A Meta-Model Facility for a Family of UML Constraint Languages

Tony Clark[1], Andy Evans[2], and Stuart Kent[3]

[1] Department of Computing, Kings College, London, UK
anclark@doc.kcl.ac.uk
[2] Department of Computer Science, University of York, York, UK
andye@cs.york.ac.uk
[3] Computing Laboratory, University of Kent, Canterbury, UK
s.j.h.kent@ukc.ac.uk

Abstract. With the move towards UML becoming a family of modelling languages, there is a need to view the Object Constraint Language in the same light. The aim of this paper is to identify a meta-modelling facility that encompasses the specification of the semantics of a family of object constraint languages. This facility defines a common set of model concepts, semantic domain concepts and semantic mappings that can be conveniently reused when constructing new family members.

1 Introduction

The Object Constraint Language (OCL) [Warmer98] is a language for expressing constraints on UML models. Recently, significant efforts have been made to provide a more precise description of the OCL. For example, a meta-model for the abstract syntax of OCL [Richters98] has been proposed. An outline meta-model of its semantics is presented in [Kent99], whilst a variety of other work has developed formal descriptions of the OCL semantics (see [Clark00a] for an overview).

At the time these approaches were developed, UML was viewed as a single, albeit large, modelling language. However, with the advent of UML 2.0 it is likely that UML will become a family of languages [Cook00]. In UML 2.0, it is intended that the semantics of each family member will be encapsulated by a single UML profile. Each profile will tailor UML to a specific application domain. For instance, one might have a UML for real-time modelling, e-business modelling or business process modelling. The impact of these changes will not just affect the diagrammatical components of UML. Specific extensions of OCL are also likely to be required. Indeed, there is early evidence of the need for such extensions in industry [Kleppe00, Knapman00a, Knapman00b]. The OMG's recent proposal for MDA (model driven architecture) [OMG01] also mandates the use of profiles, each with a potentially different constraint language.

A problem faced by the developers of profiles is to avoid the task of re-inventing meta-models for families of related languages. The aim of this paper is to show how the semantics of a family of OCL languages might be precisely captured within a meta-model. In doing so, it aims to highlight two key mechanisms which can aid in the large grained reuse of constraint language meta-models: language definition patterns and package extension and import.

The work presented in this paper is based on the Meta-Modelling Facility (MMF), which comprises a meta-modelling language (MML) a meta-modelling tool (MMT) and a method (MMM). This work has arisen out of an IBM funded project [Clark00] to investigate the feasibility of re-architecting UML as a family of languages. The work is currently being prepared as input to the UML 2.0 revision process under the auspices of the precise UML (pUML) group (http://www.puml.org).

This paper is structured as follows: Section 2 gives an overview of the MMF and the basic patterns and architecture used to describe meta-models. Section 3 presents the core meta-model of MML (a typical UML like core meta-model). Section 4 then extends the core model with the features of a static constraint language. Finally, section 5 discusses some general approaches to constructing profiles.

2 The MMF

The Meta-Modelling Facility (MMF) [Clark00a] aims to provide a modular and extensible method for defining and using modelling languages. MMF comprises a language (MML), used to write language definitions, a tool (MMT) used to interpret those definitions, and a method (MMM) that provides guidelines and patterns for good practice in language definition. This section introduces the language and the tool.

2.1 The Language

MML is a language for meta-modelling, which incorporates a subset of the UML. The language is defined in itself using a precise meta-modelling approach, in which mappings are defined between its modelling concepts and the concepts in its semantic domain. The development of the language is ongoing and has been supported by IBM.

The main modelling features of MML are packages, classes, attributes and a constraint language. These are introduced briefly below, being careful to identify differences with the standard UML interpretation. Examples of their use can be found in subsequent sections.

A package is a container of classes and/or other packages. This is familiar UML. Perhaps more unfamiliar is the ability to specialize and import packages, which has been borrowed from Catalysis [D'Souza98] and enhanced. Packages and package import and specialisation are the key constructs in MML that support modular, extensible definitions of languages based on a set of fundamental patterns. Our contribution is to provide a precise, tool-supported definition of this concept, and to show it can facilitate the development of precise meta-modelling definitions.

Package import is shown by placing a UML dependency arrow between the packages. The child package copies all (and therefore contains all) of the contents of the parent package. In addition, it may specialise the copied contents, in which case the copies are hidden and only the specialised contents are shown. A class, package, attribute or method within the scope of the package may be renamed through specialisation. This is indicated by a specialisation annotation on the corresponding import arrow. So if a package B imports A, copying class C and specialising it to D and copying attribute a in C and specialising it to b, one would annotate the arrow with [D<C[b<a]]. This is a nested syntax, reflecting the notion that there is a containment hierarchy of modelling elements.

An example of package import is shown in Figure 1. The elements in red (lighter colour) represent those elements that appear in the child package by virtue of import from the parent(s). The blue elements (darker colour) have been added separately. They have been included to help clarify the meaning of package import. They could be generated automatically from the import relationship and renaming annotations, and, indeed, are by MMT. When an attribute is imported, its multiplicity may change, for example from "*" (zero or many) to "1" (exactly one). This can be included in the renaming annotation, so if, in the preceding example, the multiplicity of the attribute a was also specialised to 1, the renaming annotation would have been [D<C[b<a[1<*]].

