

A Conceptual Framework for Large-scale Ecosystem Interoperability and Industrial Product Lifecycles

Matt Selway, Markus Stumptner, Wolfgang Mayer, Andreas Jordan, Georg Grossmann

Advanced Computing Research Centre, School of IT & Mathematical Sciences, University of South Australia

Michael Schrefl

Johannes Kepler University Linz, Austria

Abstract

One of the most significant challenges in information system design is the constant and increasing need to establish interoperability between heterogeneous software systems at increasing scale. The automated translation of data between the data models and languages used by information ecosystems built around official or de facto standards is best addressed using model-driven engineering techniques, but requires handling both data and multiple levels of metadata within a single model. Standard modelling approaches are generally not built for this, compromising modelling outcomes. We establish the SLICER conceptual framework built on multilevel modelling principles and the differentiation of basic semantic relations (such as specialisation, instantiation, specification and categorisation) that dynamically structure the model. Moreover, it provides a natural propagation of constraints over multiple levels of instantiation. The presented framework is novel in its flexibility towards identifying the multilevel structure, the differentiation of relations often combined in other frameworks, and a natural propagation of constraints over multiple levels of instantiation.

Keywords: Metamodelling, Conceptual models, multilevel modelling, Ecosystem Interoperability

1. Introduction

Lack of interoperability between computer systems remains one of the largest challenges of computer science and costs industry tens of billions of dollars each year [1, 2]. Standards for data exchange have, in general, not solved the problem: standards are not universal nor universally applied (even within a given industry) leading to *heterogeneous ecosystems*. These ecosystems comprise large groups of software systems built around different standards that must interact to support the entire system lifecycle. We are currently engaged in the “Oil and Gas Interoperability Pilot” (or simply OGI Pilot), an instance of the Open Industry Interoperability Ecosystem (OIIE) initiative that aims for the automated, model-driven transformation of data during the asset lifecycle between two of the major data standards in the Oil & Gas industry ecosystem. The main standards considered by the project are the ISO15926 suite of standards [3] and the MIMOSA OSA-EAI specification [4]. These standards and their corporate use¹ are representative of the interoperability problems faced in many industries today. To enable sensor-to-boardroom reporting, the effort to establish and maintain interoperability solutions must be drastically reduced. This is achieved by developing model transformations based on high level conceptual models.

Email addresses: matt.selway@unisa.edu.au (Matt Selway), mst@cs.unisa.edu.au (Markus Stumptner), wolfgang.mayer@unisa.edu.au (Wolfgang Mayer), andreas.jordan@mymail.unisa.edu.au (Andreas Jordan), georg.grossmann@unisa.edu.au (Georg Grossmann), schrefl@jku.at (Michael Schrefl)

¹Industrial participants in past demonstrations included: Bentley, AVEVA, Worley-Parsons, for ISO15926; IBM, Rockwell Automation, Assetricity for MIMOSA; various Oil & Gas or power companies as potential end users.

In our previous work [5] we presented three core contributions: (1) we compared the suitability of different multi-level modelling approaches for the integration of ecosystems in the Oil & Gas industry, (2) introduced the core SLICER (Specification with Levels based on Instantiation, Categorisation, Extension and Refinement) relationship framework to overcome limitations of existing approaches with respect to the definition of object/concept hierarchies, and (3) evaluated the framework on an extended version of the comparison criteria from [6].

The current work extends these contributions by: (1) expanding on the explicit handling of descriptions in the SLICER framework, (2) extending the core SLICER relationships with a complete treatment of attributes, relationships, and their integrity constraints, (3) presenting the formalisation of SLICER core and the treatment of attributes, and (4) illustrating mappings between a SLICER model and alternatives making use of SLICER's finer semantic distinctions to identify patterns of meaning in the original models.

2. Ecosystem Interoperability

The suite of standard use cases defined by the Open O&M Foundation covers the progress of an engineering part (or plant) through the Oil & Gas information ecosystem from initial specification through design, production, sales, deployment, and maintenance including round-trip information exchange. The data transformations needed for interoperability require complex mappings between models covering different lifecycle phases, at different levels of granularity, and incorporating data and (possibly multiple levels of) metadata within one model.

Notably, different concepts are considered primitive objects at different stages of the lifecycle. For example, during design, the specification for a (type of) pump is considered an object that must be manipulated with its own lifecycle (e.g. creation, revision, obsolescence), while during operations the same object is considered a type with respect to the physical pumps that conform to it and have their own lifecycle (e.g. manufacturing, operation, end-of-life). Furthermore, at the business/organisational level, other concepts represent categories that perform cross-classifications of objects at other levels. This leads to an apparent three levels of (application) data: business level, specification level, and physical entity level. To describe these different levels multi-level modelling (MLM) approaches to model-driven engineering seem a natural fit. Ideally, a flexible conceptual framework should represent the entire system lifecycle, in a way that simplifies the creation of mappings between disparate models by the interoperability designer.

The ecosystem transformations use a joint metamodel that serves as the common representation of the information transferred across the ecosystem (cf. Figure 1) and must be able to handle the MLM aspects. As pointed out in [7], such complex domains generally are not dealt with using the classical GAV or LAV (Global/Local As View) querying approach, but require a more general form of mapping describing complex data transformations. Notably, the data integration systems surveyed in [7] generally use languages that do not have MLM or even metamodeling capabilities, and the automated matching capability of the systems listed (e.g., MOMIS, CLIO) is probabilistic. As transformations in the engineering domain must guarantee correctness (e.g., an incorrectly identified part or part type can result in plant failures), probabilistic matching cannot replace the need for a succinct and expressive conceptual model for designing the mappings.

The traditional UML-based approach is insufficient for this purpose. Consider Figure 2 illustrating a UML-based approach in which ProductCatalogue, ProductCategory, ProductModel, and ProductPhysicalEntity are explicitly modelled as an abstraction hierarchy using aggregation in an attempt to capture the multi-level nature of the domain.² Specialisation is used to distinguish the different categories (i.e. business classifications), models (i.e. designs), and physical entities of pumps. Finally, instantiation of singleton classes is used to model the actual catalogue, categories, models, and physical entities.

