->forall(c | cs->count(c) = 1 and s.member(c.first) and s.member(c.second))
post : theSet = s and couples = cs

OrderRelation::isTheDefinitionSet(s : QMLSet) : Boolean
pre  : s <> null
post : result = (s = theSet)

OrderRelation::addCouple(c : Couple)
pre  : c <> null and theSet.member(c.first) and theSet.member(c.second)
post : couples->includes(c) and couples@pre = couples->excluding(c)

OrderRelation::removeCouple(c : Couple)
pre  : c <> null and theSet.member(c.first) and theSet.member(c.second)
       and couples->includes(c)
post : couples = couples@pre->excluding(c)
```

La classe `QMLPOSet` modélise les ensembles partiellement ordonnés par agrégation d'une instance de `QMLSet` et d'une instance de `OrderRelation`. L'invariant de la classe impose que les deux instances agrégées soient définies et que l'ensemble de définition de la relation d'ordre soit le même que l'ensemble à partir duquel est défini l'ensemble partiellement ordonné :

```
QMLPOSet
invariant : theSet <> null and theOrderRelation <> null and
            theOrderRelation.isTheDefinitionSet(theSet)
```

Lors de l'initialisation d'un ensemble partiellement ordonné, les éléments sont passés comme une séquence et la relation d'ordre est définie par la séquence des couples engendrant la relation d'ordre (pour tenir compte de la transitivité imposée). Les contraintes posées sur ces deux séquences en définissent la cohérence. Outre le fait que la séquence d'éléments de l'ensemble soit non-vide, les éléments doivent être uniques et les éléments mentionnés dans les couples doivent faire partie de la séquence définissant l'ensemble. L'ensemble est alors défini et contient les éléments, alors que le couples correspondent bien à des éléments mis en relation par la relation d'ordre créée.

```
QMLPOSet::QMLPOSet(elements : Sequence(Object), couples : Sequence(Couple))
pre  : elements <> null
pre  : es->forall(e | es->count(e) = 1)
pre  : couples->forall(c1 | elements->include(c1.getFirst()) and
                            elements->include(c1.getSecond()))
post : theSet <> null and theOrderRelation <> null
post : elements->forall(e | theSet.member(e))
post : couples->forall(p |
          theOrderRelation.areInRelation(p.getFirst(), p.getSecond))
```

La méthode `member` répercute tout simplement le message sur l'ensemble agrégé, et la méthode `areInRelation` fait de même pour la relation d'ordre :

```
QMLPOSet::member(e : Object) : Boolean
pre  : e <> null
post : result = theSet.member(e)
```

```
QMLPOSet::areInRelation(e1 : Object, e2 : Object) : Boolean
pre  : e1 <> null and e2 <> null
post : result = theOrderRelation.areInRelation(e1, e2)
```

La méthode `subsetInPOSet` vérifie que deux sous-ensembles d'élements de l'ensemble agrégé par l'ensemble partiellement ordonné sont sous-ensembles au sens définis par QML pour les ensembles partiellement ordonnés, c'est-à-dire que soit on a l'inclusion du premier sous-ensemble dans le second, soit tous les éléments du premier sous-ensemble sont majorés par au moins un élement du second, selon la relation d'ordre défini par sur le receveur :

```
QMLPOSet::subsetInPOSet(QMLSet s1, QMLSet s2) : Boolean
pre  : s1.theSet.elements->forall(e | self.member(e))
pre  : s2.theSet.elements->forall(e | self.member(e))
post : result = (s1.subset(s2) or
                 s1.theSet.elements->forall(e |
                    s2.theSet.elements->exists(e1 | self.areInRelation(e, e1))))
```

5.2.3 Types de dimension, contraintes et aspects

La modélisation des types de dimensions répond au besoin de représenter les valeurs des domaines sémantiques **DimensionType** et **DimensionSort**. Ces valeurs sont en réalité des fonctions dont le rôle dans notre sémantique consiste à vérifier si un ensemble de valeurs de dimension mesurées respectent bien les contraintes émises par le contrat sur ses différentes dimensions. Ce fai-

sant, les fonctions doivent également vérifier la compatibilité des contraintes au type de dimension auquel elles se rapportent.

Certaines de ces règles de compatibilité se rapportent à la sémantique statique et sont donc vérifiables dès la création des contrats. Dans certains cas, les contraintes sont exprimables sous la forme de contrats OCL. Dans d'autres cas, la vérification demeure dynamique.

Évidemment, comme nous l'avons vu dans notre sémantique dénotationnelle, la vérification n'est pas la même selon les types et les sortes de dimension. En sémantique dénotationnelle, la sélection des fonctions sémantiques dirigée par la syntaxe permet d'associer à chaque cas son traitement propre. Comme nous l'avons souligné au début de cette section, en approche objet, c'est la liaison tardive d'un appel de méthode à la méthode exécutée, et donc l'utilisation de l'héritage, qui doit permettre d'atteindre cet objectif. Nous introduisons donc à la figure 8 une classe abstraite `DimensionType` et cinq classes concrètes `Enum`, `Numerical`, `EnumWith`, `SetDimension` et `POSetDimension` pour modéliser respectivement les types de dimension et les sortes de dimensions. Deux classes abstraites, `Ordered` et `NonNumercalWithOrder` sont également introduites pour partager les caractéristiques communes entre les sortes de dimension.

D'un point de vue structurel, la classe `DimensionType` possède une variable `declaredUnit` qui va contenir l'unité de mesure déclarée pour le type de dimension. L'invariant de cette classe se borne à exiger que la valeur d'unité déclarée soit bien initialisée, ce qui se répercute tel quel dans le contrat du contructeur :

```
DimensionType
invariant : declaredUnit <> null
```

```
DimensionType::DimensionType(u : Unit)
pre  : u <> null
post : declaredUnit = du
```

La classe `DimensionType` déclare trois méthodes `compatible` dont le rôle est de vérifier les règles «statiques» de conformité de la contrainte posée sur une dimension à la déclaration de type de dimension. La méthode abstraite `compatible(ConstraintOperator) : Boolean` vérifie la compatibilité de l'opérateur de la contrainte déclarée dans le contrat avec la sorte de dimension. Par exemple, elle vérifiera qu'une contrainte sur une sorte de dimension `Enum` comporte nécessairement un opérateur de contrainte d'égalité. Son contrat est trivialement vrai. La méthode concrète `compatible(DimensionValue) : booolean` vérifie que la valeur de dimension cible de la contrainte est de l'unité de mesure déclarée par le type de dimension et que la valeur exprimable associée est elle aussi compatible :

```
DimensionType::compatible(target : DimensionValue) : Boolean
pre  : target <> null
post : result = (target.getUnit() = declaredUnit and self.compatible(dval.getV()))
```

La méthode abstraite `compatible(V) : booolean` vérifie qu'une valeur exprimable cible de la contrainte fait partie des valeurs admissibles par la sorte de dimension :

```
DimensionType::compatible(measured : V) : Boolean
pre  : measured != null
```

Les quatre méthodes `verify` sont utilisées pour vérifier si les valeurs mesurées selon la dimension respectent la contrainte posée sur la dimension. La méthode concrète `verify(DimensionValue) : Boolean` vérifie qu'une valeur de dimension mesurée est de l'unité de mesure déclarée par le type de dimension et que la valeur exprimable associée est admissible :

```
DimensionType::verify(dval : DimensionValue) : Boolean
pre  : dval <> null
post : result = (dval.getUnit() = declaredUnit and self.compatible(dval.getV()))
```

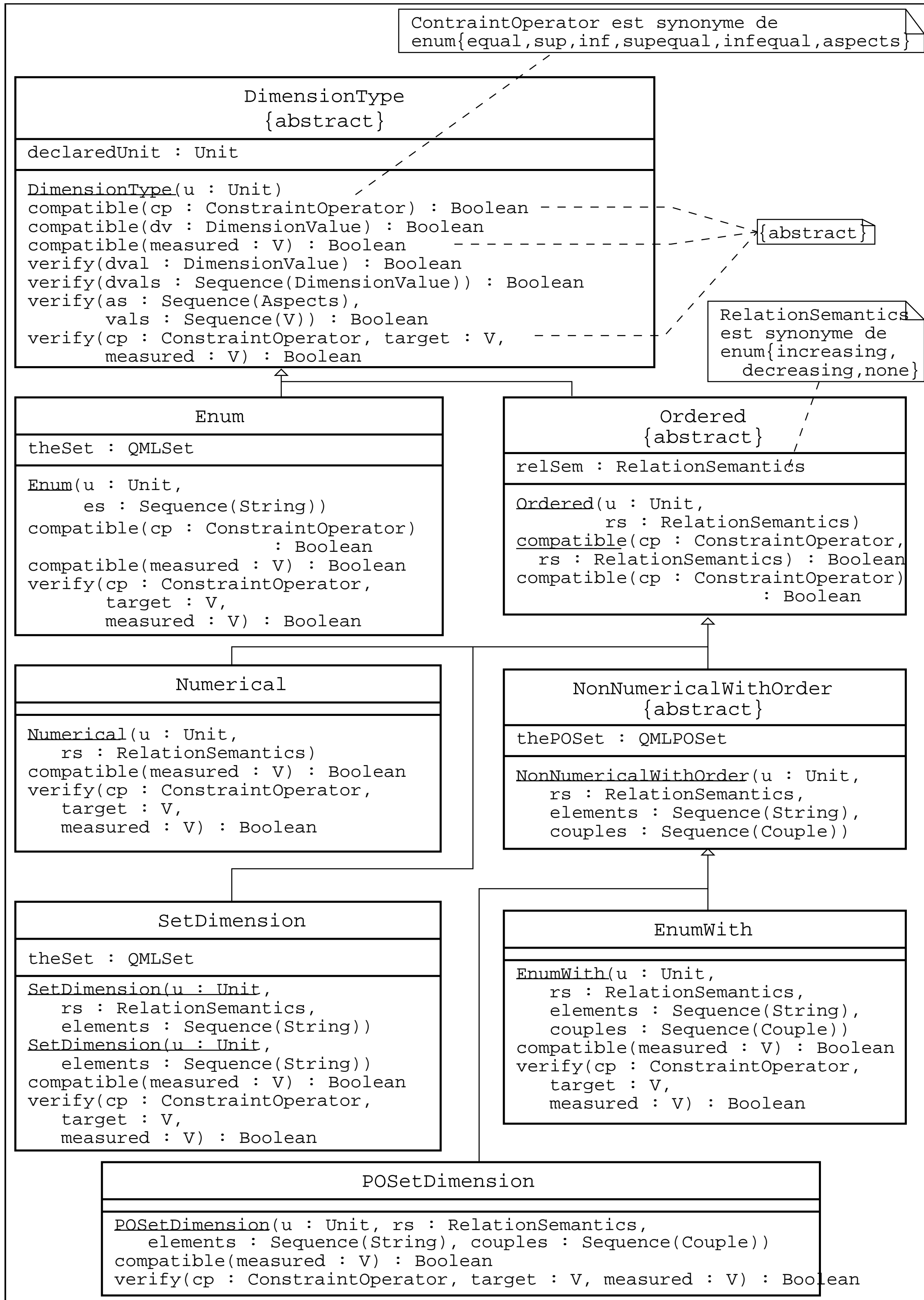

FIG. 8 – Classes de types de dimensions

La méthode concrète `verify(Sequence(DimensionValue)) : Boolean` fait de même pour une liste de valeurs de dimension mesurées (cas des sortes de dimensions contraintes selon des aspects statistiques) :