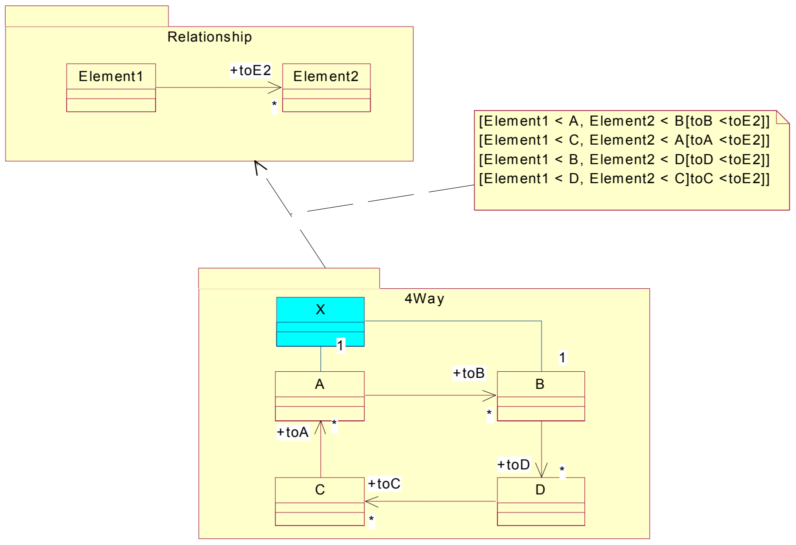

Relationship
Element1
+toE2
Element2
*
[Element1 < A, Element2 < B[toB <toE2]]
[Element1 < C, Element2 < A[toA <toE2]]
[Element1 < B, Element2 < D[toD <toE2]]
[Element1 < D, Element2 < C]toC <toE2]]
4Way
X
1
1
A
+toB
B
*
+toA
*
+toD
*
C
+toC
D
*

Figure 1 Package Import

A generalisation arrow between packages denotes package generalisation. Package generalisation results in the contents of the package being specialised by the contents of the child class package. A renaming list can be attached to the generalisation arrow in a similar way to that used by import.

2.2 The MML Architecture and Patterns

The MML meta-model is specifically structured to provide a framework that can be readily used to construct precise definitions of families of modelling languages. To support this, a modular architecture is adopted, in which variation points are separated into packages. For example, MML includes packages for data types, extension mechanisms and constraints, all of which are extended from a single core package (see Figure 2). The aim is that new languages can be rapidly constructed by changing or extending specific packages. For example, one could readily adapt the data types used by a specific modelling language by extending the data types package. If necessary, specific parts of the meta-model can be designated as mandatory, thereby ensuring cross language compatibility.

MML makes use of package generalisation to facilitate the construction of a layered framework of language definition components.

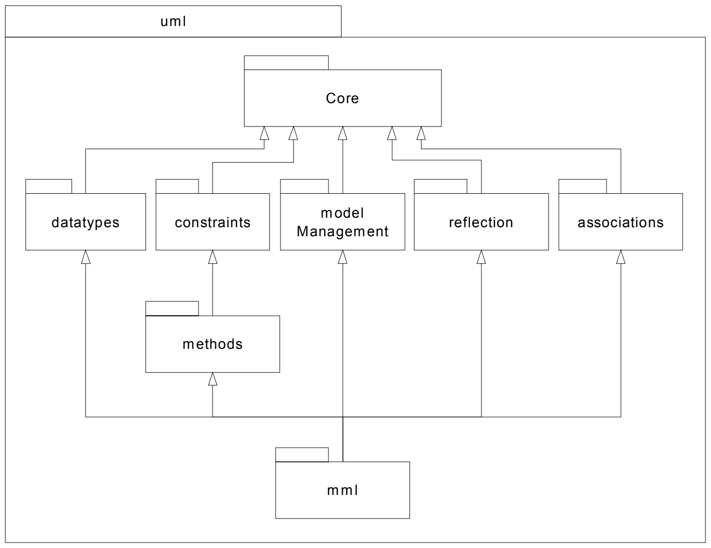


Figure 2 The MML Architecture

MML also aims to identify patterns of language elements. These capture cross cutting structures and relationships that appear throughout meta-models. They include language definition patterns and a collection of language element patterns. The language definition pattern is used to structure the concrete syntax, abstract syntax and semantic elements of languages and to define mappings between them. This pattern is shown in Figure 3.

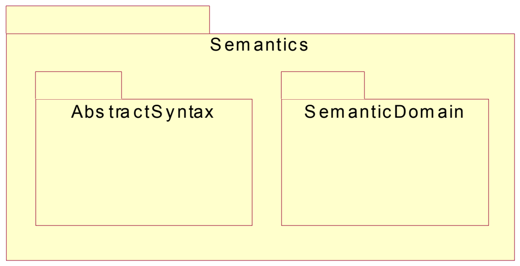


Figure 3 The MML Language Definition Pattern

2.2.1 Language Element Patterns

The language element patterns capture common structural relationships between language elements. Six core patterns are shown in Figure 4. A brief description of each pattern follows:

Instance: This pattern captures the semantic relationship found between types and their “instances”. For example, instances of classes are objects. Note that in this pattern, abstract syntax concepts and semantic domain concepts are separated into different packages. Types are viewed as abstract syntax concepts, whilst instances belong to the semantic domain.
Containment: Elements may contain other elements. An example here is a class containing its attributes, or a package containing its classes.
Conformance: Elements conform to other elements; for instance, classes may conform to their parents.
Relationship: Many elements are related to other elements.
Comparable: Elements are comparable. For example a package can be compared with another package, or an object compared to another object. An element may be sub-equivalent to another element if they have a subset of the properties necessary for equivalence. If two elements are sub-equivalent to each other they are equivalent.
Inheritance: Elements may inherit features from their parents. All parents is the transitive closure of all parents of a elements. This pattern does not permit circular inheritance.