From both a conceptual modelling and interoperability perspective, this is unsatisfactory: it is heavily redundant in that it models the same entities twice (once as a class and once as an object) and using both

²While UML 2 supports power types in limited fashion, this example is restricted to more basic UML constructs. Power types are discussed in Section 3. In addition, primitive types are used for brevity, rather than modelling units of measure in full.

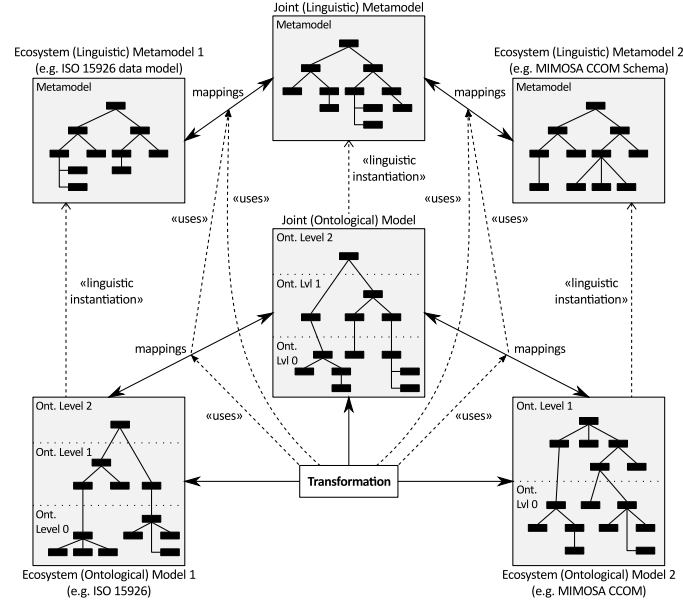


Figure 1: Ecosystem interoperability through a joint metamodel

aggregation and generalisation to represent what is otherwise the same hierarchical relationship [8, 6]; the misuse of the aggregation relationship to represent a membership and/or classification relation results in physical entities that are not intuitively instances of their product models; last it creates difficulty in modelling the lifecycles of both design and physical entities as well as the dynamic introduction of new business categories. This directly affects mapping design as the real semantics of the model are hidden in implementation.

3. Multi-Level Modelling Techniques

A number of Multi-level modelling (MLM) techniques have been developed to address the shortcomings of the UML-based model. While they improve on the UML-based model in various respects, no current approach fulfils all of the criteria necessary for ecosystem interoperability (see Section 7).

Most current MLM techniques—particularly Deep Instantiation (DI) approaches such as Melanee [9], MetaDepth [10], and Dual Deep Modelling [11]—focus on the reduction of *accidental complexity* in models [12]. In the past, accidental complexity has been reduced through higher-level programming languages that freed programmers from thinking in terms of non-essential details of the task, such as bits and registers, and allowed them think in terms of the operations and data types needed to perform a task [13]. Conceptual Modelling itself has reduced accidental complexity by allowing modellers to think in terms of the entities, relations, and constraints [14] of a domain rather than the data types and operations of a program used to manipulate them. The achievement of DI techniques, which use a special attribute called *potency* to determine the transfer of information across multiple levels of instantiation, has been to reduce accidental complexity by minimising the elements of a conceptual model on the assumption that ‘...if one model conveys the same information as another model, but in a more concise way using less modeling elements and concepts, it is less complex’ [12]. The result is conceptual modelling approaches that hide elements by collapsing them into a single object at the top-most level. For example, Figure 3a illustrates a simple DI approach where the concept at the top-level, Pump Model, contains attributes related to both pump models (i.e., types of pumps) as well as physical pumps: $temp^2$ (denoting the current pump operating temperature) is intended to be instantiated by actual pumps, while $maxTemp^1$ (denoting the maximum safe operating temperature) is intended to be instantiated by pump models. Similarly, Figure 3c shows this collapse into a top-level concept, Pump, using the M-Objects (or Multi-level Objects) [15] approach. Unlike DI approaches, M-Objects explicitly label the facets (i.e., distinct yet strongly related

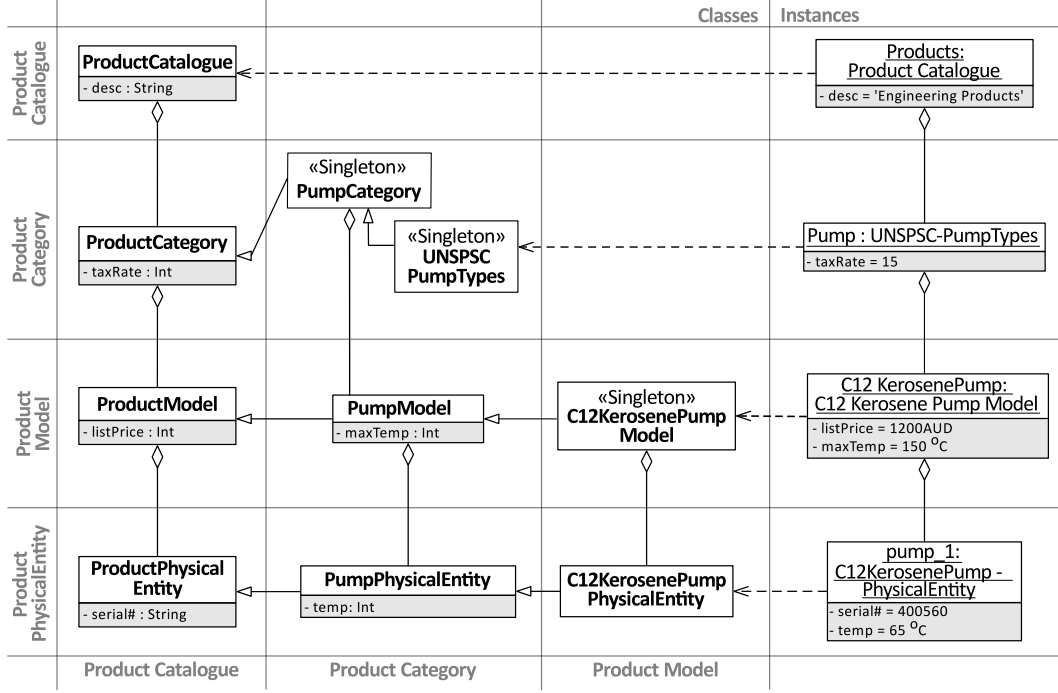


Figure 2: Product catalogue modelled in plain UML (adapted from [6])

aspects of the domain) to which the attributes relate: the attributes in the <model> section relate to pump models and the attributes in the <physical entity> section relate to physical pumps.

