COMPUTER STANDARDS & INTERFACES 00 (2019) 000-000

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

TOTEM: Reconciling multi-level modelling with standard two-level modelling

Santiago P. Jácome-Guerrero[a], *Juan de Lara*[b]

[a]*Universidad de las Fuerzas Armadas ESPE, Sangolquí (Ecuador)*
[b]*Universidad Autónoma de Madrid, Madrid (Spain)*

A R T I C L E I N F O

Article history:
Received 00 December 00
Received in revised form 00 January 00
Accepted 00 February 00

Keywords:
Model-driven Engineering
Meta-modelling
Multi-level Modelling
Modelling Tools

A B S T R A C T

Model driven Engineering (MDE) advocates the active use of models throughout the different software development phases. In MDE, models are described using meta-models, one meta-level above. This approach effectively leaves developers with one single meta-level to create their models. However, there are scenarios where the use of multiple meta-levels results in simpler models with less accidental complexity. Hence, to simplify modelling in these cases, several multi-level modelling approaches and tools have recently emerged to increase the flexibility in modelling. While they provide advanced primitives to simplify modelling, there are possibilities to improve interoperability with mainstream two-level modelling approaches based on the Meta-Object Facility (MOF) standard of the Object Management Group (OMG), and achieve wider adoption.

For this purpose, we first characterize the design space of multi-level modelling approaches using a feature model. On such a basis, we provide a detailed comparison of existing multi-level modelling tools, identifying gaps and research opportunities. As a result of this gap analysis, we propose a new approach to multi-level modelling that embeds multiple meta-levels within one meta-model (i.e., encoding objects as classes, and instantiation as inheritance), and a tool – called TOTEM – which implements these concepts. The tool capabilities and its benefits in terms of interoperability with mainstream, standard modelling frameworks are illustrated through an example, as well as with empirical and analytical evaluations.

1. Introduction

Model Driven Engineering (MDE) promotes an active use of models throughout all development phases (Da Silva, 2015; Schmidt, 2006). In MDE, models conform to a meta-model describing the elements that can be used to create the model, and the constraints that models should obey. The mainstream approach to modelling in MDE is based on a two meta-level approach, where language engineers create meta-models, which then can be instantiated by developers to create models one level below. This approach is supported by standards, like the Meta-Object Facility (MOF) of the Object Management Group (OMG) (MOF, 2016), and widely used by de-facto standard implementations like the Eclipse Modelling Framework (EMF) (Steinberg, Budinsky, Merks, & Paternostro, 2008).

Even though the two-level approach has been very successful in practice, and is the dominant approach nowadays, some modelling scenarios benefit from the use of more than two meta-levels (de Lara, Guerra, & Sánchez, 2014). These cases appear when the type-object pattern or some of its variants arise (de Lara et al., 2014; Martin, Riehle, & Buschmann, 1997). In these cases, the use of two-level modelling requires workarounds, like emulating classes using objects, or performing so called *promotion model transformations* (which transform models into meta-models). These workarounds are different ways to

* *Corresponding author.* E-mail address: psjacome@espe.edu.ec

2214-4234/$ – see front matter © 2019 Elsevier Ltd. All rights reserved.
https://doi.org/10.1016/j.csi.2019.103390 [**This is an pre-print version with some amendments to Table 2**]

emulate a multi-level architecture with more than two meta-levels, within two, and as a consequence obscure the core of the ideas underlying the model. Therefore, for these cases, the use of multi-level modelling results in reduced accidental complexity, and allows focusing on the essence of the problem being modelled (Atkinson & Kühne, 2008; de Lara et al., 2014; Macías, Guerra, & de Lara, 2017).

These limitations of the two-level modelling approach have sparked a big interest in multi-level modelling technologies within the MDE community. This interest is witnesses by the MULTI series of workshops dedicated to multi-level modelling (MULTI, 2014), a Dagstuhl seminar (Almeida, Frank, & Kühne, 2018) and the plethora of multi-level modelling tools recently proposed, like DeepTelos (Jeusfeld & Neumayr, 2016), DeepJava (Thomas Kühne & Schreiber, 2007), Deep-Ruby (Neumayr, Schuetz, Horner, & Schrefl, 2017), the DPF Workbench (Lamo et al., 2013), Dual Deep Modelling (Neumayr, Schuetz, Jeusfeld, & Schrefl, 2018), FMMLx (Clark, Sammut, & Willans, 2015b; Frank, 2014), Kite (Guerra & de Lara, 2018), Melanee (Atkinson, Gerbig, & Kühne, 2015), MetaDepth (de Lara & Guerra, 2010), ML2 (Carvalho & Almeida, 2018; Claudenir M Fonseca, 2017), MultEcore (Macías, Rutle, & Stolz, 2016), NMF (Hinkel, 2018), OMLM (Igamberdiev, Grossmann, Selway, & Stumptner, 2018), and SLICER (Selway et al., 2017), among others.

Many of these tools offer sophisticated capabilities for modelling in multiple meta-levels, like attribute mutability and durability (Gerbig, Atkinson, de Lara, & Guerra, 2016), deep references (de Lara, Guerra, Cobos, & Moreno-Llorena, 2012), dual potencies (Neumayr et al., 2018), or refactoring support (de Lara & Guerra, 2018). Others propose specialized model management languages for them, like multi-level coupled transformations (Macías, Wolter, Rutle, Durán, & Rodríguez-Echeverría, 2019). However, there is currently an interoperability gap between multi-level modelling tools and mainstream two-level modelling. Just like other members of the multi-level community, it is also our belief that for a wider adoption of multi-level modelling within the MDE community, interoperability with de-facto standard two-level frameworks, like EMF, should be improved, enabling an easy migration of both two-level solutions into multi-level and vice versa. This will facilitate reuse of existing meta-models as a basis to create multi-level models, and enable the use of tooling developed for two-level modelling, for multi-level models.

In order to improve this situation, in this paper we characterize the design space for multi-level modelling solutions using a feature model (Kang, Cohen, Hess, Novak, & Peterson, 1990). We use this feature model as a foundation for a detailed comparison of existing multi-level modelling tools, charting the different approaches used, and performing a gap analysis to identify strengths and limitations. At the practical level, we observe that interoperability with two-level modelling is one of the factors that need improvement. Hence, with this goal in mind, we propose a new approach to multi-level modelling, which emulates objects at the meta-model level, and where instantiation is emulated with inheritance. While other approaches are based on emulating classes with objects (Rossini, de Lara, Guerra, & Nikolov, 2015), on dedicated multi-level meta-models (Atkinson, Gerbig, & Fritzsche, 2015; de Lara & Guerra, 2010), or on promotion transformations (Macías et al., 2016), to the best of our knowledge, this way of multi-level modelling has not been explored up to now.

We have implemented our approach in a tool called TOTEM (A Tool for MulTi-levEl Modelling, available at http://miso.es/tools/totem.html). The tool brings several benefits. First, it permits a more direct integration with two-level modelling approaches and frameworks like the EMF, as the multiple levels can be embedded with little modifications within a meta-model, preserving the semantics of the multi-level model. This way, exporting a multi-level model to two-levels is more direct than other approaches, while modelling at the instance level can be done using standard EMF tools. Moreover, these instance-level models can be manipulated with transformation languages and tools designed for two-level modelling, like ATL (Jouault, Allilaire, Bézivin, & Kurtev, 2008), Epsilon (Paige, Kolovos, Rose, Drivalos, & Polack, 2009) or Acceleo, an implementation of the MOFM2T OMG standard (Thomas Kühne, 2018b). Second, the approach supports reusing existing EMF meta-models to build a multi-level model, facilitating migration. This is the case because our approach is based on standard meta-models annotated with multi-level annotations. We show the benefits of our approach using a running example in the component-based domain, and present empirical and analytical evaluations.

This paper makes the following contributions: (i) A characterization of the design space for multi-level modelling solutions in the form of a feature model; (ii) a classification and comparison of existing multi-level modelling tools; (iii) a new conceptual approach to multi-level modelling, embedded within two meta-levels; (iv) a practical tool called TOTEM that implements these ideas in practice; and (v) empirical and analytical evaluations.

The rest of this paper is organized as follows. Section 2 introduces a running example, which we will solve using several approaches that emulate or support multi-level modelling concepts. The section introduces the main concepts behind multi-level modelling using this case study. Section 3 proposes a criterion for classifying existing approaches – based on a feature model – and gives a detailed classification and comparison of existing tools. Section 4 describes our approach to multi-level modelling, based on an embedding within a single meta-level. Section 5 describes the architecture we propose, and the TOTEM tool. Section 6 presents empirical and analytical evaluations. Finally, Section 7 ends with the conclusions and prospects for future work.

2. The road to multi-level modelling, by example

To motivate the need for multi-level modelling and its rationale, this section introduces a motivating running example. Imagine we would like to create a simple architectural language (Medvidovic & Taylor, 2010), so that users of the language (i.e., software architects) can define component types, made of port types (which can be input ports, output ports or both). Port types can define the types of messages they can handle, and be connected using connector types. We will assume that connectors have a configurable delay. Message types need to describe the structure of the data the messages carry. In addition to the defined data, all messages have a time stamp. Then, the defined types (components, ports, messages, connectors) can be used to define the run-time configuration of a system. Altogether, this is a typical component-based language, in the style of component-connector languages (Garlan & Shaw, 1993), and languages for component-based simulation (Zhu, Lei, Alshareef, Sarjoughian, & Zhu, 2018).

In the following, we will solve this problem using several *ad-hoc* approaches, which naturally lead to a solution based on native multi-level modelling. Then, we will use this solution to introduce the basic notions of multi-level modelling.

2.1. The type-object pattern

A first attempt to describe the running example is to define a meta-model that includes primitives to create both type elements and their instances. A possible solution using this approach is shown in Figure 1. The Figure shows that the meta-model needs to contain elements to represent component types and instances, port types and instances (both input and output), connector types and instances, and message types and instances, as well as the typing relations between types and instances (references *cmpType*, *pType*, *mType*, and *cnType*). This explicit representation of type and instance elements is called the type-object pattern (Martin et al., 1997). Please note that we also need to define attributes for message types (class *Attribute*), and give values for them (class *Slot*). This is a more specialized occurrence of the type-object pattern, which is called the *dynamic features* pattern (de Lara et al., 2014). At the model level, we use objects to represent both type-like elements (e.g., *Producer*, *Consumer*, *Pin*, *Pout*, *MChannel*, *Packet*), and object-like elements (e.g., *p*, *s*, *m*, *op*, *ci*, *i1*, *i2*, *c1*, *c2*). Hence, conceptually, the model encodes two meta-levels within one, and we have used a dashed line to convey such intention.

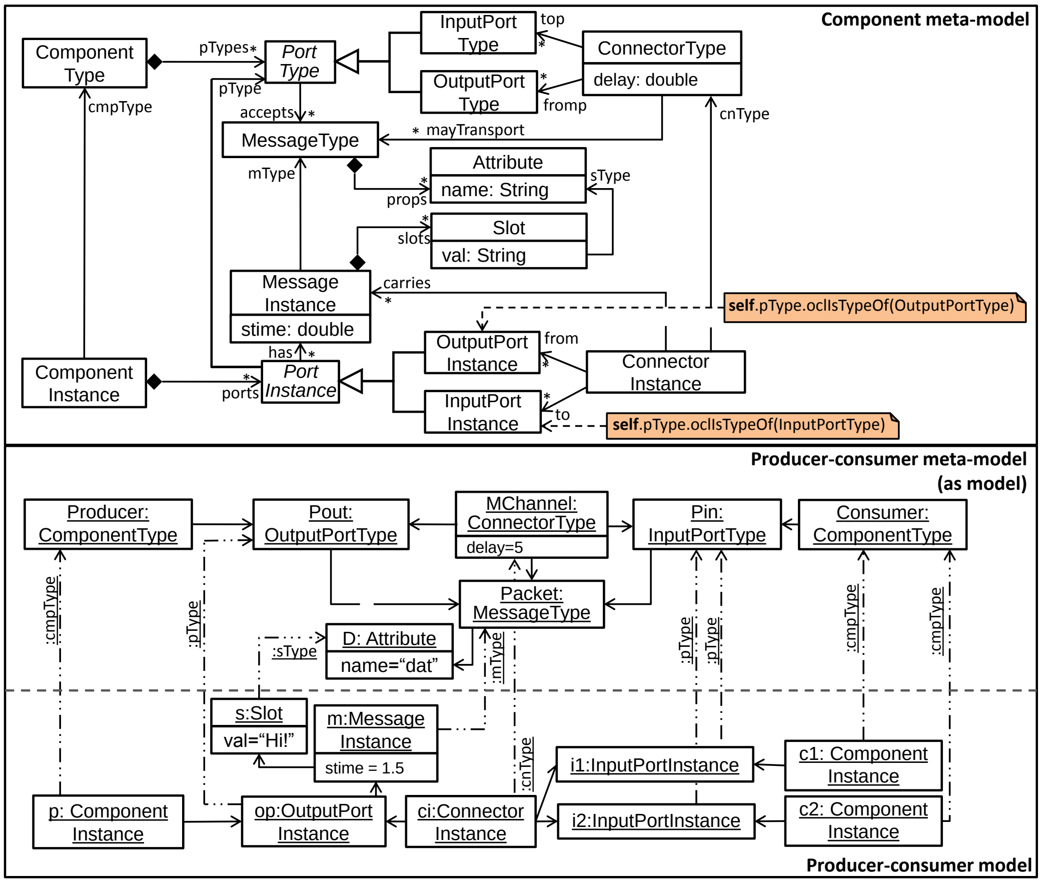


Fig. 1 - Solving the example using the type-object pattern.

Altogether, while this solution works in practice, it leads to a heavy replication of elements at the meta-model level, causing an increase of the accidental complexity of the meta-model (Atkinson & Kühne, 2008; Macías et al., 2017). In particular, it features six occurrences of the type-object pattern (*ComponentType/Instance*, *MessageType/Instance*, *PortType/Instance*, *InputPort/Instance*, *OutputPort/Instance*, *ConnectorType/Instance*), and one occurrence of the dynamic features pattern (classes *Attribute/Slot*). Moreover, should we wish to define cardinality constraints at the model level (e.g., in the relations between objects *Producer* and *Pout*, *MChannel* and *Pout*, etc) we would need to define additional classes and constraints in the meta-model, using the so called *relation configurator* pattern (de Lara et al., 2014).

2.2. The meta-model extension approach

An alternative approach consists in defining a meta-model that only includes primitives to create type-like elements, and rely on extension (through inheritance) to create specific component types, which then can be instantiated in models. This approach is called static in (de Lara et al., 2014). Figure 2 shows a solution to the example using this approach. This way, the types needed for creating the model are defined as subclasses of the component meta-model (instead of as instances). We have separated with a dashed line the elements of the original component meta-model from the extensions.

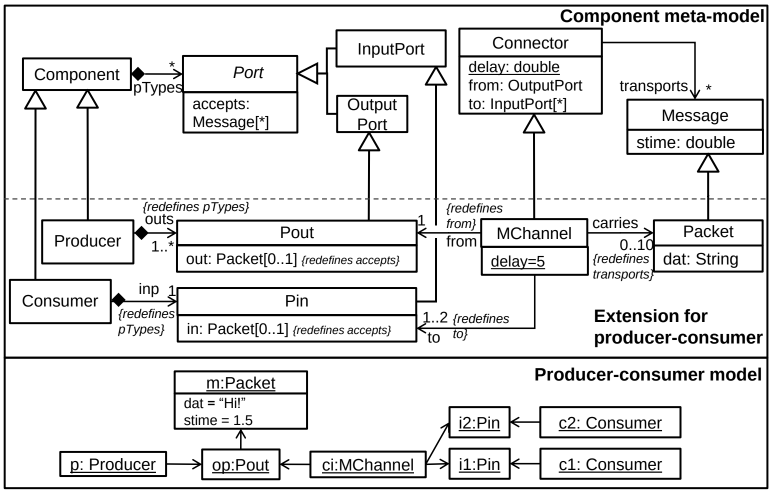


Fig. 2 - Solving the example using the meta-model extension approach.

