Opportunities and Challenges for Deep Constraint Languages

Colin Atkinson[1], Ralph Gerbig[1] and Thomas Kühne[2]

[1] University of Mannheim
{atkinson, gerbig}@informatik.uni-mannheim.de
[2] Victoria University of Wellington
thomas.kühne@ecs.vuw.ac.nz

Abstract. Structural models are often augmented with additional well-formedness constraints to rule out unwanted configurations of instances. These constraints are usually written in dedicated constraint languages specifically tailored to the conceptual framework of the host modeling language, the most well-known example being the OCL constraint language for the UML. Many multi-level modeling languages, however, have no such associated constraint language. Simply adopting the OCL for such multi-level languages is not a complete strategy, though, as the OCL was designed to support the UML's two-level class/instance dichotomy, i.e., it can only define constraints which restrict the properties of the immediate instances of classes, but not beyond. The OCL would consequently not be able to support the definition of deep constraints that target remote or even multiple classification levels. In fact, no existing constraint language can address the full range of concerns that may occur in deep modeling using the Orthogonal Classification Architecture (OCA) as an infrastructure. In this paper we consider what these concerns might be and discuss the syntactical and pragmatic issues involved in providing full support for them in deep modeling environments.

Keywords: OCL; well-formedness; deep modeling; OCA

1 Introduction

Although structural modeling languages such as UML class diagrams can sometimes be used on their own for simple tasks, for more precise modeling they usually need to be supported by accompanying constraint languages. The most well-known language for this purpose is OCL, but today quite a number of other constraint languages are available for use with the UML and other structural modeling languages. Examples include AspectOCL [1], CdmCL [2], EVL [12] and MOCQL [16].

The OCL is commonly used to augment class diagrams with further well-formedness constraints. Constraints can be applied to other UML diagram types but these are not addressed in this paper. The basic role of constraints is to add additional requirements that (token-)models at the instance level must obey in

order to be considered instances of their (type-)model at the class level. Since the sets of valid token models are sometimes referred to as the semantics of the (structural) type model, constraints can be regarded as improving the precision of the semantics of the type model.

Since mainstream constraint languages in use today were designed to support today's mainstream structural modeling languages and tools, they are based on the same underlying "two-level" modeling paradigm. In particular, this means that the domain entities considered by the users are regarded as occupying one of two levels – a type level or an instance level. This goes back to the idea that all concepts are either universals or individuals but not both at the same time. As a result, today's constraint languages are often designed in the context of a number of assumptions:

1. constraints are attached to types and are evaluated for instances of these types.
2. constraints are implicitly universally-quantified, i.e., use a "*forAll*"-semantics that applies them all instances of the type they are associated with.
3. there is only one instance level to control, i.e., deep control beyond more than one metalevel boundary is not considered.

These assumptions are not an optimal fit for deep modeling. Constraint languages that are designed to complement deep models, i.e., models with an instantiation depth higher than one, therefore have to be based on a different set of assumptions with implications on what information is necessary to fully describe a constraint in a multi-level context. Designing an approach for constraints that incorporates the extended challenges of deep modeling in an optimal way is a non-trivial task, and requires notational and pragmatic trade-offs that are influenced by many factors. The goal of this paper is not to resolve these trade-offs, but to identify the opportunities and challenges present in the expression of deep constraints and therefore the design of Deep Constraint Languages (DCLs).

As well as supporting well-formedness constraints on structural models, there are a variety of other applications for constraint languages or extensions thereof. These range from queries to transformations, however, for space reasons, in this paper we focus on well-formedness constraints. Most of the ideas apply to similar languages as well, though, for example for deep transformations [5].

Summarising, the goal of this paper is to reevaluate the role of constraints for deep modeling, identify different kinds of constraints, and establish terminology for referring to them. The rest of the paper is structured as follows: In the next section we give a brief overview of deep modeling and the principles it is based on as well as discuss the main implementation choices that can be used to implement deep models. Then, in Section 3 we introduce the different kinds of constraints that can be defined given the two orthogonal classification dimensions that underpin deep modeling. Section 4 then does the same given the multiple ontological levels that can be defined in a deep model. Finally, section 5 concludes with some closing remarks.

2 Deep Modeling

Deep modeling is built around the Orthogonal Classification Architecture (OCA) which provides an alternative way of organizing models compared to the traditional linear modeling stack that underpins mainstream modeling technologies such as UML and EMF. Figure 1 below shows the most widely used variant of the OCA which has the linguistic dimension occupied by two linguistic levels – the modeling language definition in L_1, the ontological model content in L_0 and the real world (W). Although variants with more linguistic levels are conceivable, in this paper we assume the OCA only considers these levels. It can be observed that the ontological (O-levels) and linguistic (L-levels) classification dimensions are orthogonal to each other. This alignment of levels gives the name to the orthogonal classification architecture.