We argue that such an approach is inappropriate, especially for the Ecosystem Interoperability scenario discussed earlier, as it hides important distinctions existing in the represented domain. Moreover, such compactness is not in the same spirit as the original use of *accidental complexity*. Since Multi-level modelling is focused on modelling aspects of the domain such that the model elements *represent* domain entities [16, 17], we argue that accidental complexity is reduced if the domain modeller can *naturally* represent the entities, relationships, and constraints of their domain. This does not require that concepts be merged into single entities; in contrast, it suggests that important domain entities should be made explicit, even if this makes the model less compact. That is to say, if the concept of a “pump model” is important in a domain it should explicitly exist as an entity in the model, rather than implicitly as the “model” facet of the related entity Pump. We agree with Atkinson and Kühne [12] in that the predominant conceptual modelling languages, e.g. UML, increase accidental complexity by forcing the modeller to consider workarounds to their fundamentally two-levelled nature and that conceptual modelling languages that naturally support multiple levels are required. However, the emphasis on concise models may increase accidental complexity by hiding relevant domain objects as (unnamed) facets of objects at higher-levels of abstraction. This results in a one-to-many relationship between model elements and the domain entities that they represent, i.e. *construct overload* [18], which prevents the domain modeller from thinking in terms of the relevant entities and instead about how it is encoded in the model (i.e. the focus is on which register the information is stored instead of what it means for the domain). Such complexity becomes a problem when designing interoperability solutions across models in different, but overlapping, domains. In this scenario, a conceptual modelling language is required that allows the distinctions between different domain elements to be made explicit to reduce mismatches and, thus, accidental complexity.

Potency-based MLM Techniques [8] were originally introduced for *Deep Instantiation (DI)* to support the transfer of information across more than one level of instantiation. They separate *ontological* instantiation, which are domain specific instantiation relationships which can be cascaded multiple times, from standard *linguistic* instantiation used in UML meta-modelling. Each model element (e.g. class, object, attribute,

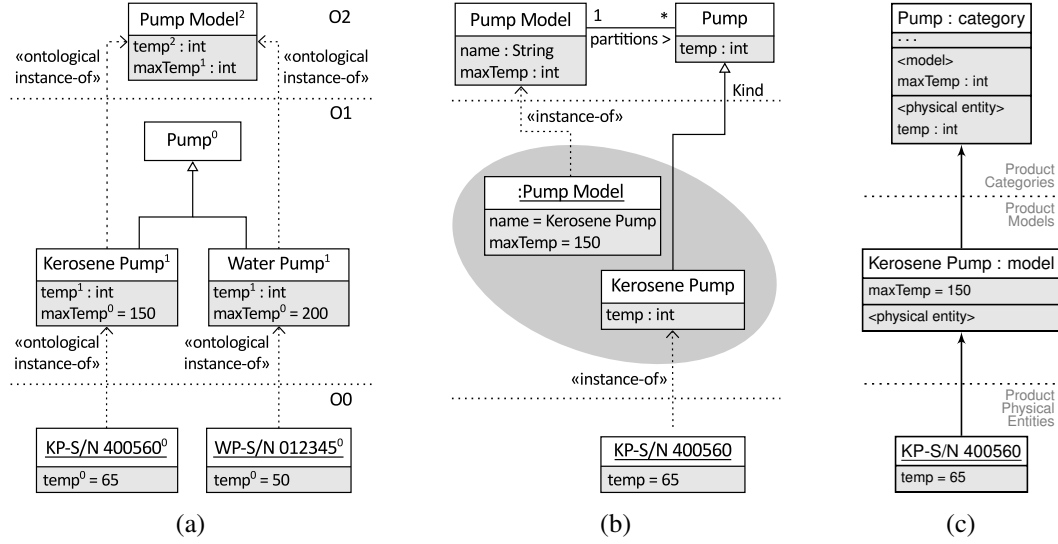


Figure 3: Extracts of the product catalogue example represented as: (a) Deep Instantiation, (b) Power Types, and (c) M-Objects. Superscripts represent *potency*; the ellipse links the type and instance facets of a concept following the notation of [19].

or association) has an associated *potency*³ value that defines how many times the model element can be ontologically instantiated. That is, a class with potency 2 can have instances of instances, while an attribute with potency 2 can be transferred across two instantiation levels. Elements with potency 0 cannot be further instantiated. For example, the class Pump Model in Figure 3a defines two attributes: *temp*² and *maxTemp*¹. Both attributes have the flat data-type as range (i.e., an integer representing a value in degrees Celsius); however, *maxTemp*¹ is instantiated only once due to its potency of 1, while *temp*² is instantiated twice and only receives its complete concrete value at the O0 level. Depending on the specific approach, it is possible to assign a value at O1, which would be interpreted as a default value for its instances.⁴

In a more complete model, the ProductCategory concept would result in an additional level of instantiation with attributes pushed up to the higher-level and potencies modified as appropriate. To model complex domains, this approach invariably results in non-instantiation relationships crossing level boundaries, which is not allowed under *strict* meta-modelling typically adhered to by potency-based models. For example, if the attributes *temp*² and *maxTemp*¹ were modelled as associations to values of the type DegreeCelsius (rather than “primitive” integers that must be interpreted as degrees), no matter at what level DegreeCelsius is placed it would result in an association crossing a level boundary at some point [20]. Moreover, the concept Pump is in violation of strict meta-modelling as it does not (ontologically) instantiate anything from the level above.⁵ The intended alternative would be to remove Pump altogether as it is mostly considered redundant in DI approaches [12]; however, this would cause a conceptual mismatch with the domain (as the concept of Pump has to be extracted by interpreting attributes with potency 2 in Pump Model), leading to increased accidental complexity in models where such a concept is relevant. Finally, the use of potency to combine hidden facets in a single object conflates specialisation and instantiation semantics, complicating model transformations.

