A Meta-Circular Basis for Model-Based Language Engineering

Tony Clark
Aston University, Birmingham, UK

ABSTRACT Meta-modelling is a technique that facilitates the construction of new languages to be used in system development. Although meta-modelling is supported by a number of tools and technologies, notably the Meta Object Facility from the OMG, there is no widely accepted precise basis for meta-modelling that can be used to develop and study language-based approaches to system development. Recent advances in meta-modelling have proposed several approaches to mixing types and instances, and allowing constraints to hold over multiple levels. This article proposes a collection of key characteristic features that are used to define a foundational self-contained unifying meta-language that is evaluated through several examples.

KEYWORDS Meta Models, Reflection, Language Engineering.

1. Introduction

Meta-languages have a long and illustrious history in Computer Science. They have been used for programming language definition (Quinlan et al. 2019), controlling proof systems (Bry 2018), architecture description languages (Oquendo 2016), the basis of tooling (Kon et al. 2002) and controlling and generating programs (Culpepper et al. 2019). Recently, motivated by interest in Domain Specific Modelling Languages (Brand et al. 2018) there is interest in meta-modelling languages for language engineering (de Lara et al. 2015) as represented by MOF and OMME (Volz & Jablonski 2010). These languages provide concepts that are used to express modelling languages as models, and typically provide mechanisms for expressing types, relationships, properties, behaviour and constraints.

Given that a modelling language can be modelled, we might expect there to be a self describing meta-language from which all other languages can be generated. A single language will support reuse of language fragments and entire languages that can be quality assured. A single meta-language implies a single API that facilitates tool reuse. A self-describing extensible meta-language lifts the benefits of reuse and quality assurance to the level of modelling tools and their semantics: a simple example is a property-based editor that can be used to interrogate any part of a multi-level model. A universal meta-language that includes semantics and operational features provides a basis for both model and tool interoperability (Van Tendeloo & Vangheluwe 2017a). Despite attempts to achieve this, for example MOF, EMOF and EMF (Fouquet et al. 2012), and the Modelverse (Van Tendeloo & Vangheluwe 2017b) a common universal implementation-independent self-describing language has yet to emerge.

Language definition consists of *syntax* and *semantics*. The abstract syntax of a language is often expressed using a class-based model. The semantics is given by attaching constraints to the model. A typical example of an object-oriented constraint modelling language is the Object Constraint Language (OCL) (Liu & Özsu 2018; Cabot & Gogolla 2012; Richters & Gogolla 1998; Gogolla & Vallecillo 2019; Doan & Gogolla 2018) that allows first-order logic formulas to be applied to instances of a class, thereby requiring instances of the class to satisfy precisely expressed conditions. As described in (Clark & Frank 2018; Clark et al. 2015a), when OCL constraints are associated with meta-classes, the conditions apply to classes, this provides a way of expressing structural and behavioural semantics for a language. Other systems of describing model-based constraints have been proposed, for example (Pschorn et al. 2019; Gil et al. 1999), however the semantics of the constraints are always given by an external system and therefore not part of the model.

JOT reference format:
Tony Clark. *A Meta-Circular Basis for Model-Based Language Engineering*. Journal of Object Technology. Vol. 19, No. 3, 2020. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2020.19.3.a11

One of the difficulties in achieving a common basis for model-based language engineering is the problem of providing a meta-circular description of constraints, which are necessary for semantic definitions. Typically, meta-languages avoid the problem, by representing constraints as external strings, for example EXTREMO (Segura & de Lara 2019). As noted in (Urbán et al. 2018):

> For any practical modeling technique, having a consistent and powerful operation language is more than a desired feature. Such a feature enables models to be truly self-contained by incorporating the semantics and the dynamic nature of the models as their integral part, instead of relying on an externally provided substitute.

The language *DMLAScript* motivates the need for a self-describing language of operations, however the semantics of the language are not described by the language itself. Object-oriented language researchers have achieved meta-circularity, for example (Cointe 1987, 1996; Herzeel et al. 2008), which has been used as the basis of the XModeler meta-modelling tool (Clark et al. 2015a,b). However, in both these cases, the key features of self-description rely on implementation issues which prevents the languages being self contained.

Extensibility in programming languages is facilitated through a technique called *bootstrapping* (Polito et al. 2015) where a collection of kernel objects are created that support compilation and interpretation of the language. Extension points are provided by the bootstrap that can be used to modify the language syntax and/or semantics. The authors (Polito et al. 2015) argue that bootstrapping is:

> A beneficial engineering practice because it raises the level of abstraction of a program making it easier to understand, optimise, evolve, etc.

In addition, many programming languages offer *reflective* features (Herzeel et al. 2008) that, with or without bootstrapping, provide access to (*introspection*) and the ability to modify (*intercession*) a running program.

This article provides a step-by-step bootstrap of a meta-circular language for language engineering. The basic representation for objects is similar to that of an object calculus (Abadi & Cardelli 2012) although we do not consider type checking. The aim of the language is to support the definition of many different languages, all of which have the same (meta)*-representation and whose semantics are defined using the same core concepts that are based on a kernel *meta-object-protocol* (MOP). MOPs have been used elsewhere to support programming language engineering (Kiczales et al. 1991) and meta-modelling (Cuadrado & de Lara 2018); the aim of this article is to combine the benefits of both by applying programming language engineering to modelling languages. The language is evaluated by showing how it can build itself, can support multiple concrete syntax definitions, and can define an extended meta-language based on *potency*.

Section 2 provides a brief overview of meta-modelling, its uses, and approaches to the issues related to mixing types and instances. Section 3 describes a collection of requirements for a universal language. Section 4 provides a step-by-step construction of a language that satisfies the requirements. Although the language is textual, many modelling languages provide a graphical syntax; section 5 shows how the language can be used to build graphical language syntax. Section 6 provides examples of the language to build modelling languages.

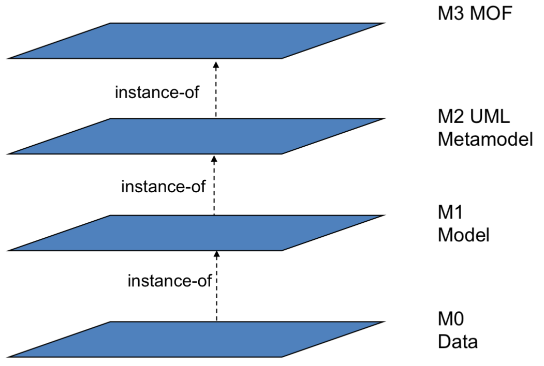

Figure 1 OMG four-level meta-modelling hierarchy (Henderson-Sellers & Unhelkar 2000)

2. Related Work

Engineering of information systems places 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) (e.g. (Sánchez-Cuadrado et al. 2012)) or model-based engineering (Selic 2012).

Models must be described in some way; typically using a notation (*a.k.a.* concrete syntax) associated with a modelling language. The language itself may be defined in many ways but typically a meta-model is used for this purpose *e.g.*, (Kühne 2006; Henderson-Sellers 2011, 2012). 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 (Fig. 1). Although in use for over two decades, a four-layer architecture like that of the Object Management Group (OMG) raises some concerns for example *strict meta-modelling* (Atkinson 1999, 1997) that constrains the instance-of relation to only be permitted between pairs of conterminous layers and never within a layer. This led to a proposal (Atkinson & Kühne 2003) to differentiate between so-called ontological and linguistic meta-modelling as a means of addressing strictness (Atkinson et al. 2011a,b).

Meta-modeling is used in a number of ways to support language-based SE. These include:

language definition: A meta-model is a language that is used to create models using *general-purpose* languages (GPL), such as UML, where there is no pre-defined application domain, and *domain-specific* languages (DSL), where the languages support a particular application domain such as telecoms or finance (Reinhartz-Berger et al. 2013). There may be mixtures of GPLs and DSLs and there are interesting variations whereby a DSL can be categorised as *external* where it is effectively stand-alone, or *embedded* when it is used as part of a wider language. Meta-modelling languages support multiple types of embedding, such as *homogeneous* whereby the host language provides extension facilities for defining the embedded language, and *heterogeneous* when the language extension is resolved externally, perhaps through some form of library or rewrite system (Clark & Tratt 2010).

language integration: Meta-modelling is often used to achieve the integration of a number of disparate languages by supplying a single data definition that allows elements from one language to be referenced meaningfully from a different language. The multiple languages of UML (UML Profiles) are an example of this. In practical terms, a single meta-modelling language also provides a uniform export format that allows multiple independent tools to work together to form tool-chains.

methodology integration: A meta-model is a useful conceptual framework to provide the necessary links between the language (and even language elements) and the process-, product- and people-related aspects of the methodology within which it will be used. For example, the ISO/IEC 24744 meta-model (Henderson-Sellers & Gonzalez-Perez 2008) provides the necessary constructs to state which work products are expressed in what languages and who and when are expected to create and use them during enactment.

uniformity: A uniform representation is key to providing a rich collection of tools that support the system development process. Given the rise of domain-specific techniques, it is important that freshly minted languages integrate with existing languages as described above. It is also important that modelling tools do not become obsolete as new languages are developed. For example, model editors or code generators that rely on general structures of languages should run against a wide range of current and future language definitions. Therefore, a key feature of meta-modelling technologies is that tools can be written against a uniform representation.

There are a number of current approaches to meta-modelling:

strict meta-modelling: As noted above, instances, models and their meta-models exist in different domains that cannot mix freely.

clabjects: Soon after the publication of the first version of the UML in 1997, concerns were raised about the notation used that was not consistent across models and meta-models. This led to the idea of *potency* associated with the notion of deep instantiation (Atkinson & Kühne 2001; De Lara & Guerra 2010; de Lara et al. 2012), also introducing the idea of an entity with both a class facet and an object facet, given the name *clabject* (Atkinson 1999).