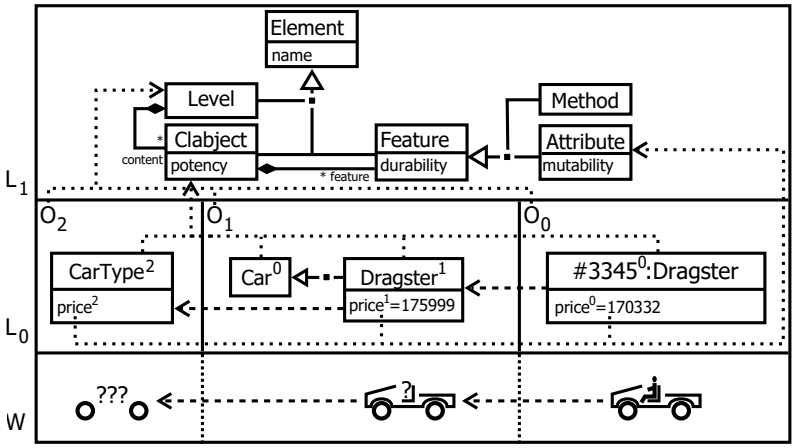


Fig. 1. Three level OCA.

In Figure 1 the linguisistic dimension contains Level L_1, which is often referred to as the "linguistic type model" or (less precisely) "metamodel". This linguistic level defines the basic concepts of *Level*, *Clabject* and *potency* etc. Level L_0 contains the ontological content defined by the end user. In this level the clabjects are organized into multiple ontological classification relationships depending on whether they model objects, types, types of types, etc. in the domain of interest. Liniguistic classification is indicated through dotted vertical classification arrows while ontological classification is indicated through dashed horizontal classification arrows. O_0 contains pure objects (i.e. clabjects with no type facet), O_1 contains normal types (i.e. clabjects that have both a type and an instance facet) and O_2, in this example, contains types (of types) at the top ontological level (i.e. clabjects which have no instance facet).

A second core idea underpinning deep modeling is that classes and objects, which are modeled separately in traditional modeling approaches, are integrated

into a single, unified concept known as "clabject". In general, clabjects are classes and objects at the same time, and thus simultaneously possess both type and instance facets, e.g., consider *Dragster* at the middle level, O_1, which is an instance of *CarType* and type for *#3345* at the same time. The third core idea underpinning deep modeling is the notion of deep instantiation which uses the concept of potency, attached as a superscript to the name of each clabject and their features as seen in Figure 1. This concept limits the depth of the instantiation tree of a clabject thus defining the degree to which a clabject can be regarded as a type. In the example, *CarType* with potency two can be instantiated on the following two levels. It is then instantiated with *Dragster* which can be instantiated one level further below and *#3345*, an instance of *Dragster* which cannot be instantiated further as indicated by its potency zero value.

The features of deep modeling that present the biggest opportunities and challenges from the perspective of defining constraints on deep models beyond what is required for traditional "two-level" modeling are:

1. the two distinct, orthogonal classification dimensions,
2. dual facets of model elements (i.e. the existence of linguistic and ontological attributes), and
3. an unbounded number of ontological classification levels.

2.1 Realization Strategies

Modeling environments, including deep modeling environments, are typically built using traditional "two level" technology. It is possible to support multiple, logical modeling levels on top of such two-level physical architectures in two ways. The first is by supporting transformations between chains of two-level models, each capturing a different window on the underlying multi-level model (referred to as the "cascading" style in [6]). The second is by mapping the linguistic metamodel(L_2) to the type level of the implementation platform, and the domain content (L_1) to the instance level of the implementation platform, with some of the relationships in the latter level being regarded as classification relationships. For example, to support the second approach on a Java platform, the elements of the linguistic metamodel (e.g. level, clabject etc.) would be mapped to Java classes, and the L_1 level content would be represented as instances in the JVM. While many commercial tools use the first approach, the second approach is used in the majority of academic tools and ultimately provides the simplest and most flexible way of implementing deep modeling environments. We assume the latter approach in the remainder of this paper, therefore.

In general, approach (b) can be applied in one of two ways. The simplest way is for a modeler to simply model the linguistic metamodel as a class diagram in some existing, "host" modeling environment and then to apply the tenets of deep modeling itself at the instance level using certain well-known patterns [8]. If the host environment has a constraint language (e.g. OCL) this can be used to express limited kinds of constraints over the deep model at the instance level. This approach is taken by Gogolla et al. [8], for example, who represents various

deep modeling scenarios within the two-level USE tool (i.e. by modeling L_1 as a class diagram and L_0 as an object diagram) and uses standard OCL to express constraints over the L_0 content. We refer to a constraint language used in such a way as a Standard Constraint Language (SCL).