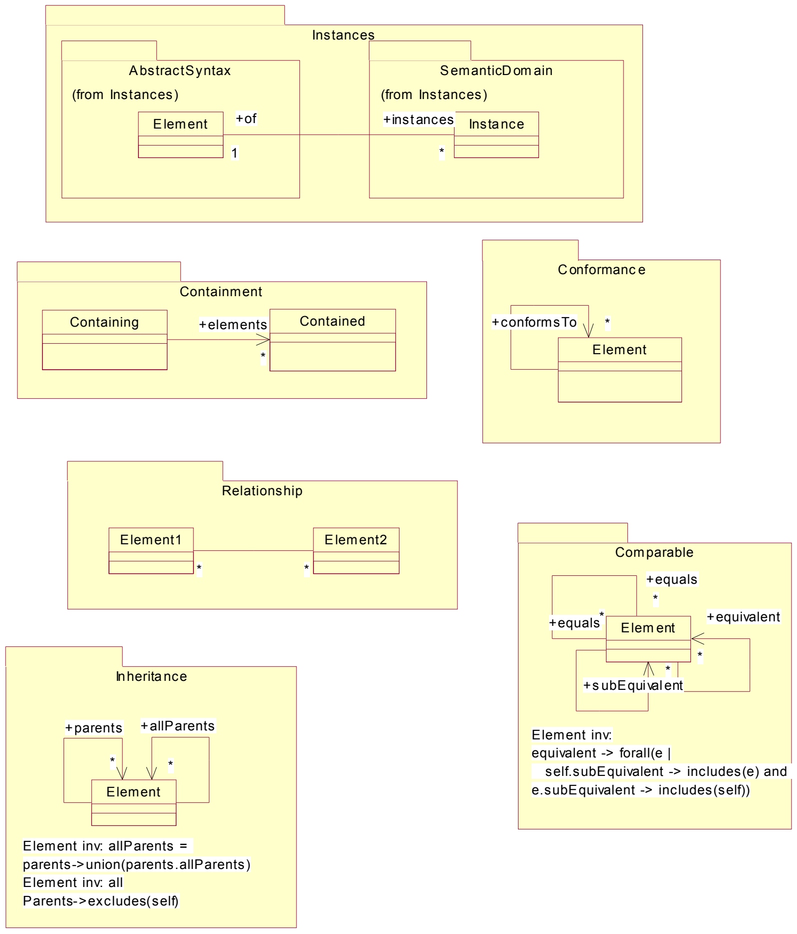


Figure 4 Language Element Patterns

2.2.2 Composite Patterns

The core language element patterns can be combined to describe more complex patterns. The first of these is shown in Figure 5. It describes the relationship between containers and their instances. A container has instances whose elements are instances of the elements of the container. The OCL constraint ensures that the relationship is commutative.

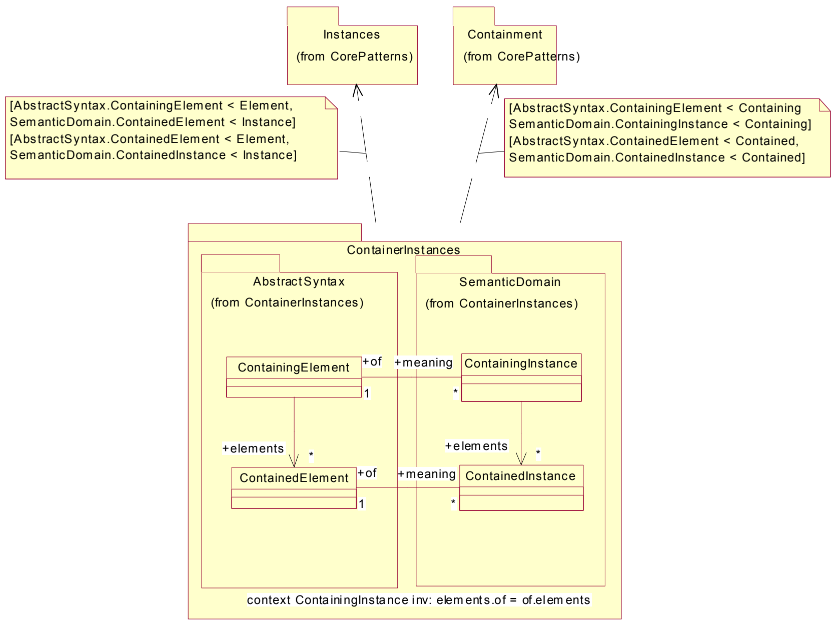

Instances
(from CorePatterns)
Containment
(from CorePatterns)
[AbstractSyntax.ContainingElement < Element, SemanticDomain.ContainedElement < Instance]
[AbstractSyntax.ContainedElement < Element, SemanticDomain.ContainedInstance < Instance]
[AbstractSyntax.ContainingElement < Containing SemanticDomain.ContainingInstance < Containing]
[AbstractSyntax.ContainedElement < Contained, SemanticDomain.ContainedInstance < Contained]
ContainerInstances
AbstractSyntax
(from ContainerInstances)
SemanticDomain
(from ContainerInstances)
ContainingElement
+of
+meaning
ContainingInstance
1
*
+elements
*
+elements
*
ContainedElement
+of
+meaning
ContainedInstance
1
*
context ContainingInstance inv: elements.of = of.elements

Figure 5 Container Instances

Another useful pattern relates to conformance. Conformance is a general property that relates language elements of the same type. Many language elements can be compared to decide whether they are conformant to the properties of their parents.

Container conformance, as shown in Figure 6, is one of many possible different kinds of conformance pattern. It states that two containers are conformant if their contained elements are conformant. As an example, consider treating a class as a container of its operations. The result of applying this pattern would be a rule for class containment that requires that for every operation belonging to a parent class, there must be an operation belonging to the child class that it is conformant to.