This solution supports a more natural way of modelling elements with type facets (like *Producer*, *Consumer*, etc) at the meta-model level. Moreover, it supports the addition of attributes to subclasses (e.g., *Packet* is added attribute *dat*, which had to be explicitly modelled in Figure 1), and the definition of cardinalities for relations between the extended classes (e.g., for *outs*, *inp*, *from, to* and *carries*). However, it requires mechanisms to redefine associations (like *pTypes*, which is redefined in *Producer.outs* and *Consumer.inp*) and attributes (the static feature *delay* is redefined in *MChannel*, assigning it a default value). Additionally, at the model level, relations like *pTypes* should not be instantiated, but only its redefinitions (*outs* and *inp*). Please note that many meta-modelling frameworks – like the EMF – do not support these redefinitions. An additional problem is that, allowing the user to modify the core meta-model (i.e., the component meta-model in our case) may lead to unexpected changes (e.g., like creating a class inheriting from both *Component* and *Message*). To solve this latter issue, some approaches support the definition of extensibility rules, governing how a meta-model can be extended (Jácome & de Lara, 2018). However, that approach does not support the emulation of slots at the type level (like *MChannel.delay*), the control of the instantiation depth in the model, nor the fact that none of the classes (and some of the associations) of the component meta-model should be instantiable at the lowest level (i.e., in the producer-consumer model we should not be able to create an instance of *Component*).

2.3. Promotion transformations

A different approach to support the double instantiation required by the example is to introduce a transformation that converts objects into types, called a *promotion transformation* (de Lara et al., 2014). Figure 3 shows a solution of the example using this approach. The idea is to permit the instantiation of the component meta-model (see (a) in the figure), and through a model-to-model transformation, synthesize a meta-model (see (c) in the figure) out of the created model (b). Then this meta-model can be instantiated once more to create a producer-consumer model (see (d) in the figure). Hence, while the previous approaches embed two meta-levels within one (either at the meta-model level or at the model level), promotion transformations split the type-object nature of the intermediate model in two, and use a model-to-model transformation to create the type facets (c) from the object facets (b).

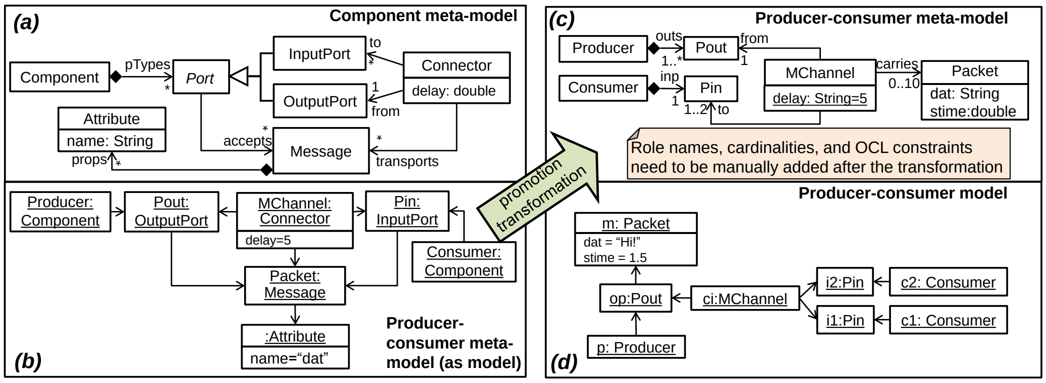


Fig. 3 - Solving the example using promotion transformations.

An ad-hoc approach based on building a promotion transformation specifically created for the given problem has several drawbacks (Drivalos, Kolovos, Paige, & Fernandes, 2008; Zschaler, Kolovos, Drivalos, Paige, & Rashid, 2009). First it needs to introduce a model-to-model transformation, which adds complexity to the solution and moves some design decisions from the meta-model to the transformation code. For example, the fact that *Message* instances at the bottom level (e.g., object *m* in model (d)) carry a time stamp *stime*, is not declared in any way in *Message*, but the transformation needs to create an attribute *stime* in every instance of *Message* (e.g., *Packet* in model (c)). Second, the approach explicitly decouples the object facet of models (i.e., model (b)) from its type facet (model (c)), which complicates model management. For example, role names, cardinalities and possible OCL invariants need to be added manually to meta-model (c) once the promotion transformation is executed. If model (b) is changed afterwards, the regeneration of (c) may override such manual changes. Finally, it is necessary to include in the meta-model (a) elements emulating type-level facets. In particular, we still need to include class *Attribute* to emulate the definition of attributes in class *Message*.

Please note also that, ad-hoc approaches using promotion require a transformation for each instantiation step, as depicted in Figure 4. In this Figure, a first transformation promotes instances of *Component* (like *Producer* in model (b)) into classes in model (c), adding them an attribute *maxInst* to control the maximum number of instances at level 0. Then, a second transformation promotes the objects in model (d) into classes in model (e), adding them an attribute *uuid* to store an identifier.

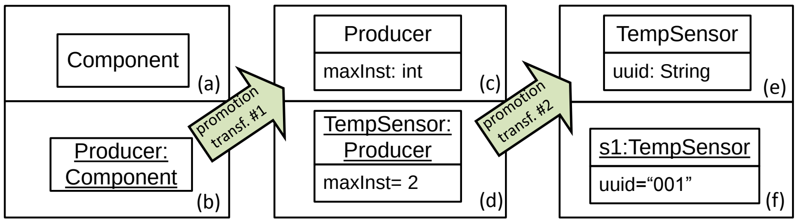


Fig. 4 - Solving the example using promotion transformations.

2.4. Multi-level modelling

The previous approaches have shown three different ad-hoc workarounds to encode a solution that naturally spans three meta-levels, into two. The workarounds, and their drawbacks are summarized in Table 1.

Table 1 – Pros and Cons of ad-hoc approaches emulating multi-level modelling.

Approach	Workaround	Advantages and Disadvantages
Type-object pattern	Includes elements in the meta-model enabling the creation of types and their instances at the model level.	✗ Heavy replication of elements, increasing the accidental complexity of the meta-model. ✓ Medium compatibility with standard two-level meta-modelling: standard tools can be used to edit and manipulate models, but constraints cannot be defined in user-defined types like *Packet* (as these types are at the model level).
Meta-model extension (static approach)	Emulates instantiation with inheritance.	✗ The core meta-model is offered to the developer. Extension control mechanisms are needed. ✗ Needs to emulate instances (e.g., attribute values) at the meta-model level. ✓ Good compatibility with standard two-level modelling: standard tools can be used to edit and manipulate models, and constraints can be defined in user-defined types like *Packet*.
Ad-hoc promotion transformations	Creates a transformation that promotes model elements into meta-model elements.	✗ Needs to create and introduce a transformation for modelling. ✗ Decouples the instance and type facets of elements. ✗ Still requires polluting the meta-model with elements to emulate meta-modelling facilities (e.g., class *Attribute* in Figure 3). ✗ Needs to create further transformations if more than three meta-levels are used. ✓ Medium compatibility with standard two-level meta-modelling: standard tools can be used to edit models, but integration with a transformation is needed. Constraints can be defined in user-defined types like *Packet*, but only in model (c) of Figure 3, not (b).

The workarounds summarized in Table 1 would not be necessary were the developer able to use several meta-levels instead of two. Figure 4 shows a solution to the running example using a multi-level modelling approach. It can be noted that the solution spans three meta-levels. The top one contains the primitives of the component meta-model, and does not require classes to emulate type-level (e.g., class *Attribute*) or instance level features (e.g., classes like *ComponentInstance* or *ConnectorInstance* in Figure 1). At the intermediate level (*producer-consumer* meta-model), we can use regular instantiation, and so we can e.g. assign a value to *MChannel.delay*. The elements at this intermediate meta-level retain both a type and an instance facet, being called clabjects (from the concatenation of the words *class* and *object*) (Atkinson, 1997).

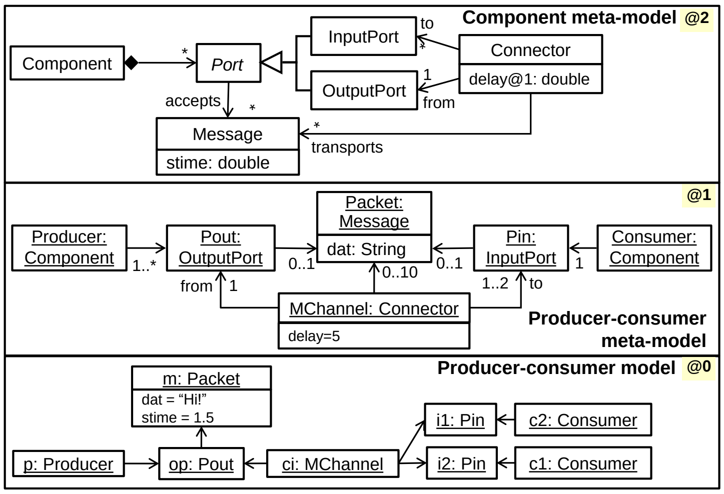


Fig. 5 - Solving the example using multi-level modelling.

At the top-level, it is interesting for clabjects to characterize their instances beyond the next immediate meta-level. This is done with the so-called *potency* (Atkinson, 1997), a natural number (or zero) that is associated to clabjects, attributes and references (which we collectively call fields). In the following figures, potencies are indicated after the "@" symbol. In our example, the top-level model is assigned potency 2, which means it can be instantiated at the next meta-level, and then one level below. The potency of a model is sometimes called its level. If a clabject does not explicitly declare a potency, it takes the potency of its container model. This way, all clabjects in the component meta-model receive a potency of 2 (however, please note that in general, clabjects with different potency may be contained in a given level). When a clabject is instantiated, the instance receives the potency of the type minus one. Clabjects with potency 0 cannot be instantiated

The rules for potency just described work in the same way for fields. Thus, in the Figure, *stime* in *Message* receives a potency of 2 from its owner clabject. This means that it will become a slot two levels below, and so it should receive a value in instances of instances of *Message* (like *m* at level 0). This is a way to control the features that indirect instances of a clabject need to have. In contrast, field *delay* in *Connector* explicitly declares a potency of 1, which means that it becomes a slot in instances of *Connector* (like *MChannel*).

In multi-level modelling, elements (models, clabjects, and fields) generally retain both a type and an instance facet. This also applies to references. Hence, references at level 1 are instances of those at the upper level, and types for those at the level below. Therefore, being instances, they should obey the cardinalities of their types, and being types they can declare a cardinality, to be obeyed by their instances. For example, in the Figure, reference *Connector.from* at level 2 declares a cardinality of 1. Then, *MChannel* (instance of *Connector*) at level 1 obeys such cardinality, as it defines exactly one instance of *from* (also called *from*). Reference *MChannel.from* defines a cardinality on its own, which is then obeyed at level 0 by clabject *ci*.

While potency permits a deep characterization of instances several meta-levels below, it is sometimes difficult to consider all possible features and primitives, especially when designing a language that is to be reused in different domains. This way, multi-level modelling allows clabjects to declare new fields (like *Packet*, which declares field *stime*), and also permits creating clabjects with no *ontological* type. Untyped fields and clabjects are called *linguistic extensions* (de Lara & Guerra, 2010).

Linguistic extensions are possible due to the so-called *Orthogonal Classification Architecture* (OCA) (Atkinson & Kühne, 2002). This architecture distinguishes two kinds of typing: *ontological* and *linguistic*. Ontological typing refers to instantiation within a domain. For example, in Figure 5, the ontological type of *m* is *Packet*. Linguistic typing refers to the meta-modelling primitive used to create the element. For example, in Figure 5, the linguistic type of *m* is *Clabject*. Therefore, linguistic extensions do not have an ontological typing, but only a linguistic one.

Figure 6 shows a scheme comparing the OCA (a) with the standard meta-modelling architecture (b). The OCA uses a linguistic meta-model that defines meta-modelling primitives, which become available at every meta-level. This means that meta-modelling facilities like inheritance can be used at any level, including level 0. In contrast, in the standard meta-modelling architecture (Figure 6(b)), the meta-modelling facilities (like inheritance, fields, and cardinalities) are only available at the meta-model level, and so they need to be reified again in the meta-model when the type-object pattern occurs. For example, in the solution of Figure 1, we needed to reify the instantiation relation, as well as the notion of attribute and slot. Please note that, in the standard four-level architecture of the OMG (MDA, 2014), in Figure 6(b) an additional meta-level would be shown at the bottom (representing the "real system"), and the top-level model is defined using itself.

Overall, for the running example, multi-level modelling yields the solution with the fewer number of elements, together with the meta-model extension approach. However, the meta-model extension approach would require mechanisms to emulate instance-level properties (slots, links) at the meta-model level, precisely define how the meta-model is allowed to be extended, and control over the (conceptual) meta-levels below. Compared with the other approaches, the main disadvantage of multi-level modelling is the lack of compatibility with standard two-level modelling approaches. As one of the advantages of the static approach is its compatibility with two-level modelling, in this paper we propose the realization and combination of multi-level modelling with the static approach (but solving its issues listed in Table 1), obtaining the benefits of both approaches.

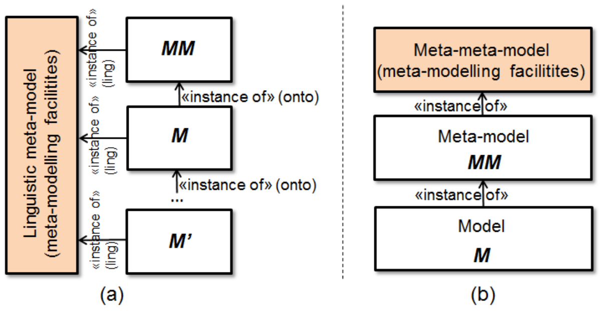

Linguistic meta-model (meta-modelling facilitites)
«instance of» (ling)
MM
«instance of» (onto)
M
M'
(a)
Meta-meta-model (meta-modelling facilitites)
«instance of»
Meta-model MM
Model M
(b)

Fig. 6 - (a) The orthogonal classification architecture; (b) The standard meta-modelling architecture.

3. Comparing and classifying multi-level modelling approaches

Once we have seen the benefits of multi-level modelling, and the scenarios where it applies, in this section we first explore the design space for multi-level modelling solutions, presenting a feature model of the different alternatives in Section 3.1. Then in Section 3.2 we perform a comparison of existing tools according to such criteria. Our goal is to identify gaps and opportunities for improving the current state of the art.

3.1. The design space of multi-level modelling solutions

Some researchers have proposed guidelines for comparing and classifying multi-level modelling approaches (Atkinson, Gerbig, & Kühne, 2014; Atkinson & Kühne, 2017). However, no systematic analysis of design decisions for multi-level modelling frameworks has been proposed by the multi-level modelling community. Such analysis could serve as a basis for comparing and classifying multi-level approaches, and identifying gaps in current research. Figure 7 shows a feature model (Kang et al., 1990) representing the space of possible alternatives to realize multi-level modelling solutions.

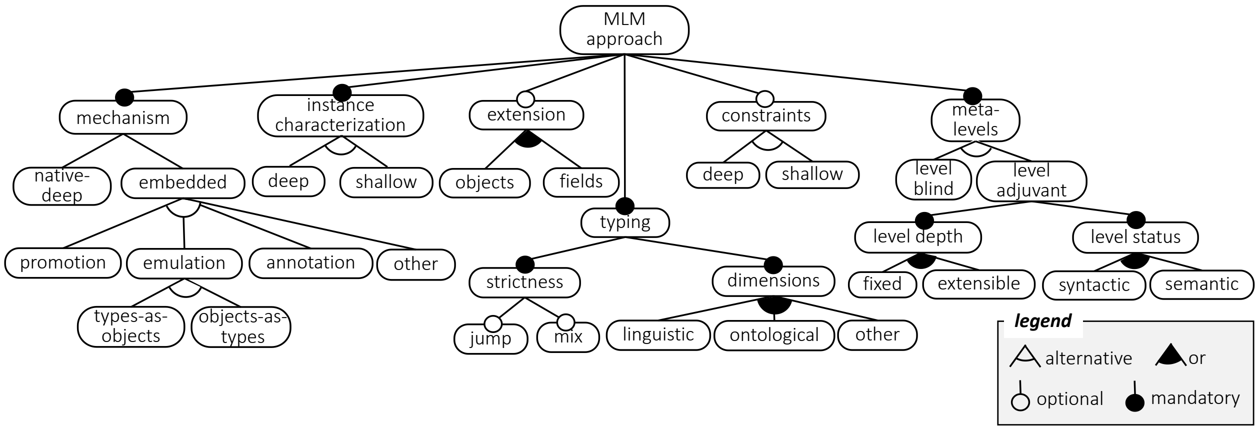


Fig. 7 - Characterizing the design space of multi-level modelling approaches.