OCA: Researchers have proposed two different kinds of meta-model structures: 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) (Atkinson & Kühne 2005) extended to become Pan Level Model (PLM) and the Level-agnostic Modeling Language (LML) (Atkinson et al. 2011a).

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* (Gonzalez-Perez & Henderson-Sellers 2006; Henderson-Sellers & Gonzalez-Perez 2005). 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.

As described in (Jácome-Guerrero & de Lara 2020) the need for creating models containing elements from multiple type levels is increasingly recognised. This has led to a number of tools including DeepTelos (Jeusfeld 2019), FMMLx (Clark & Frank 2020), Melanee (Atkinson & Gerbig 2016) and MetaDepth (De Lara & Guerra 2010). However, in many cases these approaches are based on annotations added to a model to indicate the type level at which the information is to be defined and the core meta-language is not exposed to the modeller, preventing them from defining and using a family of languages. Where the meta-language is exposed, it does not address *execution* in any sense. Our aim is to provide a unifying meta-language that can be used to explain and compare the key semantic features of tools and approaches described above.

3. Domain Analysis

Reflection has been studied by several programming language and technology platform researchers leading to a collection of language requirements (Ancona & Cazzola 2004). Although the contribution of this paper relates to the specification of model-based languages for describing systems, the requirements provide a basis as a domain analysis for the language presented in the rest of the article. This section presents the requirements that will be reviewed when the language is evaluated in section 6. The reflective language requirements are as follows:

introspection: The ability of a system to reason about its own state. This implies that there is a representation for those aspects of the system that can be introspected, and full meta-circularity implies that the *whole* system can be reasoned about to any level of detail. In the case of the language presented in this article, all data is represented by objects whose properties are available for inspection.

intercession: The ability of a system to change its own structure or behaviour. The language presented in this article is used to specify modelling languages and therefore intercession occurs in terms of the ability to change expression evaluation through an object-oriented MOP (Chari et al. 2018).

referents: Used to describe the features of a system that are reflected. A meta-circular language places a requirement that the referents used in introspection and intercession are represented in the same way at each level of the reflective tower. The language presented in this article uses objects to represent all features including basic atomic data, collections and operations. Thus when any of these items are reflected, the resulting referents are also objects.

reflective tower: The levels that are traversed when introspective steps are performed. In general this raises a question regarding the top-most level in the tower, however a meta-circular system has an additional requirement that the top-most level is self-describing supporting an infinite number of levels through a loop.

reification: The act of reflecting on a particular language element or state of computation. For programming languages with computational state, this raises an interesting challenge because referents must be found for otherwise hidden parts of the execution machinery. For a specification language this is less challenging because system descriptions are state-less.

Modelling languages are organised around concepts as exemplified by MOF (Favre & Duarte 2016) including classes, associations, operations, constraints, attributes, and packages. Our proposal is that, in order to achieve the meta-circular features defined above, all of these concepts can be implemented in terms of *objects* that support the following concepts:

type: Each object is an instance of exactly one type. It must be possible to reflect and produce a referent to the type of an object. One approach to achieve an infinite reflective tower is for the type referent to also be an object.

slot: Each object contains a collection of slots. It must be possible to reflect on an object's slots, each of which will also be objects.

classification: A type will classify a collection of objects. This is the basis for model-based specification where constraints are used to define the semantics of the model instances. Recently there has been interest in *deep* classification of models (Atkinson & Kühne 2015) that traverses multiple type levels.

message passing: The behaviour of objects are given by operations that are invoked as a result of message passing. Reflection requires that referents are provided to the operations whose structure can be accessed through object-based operations including an object-based representation for an expression language.

collections Objects are grouped together using collections. Collections are used as the glue that is used to build models (packages) and model-instances (snapshots). Introspection and a reflective tower is achieved by making collections objects.

views: Modelling languages make a distinction between abstract and concrete syntaxes. A concrete syntax is sometimes referred to as a view on the abstract syntax and it must be possible to define views over models.

integration: Meta-modelling is often used to achieve the integration of a number of disparate languages. This has recently been referred to as *globalising* modelling languages (Combemale et al. 2014). A single meta-modelling language also provides a uniform serialisation format that allows multiple independent tools to work together to form tool-chains.

Together these form the requirements for the meta-circular language defined in the rest of this article.

4. A Meta-Modelling Kernel

Our proposal is to bootstrap a reflective, extensible system whereby *everything is an object*. This is an idea first proposed by the creators of Smalltalk and analysed in the context of system modelling in (Bézivin 2005). We aim for a simple set of rules supporting configurations of objects that constitute self-describing languages. The system consists of a simple object-representation and *sugarings* that are convenient language structures that *de-sugar* into the basic representation. This section defines the kernel of the system and shows that it is self-consistent.

4.1. Object Representation

Everything is an object. An object is denoted `(T)[n ↦ o;...]` where `T` is a *type*, `n` is a name, and `o` is an object. A simple example of an object is a two-dimensional point: `(Point)[x ↦ 10;y ↦ 20]`. Note that the type `Point` and the values `10` and `20` are all objects. The identifiers `x` and `y` name *slots* and slot *reference* uses the infix _._ operator: therefore, given a suitable definition for object equality =, if `p` is the point object defined above then `p.x=10` and `p.type = Point`.

Each object has an identity `i` that can optionally be referenced, as in the object `(T,i)[...]`. In addition, an object's slots may be omitted when we write it out, therefore, `(Point,p1)[x ↦ 10]` and `(Point,p1)[x ↦ 10;y ↦ 20]` refer to the same object.

Note that the value `10` is itself an object of type `Int`. Each integer is a distinguished object. Again, with reference to a suitable equality operator it must be the case that if `(Int,i)[...] = (Int,j)[...]` then the two identities must be the same. It is convenient if we conflate each integer with its identity. The same applies to all standard atomic datatypes such as booleans and characters.

Collections of objects are organized into list-objects that are either `(Nil)[]` or a pair `(Pair)[head↦h;tail↦t]` for some objects `h` and `t`. Lists provide an obvious opportunity for sugar, where ↔ means *syntactically equivalent to*:

```
[]          ↔  (Nil)[]
[o]         ↔  (Pair)[head ↦ o; tail ↦ []]
[h | t]     ↔  (Pair)[head ↦ h; tail ↦ t]
[h,o,...]   ↔  (Pair)[head ↦ h; tail ↦ [o,...]]
```

Lists can be concatenated using `_+_` and support a collection of operations that are described in section 4.5. 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]`.

A slot is an object `(Slot)[name ↦ n; value ↦ v]` which has the sugared form `n ↦ v`. The slot `slots` is used to extract the slots of an object as instances of the type `Slot`. For example:

```
(Point)[x ↦ 1; y ↦ 2].slots = ['x' ↦ 1, 'y' ↦ 2]
```

4.2. Object Views

Beyond trivial models, rendering everything as an object will obviously become unworkable due to the number of objects and their relationships. Therefore, we seek a means to address the usability of the approach. Our proposal is to treat objects as *abstract data* and to define all *concrete* representations of objects as views that must be precisely defined with respect to the abstract data type for objects.

In this way, we can have multiple views over the same objects. General purpose views, such as UML object diagrams, can be defined that apply to any object configuration. Other views may be partial, for example UML class diagrams, but still work with a wide range of object configurations providing they conform to some general pattern. Domain specific views can be defined for configurations of objects that exhibit very specific properties. Note that the relation between object configuration and concrete view will be one-to-many. Views may be textual, graphical or a mixture of the two.

Concrete views of the abstract object representation are essentially relations between two models. The relation itself is a model and can be defined independently of the two models it relates. Section 5 shows how diagrams can be defined using this approach. Not all models have a convenient graphical representation, so we shall start by outlining how textual languages can be constructed for configurations of objects. The usual way of defining such relationships is to use grammars, however a simpler and more abstract approach can be used by inverting the relationship: specify the abstract representation by mapping to a string using an operation called `display`.

The following is a simple example for two-dimensional points where pattern matching is used to extract the values for slots named `x` and `y`, and `+` is used to concatenate strings:

```
display((Point)[x ↦ v; y ↦ w]) =
  '(' + v + ',' + w + ')'
```

Now it is possible to use the notation `(1,2)` instead of `(Point)[x↦1; y↦2]`. As described below, operations are defined by classes and invoked by message passing. We assume that there is a suitable definition of `display` for all classes, for example:

```
(Point)[x ↦ 1; y ↦ 2].display() = '(1,2)'
```

In addition, `l.displays()` omits the leading `[` and trailing `]` of a list by producing a string containing the display of each element in the list `l` separated by commas.

4.3. Operations: Invocation

Consistent with all object-oriented approaches, the class of an object defines named operations that may be invoked using the n-place `_._(_,...,_)` operator. A small number of such operations are built-in, but most are defined using the features of the kernel.

All the basic data-types, such as integer and boolean, have the expected operations, for example `1.+(2)=3`, which can be conveniently written `1+2=3`. Consider `+` as a built-in operator; as such it is a supplied function consisting of a set of triples (o,a,r) where o is the target, a is a list of arguments and r is the result. The kernel will detect the use of such built-ins and use the designated set of triples appropriately.

There is an important built-in operation called `send` that can be used to send messages to a target. It is important for meta-circularity that any built-in can also be sent using `send`, therefore: `i.+(j)=i.send('+',[j])`. Furthermore, it must be the case that `o.send(m,a)=o.send('send',[m,a])`.