Finally, a possible static semantics for container conformance is described by the pattern shown in Figure 7. This states that for an element to be conformant, its instances must be conformance to those of its parent. Here, conformance implies structural conformance, i.e. both the element and its instances must have the same structure as those of their parents.

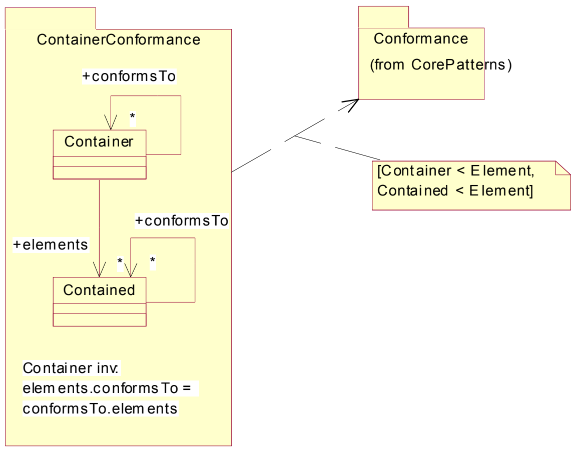


Figure 6 Container Conformance

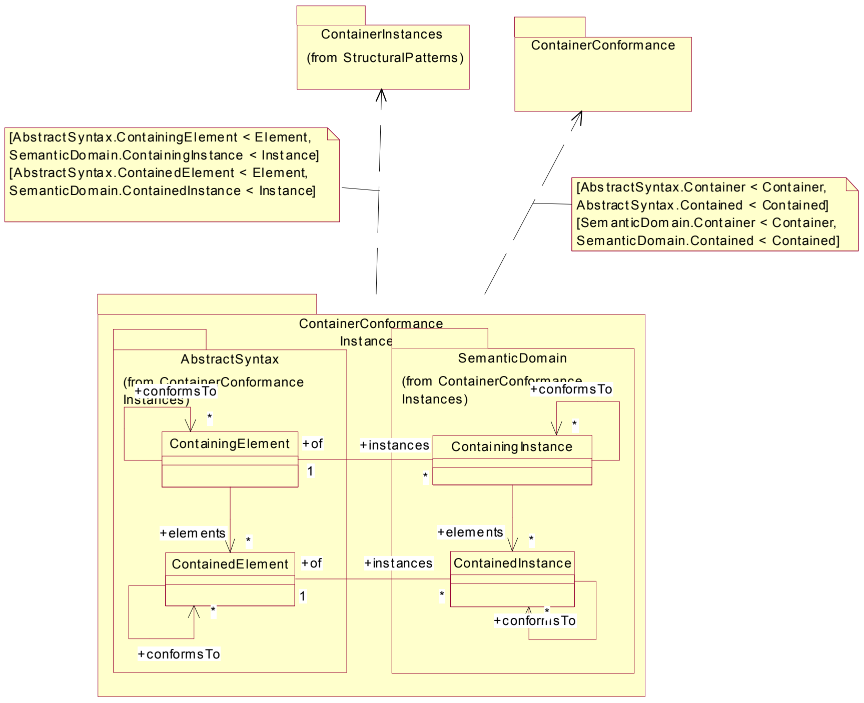
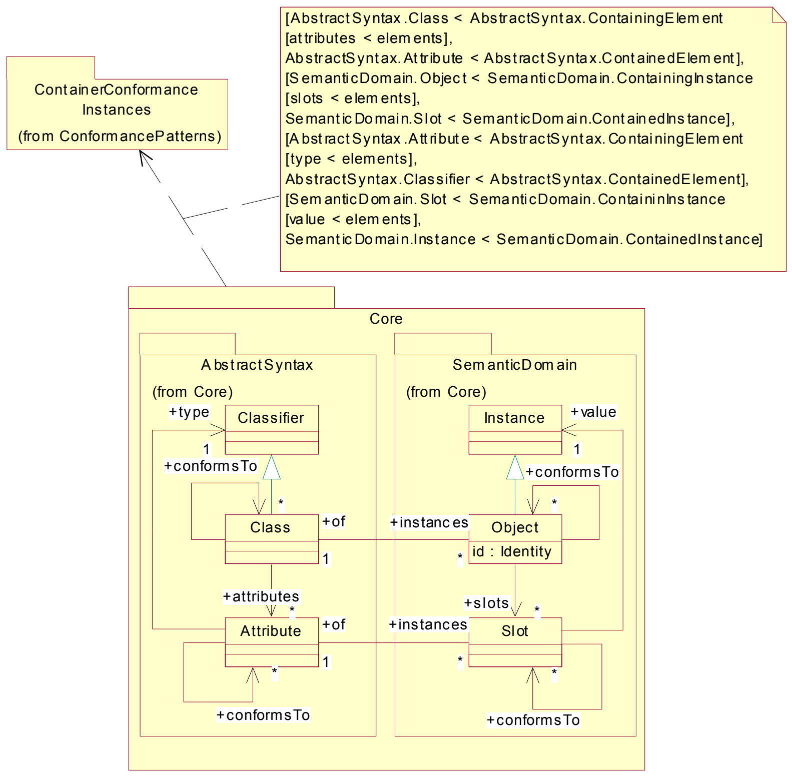

