# Advancing Generic Metamodels

Henning Berg
Department of Informatics
University of Oslo
hennb@ifi.uio.no

Birger Møller-Pedersen
Department of Informatics
University of Oslo
birger@ifi.uio.no

Stein Krogdahl
Department of Informatics
University of Oslo
steinkr@ifi.uio.no

## ABSTRACT

Domain-Specific Languages (DSLs) allow modelling concerns at a high abstraction level. This simplifies the modelling process and ensures that non-technical stakeholders can be more closely involved in software development. However, increasing the abstraction level causes details of the problem domain to be excluded from the problem space. In some situations, this may render a DSL useless since required details can not be captured by the language. In this paper we explore how generic metamodels can be parameterised to model additional details and thereby increase the reuse value of DSLs.

## Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.2 [**Programming Languages**]: Specialised application languages; D.3.3 [**Programming Languages**]: Language constructs and features; D.3.2 [**Programming Languages**]: Modules, packages

## General Terms

Languages, Design, Genericity

## Keywords

Metamodelling, specialised languages, DSLs, generic types, class nesting, virtual classes

## 1. INTRODUCTION

Generics were first introduced by the programming language Ada in 1983 [1]. Their purpose was to enable parameterisation of packages and subprograms. Today, generic types is a well-known mechanism supported by many programming languages. The most common usages are construction of generic data structures and algorithms.

Another popular mechanism in object-oriented languages is class nesting. Class nesting allows defining inner classes in the context of an enclosing class. Nested classes are often used to define local data structures, where the details of the data structure are not of interest outside of the enclosing class.

In the last decade, model-driven software engineering [4] has established a strong foothold in industry. This includes language-driven approaches [10] where languages are considered first-class entities of the software development process. An important activity in these approaches is the ability to create metamodels. A metamodel can be used to define a language by describing its abstract syntax and semantics. Metamodelling is the process of creating metamodels.

Metamodelling is for the most part concerned with making simple class models. A class model consists of metaclasses that reflect concepts of the problem domain. Generic metaclasses are supported by both *Eclipse Modeling Framework (EMF)* [2] and Kermeta [6]; that is, metaclasses with one or more formal type parameters. However, the application and usage scenarios of generics in metamodelling are not obvious. In this paper, we discuss how generic metaclasses and class nesting can be combined to create generic metamodels. A generic metamodel consists of an enclosing class that encapsulates a set of metaclasses. These metaclasses constitute the metamodel. The enclosing class has one or more formal type parameters that are accessible from the metaclasses. A generic metamodel can be parameterised with the purpose of configuring or adding new constructs to a language. This addresses how languages can be customised for different projects and processes, which is identified as desirable in the industry [9].

We will also illustrate how virtual classes [5] can be applied in metamodels for defining variability points, and discuss how model conformance can be preserved when using generic metamodels.

We will use a textual syntax based on Kermeta to illustrate the ideas of this paper[1]. Kermeta is an object-oriented metalanguage and framework for language design. It allows defining the structure and semantics of metamodels. The ideas presented are applicable to other frameworks for metamodelling, e.g. EMF.

A *Domain-Specific Language (DSL)*, named *Simple Math*, will be used to describe the ideas. The purpose of such

[1]Class nesting and virtual classes are not supported by Kermeta.

language is to model mathematical problems, which can be calculated using the dynamic semantics of the language. To keep the overview, only a small number of language constructs are included in the language definition. The language is inspired by *Matlab* which is a popular language for technical computations. The metamodel for Simple Math is given in Figure 1.

```
package simpleMath;

class Program {
  attribute expressions : Expression[0..*]
  attribute plots : Plot[0..*]

  // Shows values and graphs on screen
  operation exec() is do ... end
}
abstract class Expression {
  operation eval() : Real is abstract
}
abstract class Plot {
  reference expressions : Expression[1..*]
  attribute caption : String
  operation plot() is abstract
}
class ScalarPlot inherits Plot { ... }
class GraphPlot inherits Plot { ... }
class Add inherits Expression {
  attribute e1 : Expression[1..1]
  attribute e2 : Expression[1..1]
  operation eval() : Real is do
    result := e1.eval() + e2.eval()
  end
}
class Sub inherits Expression { ... }
class Multi inherits Expression { ... }
class Number inherits Expression {
  attribute value : Real
  operation eval() : Real is do
    result := value
  end
}
```