Intrinsic Features [21] take a similar approach to potency-based methods for transferring information across multiple levels. However, instead of a relative level of instantiation as is the case for potency, *intrinsic features* are assigned a specific level at which they are to be instantiated. At the levels above their instantiation level, intrinsic features can refine their type in conformance with the specialisation/instantiation hierarchy. Since intrinsic features are approximately equivalent to potency, they have similar

³In recent versions of Melanee [9], a Deep Instantiation-based meta-modelling tool, potency is called *durability*.

⁴In recent versions of Melanee [9], this form of instantiation can be controlled by another property called *mutability*, which defines the number of levels that the attribute is to be assigned a value.

⁵In Dual Deep Modelling [11] this problem can be solved by making Pump an *abstract instance* of PumpModel. However, this produces a conceptual mismatch as Pump is not actually a PumpModel, its subtypes are.

problems. Although intrinsic features do not necessarily enforce strict meta-modelling as potency-based approaches do, confusion arises as to what relates to what and at which level. Moreover, the adherence to a specified instantiation level, rather than a relative value, can make models with intrinsic features more difficult to update when a new level is added, for example, than their potency-based counter parts.

Power types [22] have also been applied in a multi-level context, e.g., [19]. Basically, for a power type t of a another type u , instances of t must be subtypes of u . Figure 3b shows an example of the Power Type pattern, where Pump Model is the power type for the concept Pump; the concept ProductCategory would be represented as cascading uses of the power type pattern. As discussed in [6], this leads to complex and redundant modelling, which complicates creation and maintenance of interoperability mappings. Power typing does provide a clearer distinction between the role of the concepts Pump Model and Pump than the DI approach; however, it still leads to the violation of strict meta-modelling with non-instantiation relationships crossing level boundaries no matter where the concepts are relocated. The solution argued for by Gonzalez-Perez and Henderson-Sellers [19] is to provide the *partitions* relationship (which connects a type to its power type) with instantiation semantics, allowing it to cross the level boundary without violating strict meta-modelling. However, this leads to the counter-intuitive notion that the type is a partition of itself and a conceptual mismatch in that Pump is not actually a Pump Model but a more general *kind* with subtypes that can be considered Pump Models. While explicit separation of the type and instance facet may prevent name clashes where both facets have an attribute of the same name (e.g. *owner* of the specific Pump Models and *owner* of individual Pumps), we have found it unnecessary in our large use case. Moreover, this separation leads to redundancy at best and a complex two-level implementation issue at worst, when the two facets are really the *same object* [22] and the name clash can be resolved by using attributes with different names.

M-Objects and M-Relationships (M standing for multi-level) [15] use a *concretization* relation which stratifies objects and relationships into multiple levels of abstraction within a single hierarchy. The example situation would be modelled with a top-level concept ProductCatalogue containing the definition of its levels of abstraction: category, model, and physical entity. The lower levels would include the different PumpCategories, PumpModels, and PhysicalPumps, respectively. A small extract of this is illustrated in Figure 3c. While the M-Objects technique produces concise models with a minimum number of relations, the concretization relation between two m-objects (or two m-relationships) must be interpreted in a multi-faceted way (since it represents specialisation, instantiation, and aggregation), making it difficult to identify model mappings. Compared to potency-based techniques, M-Objects clearly identify the combined facets using explicit labels in compartments of the composite objects.

4. A relationship framework for ecosystem modelling

A highly expressive and flexible approach is required to overcome the challenges involved in modelling large ecosystems and supporting transformations across their lifecycles. A key requirement is the identification of *patterns of meaning* from basic primitive relations that can be identified across modelling frameworks and assist the development of mappings between them. A core observation when building transformations for the real world complexity of the OGI pilot is that a higher level in the model, whether ontological or linguistic, expresses the relationship between an entity and its definition (or description) in two possible ways: abstraction (bottom-up) and specification (top-down).

In *abstraction*, entities are grouped together based on similar properties, giving rise to a concept that describes the group. This is a bottom-up process where the entities may be groups of concepts, thus permitting a multi-level hierarchy. In this situation, presumably, a set of common properties exist that enable differentiation between the entities (e.g. name, weight, age). Moreover, it is possible that a property value is shared between the entities in which case it would be defined at the abstracted concept. A typical example of an abstraction grouping would be the animal/species taxonomy (and other natural kinds) frequently employed as an example in the multilevel modelling literature.

In contrast, *specification* is a top-down process where an explicit description is laid down and entities (artefacts) are produced that conform to this specification [23, 24]. This is a common scenario in the use of information systems and plays a major role in our framework. For example, specification occurs in engineering when a particular model of equipment (or even one-off item) is designed, i.e. specified, before being produced. This applies equally to information objects such as processes and software components.

The common factor is that a *level of description* (and therefore the consistency constraints for a formalism that expresses these principles) is not purely driven by instantiation. A different level is also established by enriching the vocabulary used to formulate the descriptions: corresponding to the concept of *extension* in specialisation hierarchies [25], i.e., if a subclass receives additional properties (attributes, associations etc.) then these attributes can be used to impose constraints on its specification and behaviour. Identifying levels using the basic semantic relationships between entities enables a flexible framework for describing joint metamodels in interoperability scenarios. For example, relationships between entities are not restricted by a rigid, pre-assigned notion of levels; instead dynamic determination of levels enforces stratification of relationships as necessary. Furthermore, domain modellers need not utilise the primitive relations directly; rather, they are identifiable in the conceptual modelling framework of a particular model, supporting the development of mappings between models of different frameworks (see Section 6).