```
DimensionType::verify(dvals : Sequence(DimensionValue)) : Boolean
post : result = dvals->forall(dval | self.compatible(dval))
```

La méthode concrète `verify(Sequence(Aspects), Sequence(V)) : Boolean` sert à vérifier si l'ensembles des aspects contraints statistiquement selon une dimension sont vérifiés par les valeurs exprimables mesurées :

```
DimensionType::verify(as : Sequence(Aspects), vals : Sequence(DimensionValue)) : Boolean
pre  : vals->forall(val | self.compatible(val))
post : as->forall(a | a.verify(vals))
```

La méthode abstraite `verify(ConstraintOperator,V,V) : Boolean` vérifie si une valeur exprimable mesurée respecte la contrainte définie par un certain opérateur de contrainte et par une valeur exprimable cible :

```
DimensionType::verify(cp : ConstraintOperator, target : V, measured : V) : Boolean
pre  : cp <> null and self.compatible(cp)
pre  : target <> null and self.compatible(target)
pre  : measured <> null and this.compatible(measured)
```

Nous verrons un peu plus loin comment ces méthodes coopèrent pour vérifier la compatibilité et le respect des contraintes posées sur les dimensions de contrats.

Les sortes de dimension

La classe `Enum` représente les dimensions définies simplement par un ensemble de valeurs énumérées sans relation d'ordre entre elles. Les valeurs admissibles sont données par l'ensemble des valeurs énumérées, qui doit être défini :

```
Enum
invariant : theSet <> null
```

Le constructeur prend en compte l'unité de mesure déclarée :

```
Enum::Enum(du : Unit, elements : Sequence(String))
pre  : du <> null
pre  : elements <> null and elements->forall(e | elements->count(e) = 1)
post : declaredUnit = du
post : elements->forall(e | theSet.member(new IdeOfV(e)))
```

Le seul opérateur de contrainte compatible avec une dimension énumération est l'opérateur d'égalité :

```
Enum::compatible(cp : ConstraintOperator) : Boolean
pre  : cp <> null
post : result = (cp = #equal)
```

Une valeur mesurée est compatible avec la dimension énumération uniquement si c'est une valeur exprimable de type `IdeOfV` et que celle-ci fait partie de l'ensemble des valeurs énumérées :

```
Enum::compatible(measured : V) : Boolean
pre  : measured <> null
post : result = measured.oclIsKindOf(IdeOfV) and theSet.member(measured)
```

La vérification de la contrainte consiste simplement à vérifier si la valeur mesurée est égale à la valeur cible, si par ailleurs toutes les règles de compatibilité sont observées :

```
Enum::verify(cp : ConstraintOperator, target : V, measured : V) : Boolean
pre  : cp <> null and this.compatible(cp)
pre  : target <> null and this.compatible(target)
pre  : measured <> null and this.compatible(measured)
post : result = (target = measured)
```

La classe abstraite `Ordered` introduit la notion d'ordre entre les valeurs et par conséquent de la sémantique de cette relation d'ordre. Elle possède donc une variable d'instance `relsem` :

```
Ordered
invariant : relsem <> null
```

```
Ordered::Ordered(du : Unit, rs : RelationSemantics)
pre  : du <> null and rs <> null
post : relSem = rs and declaredUnit = du
```

La compatibilité d'un opérateur de contrainte est alors définie en fonction de la sémantique de relation d'ordre choisie, comme nous l'avons vu. Une méthode de classe définit la compatibilité entre un opérateur de contrainte et une sémantique de relation. La méthode `compatible` pour un opérateur de contrainte appelle cette méthode de classe :

```
Ordered::compatible(cp : ConstraintOperator, rs : RelationSemantics) : Boolean
post : result = (cp = #equal
                or ((cp = #sup or cp = #supequal) and rs = #increasing)
                or ((cp = #inf or cp = #infequal) and rs = #decreasing))
```

```
Ordered::compatible(cp : ConstraintOperator) : Boolean
post : result = Ordered.compatible(cp, relsem)
```

La classe `Numerical` est une sous-classe directe de `Ordered` et définit les dimensions dont les valeurs sont numériques. En terme structurel, elle n'ajoute rien de plus à `Ordered`. Si bien que seuls les contrats des méthodes `compatible` et `verify` sont intéressants. Une valeur exprimable mesurée est compatible avec cette sorte de dimension si elle est de type `NumOfV` :

```
Numerical::compatible(measured : V) : Boolean
pre  : measured <> null
post : result = measured.oclIsKindOf(NumOfV)
```

alors que la vérification d'une valeur mesurée, outre les vérification de compatibilités, procède par analyse de cas en fonction de l'opérateur de contrainte et de la sémantique de la relation d'ordre :

```
Numerical::verify(cp : ConstraintOperator, target : V, measured : V) : Boolean
pre  : cp <> null and self.compatible(cp)
pre  : target <> null and self.compatible(target)
pre  : measured <> null and self.compatible(measured)
post : cp = #equal implies
          result = (target.oclAsType(NumOfV) = measured.oclAsType(NumOfV))
post : relSem = #increasing and cp = #sup implies
          result = target.oclAsType(NumOfV).lessThan(measured.oclAsType(NumOfV))
post : relSem = #increasing and cp = #supequal implies
          result = (target.oclAsType(NumOfV).equals(measured.oclAsType(NumOfV))
             or target.oclAsType(NumOfV).lessThan(measured.oclAsType(NumOfV)))
post : relSem = #decreasing and cp = #inf implies
          result = measured.oclAsType(NumOfV).lessThan(target.oclAsType(NumOfV))
post : relSem = #decreasing and cp = #infequal implies
          result = (target.oclAsType(NumOfV).equals(measured.oclAsType(NumOfV))
             or measured.oclAsType(NumOfV).lessThan(target.oclAsType(NumOfV)))
```

La classe concrète `SetDimension` est aussi sous-classe directe de la classe `Ordered`, mais ses valeurs sont des ensembles de noms faisant partie d'une énumération donnée à la création du type de dimension. Elle possède donc une variable `theSet` qui contient l'ensemble des valeurs admissibles dans ces ensembles de noms :

```
SetDimension
invariant : theSet <> null
```

```
SetDimension::SetDimension(du : Unit, rs : RelationSemantics, elements : Sequence(String))
pre  : du <> null
pre  : rs <> null
pre  : elements <> null and elements->forall(e | elements->count(e) = 1)
post : elements->forall(e | theSet.member(new IdeOfV(e)))
```

Une valeur expressible mesurée est compatible avec une telle dimension si elle est une instance de la classe `SeqIdeOfV` des séquences de noms, et si l'ensemble des valeurs ainsi formées est un sous-ensemble de l'ensemble des valeurs admissibles `theSet` :

```
SetDimension::compatible(measured : V) : Boolean
pre  : measured <> null
post : result = measured.oclIsKindOf(SeqIdeOfV) and
         (new QMLSet(measured.oclAsType(SeqIdeOfV).ides)).subset(theSet)
```

La vérification qu'une valeur mesurée respecte la contrainte se fait encore une fois par analyse de cas :

```
SetDimension::verify(cp : ConstraintOperator, target : V, measured : V) : Boolean
pre  : cp <> null and self.compatible(cp)
pre  : target <> null and self.compatible(target)
pre  : measured <> null and self.compatible(measured)
post : let targetSet = new QMLSet(target.oclAsType(SeqIdeOfV).ides)
       and measuredSet = new QMLSet(measured.oclAsType(SeqIdeOfV).ides)
       in cp = #equal and result = (targetSet = measuredSet)
          or relSem = #increasing and cp = #sup and
             result = targetSet.strictSubset(measuredSet)
          or relSem = #increasing and cp = #supequal and
             result = targetSet.subset(measuredSet)
          or relSem = #decreasing and cp = #inf and
             result = measuredSet.strictSubset(targetSet)
          or relSem = #decreasing and cp = #infequal and
             result = measuredSet.subset(targetSet)
```

La classe abstraite `NonNumericalWithOrder` est sous-classe directe de `Ordered` et regroupe les caractéristiques communes aux classes `EnumWith` et `POSetDimension` qui définissent les sortes de dimensions impliquant des valeurs énumérées avec relation d'ordre. Cette classe possède donc un ensemble de valeurs avec relation d'ordre, c'est-à-dire une instance de `QMLPOSet` avec les règles d'unicité des valeurs, d'appartenance des valeurs des couples aux éléments de l'ensemble et de transitivité de la relation d'ordre définie :

```
NonNumericalWithOrder
invariant : thePOSet <> null
```

```
NonNumericalWithOrder::NonNumericalWithOrder(du : Unit,
       rs : RelationSemantics, elements : Sequence(String),
       couples : Sequence(Couple))
pre  : du <> null
pre  : rs <> null
pre  : elements->forall(e | elements->count(e) = 1)
```

```
pre  : couples->forall(c |
          couples->count(c) = 1 and
          elements->includes(c.getFirst()) and
          elements->includes(c.getSecond()))
post : thePOSet <> null and
       elements->forall(e | thePOSet.theSet.member(new IdeOfV(e))) and
       couples->forall(c |
             thePOSet.areInRelation(
                new IdeOfV(c.getFirst().oclAsType(String)),
                new IdeOfV(c.getSecond().oclAsType(String)))) and
       couples->forall(c1 |
          couples->forall(c2 |
             thePOSet.areInRelation(
                new IdeOfV(c1.getSecond().oclAsType(String)),
                new IdeOfV(c2.getFirst().oclAsType(String)))
             implies
                thePOSet.areInRelation(
                   new IdeOfV(c1.getFirst().oclAsType(String)),
                   new IdeOfV(c2.getSecond().oclAsType(String)))))
post : relSem = rs
post : declaredUnit = du
```

La classe concrète `EnumWith` est une sous-classe directe de `NonNumericalWithOrder` qui n'ajoute structurellement rien à cette dernière. Par contre, elle définit les méthodes `compatible(V)` et `verify(ConstraintOperator, V, V)`. Une valeur mesurée est compatible avec une dimension énumération avec relation d'ordre si elle est instance de la classe `IdeOfV` et si elle fait partie de l'énumération :

```
EnumWith::compatible(measured : V) : Boolean
pre  : measured <> null
post : result = measured.oclIsKindOf(IdeOfV) and thePOSet.member(measured)
```

La vérification du respect de la contrainte par une valeur mesurée se fait à nouveau par analyse de cas en fonction de l'opérateur de contrainte et la sémantique de relation :

```
EnumWith::verify(cp : ConstraintOperator, target : V, measured : V) : Boolean
pre  : cp <> null and self.compatible(cp)
pre  : target <> null and self.compatible(target)
pre  : measured <> null and self.compatible(measured)
post : cp = #equal implies result = (target = measured)
post : relSem = #increasing and cp = #sup implies
          result = thePOSet.areInRelation(target, measured)
post : relSem = #increasing() and cp = #supequal implies
          result = (target = measured or
             thePOSet.areInRelation(target, measured))
post : relSem = #decreasing and cp = #inf implies
          result = thePOSet.areInRelation(measured, target)
post : relSem = #decreasing and cp = #infequal implies
          result = (target = measured or
             thePOSet.areInRelation(measured, target))
```

Finalement, la classe `POSetDimension` s'inspire à la fois de la classe `SetDimension` et de la classe `EnumWith`. De la première, elle reprend le fait que ses valeurs sont des ensembles, et de la seconde qu'une relation d'ordre est définie entre ces valeurs. Par rapport à sa superclasse directe `NonNumericalWithOrder`, `POSetDimension` n'ajoute rien structurellement parlant.

Une valeur mesurée est compatible avec cette sorte de dimension si elle est une instance de la classe `SeqIdeOfV` et si toutes les valeurs de cette séquence font partie de l'ensemble des valeurs admissibles :