Mechanism. In the first place, the multi-level modelling *mechanism* can be *native*, or *embedded* in a two-level technology. As seen in Section 2.4, native mechanisms do not embed multi-level modelling within a two-level framework, but instead provide primitives and artefacts specifically designed for multi-level modelling. For example, MetaDepth (de Lara & Guerra, 2010) provides a native mechanism supported by a linguistic meta-model that models the instantiation relation explicitly, and includes the notion of potency.

Regarding embedded approaches, there are several alternatives to emulate multiple meta-levels within a two-level framework, some of which we have analyzed in Section 2. One possibility is to resort to promotion transformations, as seen in Section 2.3. Approaches using promotion transformations can be either ad-hoc, which rely on the construction of domain-specific transformations, for example, for traceability (Drivalos et al., 2008; Zschaler et al., 2009) or general purpose (Macías et al., 2016). While ad-hoc approaches suffer from the drawbacks of Table 1, general purpose approaches may circumvent them, as we will see in the analysis of tools of Section 3.2.

Another approach, seen in Sections 2.1 and 2.2, consists in emulating the type facet in objects (feature *types-as-objects*), or the object facet in types (feature *objects-as-types*). The former approach is known as the type-object pattern (Martin et al., 1997), while the latter is the meta-model extension approach, also known as the static approach (de Lara et al., 2014). Using the type-object pattern, the multi-level model is embedded at the model level, and the instantiation relation is modelled as a normal relation. Instead, in the static approach, the multi-level model is embedded in the meta-model, and instantiation is emulated with inheritance. Annotation-based approaches (feature *annotation*) annotate the underlying language meta-model (e.g., the UML) with information about the characteristics that elements in the domain model have at different levels. For example, UML stereotypes is an annotation based mechanism (UML, 2017). We refer to (de Lara et al., 2014) for examples of multi-level modelling using stereotypes.

While these are the most common idioms for embedding multi-level modelling into two-level frameworks according to (de Lara et al., 2014), the feature model is not closed, indicating that *other* mechanisms may exist as well.

Instance characterization. Some approaches to multi-level modelling are able to characterize instances at lower meta-levels, not only the next one (as in standard two-level modelling). This is called *deep characterization* (Atkinson & Kühne, 2001). *Potency* is one such mechanism for deep characterization. In contrast, other approaches only support defining features for instances exactly one level below. We say they support *shallow* characterization. For example, multi-level approaches based on *powertypes* do not support deep characterization (Odell, 1994).

Extension. As mentioned in Section 2, when designing a multi-level model, it is difficult, if not impossible, to foresee all possible primitives that are to be used at the level below. This is especially true when designing generic languages (e.g., process modelling languages), applicable to several domains (e.g., software process modelling, modelling of production processes, etc). In these cases, the different domains may introduce new primitives, or add new fields to objects, not foreseen by their types. In the running example, we needed to add new fields (like *dat*) to instances of *Message*. For this reason, some approaches permit linguistic extensions, that is, clabjects with no ontological typing, or new fields with no ontological typing.

Typing. Often, meta-modelling approaches advocate strict typing (Atkinson, 1998), in which every element at each level (except at the top) is typed by exactly one element at the level above. This implies three aspects: the typing relation does not jump meta-levels (*no jump*), instances and their types do not coexist in the same level (*no mix*) and there are no untyped elements (*no extension*). Non-strict meta-modelling is sometimes termed *loose* meta-modelling (Atkinson, 1998). In the feature model, we are more precise, by identifying the cause for non-strictness, we model *jump* and *mix* as two subfeatures of *strictness*, while the support for untyped elements is modelled by the *extension* feature.

Additionally, approaches may support different kinds of typing, like ontological or linguistic typings (see Section 2). While these are the most common dimensions, some works have proposed other ones, like the *realization* dimension (Igamberdiev et al., 2018). Please note that, in addition to dimensions, some approaches allow several typings of the same kind, typically ontological (Balaban, Khitron, Kifer, & Maraee, 2018), which are sometimes called *supplementary* dimensions (Macías, Rutle, Stolz, Rodriguez-Echeverria, & Wolter, 2018). However, we do not capture this fact in the feature model, because it is of lower relevance to multi-level modelling.

Constraints. In two-level modelling, the integrity constraints defined in meta-models apply to models exactly one level below. These constraints include cardinality constraints, constraints associated to constructs like compositions, and others defined in constraints in languages like OCL (OCL, 2014). In a multi-level setting, constraints may be defined for the next meta-level (shallow constraints), but it is also useful to define constraints for models at meta-levels below the next one (deep constraints). Please note that deep constraints subsume shallow ones. Some approaches have extended the shallow semantics of elements like cardinalities to a multi-level context (making them deep) (Atkinson, Gerbig, & Fritzsche, 2015), others have extended constraint languages with deep semantics (Clark & Frank, 2018; de Lara & Guerra, 2010), or proposed transformation languages that can be used to express multi-level constraints (Macías et al., 2019).

Meta-levels. Some notion of level is frequent in multi-level modelling approaches (as the word "level" is even within the name of the approach). These approaches are called *level-adjuvant* (Atkinson et al., 2014; Thomas Kühne, 2018b). Instead, a few approaches are *level-blind*, where a single level may contain arbitrary structures of ontological classification relations (Henderson-Sellers, Clark, & Gonzalez-Perez, 2013; Mylopoulos, Borgida, Jarke, & Koubarakis, 1990; Neumayr et al., 2018). The notion of level can be given a *semantic* or a *syntactic* status (Balaban et al., 2018). In the former case, a level emerges from the order (i.e., instantiation power) of the elements it contains (Thomas Kühne, 2018b). The latter provides just a packaging mechanism for related elements.

It must be noted that most multi-level modelling approaches use the instantiation relation to relate levels. This has the advantage of enabling the use of mechanisms for deep characterization that work across instantiation relations, like potency. The notion of level is therefore inherent to instantiation, as instantiating an element is done at the level below the type. One may wonder therefore, whether other relations, like generalization (Borgida, Mylopoulos, & Wong, 1984; T. Kühne, 2009; Mylopoulos et al., 1980; Poo, Kiong, & Ashok, 2008) could be used to relate levels. On one hand, generalization provides a native solution (based on multiple inheritance) for multiple classification (where an object may have several types), while this problem is challenging for standard instantiation-based frameworks (de Lara & Guerra, 2017; SMOF, 2013). On the other, it complicates deep characterization. As one can have inheritance hierarchies of arbitrary depth within a level, a notion of potency that would work across inheritance would be challenging without introducing some packaging mechanism to emulate levels. Actually, in this work we propose emulating instantiation with inheritance, for which we introduce annotations (e.g. for potency) and use packages to emulate levels. By using a package to encapsulate models, a type model can be "instantiated" (i.e., extended) as many times, by creating additional "instance" packages. Just like instantiation, inheritance does not pollute the type level model, because the dependency is from the child model to the parent model.

Multi-level modelling systems support modelling using more than two meta-levels. However, in some systems, the level of a model has to be decided when such model is constructed, and this decision cannot be changed when working with models at the lower levels. For example, this is the case if we assign a given potency (say 3) to a model. Other approaches are more flexible, enabling the extension of the number of meta-levels when modelling at lower levels. This is for example the case of approaches permitting "*" (star, or unbounded) potency (de Lara & Guerra, 2010; Kennel, 2012), the recent notion of characterization-potency (Thomas Kühne, 2018a), or approaches where levels can always be extended (Macías et al., 2018). The latter approaches have the advantage of more flexible instantiation (e.g., the developer does not need to worry about increasing the potency of a model from say 2 to 3 if more instantiations are required), at the cost of a lower control of the maximum allowed depth.

3.2. Classifying multi-level modelling approaches

Next, we analyse some prominent multi-level tools, using the feature model of Figure 7 as a basis. Table 2 summarizes the resulting classification. In addition to the features of the feature model, we have also included other criteria, to provide a better understanding of a tool's aim, its target and capabilities. These extra criteria are:

- *Purpose*: whether the aim is modelling (in the scope of model-driven engineering, or information systems), knowledge representation or programming, among others. This characteristic is useful to better understand the provenance of the tool and the rationale for its features.
- *Support for management languages*: whether languages are provided to manage the multi-level model. In the context of MDE, management languages include those for model transformation or code generation. This feature helps in understanding the usefulness of the system for software development tasks.
- *Compatibility with two-level modelling*: the degree in which the approach is compatible with tools developed for standard two-level modelling, like editors, transformation or constraint languages.

Next we comment on some aspects of these tools in more detail.

Table 2 – Classification of multi-level modelling tools.

Tool / Feature	*MetaDepth*	*Melanee*	*MultEcore*	*DPF Workbench*	*DeepRuby*	*DeepJava*	*DeepTelos*	*OMLM*	*ML2*	*DDI*	*FMMLx*
Mechanism	Native-deep (MLM meta-model)	Native-deep (MLM meta-model)	Automated promotion transformations	Native-deep (specifications)	Other (Meta-prog. in Ruby)	Other (Extension of Java)	Native-deep (Telos propositions)	Native-deep (MLM meta-model, Flora- 2)	Native-deep (powertypes)	Native-deep (Telos propositions)	Native-deep (Xcore extension with intrinsic feature)
Instance charact.	Deep (potency, leap potency, star potency)	Deep (potency, durability, mutability, star potency)	Deep (potency, range potency)	Shallow	Deep (dual potency)	Deep (potency)	Deep (most general instances)	Deep (potency, leap-potency)	Deep (regularity attributes)	Deep (dual potency)	Deep (level attribute)
Extension	Yes (objects, fields)	Yes (objects, fields)	Yes (objects, fields)	No	No	Yes (object, fields)	Yes (objects, fields)	No	No	No	Yes (objects, fields)
Typing	Non-Strict (extension, jump, no-mix) Linguistic, ontological	Non-Strict (extension, no-mix, no-jump) Linguistic, ontological	Non-Strict (extension, jump, no-mix) Linguistic, ontological	Strict Ontological	Non-strict Linguistic, ontological	Non-strict Linguistic, ontological	Non-strict Linguistic, ontological	Non-strict Linguistic, Ontological, Realization	Non-strict (no-jump)	Strict Linguistic, Ontological	Non-Strict Linguistic, ontological
Meta-Levels	Level-adjuvant Syntactic Semantic Extensible	Level-adjuvant Syntactic Semantic Extensible	Level-adjuvant Syntactic Extensible	Level-adjuvant Syntactic Extensible	Level-blind	Level-blind	Level-blind	Level-adjuvant Syntactic Fixed	Level-adjuvant (via ordered types)	Level-adjuvant Syntactic Semantic Fixed	Level-adjuvant Syntactic Semantic Extensible
Constraints	Deep (Epsilon object language)	Deep (Deep OCL)	Shallow (only multiplicities)	Deep	Deep	Deep	Deep	Deep MULLER/ Flora2	Shallow	Deep (axioms and constraints in ConceptBase)	Deep (XOCL, a variant of OCL)
Purpose	MDE	MDE	MDE	MDE	Programming	Programming	Knowledge Representation	Knowledge Representation	Knowledge Representation	Knowledge Representation	Information systems
Support for management languages	Yes (Epsilon languages, multi-level refactorings)	Yes (Deep ATL)	Yes (Multi-level coupled transformations)	Yes (code generation with Xpand)	Yes (Ruby)	Yes (Java)	No	No	No	No	Yes (XMF action language)
Comp. with two-level modelling and tools	No	Medium (based on EMF)	Medium (based on EMF)	No	N/A	N/A	No	No	No	No	No

MetaDepth (de Lara & Guerra, 2010) is a framework for multi-level modelling using a textual concrete syntax. The approach is based on a native embedding, using a linguistic meta-model that supports deep characterization through potency. Several variations of potency are considered, including leap potency and star (unbounded) potency (de Lara et al., 2012). Leap potency means that if a clabject is assigned leap potency *n*, it will be possible to create instances of the clabject *n* levels below, without needing to instantiate the clabject at intermediate levels. MetaDepth supports linguistic extensions, where both untyped clabjects and fields can be created, due to the support for a dual linguistic and ontological typing (de Lara & Guerra, 2010). Due to leap potency and linguistic extensions, the ontological typing is non-strict, but a clabject and its type cannot be in the same model. The tool supports an explicit notion of level (the potency of the model), both syntactic and semantic. The number of meta-levels is extensible, due to unbounded potency. Integrity constraints are expressed in EOL (Kolovos, Paige, & Polack, 2006) (a variant of OCL), and are deep, as they can be assigned a potency. The purpose of MetaDepth is its use for building MDE solutions, and for that purpose it has been integrated with the Epsilon languages for model transformation (in-place and model-to-model) and code generation (Paige et al., 2009). It also supports other advanced features, like multi-level refactorings (de Lara & Guerra, 2018). The tool is not directly compatible with two level modelling, and cannot use tooling from standard meta-modelling frameworks.

Melanee supports modelling through deep, multi-format, multi-notation user-defined languages (Atkinson & Gerbig, 2016). The deep modelling component of the tool, underpinned by the orthogonal classification architecture and deep instantiation, supports an arbitrary number of meta-levels. The concept of deep instantiation is implemented through potencies, which are attached to clabjects (*potency*), features (*durability*) and attribute values (*mutability*). Melanee has an explicit notion of level (hence it is a level-adjuvant approach) with both syntactic and semantic support. It also defines the notion of star potency, which permits an unbounded number of levels, and is a means to support the representation of types having instances of different potencies. Melanee does not support leap potency, but, as it supports untyped clabjects and fields, its ontological typing cannot be considered strict. The tool supports a variant of OCL with deep semantics and has been integrated with the Atlas Transformation Language (ATL) for model transformations (Atkinson, Gerbig, & Tunjic, 2015). The aim of Melanee is its use in model-based software projects, and is built on the Eclipse Platform using EMF (Atkinson & Gerbig, 2016). However, it still needs importers and exporters for their compatibility with standard Ecore meta-models and two-level architectures (i.e., adaptation is required for interoperability with standard modelling tools).

MultEcore is a novel graphical multi-level modelling framework built using EMF (Macías et al., 2017; Macías, Rutle, Stolz, Rodriguez-Echeverria, & Wolter, 2018). It is based on repeatable promotion transformations, which are applied each time the user needs to create a new instance model. However, it does not incur in any of the drawbacks mentioned in Table 1, since the approach is not ad-hoc, but generic and fully automated. MultEcore supports *range potency*, which extends leap potency with the possibility to instantiate an element within a range of meta-levels, hence its typing is not strict. MultEcore is level-adjuvant, having a syntactic notion of level. Even though it does not support star potency, it permits an arbitrary number of meta-levels, as models do not have an explicit potency, and hence can always be instantiated (hence we classify its level notion as non-semantic, since it does not carry an explicit potency). The approach has been formalized using category theory. MultEcore supports untyped clabjects and fields. MultEcore contains a dedicated transformation language, which permits expressing model transformations (Macías et al., 2019). This way, multi-level constraints, defined using multi-level coupled transformations could be used to define constraints which span across multiple levels. The purpose of the MultEcore is its use within MDE projects. However, it is not directly compatible with two-level architectures and tools, but preserving the semantics of the multi-level model would require an exporter accumulating all meta-models created in the applied promotion transformations.

DPF Workbench supports modelling with an arbitrary number of meta-levels (Lamo et al., 2013), but not deep characterization, or untyped elements. Hence, the approach is level-adjuvant, providing syntactic support for levels, but similar to MultEcore, we classify its level notion as non-semantic, since it does not carry an explicit potency. Typing is strict and the approach is based on the Diagram Predicate Framework (DPF), which provides a formalization of models/meta-models and model transformations. The DPF is an extension of the generalized framework of sketches (Diskin, 2005), based on category theory (Barr & Wells, 1990). This way, constraints are not specified in OCL, but using graphical signatures, with formal semantics. The tool is based on Eclipse, but a web version is also available. DPF is a tool for creating MDE solutions, and hence includes a code generator facility, supporting the *Xpand* language.

DeepRuby is a Ruby implementation of the Dual Deep Instantiation (DDI) approach (Neumayr et al., 2017). For this purpose, the approach uses the meta-programming facilities offered by Ruby, especially the notion of *eigenclass*. Being embedded within a programming language, it supports defining behavior (methods). We consider it level-blind, since the notion of level was not embedded in the language. Being a programming language, we do not consider compatibility with two-level modelling approaches in this case.

