A Foundation for Multi-Level Modelling

Tony Clark[1], Cesar Gonzalez-Perez[2], Brian Henderson-Sellers[3]

[1] Middlesex University, London, UK. `t.n.clark@mdx.ac.uk`
[2] Institute of Heritage Sciences Santiago de Compostela, Spain
`cesar.gonzalez-perez@incipit.csic.es`
[3] University of Technology, Sydney, Australia
`brian.henderson-sellers@uts.edu.au`

Abstract. Multi-level modelling allows types and instances to be mixed in the same model, however there are several proposals for how meta-models can support this. This paper proposes a meta-circular basis for meta-modelling and shows how it supports two leading approaches to multi-level modelling.

1 Introduction

Contemporary and future engineering of information systems place an increasing emphasis on the use of models, either directly to aid design and implementation, in a more formal sense for code generation or as the backbone to model-driven engineering (MDE) [27]. Models must be described using a language that itself may be defined in many ways but typically using a meta-model *e.g.*, [26, 20]. That meta-model must itself be defined, by a meta-meta-model. Together with the instances conformant to the model, this leads to an identification of four abstraction levels of interest to the modeller and meta-modeller. Although in use for almost two decades, a four-layer architecture like that of the Object Management Group (OMG) raises some concerns both theoretically and pragmatically; a prime problem being the use of strict meta-modelling [5, 4] that constrains the instance-of relation to only be permitted between pairs of conterminous layers and never within a layer (see also [5]). This led several researchers (*e.g.*, [7, 6]) to seek a way of describing models and modelling languages without the use of this 'strict meta-modelling' hierarchy of the OMG.

A foundation for meta-modelling should be unifying and complete in the sense that it supports the development of both general-purpose and domain-specific languages and also integrates their representation so that tools can work across multiple languages. Leading approaches include: **strict meta-modelling**: The OMG strict meta-modelling architecture has been criticized, especially when applied to processes and methodologies (see summary in [18]) since the traditional strict meta-modelling approach is unable to support enactment *e.g.*, [2]; it defines attributes at level M2, thus giving them values at M1 by virtue of the prevailing type-instance semantics, when what is actually needed is values at M0. This enactment support is provided by the architecture used by ISO/IEC 24744 but at the expense of relying on power-type patterns, which do not accord with the philosophy of strict meta-modelling. **clabjects**: Potency is associated with the notion of deep instantiation, [8, 12, 13], and introduces the idea of an entity with both a class facet and an object fact, entity given the name *clabject* [5]. **OCA**: Two different kinds of meta-model structures have been identified: ontological meta-modelling in contrast to the linguistic meta-modelling utilized in a strict meta-modelling architecture. This was later called the Orthogonal Classification Architecture (OCA) [9]. In [22] we describe these ideas and relate them to some more recent concerns raised by the application of language use theory to this approach. More recently, Atkinson and colleagues have extended the OCA in their description of the Pan Level Model (PLM) and the Level-agnostic Modeling Language (LML) [7]. **powertypes**: The need to provide access to, and control over, the meta-types of elements in a model when designing languages led to proposals for *powertypes* [17, 23]. This is a methodological approach that uses standard classes both conventionally and as meta-classes by disciplined use of instance-of associations. The approach allows the modeller to control attribute definitions at M2 that affect the properties in model elements at M1.

Our claim is that none of the approaches above are complete as a basis for meta-modelling. In particular, such a basis must achieve the following features: **meta-circularity**: Self description is key to achieving virtually all of the desirable features for language engineering. Just as it is possible to embed a λ-calculus interpreter in itself and thereby characterize an infinite tower of operational languages, we seek to construct a self describing basis for an infinite tower of modelling languages. **uniformity**: Any basis for meta-modelling that is self-describing implies a precisely defined relation between representations for type and instance. A system that achieves the *conflation* of these representations, *i.e.*, uses the identity relationship, is *minimal* in the family of such relationships. Furthermore, a uniform representation is essential if we are not to encounter limitations on the type of languages that can be defined, for example where we need to mix instances and types. Therefore, we seek to provide a single representation for types and instances at any level. **extensibility**: We assume that any family of modelling languages will use type-based extension (sub-classes, inheritance, *etc.*), and that new languages are based on extending existing languages. Meta-circularity and extensibility implies that languages can be extended at both type and meta-type levels and therefore the question arises as to whether there is a limit to the levels over which extension can be applied. We seek a basis that places no restriction on the number of levels of both extensibility and instantiation. **views**: Languages should support multiple modes of interaction that are defined at the meta-level. Although we will use multiple language views, we will not consider this aspect further.