```
POSetDimension::compatible(measured : V) : Boolean
pre  : measured <> null
post : result = measured.oclIsKindOf(SeqIdeOfV) and
         (new QMLSet(measured.oclAsType(SeqIdeOfV).ides)).subset( thePOSet.theSet)
```

La vérification du respect de la contrainte par une valeur mesurée se fait à nouveau par analyse de cas en fonction de l'opérateur de contrainte et la sémantique de relation :

```
POSetDimension::verify(cp : ConstraintOperator, target : V, measured : V) : Boolean
pre  : cp <> null and self.compatible(cp)
pre  : target <> null and self.compatible(target)
pre  : measured <> null and self.compatible(measured)
post : let targetSet := new QMLSet(target.oclAsType(SeqIdeOfV).ides)
       and measuredSet := new QMLSet(measured.oclAsType(SeqIdeOfV).ides)
       in cp = #equal and result = targetSet = measuredSet
          or relSem = #increasing and cp = #sup and
                result = not (targetSet = measuredSet) and
                   thePOSet.subsetInPOSet(targetSet, measuredSet)
          or relSem = #increasing and cp = #supequal and
                result = thePOSet.subsetInPOSet(targetSet, measuredSet)
          or relSem = #decreasing and cp = #inf and
                result = not (targetSet = measuredSet) and
                   thePOSet.subsetInPOSet(measuredSet, targetSet)
          or relSem = #decreasing and cp = #infequal() and
                result = thePOSet.subsetInPOSet(measuredSet, targetSet)
```

Contraintes et aspects statistiquement contraints

Si un type de contrat est composé de déclarations de types et de sortes de dimension, le contrat lui est composé de contraintes posées sur ces dimensions. Une contrainte peut être simple, et formée d'un opérateur de contrainte et d'une valeur cible. Une contrainte peut porter sur des aspects statistiques de la dimension, moyenne et variance des valeurs mésurées (fréquences et valeurs au percentiles également, mais non-modélisé pour l'instant). Les contraintes sont donc modélisées par une classe abstraite `Constraint` et deux sous-classes concrètes `SimpleConstraint` et `StatisticalConstraint`. Les diagrammes de ces classes apparaissent à la figure 9.

À la base, une contrainte se définit par rapport au type de la dimension sur laquelle elle est posée, qu'elle doit donc connaître. La classe `Constraint` déclare donc une variable d'instance `dimType` qui contient l'objet type de dimension auquel elle correspond. Au niveau de la classe abstraite, l'invariant n'impose guère que cet objet soit défini :

```
Constraint
invariant : dimType <> null
```

```
Constraint::Constraint(dt : DimensionType)
pre  : dt <> null
post : dimType = dt
```

Une contrainte simple est définie par son opérateur de contrainte et par sa valeur cible. La classe `SimpleConstraint` déclare donc deux variables d'instance `oper` et `targetValue` pour contenir ces informations. Les valeurs de ces deux variables doivent être compatible avec le type de la dimension sur laquelle la contrainte est posée :

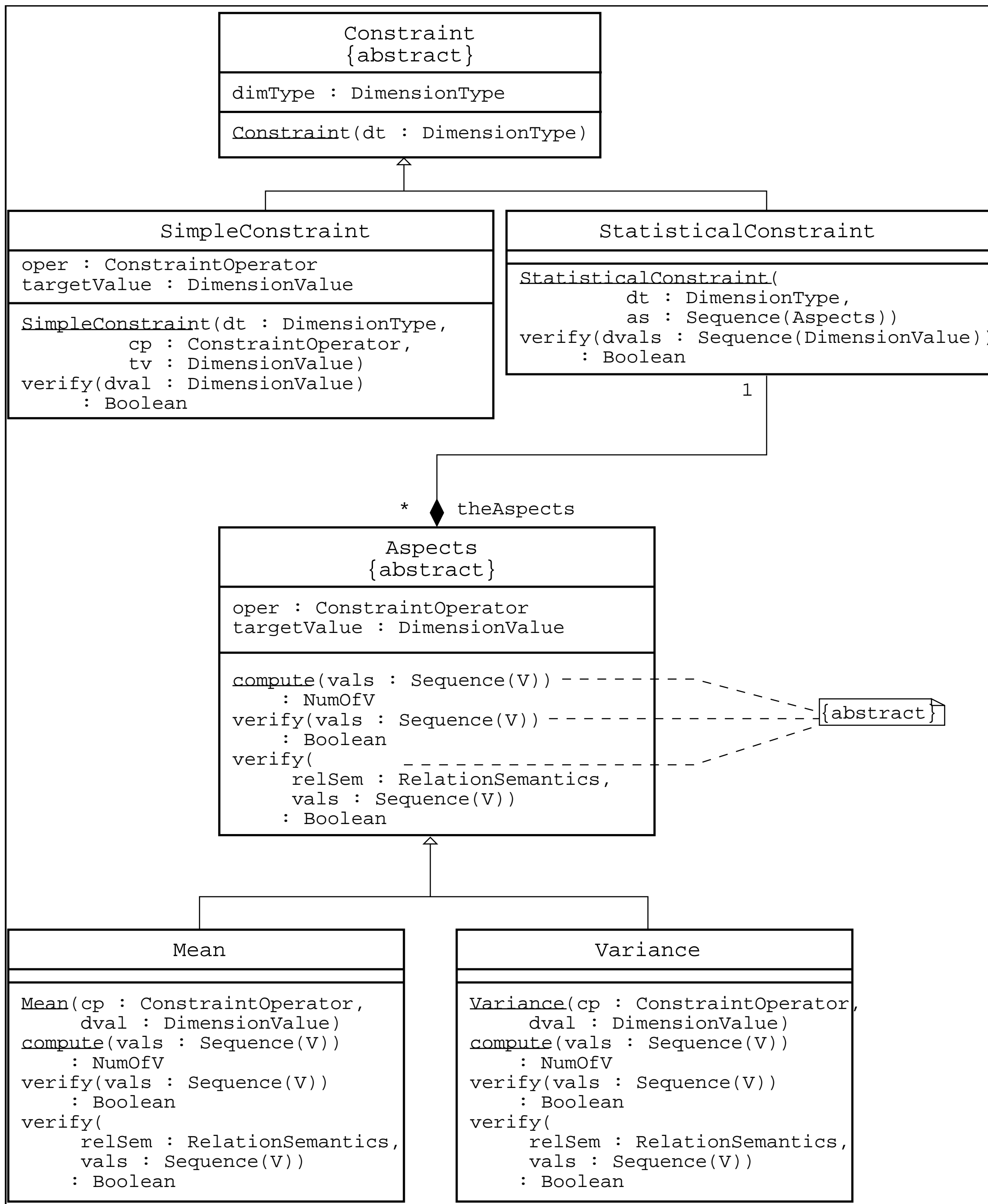

FIG. 9 – Classes de contraintes et d'aspects

```
SimpleConstraint
invariant : dimType.compatible(oper) and dimType.compatible(targetValue)

SimpleConstraint::SimpleConstraint(dt : DimensionType, cp : ConstraintOperator,
       tv : DimensionValue)
pre  : dt <> null and tv <> null
```

```
pre  : dt.compatible(cp) and dt.compatible(tv)
post : dimType = dt and oper = cp and targetValue = tv
```

La méthode `verify` prend une valeur mesurée et retourne vrai si cette valeur satisfait la contrainte et faux sinon.

```
SimpleConstraint::verify(dval : DimensionValue) : Boolean
pre  : dval <> null
post : result = (dimType.verify(dval) and
                   dimType.verify(oper, targetValue.getV(), dval.getV()))
```

Pour les dimensions statistiquement contraintes, la contrainte est définie par une séquence non-vide de contraintes sur chacun des aspects :

```
StatisticalConstraint
invariant : theAspects->notEmpty

StatisticalConstraint::StatisticalConstraint(dt : DimensionType,
       as : Sequence(Aspects))
pre  : dt <> null and as->notEmpty
post : dimType = dt and theAspects = as
```

La méthode `verify` prend une séquence de valeurs mesurées, en vérifie la compatibilité avec le type de dimension, ce qui revient à vérifier les unités et le type des valeurs, et puis fait vérifier la satisfaction de tous les aspects par les valeurs «brutes» (sans unités) correspondantes :

```
StatisticalConstraint::verify(dvals : Sequence(DimensionValue)) : Boolean
post : result = dimType.verify(dvals) and
                dimType.verify(theAspects, dvals->collect(dval | dval.getV()))
```

Un aspect est également défini par un opérateur de contrainte et une valeur cible (en ce qui concerne la moyenne et la variance, tout du moins). La classe abstraite `Aspects` définit les caractéristiques communes aux deux sous-classes concrètes `Mean` et `Variance`. L'opérateur spécial **aspects** est bien sûr proscrit à ce niveau, et la valeur cible doit être définie. Les seules distinctions entre les aspects moyenne et variance sont la manière de calculer l'estimation de la statistique à partir des valeurs mesurées, bien sûr, mais aussi la sémantique de relation qui, si elle s'applique bien à la moyenne, n'a guère de sens pour la variance. QML n'étant pas précis à ce sujet, nous avons considéré que la variance devait toujours être la plus petite possible. `Mean` et `Variance` définissent une méthode de classe `compute` qui prend une séquence de valeurs mesurées et en calcule respectivement la moyenne et la variance.

```
Aspects
invariant : oper <> #aspects and targetValue <> null

Mean
invariant : targetValue.oclIsKindOf(NumOfV)

Mean::Mean(cp : ConstraintOperator, dval : DimensionValue)
pre  : cp <> #aspects and dval <> null and dval.oclIsKindOf(NumOfV)
post : oper = cp and targetValue = dval

Mean::compute(vals : Sequence(V)) : NumOfV
pre  : vals->size >= 1 and vals->forall(v | v.oclIsKindOf(NumOfV))
post : result.num = vals->iterate(e : V ; r : Real = 0.0 |
                                  r + e.oclAsType(NumOfV).num)

Variance
invariant : targetValue.oclIsKindOf(NumOfV)
```

```
Variance::Variance(cp : ConstraintOperator, dval : DimensionValue)
pre  : cp <> #aspects and dval <> null and dval.oclIsKindOf(NumOfV)
post : oper = cp and targetValue = dval
```

```
Variance::compute(vals : Sequence(V)) : NumOfV
pre  : vals->size >= 2 and vals->forall(v | v.oclIsKindOf(NumOfV))
post : let mean := vals->iterate(e : V ; r : Real = 0.0 |
                                  r + e.oclAsType(NumOfV).num)
       and sumsquare := vals->iterate(e : V ; r : Real = 0.0 |
                                  r + (e.oclAsType(NumOfV).num - mean) *
                                         (e.oclAsType(NumOfV).num - mean))
       in result = sumsquare/(vals->size - 1)
```

Une contrainte statistique sait répondre à deux messages : `verify` à un argument, la valeur mesurée, qui retourne vrai si cette valeur est compatible avec la contrainte posée sur l'aspect, et `verify` à deux argument qui prend en plus une sémantique de relation et vérifie si cette dernière est compatible avec l'opérateur de contrainte employé. La première méthode est utilisée pour les dimensions n'ayant pas de relation d'ordre définie et qui se limite à l'opérateur d'égalité dans les contraintes. La seconde est utilisée par les dimensions avec relation d'ordre, et donc sémantique de relation.