DeepJava extends the capabilities of Java, supporting the definition instantiation chains of arbitrary depths (Thomas Kühne & Schreiber, 2007). DeepJava introduces the notion of potency into the programming language, and its working scheme is based on compilation into standard Java code. Similar to DeepRuby, we classify the typing of DeepJava as non-strict, because there is no explicit notion of level, and therefore we consider it a level-blind approach. As the goal is programming, we do not consider interoperability with two-level modelling.

DeepTelos extends the Telos knowledge representation system with deep characterization through the notion of most general instances (MGI) and not with numerical potencies (Jeusfeld & Neumayr, 2016). Several core constituents of multi-level modelling were already present in Telos (Mylopoulos et al., 1990), like the uniform treatment of classes and objects, unrestricted lengths of instantiation chains, and similar ideas to the OCA. We classify the typing in DeepTelos as non-strict, as there are no explicit modelling levels. Hence, we consider it a level-blind approach. The approach is implemented in ConceptBase (Jarke, Gallersdörfer, Jeusfeld, Staudt, & Eherer, 1995). The approach is directed to knowledge representation, and so it does not consider model management languages, and is not compatible with two level frameworks.

OMLM is the acronym of Open integrated framework for Multi-Level Modeling (Igamberdiev et al., 2018). This is a tool implemented in Flora-2, an open-source implementation of F-Logic with numerous extensions (Yang, Kifer, Zhao, & Chowdhary, 2005). The tool stems from the field of knowledge representation. In addition to create multi-level models (by instantiating a native multi-level meta-model), OMLM can import existing two-level models. In the latter case, a process of linguistic and ontological transformation is used that considers the steps of identification, mapping and transformation. OMLM supports single and multi-potency. The former is similar to leap potency, while the latter corresponds to standard potency. OMLM supports querying by using the MULti-LEvel Reasoner (MULLER) framework, which can also be used to verify if the transformation from a two-level model resulted in a correct multi-level model (i.e., it is conformant to the native multi-level meta-model). The approach supports a notion of level (i.e., it is level-adjuvant), but with no semantics attached.

ML2 is a textual modelling language supporting a theory for multi-level conceptual modelling named MLT (Carvalho & Almeida, 2018) (Claudenir M Fonseca, 2017; C. M. Fonseca, Almeida, Guizzardi, & Carvalho, 2018). It includes a UML profile for visualization and combines the approaches based on powertypes and clabjects. ML2 lacks mechanisms for deep characterization of instances. However, it supports the notion of regularity feature, which permits constraining the values of features at lower levels. While ML2 lacks an explicit notion of level, types have an order, which can be used to emulate levels. At the tool level, elements can be organized within modules, which can be used to group related elements.

Dual Deep Instantiation (DDI) is a multi-level modelling approach that uses deep characterization through potency. DDI allows for relationships that connect objects at different instantiation levels (Neumayr et al., 2018). The potency may be specified separately for each end of the relationship. DDI has been formalized using F-logic, and implemented on top of ConceptBase (Jarke et al., 1995), a knowledge representation system based on Datalog semantics and the Telos data model (Mylopoulos et al., 1990). The approach is level-adjuvant, providing both syntactic and semantic support. However, it does not support star potency, and hence the number of levels is fixed a priori, and does not support linguistic extensions (but this can be emulated creating a different clabject hierarchy). Being directed to knowledge representation, it is not integrated with model management languages, and is not directly compatible with two-level modelling.

Flexible Meta-Modeling and Execution Language (FMMLx) enhances the expressivity of traditional object-oriented conceptual modelling by multi-level capabilities. FMMLx is a multi-level modelling and execution language that extends XCore (Clark, Sammut, & Willans, 2015a). A multi-level model created with FMMLx may not only include objects on different levels of classification, but also intrinsic attributes, operations, and associations (Frank, 2014). The approach can be classified as level-adjuvant, providing semantic and syntactic support for levels. FMMLx is supplemented by a meta-modelling environment that extends XModeler (LE4MM, 2018). The framework has two main components. A generic model editor enables the creation of (meta-) models by using a generic, UML-like notation. Second, a concrete syntax editor supports designing the symbols on which a graphical notation is based and additional widgets such as menus, text fields or buttons. The symbols created with this editor are mapped to their respective (meta-) model elements using a generic tree structure that is part of XModeler, thereby completing the definition of a DSML. The approach supports executability by means of the XMF action language.

Overall, we observe a large variety of tools supporting sophisticated constructs for multi-level modelling. However, many tools lack integration with model management languages (e.g., for model transformations or code generation). The reason is the high cost of adapting the semantics of a two-level transformation language to a multi-level semantics. We also observe that, while some tools support importers (from two-level models into multi-level), and some tools are based on EMF, generally the multi-level models cannot be directly interpretable as valid standard models. The latter could be achieved by flattening all meta-levels, except level 0, within a single meta-model. This would permit treating a multi-level model as a standard two-level model, and hence integration with model management languages, like ATL or Acceleo, and standard model editors, would be achieved for free. For this purpose, next section described an approach that follows this philosophy.

4. Meta-model extension approach to multi-level modelling

Our approach is based on the observation that multi-level models can be embedded within two-levels by the use of inheritance instead of instantiation, as shown in Section 2.2. This approach has the advantage of maintaining compatibility with standards, such as EMF, because it permits starting a multi-level model from a standard Ecore meta-model. This also has the advantage of reusing all meta-modelling facilities of Ecore, like composition and opposite references, and a rich set of data types. However, to solve the issues identified in Table 1, it requires improving the control of how the extension of a meta-model can be performed, as well as emulating instance facets in classes. By adding such control mechanisms, the creation of incorrect extensions or prohibited modifications of the base domain meta-model are avoided. Figure 8 shows a schematic representation of the approach.

In the first phase, a standard domain meta-model (label 1 in Figure 8) is decorated with multi-level annotations. Such annotations are defined as a model, conforming to the MLM meta-model. Then (label 2), the instantiation of such a base domain meta-model is emulated via meta-model extension. The possible extensions are regulated by the defined multi-level annotations. Please note that, while the bottom level can also be emulated with subclassing (as noted by other researchers, like (Bąk, Diskin, Antkiewicz, Czarnecki, & Wąsowski, 2013)) we provide the option to "compile" the multi-level model into a standard meta-model (label 3). This compilation accounts for the special semantics of multi-level models, like encoding the differences between instantiation and inheritance, and setting as abstract those classes with potency bigger than 1. This results in a standard meta-model that can be instantiated using standard tools. Any instance model (label 4) of the compiled meta-model conforms to the multi-level model as well. This approach has the advantage that standard MDE tools (e.g., for model transformations) can be used to manipulate the bottom-level model.

In the following, we describe each one of the steps in our approach.

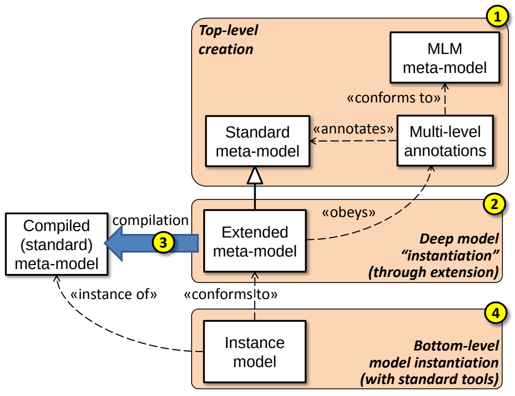


Fig. 8 - Meta-model extension approach to multi-level modelling.

4.1. Annotating standard meta-models with multi-level annotations

Figure 9 shows the meta-model we use in our approach (called MLM meta-model in Figure 8) to decorate the standard domain meta-model with multi-level modelling annotations. Overall, four types of elements can be annotated: meta-models (*EPackage* in EMF), classes (*EClass* in EMF), references (*EReference* in EMF) and attributes (*EAttribute* in EMF). This way, our meta-model contains classes annotating these elements. As we support deep characterization, all elements can be annotated with a potency (all annotation classes inherit from *MLPotency*). If the potency value is -1, it means that the potency is unbounded (*star* potency)

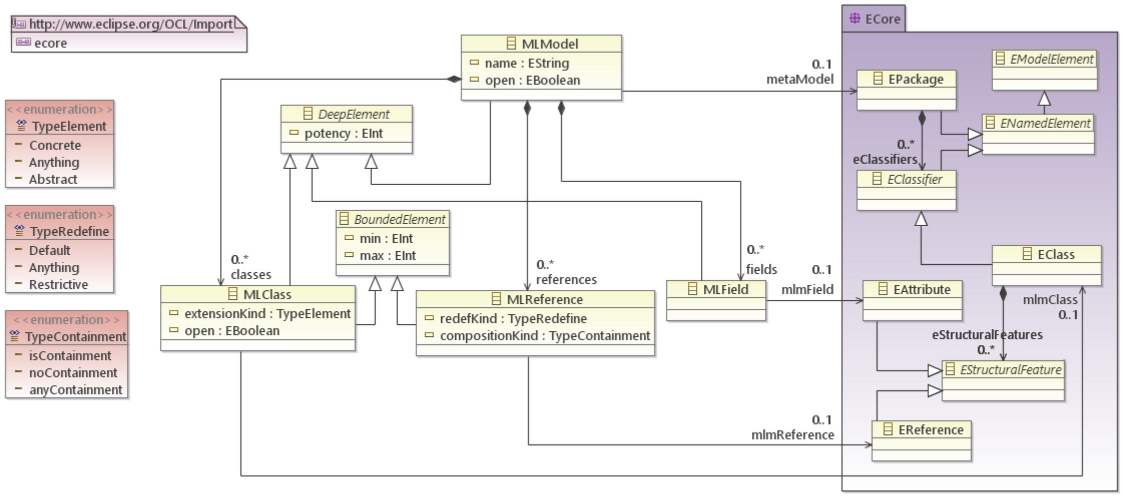


Fig. 9 - Meta-model for multi-level modelling annotations.

Domain meta-models are annotated with *MLModel* objects. These point to the *EPackage* being annotated via the *metaModel* reference. In addition, the meta-model annotations may include an optional name for the specification, and a flag *open* to control whether it is possible to add linguistic extensions in the meta-model instances.

MLClass objects annotate *EClasses* via the *mlmClass* reference. The annotation permits controlling the potency of the class; the range of instances the clabject may have at the next level (interval [min..max]); whether the instances can be linguistically extended with new fields (if *open* is true); and whether these instances should be abstract, concrete or any of the two.

EReferences can be annotated with *MLReference* objects. These support declaring a potency for the reference, a range of allowed instances for the reference (an interval [min..max]), an indication of allowed cardinality declarations for the reference instances (*redefKind*), and an indication of whether the reference instances can/should be tagged as compositions (*compositionKind*). Please note that *EReferences* carry two cardinalities: the one defined in the reference itself, and the one present in the annotation (min, max). The latter governs how many instances of the reference can be created at the next meta-level. The defined cardinality of the *EReference*, together with the flag *redefKind*, restricts the cardinality that can be assigned to the instances of the reference. This way, if the value of *redefKind* is *default*, the cardinality of the instance has to be the same as cardinality of the reference type. If *redefKind* is *restrictive*, the cardinality of the instance needs to be contained in the cardinality of the type. For example, if the cardinality of the base reference is 0..2, the instance reference can declare intervals like: 0..1, 0..2, 1..2, 1..1 and 2..2. Finally, if *redefKind* is *Anything*, the cardinality of the instance can be any value. Finally, *EAttributes* can be annotated with *MLAttribute* objects, which carry information just about the potency.

The meta-model also includes some integrity constraints, stating that the potency of the *MLModel* should be higher than or equal to the potency of their contained elements, or unbounded (while the potency of an element cannot be unbounded if the potency of the *MLModel* is not). Similarly, the potency of an *MLClass* object should be higher than or equal to the potency of the *MLAttribute* and *MLReference* objects annotating the class features. Similar to (de Lara et al., 2012), we also demand the potency of clabjects to be higher than or equal to the potency of the ancestor clabjects in inheritance hierarchies. Regarding cardinality, *BoundedElement.min* should be positive or zero, while *BoundedElement.max* should be higher than or equal to *BoundedElement.min*, or negative (to represent an unbounded cardinality). Finally, we demand the cardinality of a clabject to be wider than or equal to the interval made of the minimum cardinality of each subclabject (through the inheritance relation), and the sum of the maximum cardinalities of each subclabject (where * plus anything results in *). For example, if a clabject *A* has two subclabjects *B* and *C* with cardinalities [1..3] and [2..4], then the cardinality of *A* should be wider than or equal to [1..7] (e.g., it can be [1..7], [0..8], [0..*], etc). These constraints are provided in OCL in Appendix A.

Figure 10(a) shows the domain meta-model of the running example with multi-level annotations (shown as dark boxes attached to every meta-model element). Figure 10(b) shows a smaller excerpt (with just the class *Connector*), and the multi-level annotations shown as explicit instances of the meta-model of Figure 9. The model itself is annotated with an *MLModel* object stating that it has potency 2, and is open (so linguistic extensions are allowed at the next level below). All elements have potency 2, except attribute *Connector.delay*, which has potency 1. All elements have 0..* cardinality (and this is omitted, as it is the default) except for *Component* and *Connector*, for which we require at least one instance. Our approach supports a control of the composition nature of reference instances. This way, *Component.pTypes* instances are required to be compositions, while instances of any other reference cannot be compositions. The cardinalities of reference instances can be any, except for instances of *Connector.from*, for which cardinality equal to *Connector.from* is required. All classes are closed (instances will not support linguistic extensions), except instances of *Message*.

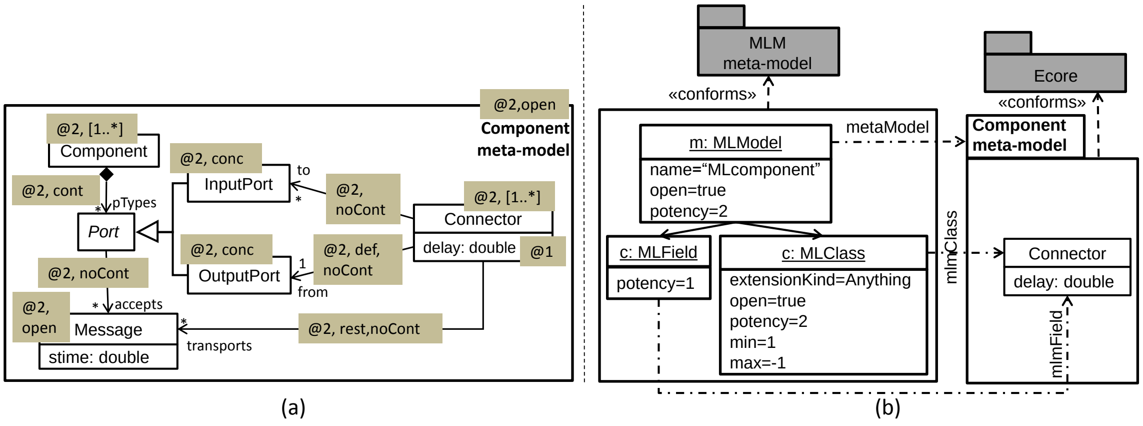


Fig. 10 – (a) Meta-model for the running example with multi-level annotations. (b) Annotations as instances of the MLM meta-model.

4.2. Deep-model instantiation through extension

Our approach replaces instantiation by inheritance, and hence we emulate clabjects with classes. When the base domain meta-model is instantiated, the created subclasses need to be annotated with multi-level annotations as well. This is illustrated in Figure 11 with the running example. Please note that we encode slot values for classes using annotations (e.g., *MChannel.delay*), and take into consideration the annotations in the parent objects as a guide to create the instances. For example, as reference *Connector.from* is annotated with default, it means that its instances (*MChannel.from*) need to have the same declared cardinality.

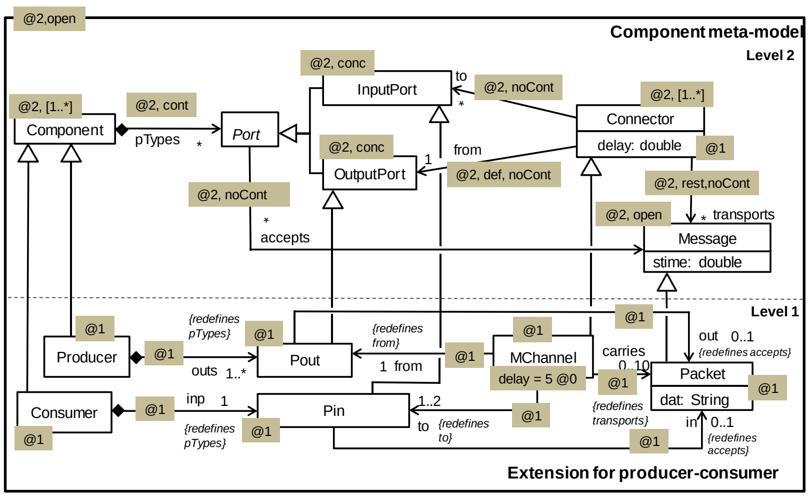


Fig. 11 - Instantiation of the running example.