Our approach (subsuming those above) is to use simple *objects* together with two simple relations: **type** A relation that exists between every object and its class and can be applied an arbitrary number of times to define the meta-classifications of instance, class and meta-class; **extension** A relation that exists between classes that provides a minimal basis for incremental addition of features. The approach is based on existing proposals for meta-classes provided by languages such as Smalltalk [16] and ObjVLisp [10]. Although Smalltalk was the first language to introduce meta-classes (and thereby three-levels of meta-class, class and instance), each meta-class is restricted to having a single instance which severely limits its use as the basis for language engineering where meta-properties are reused across multiple languages.

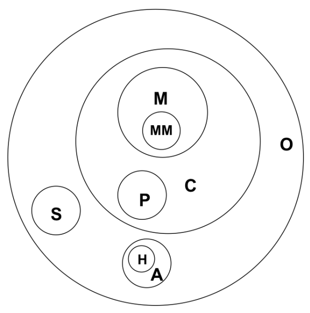


Fig. 1. Object Classifications As Sub-Sets of Object

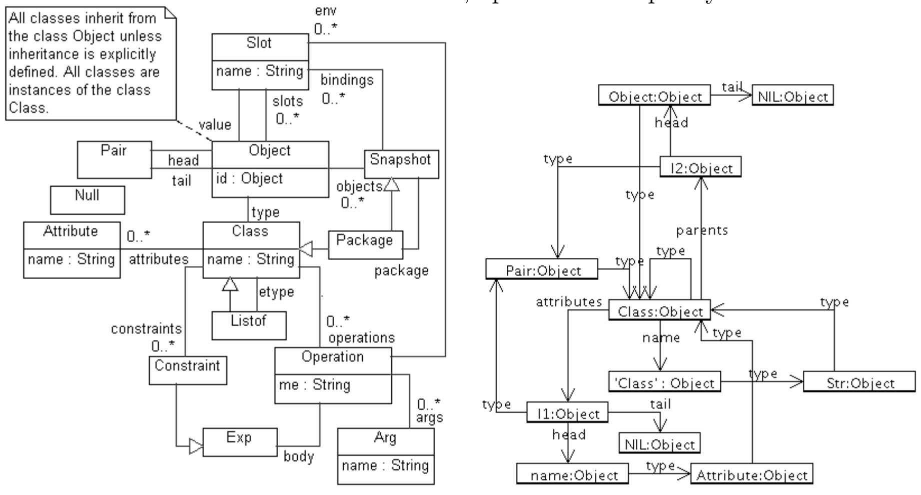
The approach to object classification and the instance-of relation is shown in Fig. 1 where circles represent sub-sets of the set **O** of objects. Consider the set **A** that denotes a set of objects representing animals. In order for an element of **A** to be well-formed, it must have an instance-of link to an object in the set **C** of all classes. Note that elements of **C** are objects (everything is an object), but they are objects that satisfy some criteria for class-hood. Since the element of **C** that represents the class Animal is itself an object, it must have an instance-of link to an object that represents its class. Such an object is a meta-class and is a member of the set of objects **M** (perhaps the class called Class). A meta-class is just an object that satisfies the constraint for membership of **M**. This means that it must have an instance-of link to a meta-meta-class in **MM**. It should be stressed at this point, that there is no limit to the instance-of regress. In addition to objects that satisfy Animal-hood. There are objects that are used to group objects: snapshots that are members of the set **S**. Snapshots contain objects that are all instances of related classes: packages that are members of the set **P**. Finally, classes can be related by extension so that there are two classes Animal and Herbivore in **C** that designate the rules for membership of the sets **A** and **H**. Of course, since every element of **M** is also in **C**, the extension relation can be defined between meta-classes that will designate different sub-sets of **C**.