**Figure 1: A DSL for modelling of mathematical calculations**

As can be seen in Figure 1, the language comprises seven constructs. A program consists of expressions and plots. An Expression is the smallest unit of computation. There are three types of binary expressions: Add, Sub and Multi, representing addition, subtraction and multiplication, respectively. Number is an unary expression and represents single floating-point numbers, e.g. 2.718 and 3.14. The result of each calculation can be shown on screen using one of the two plot types: a ScalarPlot or a GraphPlot.

$$GraphPlot_{x=5}[2+5, 2\times 4, 9]$$

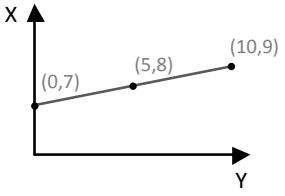


**Figure 2: Example model and printout to screen**

As an example, a graph can be made by adding several expressions to the same graph plot. Figure 2 illustrates this in a textual concrete syntax. Executing the program of Figure 2 gives a line with the three points: (0,7), (5,8) and (10,9).

## 2. TOWARDS GENERIC METAMODELS

A metamodel comprises a set of language constructs that closely reflect concepts of the language's problem domain. In the context of DSLs, these constructs usually are on a high abstraction level. This allows creating models that are more expressive. However, the high abstraction level can also make it difficult to model variations that are not captured by the constructs. In addition, reuse of a DSL may be impossible if it lacks required constructs. Both issues can be addressed using generic metamodels.

A generic metamodel is defined using class nesting. Specifically, the language constructs are encapsulated by an enclosing class. The enclosing class has one or more formal type parameters which are accessible from all language constructs. Thus, the metamodel can be parameterised by different metaclasses. The idea is illustrated in Figure 3. Here, GenericMetamodel has two type parameters: T1 and T2. These are bound by B1 and B2, as defined outside the scope of the generic metamodel. Notice that the inner classes of a generic metamodel are not grouped in a package. The enclosing class is sufficient.

```
package genericMetamodel;

class B1 { ... }
class B2 { ... }

class GenericMetamodel <T1 : B1, T2 : B2> {
  class ConstructX { attribute t1 : T1[0..1] }
  class ConstructY { attribute t2 : T2[0..*] }
  ...
}
```

**Figure 3: Conceptual overview of a generic metamodel**

All language constructs are defined as non-static inner classes, and are therefore only available through an instance of the enclosing class. Instantiation of the generic metamodel gives a parameterised metamodel in the scope where the instantiation takes place. Figure 4 illustrates this. Notice that it is possible to instantiate a generic metamodel without providing actual type arguments if usage of the type parameters is optional. Refer the multiplicities in Figure 3. (This would require additional static semantics.)

```
package genericMetamodel;

class X inherits B1 { ... }
class Y inherits B2 { ... }

var gm1 : GenericMetamodel<X,Y> init
  GenericMetamodel<X,Y>.new

var gm2 : GenericMetamodel init
  GenericMetamodel.new
```

**Figure 4: Instantiations of a generic metamodel**

## 2.1 Adding new language constructs

As argued, there may be situations where a language lacks the ability to express a concern in adequate detail. Let us return to the math language example. In some applications of the language it may be required to model more advanced calculations using trigonometric functions, integrals or similar. However, the language is not expressive enough to cover such calculations. It would be practical to facilitate more advanced calculations when needed.

A DSL should not contain a lot of unnecessary constructs that clutter the language. Adding constructs for several additional mathematical functions would thus not be a good idea, if these are to be used rarely. Instead, the metamodel of the language can be defined as a generic metamodel. With respect to the ongoing example, this would allow parameterisation of the math language with mathematical functions[2]. If no such functions are required, the language can be used with no actual type argument.