Note that since the language supports execution, there is the possibility for errors where the wrong type of argument is presented to operators. Two, not necessarily distinct, approaches present themselves: (1) perform static type analysis; (2) perform dynamic type analysis. Static analysis is challenging in such a meta-circular language system and is left as a separate research activity. Dynamic analysis is possible, but would require type checks to be inserted when identifiers are introduced (such as function arguments) and require an error handling protocol to be defined. The latter is assumed to be possible, but omitted since it would introduce additional orthogonal features to the language.

4.4. Expressions and Constraints

An expression is an object that supports an operation `eval` with a single argument, a list of slots called an *environment*, mapping free variables to their values. Constraints are boolean expressions that apply to configurations of objects. Classes use constraints to define *instance-hood* for candidate objects, *i.e.*, whether a given object can be considered to be an instance of a particular class.

Constraints are objects that conform to a particular interface that is defined by an abstract class `Constraint`. For example:

```
(Greater,c)[
  left ↦ (Add)[
    left ↦ (Int,10)[];
    right ↦ (Int,20)[]];
  right ↦ (Int,2)[]
]
```

is a constraint. For convenience, constraint views are defined using the usual expression-based representation: `c = 10 + 20 > 2`. An expression may include variables such as `10 + 20 > x` where a variable is an object `(Var)[name ↦ 'x']`.

Constraints are attached to a class and have a designated variable that will be supplied with the value of a candidate object. If the expression evaluates to `true` then the candidate is classified by the class. This, in conjunction with meta-circularity, allows the kernel to impose structural conditions on objects, even those that are part of its own definition. A simple example is the classification constraint of `Point` that requires the candidate to have two slots with names `x` and `y` whose values are classified in turn by the class `Int`.

4.5. Operations: Definition

Operations are parameterised expressions. They serve to abstract over patterns of expressions and form the basis of queries over objects. Note that there is no notion of side-effect built into the kernel, and therefore no notion of state-based execution.

Consider defining a simple operation to determine whether an element exists in a list. The operation has the form:

```
(Operation,o)[
  env   ↦ [member ↦ o];
  args ↦ [
    (Arg,e)[name ↦ 'element'],
    (Arg,l)[name ↦ 'list']
  ];
  body ↦ (If)[
    test ↦ (Eql)[
      left  ↦ e;
      right ↦ (Dot)[
        object ↦ l;
        name   ↦ 'head'
      ]
    ];
    conseq ↦ true;
    alt    ↦ (Apply)[
      op   ↦ (Var)[name ↦ 'member'];
      args ↦ [
        e,
        (Dot)[
          object ↦ l;
          name   ↦ 'tail'
        ]
      ]
    ]
  ]
]
```

Notice that the operation makes use of self reference in order to recursively process the list. The recursion occurs through the *closed-in* environment that is part of an operation. Concrete syntax makes the object easier to work with:

```
λ[member](element,list)
  if element = list.head
  then true
  else member(element,list.tail)
```

The symbol λ introduces an operation. It is followed by an optional name within [and] that can be used for self reference. This is followed by the argument names, and finally the operation body. Since operations are objects just like everything else, we are free to mix objects and operations.

Operations form the basis of a wide range of list-manipulation. The following list operations are provided: ∀ where `l.∀(p)` is `true` when the predicate `p` returns `true` for each element in the list `l`; ∃ where `l.∃(p)` is `true` when the predicate `p` returns `true` for any element in the list `l`; `l.∃!(p)` is `true` when the predicate `p` returns `true` for exactly one element in the list `l`; `filter` where `l.filter(p)` is the list created by removing each element in turn that does not satisfy the predicate `p`; ∋ where `l.∋(x)` is true when the element `x` is contained in the list `l`; `select` where `l.select(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; `flatten` where `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.

Arguments of an operation will support patterns so that when the operation `λ((C)[s ↦ v])e` is supplied with an object of type `C` it extracts the value of the slot named `s` and binds it to the name `v` for the execution of the body `e`.

4.6. Naming, Scoping and Recursion

Objects can refer to themselves via their identity:

```
(Pair,ones)[
  head ↦ 1;
  tail ↦ ones
]
```

producing the list `[1,1,1,...]` The *scope* of such names is limited to the body of the object being defined. Therefore, the name `ones` above is limited to the values of the slots `head` and `tail`. Mutual recursion can be achieved in such a system if we assume that objects that wish to refer to each other are surrounded by an enclosing definition that itself is recursive, for example:

```
(Object,defs)[
  ones ↦ (Pair)[
    head ↦ 1;
    tail ↦ defs.twos
  ];
  twos ↦ (Pair)[
    head ↦ 2;
    tail ↦ defs.ones
  ]
]
```

producing the list `[1,2,1,2,1...]`.

Where an object contains named elements, we will typically use nested slots. Therefore, a dictionary can be constructed as:

```
(Dictionary,dict) [
  a      ↦ (Object)[...],
  b      ↦ (Object)[...],
  lookup ↦ λ(k,d)
    dict.slots.select(
      λ(s) s.name = k,
      λ(s) s.value,
      d)
]
```

Containers of named elements such as classes and packages support an operation called `lookup` which is supplied with a name and and a default value, and returns the value of the element with the supplied name or else returns the default value. The expression `dict.lookup('a',d)` is written `dict::a` to be consistent with namespace syntax in modelling languages such as UML.

4.7. Classification

A class is an object that *classifies* its *instances*. The classification relation is defined using constraints. The following is an object that classifies two-dimensional points:

```
(Class,Point)[
  name       ↦ 'Point';
  supers     ↦ [Object];
  attributes ↦ [
    (Attribute)[name ↦ 'x'; type ↦ Int],
    (Attribute)[name ↦ 'y'; type ↦ Int]
  ]
]
```

Note that the classifier of this object is `Class` and therefore such an object is termed a *class*. As in most object-oriented systems, the class has a name, super-classes, and defines some attributes that determine the slots that instances of the class must have. Features are inherited by sub-classes. Since classes are ubiquitous the following syntax is defined:

```
display((Class)[ name        ↦ n;
                 supers      ↦ S;
                 attributes  ↦ A;
                 constraints ↦ C;
                 operations  ↦ O]) =
  'class ' + n +  ' extends ' + S.displays() + '{' +
     A.displays() +
     'constraints {' +
       C.displays() +
     '}' +
     'operations {' +
       O.displays() +
     '}' +
  '}'
```

This leads to a fairly standard class-definition construct. The following class definition uses the new syntax for two-dimensional points:

```
class Point extends Object {
  x:Int;
  y:Int;
  operations {
    display() =
      '(' + x.display() + ',' + y.display() + ')'
  }
}
```

An important feature of classes is their ability to classify their instances. Every class defines a collection of constraints that must evaluate to `true` for any instance of the class. The candidate instance is supplied to the constraint as the value of the variable `self` and the slot-values of the candidate are available as variables with the same name as the slots:

```
class PosPoint extends Point {
  constraints {
    x >= 0 and y >= 0
  }
}
```

Now it is possible to check for any specific instance whether it conforms to the type or not: `PosPoint.instance?((10,20)) =true` and `PosPoint.instance?((-10,20))=false`. For every class `C` there is a predicate `C?` that is defined by `C? = λ(o) C.instance?(o)`.

A class may define operations that are available to its instances:

```
class Point extends Object {
  x:Int;
  y:Int;
  operations {
    distance(p) = $\sqrt{(x-p.x)^2 + (y-p.y)^2}$
  }
}
```

Now `(0,0).distance((2,3))=3.61`. The class `Attribute` is defined:

```
class Attribute extends Object {
  name:Str;
  atype:Class;
  operations {
    display() = name + ':' + atype.name
  }
}
```

The class `Operation` is defined with attributes `me` being the name used for recursive invocation within the operations, `env` containing bindings for free variables in the body, a sequence of argument names and a body. When the operation is invoked it is supplied with the target of the invocation that will be bound to the name `self` when the body is evaluated together with the argument values:

```
class Operation extends Object {
  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])
  }
}
```

A class `c` has an ordered list of super-classes `c.supers`. The kernel uses this relation to implement inheritance so that each feature defined by a super-class is available in the sub-class. For example, an attribute defined by a super-class is used when calculating the slots of an instance of a sub-class. The usual requirement that there are no cycles in the inheritance relation for a class is enforced. The super-classes of a class defaults to `[Object]`. Given that there are potentially multiple super-classes for `c`, the inheritance relation is a lattice where certain features may occur multiple times. There seems to be no agreed way to resolve such repetitions: some languages force explicit resolution, others impose a resolution rule. Given that the semantics of inheritance is extensible in the kernel, we provide a default rule: repeatedly inherited classes are resolved to their final occurrence in a left-to-right depth-first lattice traversal. Where named elements are redefined in different classes that occur in the flattened lattice, the first occurrence overrides later occurrences.