Our basis for meta-modelling is defined as a self-describing object-oriented kernel. The Kernel is essentially a logic. However, unlike a traditional logic that consists of boolean valued formulas whose sub-expressions denote values drawn from a collection of predefined types, the Kernel can only denote objects. Some objects are designated *classes* because they conform to a particular object-interface that includes boolean valued expressions (or *constraints*) that characterize objects designated as well-formed instances of the class. Such a self-describing logic might lead to doubts related to Russell's Paradox, although the use of types and identities as described below, together with an implementation of the approach that supports a collection of real-world applications (including itself), gives us confidence that this is not a problem. Our claim is that this approach is novel and that it subsumes existing approaches to meta-modelling. Our contribution is the definition of a meta-circular foundation for model-based language engineering in the form of a kernel language that is validated in terms of an implementation as a toolkit that has been used for a variety of real-world applications. In addition we show that other approaches to multi-level modelling can be represented in the Kernel.

2 A Meta-Modelling Kernel

Our proposal is to set up a system whereby *everything is an object* [21] and where a simple set of rules governs the ability to construct configurations of objects that constitute self-describing languages. The system consists of an object-representation and then *sugarings* that are convenient language structures defined to *de-sugar* into the basic representation.

Figure 2(a) shows the proposed kernel language as a diagram. Fundamentally, everything is an object and a partial view of the Kernel as a collection of objects and slots is shown in figure 2(b). An object has a unique id, some slots, and a type. The type of an object is a class. Classes are organised into packages whose instances are snapshots that are assemblies of objects. Since classes are just objects that conform to some structural conditions, packages can be similarly viewed as snapshots with appropriate conditions. Collections of objects are organised as sequences in terms of pairs and `Null`. Since types are always implemented as classes, there is a special class called `Listof` whose instances are lists. There is no need to special types of atomic value such as integers and booleans because we can designate special objects via their identities as being members of these data types. Expressions are objects that can be asked to evaluate themselves in a supplied context. Constraints are special types of expressions that always return boolean values. Constraints are important because they are used in classes to *classify* objects that are considered to be *instances*. Classes have operations, that are objects used to handle messages sent to instances of the class. Note that there is no notion of side-effect, operations are purely functional.



(a) `Kernel` as a Class Diagram (b) `Kernel` as an Object Diagram

Fig. 2. Two Views of the Kernel

Fig. 3 shows the complete textual definition of the Kernel. It uses a number of external definitions and notational conventions that are outlined as follows: classes define a predicate ? that is used to determine instance-hood; operations use λ-notation where arguments are patterns; objects are `(C,i)[s ↦v]` where `C` is the class of the object, `i` is the id, `s` is a slot name and `v` is the corresponding value; `intern` maps a class and slots to an object; lists are `[v1,...,vn]` and can be appended using `+`; `::` is used to dereference names in a name-space; ⇑ is an inheritance relationship between classes.

Since the Kernel is essentially a logic we need something equivalent to OCL. We use the following shorthand where `l` is a list: `l.∀(p)` is `true` when the predicate `p` returns `true` for each element in the list `l`; `l.∃(p)` is `true` when the predicate `p` returns `true` for any element in the list `l`; `l.∋(x)` is true when the element `x` is contained in the list `l`; `l.⇐(p,a,y)` is the result of applying operation `a` to the first element `x` of `l` for which `p(x)` is `true` and `y` if no such element exists; `l.flatten()` expects `l` to be a list of lists and returns a list formed by appending all elements of `l` in order. # maps a list to its length. It is convenient to be able to construct and manipulate lists using *comprehension* expressions. For example, if `l` is the list `[2,3,4]` then `[x*2 | x ←l]` is the list `[4,6,8]`. Predicates may be used to filter lists as in `[x | x←l,?even(x)] = [2,4]`.

In order for this to be meta-circular, we require that and `Kernel.?(Kernel)` holds. This is difficult to establish without tooling since all the objects in the definition must be checked against their classes, and, since the classes themselves are part of the package, this requires the classes to be self-describing. The Kernel has been implemented as part of the XModeler toolkit and has been used to implement the rest of the tools including diagram tools, model browsers, model editors, model transformers and libraries. The XModeler Kernel contains many more classes than the language described in this article, but the essential features are the same. XModeler can be instructed to apply the Kernel-defined constraints to itself (over 100 classes) and to produce a report that shows that it is self-consistent.