```
Mean::verify(vals : Sequence(V) : Boolean
pre  : oper = #equal
post : vals->size < 1 implies result = true
post : vals->size >= 1 implies
          result = (Mean.compute(vals) = targetValue.getV().oclAsType(NumOfV))
```

```
Mean::verify(relSem : RelationSemantics, vals : Sequence(V) : Boolean
post : vals->size < 1 implies result = true
post : vals->size >= 1 implies
          let theMean := Mean.compute(vals)
          and target := targetValue.getV().oclAsType(NumOfV)
          in Ordered.compatible(oper, relsem) implies
             (oper = #equal and result = (theMean = target)) or
             (oper = #sup and result = (not theMean.lessThan(target) and
                                         not theMean = target)) or
             (oper = #supequal and result = (not theMean.lessThan(target))) or
             (oper = #inf and result = (theMean.lessThan(target))) or
             (oper = #infequal and result = (theMean.lessThan(target) or
                                                (theMean = target)))
```

```
Variance::verify(vals : Sequence(V) : Boolean
pre  : oper = #equal
post : vals->size < 2 implies result = true
post : vals->size >= 2 implies
          result = (Variance.compute(vals) = targetValue.getV().oclAsType(NumOfV))
```

```
Variance::verify(relSem : RelationSemantics, vals : Sequence(V) : Boolean
post : vals->size < 2 implies result = true
post : vals->size >= 2 implies
          let theVariance := Mean.compute(vals)
          and target := targetValue.getV().oclAsType(NumOfV)
          in Ordered.compatible(oper, relsem) implies
             (oper = #equal and result = (theVariance = target)) or
             (oper = #sup and result = (not theVariance.lessThan(target) and
                                         not theVariance = target)) or
             (oper = #supequal and result = (not theVariance.lessThan(target))) or
```

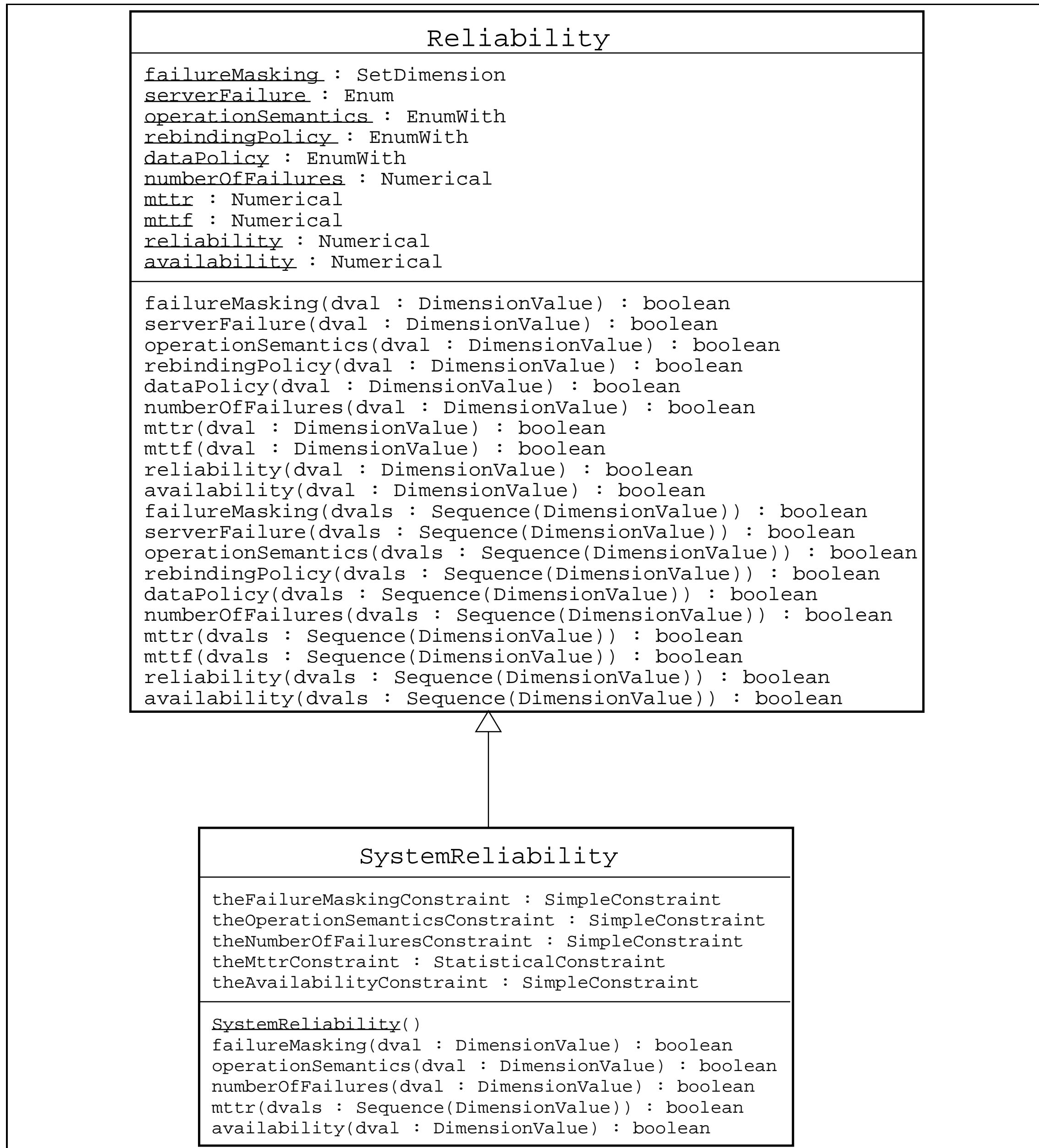

FIG. 10 – Exemple du type de contrat `Reliability` et du contrat `SystemReliability`, vu à la figure 2 exprimé en UML.