In contrast, most of the MLM approaches summarised in Section 3 focus on the pre-layering of levels as the main designer input, a corollary from the natural rigidity of the level system in linguistic instantiation as intended by UML. However, it is generally assumed by prior specialized work on relationships such as specialization [25], metaclasses [26], materialization, and aggregation [27] that there can be arbitrary many levels of each. Therefore, interoperability solutions must accommodate mappings to models that: are not designed according to the principles of strict meta-modelling, may cover domains at different levels of granularity, and cannot (from the view of the interoperability designer) be modified. For example, a primary aim of the OGI Pilot is to provide “digital handover” of design information to operational-side systems [28]; chiefly, to transform ISO15926-based design data into MIMOSA CCOM-based operations and maintenance data. ISO15926 is intended to be a generic model that can encompass an entire product lifecycle, while CCOM is focused on operations and maintenance activities. As a result, ISO15926 includes at least three levels of data (business level, specification level, physical level), while CCOM focuses on two (specification and physical level). In both cases, some of the relevant domain information is specified in the linguistic meta-models of these standards (more so in the case of CCOM) and violates strict meta-modelling: in ISO15926 this is often an attempt to model (unenforceable) constraints, while for CCOM it is a limitation of trying to model two-levels of information within a single instance-level of its UML-based definition.

4.1. Core Relationships

To support the flexible definition of (meta-)models, we present the SLICER (Specification with Levels based on Instantiation, Categorisation, Extension and Refinement) framework based on a flexible notion of levels as a result of applying specific semantic relationships. In the following description of the relations forming the core SLICER framework we refer to Figure 4, which shows its application to the product catalogue example in the context of the OGI Pilot. The diagram exemplifies the finer distinctions made by the SLICER framework including: the increase of detail or specificity (from top to bottom) through the addition of characteristics, behaviour, and/or constraints that are modelled; the explicit identification of characteristics for describing models of equipment (i.e. specifications) themselves, not only the physical entities; the second level of classification through the identification of categories and the objects that they categorise; and the orthogonal concerns of the different stakeholders and stages of the lifecycle (shown by separation into 3 dimensions). Table 1 summarises the properties of the relations described in this section. In addition, the description of the framework makes reference to formalisation axioms which are included in Appendix A.

Instantiation and Specialisation. Like other frameworks, SLICER uses instantiation and specialisation as the core relations for defining hierarchies of concepts based on increasing specificity (or decreasing abstraction). In contrast to previous MLM techniques, the levels are not strictly specified, but dynamically

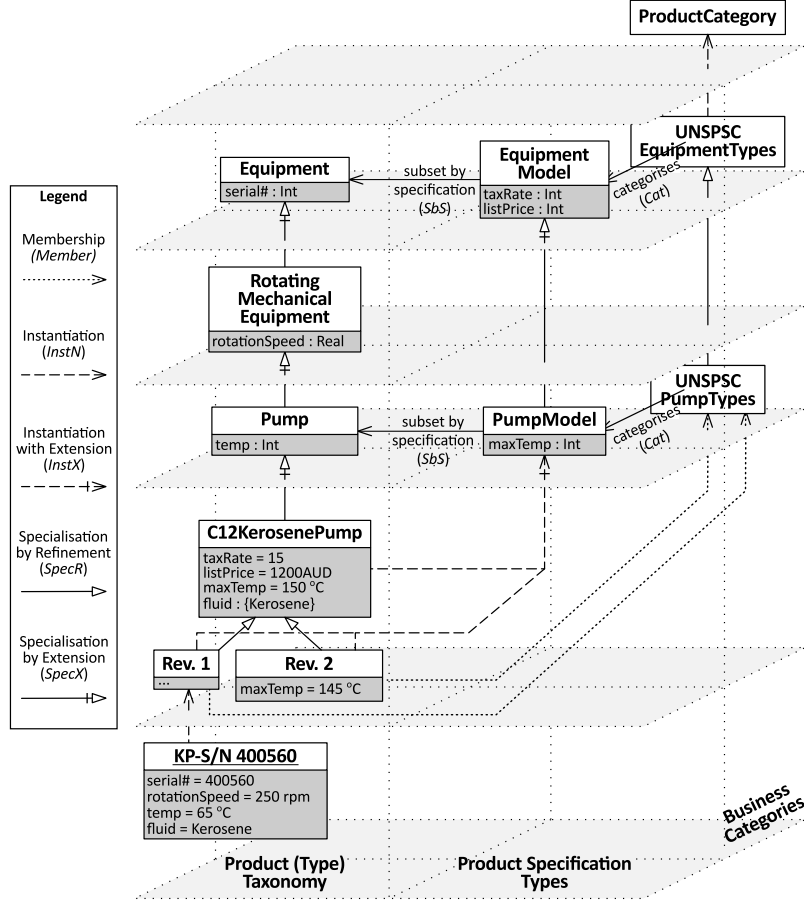


Figure 4: Product catalogue example modelled using SLICER core relationships

derived based on finer distinctions of relations between more or less specific types. We characterise specialisation relationships along the lines of [25] to distinguish a relationship that *extends* the original class (by adding attributes, associations, or behaviour, i.e., constraints or state change differentiation) or *refines* it (by adding granularity to the description). The latter generally means restricting the range of the attribute or association. In the extreme, it can be seen as restricting an attribute to a singleton domain, i.e., pre-specifying the value upon instantiation.