4.8. Meta-Types

The classifier of an object is also an object. Therefore, classes have classes and so on. Such an arrangement is an example of a *golden braid*: instance, class and meta-class (Hofstadter 1979). The meta-class `Class` is used as the classifier of the class `Point`. The kernel relies on a self-describing golden-braid. The following class is used to classify lists of the same type of element:

```
class Listof extends Class {
  etype:Class;
  operations {
    instance?(o) =
      o.list?() and o.∀(λ(x) etype.instance?(x))
    display() = '[' + etype.name + ']'
  }
}
```

The definitions given so far are sufficient to define the meta-class `Class` as an instance of itself. Note that we will incrementally define the class and use `def C::o(a...) = e` to mean an operation named `o` that is owned by class `C`.

```
class Class extends Object {
  name        : Str;
  supers      : [Class];
  attributes  : [Attribute];
  operations  : [Slot(Operation)];
  constraints : [Constraint];
}
```

The operations for `Class` define what we mean by inheritance of features and the structural relation that must hold between a class and its instances. We define a class to inherit from another class `c` as follows:

```
Class::supers() =
  [self] +
  [c | p ← supers; c ← p.supers()].remDups()
Class::inherits?(c) = supers().∃(c)
```

The super-classes of a class are used to define all the attributes and constraints that are available to it:

```
Class::atts() = [a|c ← supers(),a ← c.attributes]
Class::ops()  = [b|c ← supers(),b ← c.operations]
Class::cond() = [a|c ← supers(),a ← c.constraints]
```

A class is able to check whether it classifies a candidate object. The `instance?` relation checks that the candidate has a type that inherits from the receiver, checks that slots of the candidate match the attributes of the receiver and checks that all the constraints of the receiver hold for the candidate:

```
Class::instance?(o) =
  o.type.inherits?(self) and
  atts().∀(λ(a)
    o.slots.∃!(λ(s)
      s.name = a.name and
      a.type.instance?(s.value))) and
  o.slots.∀(λ(s)
    atts().∃!(λ(a)
      s.name = a.name and
      a.type.instance?(s.value))) and
  cond().∀(λ(c)
    c.eval([self ↦ o] +
           [s.name ↦ s.value | s ← o.slots]))
```

The definition of `Class` works in conjunction with `Object` to ensure that all instances are correct:

```
class Object {
  type:Class;
  slots:[Slot];
  constraints {
    type.instance?(self)
  }
  operations {
    dot(n) =
      slots.select(λ(m ↦ _) n=m,λ(_ ↦ v) v, error)
    send(n,vs) =
      type.ops().select(
        λ(m ↦ (Operation)[args ↦ as])
          n=m and #vs = #as,
        λ(_ ↦ f)
          f.invoke(self,vs),
        error)
    display() =
      '(' + name(type) + ')' +
      '[' + displays(slots) +  ']'
  }
}
```

The operations `dot` and `send` describe how the operators `_._` and `_._(_,...,_)` behave and provide the *meta-object protocol*. Just as atomic data are viewed as being governed by external rules, the expected behaviour is defined for any object whose slot access and operation invocation would be governed by the definitions given above. This does not preclude sub-classes of `Object` from redefining either of these operations, thereby providing language-centric MOPs.

Meta-classes can manipulate objects in terms of their fundamental components via `type`, `id` and `slots`. These slot names are axiomatic and have special meaning so that we can drill into object-structures to any depth without encountering data types that are unknown within the world of the kernel. The axiomatic `intern` operator is provided in order to restore the balance. Given an object `o`, the following rule must hold:

```
o = intern(o.type,o.id,o.slots)
```

4.9. Snapshots and Packages

The notion of *object* as a particular instance of a *class* can be generalised to the idea of a *snapshot* (object container) being an instance of a *package* (class container). This forms the basis of a wide range of specialised types of model.

A package is a class container and therefore specialises `Class` with a new attribute `classes`. The parents of a package must all be packages and the attributes of a package must all refer to classes defined by the package. The definition of `inherits?` is specialised to require that every class in a package extends a class in a parent package:

```
class Package extends Snapshot, Class {
  constraints {
    objects.∀(Class?);
    attributes.∀(λ(a)
      objects.∃(a.atype));
    supers.∀(λ(p)
      p.type.inherits?(Package)) }
  operations {
    display() = ... as shown concretely below...
    allObjects() =
      objects +
      [ p.objects | p ← supers ].flatten()
    inherits?(p) =
      objects.∀(λ(c)
        p.allObjects().∃(λ(s)
          c.inherits?(s)))
    instance?(o) =
      o.type.inherits?(Snapshot) and
      o.package.inherits?(self) and
      o.objects.∀(λ(o)
        objects.∃(λ(c)
          c.instance(o))) and
      Class::instance?(intern(self,o.slots))
  }
}
```

A snapshot is an object container that refers to a package in order to classify its contents. Since the slots of the snapshot itself are used for managing the relation between the objects it contains and the package that classifies those objects, an extra attribute called `bindings` is defined that can be used to name the objects in the snapshot. A constraint is used to ensure that the package correctly classifies the snapshot:

```
class Snapshot extends Object {
  package:Package;
  objects:[Object];
  bindings:[Slot];
  constraints {
    package.instance?(self);
    bindings.∀(λ(b)
      objects.contains?(b.value))
  }
  operations {
    display() = ... as shown concretely below ...
    lookup(k,d) =
      bindings.select(
        λ(s) s.name = k,
        λ(s) s.value,
        d)
  }
}
```

We can define some syntax for snapshots and packages. The following package defines a simple model for people and their pets. Since a package is also a snapshot, we must define its *meta-package*, in this case `Kernel`. The name of the package is `People`:

```
package People:Kernel
  class Person {
    name:Str;
    pets:[Animal]
  }
  class Animal {
    type:Str
  }
}
```

The following snapshot is an instance of `People`

```
snapshot s:People {
  (People::Person,fred)[
    name ↦ 'Fred';
    pets ↦ [s::fido]
  ];
  (People::Animal,fido)[
    type ↦ 'Dog'
  ]
}
```

Packages provide the basis of languages. Extensions of `Kernel` contain meta-classes that can be used in models. For example, classes in OCL expressions can be used to refer to all their instances: `C.allInstances()`. The following package extends `Class` with a list of its instances:

```
package IKernel:Kernel extends Kernel {
  class IClass extends Class {
    instances:[self];
  }
  class IPackage extends Package {
    constraints {
      objects.∀(IClass?)
    }
    operations {
      instance?(o) = ... as before... and
      o.objects ⊆      ⋃      c.instances
                  c∈objects
    }
  }
}
```

Models based on `IKernel` rather than `Kernel` create instances of `IPackage` instead of instances of `Package`. The new form of package requires each class to have a slot named `instances` that manages a list of their instances. Furthermore, any snapshot that is an instance of an `IPackage` can only contain objects that are elements of the class-instance lists.

4.10. Meta-Circularity

The kernel definition is completed with the following meta-circular definition for the package `Kernel` (assuming atomic types such as `Str` are builtin):

```
package Kernel:Kernel {
  class Object      { ... }
  class Slot        { ... }
  class Class       { ... }
  class Constraint  { ... }
  class Operation   { ... }
  class Arg         { ... }
  class Exp         { ... }
  class Listof      { ... }
  class Pair        { ... }
  class Nil         { ... }
  class Attribute   { ... }
  class Snapshot    { ... }
  class Package     { ... }
}
```

where the definitions for the classes have been given in the preceding sections. Bootstrapping the `Kernel` is achieved by creating a recursive data structure from basic objects that is consistent with the definition given above, together with an evaluator described below.

Figure 2 shows an evaluator for the kernel language. The function `eval` accepts a kernel expression `e` and an environment ρ and evaluates `e` in the context of ρ. The evaluator proceeds by case analysis on the expression `e`.

If `e` is an object expression then the result is a new object (note that operators starting with an upper-case letter are data constructors). The object references itself using the recursion operator `Y`.

A dot-expression handles the pseudo-slots `type`, `slots` and `id` specially. The `dot` function checks whether the value `o` has redefined the operation `dot` and uses message passing to invoke the redefined operation. If `dot` has not been redefined then the interpreter can handle the slot access directly.

A send-expression is handled specially in the case that the message is `invoke` and the target is an instance of `Operation`. In that case the interpreter can handle the message using `eval`. In the case of $+$, the interpreter can directly handle addition of numbers. The ellipses indicate that all other built-in operators such as $+$ are handled in the same way.

The essential interpretive actions are: invocation, slot-access and message-passing. Note how checks are performed by the interpreter that allow the basic definitions for these actions to be redefined. Note also that the definitions of operations in kernel classes `Operation` and `Object` are consistent with the default

```
eval(e,ρ) =
  case e {
    ObjExp(c,i,ss)    ⟶ Y(λ(o) Obj(eval(c,ρ[i ↦ o]),[n ↦ eval(e,ρ[i ↦ o]) | Slot(n,e) ← ss]))
    DotExp(e,n)       ⟶ dot(eval(e,ρ),n)
    SendExp(e,n,es)   ⟶ send(eval(e,ρ),n,[eval(e,ρ) | e ← es]
    InternExp(e1,e2)  ⟶ Obj(eval(e1,ρ),eval(e2,ρ))
    OpExp(i,ns,e)     ⟶ mkOp(ρ,i,ns,e)
  }

dot(o,n) =
  if redefinesDot?(o)
  then send(o,'dot',[n])
  else
    case n {
      'type'   ⟶ type(o)
      'slots'  ⟶ slots(o)
      'id'     ⟶ id(o)
      else getValue(getSlot(o,n))
    }

send(o,n,vs) =
  if redefinesSend?(o)
  then send(o,'send',[n,vs])
  else
    case n {
      'invoke' ⟶
        if isOp?(o)
        then eval(dot(o,'body'),
                  dot(o,'env') +
                  [dot(o,'me') ↦ o, 'self' ↦ head(vs)] +
                  [ n ↦ v | (n,v) ← dot(o,'args') * head(tail(vs)) ] +
                  dot(o,'slots') + send(type(o),'ops',[]))

        else send(o,'send',['invoke',[n,vs]])
      '+'  ⟶ if numbers?(vs)
             then add(vs)
             else send(getOp(o,n),'invoke',[o,vs])
      ...
      else send(getOp(o,n),'invoke',[o,vs])
    }
```

Figure 2 Meta-Circular Kernel Evaluator

actions of the interpreter. In addition note that the interpreter supports reflection in the sense that data in the interpreter (ρ for example) is available as slot-bindings. These features result in a meta-circular kernel.

