Multi-Level Constraints

Tony Clark[1] and Ulrich Frank[2]

[1] Aston University, UK, tony.clark@aston.ac.uk
[2] University of Duisburg-Essen, DE, ulrich.frank@uni-duisburg-essen.de

Abstract. Meta-modelling and domain-specific modelling languages are supported by multi-level modelling which liberates model-based engineering from the traditional two-level type-instance language architecture. Proponents of this approach claim that multi-level modelling increases the quality of the resulting systems by introducing a second abstraction dimension and thereby allowing both intra-level abstraction via sub-typing and inter-level abstraction via meta-types. Modelling approaches include constraint languages that are used to express model semantics. Traditional languages, such as OCL, support intra-level constraints, but not inter-level constraints. This paper motivates the need for multi-level constraints, shows how to implement such a language in a reflexive language architecture and applies multi-level constraints to an example multi-level model.

1 Introduction

Conceptual models aim to bridge the gap between natural languages that are required to design and use a system and implementation languages. To this end, general-purpose modelling languages (GPML) like the UML consist of concepts that represent semantic primitives such as *class*, *attribute*, *etc.*, that, on the one hand correspond to concepts of foundational ontologies, *e.g.*, [4], and on the other hand can be nicely mapped to corresponding elements of object-oriented programming languages.

Since GPML can be used to model a wide range of systems, they promise attractive economies of scale. At the same time, their use suffers from the fact that they offer generic concepts only. Domain-specific modelling languages (DMSL) address this limitation by providing concepts drawn from a particular domain of interest. Thus, modellers are not forced to find ways of representing specific concepts in terms of more general modelling element types, but can reuse (hopefully) thoroughly specified concepts that come with a DSML.

While DSMLs promise clear advantages with respect to productivity and model quality, their construction and use are compromised by serious challenges. First, the designer of a DSML must decide whether a concept should be part of the language or be modelled *using* the language. In many cases there are no clear criteria that would support such a decision. Second, the design of a DSML requires dealing with a trade-off between range of reuse and productivity of reuse (see figure 1).

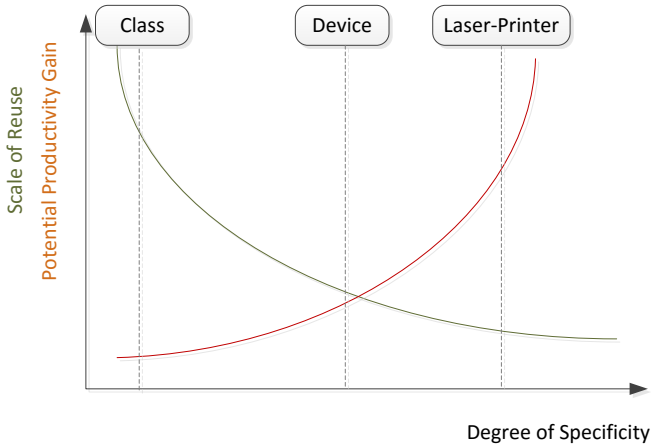

Fig. 1. Principal Conflict between Range of Reuse and Productivity of Reuse

Multi-level modelling is suited to overcome these challenges by allowing an arbitrary number of classification levels. As a consequence, traditional OCL constraints are not sufficient, since they are attached to classes and constrain the structure of the instances of those classes. In a multi-level model, constraints can be attached to a meta-class and therefore apply to its instances (which are classes) and the instances of its instances (which may be classes or ground instances).

There are various approaches to multi-level modelling (e.g., [2], [10], [11], [1], [12]. However, in there is no agreement on a unified multi-level object constraint language (MOCL). In this paper, we present a MOCL that could serve this purpose. The MOCL has been implemented in the XModeler [8,6,7] as an extension to its XCore meta-kernel. The paper is structured as follows. A brief overview of multi-level modelling in section 2 serves to illustrate requirements for corresponding language architectures, and demonstrates the need for MOCLs. MLM places requirements on a modelling language architecture - section 3 describes a particular meta-architecture called XCore that we believe is ideally suited to this purpose. A MOCL is defined as a language extension to XCore in section 4. We demonstrate the utility of MOCL for MLM in section 5 where constraints are attached to meta-types in order to express domain-specific semantics that range over multiple type-levels.