ContainerInstances
(from StructuralPatterns)
ContainerConformance
[AbstractSyntax.ContainingElement < Element, SemanticDomain.ContainingInstance < Instance]
[AbstractSyntax.ContainedElement < Element, SemanticDomain.ContainedInstance < Instance]
[AbstractSyntax.Container < Container, AbstractSyntax.Contained < Contained]
[SemanticDomain.Container < Container, SemanticDomain.Contained < Contained]
ContainerConformance Instance
AbstractSyntax
(from ContainerConformance Instances)
SemanticDomain
(from ContainerConformance Instances)
+conformsTo
*
ContainingElement
+of
1
+instances
*
ContainingInstance
+conformsTo
*
+elements
*
ContainedElement
+of
1
+instances
*
ContainedInstance
+elements
*
+conformsTo
*
+conformsTo
*

Figure 7 Container Conformance Instances

3 The Core Package

The core package (see Figure 8) provides the core modelling concepts required for meta-modelling in the MML. It is used here to illustrate the typical core modelling components of a static UML-like language. Note that we will not address the behavioural aspects of the core model (these will be described elsewhere). In Section 4, it will be extended with a static constraint language.

The core is divided into two packages: abstract syntax and semantic domain. As described above, an instances association is used to map between elements of the abstract syntax and elements of the semantic domain.



```
[AbstractSyntax.Class < AbstractSyntax.ContainingElement
[attributes < elements],
AbstractSyntax.Attribute < AbstractSyntax.ContainedElement],
[SemanticDomain.Object < SemanticDomain.ContainingInstance
[slots < elements],
SemanticDomain.Slot < SemanticDomain.ContainedInstance],
[AbstractSyntax.Attribute < AbstractSyntax.ContainingElement
[type < elements],
AbstractSyntax.Classifier < AbstractSyntax.ContainedElement],
[SemanticDomain.Slot < SemanticDomain.ContaininInstance
[value < elements],
SemanticDomain.Instance < SemanticDomain.ContainedInstance]
```

Figure 8 Core Package

The container conformance instances pattern has been applied twice to construct a type/instance relationship for classes and attributes. The instances of a class are given by the collection of objects whose contents (slots) conform to the contents of the class (attributes). Objects and slots conform to the containership of class and attribute and are structurally conformant.

The effect of applying the patterns generates the following constraints:

[1] The slots of an object commute with the attributes of the object's class (from ContainerInstances):

```
context  Object inv: slots.of = of.attributes
```

[2] The value of a slot commutes with the type of the slot's attribute (from ContainerInstances):

```
context  Slot inv: value.of = of.type
```

A number of conformance constraints are also generated for classes, attributes, objects and slots. For example, a class conforms to another class if its instances (objects) conform to the instances of the parent.

4 Constraints

This section extends the core modeling concepts defined in section 3 with a simple static language for describing constraints. To support a family of constraint languages, the definition is structured to make a clear distinction between core concepts and language specific concepts.

The core constraints package (see Figure 9) identifies concepts common to all object constraint languages. We assume that classifiers are associated with expressions. Expressions describe the *constraints* that must be evaluated on their instances. Expressions may also have sub-expressions and are associated with a collection of free variables. An important constraint that relates to free variables is that they are inherited by sub-expressions.

The static semantics of expressions are given by calculations. A calculation associates an expression and an environment (a set of variable bindings) with a value. The value is the result of evaluating the expression in the context of the binding. A distinguished variable, self, identifies the instance the expression is being evaluated against.

In the core constraints package, the pattern for container conformance is applied to classifiers, their expressions and their semantic domains: instances and calculations. This imposes properties of structural conformance and containership. The same properties also apply to expressions, their free variables, calculations and bindings. An implication is that the conformance of one expression to another is expressed in terms of substitutability of the values that are obtained from evaluating an expression. An expression is substitutable for another expression if the set of calculations, variable bindings and results satisfying the expression is a subset of those of its parent.