Please note that, compared to an ad-hoc static approach like the one described in Section 2.2 (cf. Figure 2), our annotations permit emulating attribute values at the class level (e.g, *MChannel.delay*), while ensuring a correct emulation of instantiation with inheritance. For example, they disallow multiple inheritance from two classes (e.g, *Component* and *Port*) of the top meta-model, while controlling the specified instantiation cardinality. This solves the issues of ad-hoc static approaches identified in Table 1.

4.3. Compiling the multi-level model

Optionally, the developer can compile the multi-level model into a standard meta-model. This process removes the multi-level annotations, encoding the semantics of the different multi-level constructs in the standard meta-model, including the semantic difference between inheritance and instantiation (T. Kühne, 2009). This compilation process takes as reference the procedure described in (Guerra & de Lara, 2017), and has four steps:

- *Compilation of clabjects*: each class at level 2 or higher is set to abstract, while classes with potency 0 become abstract as well. This is necessary to avoid instantiating those clabjects at level 0. In the first case, we would be jumping levels when instantiating. In the second, we would be instantiating clabjects with potency 0.

  Figures 12(a) and (b) illustrate these two cases. In both of them, the multi-level model appears to the left of the arrow, and the compiled standard meta-model to the right. In Figure 12(a), we cannot set class *A* to concrete, as we could instantiate it at level 0, which is not allowed in the multi-level model. In such Figure, class *B* becomes concrete, because the multi-level model permits instantiation. In Figure 12(b), both classes *A* and *B* become abstract, as we cannot instantiate any of them at level 0. In our example, we set to abstract classes *Component*, *Port*, *Message*, *InputPort*, *OutputPort* and *Connector.*

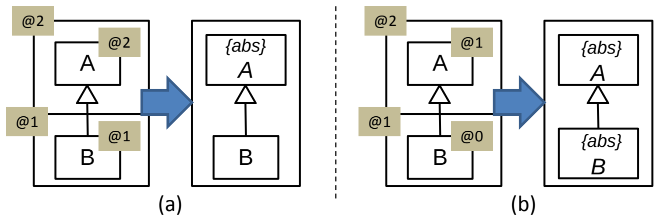


Fig. 12 - Compiling classes with potency annotations: (a) Classes with same potency and level; (b) Classes with potency 0.

- *Compilation of attributes*: attributes (with primitive data type) are just copied. However, depending on their potency, they may need to be set as static, and their values redefined. In particular, attributes whose potency is lower than the model level, need to be set as static (a class attribute). This is required to emulate that such attribute values are assigned at the class level, and is the same for all the instances of such class. Please note that, in meta-modelling system that do not support class attributes (like EMF), the value can be emulated with an OCL invariant, or as a derived attribute, but in such case, the attributes remain read-only.

  For example, *delay* has potency 1, while its owner clabject *Connector* has potency 2, and is located in a model with level 2. This way, the attribute is compiled into a static attribute, where the value is redefined in the *MChannel* subclass, as shown in Figure 13(a). In case of EMF, the class attribute value would be emulated with an OCL invariant in the context of *MChannel* (stating *self.delay=5)* or with a derived attribute. Figure 13(b) shows another case where the attribute (*stime*) has potency 2, and the owner's level is 2 as well. Therefore, it is compiled as an instance attribute.

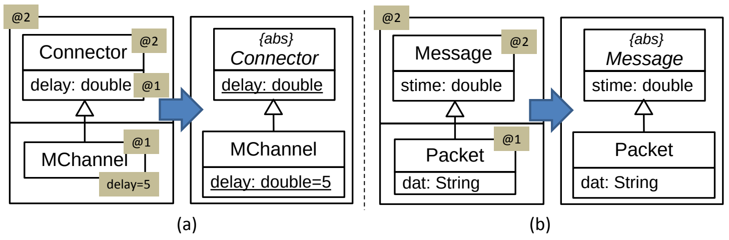


Fig. 13 - Compiling attributes: (a) Attributes with potency lower than the level; (b) Attributes with same potency as the level.

- *Compilation of references*: similar to clabjects, to avoid illegal instantiations, references at level 2 or higher are removed, while references at level 1 with potency 0 are removed as well. To avoid losing information (and to be able to write OCL constraints that consider those elements), the removed references are substituted by query operations returning the specified classes. In meta-modelling system supporting OCL, the operation code can be given in OCL.

  Figure 14 illustrates this compilation. Reference *A.r1* is removed and substituted by an abstract query operation *r1()* in the compiled meta-model. Then, such operation is overridden in *A1* and *A2* accordingly. Please note that we take advantage of covariant return in method overriding. This way, as all instances of *r1* in *A1* return objects of type *A2*, method *A1.r1()* returns a collection of *A2*. Instead, there are two instances of *r1* in *A2*, namely *r3* (with target *A1*) and *r4* (with target *A2*). As they are not the same class, we take the most specific common superclass of all of them, which in this case is *A*. As there are two instances, then *A2.r1()* returns the union of such instances. As it can be noted, the goal is to avoid instantiating *r1*, while emulating reference redefinition via operation overriding. As references *r2*, *r3* and *r4* have potency 1, they can be instantiated normally at level 0.

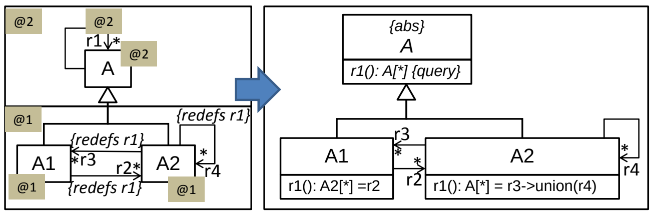


Fig. 14 - Compiling references.

- *Compilation of annotations*: the untyped clabjects created by the developer at any meta-level may contain annotations. The compilation only needs to tackle the class cardinality annotations. These produce an OCL constraint restricting the number of instances of the class. Such OCL constraint is added in the root class. This is an idiom of EMF, in which the root class is expected to be instantiated exactly once, and contain all other classes directly or indirectly through containment references. We will discuss more on root classes in Section 5.

As an example, Figure 15 shows the compilation of the multi-level model in Figure 11 using the procedure just described. Clabjects with potency 2 at level 2 are compiled into abstract classes, while their instances become their subclasses. References at level 2 are removed and replaced by abstract methods, returning objects of the appropriate class and cardinality (taken from the original references). Instances of clabjects override these methods depending on the reference instantiation. For example, class *Component* declares method *pTypes()* returning a set of *Port* objects, as it had a reference to *Port*. Class *Producer* overrides operation *pTypes* to return a set of *Pout* objects (those in reference *outs*). Should *Producer* define more references of type *pTypes*, the method would be overridden returning the common most specific class of the reference targets. Please note that, as *Connector.delay* has potency 1, it becomes a static attribute, whose initial value is set in *MChannel*.

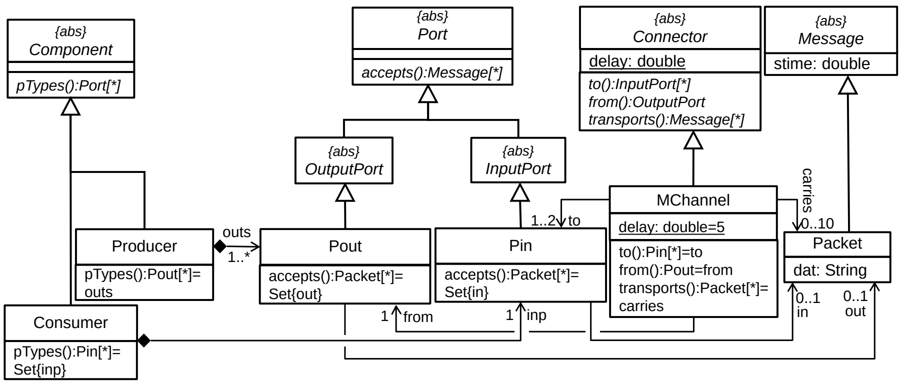


Fig. 15 - Compilation of the multi-level model in Figure 11.

In case the compilation targets EMF, it would add a root class containing all the class of level 2 (*Component*, *Port*, *Connector* and *Message*), the possible linguistic extensions at level 1 not contained in other classes, and OCL invariants taking care of the declared clabjects cardinalities (e.g., cardinality [1..*] of *Component* and *Connector*).

Please note that, while in all example Figures used in this section, the maximum level is 2, our compilation works the same for other level values. Another point is that in our approach, the compilation specifically targets the creation of a meta-model that enables building a model at level 0. However, it could be used to create models e.g., at level 1. This could be done by setting an explicit target level for compilation (as in (Guerra & de Lara, 2017)), instead of assuming level 0. This would mean that, e.g., in Figure 12, we would make concrete those classes having a potency equal to the target level plus 1. However, in this case, we could only create the instance facet of those elements, but not the type facet (e.g., we could not add additional attributes with type facet to objects), or create untyped clabjects. The idea of compilation for levels other than 0 was also sketched in (Neumayr et al., 2018).

4.4. Creating the level-0 model

As mentioned, the level-0 model can be created in two ways: (i) by instantiating the compiled meta-model, and (ii) by extending the multi-level model. The first option is preferred when compatibility with two-level frameworks is desired. For example, if the developer needs to define a model transformation to manipulate the level-0 model, then such transformation will be defined over the compiled meta-model. Interestingly, any level-0 model built in this way will be a valid instance of the original multi-level model represented using meta-model extension. The second way to create the model emulates objects with classes, and does not require compiling the multi-level model. This is the preferred option when the developer does not want to use a different tool (e.g. a two-level standard model editor) to create the level-0 model.

As an example, Figure 16 (a) depicts a level-0 model (made of just on object *m* of type *Packet*), showing how it is an instance of the multi-level model. In particular, we use «instance of» to convey that the model has been created using the compiled meta-model, and «conforms to» to convey that the model is conforming to the extension-based representation of multi-level models. Please note that the converse does not hold: there are instance models of the extension-based representation (in particular, those with an object of type *Message*) that do not conform to the compiled meta-model representation. However, those models are undesired, as in this case we would be jumping meta-levels when instantiating the *Message* class. Effectively, the compilation makes explicit the semantics of the multi-level model using standard meta-modelling facilities.

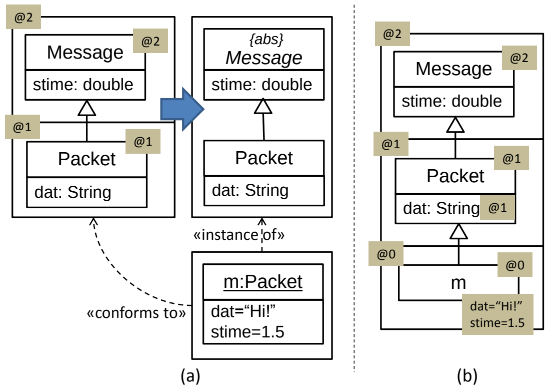


Fig. 16 - (a) Object-based representation of level 0 models; (b) Class-based representation of level 0 models.

Figure 16 (b) shows the same model expressed using a class-based representation. Hence, this approach represents clabjects at level 0 using classes. As these clabjects will frequently have instance fields, annotations are used to store their values.

5. Architecture and tool support

We have realized the described approach in a tool called TOTEM. The tool is an Eclipse plugin, freely available at http://miso.es/tools/totem.html. The tool web page contains several examples and case studies, including the running example and the "Bicycle challenge", a case study proposed in the MULTI series of workshops (see https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/#challenge). The architecture of TOTEM is shown in Figure 17.

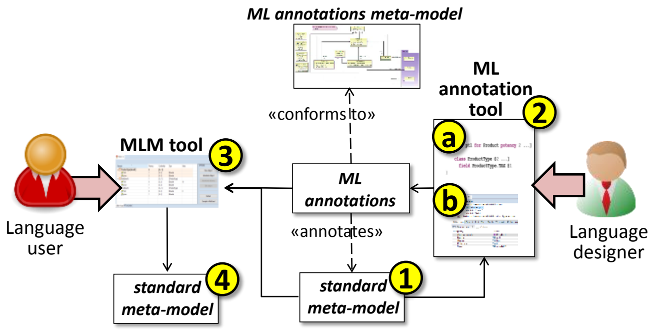


Fig. 17 - Architecture of TOTEM.

TOTEM is made of two main components (labels 2 and 3 in the Figure). The first component is the multi-level annotation tool. With this tool, an existing EMF (Steinberg et al., 2008) domain meta-model (label 1 in the Figure) is annotated with multi-level modelling annotations by the language designer. These annotations are realized as a model conforming to the meta-model shown in Figure 9. The elements of this annotation model have cross-references to the elements in the standard domain meta-model (cf. Figure 10(b)). To perform this task, the annotation tool offers two editors to the designer: a tree-based editor and a textual editor (labels 2a and b in the Figure). The annotation tool is described in Section 5.1 in more detail.

The second component of TOTEM is the multi-level modelling tool (label 3 in the Figure). This tool receives a standard meta-model with multi-level annotations and permits its instantiation at any depth, including level 0, according to the defined annotations. Additionally, TOTEM can produce a standard meta-model using the compilation procedure in Section 4.3 (label 4 in the Figure). This way, standard EMF tooling can be used to instantiate the meta-model, or to define model transformations over it. This second component will be described in more detail in Section 5.2.

TOTEM supports two approaches to create a full multi-level model. In the first one, an existing standard EMF meta-model is annotated with multi-level annotations using the annotation tool, and then instantiated using the multi-level modelling tool. This scenario enhances compatibility with EMF, and facilitates migration into a multi-level architecture. In the second scenario, the language designer does not start from an existing EMF meta-model, but instead uses the multi-level modelling tool to create the top-level model of the language. We will discuss how this is achieved in Section 5.2.

For simplicity, and to facilitate comprehension, in this Section we will use as example the multi-level model depicted in Figure 18. The model is a well-known example used in the multi-level literature (Atkinson & Kühne, 2002), where the idea is being able to create product types (like *Book* or *Food*), each with a different *tax*, and then instantiate these types at level 0, and assign them a concrete *price*.

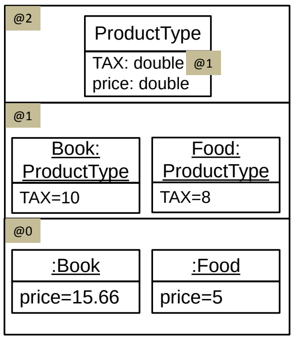
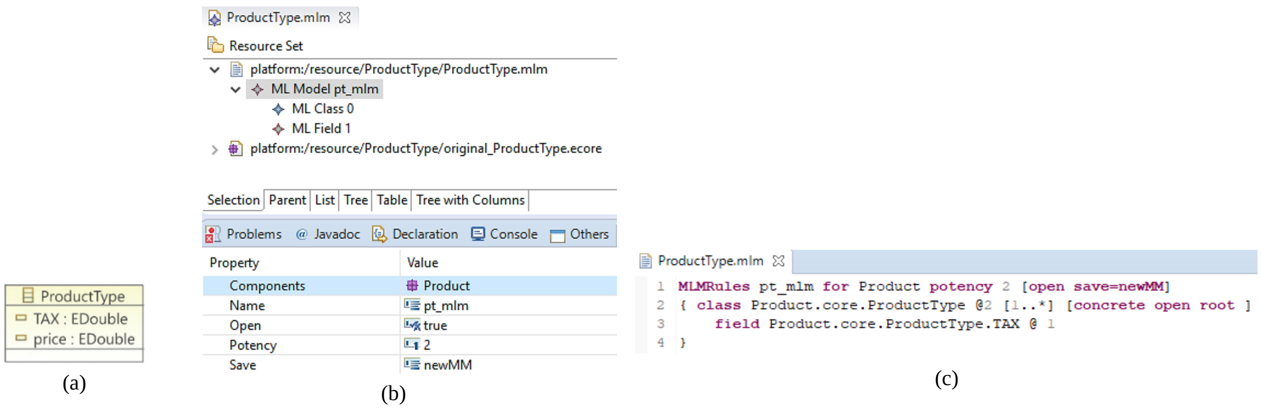


Fig. 18 – Multi-level example used in this section.