2 Multi-Level Modelling in a Nutshell

While the various approaches to multi-level modelling differ with respect to particulars, they all share a few essential properties. First, they allow for an arbitrary number of classification levels. This feature is important because it supports abstraction at the type-level and as a consequence provides a mechanism for the introduction of new language features. A DSML language designer needs this,

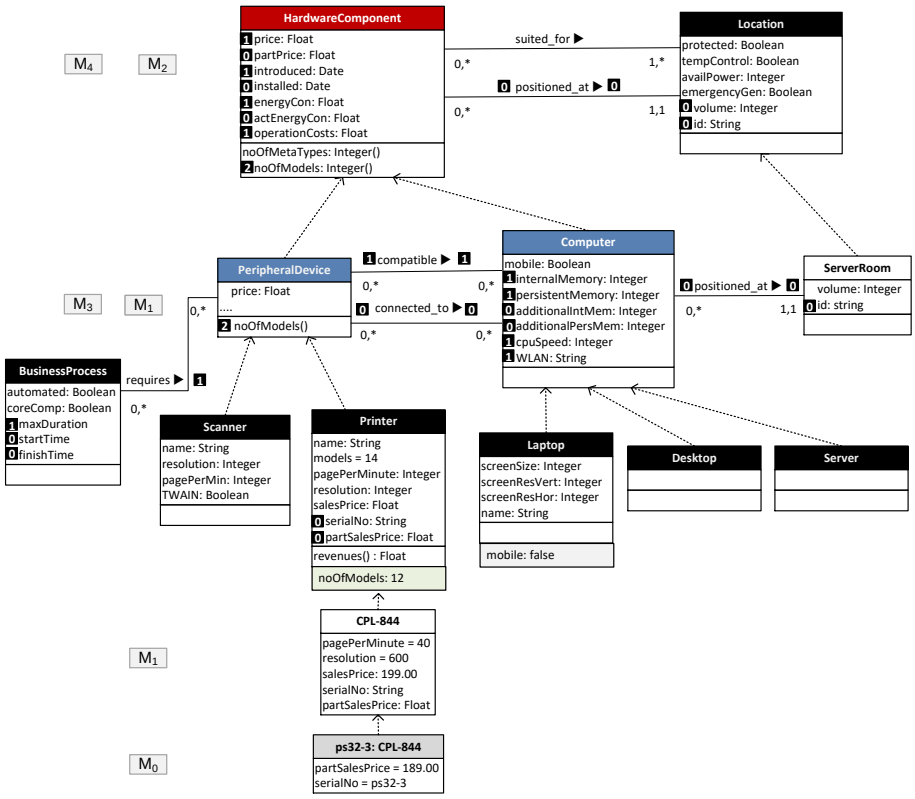

Fig. 2. Example of Multi-Level Model

but we would argue that a conventional modeller would find this attractive too in order to introduce new meta-concepts in a structured way.

Second, every class, no matter on what level, is an object at the same time. This is important if technologies are to be reusable with respect to models that have an arbitrary number of type-levels. In addition, new type-levels imply that new types are defined including the ability to specify additional properties at the type-level. As we shall see, it is natural to introduce new properties for types and to treat them as objects by accessing and updating the property values.

Third, MLM approaches support deferred or deep instantiation, which means that attributes of a class do not have to be instantiated with the direct instances of that class, but only later in instances of instances. The example class diagram in figure 2 illustrates the construction of a multi-level model. The class `HardwareComponent` is located on `M4`. Its specification should include everything we know about its instances, and instances of those instances *etc.* in order to achieve reuse and to prevent redundant specification on lower levels. For example, we know that every hardware component has a sales price which is defined on the level of a particular product type, that is, on `M1`.

The intended instantiation level of a property (attribute, operation, association) is represented as an integer printed white on a black rectangle next to the property. We also know that any hardware component (represented on `M3`) may be suited for a certain type of location. In addition, it is obvious that a particular exemplar of a hardware component is located at a particular location, both represented by objects on `M0`. As the example shows, associations are possible between classes on different levels.

The diagram shows that there is need for constraints that span various levels. For example, the definition of an attribute like `price` that is supposed to be instantiated only a few instantiation levels further down the instantiation chain requires a constraint that applies to all affected instances. The deferred instantiation of the association `positioned_at` requires a constraint that is dynamically adapted on every instantiation level up to `M0`. At the level where it is defined, the class attached to the association end of `positioned_at` typed `Computer` is unknown since at M0 the corresponding link must be an instance of an instance of `Computer` (for example a lap-top or a desktop). With every instantiation, the range of possible choices is narrowed. Hence, a MOCL should allow for specifying constraints that applies to the entire range of a meta-class, that is, all its instances and instances of its instances. To this end, the MOCL must be based on a multi-level language architecture that enables multiple classification levels.