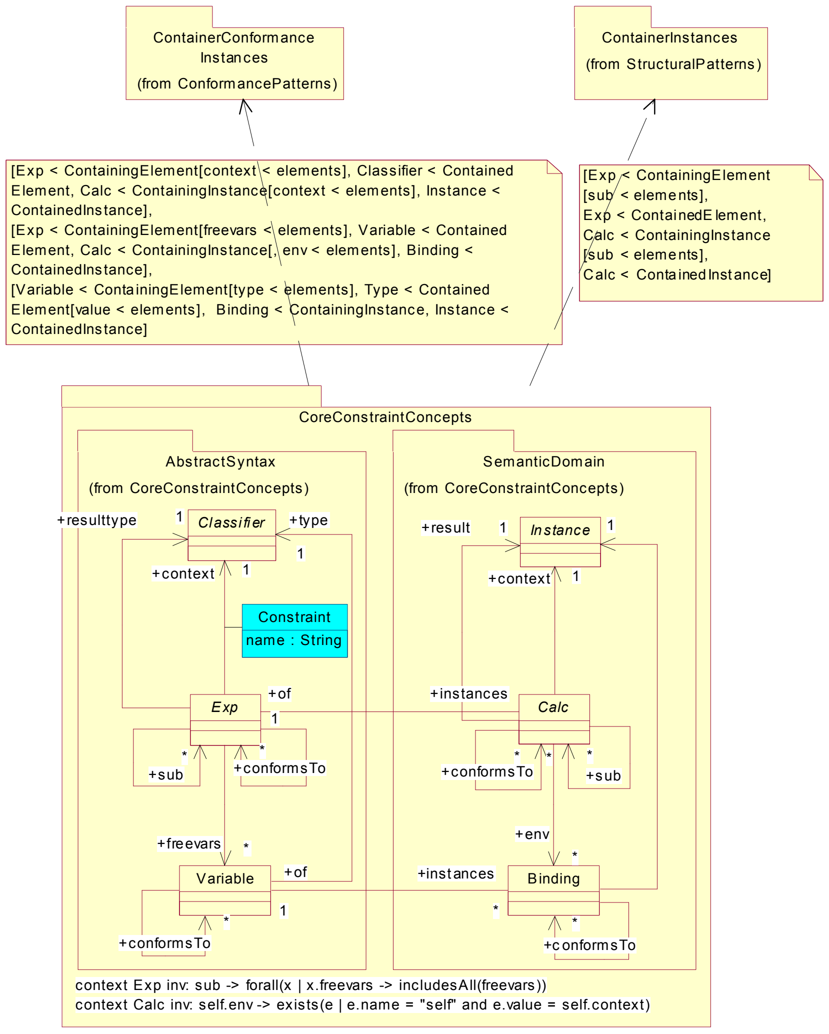

ContainerConformance Instances
(from ConformancePatterns)
ContainerInstances
(from StructuralPatterns)
[Exp < ContainingElement[context < elements], Classifier < Contained Element, Calc < ContainingInstance[context < elements], Instance < ContainedInstance],
[Exp < ContainingElement[freevars < elements], Variable < Contained Element, Calc < ContainingInstance[, env < elements], Binding < ContainedInstance],
[Variable < ContainingElement[type < elements], Type < Contained Element[value < elements], Binding < ContainingInstance, Instance < ContainedInstance]
[Exp < ContainingElement [sub < elements], Exp < ContainedElement, Calc < ContainingInstance [sub < elements], Calc < ContainedInstance]
CoreConstraintConcepts
AbstractSyntax
(from CoreConstraintConcepts)
SemanticDomain
(from CoreConstraintConcepts)
+resulttype
1
Classifier
+type
1
+context
1
Constraint
name : String
Exp
+of
1
*
+sub
*
+conformsTo
+freevars
*
Variable
+of
1
*
+conformsTo
+result
1
Instance
1
+context
1
+instances
Calc
*
+conformsTo
*
*
+sub
+env
*
+instances
Binding
*
*
+conformsTo
context Exp inv: sub -> forall(x | x.freevars -> includesAll(freevars))
context Calc inv: self.env -> exists(e | e.name = "self" and e.value = self.context)

Figure 9 Core Constraints Semantics

4.1 Language Specific Constraints

This section gives an example of an extension to the core constraint package (Figure 10). It defines the core language expressions of a simple OCL-like profile for describing static constraints. These include logical expressions (and, not, equals, includes), slot references, variables and iterations. Note that most other constructs of an OCL-profile, for example or, collect, set union, could be easily defined in terms of the basic expressions defined here. This could be achieved by extension of translation (see Section 5).

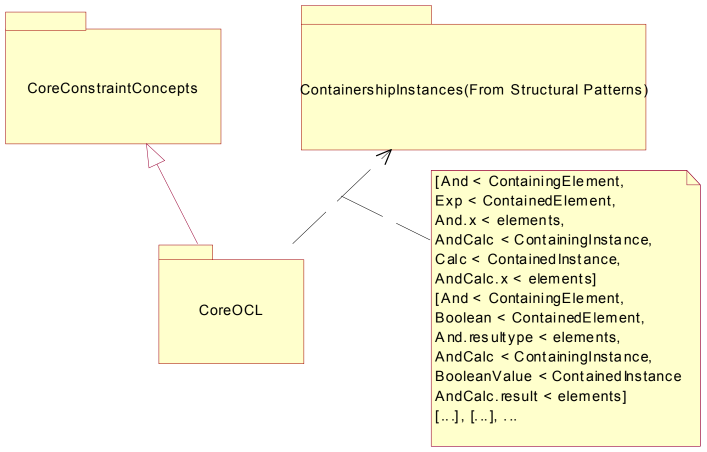

CoreConstraintConcepts
ContainershipInstances(From Structural Patterns)
CoreOCL
[And < ContainingElement,
Exp < ContainedElement,
And.x < elements,
AndCalc < ContainingInstance,
Calc < ContainedInstance,
AndCalc.x < elements]
[And < ContainingElement,
Boolean < ContainedElement,
And.resultype < elements,
AndCalc < ContainingInstance,
BooleanValue < ContainedInstance
AndCalc.result < elements]
[...], [...], ...

Figure 10 OCL Language Extension

The abstract syntax for the core OCL package is shown in Figure 11, whilst the corresponding semantic domain package is shown in Figure 12. Applying the containership instances pattern to the core OCL package results in the appropriate instances associations and constraints being set up between each expression in the abstract syntax and its corresponding semantic domain element (note that not all the relevant substitutions are given in Figure 10 for brevity). For example, the instances of an "And" expression will be "AndCalc", and so on.