5.1. The multi-level annotation tool

The multi-level annotation tool is useful for the case in which we base our multi-level model on an existing EMF meta-model. The annotations can be specified using a tree-based editor or using a domain-specific textual language we have created using Xtext. As an example, Figure 19(a) shows a standard Ecore meta-model corresponding to the level 2 of Figure 18, while Figures (b) and (c) show the annotations we have added to such meta-model using the tree editor and the textual editor. In both cases, the meta-model itself is set potency 2, class *ProductType* is set potency 2, while attribute *TAX* is set potency 1. Attribute *price* is not set an explicitly potency, and so it obtains the potency of its enclosing class (potency 2, cf. Figure 17).

Additionally, Figure 19(c) shows some more details that are not visible in Figure 19(b). In particular, it declares the meta-model as *open*, which means that we can include clabjects with no type in the level below. It also sets a persistence flag (*save*) indicating that, upon instantiation, the resulting model is to be persisted in a separate file (*newMM*). The other alternative for persistence is saving in the same file. Class *ProductType* is assigned an instantiability interval of [1..*] (so its instantiation is mandatory). It is also tagged as *concrete* (so that instances of this class cannot be abstract), *open* (so that instances can add untyped fields and references), and *root*. The latter flag is used to indicate that this will be the root class. Having a root class is a common idiom in EMF, so that frequently, models contain just one object of the root class, which contains directly or indirectly all other objects. Several classes can be tagged as root. In our implementation, the compilation adds an additional class acting as the root container, which contains the instances of the classes tagged as root. Please note that we omitted the persistence and root flags from the meta-model in Figure 9 as they are implementation details.



platform:/resource/ProductType/ProductType.mlm
ML Model pt_mlm
ML Class 0
ML Field 1
platform:/resource/ProductType/original_ProductType.ecore

Property	Value
Components	Product
Name	pt_mlm
Open	true
Potency	2
Save	newMM

```
MLMRules pt_mlm for Product potency 2 [open save=newMM]
{ class Product.core.ProductType @2 [1..*] [concrete open root ]
    field Product.core.ProductType.TAX @ 1
}
```

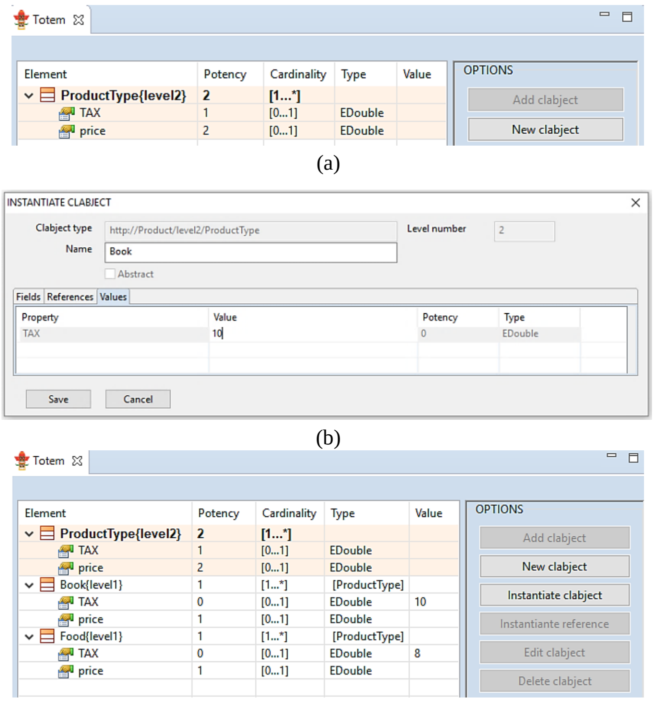
Fig. 19 - (a) Example standard meta-model; (b) Multi-level annotations using the tree-based editor; (c) Textual DSL.

5.2. The Multi-level modelling tool

Once the multi-level annotations are specified, the annotated meta-model can be instantiated with the multi-level modelling tool. Figure 20 shows some screenshots of the tool being used to instantiate the annotated meta-model of Figure 18. Its main interface (a) is organized in three sections. The left section shows all the elements of the base meta-model (clabjects, fields and references), which can be instantiated according to their annotations. In the columns to the right, several features appear, like the potency, cardinality, type (reference type for associations, or datatype for attributes), and value (target class for reference or value for attributes).

The right section of the interface (panel labelled "OPTIONS") contains buttons to instantiate clabjects and references, create untyped clabjects or edit existing clabjects. Figure 20(b) shows the dialog used to instantiate a clabject. It allows setting the value of attributes, creating new fields if the clabject type is open, and instantiate references as appropriate. Figure 20(c) shows the tool after creating two instances of *ProductType*. In both such instances we can assign a value to TAX (as it has potency 0). Moreover, attribute *price* (with potency 1) is also visible, to permit assigning it a default value.

TOTEM supports an arbitrary number of meta-levels, and also unbounded potency. To support the scenario where the designer does not start from an existing EMF meta-model, we use a generic annotated meta-model with no classes. This meta-model just has unbounded potency and is open. This means that we can instantiate it to create a model of some arbitrary potency, and untyped clabjects within it.



Element	Potency	Cardinality	Type	Value
ProductType{level2}	2	[1...*]		
TAX	1	[0...1]	EDouble	
price	2	[0...1]	EDouble	

(a)

Clabject type: http://Product/level2/ProductType
Level number: 2
Name: Book
Abstract
Fields | References | Values

Property	Value	Potency	Type
TAX	10	0	EDouble

(b)

Element	Potency	Cardinality	Type	Value
ProductType{level2}	2	[1...*]		
TAX	1	[0...1]	EDouble	
price	2	[0...1]	EDouble	
Book{level1}	1	[1...*]	[ProductType]	
TAX	0	[0...1]	EDouble	10
price	1	[0...1]	EDouble	
Food{level1}	1	[1...*]	[ProductType]	
TAX	0	[0...1]	EDouble	8
price	1	[0...1]	EDouble	

(c)

Fig. 20 - Using the multi-level modelling tool: (a) Main GUI; (b) Instantiating a clabject; (c) Instantiating a model.

5.3. Extracting a standard meta-model

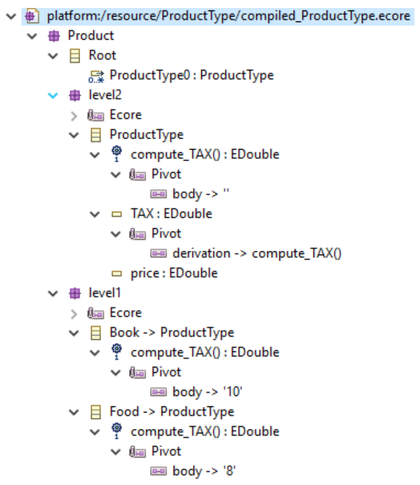
TOTEM permits modelling the level 0 within the tool, or it can produce a standard meta-model according to the procedure described in Section 4.4. Figure 21(a) shows the generated meta-model, while Figure 21(b) depicts a (level 0) model created with the standard tree editor of EMF. Please note that TAX has potency 1, and then receives values at level 1. This is encoded as a derived attribute, which takes the value using operation *compute_TAX*(), which is overridden to yield the appropriate values for clabjects *Book* and *Food*. In addition, the tool generates a root class containing all classes declared as root (*ProductType* in this case). This is used in the model of Figure 21(b) as the container for the model objects.

6. Evaluation

In this section, we provide both an experimental evaluation and an analytic one. Our aim is answering two research questions: "*RQ1: is TOTEM able to produce solutions comparable to state-of-the-art multi-level modelling tools?*", and to "*RQ2: what are the advantages of TOTEM with respect to state-of-the-art multi-level modelling tools?*". For this purpose, we fully develop the running example, showing its realization with TOTEM and then comparing its realization with an implementation using MetaDepth and Melanee (Section 6.1). Second, we perform an analytical evaluation of TOTEM in comparison with current multi-level modelling tools (Section 6.2).

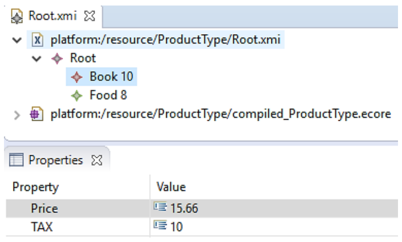
6.1. Empirical evaluation: comparing the running example using TOTEM, MetaDepth and Melanee

In this section we compare a TOTEM model of the running example, with an encoding using MetaDepth and Melanee. They have been selected as representative state-of-the-art multi-level modelling tools (they are among the oldest tools supporting both potency and levels). Our aim is to measure the complexity of both solutions, in terms of metrics like size, and on the multi-level features available in each tool. This experiments aims at answering *RQ1* and *RQ2*.



```
platform:/resource/ProductType/compiled_ProductType.ecore
  Product
    Root
      ProductType0 : ProductType
    level2
      Ecore
      ProductType
        compute_TAX() : EDouble
          Pivot
            body -> ''
        TAX : EDouble
          Pivot
            derivation -> compute_TAX()
        price : EDouble
    level1
      Ecore
      Book -> ProductType
        compute_TAX() : EDouble
          Pivot
            body -> '10'
      Food -> ProductType
        compute_TAX() : EDouble
          Pivot
            body -> '8'
```

(a)



```
Root.xmi
platform:/resource/ProductType/Root.xmi
  Root
    Book 10
    Food 8
platform:/resource/ProductType/compiled_ProductType.ecore
```

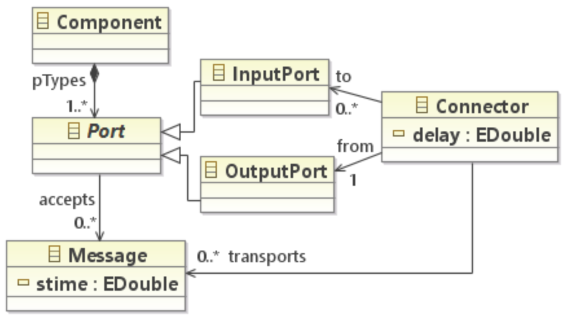
Properties

Property	Value
Price	15.66
TAX	10

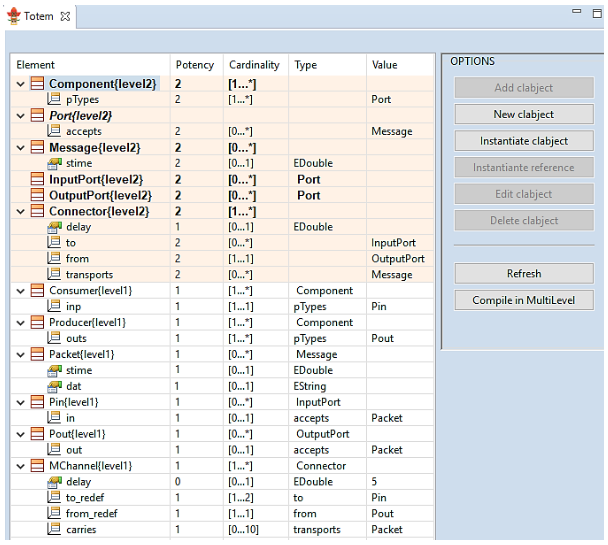
(b)

Fig. 21 - (a) Compiled standard meta-model; (b) A model at level 0 created using the standard EMF tree editor.

Figure 22 (a) shows the EMF meta-model we started with to model level 2, while Figure 22 (b) shows TOTEM being used to create the model at level 1.



(a)



Element	Potency	Cardinality	Type	Value
Component{level2}	**2**	**[1...*]**		
pTypes	2	[1...*]		Port
Port{level2}				
accepts	2	[0...*]		Message
Message{level2}	**2**	**[0...*]**		
stime	2	[0...1]	EDouble	
InputPort{level2}	**2**	**[0...*]**	**Port**	
OutputPort{level2}	**2**	**[0...*]**	**Port**	
Connector{level2}	**2**	**[1...*]**		
delay	1	[0...1]	EDouble	
to	2	[0...*]		InputPort
from	2	[1...1]		OutputPort
transports	2	[0...*]		Message
Consumer{level1}	1	[1...*]	Component	
inp	1	[1...1]	pTypes	Pin
Producer{level1}	1	[1...*]	Component	
outs	1	[1...*]	pTypes	Pout
Packet{level1}	1	[0...*]	Message	
stime	1	[0...1]	EDouble	
dat	1	[0...1]	EString	
Pin{level1}	1	[0...*]	InputPort	
in	1	[0...1]	accepts	Packet
Pout{level1}	1	[0...*]	OutputPort	
out	1	[0...1]	accepts	Packet
MChannel{level1}	1	[1...*]	Connector	
delay	0	[0...1]	EDouble	5
to_redef	1	[1...2]	to	Pin
from_redef	1	[1...1]	from	Pout
carries	1	[0...10]	transports	Packet

(b)

Fig. 22 - (a) Top-level model for the running example as an Ecore meta-model; (b) Levels 2 and 1 in the TOTEM GUI.

Listing 1 shows the running example using the textual syntax of MetaDepth. Listing 1(a) shows level 2, where clabjects *Component*, *Port*, *InputPort*, *OutputPort* and *Connector* are defined. Clabjects with no ontological type are declared using the keyword *Node*, and clabject cardinalities are specified next to the clabject name (e.g., [1..*] for *Connector*). If no cardinality is specified, then [0..*] is assumed. By default, instances of a model may include clabjects with no type (i.e., linguistic extensions), and similarly, instances of a clabject can be added fields with no type. However, this can be controlled prefixing models and clabjects with the keyword *strict*. Potency is indicated after the "@" symbol. If a clabject or field does not explicitly declare potency, it takes the one of its container.

Listing 1(b) shows an instance of the *ComponentSystem* model, with the producer-consumer system of the running example. Instantiating a clabject requires prefixing the clabject name with its type (e.g., "*Component Producer"* creates a clabject with identifier *Producer*, of type *Component*). At this level some clabjects assign values to fields with potency 0 (e.g., *delay* = 5), and as all clabjects are extensible, we can declare new fields (e.g., *dat: String*). Instantiating a reference requires declaring its target clabject type (e.g., *out : Packet*), the cardinality (e.g., *[0..1]*) and the type of the parent reference (e.g., *{accepts}*). Finally, Listing 1(c) shows a level 0 model.

Overall, both solutions are similar, leading to very similar size metrics. In both cases, level 2 has 6 clabjects, 5 references, 2 fields and 2 inheritance relations. Level 1 has 6 clabjects in both cases, 9 fields in case of MetaDepth and 10 in case of TOTEM. The difference concerns attribute *stime*, which is shown in *Packet* at level 1 in TOTEM, so that it can be assigned a default value, if desired. In MetaDepth the attribute would only need to be explicitly declared within *MChannel* if an initial value is to be given. Therefore, the size of both specifications is similar. Please note that we count size as it is perceived by the designer, hence we do not include the annotation model internally required by TOTEM (cf. Figure 10(b)) in the count of the model size (just like for MetaDepth, we do not count internal Java objects, but only count those elements that the designer actually deals with).

```
Model ComponentSystem@2
  Node Component [1..*] {
    pTypes : Port[*];
  }
  abstract Node Port {
    accepts : Message[*];
  }
  Node InputPort : Port{}
  Node OutputPort : Port{}
  Node Message {
    stime: double;
  }
  Node Connector [1..*] {
    delay@1 : double;
    from    : OutputPort;
    to      : InputPort[*];
    transports : Message[*];
  }
}
```

(a)

```
ComponentSystem ProducerConsumer {
   Component Producer{
      outs : Pout[1..*] {pTypes};
   }
   Component Consumer{
      inp  : Pin {pTypes};
   }
   OutputPort Pout{
      out : Packet[0..1]{accepts};
   }
   InputPort Pin{
      in : Packet[0..1]{accepts};
   }
   Connector MChannel {
      delay = 5;
      fromp   : Pout {from};
      top     : Pin [1..2] {to};
      carries : Packet[0..10]{transports};
   }
   Message Packet {dat : String;}
}
```

(b)

```
ProducerConsumer aModel {
   Producer p {
     outs = [po];
   }
   Pout po {
     out = [pk];
   }
   Packet pk {
     dat = "Hi!";
     stime = 1.5;
   }
   MChannel mc {
     fromp = po;
     top = [pi1, pi2];
   }
   Pin pi1;
   Pin pi2;
   Consumer c1 { inp = [pi1]; }
   Consumer c2 { inp = [pi2]; }
}
```

(c)

Listing 1 - Running example in MetaDepth: (a) Component meta-model (level 2); (b) Producer-consumer meta-model (level 1); (c) A Producer-consumer model (level 0).