3 Validation

Section 1 describes a list of features that we claim to be characteristic for any language that is used as a basis for meta-modelling. We have introduced such a language and used it to build a model of itself. This section analyses the Kernel language with respect to the characteristic features: **type**: In `Kernel` everything is an object and all objects have an intrinsic type property. **meta-circularity**: This property is essential for multi-level modelling and in order to be able develop tools (such as serializers) that are language-level agnostic [25]. The XModeler tool can be shown to establish that `Kernel.?(Kernel)`. **uniformity**: We have used a single representation (with a small number of externally defined conventions and rules) for all data in `Kernel`. **extensibility**: Extension is supported through class relationships that are then used by constraints in order to place conditions

```
class Object {
 id    : Object;
 type  : Class;
 slots : [Slot]
 constraints { type.?(self) }
 operations {
  dot(n) = slots.⇐(
   λ(n' ↦ _)n=n',λ(_ ↦ v) v,error)
  send(n,args) =
   type.ops().⇐(
    λ(n' ↦ (Operation)[args ↦ args'])
     n=n' and #args = #args',
    λ(_ ↦ f) f.invoke(self,args),
    error)
 }
}
class Slot {name:Str;value:Object}
class Operation {
 me   : Str;
 env  : [Slot];
 args : [Arg];
 body : Exp
 operations {
  invoke(target,values) =
   body.eval(env+['self' ↦ target] +
    [me ↦ self] + target.slots +
    target.type.ops()  +
    [a ↦ v | (a,v) ← args * values])
 }
}
class Listof extends Class {
 etype : Class;
 operations {
  ?(o) = list?(o) and
         o.∀(λ(x)etype.?(x))
 }
}
class Snapshot extends Object {
  package  : Package;
  objects  : [Object];
  bindings : [Slot]
  constraints {
   package.?(self);
   bindings.∀(λ(b)objects.∋(b.value))
  }
  operations {
   ::(k,d) = bindings.⇐(
    λ(s)s.name=k,λ(s)s.value,d)
  }
}
```

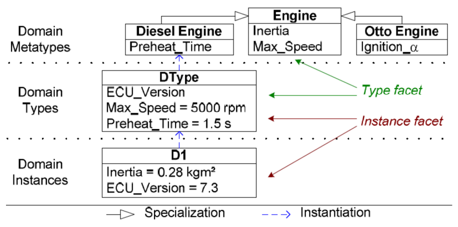
```
class Class {
 name        : Str;
 supers      : [Class];
 attributes  : [Attribute];
 operations  : [Binding];
 constraints : [Constraint]
 operations {
  supers() = [self] +
   [c | p ← supers;
        c ← p.supers()].remDups()
  ⤊(c) = supers().∋(c)
  atts() =
   [a | c ← supers(),a ← c.attributes]
  ops()  =
   [b | c ← supers(),b ← c.operations]
  cond() =
    [a | c ← supers(),a ← c.constraints]
  ::(n,d) =
    atts().⇐(λ(n' ↦ a)n'=n,λ(n ↦ a)a,
      ops().⇐(λ(n' ↦ o)n'=n,λ(n ↦ o)o,d))
  ?(o) = o.type.⤊(self) and
   atts().∀(λ(a)o.slots.∃(λ(s)
    s.name = a.name and
    a.type.?(s.value))) and
    cond().∀(λ(c) c.eval([self ↦ o] +
     [s.name ↦ s.value | s ← o.slots]))
 }
}
class Package extends Snapshot,Class {
 constraints {
  objects.∀(Class?);
   attributes.∀(λ(a) objects.∋(a.atype));
   parents.∀(λ(p) p.type.⤊(Package))
 }
 operations {
  ::(n,d) = obj().⇐(λ(o)o.n=n,λ(o)o,d)
  obj() = objects +
   [p.objects | p ← parents].flatten()
  ⤊(p) = objects.∀(λ(c)obj().∃(λ(c')c.⤊(c')))
  ?(o) = o.type.⤊(Snapshot) and
   o.package.⤊(self) and
   o.objects.∀(λ(o)
    objects.∃(λ(c) c.?(o))) and
    Class::?(intern(self,o.slots))
 }
}
class Pair {head:Object;tail:Object}
class Nulll {}
class Constraint extends Exp {}
class Arg { name:Str }
```