In contrast, a DCL is a constraint language which includes extra support (explicitly implemented as an extension to the host environment) for the concepts embodied by deep modeling. In other words, a DCL provides additional features for expressing constraints on deep modeling content which are not available in an SCL. As with all DSLs, this support can take the form of additional library functionality (cf. [8]), in which case it is an internal DSL, or it can be provided along with additional syntax, in which case it is an external DSL) [7].

In the following sections we investigate, in turn, the consequences and opportunities resulting from the key features of deep modeling identified previously. In each case we will identify different kinds of constraints that may occur in deep models, show examples of these constraint kinds on a small running example and discuss possible syntactic alternatives for expressing the constraints. We show examples in a suggested syntax which includes notational ideas from three existing constraint languages — the OCL, which plays the role of an SCL in this context, MetaDepth [13] which is an external DSL built on top of the Epsilon Object Language [11], and Deep OCL [10] from Melanee [3], which is an external DSL built on top of the Eclipse MDT OCL.

3 Linguistic, Ontological and Hybrid Constraints

The first important characteristic of deep modeling is that there are two distinct dimensions across which constraints can operate – the linguistic and the ontological dimensions. In principle, modelers may wish to reference the ontological and/or the linguistic dimension when defining constraints. This gives rise to three kinds of constraints – constraints only referencing the ontological dimension, constraints only referencing the linguistic dimension, and hybrid constraints.

3.1 Linguistic Constraints

Linguistic constraints reference concepts in terms of linguistic types and are therefore independent of any ontological types. An example application for them is the definition of the classification semantics of deep modeling. Constraint 1 shows how a linguistic constraint on Figure 2 can be used to define the basic rules of deep instantiation – namely that the potency of an instance of a clabject must be one lower than the potency of that clabject. Constraint 1 is a standard OCL constraint, just applied within the linguistic dimension.

Constraint 1 *The value of the potency of every clabject must be one lower than that of its direct type*

context *Clabject*
getDirectTypes() implies forAll(t | t.potency = potency+1)

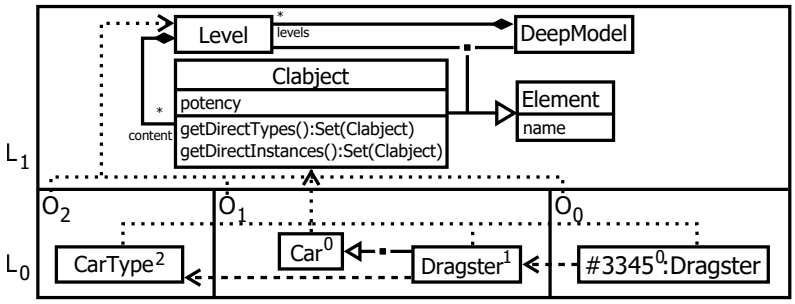


Fig. 2. Linguistic constraints example.

Constraints which enforce certain modeling styles are also possible. For example it is possible to require that clabject potency values must always match level values (as in [13]) or to limit the number of levels available in a deep model (Constraint 3).

Constraint 2 *The value of the potency and level of every clabject must be equal*

context *Clabject*
self.potency = self.level

Constraint 3 *The number of levels in a deep model must be 3*

context *DeepModel*
self.levels implies size = 3

These constraints are useful in scenarios where the number of ontological levels has to be fixed, e.g. when model execution software is written against a certain level of a deep-model. Like Constraint 1, both Constraint 2 and Constraint 3 are standard OCL constraints whose context is the linguistic meta-model.

3.2 Ontological Constraints

Ontological constraints operate within level L_0 to express well-formedness conditions on the content of ontological levels. These well-formedness conditions include the kind of constraints that end users (i.e. modellers) typically write in conventional modeling environments to constrain the properties of domain instances based on their domain types.

An example of a traditional constraint in the context of Figure 3 is the constraint for a second-hand dealer that the default used *price* of a *ProductType* (e.g., *Lorry*) has to be lower than the respective recommended retail price (*RRP*). Using an OCL-like syntax, a DCL should allow this constraint to be expressed in a way similar to Constraint 4. This constraint is defined "on" *ProductType* and ensures that all ontological instances of *ProductType*, here *Lorry* and *Dragster*, have an *RRP* that is higher than their *price*.

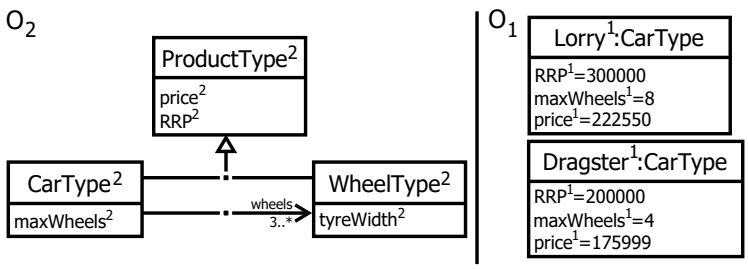


Fig. 3. Ontological constraints example.