More information about MLM, including features such as *deep*, *struct*, *potency*, *clabjects* and *formalisation* can be found in [5,13].

3 Foundations: XCore with Single Level Constraints

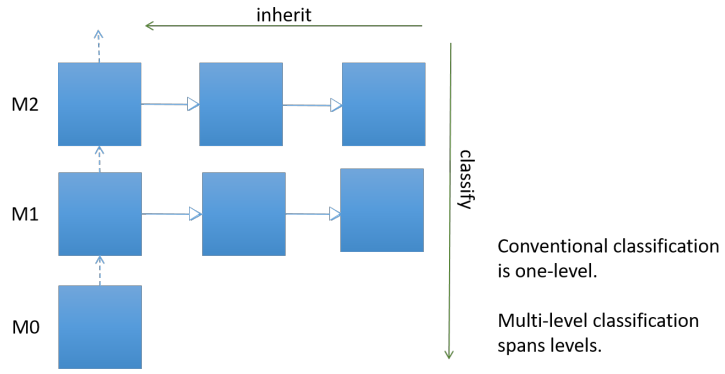

Fig. 3. Abstraction: Two Dimensions

Model based engineering involves hierarchy in two dimensions: inheritance and types, as shown in figure 3. Traditional approaches tend to promote the former and ignore the latter. For example, when defining the semantics of a

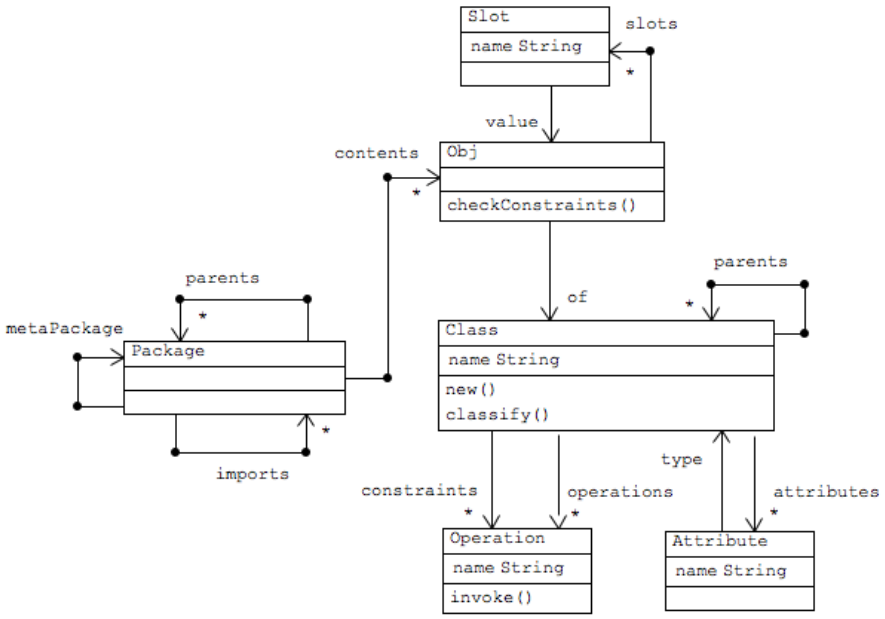

Fig. 4. XCore

model, OCL can be used to attach model-specific constraints. There is limited support for constraints attached to meta-classes[3] which motivates the need for a suitable kernel language architecture.

Figure 4 shows XCore. All classes (except `Obj`) inherit from `Obj` and all objects refer to their class (which is also an object) via '`of`'. Models are instances of `Package`. This provides a uniform type-agnostic representation: by default models can contain objects, classes, meta-classes and so on without any restrictions. A package refers to its `metaPackage` that contains the types for the elements in the package. The default meta-package for any package is `XCore`.

The basic definition of constraints in XCore works like OCL: a constraint is attached to a class and applies to direct instances of the class. XModeler provides a convenient syntax construct for a *constraint* that abstracts the details of the underlying operation:

```
context C
  @Constraint ConstraintName
     booleanExpression
  end
```

The candidate object is referenced using `self` in the boolean expression. This can be termed *single-level constraints*.