In order for this to be meta-circular, we require that `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[1] and has been used to implement the rest of the toolkit 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.

[1] https://www.wi-inf.uni-duisburg-essen.de/LE4MM/

5. Concrete Syntax, Views and Diagrams

Languages are defined in terms of their *abstract syntax* and their *concrete syntax*. The abstract syntax is a machine-oriented view of the language that can be conveniently stored and manipulated by an algorithm. The concrete syntax is a human-oriented view of the language and consists of rules that govern how the abstract syntax is displayed on the page or the screen. Note that it is not possible to denote elements of a language abstractly; as soon as we make a mark on a page or screen, we are using a concrete syntax. This leads us to the conclusion that a language consists of a single abstract syntax, concretely expressed as a data model, and potentially many concrete syntaxes with a web of relationships holding the models together. To be specific, consider a traditional programming language, the abstract syntax is concretely defined as a collection of data type definitions, the concrete syntax is concretely defined as a BNF grammar and the parser defines a relation that holds the two together.

Modelling languages of the kind we are considering have an abstract syntax whose instances are objects and links. A convenient way to concretely represent such an abstract syntax is as a class diagram or as a text definition. We refer to both of these as *views* of the language, neither being more important than the other. Consider the class diagram: it is essentially a graph where nodes are labelled with boxes and text. Therefore, we can create a model of such diagrams and concretely represent that model as a class diagram. The modelling language must be related to the class diagram model; this can be achieved (as shown below) as a relationship model R that re-uses existing language constructs. Therefore, the model of class diagrams is related to itself via R. Again, we find that meta-circularity has appeared since the model of class diagrams is represented as a self-describing class diagram.

Concrete syntax can take many forms. Earlier examples in this article showed a `display()` operation used to implement the relation between models and the very simple concrete syntax model for strings. Diagrams are used extensively for models in order to convey structure and relationships (Moody 2009); therefore this section addresses diagrams and how concrete syntax models of diagrams can be related to the abstract syntax models of a language. In principle, a diagram can show objects from any mixture of different levels in the golden braid. The following is a simple package that represents diagrams based on graphs:

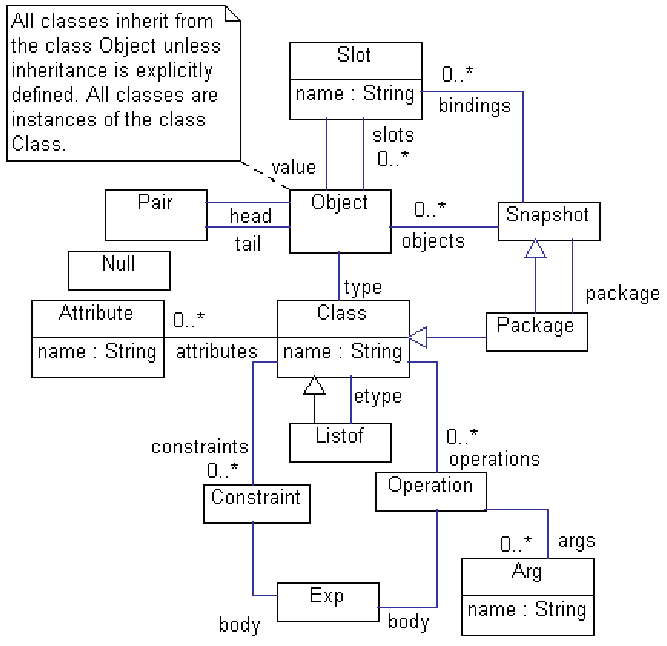

Figure 3 `Kernel` Abstract Syntax as a Diagram

```
package Diagrams:Kernel {
  class Diagram extends Object {
    N:[Node];
    E:[Edge];
    constraints {
      [ e.src | e ← E ] ⊆ N;
      [ e.tgt | e ← E ] ⊆ N
    }
  class Node extends Object {
    x:Int;
    y:Int;
    width:Int;
    height:Int;
    operations {
      above?(n) = y > n.y + n.height
      isIn?(n) =
        isIn?(n.x,n.y) and
        isIn?(n.x + n.width, n.y + n.height)
      isIn?(x!,y!) =
        x! >= x          and
        x! <= x + width  and
        y! >= y          and
        y! <= y + height
    }
  }
  class Box extends Node {
    contents:[Node];
    constraints {
      contents.∀(λ(d)
        isIn?(d)) and
        width=max([d.width | d ← contents]) and
        height = max([d.height | d ← contents])
      }
  }
  class Text extends Node {
    text:Str;
    constraints {
      width  = #(text) * CHARWIDTH;
      height = CHARHEIGHT
    }
  }
  class Triangle extends Node {
    operations {
      incident?(n) =
        (x + width/2,y + height) =
        (n.x + n.width/2,n.y)
    }
  }
  class Edge extends Object {
    src:Node;
    target:Node;
    label:Node;
  }
}
```

A diagram consists of nodes and edges. A node has a position and may be a box, some text or a triangle. A box contains other displayable elements. A triangle implements a predicate that determines whether it is incident on the base of a node. Edges gave source and target nodes and use nodes as labels.

A diagram model such as that shown in figure 4 can be used as the basis of a wide range of views for different packages of model element. Figure 3 shows the `Kernel` package drawn using these diagram elements and figure 4 shows the `Diagrams` package drawn as a diagram. The rest of this section describes an approach for constructing graphical views of models that is built entirely from modelling elements defined in the `Kernel`.

The approach uses relationship classes to associate types of model element with diagram elements. The relationships define a view on the models in much the same way as a model-view-controller. The underlying modelling language does not know anything about how its elements are being represented on a diagram so that it is easy to define multiple views.

Consider the following definition of `Packages_x_Diagrams` that defines a simple representation for class diagrams. The key idea is to set up a collection of relationship classes of the form `A_x_B` that relate instances of `A` to instances of `B`. The constraints on the relationship class limit particular instances of `B` to be associated with an instance of class `A`, and vice versa.

Where instances of class `A` can refer (directly or transitively) to instances of type `G`, and similarly where `B` can refer to instances of type `H` then it is convenient to allow instances of the

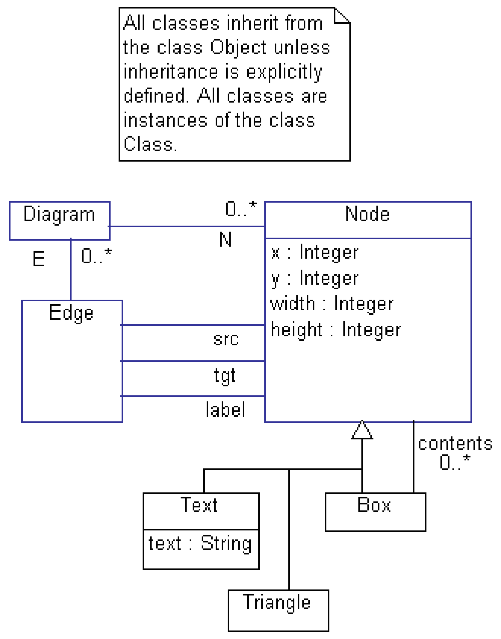
All classes inherit from the class Object unless inheritance is explicitly defined. All classes are instances of the class Class.
Diagram
Node
x : Integer
y : Integer
width : Integer
height : Integer
0..*
N
E
0..*
Edge
src
tgt
label
contents
0..*
Text
text : String
Box
Triangle

Figure 4 `Diagrams` as a Diagram

relationship class `A_x_B` to refer to instances of a further relationship class `G_x_H` such that for any `a_x_b:A_x_B` the following constraints hold:

```
a_x_b.a.g = a_x_b.g_x_h.g
a_x_b.b.h = a_x_b.g_x_h.h
```

where `a_x_b.g_x_h` refers to the (suitable constrained) instance of `g_x_h` associated with an instance of `a_x_b`. In this particular case, the relationship class `Package_x_Diagram` defines those diagrams that are appropriate views of a given package. We deliberately leave the relation underspecified so that it is only defined up to an equivalence class of possible layouts. The root class `Package_x_Diagram` associates a package with a diagram in such a way that classes are represented as nodes and inheritance relationships are represented as edges between the appropriate nodes:

```
package Packages_x_Diagrams:Kernel {
  class Package_x_Diagram extends Object {
    p:Package;
    d:Diagram;
    N:[Class_x_Node];
    M:[Attribute_x_Association];
    E:[InheritanceEdge];
    A:[Association];
    constraints {
      p.objects = [n.c | n ← N];
      d.N       = [n.n | n ← N];
      d.E       = E ∪ A;
      [ [i.src, i.tgt] | i ← E] =
        [ [s.n, t.n] | s ← N,
                       t ← N,
                       ?s.c.parents.∋(t.c) ];
      [ a | c ← p.objects, a ← c.attributes ] =
        [ m.a | m ← M ] ∪
        [ a.a | n ← N, a ← n.atts]
    }
  }
}
```

The relationship class `Package_x_Diagram` associates a package `p` with a diagram `d`. The sub-relationships `N` and `M` ensure that the classes are correctly associated with diagram nodes and the attributes are correctly associated with association-edges between classes where the type of the attribute is not simple (*i.e.*, one of `Str`, `Int`, *etc.*)

Constraints on `Package_x_Diagram` enforce relationships between instances of the associated classes: all the classes in the package have exactly one node on the diagram and *vice versa*. The edges on the diagram are either inheritance or association where the inheritance edges hold when one class inherits from another. An attribute is either shown as text within a class-node or as an association-edge.