Constraint 4 *The value of a ProductType's price attribute has to be smaller than or equal to its RRP attribute*

origin*(1) ProductType*
self.price <= self.RRP

An important notational difference in Constraint 4 is that we use the term "origin" rather than "context" to specify to which element the constraint is attached in order to avoid the traditional meaning associated with the term "context" in OCL. In OCL, the class to which a constraint is attached does not coincide with the evaluation context for the constraint, as OCL implicitly assumes universal quantification of the constraint over all instances of the class. We believe, in order to support more flexibility, it is worthwhile not always making this assumption of implicit universal quantification over instances at the level below. Therefore, in the remainder of the paper we express constraints in the following form —

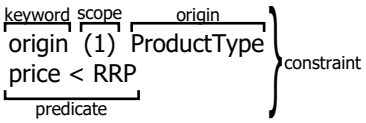


Fig. 4. General structure of a constraint.

where the clabject appearing in the top line after the keyword "origin" is regarded as the definitional anchor for the constraint, whereas the "scope" of the constraint (i.e., to what levels it applies to) is specified by the given range (e.g. *(1)*) as further explained below. The constraint body, defined underneath the header, specifies the predicate that must evaluate to true for all elements of type "origin" within the scope.

3.3 Hybrid Constraints

The constraints discussed in the previous section either operated purely across the linguistic dimension or purely across the ontological dimension. In some situations, however, there is a need to mix the linguistic and ontological dimensions. Such a scenario is shown in Constraint 5 on Figure 5 which ensures that each concrete car must have a price greater than zero and that each car type must have a default price greater than zero. This is achieved using a hybrid constraint whose origin is *CarType*. This constraint uses the linguistic dimension to apply the constraint to all *CarType* instances at levels 1 and 0, and the ontological dimension to ensure that the price of these model elements is greater than zero. Note that Constraint 5 could be expressed more concisely using our proposed scoping mechanism (cf. Section 4.2) and hence demonstrates the latter's utility; Constraints 8 and 9 are more natural examples of hybrid constraints.

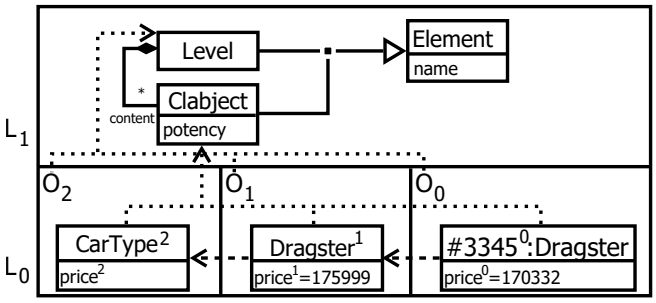


Fig. 5. Hybrid constraints example.

Constraint 5 *The (default) prices for instances of CarType and their instances must be greater than zero*

***origin**(1..2) CarType*
(a) self.level < 2 implies self.price > 0
(b) self._l_.level < 2 implies self._o_.price > 0
(c) self._l_.level < 2 implies self.price > 0
(d) self.^level < 2 implies self.price > 0

The example suggests four different possible notations to define the hybrid constraint. The first notation (a) does not make a syntactic distinction between the dimension in which a called attribute, method, etc. is located. For this approach to work in general, however, it is necessary to ensure that all names used in the L_1 (meta)-model do not appear in L_0 (i.e. in any ontological levels). However, this could be difficult in practice because the L_1 model naturally contains names that could appear in many domains. For example, "level" may very well not just occur in the linguistic dimension to represent the level a model-element

resides in, but may also be used to express the location of an elevator in the ontological dimension. To avoid such naming clashes, either names used in the L_1 model must be changed to highly unnatural ones (e.g. "linguistic-residence-level") etc. or the use of the most natural names (e.g. level) has to be prohibited in domain models. Neither of these would be particularly desirable.

An alternative approach shown in (b) is to introduce a special syntax for selecting between linguistic and ontological features, so that no ambiguity exists even when names from the L_1 (meta)-model are used in an ontological model. For every attribute, method, etc. this approach requires the origin of each referenced element to be identified (i.e. "_l_" for linguistic and "_o_" for ontological), which creates a big overhead when creating hybrid constraints. To minimize this overhead, the notation can be further refined as suggested in (c). This notation assumes the dimension in which the origin of the constraint resides as the default. A dimension must then only be explicitly specified if a user intends to reference the other dimension. In the example the default dimension is the ontological dimension as *CarType* is an O_2-level element.

The notation shown in (d), presented in [14], is used by MetaDepth to resolve linguistic and ontological name disambiguities. MetaDepth allows constraints accessing the ontological and linguistic dimension to be defined in the style of (a). If an ambiguity occurs, the linguistic dimension is marked by prefixing it with a ^ symbol as shown in (d).