[3] Note that by *model-based language engineering* we do not mean the use of models at M1 to denote languages. This is used, for example, in the definition of UML.

```
context Obj
  @Operation checkConstraints():Boolean
    self.of().classify(self)
  end
context Class
  @Operation classify(o:Obj):Boolean
    classifier.allConstraints()→forAll(c | c(o))
  end
context Obj
  @Constraint AllConstraintsSatisfied
    self.checkConstraints()
  end
```

Fig. 5. Object Classification

Within XCore, every object `o` has a class (equivalently its *type*) that is returned as `o.of()` and therefore `o` is an instance of `o.of()`. An object `o` is also an instance of any class that can be reached by traversing `parents` links from `o.of()`. Every object also has a meta-class (equivalently its *meta-type*) that is returned as `o.of().of()`. The defining feature of a class is that it is an instance of a meta-class; the defining feature of a meta-class is that it inherits from `Class`. Meta-circularity follows from `Class.of() = Class`.

The number of type-levels between an object `o` and a class `c` is the number of uses of `of()` that need to be applied to `o` to reach `c`. In general it is not meaningful to talk of an XCore model being at level n since a type-level is contingent on the object and type in question, and an XCore package can contain mixtures of objects, classes and meta-classes. Therefore,, in general we refer to an object being defined at level M_i when there are i type-levels between the object and `Class`. If it is required, strict type-levels can be imposed on package structures by requiring that a package is linked to a meta-package as follows:

```
context Package
  @Constraint strictness
    metaPackage.contents = contents→collect(o | o.of())
  end
```

XCore contains operations that can be called with arguments and which return results. Boolean valued operations (without side-effects) are predicates. Attaching predicates to classes implements constraints (in the sense of OCL). Since an object can refer to its class, we can write a constraint on `Obj` that requires every object to satisfy the constraints defined by its class. Since class is an object we can require that all models are correctly formed and also define how constraints are applied to instances. Since the architecture of XCore is reflexive, the definition of constraints allows us to bootstrap an arbitrarily extensible language architecture.

Figure 5 shows the meta-circular definition of object classification defined in XCore. Every object (that is everything in the system) can be asked to check its constraints. To do so, the receiver navigates to its classifier using `self.of()` and then asked the resulting class to classify `self`. A class uses `classify` to check whether the supplied object meets its classification conditions as defined by its

constraints. The operation `allConstraints` returns a list of constraints that are either defined by a class or one of its parents. A constraint is just an operation and so it can be applied to a candidate object to return `true` or `false` to indicate whether or not it is satisfied. The language construct `Constraint` is used to add an operation to the constraints list of a class and abstracts away from the arguments common to all constraints. The constraint `AllConstraintsSatisfied` is added to `Obj` and requires that all objects are correctly classified by their types.

The constraints that are maintained by a class are all *single-level* constraints and are consistent with the constraints represented by OCL in UML. This means that they apply to the direct instances of the class or one of its parents. With respect to figure 3, single-level constraints can only reach down from M_n to M_{n-1} in the vertical direction. It is not possible for single-level constraints defined at M_2 to refer to objects at level M_0. We argue, however, that this is unreasonable since single-level constraints are able to cross as many horizontal levels as is required (due to inheritance). This restriction must be removed to achieve an MOCL; an approach is described in the next section.

Model-based abstraction is supported in two dimensions: *inheritance* and *types*. Inheritance-based abstraction does not name the abstraction levels, for example we do not talk of `Animal` being one inheritance-level removed from `Dog`. Conversely, type-levels are often numbered with `0` being the lowest, or *ground*, level that categorizes objects that are not types, and with n being types whose instances are at level `n-1`.

The semantics of modelling languages describe how inheritance works (both in terms of classification and reuse), and how the relationships between elements at different type-levels work. However, whilst the definition of inheritance applies no matter how deep the inheritance hierarchy gets, the definition of type usually just applies between level `n` and `n-1`.

The limitation on type-level semantics places a restriction on how effective type relationships are with regard to achieving abstraction in models. Meta-types (at level `2` or above) arise naturally in a wide range of application areas where linguistic terms are coined and their application is defined by regulation.