```
(oper = #inf and result = (theVariance.lessThan(target))) or
(oper = #infequal and result = (theVariance.lessThan(target) or
                                  (theVariance = target)))
```

5.2.4 Contrats et types de contrats

Reprenons dans leurs grandes lignes la définition des concepts de types de contrats et de contrats de QML [FK98, p. 11–12, 16–18]. Un type de contrat définit un ensemble de dimensions du contrat et associe à chacune un type de dimension. Le type de dimension est défini par une sorte

de dimension, un ensemble de valeurs admissibles et éventuellement une sémantique de relation permettant de déterminer quelles sont les valeurs préférables d'un point de vue qualité de service parmi les valeurs admissibles. Un contrat est une instance d'un type de contrat qui associe à une ou plusieurs des dimensions déclarées par le type de contrat, une contrainte sur les valeurs mesurées selon chacune de ces dimensions pour le service auquel sera imposé le contrat. Lorsqu'un contrat ne définit pas de contrainte sur une dimension, celle-ci est par défaut la contrainte `true`, c'est-à-dire que toute valeur mesurée admissible selon la dimension est réputée satisfaire la contrainte.

Par ailleurs, un contrat peut être défini comme le raffinement d'un autre contrat. Un raffinement est déclaré statiquement et il peut soit se contenter d'ajouter des contraintes sur des dimensions non-contraintes par le contrat raffiné, soit remplacer certaines contraintes sur des dimensions déjà contraintes par des contraintes plus fortes. Nous souhaitons finalement pouvoir vérifier qu'une valeur mesurée satisfait la contrainte sur une dimension donnée d'un contrat.

Ces caractéristiques doivent guider notre modélisation. Notre sémantique dénotationnelle montre que contrats et types de contrats sont essentiellement des mécanismes d'organisation des espaces de noms et de liaison de ces noms à leur valeur. En programmation par objets, les classes et les objets jouent effectivement ce rôle. La classe définit un ensemble de variables, qui sont liées à leur valeur par l'objet, et elle lie elle-même un ensemble de noms de méthodes à leur corps. Il paraît donc intéressant de faire correspondre type de contrats et contrats avec classes ou objets. Et plus particulièrement, il est tentant de voir un type de contrats comme une classe et un contrat comme un objet, instance de la classe représentant son type.

Malheureusement, cette vision initiale résiste difficilement aux autres aspects de la description précédente. En effet, d'une part le contrat n'est pas une instantiation stricte du type de contrat dans la mesure où il n'a pas à définir des contraintes pour toutes les dimensions déclarées. D'autre part, «le raffinement est une relation définie statiquement [...] à la manière de l'héritage dans les langages à objets» [FK98, p. 11]. Pour faire correspondre au raffinement de contrat à l'héritage, il faut promouvoir le contrat au niveau de la classe. On en conclut immédiatement qu'il faudrait promouvoir le type de contrat au niveau de la métaclasse !

L'ennui de cette promotion est justement l'introduction de métaclasses. Si le concept est gérable en UML, par le stéréotype `«metaclass»`, il téléguide en quelque sorte le choix du langage de programmation dans la mesure où tous les langages n'offrent pas le mécanisme de métaclasse. En particulier, le langage que nous avons choisi, Java, n'offre pas ce mécanisme.

Par contre, un examen rapide nous convainc que le type de contrats ne fait que définir les noms de dimensions et leur associer des types de dimension, sans que cela puisse être remis en cause. Il suffit donc pour cela de définir cette association sous forme de variables de classe, donc non redéfinissables, et qui plus est constantes puisque définies une fois pour toutes pour tous les contrats du type. On constate d'ailleurs que l'instantiation et le raffinement en QML, plutôt que de correspondre précisément à l'instantiation et l'héritage d'un langage à classes, sont plutôt d'une nature mixte entre le modèle à classes et le modèle à prototypes qui utilise plutôt le clonage et la délégation [Mal96].

Une solution consiste à définir un type de contrat comme une classe racine d'une hiérarchie. Cette classe va définir pour chaque dimension :

- une variable (constante) de classe du nom de la dimension qui contiendra l'objet instance d'une sous-classe de `DimensionType` représentant le type et la sorte de cette dimension, et
- deux méthodes d'instance également du nom de la dimension qui exprimeront la contrainte minimale imposée sur cette dimension par le type de contrat en prenant comme argument respectivement une valeur mesurée et une séquence de valeurs mesurées, en retournant vrai si cette valeur ou ces valeurs sont admissibles, c'est-à-dire qu'elles ont la bonne unité de mesure et qu'elles font partie de l'ensemble des valeurs possibles sur cette dimension.

Le contrat sera représenté par une sous-classe de la classe représentant le type de contrat. Cette classe de contrat redéfinit les méthodes selon les contraintes à imposer sur chacune des dimensions et déclare autant de variables d'instance pour contenir les objets contraintes correspondants. Les

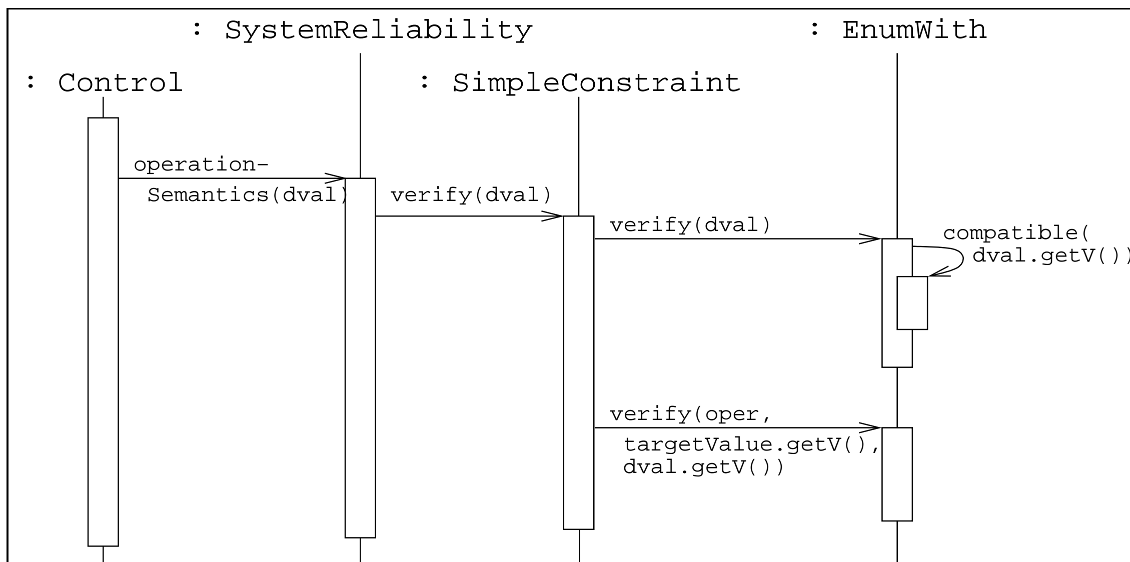

FIG. 11 – Diagramme de séquence pour la vérification d'une valeur mesurée sur une contrainte simple (`operationSemantics` du contrat `SystemReliability`).

méthodes seront chargées de vérifier si la valeur mesurée satisfait la contrainte, retournant vrai si c'est le cas et faux sinon. La figure 10 illustre cette approche en reprenant l'exemple QML de la figure 2 pour l'exprimer dans cette structure de classes en UML.

Dans cet exemple, les variables de classe de `Reliability` vont contenir les instances des classes de types de dimension, dont la racine est la classe abstraite `DimensionType`. Ces objets vont représenter les types et les sortes de dimension QML associés à chacune des dimensions déclarée par le type de contrat. Les méthodes de `Reliability` exercent un contrôle minimal pour les valeurs mesurées en vérifiant simplement si leur unité de mesure correspond et si la valeur fait partie des valeurs admises par le type de dimension. Pour cela, elle envoie tout simplement le message `verify(dval)` à l'instance contenue dans la variable de classe correspondant à la dimension. Les variables de la classe `SystemReliability` contiennent des instances de contraintes, dont la racine est la classe abstraite `Constraint`. Les méthodes de cette classe exercent le contrôle complet des valeurs mesurées, à savoir qu'elles soient admissibles par le type de dimension et qu'elles satisfont la contrainte posée sur le dimension. Pour cela, ces méthodes envoient le message `verify` à l'instance de constrainte correspondnte dans le contrat.

Pour mieux comprendre comment la vérification est faite dans chaque cas, la figure 11 présente le diagramme de séquences pour l'appel à la méthode `operationSemantics` sur une instance de contrat `SystemReliability`. Cet appel se répercute par un appel à la methode `verify(DimensionValue)` de l'objet contrainte correspondant, qui lui va appeler l'instance d'`EnumWith` représentant le type de dimension associé à la dimension `operationSemantics`, d'abord pour vérifier que la valeur mesurée est admissible pour cette dimension, puis pour vérifier que la valeur mesurée satisfait la contrainte posée sur cette dimension. Le premier appel à l'instance d'`EnumWith` induit une appel à `self` pour vérifier la compatibilité de la valeur «brute» (sans unités de mesure), puis la correspondance des unités est elle-même vérifiée.

Le cas de contraintes posées sur les aspects statistiques est un peu plus complexe ; il est présenté à la figure 12. La vérification est initiée par l'envoi du message `mttr` à l'instance de contrat `SystemReliability` avec pour paramètre une séquence de valeurs mesurées. Ceci se répercute par un envoi de message `verify` à l'instance de la contrainte statistique `StatisticalConstraint` qui va ensuite se répercuter par le message `verify` envoyé à l'instance de la sorte de dimension `EnumWith`. Ce dernier message va vérifier chacune des valeurs de la séquence en se rappelant répé-

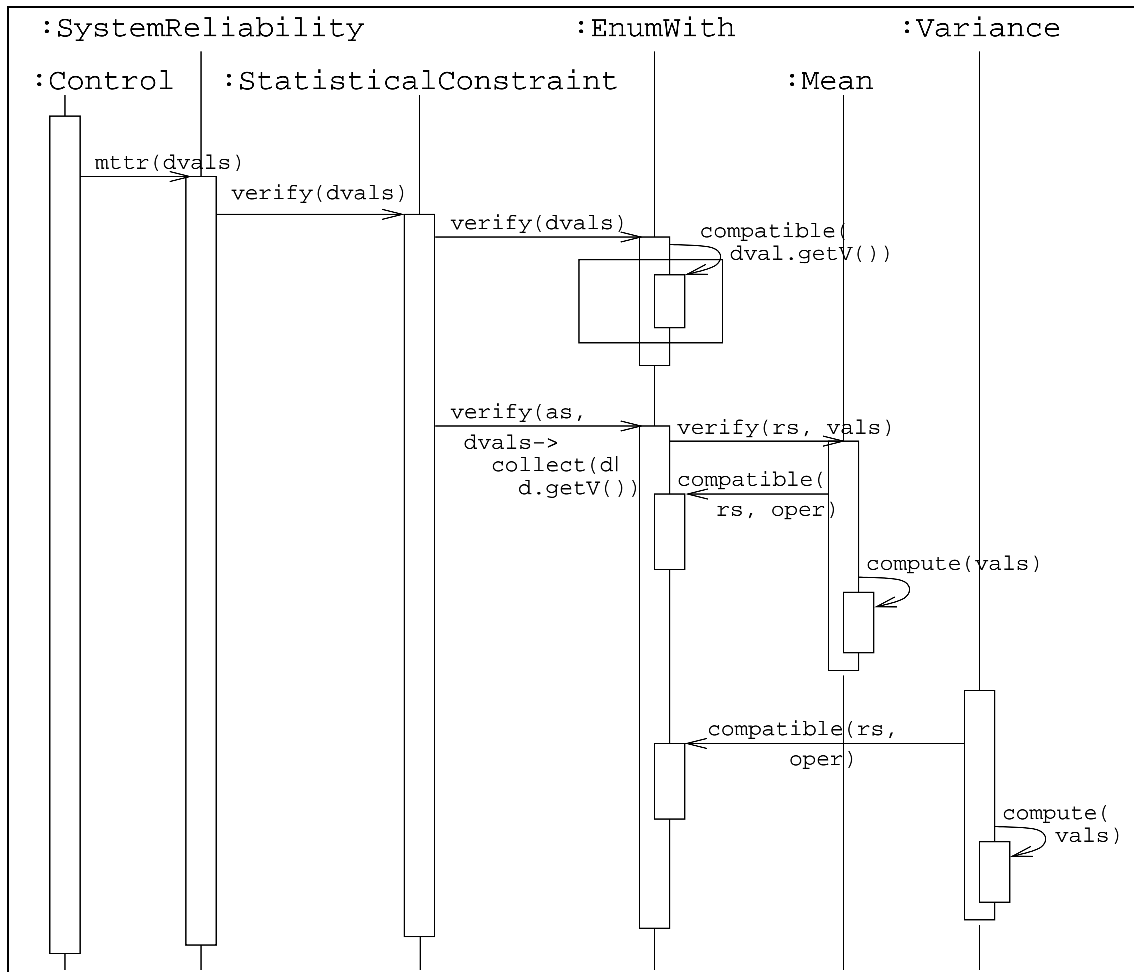

FIG. 12 – Diagramme de séquence pour la vérification d'une séquence de valeurs mesurées sur une contrainte statistique (`mttr` du contrat `SystemReliability`).

titivement avec le message `compatible` sur chaque valeur «brute», tout en vérifiant les unités de mesure au passage. Ceci fait, le contrôle revient à l'instance de `StatisticalConstraint` qui va envoyer à l'instance d'`EnumWith` le message `verify` cette fois-ci avec pour paramètre la séquence d'instances d'`Aspects` (`Mean` et `Variance`) et la séquence de valeurs «brutes» (sans unités). L'instance d'`EnumWith` va ensuite envoyer un message `verify` à chaque instance d'`Aspects` avec pour paramètre la sémantique de relation définie sur cette sorte de dimension et la liste de valeurs «brutes». L'instance de `Mean` (et ensuite de `Variance`) vérifie la compatibilité de son opérateur de contrainte et de la sémantique de relation en envoyant le message `compatible` à l'instance d'`EnumWith`, puis elle va calculer la statistique en s'envoyant à elle-même le message `compute` pour enfin vérifier que cette valeur respecte la contrainte posée sur cet aspect.

Cette approche permet de faire correspondre le raffinement de QML à l'héritage en UML. Les classes représentant les types de contrats et les contrats ont cependant une forme très particulières, avec des contraintes de construction fortes. Les méthodes sont par ailleurs toutes similaires, à tel point qu'elles peuvent être générées automatiquement. Pour définir ces contraintes et introduire les concepts de types de contrats et de contrats en UML, nous allons utiliser les mécanismes d'extension d'UML pour définir au niveau du métamodèle les entités types de contrats QML et contrats QML desquelles les classes de type de contrats et de contrats seront instance via

- ***Description (Description)***
  Ceci définit une extension d'UML pour la modélisation des contrats de qualité de service à la QML. Deux stéréotypes sont introduits pour définir les types de contrats et les contrats.
- ***Extensions prérequises (Prerequisite Extensions)***
  Cette extension n'utilise aucune autre extension pour sa définition, mais par contre utilise le *package* QML formé des classes modélisées jusqu'ici dans ce rapport.
- ***Stéréotype (Stereotype)***
  - Nom (Name) : `QMLContractType`
  - Classe dans le métamodèle (Metamodel Class) : `Class`
  - Sémantique (Semantics) : Une classe représentant un type de contrat à la QML.
  - Icône syntaxique (notation) (Syntax (Notation) Icon) : aucune.
  - Contrainte (Constraint Property) :

    Une classe sétéréotypée par <<QMLContractType>> doit posséder autant de variables de classes qu'il y a de types de dimensions dans le type de contrat. La valeur de chacune de ces variables doit être instance d'une sous-classe de la classe `DimensionType` du *package* QML. Elle doit aussi posséder deux méthodes d'instance à résultat booléen pour chaque variable de classe, et ces méthodes doivent avoir les mêmes noms que les variables de classe et posséder respectivement un argument de type `DimensionValue` et un de type `Sequence(DimensionValue)`.
  - Valeur étiquetée (Tagged Value Property) : aucune
- ***Stéréotype (Stereotype)***
  - Nom (Name) : `QMLContract`
  - Classe dans le métamodèle (Metamodel Class) : `Class`
  - Sémantique (Semantics) : Une classe représentant un contrat à la QML.
  - Icône syntaxique (notation) (Syntax (Notation) Icon) : aucune.
  - Contrainte (Constraint Property) :

    Une classe sétéréotypée par <<QMLContract>> doit avoir une et une seule superclasses stéréotypée par <<QMLContractType>>. Elle doit posséder autant de variables de classes'instance qu'il y a de dimensions contraintes dans le contrat. La valeur de chacune de ces variables doit être instance d'une sous-classe de la classe `Constraint` du *package* QML. Elle doit aussi posséder une méthode d'instance à résultat booléen pour chaque variable d'instance, et cette méthodes doit avoir les mêmes noms que les variables de classe de ces dimensions dans sa superclasse et cette méthode doit posséder un argument de type `DimensionValue` ou un de type `Sequence(DimensionValue)` selon que la dimension est contrainte sur les valeurs mesurées ou sur un aspect statistique des valeurs mesurées.
  - Valeur étiquetée (Tagged Value Property) : aucune
- ***Règles d'utilisation (Well-Formedness Rules)***
  à faire.
- ***Commentaires (Comments)***
  à faire.

FIG. 13 – Extension d'UML pour les contrats de qualité de service QML (UML Extension for defining QML-like Quality of Service Contracts).

classification par stéréotypage UML [SA99]. La figure 13 donne une première idée d'une extension d'UML pour inclure les contrats à la QML.

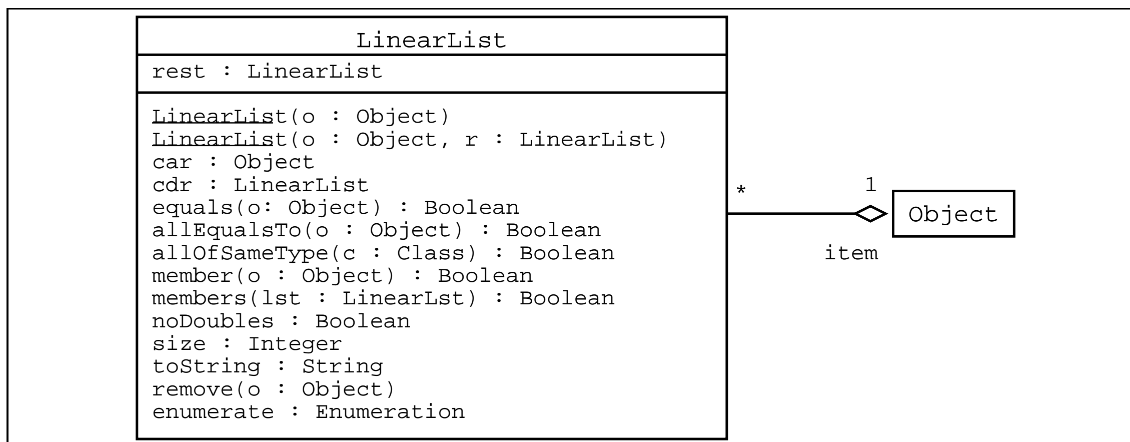

FIG. 14 – Diagramme UML de la classes `LinearList`

6 Implantation en Java

Nous avons réalisé une implantation complète en Java du modèle UML présenté à la section précédente. Les contrats dans cette mise en œuvre sont définis selon la norme de fait proposée par l'outil *iContract* [Kra98]. Les séquences ont été mises en œuvre par une classe `LinearList` que nous avons défini, bien qu'une utilisation de la classe standard `java.util.LinkedList` serait plus judicieuse et sera adoptée dans les versions subséquentes du *package*. La principale question soulevée par cette mise en œuvre concerne les types énumérations, disponibles en UML mais absent de Java. Les domaines sémantiques **ConstraintOperator** et **RelationSemantics** sont concernés. Nous présentons une approche de mise en œuvre de tels types répondant aux critères de la programmation par objets et d'unicité des valeurs de ces types. Le reste de la section indique les principales décisions de mise en œuvre en Java du modèle UML.

6.1 La classe `LinearList`

Le diagramme UML d'implantation de la classe `LinearList` apparaît à la figure 14. Les listes sont représentées par des objets paires *à la Lisp*, avec des méthodes `car` et `cdr` pour en saisir le premier élément et la suite des éléments. Nous n'avons pas posé de contrainte particulière comme invariant de la classe `LinearList`. Par contre les méthodes de cette classe sont soumises aux pré- et postconditions suivantes. Le constructeur sans reste construit une paire où le paramètre devient l'élément de la liste singleton et le reste de la liste est initialisé à `null` :