The math language, as defined using a generic metamodel, is given in Figure 5. An enclosing class SimpleMath is used to group the language constructs. It features a type parameter F bound by FunctionDef. The purpose of this type parameter is to allow parameterisation of the metamodel with externally defined functions.

```
package genericSimpleMath;

abstract class FunctionDef {
  operation calculate() : Real is abstract
}
class SimpleMath<F : FunctionDef> {
  class Program{
    attribute expressions : Expression[0..*]
    attribute plots : Plot[0..*]

    // Shows values and graphs on screen
    operation exec() is do ... end
  }
  abstract class Expression {
    operation eval() : Real is abstract
  }
  ...
  class Number inherits Expression {
    attribute value : Real
    operation eval() : Real is do ... end
  }
  class Function inherits Expression {
    attribute fDef : F[0..1]

    operation eval() : Real is do
      result := fDef.calculate()
    end
  }
}
```

**Figure 5: The math language defined using a generic metamodel**

An additional expression type, named Function, has been added to the math language. It has an (optional) attribute typed by the type variable F. Thus, a Function object may be composed by an object of the actual type argument. An interesting implication of using containment references is that a single argument may in fact represent several metaclasses - a metamodel fragment. Consider the metaclasses of Figure 6 which consitute a metamodel for modelling of definite integrals over polynomials.

[2]It can be argued that an actual type argument acts as a plug-in for the language.

```
package integral;

class Integral inherits FunctionDef {
  attribute polynomial : Polynomial[1..1]
  attribute varsOfInt : VarOfIntegration[1..*]
  operation calculate() : Real is do ... end
}
class Polynomial {
  attribute terms : Term[1..*]
}
class VarOfIntegration {
  attribute name : String
  attribute upperLimit : Real
  attribute lowerLimit : Real
}
class Term {
  attribute variables : Variable[0..*]
  attribute coefficient : Real
}
class Variable {
  reference varOfInt : VarOfIntegration[1..1]
  attribute degree: Integer
}
```

**Figure 6: A metamodel for modelling of definite integrals**

The metamodel for modelling of definite integrals can be used to parameterise the math language by importing the classes of the integral package into the genericSimpleMath package, and instantiating the SimpleMath class with Integral as actual type argument. As can be seen, objects of the Variable class are contained by objects of the Term class. Objects of Term are contained by an object of Polynomial. An object of this class is contained by an Integral object. Objects of VarOfIntegration are contained by the Integral object as well. Consequently, using Integral as an actual type argument for SimpleMath results in five new classes being (temporarily) added to the math language. These classes define a new function that can be utilised by the other constructs of the language. Integral supports modelling of higher-order integrals. A simple model illustrates how integrals can be used with the other expression types in the language. See Figure 7.

$$ScalarPlot[\int_0^5 \int_1^7 \int_3^5 3+xy^2+5x^7z^3 \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z)+10.5\times 4.1]$$

$$Result : 225\ 042\ 283.05$$

**Figure 7: An example model of an integral, addition and multiplication**

Generic types allow static type checking. This ensures that actual type arguments comply with the structure and semantics of the generic metamodel. In particular, language extensions can be defined without knowing the inner details of a language. The only criterion is that the type argument fulfills the constraints of the type parameter bound.

## 2.2 Configuring a language

We have seen how new constructs can be added to a language. An actual type argument can also be used to configure a language by changing its properties. With respect to the math language, it may be convenient to be able to specify how results are plotted to screen. This can be achieved by defining custom plot functions that are provided as actual arguments to the generic metamodel.

```
package genericSimpleMath;

abstract class ScalarPlotDef {
  operation plot() is abstract
}
abstract class GraphPlotDef {
  enumeration Colour{ red; blue; green; ... ; }
  operation plot() is abstract
  operation getPointColor() : Colour is abstract
  operation getGraphColor() : Colour is abstract
  ...
}
class SimpleMath<SP : ScalarPlotDef,
  GP : GraphPlotDef> {
  abstract class Plot {
    reference expressions : Expression[1..*]
    attribute caption : String
    operation plot() is abstract
  }
  class ScalarPlot inherits Plot {
    attribute sp : SP[0..1]

    operation plot() is do
      if( sp != void )
        // Custom plot
        sp.plot()
      else
        // Default plot
        ...
      end
    end
  }
  class GraphPlot inherits Plot {
    attribute gp : GP[0..1]
    ...
  }
  ...
}
```

**Figure 8: Using type parameters to configure a language**

A slightly different generic metamodel for the math language is given in Figure 8. This time, type parameters are used to allow customisation of the plot semantics. ScalarPlotDef and GraphPlotDef define operations that need to be implemented by an actual type argument. These operations are invoked from the respective plot constructs. The default plot styles are used if the metamodel is instantiated without type arguments. This is illustrated in the plot() operation of the ScalarPlot class.