Fig. 3. Definition of `Kernel`

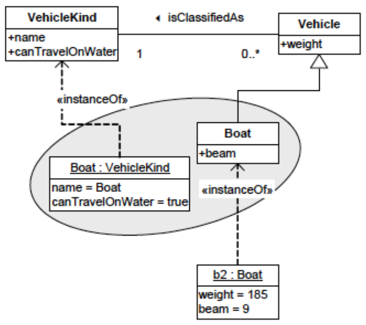
on objects that are instances of a sub-class. The definitions are `Class::?` and `Package::?` in Fig. 3.

Our claim is that the Kernel is a suitable basis for multi-level modelling. In order to validate this claim we present the definition of two different languages, each based on independent approaches, both defined in the Kernel. Models written in the languages are shown in Fig. 4.

The model in figure 4(a) shows the use of *type facets* that allow classes to have properties. These can be implemented by including a *potency* as part of an attribute definition. The potency is an integer value indicating the number of type-levels (3 are shown in the model) spanned by the relationship between



(a) A Use of Potency (in [3])



(b) A use of Power-Types (in [23])

Fig. 4. Two Approaches to Multi-Level Modelling

an attribute and its corresponding slots. The model defines a language (Domain Metatypes) of engines. The class `Engine` defines a type facet called `max_speed` that results in a slot at the domain type (model) level, and an instance facet called `inertia` that becomes a slot at a remove of two type-levels.

The model in figure 4(b) shows the power-type pattern where a class (in this case `Vehicle`) is classified by another class (`VehicleKind`). Instances of `VehicleKind` are used to partition subclasses of `Vehicle` as shown in the ellipse, forming a *clabject*. The result is that an object is contributing to the type-level information in a class that will eventually affect instances of the class.

Each language definition takes the form of a package that is both an instance and an extension of `Kernel`. By the definition of `Package::?`, an instance of a package `P` should be a snapshot whose contents are all instances of classes in `P`. By the definition of `Package::extends?`, a package `P` extends a package `Q` when every class in `P` extends some class in (or inherited by) `Q`. Therefore, by extending and instantiating `Kernel` a package is a well-formed language definition in its own right, that can, by the definition of extension, modify the basic definition of `Class::?`. Such a modification might place extra conditions on instance-hood, or even relax existing conditions.

Fig. 5 contains the definition for the language and models shown in figure 4(a). The class `CAtt` extends `Attribute` with an attribute for potency-level. The class `CClass` modifies `atts` so that it gathers together all attributes that apply to this level. This is achieved using a counter that is incremented when the type-level is traversed. A concrete-syntax for potency-level in attributes is used in the definition of the package `DomainMetaTypes`, and slots are permitted in class definitions due to potency-levels becoming `0` in `DomainTypes`. The snapshot `DomainInstances` contains a single object whose slots correspond to attributes from different type-levels as defined by their respective potency-levels.

Fig. 6 contains the definition for the language and models shown in figure 4(b). The meta-class `PowClass` defines an attribute `classifier` and the constraint on `PartClass` requires that all its descriptor objects are instances of the classifier inherited by a parent power-class. The package `Vehicles` contains a single power-class `Vehicle` that is classified by `VehicleKind` and a partitioned-class `Boat`

```
package CKernel:Kernel extends Kernel {
 class CAtt extends Attribute {
  level:Integer;
 }
 class CClass extends Class {
  operations {
   atts() = catts(1,self)
   catts(n,c=(_,c)[]) = []
   catts(n,c) =
     [a | a ← c.atts(),
          ?a.type=CAtt,a.level=n] +
     catts(n+1,c.type)
  }
  constraints {atts.∀(λ(a)a.type=CAtt)}
 }
}
snapshot DomainInstances:DomainTypes {
 (DType)[inertia↦0.28;ECU_version↦7.3]
}
```

```
package DomainMetaTypes:CKernel  {
 class Engine:CClass extends CClass {
  inertia[2]:Float;
  max_speed[1]:Integer
 }
 class DieselEngine:CClass extends Engine {
  preheat_time[1]:Float
 }
 class OttoEngine:CClass extends Engine {
  ignition_alpha[1]:Float
 }
}
package DomainTypes:DomainMetaTypes {
 class DType:DieselEngine {
  ECU_version[1]:Float;
  max_speed=5000;
  preheat_time=1.5
 }
}
```