```
LinearList::LinearList(o : Object)
post : item = o and rest = null
```

Le constructeur avec reste de liste connu crée une paire dont l'élement est son premier paramètre et le reste son second paramètre :

```
LinearList::LinearList(o : Object, r : LinearList)
post : self.item = o and self.rest = r
```

Pour les méthodes `car` et `cdr`, le résultat est respectivement égal à l'élément et au reste de la paire réceptrice :

```
LinearList::car : Object
post : result = self.item
```

```
LinearList::cdr : LinearList
post : result = self.rest
```

Pour la méthode `equals`, les choses sont légèrement plus complexes, dans la mesure où, la méthode étant récursive, il paraît judicieux d'établir un contrat «local» en quelque sorte, c'est-à-dire qui ne concerne qu'une paire et non toute la liste. Ce contrat sera nécessairement une sous-spécification du résultat, mais pourra détecter nombre d'erreurs sans avoir à calculer le résultat. L'idée de cette sous-spécification est que, pour une implantation récursive, la vérification du contrat sur chaque élément de la liste va garantir l'exactitude de la méthode. Nous utiliserons généralement cette approche pour écrire les contrats des méthodes récursives.

Ici, on se contentera donc d'indiquer que le résultat est nécessairement faux si le paramètre `o` a pour valeur `null`, si le paramètre `o` n'est pas instance de la classe `LinearList`, si les restes du receveur et de l'argument ne concordent pas (tous deux `null` ou non-`null`) et enfin si les `item` des deux ne sont pas égaux :

```
LinearList::equals(o : Object)
post : o = null implies result = false
post : o <> null and not o.oclIsKindOf(LinearList) implies result = false
post : self.rest <> null and o.oclAsType(LinearList).rest = null
       implies result = false
post : self.rest = null and o.oclAsType(LinearList).rest <> null
       implies result = false
post : o <> null and self.item <> o.oclAsType(LinearList).item
       implies result = false
```

Pour les méthodes `allEqualsTo` et `allOfSameType`, qui vérifient respectivement si tous les éléments de la liste sont égaux à l'argument et si tous les éléments sont directement ou indirectement instance de l'argument, même idée de contrat «local» incomplet. Si la valeur d'`item` de l'élément courant n'est pas égal (resp. pas instance de) à l'argument, alors le résultat est nécessairement faux, mais par contre si ces deux valeurs sont égales (resp. la valeur instance de l'argument) et le reste est égal à `null`, alors le résultat est nécessairement vrai. Le contrat de la méthode `member` est fort similaire, mais en prenant un argument inverse.

```
LinearList::allEqualsTo(o : Object) : Boolean
post : o <> self.item implies result = false
post : self.item = o and self.rest = null implies result = true
```

```
LinearList::allOfSameType(c : Class) : Boolean
pre  : c <> null
post : not self.item.oclIsKindOf(c) implies result = false
post : self.item.oclIsKindOf(c) and self.rest = null implies result = true
```

```
LinearList::member(o : Object) : Boolean
post : self.item = o implies result = true
post : self.item <> o and self.rest = null implies result = false
```

La méthode `members` vérifie que tous les éléments de la liste `lst` passée en paramètre sont membre de la liste réceptrice. Le contrat établit que si `lst` est `null`, alors le résultat est vrai, mais que si `lst` n'est pas `null` et que son premier élément ne fait par partie de la liste réceptrice, alors le résultat est faux :

```
LinearList::members(lst : LinearList) : Boolean
post : lst = null implies result = true
post : lst <> null and not self.member(lst.car()) implies result = false
```

La méthode `noDoubles` retourne vrai si aucun élément de la liste n'apparaît plus d'une fois dans celle-ci. Même genre de sous-spécification ici :

```
LinearList::noDoubles : Boolean
post : self.rest = null implies result = true
post : self.rest <> null and self.rest.member(self.item) implies result = false
```

Même genre de sous-spécification encore pour la méthode `size` qui retourne la taille de la liste. Si le reste du receveur est `null`, alors le résultat est 1, sinon il est supérieur à 1 :

```
LinearList::size : Integer
post : self.rest = null implies result = 1
post : self.rest <> null implies result > 1
```

Enfin, pour la méthode `remove`, qui retire un élément de la liste si celui-ci y apparaît, si l'`item` du receveur est égal à l'argument, le résultat est égal au `rest` du receveur, sinon, il est égal au receveur :