A class-node is a box with a name above a collection of attributes:

```
class ClassNode extends Box {
  name:Text;
  attributes:[Text];
  constraints {
    contents.∋(name);
    attributes.∀(λ(a)
      contents.∋(a) and
      name.above?(a)))
  }
}
```

The relationship between a class and a class-node is defined as follows:

```
class Class_x_Node extends Object {
  c:Class;
  n:ClassNode;
  atts:[Field_x_Text];
  constraints {
    n.name = c.name.text;
    [a.a | a ← atts] ⊆ c.attributes;
    n.attributes = [a.t | a ← atts]
  }
}
```

The name in the class-box must be the same as that of the class. Not all attributes of the class need be shown as fields within the class-node because a field is required to have an atomic type as shown below.

```
class Field_x_Text extends Object {
  a:Attribute;
  t:Text;
  constraints {
    text.text = a.name + ':' + a.atype.name;
    ['Str','Int',...].∋(text.type.name)
  }
}
```

An attribute that is displayed as an association must have a non-atomic data type. Whether the multiplicity is shown as `*` or not is determined by whether the type is an instance of the Kernel class `Listof`:

```
class Association extends Edge {
  name:Text;
  mult:Text;
}
class Attribute_x_Association extends Object {
  a:Attribute;
  r:Association;
  constraints {
    a.name = r.name.text;
```

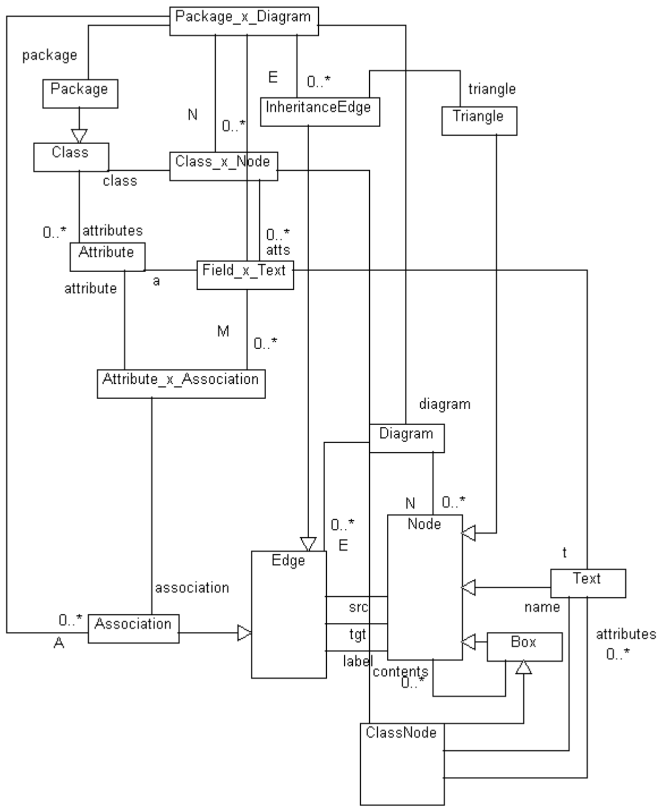

Figure 5 `Package_x_Diagram` as a Diagram

```
    a.atype = Listof ⟹ mult.text = '0..*';
    a.atype ≠ Listof ⟹ mult.text = '';
    not(['Str','Int',...].∋(a.atype.name))
  }
}
```

Finally, an inheritance edge only holds between class-nodes and includes a triangle that is incident upon the super-class:

```
class InheritanceEdge extends Edge {
  t:Triangle;
  constraints {
    source.type = ClassNode;
    target.type = ClassNode;
    label = t;
    label.incident?(target)
  }
}
```

Figure 5 shows the relation between packages and diagrams viewed as a diagram. Given the three packages `Kernel`, `Diagrams` and `Packages_x_Diagrams` it is possible to display graphical views for *any* package. Therefore, we have used language definition to view them in two different ways: textual and graphical. In addition, it is always possible to have a *lowest common denominator* view of a package: as a nested collection of objects. This is only one of many different representations for the configuration of objects that conform to the type `Package`, thereby supporting our proposal that `Kernel` is a basis for modelling as *language engineering*.

6. Evaluation

The evaluation criteria for the `Kernel` language is to exhibit the features defined in section 3. This section discusses how `Kernel` supports the foundational features and then proceeds to define two languages using `Kernel`: a MOF-like language that is a simplification of `Kernel` and a potency-based language where attributes can be labelled with instantiation levels.

6.1. Evaluation of Key Features

Section 3 set out the key features of a meta-circular language for model-based language engineering. The `Kernel` language achieves **introspection** and **reification** through the uniform representation of all data as objects and the meta-slots `type`, `slots`, `id`, and the meta-operation `intern()`. There is no limit to the depth of introspection such that models can be transformed and re-constructed to achieve **intercession**. Execution is based exclusively on a MOP in terms of `dot` and `send`, which allows the language system to be arbitrarily extensible and therefore **intercession** is achieved within the context of a specification language. All aspects of the language have corresponding **referents** that can be **reflected** upon via the meta-operations. Since the type of an object is also an object and `Class` is an instance of itself, the `Kernel` achieves an infinite self-contained **reflective-tower**.

Although the `Kernel` does not contain basic modelling elements it has been designed to support the key features required by languages such as MOF. We have shown that classes, attributes, operations and constraints are easily supported and can be varied through specialisation. Packages and snapshots provide a basis for constructing both language definitions and user-level models. Views can be supported by following a pattern of specification for a model-view controller. Although each object is limited to have a single class that is the value of the slot named `type`, this is sufficient to support a range of classification strategies including languages that support multiple roles (Georg & France 2002) or powertypes (Gonzalez-Perez & Henderson-Sellers 2006; Henderson-Sellers & Gonzalez-Perez 2005; Clark et al. 2014).

6.2. A MOF-based Language

MOF is very similar to `Kernel` in that it is a self-describing extensible structural model, although MOF does not contain an expression evaluator that is necessary to support constraints. MOF can be defined as an extension to `Kernel` where the key elements of MOF are simply extensions to the equivalent elements in `Kernel` with some additional constraints. MOF introduces some extra features that are not present in `Kernel` such as `Namespace`, `PackageableElement`, and `Classifier`. All of these elements can be defined as extensions of `Kernel::Object` and `Kernel::Class` with suitable constraints. Space precludes a complete definition, although the following provides an outline of the approach:

```
package MOF:Kernel extends Kernel {
  class MOFClass extends Class {}
  class MOFStructuralFeature extends Attribute {}
  class MOFPackage extends Package {
    constraints {
```

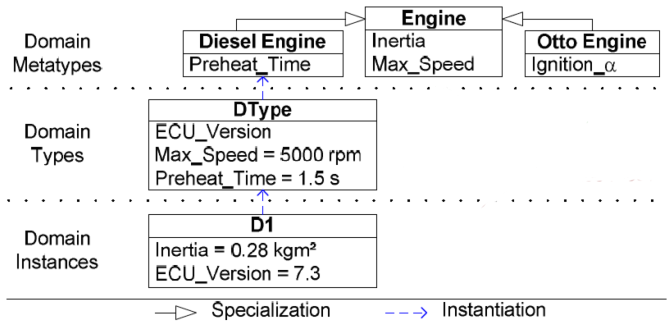

Figure 6 A model written in a potency-based language (adapted from (Aschauer et al. 2009))

```
      objects.∀(λ(o)
        MOFClass.instance?(o))
    }
  }
  ...
}
```

Note that `MOF` inherits all features from `Kernel`, but the additional constraint limits `MOF` packages to contain `MOF` classes. This definition is representative of the class of MOF-like languages that can be defined as an extension of `Kernel` with some additional structural constraints.

6.3. A Potency-Based Language

Deep instantiation, jointly with potency, has been proposed as a mechanism that allows multi-level modelling (de Lara & Guerra 2020). As has been established at the start of this article, everything is an object and therefore there are no constraints on level-mixing other than those that are defined within the language definitions themselves. Potency allows attributes to be extended with extra information in the form of a numeric value describing how many type levels must be spanned by instance-of relations before the attribute is actually instantiated. For example, an attribute with potency 1 behaves as a regular attribute, yielding a slot when its class gets instantiated. However, an attribute with potency 2 yields an identical attribute with potency 1 when its class gets instantiated; one extra instance-of relation would be necessary to obtain a slot for this attribute.

Our claim is that the kernel language proposed here is a suitable basis for all such languages. Consider the model shown in figure 6 (taken from (Aschauer et al. 2009)) that shows a language (Domain Metatypes) of engines. The class `Engine` defines an attribute called `max_speed` with potency 1 that results in a slot at the domain type (model) level, and an attribute called `inertia` with potency 2 that becomes a slot at a remove of two type levels.

The engine-based language involves some novel use of attributes at the class level. This can be achieved using our approach by extending `Kernel` with two new meta-classes called `DAtt` and `DClass` as follows:

```
package DKernel:Kernel extends Kernel {
  class DAtt extends Attribute {
    level:Int;
    operations {
      display() = name + '[' + level + ']:' + type
    }
  }
  class DClass extends Class {
    operations {
      atts() = datts(1,self)
      datts(n,c=(_,c)[]) = []
      datts(n,c) =
        [a | a ← c.atts(),?a.type=DAtt,a.level=n] +
        datts(n+1,c.type)
    }
    constraints {
      atts.∀(λ(a)
        a.type = DAtt)
    }
  }
}
```

A *deep-attribute* is an instance of `DAtt` that extends attributes with a potency level. The idea is that this level defines the number of instance-of relationships that must be traversed before the attribute turns into a slot. In addition, a deep-attribute supports the level as part of the `display`.