4 XCore with Multi-Level Constraints

Single-level constraints are sufficient for single-level modelling, however this results in a number of compromises since single-level models cannot use type-level abstraction which is fundamental to a language-driven approach to software engineering. Figure 5 defines the XCore meta-circular classification scheme involving constraints. New types can be added to XCore by extending `Class`, for example if `MC` extends `Class` and adds an attribute `a` and an operation `o` then all instances of `MC` are classes that have all the expected slots and behaviours of a class, but also have a slot named `a` that can be manipulated via the operation `o`. Furthermore, XCore defines a meta-circular language defined in terms of operations such as `classify` and `new`. Since XCore defines the semantics of extension, then new lan-

guages can be defined by extending classes such as `Class`. Multi-level constraints are defined using this approach.

```
@Package MultiConstraints extends XCore
  @Class ClassWMC extends Class
    @Attribute multiConstraints : [Operation] (+,-) end
    @Operation classify(o:Obj):Boolean
      super(o) and self.metaClassify(o,0)
    end
    @Operation metaClassify(o:Obj,level:Integer):Boolean
      let C:[Operation] = self.allMultiConstraints()
      in C→forAll(c | c(o,level)) and self.guardedMetaClassifyUp(o,level)
      end
    end
    @Operation allMultiConstraints():[Operation]
      parents→iterate(p C = multiConstraints |
        C + if p.isKindOf(ClassWMC) then p.allMultiConstraints() else [] end)
    end
    @Operation guardedMetaClassifyUp(candidate:Obj,level:Integer):Boolean
      if self.of().isKindOf(ClassWMC)
      then self.of().metaClassify(candidate,level+1)
      else true
      end
    end
  end
end
```

Fig. 6. Multi-Level Constraints

A class `ClassWMC` (*class with multi-level constraints*) is defined in figure 6. It is a meta-class (because it extends `Class` and introduces a new attribute `multiConstraints` whose value in any given instance is a list of operations. When an instance is asked to classify an object it uses `metaClassify` to run over the meta-constraints at each level. Unlike normal constraint checking (such as OCL), the meta-constraint is supplied with the level number that can be used to determine whether the constraint applies to the supplied candidate object.

Note that, since `ClassWMC` is an extension to `Class` we must be careful to guard against a reference to the slot `multiConstraints` when the a meta-classifier is not an instance of `ClassWMC`. This guard is implemented by `guardedMetaClassifyUp` which checks if the classifier at level `n+1` is a `ClassWMC` before calling `metaClassify`.

A construct is provided for multi-level constraints abstracting away from two arguments: the candidate and the level number. In addition, a multi-level constraint can be specified with a specific level number in which case the constraint only applies to those instances `l` type-levels removed from the class:

```
context C
  @MultiConstraint ConstraintName(l)
    booleanExpression
  end
```

The candidate object is referenced using `self` and the level as `level` in the boolean expression. A single-level constraint is simply sugar for a multi-level

constraint with level number `0` showing that multi-level constraints are the more general language construct.

5 Case Study

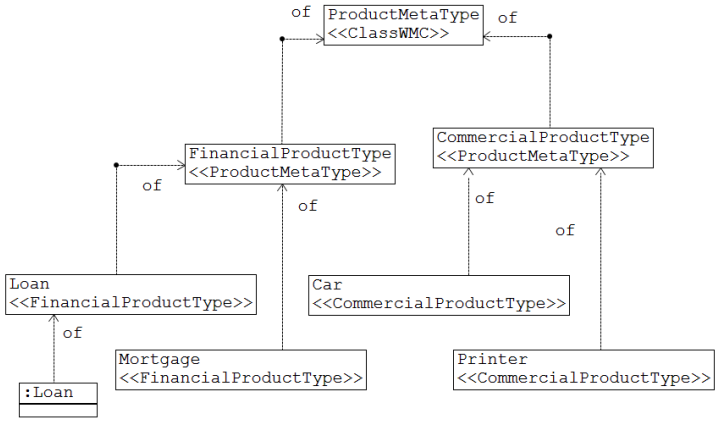

Fig. 7. Meta Levels

Figure 7 provides a simple example of a model involving multiple levels. The application requires a model of products that will be used as the basis of an E-Commerce platform. New customers can use an existing product type or can create their own, therefore we require a meta-type `ProductMetaType` whose instances are product types. Two product types are shown: `CommercialProductType` and `FinancialProductType`. The class `Printer` is an example of a commercial product type and `Loan` is an example of a financial product type. A single instance of `Loan` is shown.

Using traditional level numbering, the single instance of `Loan` is at level `0` (or the *ground* level), `Loan`, `Mortgage`, `Car` and `Printer` are all at level `1`, `FinancialProductType` and `CommercialProductType` are both at level `2`, and `Product MetaType` is at level `3`. Since everything is an instance of *something*, there must be a level `4` which contains the definition of the classifier for `ProductMetaType`; this is not shown, but is `ClassWMC` as defined below.