```
LinearList::remove(elem : Object)
post : self.item = elem implies result = self.rest
post : self.item <> elem implies result = self
```

Tout cela est donc très standard, à ceci près que des méthodes comme `allEqualsTo`, `noDoubles` et `allOfSameType` ont été introduites pour définir des contrats solides sur les listes sans passer par des expressions `forall` et `exists` complexes.

6.2 Opérateurs de contraintes et sémantiques de relation

Les six opérateurs de contraintes et les trois sémantiques de relation d'ordre de QML ont été dénotées à l'aide d'ensembles finis de valeurs dans notre sémantique dénotationnelle. Dans un langage impératif classique, de tels ensembles de valeurs sont généralement représentés à l'aide de types énumération. Dans plusieurs langages à objets, les types énumération n'existent pas directement. Une première solution simple consisterait à définir un encodage des différentes valeurs par des entiers. Évidemment, on évitera d'utiliser directement ces entiers dans le programme. On peut ainsi définir une classe, potentiellement abstraite, définissant autant de constantes de classe qu'il y a de valeurs du type à représenter. On se donne alors comme discipline de programmation de n'utiliser que ces (noms de) constantes dans le programme.

Le principal défaut de cette solution simple est l'abandon de la vérification de type statique. L'encodage en entier, par exemple, amène à déclarer toutes les variables pouvant contenir des valeurs du type comme des variables entières. Rien ne permet de s'assurer statiquement que seules des valeurs du type seront affectées dans ces variables. De plus, lors des comparaisons, il reste possible d'obtenir un résultat vrai en comparant un entier et une valeur du type. Cette confusion de valeurs entre entiers et valeurs du type peut provoquer des erreurs de programmation difficiles à repérer. L'«unicité» des valeurs du types est donc ardamment souhaitable.

Dans les langages comme Lisp, une pratique courante pour éviter cette non-unicité des valeurs consiste à représenter chacune des valeurs par une paire, par définition unique dans un programme.[7] L'avantage de cette approche est d'éviter la confusion des valeurs entre le type d'encodage et le type encodé. De plus, on récupère l'égalité simple des pointeurs comme comparaison d'égalité entre deux valeurs du type encodé.

L'équivalent objet de cette idée consiste à représenter chaque valeur par l'instance unique d'une classe singleton [GHJV95]. Nous proposons donc ici une extrapolation assez simple de cette idée, que l'on pourrait appeler le patron *énumération* : chaque ensemble fini de valeurs est représenté par une classe abstraite et autant de sous-classes concrètes singleton qu'il y a de valeurs. La classe

[7]Attention, cette affirmation n'est vraie que si la gestion de mémoire n'est pas trop «aggressive». Certains compilateurs Lisp détectent en effet les paires dont les deux valeurs sont `nil`, et les représentent par une seule et unique valeur. Cette propriété a avantage à être vérifiée pour s'assurer que la modélisation que nous proposons ici fonctionne effectivement. Nos tests montrent qu'en Java sous Linux, cela fonctionne parfaitement.

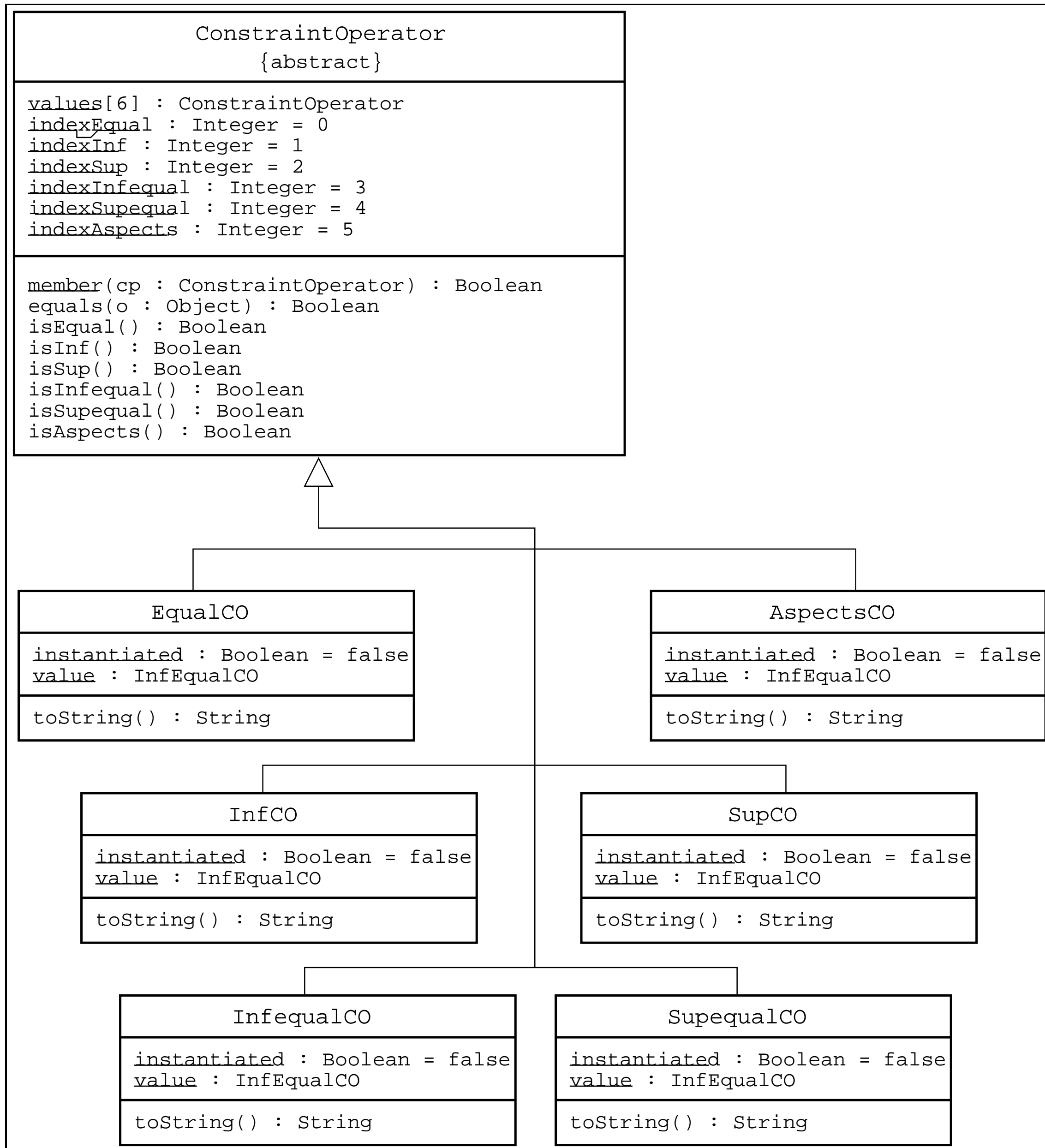

FIG. 15 – Classes d'opérateurs de contraintes

abstraite est dotée d'un tableau de toutes les valeurs du type et elle implante un méthode de classe `member` qui vérifie si l'argument est bel et bien une valeur du type. Les figures 15 et 16 donnent les modèles d'implantation de ces classes utilisant ce «patron» pour représenter les opérateurs de contraintes et les sémantiques de relation d'ordre de QML.

Les opérateurs de contraintes

La classe abstraite `ConstraintOperator` définit le type des valeurs d'opérateurs de contraintes. Elle comporte une variable de classe `values` qui est un tableau des six valeurs qui sera initialisé

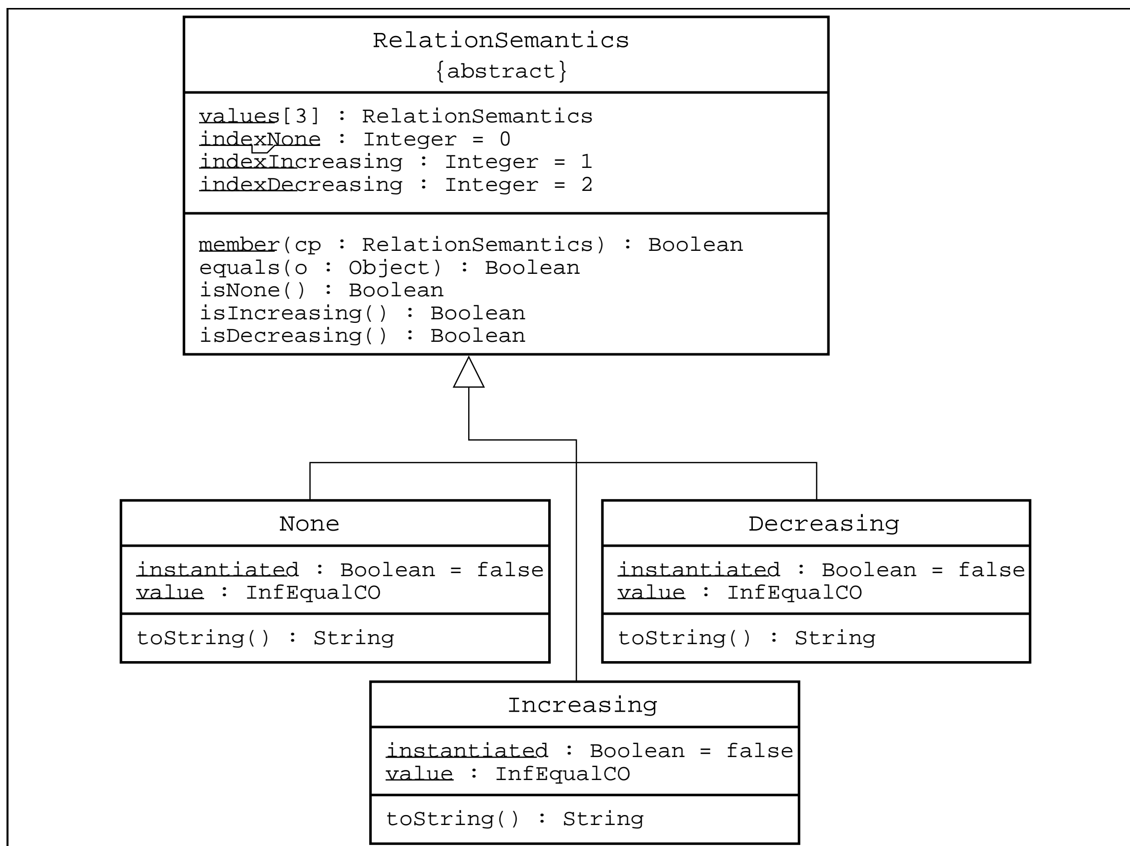

FIG. 16 – Classes de sémantiques de relation

statiquement à partir des instances uniques des six sous-classes singleton, ce qui est indiqué par son invariant :

```
ConstraintOperator
invariant : self.values <> null and
            self.values[self.indexEqual] = EqualCO.value and
            self.values[self.indexInf] = InfCO.value and
            self.values[self.indexSup] = SupCO.value and
            self.values[self.indexInfequal] = InfequalCO.value and
            self.values[self.indexSupequal] = SupequalCO.value and
            self.values[self.indexAspects] = AspectsCO.value
```

Elle déclare également six constantes de classe donnant les indices des valeurs dans le tableau précédent. Enfin elle définit la méthode de classe `member` et la méthode d'instance `equals`, pour vérifier respectivement l'appartenance et l'égalité de deux valeurs du type. Ces valeurs sont contraintes à être l'une des instances unique des six sous-classes apparaissant dans le tableau `values` de la classe.

```
ConstraintOperator::member(cp : ConstraintOperator) : Boolean
post : result = (cp = self.values[indexEqual] xor cp = self.values[indexInf] xor
          cp = self.values[indexSup] xor cp = self.values[indexInfequal] xor
          cp = self.values[indexSupequal] xor cp = self.values[indexAspects])
```

```
ConstraintOperator::equals(o : Object) : Boolean
post : result = (ConstraintOperator.member(o) and ConstraintOperator.member(self)
       and o = self)
```

Cinq méthodes, `isEqual`, `isInf`, `isSup`, `isInfequal`, `isSupequal` et `isAspects`, permettent de savoir quelle valeur parmi les six possibles on manipule. Compte tenu du choix de modélisation, il leur suffira de vérifier l'égalité du receveur à la valeur unique de la classe correspondante.

Les six sous-classes singleton `EqualCO`, `InfCO`, `SupCO`, `InfequalCO`, `SupequalCO`, et `AspectsCO` représentent respectivement les opérateurs égal, inférieur, supérieur, inférieur ou égal, supérieur ou égal et finalement l'opérateur «spécial» aspects désignant les contraintes statistiques. Chacune de ces classes définit deux variables de classes : `instantiated` et `value`, la première servant à vérifier l'intantiation unique de la classe et la seconde contenant l'instance unique de la classe ontenue par initialisation statique.

L'invariant sur la classe `EqualCO` indique que cette classe a une seule instance et que cette instance est contenue dans sa variable `value` :

```
EqualCO
invariant : instantiated = true and EqualCO.allInstances->size = 1
            and EqualCO.allInstances->includes(EqualCO.value)
```

Un invariant similaire existe sur chacune des classes sœurs d'`EqualCO`.

Les sémantiques de relation d'ordre

La classe abstraite `RelationSemantics` définit le type des valeurs de sémantique de relation d'ordre. Elle est construite de manière analogue à `ConstraintOperator`. Ses sous-classes singleton concrètes, `None`, `Increasing` et `Decreasing`, sont elles aussi construites de la même manière que les classes d'opérateurs de contraintes, et elles représentent respectivement les valeurs «sans sémantique de relation», «sémantique croissante» (le plus grandes valeurs seon la relation d'ordre sont les plus intéressantes) et «sémantique décroissante» (le plus petites valeurs sont les plus intéressantes).

L'invariant de la classe `RelationSemantics` est construit similairement à celui de la classe `ConstraintOperator` :

```
RelationSemantics
invariant : self.values <> null
            and self.values[self.indexNone] = None.value and
            self.values[self.indexIncreasing] = Increasing.value and
            self.values[self.indexDecreasing] = Decreasing.value
```

Les contrats et invariants sur les autres classes s'obtiennent de la même manière, par simple substitution de rôle d'une application du patron à l'autre.

Mise en œuvre en Java

La mise en œuvre en Java de ce modèle d'implantation ne pose qu'un seul problème : comment gérer la création des instances des classes singleton ? L'idée que nous avons adoptée consiste à utiliser un initialiseur statique chargé de créer l'instance unique de la classe puis de définir une méthode d'initialisation qui émet un message d'erreur s'il est appelé. Une variable booléenne **instantiated** indique que l'instance unique a été crée, ce qui permet d'exécuter le contructeur normalement lors de la première instantiation. Par exemple, sur la classe `EqualsCO`, nous avons :