Fig. 5. Definition and use of `CKernel`

that includes an instance of `VehicleKind` as its descriptor. The snapshot `ABoat` is governed by the classes defined in the package `Vehicles` which in turn are governed by the language `PKernel` therefore, `ABoat` is constrained by the clabject `Boat` and `Boat.descr`.

```
package PKernel:Kernel extends Kernel {
  class PowClass extends Class {
   classifier:Class
  }
  class PartClass extends Class {
   descr:[Object]
   constraints {
    supers().∀(λ(c) PowClass?(c));
    descr.∀(λ(o)
     supers().∃(λ(c)
      c.classifier.?(o)))
   }
  }
}
snapshot ABoat:Vehicles {
 (Boat)[beam ↦ 9;weight ↦ 185]
}
```

```
package Vehicles:PKernel {
 class Vehicle:PowClass {
  classifier=VehicleKind
  weight:Int
 }
 class VehicleKind {
  name:Str;
  canTravelOnWater:Bool
 }
 class Boat:PartClass extends Vehicle {
  descr=[(VehicleKind)[
           name↦'Boat';
           canTravelOnWater ↦ true]]
  beam:Int
 }
}
```

Fig. 6. Definition and use of `PKernel`

The examples described above contribute evidence that `Kernel` can define different languages and is not restricted to a fixed number of type-levels, and that objects, classes and meta-classes can be mixed. This is possible because of the uniformity of representation, the unrestricted access to type-level information and meta-circularity. Although outside the scope of this paper, the formulation of `Kernel` makes it possible to write level-agnostic tools, such as those for model-management, that can be used on any type-level.

4 Conclusion

Our aim is to produce a meta-circular level-agnostic basis for model-based language engineering. We have reviewed the current proposals for such a basis and argued that they are not optimal by providing a new language definition that is self-describing and can be used to embed the competing approaches. The Kernel language is simple and can be implemented as demonstrated by the XMF and XModeler toolkit [11] that is capable of both describing and reasoning about itself. The toolkit was reported as a leading technology for Software Engineering [19] and has been used for a variety of applications including modelling languages for aerospace applications, telecoms applications [1], and is currently being used to implement aspects of the MEMO enterprise modelling language [24, 14].

In [15], the authors show how the OMG levels M0-M3 can be represented on a single object-diagram. This allows OCL constraints to range over all levels and thereby support clabjects and potency. This is consistent with our approach, although OCL is just one of the languages that could be used with our approach (as a view of models and constraints) and the authors of [15] do not claim to be a foundation for model-based language engineering.

Our intention is that the Kernel language defined in this article provides a basis for ourselves and others to experiment with language definitions. Because all such kernel-defined languages are based on a single object representation, it is feasible to build a collection of tools that work against well defined sub-sets of objects (as shown in figure 1) and thereby incrementally develop a shared library.