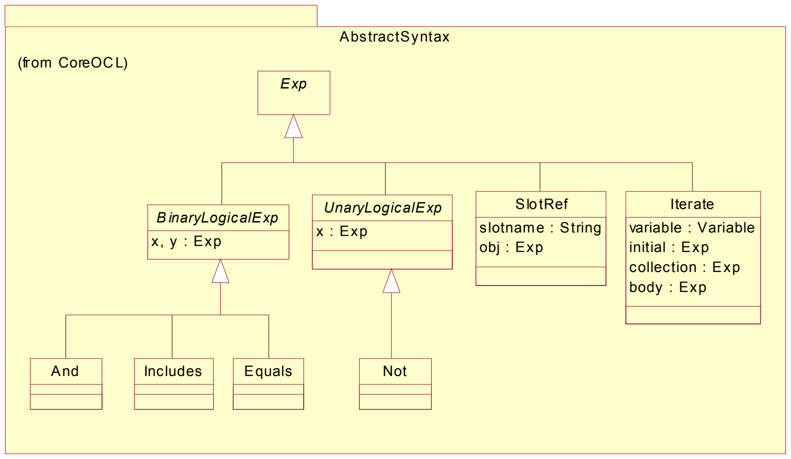


Figure 11 Core OCL Abstract Syntax

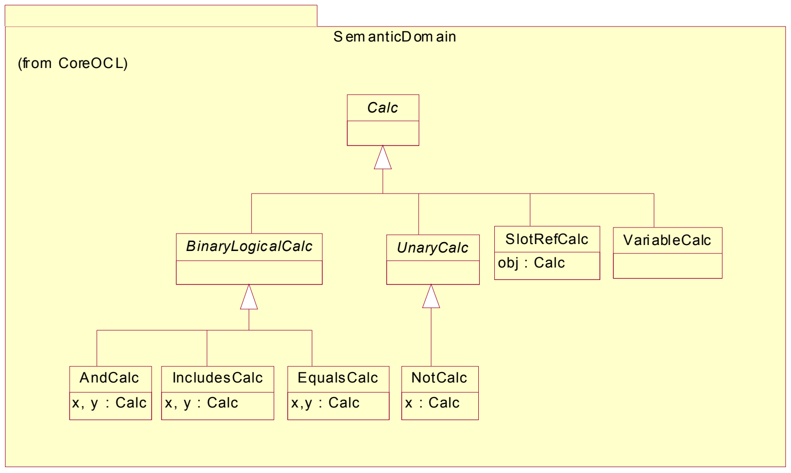


Figure 12 Core OCL Semantic Domain

Finally, to complete the meta-model, a constraint is required for each calculation to describe its evaluation. For example, the result of an "And" expression is the conjunction of the value of its x and y expressions:

[1] And implies conjunction.

```
context AndCalc inv:
        self.value = self.x.value and self.y.value
```

Similar constraints are required for other calculations, for example:

[2] The result of a not expression is the negation of the value of its x expression.

```
context NotCalc inv:
        self.value = not self.x.value
```

[3] The result of an includes expression is true if the value of its y expression is a member of its x expression.

```
context Includes inv:
self.value = self.x.value ->includes(self.y.value)
```

[4] The result of an equals expression is true if the value of its y expression is equal to its x expression.

```
context Equals inv:
        self.value = self.x.value = self.y.value
```

[5] The result of a slot reference calculation is the value of the slot belonging to obj whose slotname is equal to the referenced slot name.

```
context SlotRefCalc inv:
        self.obj.value.slots -> exists(b |
                b.value = self.value and b.name = self.exp.slotname)
```

[6] The result of evaluating a variable expression is the variable's binding.

```
context VariableCalc inv:
        self.env -> exists(e |
                e.value = self.value and e.variable.name = self.exp.name)
```

A similar approach can be used to define the semantics for iterate, although the resulting constraints are too large to include here. The reader is referred to [Clark00] for an illustration of the approach.

4.2 Example

Figure 13 gives a simple example of a constraint expressed as an instance of the core OCL meta-model. The expression “x > 5” is associated with a class A, which contains an attribute x of type Integer (we assume there is an appropriate package of data types and data values).

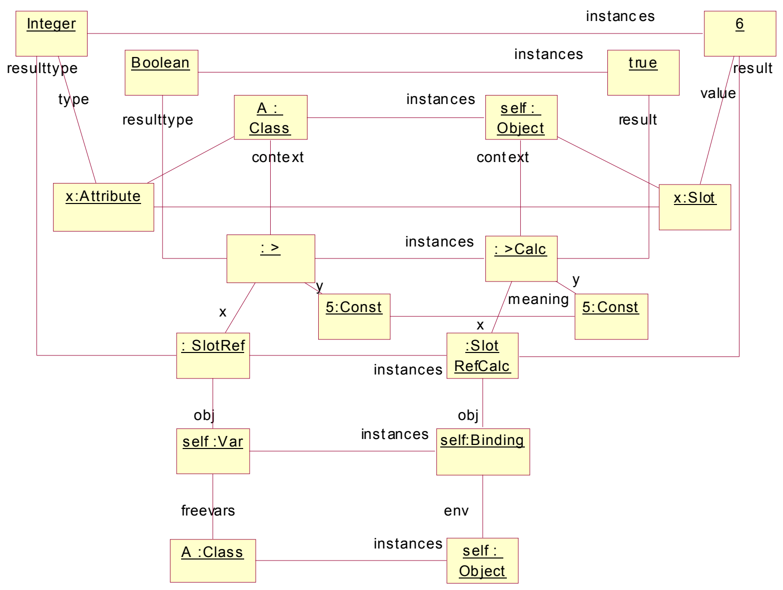


Figure 13 Constraint Instance Example

An object satisfies the properties of A if:

- The object contains a slot corresponding to the attribute of A. This follows from the containership instances pattern.
- There is a calculation which evaluates the “x > 5” expression of A. This follows from the containership instances pattern, which ensures that any calculations associated with A conform to A’s constraint expression.
- The result of comparing the value of the slot is greater than the constant “5” is true. This follows from the “greater than” evaluation constraint.

5 Profiles

This section describes two strategies that can be used to defining new profiles in MML. These strategies form an important part of the meta-modelling method (MMM). The first strategy, which we call the *extensional* strategy, involves extending appropriate core meta-model packages using package extension. This is the approach used in this paper. We identified core-modelling elements common to a number of constraint languages and extended their definition, gradually constructing a layered definition of a simple constraint language (a profile). Further extensions to this language can be described by additional packages. Figure 14 illustrates how a family of constraint languages might be constructed by extending the core package.