```
  // Initialiseur statique
  static {
```

```
    value = new EqualCO() ;
    ConstraintOperator.values[ConstraintOperator.indexEqual] = value ;
    instantiated = true ;
  }

  // Constructeur
  public                EqualCO() {
    if ( instantiated ) {
        System.out.println("EqualCO shouldn't be instantiated!") ;
    }                                           //  if ( instantiated )
  }
```

Par ailleurs, la traduction d'OCL vers le langage de contrats associé à Java se heurte à l'impossibilité en Java d'énumérer toutes les instances d'une classe. Pour contourner cette difficulté, l'invariant a été modifié en utilisant le fait que l'unique instance est conservée dans la variable de classe `value` pour tester que le receveur est bel et bien égal au contenu de cette variable (i.e. `value == this`), ce qui impose qu'elle contienne cette instance unique.

6.3 Autres décisions importantes pour la mise en oeuvre

Les classes `QMLSet` et `QMLPOSet` sont mises en œuvre en utilisant la classe `LinearList` pour représenter l'ensemble des valeurs.

La classe `OrderRelation` représente une relation d'ordre R à l'aide d'une table de hachage, instance de la classe `Hashtable` de Java. Les couples sont représentés par des instances de la classe `Couple`, qui sont utilisées comme clé dans la table. Ce choix vient du fait qu'une même clé ne peut se retrouver plusieurs fois dans la table de hachage. Ainsi, l'utilisation du premier élément du couple comme clé aurait empêché qu'un même élément puisse être en relation avec plusieurs éléments de l'ensemble. Lors de l'insertion, la paire est donc utilisée à la fois comme clé et comme cible. Pour vérifier si un élément a est en relation avec un élément b, il suffit de lancer une requête à la table de hachage à partir de la paire `new Couple(a, b)`. Si cette requête retourne `null`, alors a n'est pas en relation avec b. Sinon, a est en relation avec b.

Dans la classe `Couple`, la méthode `hashCode` a été introduite en lien avec cette proposition de représentation des relations d'ordre par table de hachage. En Java, tout objet pouvant servir de clé dans une table de hachage doit savoir répondre de manière appropriée aux messages `equals` et `hashcode`.

Une relation d'ordre est définie par une séquence de couples d'élements sur un ensemble donné. Comme nous l'avons dit cependant, la relation définie sur ces couples doit êtr comprise comme transitive. Notre implantation calcule la fermeture transitive de la relation en utilisant un algorithme adapté de R. Sedgewick [Sed92, pp. 473-476]. Cet algorithme balaie tous les éléments de l'ensemble en tant que source s et destination d, et si s est en relation avec d, alors pour tous les d' dans l'ensemble tels que d est en relation avec d', on ajoute le couple (s, d') à la relation.

La classe `OrderRelation` offre deux méthodes `addCouple` et `removeCouple` permettant respectivement d'ajouter et de retirer des couples dans la relation. Pour faciliter le calcul de la fermeture transitive lors du retrait d'un couple, la variable `couples` contient la liste des couples initaux de la relation.

7 Conclusions, évaluation et perspectives

Nous avons procédé à une étude sémantique du langage QML en trois grandes étapes. Dans un premier temps, nous avons proposé une sémantique dénotationnelle de QML orientée vers l'éva-

luation dynamique des contrats. Cette sémantique a permis d'identifier les éléments importants de la dénotation du langage ainsi que les liens entre les différentes déclarations du langage et l'objectif de la sémantique. Cette sémantique a été prototypée en Scheme et fournit ainsi un outil permettant de valider des contrats QML et d'en vérifier l'évaluation sur des données spécifiques.

Dans un second temps, nous avons proposé un modèle UML de QML fondé sur les informations obtenues de la sémantique dénotationnelle. Nous avons fait ressortir les différences entre sémantique dénotationnelle et modèle UML, et nous avons proposé des transformations permettant de préserver les propriétes de la sémantique formelle tout en tirant partie des avantages de la structuration en termes de classes. Nous avons insisté pour contractualiser en OCL la plupart des éléments de notre modèle, ce qui a permis d'exprimer sous forme de contrats certaines parties de la sémantique statique de QML. Nous avons proposé une première ébauche d'une extension «contrat» à la QML pour UML par stéréotypage des classes représentant les types de contrats et les contrats.

Enfin, nous avons proposé une implantation en Java complète, documentée en Javadoc et contractualisée selon la norme établie par l'outil *iContract*. Cette implantation permet de créer des types de contrats, des contrats et de vérifier leur évaluation sur des données spécifiques. L'utilisation de cette implantation est conditionnée par l'utilisation directe de Java et non de la syntaxe de QML. La génération de classes représentant les types de contrats et les contrats directement depuis la syntaxe QML reste à faire, mais ne pose pas de problèmes techniques particuliers.

Évaluation

Le point fort de QML est sans conteste sa capacité à exprimer des ontologies générales de la qualité de service, en termes de dimensions sur des types de contrats et sur les contrats. Par contre, QML reste faible sur de nombreux points. Nous regrettons en particulier le choix de lier la déclaration des dimensions aux types de contrats. Certes plusieurs types de contrats peuvent définir les mêmes dimensions, mais cela se fait de manière indépendante. Cela rend très difficile le partage de définitions de dimensions entre contrats, d'autant que si des mécanismes d'extension des contrats existent, rien d'équivalent n'est prévu pour les types de contrats. Il apparaît donc assez clairement qu'aux côtés des contrats et des types de contrats, les types de dimensions et les dimensions sont aussi des entités qui devraient être de plein droit dans QML de manière à ce qu'un type de dimension puisse être déclaré une fois pour toutes et utilisé dans différents types de contrats.

La deuxième grande critique que nous ferons à QML porte sur la notion de profil. D'une part, les profils de QML rendent la liaison d'une expression d'un contrat et des entités appartenant à l'interface plutôt difficile. Les contrats à la Eiffel (et OCL) utilisent directement les entités comme les paramètres et le résultat des méthodes dans leur expression. QML permet d'attacher certaines contraintes de qualité de service aux entités de l'interface, mais la liaison reste pauvre. Par ailleurs, s'il est exact qu'une même interface peut en pratique être proposée avec différents contrats, il nous semble que le lien entre expressions du contrat et l'interface est à ce point fort que l'existence de deux entités séparées (interface et profil) ne paraît pas judicieuse. De plus amples expériences sont nécessaires ici pour valider l'une ou l'autre des approches.

Enfin, la troisième grande critique que nous ferons porte sur les contraintes qui demeurent relativement pauvres et sémantiquement limitées. En effet, en contractualisation sur la performance des composants, il paraît évident qu'il serait utile de lier par exemple le temps de réponse à la taille des paramètres réels passés à la méthode via l'interface. Cette idée est présente dans toute analyse théorique ou expérimentale de la performance. Or, le langage de contraintes de QML ne permet pas d'exprimer un tel lien.

Perspectives

Outre les extensions immédiates aux outils que nous avons produits, les grandes perspectives ouvertes par ces travaux dans le domaine des langages de spécification des contrats de qualité de service sont très nombreuses. Les plus intéressantes de notre point de vue concernent l'extension de l'expressivité des contraintes pouvant être exprimées dans les contrats. Une première perspective à court terme pour refermer la boucle de l'approche contractuelle consiste à intégrer à QML des contrats fonctionnels classiques à la Eiffel. Cet exercice exigera de revoir la relation entre contrats et interfaces, et donc la notion de profil QML, dans le sens d'un meilleur lien entre interfaces et contrats.

Ensuite, la puissance d'expression des contraintes de qualité de service devra être abordée. Cela pose essentiellement trois questions :

1. quel niveau d'expressivité est-il nécessaire pour exprimer les contraintes de qualité de service ?
2. quels outils peuvent aider à traiter ces contrats versus le problème de la conformité entre contrats ?
3. en fonction du langage d'expression choisi, comment ces contrats seront-ils vérifiés, statiquement et dynamiquement ?

Il va de soi que ces éléments sont fortement interreliés. Idéalement, le maximum d'expressivité doit être recherché, mais sans compromettre complètement la possibilité de vérifier la conformité et encore moins la possiblité de vérification dynamique de l'observation des contrats. À ce titre, nous exprimons notre accord avec Frølund et Koistinen pour dire que le problème de conformité est un problème de satisfaction de contraintes. À ce titre, nous croyons que la prochaine étape sera de fixer l'expressivité des contraintes en s'inspirant des langages de programmation par contraintes qui donnent une indication précise du type de problèmes de satisfaction de contraintes que nous savons résoudre à peu près efficacement à ce jour. L'utilisation de moteurs de résolution de contraintes pour traiter le problème de conformité est bien sûr la suite logique de cette pétition de principe.

La question de la vérification dynamique des contrats n'est pas abordée par QML, qui voit les contrats comme des moyens de spécification (d'où le traitement privilégié du problème de la conformité). Dans la tradition de la programmation par contrats, il paraît utile de pouvoir vérifier dynamiquement les contrats pour réagir à leur violation éventuelle. Pour autant, il faut déterminer des moyens de vérification raisonnables. Cela pose en particulier des problèmes sur les systèmes répartis et encore plus dans le contexte d'applications embarquées où les ressources sont rares, d'où l'intérêt de la contractualisation, mais où le coût des vérifications dynamiques ne peut être supporté par la cible. Parce qu'il est peu probable qu'une approche d'évaluation générique soit acceptable dans tous les cas, il apparaît que des indications pourraient être fournies lors de la définition de ces contrats. Cela nous amène directement à une notion de métacontrat qui mérite d'être étudiée dans le détail.

Enfin, l'intégration d'un QML ainsi amélioré à UML reste en grande partie à faire, sur les bases de l'extension à UML par stéréotypage que nous avons commencé à dégrossir.

Références

[FK98] S. Frølund et J. Koistinen. QML : A Language for Quality of Service Specification. Rapport technique n° HPL-98-10, Software Technology Laboratory, Hewlett-Packard, February 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, et J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Kra98] R. Kramer. iContract – The Java(tm) Design by Contract(tm) Tool. In *Proceedings of the 26th International Conference on Technology of Object-Oriented Languages and Systems, TOOLS USA'98.* IEEE, 1998.

[Mal96] J. Malenfant. Abstraction et encapsulation en programmation par prototypes. *Technique et science informatiques*, 15(6) :709–733, 1996.

[Mey97] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, 2nd édition, 1997.

[Mos90] P.D. Mosses. Denotational Semantics. In *Handbook of Theoretical Computer Science*, chap. 11, pages 575–631. Elsevier Science Publishers & MIT Press, 1990.

[SA98] Sinan Si Alhir. *UML in a nutshell*. O'Reilly & associates, 1998.

[SA99] Sinan Si Alhir. *Extending the Unified Modeling Language (UML)*, janvier 1999. disponible sur le site www de l'auteur.

[Sed92] R. Sedgewick. *Algorithms in C++*. Addison-Wesley, 1992.

[WK99] J. Warmer et A. Kleppe. *The Object Constraint Language — Precise Modeling with UML*. Addison-Wesley, 1999.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes 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 Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

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