References

1. Achilleas Achilleos, Nektarios Georgalas, and Kun Yang. An open source domain-specific tools framework to support model driven development of oss. In *Model Driven Architecture-Foundations and Applications*, pages 1–16. Springer, 2007.
2. Anat Aharoni and Iris Reinhartz-Berger. A domain engineering approach for situational method engineering. In *Conceptual Modeling-ER 2008*, pages 455–468. Springer, 2008.
3. Thomas Aschauer, Gerd Dauenhauer, and Wolfgang Pree. Representation and traversal of large clabject models. In *Model Driven Engineering Languages and Systems*, pages 17–31. Springer, 2009.
4. Colin Atkinson. Meta-modelling for distributed object environments. In *Enterprise Distributed Object Computing Workshop [1997]. EDOC'97. Proceedings. First International*, pages 90–101. IEEE, 1997.
5. Colin Atkinson. Supporting and applying the UML conceptual framework. In *The Unified Modeling Language. UML 98: Beyond the Notation*, pages 21–36. Springer, 1999.
6. Colin Atkinson, Bastian Kennel, and Björn Goß. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In *Procs. 7th Int. Workshop on Semantic Web Enabled Software Engineering, Bonn (October 24, 2011)*.
7. Colin Atkinson, Bastian Kennel, and Björn Goß. The level-agnostic modeling language. In *Software Language Engineering*, pages 266–275. Springer, 2011.
8. Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 19–33. Springer, 2001.

9. Colin Atkinson and Thomas Kühne. Concepts for comparing modeling tool architectures. In *Model Driven Engineering Languages and Systems*, pages 398–413. Springer, 2005.
10. Jean-Pierre Briot and Pierre Cointe. The objvlisp model: Definition of a uniform, reflexive and extensible object oriented language. In *ECAI*, pages 225–232, 1986.
11. Tony Clark and James Willans. Software language engineering with xmf and xmodeler. *Formal and Practical Aspects of Domain Specific Languages: Recent Developments. IGI Global, USA*, 2012.
12. Juan De Lara and Esther Guerra. Deep meta-modelling with metadepth. In *Objects, Models, Components, Patterns*, pages 1–20. Springer, 2010.
13. Juan de Lara, Esther Guerra, Ruth Cobos, and Jaime Moreno-Llorena. Extending deep meta-modelling for practical model-driven engineering. *The Computer Journal*, page bxs144, 2012.
14. Ulrich Frank. Multi-perspective enterprise modeling: foundational concepts, prospects and future research challenges. *Software and System Modeling*, 13(3):941–962, 2014.
15. Martin Gogolla, Jean-Marie Favre, and Fabian Büttner. On squeezing m0, m1, m2, and m3 into a single object diagram. *Proceedings Tool-Support for OCL and Related Formalisms-Needs and Trends*, 2005.
16. Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., 1983.
17. Cesar Gonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software & Systems Modeling*, 5(1):72–90, 2006.
18. Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for software engineering.* Wiley Publishing, 2008.
19. Simon Helsen, Arthur Ryman, and Diomidis Spinellis. Where's my jetpack? *Software, IEEE*, 25(5):18–21, 2008.
20. Brian Henderson-Sellers. *On the mathematics of modelling, metamodelling, ontologies and modelling languages.* Springer, 2012.
21. Brian Henderson-Sellers, Tony Clark, and Cesar Gonzalez-Perez. On the search for a level-agnostic modelling language. In Camille Salinesi, Moira C. Norrie, and Oscar Pastor, editors, *CAiSE*, volume 7908 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2013.
22. Brian Henderson-Sellers, Owen Eriksson, Cesar Gonzalez-Perez, and Pär J Ågerfalk. Ptolemaic metamodelling? the need for a paradigm shift. *Cueva Lovelle JM, Pelayo García-Bustelo C, Sanjuán Martínez O (eds) Progressions and innovations in model-driven software engineering. IGI Global, Hershey, PA*, pages 90–146, 2013.
23. Brian Henderson-Sellers and Cesar Gonzalez-Perez. Connecting powertypes and stereotypes. *Journal of Object Technology*, 4(7):83–96, 2005.
24. Thomas Johanndeiter, Anat Goldstein, and Ulrich Frank. Towards business process models at runtime. In Nelly Bencomo, Robert B. France, Sebastian Götz, and Bernhard Rumpe, editors, *MoDELS@Run.time*, volume 1079 of *CEUR Workshop Proceedings*, pages 13–25. CEUR-WS.org, 2013.
25. Fabio Kon, Fabio Costa, Gordon Blair, and Roy H Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.
26. Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.
27. Jesús Sánchez-Cuadrado, Juan De Lara, and Esther Guerra. Bottom-up metamodelling: An interactive approach. In *Model Driven Engineering Languages and Systems*, pages 3–19. Springer, 2012.