The advantage of the extensional approach is that is intuitive – extending a language simply involves choosing an appropriate extension point and then defining the special cases. Furthermore, meta-models built in this way tend to have pleasing structural properties, for example one soon identifies a structure preserving relationship between elements in the abstract syntax and elements in the semantic domain.

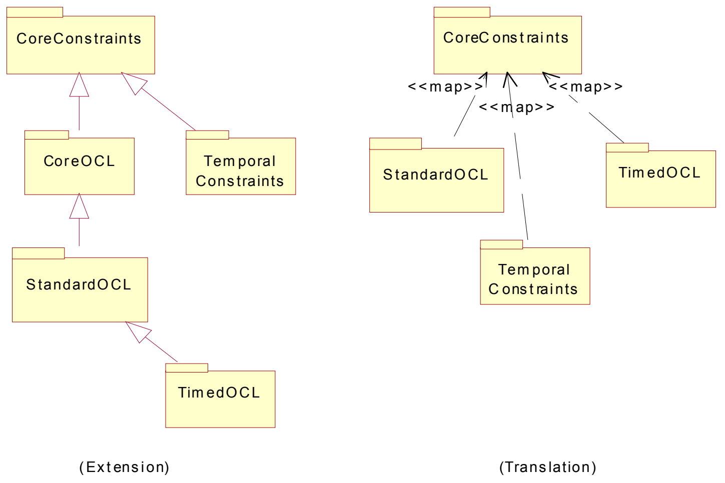


Figure 14 Extension versus Translation

However, when it comes to implementation, the extensional approach suffers from weak interoperability. Consider two implemented profiles that extend a common core package. Tools that implement these profiles cannot readily share facilities for analysing properties of models because the relationship between the two models is based on the arbitrary extension of their profiles.

This leads to an alternative approach to profile construction, which we call the *translation* or *interpretative* approach (Figure 14). Here, a common core language is defined with a minimal but expressive collection of modelling elements. Profiles are constructed by defining mappings from elements in the profile to elements in the core language. The key advantage of this approach is as follows - any facilities offered by a tool that implements the core language can be applied to any profile, provided that the profile is first translated into the core language. The trade-off here is that the core language must be sufficiently expressive to capture a wide variety of useful profile interpretations.

To support the definition of a family of constraint languages using the translation approach, a core constraint language must provide a minimal but expressive collection of logical expressions. For example, a core OCL-like constraint language should at least have the basic expressions of predicate and first-order logic, and arguably, higher order logics. Identifying the most appropriate expression set is not easy. Luckily, existing formal languages, many of which have been developed with minimal expression sets in mind, can guide us.

5 Conclusions

Over the next few years it is likely that an increasing number of UML profiles will be required. However, designing a profile from scratch is both time-consuming and inefficient. It ignores the significant amount of work invested in the development of other profiles. Instead, common variations between profiles must be identified and reused. A modular architecture should be designed in a way that allows new components of profiles to be readily plugged in to a common specification framework. The use of patterns to help identify common meta-modelling structures is essential if important cross cutting concerns such as conformance, modularity and consistency are to be properly addressed.

This paper has proposed one such framework for constraints based on the MMF. Constraint languages are likely to vary across profiles, and may even be shared. Constraint languages are also difficult to define from scratch, as they require a detailed knowledge of the principles of constraint language semantics. Thus, identifying core constraint language concepts is an essential step towards facilitating the development of families of constraint languages. This paper has defined such a core, and has described two approaches to viewing constraint language profiles as extensions or interpretations of the core.

References

[Warmer98] J.Warmer and A.Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[Richters98] M.Richters and M.Gogolla. *On Formalising the UML Object Constraint Language.* In Tok Wang Ling, Sudha Ram and Mong Li Lee, editors, *Proc 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of LNCS, pages 449-464, Springer, 1998.

[Kent99] S.Kent, S.Gaito, N.Ross. *A meta-model semantics for structural constraints in UML* In H.Kilov and B.Rumpe, editors, *Behavioural Specifications for Businesses and Systems,* Kluwer, 1999.

[Clark00a] A.Clark, S.Kent and J.Warmer. OCL Semantics FAQ, Workshop on the Object Constraint Language (OCL), Computing Laboratory, University of Kent, Canterbury, UK. Internet: http://www.cs.ukc.ac.uk/research/sse/oclws2k/index.html, March 2000.

[Evans00] A.Evans, S.Kent and B.Selic, editors, *Proc 3rd Int. Conf. The Unified Modeling Language (<<UML>>2000)*, volume 1949 of LNCS, Springer, 2000.

[Cook00] S.Cook. *The UML Family: Profiles, Prefaces and Packages.* In [Evans00].

[Kleppe00]] A.Kleppe and J.Warmer. *Extending OCL to Include Actions.* In [Evans00].

[Knapman00a] J.Knapman. *Business- Oriented Constraint Language*. In [Evans00].

[Knapman00b] J.Knapman. *Statistical Constraints for EAI*. In [Evans00].

[OMG01] *Model Driven Architecture*. Available from http://www.omg.org/mda.

[Clark00b] A.Clark, A.Evans, S.Kent. *Rearchitecting UML as a Family of Language using a Precise OO Meta-Modelling Approach.* Available from http://www.puml.org/mmf, 2000.

[Clark01] A.Clark, A.Evans, S.Kent. *Engineering Modelling Languages: A Precise Meta-Modelling Approach.* Available from http://www.puml.org/mmf/langeng.ps, 2001.

[D'Souza98] D.D'Souza, A.Wills. *Object Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.