We identify a specialisation relationship that extends the original class as *Specialisation by Extension* (*SpecX* for short), axiom 15 defined in Appendix A, which adopts standard monotonic specialisation semantics. While it can include (but does not necessitate) *refinement*, (axiom 43), it is distinguished based on incorporation of additional attributes (axiom 42). Most importantly, *SpecX* introduces a new model level (axiom 37). For example, in Figure 4, the conceptual entity Pump is a specialisation by extension of RotatingMechanicalEquipment due to the addition of *temp*.

In contrast, *Specialisation by Refinement* (denoted *SpecR*, axiom 9) only *refines* the original class (axiom 41), supporting subtypes that restrict the extension of a class (e.g. by restricting the domains of properties, or adding domain constraints on properties, see axiom 43) without introducing additional model levels (axiom 36). This allows for an arbitrary number of subtypes to simply increase the granularity of a level and expand the domain vocabulary. For example, the two revisions of the C12KerosenePump design (an important distinction in the example domain) only refine the attributes and constraints of C12KerosenePump and are specialisations by refinement.

Similarly, instantiation is characterised as *Instantiation with Extension* (*InstX*), axiom 13, or *Standard Instantiation* (*InstN*), axiom 12). Instantiation always introduces additional model levels (axiom 35) and all attributes of the instantiated type have values assigned from their domain (axiom 47). However, *InstX* allows additional attributes, behaviour, etc. to be added that can then be instantiated or inherited further

Table 1: Summary of Relation Properties

	<i>SpecX</i>	<i>SpecR</i>	<i>InstX</i>	<i>InstN</i>	<i>Cat</i>	<i>Member</i>	<i>SbS</i>
Notation	$\dashv\rightarrow$	\rightarrow	$---\dashv\rightarrow$	$---\rightarrow$	\xrightarrow{Cat}	$\cdots\rightarrow$	\xrightarrow{SbS}
Refinement	x	x					
New attributes/relations	x		x				
$^a level(x) - level(y) \geq 1$	x		x	x		x	
$^a level(x) - level(y) \geq 0$		x					
$^a level(x) - level(y) = 0$					x		x
Assign values to attr.			x	x			
Inst. can have inst.	x	x	x			x	x
Propagates constraints	x	x	x	x	x	x	x
Cat.-Type Relation					x		
Base type attr. access					x	(x)	x
Used for Powertype			x				x

^a Where $\phi(x, y)$, $\phi \in \{SpecX, SpecR, InstX, InstN, Cat, Member, SbS\}$

(axiom 45), while *InstN* does not (axiom 46). As such, objects that are the result of *InstN* cannot have instances themselves (axioms 27, 32 and 33). The pump model C12KerosenePump demonstrates *InstX* as it introduces characteristics specific to that type of pump (e.g. *fluid* with a singleton domain), while KP-S/N 400560, representing a physical pump, demonstrates *InstN*. Values assigned through instantiation can be inherited by the specialisations of the instance unless otherwise specified (axiom 48). For example, Rev. 1 and Rev. 2 inherit the values for *taxRate* and *listPrice* from C12KerosenePump, but only Rev. 1 inherits the value for *maxTemp* as Rev. 2 assigns its own value.

The inheritance of attributes across specialisation and instantiation relations is kept track of via the *From* relation (axiom 22). This differentiates the path(s) through which an attribute is inherited and can be used in the definition of user views, for example, to distinguish between attributes belonging to the different facets (class vs. instance) of an object. For example, a view could be created consisting of EquipmentModels and their instances (i.e., subtypes of Equipment) such that only the properties directly instantiated from EquipmentModel and its subtypes are visible. Using the *From* relation, the attributes *taxRate*, *listPrice*, and *maxTemp* of C12KerosenePump would be identified, for example, while *temp* and *fluid* would be ignored. Furthermore, the object (C12KerosenePump) Rev. 2 would show the attribute *maxTemp* as it instantiates the attribute directly from PumpModel, while (C12KerosenePump) Rev. 1 would not, as it inherits the *maxTemp* attribute through specialisation.

Specifications. The second means of relating an entity to its description is specification. For this we introduce the *Subset by Specification* (*SbS*, axiom 18) relation to identify specification types and the parent type of the specification concepts (axiom 50). The result of applying the *SbS* relation to a pair of concepts, for example *SbS*(EquipmentModel, Equipment), is that the instances of EquipmentModel are specialisations of Equipment. The specification class exists at the same level as the type to which it refers (axiom 49) as it can define constraints with respect to that type (axiom 51). Subtypes of the specification type can be defined to reference particular properties (axioms 57 to 60); so EquipmentModel can be specialised (e.g. PumpModel) to refer to properties of Pumps. Together with *InstX*, this relationship can be used to construct the powertype pattern [22] (also called *Characterisation* in [29]).

In the presence of multiple specification types for a concept, SLICER supports multiple instantiation (axioms 30 and 31); that is, an object x can be an instance of concepts c_1, \dots, c_n if they are instances of specification types s_1, \dots, s_m for the same concept z . For example, if a second specification type were related to Pump then KP-S/N 400560 could instantiate specifications related to both types. This form of multiple partitioning of a type [22] is not supported in standard meta-modelling nor multi-level modelling approaches that allow only single instantiation. Moreover, the partitioning can be controlled by specifying that a specification type *covers* (axiom 52), is *disjoint* over (axiom 53), or *partitions* (axiom 54) the target type. A specification type that *Covers* a type requires that all instances of the target type be instances of

instances of the specification type (axiom 55). The instances of a type with a *Disjoint* specification type cannot be an instance of more than one instance of the specification type (axiom 55). Finally, partitioning combines the *Covers* and *Disjoint* relations and, therefore, the instance of a partitioned type must be an instance of exactly one instance of the specification type (axiom 54).

Categories. Categories are concepts that provide external (“secondary”) grouping of entities based on some common property and/or explicit enumeration of its members (in which case the common property is considered to be explicit inclusion in the category). In the SLICER framework, we explicitly represent categories through two relationships: *Categorisation* (*Cat*, axiom 17) and *Membership* (*Member*, axiom 16). *Categorisation* relates two concepts, one representing a category and the other a type, where the *members* of the category are *instances* of the type (axiom 63).