The description given above implies that, for any given type at level `n`, there is a classifying type at level `n+1`. In order for this to hold, it will be the case that there is a classifier (called `Class`) that classifies itself. in this way there is no artificial limit to the level of classification starting with ground objects. Furthermore, as defined in section 3, the level associated with the classifier `Class` *classifies itself and any level below it* which produces a kernel level that is both arbitrarily extensible and self-classifying.

Consider what we may know about the domain when constructing this type hierarchy:

ProductMetaType: Some product types are regulated which means that they will need to record whether or not the type has been checked by the regulatory authority. Note that this is not a property of a particular ground object such as a loan, but is a property of the product type `Loan` itself. Even at the meta-type-level, it is known that all product objects have an identifier that is unique to the type of product. Furthermore, it is known that any product type manages a collection of the product instances so that product analysis can be performed across the complete collection of products (for example to work out total profit).

CommercialProductType: Some product types are bought and sold as individual items and therefore are instances of the type `CommercialProductType`. Such types define a list-price that is common to all products of that type each of which has its own sales-price which must be no greater than the list-price.

FinancialProductType: We know that all financial products are regulated and much be checked by the appropriate authority before they can be bought and sold.

Printer, Car: Both `Printer` and `Car` are commercial product types that define particular list prices. A car has a registration number.

FinanceLoan, Mortgage: Both `FinanceLoan` and `Mortgage` are financial product types that have interest rates for repayment. They also define whether or not they have been checked by the appropriate regulatory body.

The description above shows four different type-levels (and an implicit fifth which is the type of `ProductMetaType`). Each class defines properties and behaviour that must hold for objects at lower levels and the information is placed at the highest possible level. For example, when defining `ProductMetaType` it is known that all products must have a unique identifier, even though the `id` property will be attached to objects three type-levels below.

Figure 8 shows a class diagram for the product language that uses `ClasswMC` (note that we do not show the `of` links which are shown in figure 7). The model shows how the attributes defined in meta-classes become slots in classes, for example the `listPrice` attributes defined by `CommercialProductType` becomes a slot with the same name in `Printer` and `Car`.

The semantics of the product language is defined by meta-constraints. The rest of this section lists the meta-constraints and describes their effect on the model. Much of the semantics for products is known when `ProductMetaType` is defined. We know that any regulated product must be checked by an appropriate authority:

```
context MetaProductType
  @MultiConstraint Reglation Checked(1)
    self.isKindOf(RegulatedProductType) implies self.checked
  end
```

The meta-constraint `Regulation Checked` applies only to candidates whose type-level is `1`, *i.e.* where `candidate.of().of() = MetaProductType`. With respect to