A crucial difference in the modelling approach in both cases is that, when elements are instantiated in TOTEM, the fields are automatically instantiated as well, and shown in the GUI. Because MetaDepth has a textual syntax, the developer needs to explicitly write such field instances, which requires more effort.

Both approaches are similar in that both enable expressing clabjects cardinalities, and controlling whether linguistic extensions are allowed. On a more detailed comparison, we observe that MetaDepth lacks composition relations, while compositions are native in EMF, and therefore TOTEM supports them. Hence, an advantage of TOTEM is that, by reusing the EMF for multi-level modelling, we immediately acquired its meta-modelling facilities (e.g., opposite references, composition references, enumerations). Moreover, TOTEM supports the specification of whether the instances of a reference need to be compositions as well. A faithful modelling of the running example in MetaDepth would require OCL constraints emulating the composition semantics. MetaDepth does not require root classes, as models are first class entities in MetaDepth. In contrast, being based on EMF, TOTEM requires marking the classes that are to be taken as roots (which produces an additional class when compiling the meta-model). Finally, TOTEM has flags to control the allowed cardinalities of the instances of a given relation (like default or restrictive), while this requires OCL constraints in MetaDepth.

Figure 23 shows the running example built using Melanee. For comprehensibility we have labeled the three levels with "Level 2", "Level 1" and "Level 0", but the tool indexes the top level as level 0.

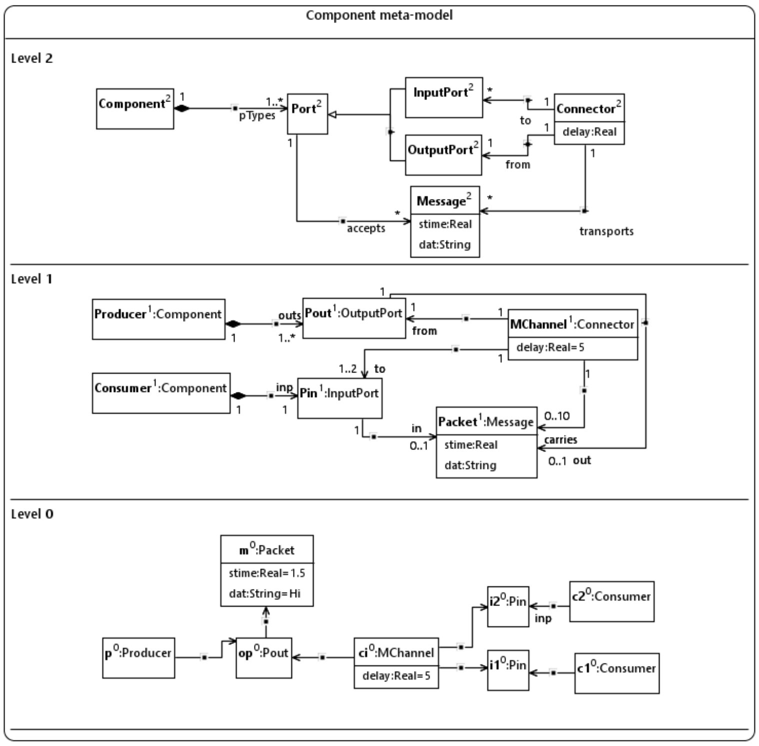
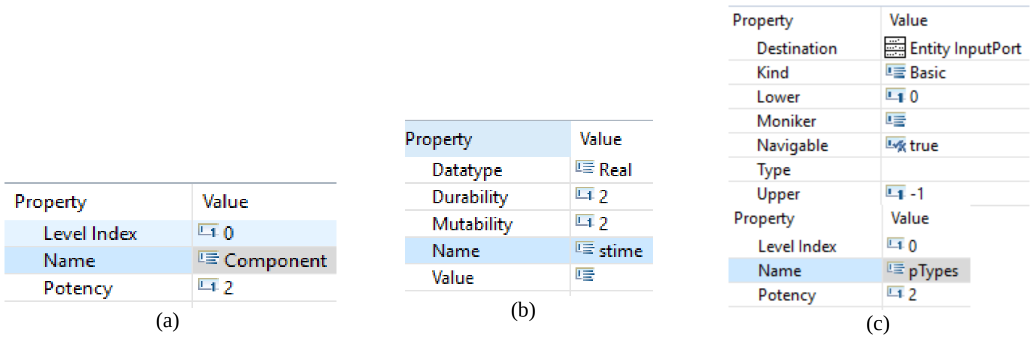


Fig. 23 – Running example in Melanee.

Melanee provides a palette for the graphic construction of the multi-level model. Figure 24 shows some of the dialog boxes used by Melanee to specify values of the properties of some elements of the multi-level model. Figure 24 (a) shows the specification of the properties of the *Component* clabject with potency 2 and level 2. The property *Level Index* = 0, specifies the level of the model in which the clabject is located. When an entity is created in the first level of the model (the top), the property will have value 0, which is assigned automatically. As the clabject is instantiated, the value of the property increases by 1.



(a)

Property	Value
Level Index	0
Name	Component
Potency	2

(b)

Property	Value
Datatype	Real
Durability	2
Mutability	2
Name	stime
Value	

(c)

Property	Value
Destination	Entity InputPort
Kind	Basic
Lower	0
Moniker	
Navigable	true
Type	
Upper	-1
Property	Value
Level Index	0
Name	pTypes
Potency	2

Fig. 24: Properties of some elements in the solution using Melanee: (a) clabject; (b) attribute; (c) reference.

Figure 24 (b) shows the specification of the properties of the *stime* attribute of the *Message* clabject with potency 2 at level 2 (see Figure 21). In Melanee attributes declare both a durability (in how many later levels the attribute may exist), and a mutability (how many levels the value of an attribute can change). The *Value* property is used to assign an initial value.

Figure 24 (c) shows the specification of the properties of the reference (called *connection* in *Melanee*) *pTypes* of clabject *Component*. The dialog permits assigning a target clabject, a kind (basic, composition or aggregation), a cardinality (*lower*, *upper*), a moniker (a role name), a navigability, and an optional type (at the higher meta-level). Just like clabjects, associations can specify the level of the model in which the connection is located, the name and the potency.

Once all elements at level 2 have been included and configured, they can be instantiated at level 1, through the tool palette. The value of the potencies of the elements (clabject, attributes and references) is automatically controlled by the tool. The assignment of values to the fields with potency 0 (for example, *MChannel.delay = 5*) of the clabjects has to be done manually through the properties window (Figure 24 (b)). Because all clabjects are extensible, new attributes can be declared (for example, *dat: String*) through the tool palette. The instantiation of a reference requires that the origin and destination clabject have been previously instantiated at the level above. In such case, it is possible to redefine the cardinality of the instantiated references, as well as assign them a new name.

In general, the solutions using Melanee and TOTEM are similar, which leads to very similar size metrics. In both cases, level 2 has 6 clabjects, 5 references, 3 fields (because the *dat* field that is added in level 1 is also added automatically in level 2) and 2 inheritance relationships. Level 1 has the same number of elements in both cases: 6 clabjects in both cases, 7 references and 3 fields.

The main difference concerns the support for clabjects cardinalities and abstract clabjects in TOTEM, which are not supported in Melanee. Melanee can emulate abstract clabjects adding them potency 0, but this was not possible with clabject *Port* due to the potency required for the subclabjects *InputPort* and *OutpuPort*. Additionally, Melanee by default allows creating untyped clabjects (linguistic extensions) at any level of the model, while in TOTEM this can be controlled. The two tools support composition relationships. However, TOTEM allows specifying whether the instances of a reference may or may not also be compositions. Melanee also supports aggregation references, and does not require root classes, since the models are first class entities.

Table 3 shows a summary of the comparison of the approaches with respect to the example.

Table 3 – Comparison of the solutions using MetaDepth, Melanee and TOTEM.

Tool / **Aspect**	**MetaDepth**	**Melanee**	**TOTEM**
Size of modeling elements	*Level 2:* 6 clabjects, 5 refs, 2 fields and 2 inheritance relationships. *Level 1:* 6 clabjects, 7 refs, 2 fields.	*Level 2:* 6 clabjects, 5 refs, 3 fields, 2 inheritance relationships. *Level 1:* 6 clabjects, 7 refs, 3 fields.	*Level 2:* 6 clabjects, 5 refs, 2 fields, 2 inheritance relationships. *Level 1:* 6 clabjects, 7 refs, 3 fields.
Specification	Textual	Multiple formats and notations	Tabular
Compositions	No	Yes (also aggregations)	Yes (and controlling composition of instances)
Reference cardinalities	Yes	Yes	Yes, two types of cardinality for references: 1) to specify how many instances of the reference can be created at the next meta-level, 2) the one in the reference itself, which is a template for the cardinality of instances.
Clabject cardinalities	Yes	No	Yes
Clabject abstractness	Yes	No	Yes
Control of linguistic extensions	Yes	No	Yes
Deep instantiation mechanism	Potency, leap potency	Potency, mutability, durability	Potency
Unlimited potency	Yes	Yes	Yes
Requires defining root class	No	No	Yes

Overall, regarding the research question *RQ1*, the solution using TOTEM is equal in size to the solutions using state-of-the-art multi-level modelling tools, like MetaDepth and Melanee. Regarding *RQ2*, TOTEM includes advanced constructs to control compositions and reference cardinalities, which are not present in MetaDepth; and clabject cardinalities, abstractness and the control of composition in reference instances, which are not present in Melanee. However, the main advantage of TOTEM is the enhanced compatibility with two-level modelling. First, it is possible to start the multi-level model with a standard Ecore meta-model (cf. Figure 22(a)), which is not possible in MetaDepth or Melanee. Second, TOTEM supports the possibility to export the multi-level model into a standard meta-model, and then use standard EMF tooling to edit the level 0 model, or to define model manipulations using standard transformation languages, like ATL or Acceleo.

Threats to validity. Regarding internal validity (confounding factors that may influence the results), we have only used one example for the comparison. However, the example is substantial, exercising many of the features of both tools. For external validity (generalizability of results), we have only compared with MetaDepth and Melanee. While these are two of the most well-established multi-level modelling tools available today (and two of the oldest ones supporting potency and levels), generalization of the answers to the research questions would require comparison with other tools, e.g., the ones analysed in Section 3.2. An example-based comparison would run the risk of proposing non-optimal solutions (as we are not the authors of those tools). For this purpose, we have conducted an analytical evaluation in Section 6.2, comparing with the tools analysed in Section 3.2.

6.2. Analytical evaluation

In this section, we report on an analytical evaluation, comparing the features of TOTEM with those of other multi-level modelling tools. For this purpose, we base our analysis on the features extracted in Table 2, which are summarized in Table 4.

As the table shows, our approach is based on meta-model extension, a unique approach among the analysed approaches in Table 2. This approach embeds the multi-level model within an Ecore meta-model, presenting advantages regarding compatibility with the EMF standard, as we can directly reuse existing Ecore meta-models.

Table 4 - Features of TOTEM.

Feature	TOTEM
Mechanism	Meta-model extension
Instance characterization	Deep (potency, star potency)
Extension	Yes (objects, fields)
Typing	Non-Strict (extension, no-mix) Linguistic, ontological
Meta-Levels	Level-adjuvant, extensible, syntactic, semantic
Constraints	Standard OCL, but only affecting level-0 models
Purpose	MDE
Support for management languages	Yes, any available for standard EMF
Compatibility with two-level modelling and tools	Yes, both as a starting top-level meta-model and for editing level 0 models

TOTEM supports deep instance characterization through (clabject and field) potency. Additionally, it also supports star potency, which permits an unbounded instantiation depth, and a number of meta-levels that does not need to be fixed a priori. The approach supports linguistic extensions, as both meta-models and clabjects can be tagged as open. As the multi-level model is embedded within an Ecore meta-model, constraints can be added using the standard OCL. However, such constraints can only apply to level 0. TOTEM has a dual ontological/linguistic typing. On the one hand, we use Ecore enriched with the multi-level annotations as the linguistic meta-model. On the other, ontological typing is encoded within the meta-model, using inheritance relations.

The purpose of TOTEM is its use within MDE solutions, and for this purpose it has been built over EMF, as an Eclipse plugin. This decision brings two benefits. First, it receives all meta-modelling facilities of Ecore (e.g., opposite references, composition references, a rich set of datatypes, packages). Second, it is compatible with all model management languages of the EMF ecosystem, including ATL, Epsilon or Acceleo. Typically, the model manipulations will affect the level 0 models, but we can use these standard languages to manipulate the meta-model level as well. The compatibility with standard two-level modelling and tools is high, because the modelling process can start by annotating an existing standard meta-model, and the tool can produce a standard meta-model at the end, which can be instantiated using standard EMF tooling. Please note that an ad-hoc solution based on promotion transformations requires one transformation to instantiate each level below the top one. Instead, by relying on inheritance, we need just one compilation if we want to instantiate level 0 using a standard editor.

Overall, we can conclude that the approach of TOTEM is unique in current tooling. It supports most advanced constructs and requirements of useful multi-level modelling tools (deep instance characterization mechanisms, extensibility, extensible meta-levels, and constraints). Further answering *RQ2*, and compared to other existing approaches, it brings an increased compatibility with standard two-level modelling and tools, both to start a multi-level model using an existing Ecore meta-model, or to instantiate and manipulate the level 0 using standard tools.

7. Conclusions and future work

In this paper we have discussed the advantages of multi-level modelling in the scenarios where the type-object pattern or some of its variants arise. We have reviewed different approaches to multi-level modelling, characterizing their design space through a feature model. We have used this model as a basis to compare some representative multi-level modelling tools. From this review, we noticed a lack of integration with standard meta-modelling technologies – also observed by other members of the multi-level community – and argued that better compatibility with standards could lead to a wider adoption of multi-level modelling, and the benefits that it brings in terms of reduced accidental complexity (Atkinson & Kühne, 2008). To improve this situation, we have proposed a novel approach to multi-level modelling, which emulates clabjects with classes and instantiation with inheritance. We have realized these ideas in the TOTEM tool, and shown its benefits through a case study.

In the future, we plan to improve different aspects of the tool, e.g., creating a graphical editor for meta-modelling, integrating the catalogue of multi-level refactorings proposed in (de Lara & Guerra, 2018), and achieving better integration with model management languages. In particular, we plan to make possible the evaluation of standard OCL at levels different from level 0, which is currently a limitation of our approach. This would require rewriting the OCL expression to convert navigations using instantiation into navigations using inheritance (e.g., *Component.allInstances()* would be rewritten into *EClass.allInstances()->select(c|c.allSuperTypes()->includes(cmp|cmp.name='Component'))*).

Another avenue of research is taking advantage of the use of inheritance to emulate more flexible ways of instantiation, in the style of (de Lara & Guerra, 2017). In this sense, The MOF Support for Semantic Structures (SMOF) (SMOF, 2013) observes some rigidities in the typing as currently realized by MOF (MOF, 2016), for example to represent objects holding several classifiers (e.g., a person object that is classified as both *Author* and *Reviewer* in case of having authored and reviewed articles). While using the standard instantiation support given by MOF rules out this possibility, our approach based on emulating instantiation with inheritance opens the door to support multiple classification. This way, we plan to investigate ways to specify and control multiple classification with our approach.

Finally, we plan to expand the evaluation with comparison to other multi-level tools. However, we believe this should be a joint effort of the whole multi-level modelling community.

REFERENCES

Almeida, J. P. A., Frank, U., & Kühne, T. (2018). Multi-Level Modelling (Dagstuhl Seminar 17492). Dagstuhl Reports, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. pp. 18-49. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Atkinson, C. (1997). Meta-modeling for distributed object environments. Proceedings of the 1997 1st International Enterprise Distributed Object Computing. pp. 90-101. Gold Coast, Australia: IEEE Computer Society.

Atkinson, C. (1998). Supporting and applying the UML conceptual framework. International Conference on the Unified Modeling Language. Lecture Notes in Computer Science. Vol. 1618. pp. 21-36. Mulhouse; France: Springer.