4 Deep Constraints

The aspect of deep modeling which creates the most interesting opportunities is also the most challenging for DCLs: It is the unbounded number of ontological levels that may appear in a deep model. Deep constraints are constraints that may target levels more than one level below and may even have multiple levels in their scope. In the OCA implementation assumed in this paper, linguistic constraints cannot be deep because there are only two linguistic levels. In general, however, deep linguistic constraints may be useful to constrain instances of languages in a language family or ensure consistency across multiple language levels.

Deep Constraints can be classified as either level-specific or level-spanning constraints. In order to clarify the difference it is necessary to introduce some further terminology. More specifically, it is necessary to distinguish between the "instances" of a clabject and the "offspring" of a clabject. The instances of a clabject exist at the level immediately below that clabject, and can be direct or indirect instances. The offspring of a clabject, on the other hand, are all clabjects in the transitive closure over the "classifies" relationship starting with the subject. In other words, the set of offspring of a clabject includes all the instances of the clabject plus all the instances of those instances, and so on. The set of "direct offspring" is the set of all direct instances, plus all direct instances of those direct instances and so on recursively. In contrast to regular offspring, all indirect instances (at any depth) are excluded.

4.1 Level-Specific Constraints

Level-specific constraints are constraints that restrict the properties of clabjects at one specific level in the deep model relative to the origin clabject. The "scope" of the constraint then just comprises this single level. As explained above, we avoid the term "context" since it may lead to confusion because of the implicit universal quantification semantics of OCL constraints. In general, three cases can be identified – (a) some arbitrary specified level below the starting element, (b) the lowest level containing offspring of the origin and (c) the level containing the starting element itself.

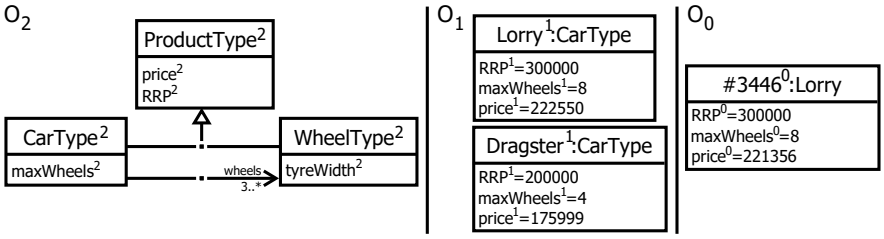


Fig. 6. Level specific constraint example.

When the specified level is defined to be the level immediately below the origin, the scope of the constraint coincides with that implied by an OCL "context". An example using the model elements of Figure 6 is shown in Constraint 6. The constraint ensures that all instances of *CarType* have an RRP between 10k and 400k, but specifically does not target the RRP values of cars. The scope of the level is specified in brackets after the origin keyword. Here, the "1" specifies that the relative distance of the scope to the level on which the constraint has been defined is one. Obviously, this kind of single level scoping also provides the option to evaluate the constraint on any arbitrary level below the origin.

Constraint 6 *The recommended retail price must be between 10k and 400k*

***origin**(1) CarType*
self.RRP > 10000 and self.RRP < 400000

The second scoping category defines constraints that apply to the lowest-level containing offspring of the origin clabject without making an explicit reference to the respective instantiation depth. The constraint given in Constraint 7 requires that all instances of *CarType*, at the bottom level have a *price* attribute which is no more than 80% of the RRP. This is specified using the symbol "_" for the scope of the constraint.

Constraint 7 *The price actually paid for a car shall be no more than 80% of the recommended retail price*

***origin**(_) CarType*
*self.price <= self.RRP * 0.8*

The last category represents constraints which cannot be supported in existing constraint languages even though the level they operate on (i.e. the scope) "exists" in traditional modeling approaches. This is the level of the origin element itself. For this reason we refer to this category of constraint as "intra-level" constraints. Such constraints cannot be expressed in traditional environments because in OCL the constrained elements always occupy the instance level while the starting element (referred to as the context in OCL) always occupies the type level.

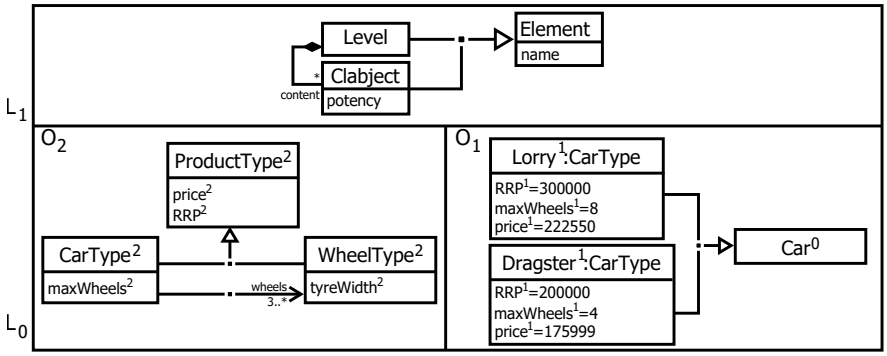


Fig. 7. Intra-level constraint example.