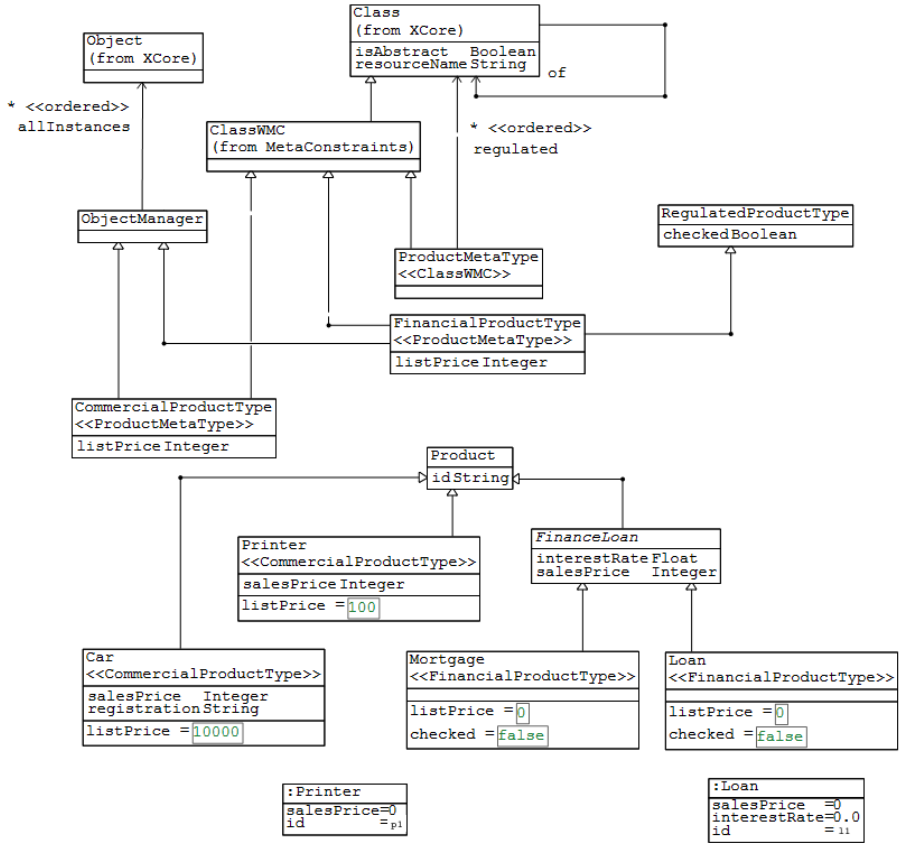

Fig. 8. Product Classes Using Multi-Level Constraints

figure 8, classes `Car` and `Mortgage` are both examples of classes at level 1 with respect to this meta-constraint and where `Mortgate` is a `RegulatedProduct` but `Car` is not. The meta-constraint requires `Mortgage.checked` to be true.

At level 3 it is also known that products should be managed by their product type so that all available products of that type can be analyzed. To achieve this we can set up a class called `ObjectManager` that is used as a *mixin* to product types and which adds a slot `allInstances` that will hold all the instances of the type, and add meta-constraints that require product types to manage products in the required way:

```
@Class ObjectManager
  @Attribute allInstances:[Obj] (+) end
  @Constraint checkAllInstances
    allInstances→forAll(o | o.isKindOf(self))
  end
end
context MetaProductType
  @MultiConstraint Inherit Object Manager(0)
    self.inheritsFrom(ObjectManager)
  end
  @MultiConstraint Recorded Instances(2)
    self.of().allInstances.includes(self)
  end
```

The class `ObjectManager` has a constraint that requires the value of `allInstances` to be a list of objects that are all instances of the class to which `ObjectManager` is added as a parent. Note that this is defined as a constraint since it does not care what the level number is.

The meta-constraint `Inherit Object Manager` requires that all meta-product types inherit from `ObjectManager` and therefore all product-types will have the slot `allInstances`. The meta-constraint `recorded Instances` requires that the products are recorded in the slot.

A key feature of products is that they must have an identifier that is unique among products of that particular type. Meta-constraints can be used to require a mixin `Product` to be a parent of all particular product types (`Is A Product`), and for the identifier to be unique (`Unique Identifiers`):

```
@Class Product
  @Attribute id:String end
end
context MetaProductType
  @MultiConstraint Is A Product(2)
    self.isKindOf(Product)
  end
  @MultiConstraint Unique Identifiers(2)
    self.of().allInstances→forAll(p1 |
      self.of().allInstances→forAll(p2 |
        p1.id = p2.id implies p1 = p2))
  end
  @MultiConstraint Is A Product(2)
    self.isKindOf(Product)
  end
```

A commercial product type has a further constraint that requires the sales price to be less than the list price:

```
context CommercialProductType
  @MultiConstraint Sales Price <= List Price(1)
    self.salesPrice <= self.of().listPrice
  end
```

XModeler provides a tool that records the results of checking constraints and displays them as a tree where each node is labelled with the name of the constraint and is coloured green of the constraint was satisfied and red if it failed. Figure 9 shows the result of performing various constraint checks on elements of the product model. Figure 9(a) shows a loan object with id `l1`, interest rate `0.0` and sales price `10`. All of the multi-level constraints are satisfied by the loan. Figure 9(b) shows a printer that does not satisfy the multi-level constraints because it is a second printer created with the id `p1` and the sales price `101` is

(a) Loan

(b) Printer

(c) Mortgage

(d) Commercial Product

Fig. 9. Constraint Checking in XModeler

greater than the list price `100` associated with the product-class `Printer`. Figure 9(c) shows that the financial product-type `Mortgage` fails because it is a regulated type that has not been checked. Figure 9(d) shows that the commercial product-type satisfies all constraints.