A *deep-class* is a class whose attributes are deep-attributes and that redefines the operation `atts()` in order to take potency into account. The new definition calculates a list of attributes using `datts` in terms of a level `n` and a class `c`. If the class is self-describing (essentially `Class`) then no deep-attributes are produced. Otherwise, only those deep-attributes with level `n` are returned in addition to those from the class of `c` with level `n+1`. This has the effect of ensuring that the potency-level is correctly applied to any instance of a deep-class[2].

Having defined a new language, we can use it to define the models shown in figure 6. Firstly, the meta-model is defined as an instance of the new language. Each class is an instance of `DClass` and is therefore required to define deep-attributes:

```
package DomainMetaTypes:DKernel  {
  class Engine:DClass extends DClass {
    inertia[2]:Float;
    max_speed[1]:Int;
  }
  class DieselEngine:DClass extends Engine {
    preheat_time[1]:Float
  }
  class OttoEngine:DClass extends Engine {
    ignition_alpha[1]:Float
  }
}
```

A model written in the language of domain meta-types has classes that are instances of `Engine` or one of its sub-classes. As such an engine-class will have slots corresponding to deep-attributes with potency level 1:

```
package DomainTypes:DomainMetaTypes {
  class DType:DieselEngine {
    ECU_version[1]:Float;
    max_speed=5000;
    preheat_time=1.5;
  }
}
```

A snapshot that is an instance of the model defined above will

[2] An implementation of a fully-defined language would need to override the `atts()` operation so that it can be used to compute attributes that are used in different situations including class instantiation, class display, inheritance, *etc.*

contain objects whose slots correspond to the deep-attributes with the appropriate potency-levels as shown below:

```
snapshot DomainInstances:DomainTypes {
  (DType)[
    inertia=0.28;
    ECU_version=7.3
  ]
}
```

There are three occurrences of the golden braid in this example including one reflexive traversal of `Kernel:Kernel`). `Kernel`, `MOF` and `DKernel` are examples of meta-languages that can be defined in the framework, and we claim that other languages, such as those described in (Atkinson et al. 2014) can also be represented and analysed.

The meta-classes in `DKernel` can be reused across multiple languages (`DomainMetaType` being one example), unlike the meta-classes of Smalltalk. Given that the semantics of `Kernel` is defined in terms of a constraint that uses `atts`, and that the `parents` relation allows redefinition of operations, this means that the core semantics of the `Kernel` is both extensible and replaceable. The redefinition of `display` in `DAtt` provides a domain-specific view of the new language and allows potency values to be specified as part of an attribute definition.

7. Conclusion

Our aim is to produce a meta-circular level-agnostic basis for model-based language engineering. We have reviewed current advances in meta-modelling and defined a self-contained new language that supports an arbitrary number of golden braid occurrences based on a single representation for all model elements, that is highly extensible including its own core semantics. We have also shown how the language can support both text and graphical views that are equivalent to domain-specific languages.

The kernel language is simple and can be implemented as demonstrated by the XMF and XModeler toolkit that is capable of both describing and reasoning about itself. The toolkit was reported as a leading technology for Software Engineering (Helsen et al. 2008) and has been used for a variety of applications including modelling languages for aerospace applications, telecoms applications (Achilleos et al. 2007), and enterprise modelling languages and methods (Johanndeiter et al. 2013; Frank 2014, 2019).

Our intention is that the kernel language defined in this article provides a basis for ourselves and others to experiment with language definitions. Language features such as objects having multiple types, mixed static and dynamic typing, delegation, prototypes, aspects, facets, multi-level modelling, projective views, weaving, product-lines, and different approaches to modularity, can all be constructed and made to interoperate.

In (Gogolla et al. 2005), the authors show how the golden braid can be represented on a single object-diagram. This allows OCL constraints to range over the all levels and thereby support clabjects and potency. This is very similar to the approach developed in this article and it would be interesting to see how the two approaches can co-exist through views, perhaps through the use of projectional editors (Voelter & Pech 2012). 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 and thereby develop a shared library.

The `Kernel` has some limitations that we plan to address through further work. Several modelling frameworks support action languages that can create and modify objects. The `Kernel` is a specification language currently does not deal with side-effects and state, but could be extended with features such as pre and post-conditions as used in OCL. The `Kernel` lacks a formal semantics that can be used to validate it in terms of completeness and consistency. All `Kernel` objects are an instance of exactly one class which might be seen as a limitation. We claim that languages where objects can exist in multiple classification-states at the same time can be defined, although this has yet to be demonstrated.

Finally, the use of reflection and meta-circularity through a MOP can compromise the efficiency of language execution. To a greater extent, the purposes of this article is to demonstrate that a meta-circular kernel can be defined and shown to support a variety of useful language extensions. Efficiency can be achieved, as demonstrated by XModeler where most of the tooling (approximately 200k lines of source code) is written in a kernel-like language that compiles to a bespoke virtual machine written in Java. In making XModeler efficient, care is taken to identify standard data usage that can be handled conventionally, and those uses that require a MOP. In addition, XModeler adds side effects to the kernel and supports updates to definitions such as those presented in (de Lara et al. 2018). Adding actions to the kernel language presented in this article would achieve the same effect at the expense of losing referential transparency.

References

Abadi, M., & Cardelli, L. (2012). *A theory of objects*. Springer Science & Business Media.

Achilleos, A., Georgalas, N., & Yang, K. (2007). An open source domain-specific tools framework to support model driven development of oss. In *Model driven architecture-foundations and applications* (pp. 1–16).

Ancona, M., & Cazzola, W. (2004). Implementing the essence of reflection: a reflective run-time environment. In *Proceedings of the 2004 acm symposium on applied computing* (pp. 1503–1507).

Aschauer, T., Dauenhauer, G., & Pree, W. (2009). Representation and traversal of large clabject models. In *Model driven engineering languages and systems* (pp. 17–31). Springer.

Atkinson, C. (1997). Meta-modelling for distributed object environments. In *Enterprise distributed object computing workshop [1997]. edoc'97. proceedings. first international* (pp. 90–101).

Atkinson, C. (1999). Supporting and applying the UML conceptual framework. In *The unified modeling language. uml 98: Beyond the notation* (pp. 21–36). Springer.

Atkinson, C., & Gerbig, R. (2016). Flexible deep modeling with melanee. *Modellierung 2016-Workshopband*.

Atkinson, C., Gerbig, R., & Kühne, T. (2014). Comparing multi-level modeling approaches. In *Proceedings of the workshop on multi-level modelling co-located with ACM/IEEE 17th international conference on model driven engineering languages & systems (models 2014), valencia, spain, september 28, 2014.* (pp. 53–61). Retrieved from http://ceur-ws.org/Vol-1286/p6.pdf

Atkinson, C., Kennel, B., & Goß, B. (2011a). The level-agnostic modeling language. In *Software language engineering* (pp. 266–275). Springer.

Atkinson, C., Kennel, B., & Goß, B. (2011b). 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).*

Atkinson, C., & Kühne, T. (2001). The essence of multi-level metamodeling. In *Uml 2001 the unified modeling language. modeling languages, concepts, and tools* (pp. 19–33). Springer.

Atkinson, C., & Kühne, T. (2003). Model-driven development: A metamodeling foundation. *IEEE Software*, *20*(5), 36-41.

Atkinson, C., & Kühne, T. (2005). Concepts for comparing modeling tool architectures. In *Model driven engineering languages and systems* (pp. 398–413). Springer.

Atkinson, C., & Kühne, T. (2015). In defence of deep modelling. *Information and Software Technology*, *64*, 36–51.

Bézivin, J. (2005). On the unification power of models. *Software & Systems Modeling*, *4*(2), 171–188.

Brand, M. v. d., Verhoeff, T., et al. (2018). Exploration of modularity and reusability of domain-specific languages. *Computer Languages, Systems and Structures*, *51*(C), 48–70.

Bry, F. (2018). In praise of impredicativity: A contribution to the formalization of meta-programming. *Theory and Practice of Logic Programming*, 1–48.

Cabot, J., & Gogolla, M. (2012). Object constraint language (ocl): a definitive guide. In *International school on formal methods for the design of computer, communication and software systems* (pp. 58–90).

Chari, G., Garbervetsky, D., Marr, S., & Ducasse, S. (2018). Fully reflective execution environments: Virtual machines for more flexible software. *IEEE Transactions on Software Engineering*, *45*(9), 858–876.

Clark, T., & Frank, U. (2018). Multi-level constraints. In *Ceur workshop proceedings* (Vol. 2245, pp. 103–117).

Clark, T., & Frank, U. (2020). Multi-level modelling with the fmmlx and the xmodelerml. *Modellierung 2020.*

Clark, T., Gonzalez-Perez, C., & Henderson-Sellers, B. (2014). A foundation for multi-level modelling. In *Proceedings of the workshop on multi-level modelling co-located with ACM/IEEE 17th international conference on model driven engineering languages & systems (models 2014), valencia, spain, september 28, 2014.* (pp. 43–52). Retrieved from http://ceur-ws.org/Vol-1286/p5.pdf

Clark, T., Sammut, P., & Willans, J. (2015a). Applied metamodelling: a foundation for language driven development. *arXiv preprint arXiv:1505.00149*.

Clark, T., Sammut, P., & Willans, J. (2015b). Super-languages: Developing languages and applications with xmf. *arXiv preprint arXiv:1506.03363*.