While *Membership* resembles instantiation (the two are mutually exclusive, axioms 34 and 65), it does not place any constraints on the assignment of values to attributes, i.e., the category and its members can have completely different sets of attributes. This does not preclude the specification of membership criteria, or constraints, for allowing or disallowing the possible members of a category.

In addition, categories exist on the same level as the type they categorise (axiom 62). This is intuitive from the notion that their membership criteria (if any) are defined based on the type they categorise (axiom 66). Moreover, sub-categories can be specified through *SpecR* only (axiom 64), supporting refinement of the type of the category and/or the membership constraint.

For example, the concepts UNSPSCEquipmentTypes and UNSPSCPumpTypes are a pair of categories indicating that their members are equipment models (or specifically pump models for the subcategory) conforming to the UNSPSC standard. The two revisions of C12KerosenePump are both *members* of UNSPSCPumpTypes (and hence are members of UNSPSCEquipmentTypes).

Descriptions and Constraints. The final SLICER component is the explicit handling of object descriptions and constraints (axiom 9) through the different relations introduced above. Similar to work on constraint-based systems [30], each object has a description, which is the set of constraints that the object must *satisfy* to be valid (axiom 71). The constraints of a description are inherited across specialisation, instantiation, and membership relations (axioms 73 to 75). Therefore, a specialised object must satisfy all of the constraints of the more general object. Most important, however, is the handling of situations that allow the propagation of constraints across multiple instantiation relations. For example, during the design, *maxTemp* represents a requirement that each pump is able to achieve this maximum (operating) temperature without failure. This may impact, or put constraints on, other parts of the design (e.g. the selection of casing material, valve types, or seal types) as they must all be able to achieve the temperature without failure. During operation, however, *maxTemp* is considered to be an upper limit at which the pump should be shutdown if it is reached or exceeded. Therefore, the specification type Pump Model may define a constraint between *maxTemp* (defined by Pump Model itself) and *temp* (defined by Pump) such that *temp* should be less than or equal to *maxTemp* (with shutdown of the pump being triggered if the constraint is violated). In this situation, the constraint is not applicable to direct instances of Pump Model as they do not assign a value to *temp*; instead, the constraint applies to the instances of Pump (such as KP-S/N 400560), two instantiations away from Pump Model, where the constraint is originally defined.

Such complex constraints are common in detailed design and cannot be handled implicitly through standard attributes and their range constraints. From the interoperability designer’s point of view, complex constraints can be incorporated into the joint meta-model to link attributes of different models that would otherwise be disconnected. For example, the constraint between *temp* and *maxTemp* may not occur in any single model of the ecosystem, but is what is expected of the ecosystem as a whole. In addition, constraint propagation, rather than evaluating constraint where they are defined, allows interoperability designers to view all constraints that apply to an object and/or its instances at any particular time. By tracing the propagation, the designer can view from where constraints are inherited. This allows designers to maintain a wider view of the constraints and be aware of possible interactions between them: further supported by automated validation and consistency checking.

To support complex constraints spanning multiple instantiation relations, we provide a natural and intuitive notion of constraint propagation. Basically, we propagate the part of the description that is not applicable at a certain level across instantiation relations until it reaches an object for which it can be evaluated (see axiom 74). For those attributes that do have values, the attributes are substituted by their values in the propagated constraints. Similarly, a constraint (i.e. a membership criterion) on UNSPSC Pump Types that refers to attributes of Pump Model would be propagated to its members. For example, if the criterion were $maxTemp \geq 150$, C12 Kerosene Pump Rev. 2 would violate that constraint and, hence, should not be a member of UNSPSC Pump Types. The propagation of constraints is traced through the hierarchy using an overloaded *From* relation. This is useful in the definition of user views, and to guide a user to the cause of an error when a constraint is violated (e.g. when violating a membership criterion).

Objects can be *validated* against their description by evaluating the constraints that apply to them. An object is *valid* if all of its *ground* constraints, i.e., the constraints for which it has all concrete values (axiom 72), evaluate successfully. Since an object may be associated with non-ground constraints, we consider whether or not an object is *consistent* (axiom 76). An object is *consistent* with its description iff it satisfies its description and those constraints containing attributes without values do not contradict any other constraint. This enables us to check constraints *before* they are fully instantiated. For example, if an instance of Pump Model were to assign the value 100°C to *maxTemp* while specifying a range for *temp*, such as 101°C–200°C, we can detect the inconsistency in the constraint $temp \leq maxTemp$ ($maxTemp \mapsto 100$) before further instantiation is performed. This kind of partial constraint evaluation is important during the design of physical assets where the distinction must be made between the object being valid wrt. the constraints that apply to itself and the consistency of the constraints that it defines for its instances or that are inherited from a higher level.

For brevity, we do not explicitly consider the properties of the constraint language in this paper; however, we note that our descriptions follow the same approach as that of Generative Constraint Satisfaction (GCSP) [30]. In this scenario, designs generate constraints which are checked for satisfiability. Moreover, in the Engineering domain constraints are mostly range constraints and inequalities which are all decidable. Furthermore, we do not consider here the exact nature of the constraints with respect to their modality. For example, the constraint between *temp* and *maxTemp* described earlier is really a deontic constraint, in that it may be violated in the real-world, rather than an alethic constraint that can never be violated. From the interoperability perspective, constraint violations (regardless of modality) can be used to flag potential issues with data entering the transformation (as it could have been the result of user error) or used to inform a user that the transformed model will be incomplete (if the target model has strict enforcement of the constraint preventing such “invalid data”).