Atkinson, C., & Gerbig, R. (2016). Flexible deep modeling with Melanee. Workshop on Modeling 2016. Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI). Vol. P255. pp. 117-121. Karlsruhe; Germany.

Atkinson, C., Gerbig, R., & Fritzsche, M. (2015). A multi-level approach to modeling language extension in the enterprise systems domain. Information Systems. Vol. 54. pp. 289-307.

Atkinson, C., Gerbig, R., & Kühne, T. (2014). Comparing multi-level modeling approaches. Workshop on Multi-Level Modelling, MULTI 2014, Co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2014. pp. 53-61. Valencia; Spain: CEUR-WS.

Atkinson, C., Gerbig, R., & Kühne, T. (2015). A unifying approach to connections for multi-level modeling. (2015) 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, MODELS 2015. pp. 216-225. Ottawa;Canada: IEEE.

Atkinson, C., Gerbig, R., & Tunjic, C. V. (2015). Enhancing classic transformation languages to support multi-level modeling. Software & Systems Modeling. Vol. 14(2). pp. 645-666.

Atkinson, C., & Kühne, T. (2001). The essence of multilevel metamodeling. 4th International Conference on Unified Modeling Language, UML 2001. Vol. 2185. pp. 19-33. Springer Verlag.

Atkinson, C., & Kühne, T. (2002). Rearchitecting the UML infrastructure. ACM Transactions on Modeling and Computer Simulation (TOMACS). Vol. 12(4). pp. 290-321.

Atkinson, C., & Kühne, T. (2008). Reducing accidental complexity in domain models. Software & Systems Modeling. Vol. 7(3). pp. 345-359.

Atkinson, C., & Kühne, T. (2017). On Evaluating Multi-Level Modeling. 2017 MODELS Satellite Event. Vol. 2019. pp. 274-277. Austin; United States: CEUR-WS.

Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., & Wąsowski, A. (2013). Partial instances via subclassing. International Conference on Software Language Engineering. pp. 344-364. Springer.

Balaban, M., Khitron, I., Kifer, M., & Maraee, A. (2018). Multilevel modeling: What's in a level? A position paper. CEUR Workshop Proceedings. 2018 MODELS Workshops. Vol. 2245. pp. 693-697. Copenhagen; Denmark.

Barr, M., & Wells, C. (1990). Category Theory for Computing Science. Vol. 49. pp. 350. Prentice Hall New York.

Borgida, A., Mylopoulos, J., & Wong, H. K. T. (1984). Generalization/Specialization as a Basis for Software Specification. In M. L. Brodie, J.

Mylopoulos & J. W. Schmidt (Eds.), *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages* pp. 87-117. New York, NY: Springer New York.

Carvalho, V. A., & Almeida, J. P. A. (2018). Toward a well-founded theory for multi-level conceptual modeling. Software & Systems Modeling. Vol. 17(1). pp. 205-231.

Clark, T., & Frank, U. (2018). Multi-Level Constraints. MoDELS Workshops 2018. pp. 103-117. Copenhagen; Denmark.

Clark, T., Sammut, P., & Willans, J. (2015a). Applied metamodelling: A foundation for Language Driven Development (Third Edition). CoRR abs/1505.00149. Vol. 1. pp. 228.

Clark, T., Sammut, P., & Willans, J. (2015b). Super-languages: Developing languages and applications with XMF (Second Edition). arXiv preprint arXiv:1506.03363. Vol. 1. pp. 420.

Da Silva, A. R. (2015). Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structures. Vol. 43. pp. 139-155.

de Lara, J., & Guerra, E. (2010). Deep meta-modelling with Metadepth. Lecture Notes in Computer Science. Vol. 6141. pp. 1-20. Málaga,Spain: Springer.

de Lara, J., & Guerra, E. (2017). A posteriori typing for model-driven engineering: Concepts, analysis, and applications. ACM Transactions on Software Engineering and Methodology. Vol. 25(4). pp. 156-165.

de Lara, J., & Guerra, E. (2018). Refactoring multi-level models. ACM Transactions on Software Engineering and Methodology (TOSEM). Vol. 27(4). pp. 1-56.

de Lara, J., Guerra, E., Cobos, R., & Moreno-Llorena, J. (2012). Extending deep meta-modelling for practical model-driven engineering. The Computer Journal. Vol. 57(1). pp. 36-58.

de Lara, J., Guerra, E., & Sánchez, J. (2014). When and how to use multilevel modelling. ACM Transactions on Software Engineering and Methodology (TOSEM). Vol. 24(2). pp. 12.

Diskin, Z. (2005). Mathematics of Generic Specifications for Model Management, I *Encyclopedia of Database Technologies and Applications* pp. 351-358: IGI Global.

Drivalos, N., Kolovos, D. S., Paige, R. F., & Fernandes, K. J. (2008). Engineering a DSL for software traceability. International Conference on Software Language Engineering. pp. 151-167. Springer.

Fonseca, C. M. (2017). ML2: An Expressive Multi-Level Conceptual Modeling Language. pp. 1-93 Master's thesis, Federal University of Espírito Santo, Brazil.

Fonseca, C. M., Almeida, J. P. A., Guizzardi, G., & Carvalho, V. A. (2018). Multi-level conceptual modeling: from a formal theory to a well-founded language. Lecture Notes in Computer Science. Vol. 11157 LNCS. pp. 409-423.

Frank, U. (2014). Multilevel modeling. Toward a new paradigm of conceptual modeling and information systems design. Business & Information Systems Engineering. Vol. 6(6). pp. 319-337.

Garlan, D., & Shaw, M. (1993). An introduction to software architecture *Advances in software engineering and knowledge engineering* pp. 1-39: World Scientific.

Gerbig, R., Atkinson, C., de Lara, J., & Guerra, E. (2016). A Feature-based comparison of Melanee and Metadepth. 3rd International Workshop on Multi-Level Modelling, MULTI 2016. pp. 25-34. Saint-Malo; France.

Guerra, E., & de Lara, J. (2017). Automated analysis of integrity constraints in multi-level models. Data & Knowledge Engineering. Vol. 107. pp. 1-23.

Guerra, E., & de Lara, J. (2018). On the Quest for Flexible Modelling. 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018. pp. 23-33. Copenhagen; Denmark: ACM.

Henderson-Sellers, B., Clark, T., & Gonzalez-Perez, C. (2013). On the search for a level-agnostic modelling language. Lecture Notes in Computer Science. Vol. 7908 LNCS. pp. 240-255.

Hinkel, G. (2018). Using structural decomposition and refinements for deep modeling of software architectures. Software and Systems Modeling. Article in Press. Vol. 2018. pp. 1-33.

Igamberdiev, M., Grossmann, G., Selway, M., & Stumptner, M. (2018). An integrated multi-level modeling approach for industrial-scale data interoperability. Software & Systems Modeling. Vol. 17(1). pp. 269-294.

Jácome, S., & de Lara, J. (2018). Controlling Meta-Model Extensibility in Model-Driven Engineering. IEEE Access. Vol. 6. pp. 19923-19939.

Jarke, M., Gallersdörfer, R., Jeusfeld, M. A., Staudt, M., & Eherer, S. (1995). ConceptBase - A deductive object base for meta data management. Journal of Intelligent Information Systems. Vol. 4(2). pp. 167-192.

Jeusfeld, M. A., & Neumayr, B. (2016). DeepTelos: Multi-level modeling with most general instances. 35th International Conference on Conceptual Modelling. Lecture Notes in Computer Science. Vol. 9974 LNCS. pp. 198-211. Gifu; Japan: Springer.

Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). ATL: A model transformation tool. Science of Computer Programming. Vol. 72(1-2). pp. 31-39.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study: Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

Kennel, B. (2012). A unified framework for multi-level modeling. pp. 1-267 Doctoral dissertation, Universität Mannheim.

Kolovos, D. S., Paige, R. F., & Polack, F. A. (2006). The epsilon object language (EOL). European Conference on Model Driven Architecture-Foundations and Applications. pp. 128-142. Springer.

Kühne, T. (2009). Contrasting classification with generalisation. Conferences in Research and Practice in Information Technology Series. 6th Asia-Pacific

Conference on Conceptual Modelling, APCCM 2009. pp. 8. Wellington; New Zealand.

Kühne, T. (2018a). Exploring Potency. 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018. pp. 2-12. Copenhagen; Denmark: Association for Computing Machinery, Inc.

Kühne, T. (2018b). A story of levels. CEUR Workshop Proceedings. 2018 MODELS Workshops. pp. 673-682. Copenhagen; Denmark.

Kühne, T., & Schreiber, D. (2007). Can programming be liberated from the two-level style? Multi-level programming with DeepJava. OOPSLA 2007: 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 229-244. Montreal, QC; Canada: ACM.

Lamo, Y., Wang, X., Mantz, F., Bech, Ø., Sandven, A., & Rutle, A. (2013). DPF workbench: a multi-level language workbench for MDE. Proceedings of the Estonian Academy of Sciences. Vol. 62(1). pp. 3-15.

LE4MM. (2018). XModeler. https://www.wi-inf.uni-duisburg-essen.de/LE4MM/

Macías, F., Guerra, E., & de Lara, J. (2017). Towards rearchitecting meta-models into multi-level models. Lecture Notes in Computer Science. Vol. 10650 pp. 59-68. Valencia; Spain: Springer.

Macías, F., Rutle, A., & Stolz, V. (2016). MultEcore: Combining the best of fixed-level and multilevel metamodelling. 3rd International Workshop on Multi-Level Modelling, MULTI 2016. pp. 66-75. Saint-Malo; France: CEUR-WS.

Macías, F., Rutle, A., Stolz, V., Rodriguez-Echeverria, R., & Wolter, U. E. (2018). An approach to flexible multilevel modelling. Enterprise Modelling and Information Systems Architectures (EMISAJ), International Journal of Conceptual Modeling. Vol. 13. pp. 1-35.

Macías, F., Wolter, U., Rutle, A., Durán, F., & Rodríguez-Echeverría, R. (2019). Multilevel coupled model transformations for precise and reusable definition of model behaviour. J. Log. Algebr. Meth. Program. Vol. 106. pp. 167-195.

Martin, R. C., Riehle, D., & Buschmann, F. (1997). Pattern Languages of Program Design 3 (Firts edition). Vol. 1. pp. 656. Addison-Wesley Longman Publishing Co., Inc.

MDA. (2014). Model Driven Architecture. https://www.omg.org/mda/

Medvidovic, N., & Taylor, R. N. (2010). Software architecture: Foundations, theory, and practice. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. pp. 471-472. Cape Town; South Africa: ACM.

MOF. (2016). Meta Object Facility (MOF) specification 2.5.1. https://www.omg.org/spec/MOF

MULTI. (2014). MULTI series of workshops at the MODELS conference. http://miso.es/multi/

Mylopoulos, J., Borgida, A., Jarke, M., & Koubarakis, M. (1990). Telos: Representing knowledge about information systems. ACM Transactions on Information Systems (TOIS). Vol. 8(4). pp. 325-362.

Mylopoulos, J., Feather, M. S., Meyer, B., Paolini, P., Smith, D. C. P., & Hendrix, G. G. (1980). Relationships Between and Among Models. Vol. 1. pp. 77-82. Pingree Park, Colorado, USA: ACM.

Neumayr, B., Schuetz, C. G., Horner, C., & Schrefl, M. (2017). DeepRuby: Extending Ruby with Dual Deep Instantiation. MoDELS 2017 (Satellite Events). Vol. 2019. pp. 252-260. Austin; United States: CEUR-WS.

Neumayr, B., Schuetz, C. G., Jeusfeld, M. A., & Schrefl, M. (2018). Dual deep modeling: multi-level modeling with dual potencies and its formalization in F-Logic. Software & Systems Modeling. Vol. 17(1). pp. 233-268.

OCL. (2014). The Object Constraint Language (OCL) specification 2.4. https://www.omg.org/spec/OCL/

Odell, J. (1994). Power types. Journal of Object-Oriented Programming (JOOP). Vol. 7(2). pp. 8-12.

Paige, R. F., Kolovos, D. S., Rose, L. M., Drivalos, N., & Polack, F. A. (2009). The design of a conceptual framework and technical infrastructure for model management language engineering. 14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009. pp. 162-171. Potsdam, Germany: IEEE.

Poo, D., Kiong, D., & Ashok, S. (2008). Classification, Generalization, and Specialization *Object-Oriented Programming and Java* pp. 51-59: Springer.

Rossini, A., de Lara, J., Guerra, E., & Nikolov, N. (2015). A comparison of two-level and multi-level modelling for cloud-based applications. 11th European Conference on Modelling Foundations and Applications, ECMFA 2015. Lecture Notes in Computer Science. Vol. 9153. pp. 18-32. L'Aquila; Italy: Springer.

Schmidt, D. C. (2006). Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer. Vol. 39(2). pp. 25-31.

Selway, M., Stumptner, M., Mayer, W., Jordan, A., Grossmann, G., & Schrefl, M. (2017). A conceptual framework for large-scale ecosystem interoperability and industrial product lifecycles. Data & Knowledge Engineering. Vol. 109. pp. 85-111.

SMOF. (2013). MOF Support for Semantic Structures (SMOF). https://www.omg.org/spec/SMOF/About-SMOF/

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). EMF: Eclipse Modeling Framework. Vol. Second edition. pp. 744. Pearson Education.

UML. (2017). The Unified Modelling Language (UML) specification 2.5.1. https://www.omg.org/spec/UML

Yang, G., Kifer, M., Zhao, C., & Chowdhary, V. (2005). FLORA-2: User's manual. Vol. 1. pp. 137. Department of Computer Science. State University of New York at Stony Brook.

Zhu, Z., Lei, Y., Alshareef, A., Sarjoughian, H., & Zhu, Y. (2018). Domain Specific MetaModeling for Deep Semantic Composability. IEEE Access. Vol. 6. pp. 18276-18289.

Zschaler, S., Kolovos, D. S., Drivalos, N., Paige, R. F., & Rashid, A. (2009). Domain-specific metamodelling languages for software language engineering. 2nd International Conference on Software Language Engineering, SLE 2009. Lecture Notes in Computer Science. Vol. 5969 LNCS. pp. 334-353. Denver, CO; United States: Springer.

Appendix A.

This appendix contains the OCL invariants for the MLM meta-model of Figure 8.

```
-- The potency of the MLModel should be higher than or equal to the potency of
-- their contained clabjects, or unbounded
context MLModel inv:
  potency=-1 or classes.forAll(c | potency>=c.potency)


-- The potency of an element cannot be unbounded if the potency of the MLModel is not.
context MLModel inv:
  potency<>-1 implies classes->union(references)->union(fields).forAll(potency<>-1)


-- The potency of an MLClass object should be higher than or equal to the potency of
-- the MLAttribute and MLReference objects annotating the class features
context MLClass inv:
   potency=-1 or
   (MLReference.allInstances()->select(r | mlmClass.eAllReferences->includes(r.mlmReference))->
                                forAll(r | potency>=r.potency) and
    MLField.allInstances()-> select(f | mlmClass.eAllAttributes->includes(f.mlmField))->
                             forAll(f | potency>=r.potency))

-- The potency of clabjects is higher than or equal to the potency of the ancestor clabjects
-- in inheritance hierarchies (or unbounded, if some ancestor has unbounded potency).
context MLClass inv:
   MLClass.allInstances()->select(c | mlClass.eAllSuperTypes->includes(c.mlClass))->
                           forAll(c | (c.potency=-1 implies potency=-1) and potency >= c.potency)


-- BoundedElement.min should be positive or zero, while BoundedElement.max should be higher
-- than or equal to BoundedElement.min, or -1
context BoundedElement inv:
   min>-1 and (max=-1 or max>=min)

-- The cardinality of a clabject has to be wider than or equal to the interval made of the minimum
-- cardinality of each subclabject (through the inheritance relation), and the sum of the maximum
-- cardinalities of each subclabject (where * plus anything results in *).
context MLClass inv:
   MLClass.allInstances()->select(c | c.mlClass.eAllSuperTypes->includes(mlClass))->
                           forAll(c | min <= c.min)
context MLClass inv:
   if (MLClass.allInstances()->
                           select(c | c.mlClass.eAllSuperTypes->includes(mlClass))->
                           exists(c | c.max = -1))
   then max=-1
   else (max=-1 or
         max >= MLClass.allInstances()->select(c | c.mlClass.eAllSuperTypes->includes(mlClass)).max.sum())
   endif
```