Clark, T., & Tratt, L. (2010). Formalizing homogeneous language embeddings. *Electronic Notes in Theoretical Computer Science*, *253*(7), 75 - 88. Retrieved from http://www.sciencedirect.com/science/article/pii/S157106611000112X (Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)) doi: http://dx.doi.org/10.1016/j.entcs.2010.08.033

Cointe, P. (1987). Metaclasses are first class: the objvlisp model. In *Acm sigplan notices* (Vol. 22, pp. 156–162).

Cointe, P. (1996). Reflective languages and metalevel architectures. *ACM Comput. Surv.*, *28*(4es), 151.

Combemale, B., Deantoni, J., Baudry, B., France, R. B., Jézéquel, J.-M., & Gray, J. (2014). Globalizing modeling languages. *Computer*, *47*(6), 68–71.

Cuadrado, J. S., & de Lara, J. (2018). Open meta-modelling frameworks via meta-object protocols. *Journal of Systems and Software*, *145*, 1–24.

Culpepper, R., Felleisen, M., Flatt, M., & Krishnamurthi, S. (2019). From macros to dsls: The evolution of racket. In *3rd summit on advances in programming languages (snapl 2019).*

De Lara, J., & Guerra, E. (2010). Deep meta-modelling with metadepth. In *International conference on modelling techniques and tools for computer performance evaluation* (pp. 1–20).

de Lara, J., & Guerra, E. (2020). Multi-level model product lines: Open and closed variability for modelling language families. In *Fundamental approaches to software engineering: 23rd international conference, fase 2020, held as part of the european joint conferences on theory and practice of software, etaps 2020, dublin, ireland, april 25–30, 2020, proceedings 23* (pp. 161–181).

de Lara, J., Guerra, E., Cobos, R., & Moreno-Llorena, J. (2012). Extending deep meta-modelling for practical model-driven engineering. *The Computer Journal*, bxs144.

de Lara, J., Guerra, E., & Cuadrado, J. S. (2015). Model-driven engineering with domain-specific meta-modelling languages. *Software & Systems Modeling*, *14*(1), 429–459.

de Lara, J., Guerra, E., Kienzle, J., & Hattab, Y. (2018). Facet-oriented modelling: Open objects for model-driven engineering. In *Proceedings of the 11th acm sigplan international conference on software language engineering* (pp. 147–159).

Doan, K., & Gogolla, M. (2018). Extending a UML and OCL tool for meta-modeling: Applications towards model quality assessment. In I. Schaefer, D. Karagiannis, A. Vogelsang, D. Méndez, & C. Seidl (Eds.), *Modellierung 2018, 21.-23. februar 2018, braunschweig, germany* (Vol. P-280, pp. 135–150). Gesellschaft für Informatik e.V. Retrieved from https://dl.gi.de/20.500.12116/14935

Favre, L., & Duarte, D. (2016). Formal mof metamodeling and tool support. In *2016 4th international conference on model-driven engineering and software development (modelsward)* (pp. 99–110).

Fouquet, F., Nain, G., Morin, B., Daubert, E., Barais, O., Plouzeau, N., & Jézéquel, J.-M. (2012). An eclipse modelling framework alternative to meet the models@ runtime requirements. In *International conference on model driven engineering languages and systems* (pp. 87–101).

Frank, U. (2014). Multi-perspective enterprise modeling: foundational concepts, prospects and future research challenges. *Software and System Modeling*, *13*(3), 941-962.

Frank, U. (2019). Specification and management of methods- a case for multi-level modelling. In *Enterprise, business-process and information systems modeling* (pp. 311–325). Springer.

Georg, G., & France, R. (2002). Uml aspect specification using role models. In *Object-oriented information systems* (pp. 186–191). Springer.

Gil, J., Howse, J., & Kent, S. (1999). Constraint diagrams: a step beyond uml. *Proceedings of TOOLS USA'99*.

Gogolla, M., Favre, J.-M., & Büttner, F. (2005). On squeezing m0, m1, m2, and m3 into a single object diagram. *Proceedings Tool-Support for OCL and Related Formalisms-Needs and Trends*.

Gogolla, M., & Vallecillo, A. (2019). On softening OCL invariants. *Journal of Object Technology*, *18*(2), 6:1–22. Retrieved from https://doi.org/10.5381/jot.2019.18.2.a6 doi: 10.5381/jot.2019.18.2.a6

Gonzalez-Perez, C., & Henderson-Sellers, B. (2006). A powertype-based metamodelling framework. *Software & Systems Modeling*, *5*(1), 72–90.

Helsen, S., Ryman, A., & Spinellis, D. (2008). Where's my jetpack? *Software, IEEE*, *25*(5), 18–21.

Henderson-Sellers, B. (2011). Bridging metamodels and ontologies in software engineering. *Journal of Systems and Software*, *84*(2), 301–313.

Henderson-Sellers, B. (2012). *On the mathematics of modelling, metamodelling, ontologies and modelling languages*. Springer.

Henderson-Sellers, B., & Gonzalez-Perez, C. (2005). Connecting powertypes and stereotypes. *Journal of Object Technology*, *4*(7), 83–96.

Henderson-Sellers, B., & Gonzalez-Perez, C. (2008). Standardizing methodology metamodelling and notation: an ISO exemplar. In *Information systems and e-business technologies* (pp. 1–12). Springer.

Henderson-Sellers, B., & Unhelkar, B. (2000). *Open modeling with uml*. Pearson Education.

Herzeel, C., Costanza, P., & Hondt, T. (2008). Reflection for the masses. In *Self-sustaining systems* (pp. 87–122). Springer.

Hofstadter, D. R. (1979). *Gödel, escher, bach*. Harvester press Hassocks, Sussex.

Jácome-Guerrero, S. P., & de Lara, J. (2020). Totem: Reconciling multi-level modelling with standard two-level modelling. *Computer Standards & Interfaces*, *69*, 103390.

Jeusfeld, M. A. (2019). Deeptelos demonstration. In *2019 acm/ieee 22nd international conference on model driven engineering languages and systems companion (models-c)* (pp. 98–102).

Johanndeiter, T., Goldstein, A., & Frank, U. (2013). Towards business process models at runtime. In *Models@run.time* (p. 13-25).

Kiczales, G., Des Rivieres, J., & Bobrow, D. G. (1991). *The art of the metaobject protocol*. MIT press.

Kon, F., Costa, F., Blair, G., & Campbell, R. H. (2002). The case for reflective middleware. *Communications of the ACM*, *45*(6), 33–38.

Kühne, T. (2006). Matters of (meta-) modeling. *Software & Systems Modeling*, *5*(4), 369–385.

Liu, L., & Özsu, M. T. (Eds.). (2018). *Encyclopedia of database systems, second edition*. Springer. Retrieved from https://doi.org/10.1007/978-1-4614-8265-9 doi: 10.1007/978-1-4614-8265-9

Moody, D. L. (2009). The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Software Eng.*, *35*(6), 756-779.

Oquendo, F. (2016). Formally describing the software architecture of systems-of-systems with sosadl. In *2016 11th system of systems engineering conference (sose)* (pp. 1–6).

Polito, G., Ducasse, S., Bouraqadi, N., & Fabresse, L. (2015). A bootstrapping infrastructure to build and extend pharo-like languages. In *2015 acm international symposium on new ideas, new paradigms, and reflections on programming and software (onward!)* (pp. 183–196).

Pschorn, P., Oliveira Antonino, P., Morgenstern, A., & Kuhn, T. (2019). A constraint modeling framework for domain-specific languages. In *Proceedings of the 17th acm sigplan international workshop on domain-specific modeling* (pp. 20–29).

Quinlan, D., Wells, J. B., & Kamareddine, F. (2019). Bnf-style notation as it is actually used. In *International conference on intelligent computer mathematics* (pp. 187–204).

Reinhartz-Berger, I., Sturm, A., Clark, T., Cohen, S., & Bettin, J. (Eds.). (2013). *Domain engineering, product lines, languages, and conceptual models*. Springer.

Richters, M., & Gogolla, M. (1998). On formalizing the uml object constraint language ocl. In *International conference on conceptual modeling* (pp. 449–464).

Sánchez-Cuadrado, J., De Lara, J., & Guerra, E. (2012). Bottom-up meta-modelling: An interactive approach. In *Model driven engineering languages and systems* (pp. 3–19). Springer.

Segura, Á. M., & de Lara, J. (2019). Extremo: An eclipse plugin for modelling and meta-modelling assistance. *Science of Computer Programming*, *180*, 71–80.

Selic, B. (2012). What will it take? a view on adoption of model-based methods in practice. *Software & Systems Modeling*, *11*(4), 513–526.

Urbán, D., Theisz, Z., & Mezei, G. (2018). Self-describing operations for multi-level meta-modeling. In *Modelsward* (pp. 519–527).

Van Tendeloo, Y., & Vangheluwe, H. (2017a). Explicitly modelling the type/instance relation. In *Proceedings of models 2017 satellite event, september 17, 2017, austin, texas, usa/burgueño, loli [edit.]* (pp. 393–398).

Van Tendeloo, Y., & Vangheluwe, H. (2017b). The modelverse: a tool for multi-paradigm modelling and simulation. In *2017 winter simulation conference (wsc)* (pp. 944–955).

Voelter, M., & Pech, V. (2012). Language modularity with the mps language workbench. In *Software engineering (icse),*

2012 34th international conference on (pp. 1449–1450).
Volz, B., & Jablonski, S. (2010). Towards an open meta modeling environment. In *Proceedings of the 10th workshop on domain-specific modeling* (p. 17).

About the author

Tony Clark is a professor of Software Engineering and Deputy Dean in the College of Engineering and Physical Science at Aston University, UK. His current research focus is on software modelling tools and programming languages. His work has produced the XModeler toolkit and has contributed to the design of the UML standard. You can contact the author at tony.clark@aston.ac.uk.