6 Related Work

There are various language architectures that support multi-level modelling. The majority follow an object-oriented approach while a few are based on logic [10] or set theory [12]. None of the language architectures address unlimited multi-level constraints. Gogolla *et al.* [9] describe a 3-level language architecture where OCL can be applied to a model as an instance of a meta-model and where a language extension is added to OCL do designate whether it is being applied at the type or instance level. The XCore language architecture does not require such an extension since everything is an object and the kernel is self-describing. FOML [3] can be used in conjunction with multi-level models and, like the approach described in this paper is not limited in terms of levels; however, unlike XCore it is based on an exogeneous language (Prolog) which means that the constraints are not integrated with the modelling framework.

7 Analysis and Conclusion

We have presented a motivation for multi-level constraints to support the construction of multi-level models, and shown how these constraints can be defined within a reflexive kernel language, XCore. Our kernel language is both precise and self-describing. Other approaches to defining the semantics of MLM use formal logic, for example [5], as an external language. A key benefit of our approach is that the definition of MLM is extensible within the provided language framework as shown in this paper. We have evaluated our proposal by implementing a multi-level constraint language in XCore and applying it to a multi-level model based on products and product-types.

We have claimed that type-level should be as prominent as inheritance in modelling languages as a basis for supporting abstraction, and have shown a novel mechanism that supports it. There is work still to be done, for example just like there are multiple approaches to inheritance, there are different ways of implementing MLM and therefore MOCL. Adding multiple type-levels increases the complexity of a language and there is a need for tools to support the use of these techniques. Unlike inheritance, MLM implies a potential change in implementation technology (for example requiring that types exist at run-time) which can come with a cost that may be undesirable.

References

1. Atkinson, C., Gutheil, M., Kennel, B.: A flexible infrastructure for multilevel language engineering. IEEE Trans. Software Eng. 35(6), 742–755 (2009), `http://dblp.uni-trier.de/db/journals/tse/tse35.html#AtkinsonGK09`
2. Atkinson, C., Kühne, T.: The essense of multilevel metamodeling. In: Gorgolla, M., Kobryn, C. (eds.) UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, vol. 2185, pp. 19–33. Springer, Berlin and London, New York (2001)
3. Balaban, M., Khitron, I., Kifer, M.: Multilevel modeling and reasoning with foml. In: Software Science, Technology and Engineering (SWSTE), 2016 IEEE International Conference on. pp. 61–70. IEEE (2016)
4. Bunge, M.: Treatise on Basic Philosophy: Volume 3: Ontology I: The Furniture of the World. Reidel, Dordrecht (1977)
5. Carvalho, V.A., Almeida, J.P.A.: Toward a well-founded theory for multi-level conceptual modeling. Software & Systems Modeling 17(1), 205–231 (2018)
6. Clark, T., Sammut, P., Willans, J.S.: Applied metamodelling: A foundation for language driven development (third edition). CoRR abs/1505.00149 (2015), `http://arxiv.org/abs/1505.00149`
7. Clark, T., Sammut, P., Willans, J.S.: Super-languages: Developing languages and applications with XMF (second edition). CoRR abs/1506.03363 (2015), `http://arxiv.org/abs/1506.03363`
8. Clark, T., Willans, J.: Software language engineering with xmf and xmodeler. In: Computational Linguistics: Concepts, Methodologies, Tools, and Applications, pp. 866–896. IGI Global (2014)
9. Doan, K.H., Gogolla, M.: Extending a uml and ocl tool for meta-modeling: Applications towards model quality assessment. Modellierung 2018 (2018)

10. Jeusfeld, M.A.: Metamodeling and method engineering with conceptbase. In: Jeusfeld, M.A., Jarke, M., Mylopoulos, J. (eds.) Metamodeling for Method Engineering, pp. 89–168. MIT Press, Cambridge (2009)
11. Kühne, T., Schreiber, D.: Can programming be liberated from the two-level style: multi-level programming with deepjava. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Steele, G.L. (eds.) Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA '07). ACM SIGPLAN notices, vol. 42,10, pp. 229–244. ACM Press, New York (2007), http://atlas.tk.informatik.tu-darmstadt.de/Publications/2007/p229-kuehne.pdf
12. Neumayr, B., Grün, K., Schrefl, M.: Multi-level domain modeling with m-objects and m-relationships. In: Kirchberg, M., Link, S. (eds.) Conceptual Modelling 2009. pp. 107–116. Australian Computer Society (2009), http://crpit.com/confpapers/CRPITV96Neumayr.pdf
13. Rossini, A., Lara, J., Guerra, E., Rutle, A., Wolter, U.: A formalisation of deep metamodelling. Form. Asp. Comput. 26(6), 1115–1152 (2014)