Intra-level Constraints This kind of constraint expresses restrictions on model content that resides on the same level as the origin clabject. An example of such a constraint is Constraint 8 which can be considered to enforce one aspect of the power type pattern [9] on the example displayed in Figure 7.

Constraint 8 *Subtypes of Car with potency higher than 0 must be instances of CarType (cf. PowerType Pattern)*

***origin**(0) Car*
self.getSubclasses() implies forAll(c |
c._l_ potency > 0 implies c.isTypeOf(CarType))

Here, all subtypes of *Car* are required to also be instances of *CarType*. Such a constraint cannot be reasonably expressed at the level of *CarType* since it would be attached to *CarType* but would have to explicitly restrict its applicability to

subtypes of *Car*. If the same or a similar constraint is applied to all subtypes of another superclass, e.g., *CarPrototype* then, in the absence of intra-level constraints, it would have to be attached to *CarType* as well with the respective applicability restriction in place. Such an approach would become unwieldy over time and its complexity is simply an expression of mis-locating the constraint(s).

Constraint 9 is another example of an intra-level constraint. It requires all non-concrete subclasses of a specific class (here *Car* in Figure 7) to have more attributes than *Car*, i.e., to exclude "hollow" subclasses. Again, it may not make sense to exclude hollow subclasses for all instances of *Car's* type and it seems to be unwarranted to force users to define a dedicated "≪non-hollow≫"-stereotype that is then applied only to *CarType*. In other words, we believe there are applications for "one-off" constraints that only apply to a particular instance, without having relevance to other instances of the same type.

Constraint 9 *Subtypes of Car with potency higher than 0 must add attributes*

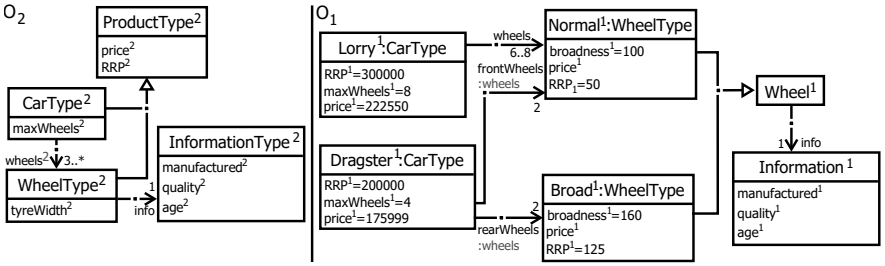
***origin**(0) Car*
self.getSubclasses() implies forAll(c |
c._l_potency > 0 implies
c.attributes→size() > self.attributes→size())

To fully support deep constraints, further capabilities are needed. First, vertical access to offspring and types at any level may be necessary to ensure certain kinds of consistency constraints. The functions isDirectOffspringOf() and isIndirectOffspringOf() could be used like their corresponding OCL statements but based on the notion of offspring rather than instances. Second, it is necessary to support horizontal, intra-level navigation to other elements by using navigation paths that are defined anywhere at levels above. A respective approach has been explored in [4].

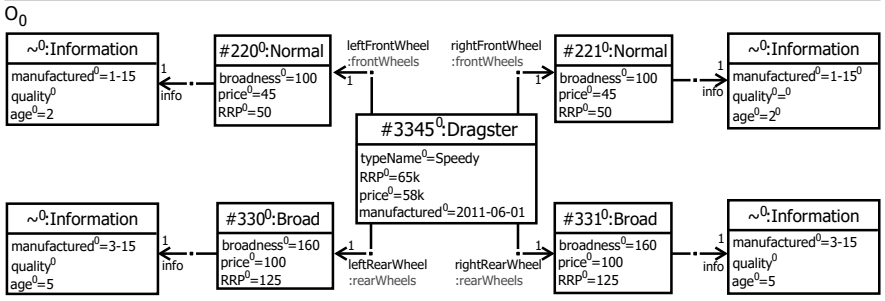
4.2 Level-Spanning Constraints

The common property of the previous category of scopes is that the constrained clabjects all occupy one specific ontological level. It is also possible, depending on the flavour of deep modeling in use, to also define constraints that span more than one ontological level. Such constraints would therefore have a scope greater than one in our terminology. This makes sense if the underlying deep modeling language supports uniform (ontological) attributes which, at all levels, possess a name, a type and a value. When an attribute has potency 1 or higher, the value of the attribute is interpreted as a default value for the corresponding attributes of the clabject's instances. In those cases where the constraint for the default values is the same as for the ultimate values, it is beneficial to interpret a constraint as being applicable over more than one level. In this case two situations seem to be useful in practice: (a) a specified arbitrary range of levels or (b) the level of the starting clabject and all its offspring.