Type parameters can be accessed by all the inner classes. This allows configuring cross-cutting concerns using type arguments. An example of such concern is calculation of statistics for languages whose domain deals with performance critical processing. The types of required statistical computations may change, something which can be addressed by using type parameters.

# 3. GENERIC METAMODELS AND SPECIALISATION

We have seen how class nesting and generic types can be used to define generic metamodels. Another interesting aspect of class nesting is the ability to subtype the enclosing class, with the intention of specialising the inner classes or adding new constructs to the language.

```
package genericSimpleMath;

class SimpleMathTrig<...>
  inherits SimpleMath<...> {
  abstract class Plot inherits SimpleMath.Plot {
    attribute colour : Colour[0..1]
  }
  class Colour {
    attribute red : Integer
    attribute green : Integer
    attribute blue : Integer
  }
  class Sin inherits Expression { ... }
  class Cos inherits Expression { ... }
  class Tan inherits Expression { ... }
}
```

**Figure 9: Defining a language variation using specialisation**

A variation of the math language with support for trigonometric functions is given in Figure 9. SimpleMathTrig subtypes SimpleMath. It specialises the Plot class and adds three new classes: Sin, Cos and Tan. The difference between using type arguments to achieve this and creating a language variation is that the latter approach should reflect core changes to the language. An advantage when subtyping the enclosing class is that specialised classes, like Plot in this case, can keep its original name. This ensures that users of the language will immediately be familiar with specialised constructs.

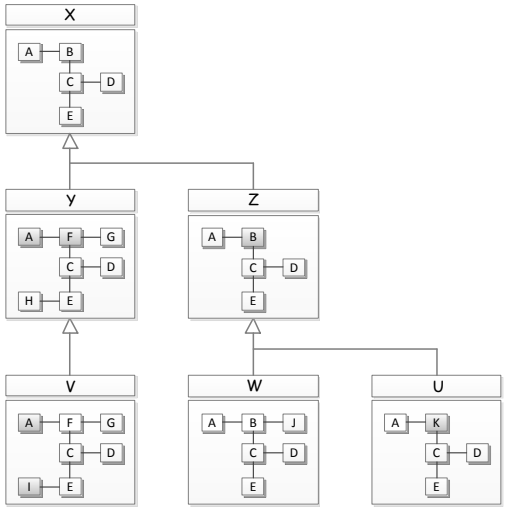


**Figure 10: Construction of a language hierarchy**

Subclassing also allows defining language hierarchies where different language variations can be used for slightly different domains. A language variation can easily be made by reusing existing language definitions. Figure 10 illustrates a

language hierarchy. The language denoted by X comprises five metaclasses. It is specialised in five variants: Y, Z, V, W and U. As an example, Y subtypes X and specialises the classes A and B of X. The specialised version of B is given the name F. Two additional classes, G and H, are added to the language. V subtypes Y and specialises the classes A and H. The specialised version of H is given the name I.

It is also possible to specialise generic metamodels. Specialisation is suitable for refining core aspects of a language, while type parameters allow further tuning of properties that differ depending on usage and context.

# 4. DEFINING VARIABILITY POINTS

Subtyping of nested classes is not always sufficient when specialising dynamic semantics. In fact, newly defined operations may be out of reach from the classes that should be able to invoke these. A solution is to define metaclasses as virtual, which ensures that the correct operations are bound at runtime.

```
package genericSimpleMath;
...
class SimpleMath<...> {
  class Program {
    attribute expressions : Expression[0..*]
    attribute plots : Plot[0..*]

    // Shows values and graphs on screen
    operation exec() is do
      ...
      var s : Screen init Screen.new
      plots.each{ p | s.prepare( p ).plot() }
    end
  }
  virtual class Screen {
    operation prepare( plot : Plot ) : Plot
      is do ... end
  }
}
```

**Figure 11: Defining a semantic variability point**

As an example, let us assume that the exec() operation of Program uses an instance of a class Screen to prepare Plot objects for visualisation on the screen. See Figure 11. The Screen class is declared as virtual and thus identifies a semantic variability point. It can be redefined as illustrated in Figure 12.

```
package genericSimpleMath;
...
class SimpleMathScreen<...>
  inherits SimpleMath<...> {

  class Screen {
    operation prepare( plot : Plot ) : Plot
      is do ... end
  }
}
```

**Figure 12: Redefinition of the Screen class**

The new definition of the Screen class can be utilised by instantiating SimpleMathScreen. This causes the newly defined Screen class to be instantiated in the exec() operation of Program. Thus, the new version of prepare(...) is used. If Screen was not declared as virtual, the Screen class of SimpleMath would instead be instantiated in exec().