In the first kind of level-spanning constraint, the range of levels over which the constraint should apply is explicitly specified relative to the origin clabject.



(a) Level O_2 and O_1



(b) Level O_0

Fig. 8. Level-spanning constraint example. Gray elements are extensions to the deep modeling approach as described in [4].

The constraint shown in Constraint 10 constrains all *WheelType* instances in Figure 8 occupying the following two levels (O_1 and O_0) to be not older than 24 months.

Constraint 10 *All offspring of wheelTypes, over two levels, are not allowed to be older than 24 months*

origin*(1..2) WheelType*
self.info.age $\leq$ *24*

The second kind of level-spanning constraint is a special case of the first kind. It simply uses the maximum scope possible, i.e., the level of the origin itself up to the lowest level containing offspring. Constraint 11 constrains all offspring of *CarType*, and *CarType* itself, to have a (default-) price of at least 10000.

Constraint 11 *A (default-) price for a car must exceed 10000*

origin*(0.._) CarType*
self.price $\geq$ *10000*

An example of a level-spanning constraint that includes intra-level navigation is shown in Constraint 12. The constraint specifies that, for security reasons, a *Dragster* must have wheels which are not older than seven months. As can

be seen in Figure 8, a *Dragster* is connected to its wheels using two different types of connections. There is one type for the front wheels and one type for the rear wheels, as dragsters require different wheel types depending on their location. For the purposes of Constraint 12, however, it would be desirable to navigate to all wheels in a convenient way, i.e., abstract away from the fact that there are front wheels and rear wheels. Thus, in version (a) using the DeepOCL syntax [10], the statement $CarType$ makes all navigations of its type available, here *wheels*. The MetaDepth version presented in [15] is shown in (b) which uses the "references" linguistic method to get the instances of the *wheel* references and access their value using the *value* method. The version in (c) displays the navigation semantics presented in [4]. The latter relies on the fact that the "wheels" role was introduced with potency two and hence makes the connection available at level O_0, plus the fact that "frontWheels" and "rearWheels" are declared to be instances of "wheels" (cf. Figure 8 (a)).

Constraint 12 *The wheels of Dragsters are not allowed to be older than 7 months*

origin*(1) Dragster*
(a) self.$CarType$.wheels.info implies forAll(age <= 7)
(b) self.references("wheels") implies
forAll(r | self.value(r).info.age <= 7)
(c) self.wheels.info→forAll(age <= 7)

Default Scope for Constraints As constraints in a deep constraint language may have a variety of scopes, the question arises as to which default scope should be assumed in case the modeler does not provide one. Default values in general, can reduce the complexity of a specification and relieve the modeler from explicitly providing a value that they would typically use in most cases. Requirements for a good default choice include

- frequency of occurrence. Making the value that occurs the most implicit, has the largest effect on specification reduction and will also most frequently relieve the modeler from providing it.
- robustness against change. Typical modifications to models should ideally not require the replacement of a default value with a different, specific one.
- generality across different host languages. One and the same deep constraint language may be applicable to a variety of different multi-level host languages. It would be desirable to be able to interpret constraints independently of the host language (e.g., MOF vs UML) and hence the default scope should ideally be always the same.
- validity of existing assumptions. It is desirable to make any extension – such as introducing depth to constraints – conservatively, i.e., not invalidate existing specifications if they do not need more than two levels. Changing the default value from a two-level technology (such as standard OCL) to another value in a multi-level context should only be done with a good justification in order to avoid gratuitously breaking the conservative extension property.

Currently there is no significant body of multi-level constraints from which solid frequency figures could be derived. Anecdotal evidence suggests, however, that most attributes follow a traditional pattern, i.e., are defined at a certain level and receive a value at a lower level. In other words, the default scope should probably not include "0", i.e., the current level.

The robustness argument suggests that it is probably not advisable to have the default scope include the lowest-level offspring (i.e., "_"). Consider the addition of a new lower level of used book copies to an existing model of (new) books. An existing constraint on prices for new books, should not be automatically re-targeted to used book copy prices, as the latter prices will obey different rules than new book prices.

Not all multi-level languages support the assignment to attributes at all levels at which they (implicitly) occur. This suggests that a default scope should probably not address multiple levels at once.

Finally, current OCL constraints assume an origin of "(1)" implying that OCL experts would have the least effort to adapt to it as a default scope. In combination, all prior observations suggest that the scope "(1)" has the highest appeal. However, further research is necessary to make a more informed choice. For example, there is a need to ascertain actual frequency of use figures, a need to observe and record typical model changes, and track the development of future multi-level languages to re-assess commonalities and differences.

5 Conclusion

In this paper we have identified the additional opportunities and challenges that arise when expressing constraints in the context of deep modeling. It is possible to write types of constraints on deep models that cannot be specified in traditional, two-level modeling environments (e.g. UML/OCL), such as hybrid constraints, deep constraints (targeting remote levels and/or spanning two or more levels) and intra-level constraints (supporting "one-off" constraints). These constraints do not increase the expressiveness compared to a regular two-level constraint language, as we do not introduce constraints over constraints and our proposed mechanisms could be translated into a two-level scheme. However, we believe that our proposed mechanisms are important for allowing modelers to adequately express constraints in a multi-level context. Expressing such constraints in a concise yet unambiguous way requires new concrete syntax and default conventions that do not yet exist. Some initial ideas for these have been presented in this paper, but it was not the goal to define or propose a definitive DCL. Nor was it the goal of this paper to describe precisely what kinds of requirements a DCL should aim to support, since the optimal language from a pragmatic perspective may not need to support every conceivable kind of constraint that can be imagined if no practical use case exists for them. The main goal of the paper was to characterize the kind of constraints that may make sense in the context of deep modeling and to provide a conceptual framework / terminology for discussing them.

References

1. Aspectocl: Extending ocl for crosscutting constraints. In: Taentzer, G., Bordeleau, F. (eds.) Modelling Foundations and Applications, Lecture Notes in Computer Science, vol. 9153, pp. 92–107. Springer International Publishing (2015), http://dx.doi.org/10.1007/978-3-319-21151-0_7
2. Ahmed, A., Vallejo, P., Kerboeuf, M., Babau, J.P.: Cdmcl, a specific textual constraint language for common data model. In: OCL 2014 – OCL and Textual Modeling: Applications and Case Studies (2014)
3. Atkinson, C., Gerbig, R.: Melanie: Multi-level modeling and ontology engineering environment. In: Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards. pp. 7:1–7:2. MW '12, ACM, New York, NY, USA (2012)
4. Atkinson, C., Gerbig, R., Kühne, T.: A unifying approach to connections for multi-level modeling. Models'15, Ottawa, Canada (2015)
5. Atkinson, C., Gerbig, R., Tunjic, C.: Enhancing classic transformation languages to support multi-level modeling. Software & Systems Modeling 14(2), 645–666 (2015)
6. Atkinson, C., Kühne, T.: Concepts for comparing modeling tool architectures. In: Briand, L., Williams, C. (eds.) Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 3713, pp. 398–413. Springer Berlin Heidelberg (2005), http://dx.doi.org/10.1007/11557432_30
7. Fowler, M.: Domain-specific languages. Pearson Education (2010)
8. Gogolla, M., Sedlmeier, M., Hamann, L., Hilken, F.: On metamodel superstructures employing uml generalization features. In: MULTI 2014–Multi-Level Modelling Workshop Proceedings. p. 13 (2014)
9. Gonzalez-Perez, C., Henderson-Sellers, B.: A Powertype-based Metamodelling Framework. Software & Systems Modeling 5(1), 72–90 (2006)
10. Kantner, D.: Specification and Implementation of a Deep OCL Dialect. Master's thesis, University of Mannheim (2014), https://ub-madoc.bib.uni-mannheim.de/37143/
11. Kolovos, D., Paige, R., Polack, F.: The epsilon object language (eol). In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture Foundations and Applications, Lecture Notes in Computer Science, vol. 4066. Springer Berlin Heidelberg (2006)
12. Kolovos, D., Paige, R., Polack, F.: On the evolution of ocl for capturing structural constraints in modelling languages. In: Abrial, J.R., Glsser, U. (eds.) Rigorous Methods for Software Construction and Analysis, Lecture Notes in Computer Science, vol. 5115, pp. 204–218. Springer Berlin Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-11447-2_13
13. de Lara, J., Guerra, E.: Deep meta-modelling with metadepth. In: Proceedings of the 48th international conference on Objects, models, components, patterns. pp. 1–20. TOOLS'10, Springer-Verlag, Berlin, Heidelberg (2010)
14. de Lara, J., Guerra, E., Cuadrado, J.: Model-driven engineering with domain-specific meta-modelling languages. Software & Systems Modeling 14(1) (2015), http://dx.doi.org/10.1007/s10270-013-0367-z
15. Lara, J.D., Guerra, E., Cuadrado, J.S.: When and how to use multilevel modelling. ACM Trans. Softw. Eng. Methodol. 24(2), 12:1–12:46 (Dec 2014), http://doi.acm.org/10.1145/2685615
16. Störrle, H.: Mocql: A declarative language for ad-hoc model querying. In: Van Gorp, P., Ritter, T., Rose, L. (eds.) Modelling Foundations and Applications, Lecture Notes in Computer Science, vol. 7949. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-39013-5_2