# 5. PRESERVING CONFORMANCE

*Model conformance* is a term that indicates whether a model comply with a metamodel according to classifier relationships. Specifically, model elements are instances of classes defined in the model's metamodel. Changing a metamodel usually compromises conformity between the metamodel and the existing models. This is not desirable since it requires models to be transformed in order to conform with the new metamodel version. An interesting consequence of using generic metamodels is that language properties can be changed without breaking conformance. In particular, a simple actual type argument will not introduce new syntax if used within the nested classes. This is the case of Figure 8. Both type arguments are used exclusively within ScalarPlot and GraphPlot, by providing new dynamic semantics. A model conforming to the raw metamodel will conform to any parameterised metamodel regardless of type arguments. Consequently, evolution of languages can be dealt with in simple terms.

# 6. RELATED WORK

A definition of reusable *metamodel components* for OMG's Meta Object Facility (MOF) [3] is proposed in [11]. Composition is supported through the use of export and import interfaces. An export interface identifies a metamodel fragment, referred to as a submodel, that is visible outside of the component. An exported submodel from one component can be bound to another component using an import interface. The approach yields a mechanism for definition of parameterised language descriptions. Thus, it resembles the purpose of generic metamodels as discussed in this paper.

A workbench for language design and code generation, named *MetaEdit+*, is described in [8]. Two important features of this workbench are: support for easy code generation and integration of different languages. In addition, evolution is addressed by providing means of seamless metamodel and model updates. It is stated that MetaEdit+ increases the efficiency of language design and maintenance.

Application of aspect-orientation in metamodelling is discussed in [7], with the purpose of being able to reuse language definitions in a non-intrusive manner. An approach that addresses how this can be achieved using relationship aspects is discussed. With respect to this paper, creating language variations should not require changes to the original language definition. We have seen how language variations can be constructed non-intrusively using a combination of type parameters, specialisation and virtual classes.

# 7. CONCLUSIONS

This paper has illustrated how generic metamodels can be defined using class nesting and type parameterisation. A generic metamodel provides means for extending a language with new constructs, and configuring a language with new dynamic semantics. This is desirable in situations where a DSL is not accurate enough to model a specific concern of its problem domain. We have also discussed how specialisation of nested classes, and virtual classes may further increase

customisation of metamodels. As a consequence, the ability to reuse DSLs is increased.

# 8. REFERENCES


[1] G. Bray. Implementation implications of ada generics. *ACM Ada Letters*, 3(2), 1983.

[2] EMF. Eclipse modeling framework, 2011.

[3] O. M. Group. Meta object facility (mof) core specification.

[4] S. Kent. Model driven engineering. In *Proceedings of IFM'02*, 2002.

[5] O. L. Madsen and B. Møller-Pedersen. Virtual classes - a powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89*, 1989.

[6] F. F. Pierre-Alain Muller and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS 2005*, 2005.

[7] A. M. R. Quintero and J. T. Valderrama. Using aspect-orientation techniques to improve reuse of metamodels. *Electronic Notes in Theoretical Computer Science*, (163), 2007.

[8] J.-P. Tolvanen and S. Kelly. Metaedit+: Defining and using integrated domain-specific modeling languages. In *OOPSLA 2009*, 2009.

[9] A. E. Tony Clark and S. Kent. Aspect-oriented metamodelling. *The Computer Journal*, 46(5), 2003.

[10] P. S. Tony Clark and J. Willans. *Applied metamodelling (second edition)*. Ceteva, 2008.

[11] I. Weisemöller and A. Schürr. Formal definition of mof 2.0 metamodel components and composition. In *Proceedings of MODELS 2008*, 2008.