4.2. Clarification of Model Levels

There has been much confusion over the meaning of model levels in (meta-)modelling. Even recent works (e.g. [31, 17]) seek to clarify their meaning. Much of the confusion stems from the OMG’s “four layered meta-model architecture” [32], illustrated in Figure 5a, in which the M0 layer has been interpreted in different ways, as: model instances, runtime (code?) instances, or the real-world. In part, this arises from UML’s concept of InstanceSpecification which is only meant to *represent* an actual instance as an example or snapshot [33, 34]. This stems from the fact that UML is a modelling language intended to be used at design time [34]; actual implementation of UML models comes from code generation to a target implementation language. Such code generation constitutes a model transformation that promotes, i.e., raises the model, from M1 to M2. For the promoted meta-model, its model instances (which can be considered to be runtime instances) exist at M1 and represent real-world entities at M0. If MOF were used to define a Domain Specific Modelling Language directly (as is the case when using the Eclipse Modelling Framework⁶) the (meta-)model would already be at the M2 level, thereby making the code generation step a horizontal transformation (rather than promotion). Moreover, if used without code generation on a model execution engine then the model instances at M1 *are* the runtime instances.

⁶<http://www.eclipse.org/modeling/emf/>

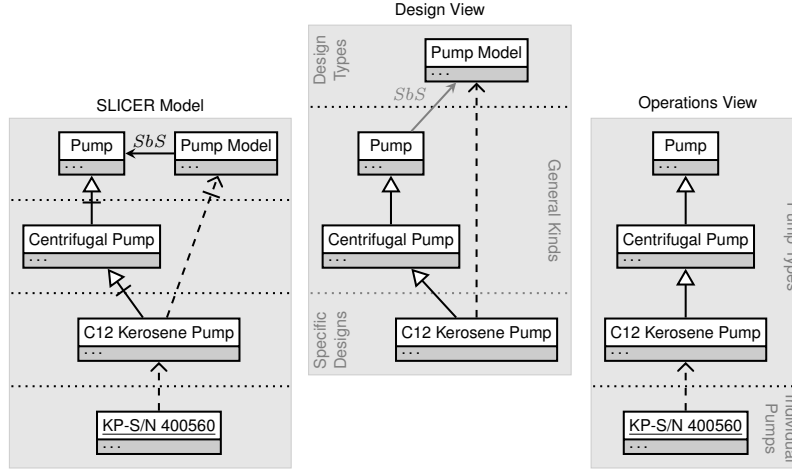


Figure 6: User views defining layers based on instantiation

Since SLICER has the view that “everything is an object”⁷, there is no distinction between attributes, associations, properties, and slots, at the primitive level. Similarly, there is no differentiation between attribute values, slot values, and links of an object. For consistency we use the term *attribute* when referring to a domain relationship that links one type of object to another, and attribute *value* for a link between one object and another for a specific attribute.

Basic Attributes. Some of the basic attribute axioms have been covered in the previous section as they are relevant to the semantics of instantiation and specialisation. They include the basic definition of *attributes* as a set of labels associated with each object, the *type* of an attribute as an object in the domain, and the *values* of an attribute as objects in the domain (axioms 5 to 8). In addition, attributes are inherited across the different semantic relations described previously, allowing refinement of their type, and enforcing consistency between the type(s) of attributes and their values in subclasses and across levels (axioms 24 to 26, 38 to 40, 43 and 46 to 48). In general, the type or value of an attribute must be defined above, or at the same level, to which it is referred (axioms 38 and 39). This ensures that certain level alignments occur in a model. For example, if two objects have attributes referring to each other as values, they must be strongly related and will appear at the same level. Moreover, an object specified as a type must be able to have instances or members (axiom 40) to be a valid definition.

Meta-Attribute Instantiation. While the basic definition of attributes is enough for most cases in SLICER and is compatible with that of two-level models, it can be desirable to have (meta-)attributes that are *instantiated* multiple times across different levels. For example, Pump Model could define an attribute *hasComponent* to another model type such as Impeller Model (to say that models of pump use certain model(s) of impeller as a component) and we want this attribute to carry down to the level of individual pumps (to say that a specific pump has a specific impeller) *without specifying a complex constraint*. This is one of the core aspects of DI [8] and we support it in an intuitive fashion by allowing the replacement of the instantiation consistency axioms (47 and 46) with new axioms that distinguish between standard instantiation (axioms 78 and 80) and instantiation with extension (axioms 79 and 81). Basically, if an object x instantiates an object y with extension, x can either assign a value to the attributes inherited from y or redefine the type as an instance of the parent attribute’s type, i.e., the meta-type.

This simple redefinition works well for symmetric meta-types, i.e., where the number of levels of instantiation are the same for both sides of the attribute, but not for asymmetric meta-types. A key contribution of DDM [11] is that it supports asymmetric meta-types through *dual* potencies, i.e., separate source and target potencies. Asymmetric meta-types can be handled by SLICER, without the addition of potencies, by revising the replaced axioms (axioms 79 and 81) further to allow the type redefinition to

⁷Class and instance are simply roles that an object can play with respect to another object.

allow not only instances, but instances of instances (ad infinitum). Similarly, the specialisation axiom (43), which was unchanged for symmetric meta-types, would be replaced with one that allows the refinement of an attribute type to include instances, instances of instances, etc.

While these extensions increase the power of SLICER, they are not part of its core as we have not encountered a need for them in our Interoperability scenario. However, the ability to include them with localised additions to the formalisation demonstrates the flexibility of SLICER.

Inverse Attributes. In SLICER the attribute primitive is uni-directional; however, in many situations it is useful to define bidirectional attributes, where an attribute from an object o to another o' has an *inverse* (or *opposite*) attribute from o' to o . This applies to both the attribute signature of a class and the values of an attribute on an instance. The inverse is indexed by the object (axiom 82) so that the same attribute label can coexist in different object hierarchies with a “different” meaning, thus enhancing modularity. Unlike DDM [11] (among other conceptual modelling languages), we leave the flexibility not to enforce an implicit inverse for every attribute since there may be situations where this is not desirable.⁸ This allows attributes to be viewed as unidirectional or bidirectional as appropriate [40]. For example, there may be many attributes from various objects to another common object, such as Money or units of measure; however, in the context of the type Money there may be only a few attributes that are actually meaningful to it. Taking this stance on inverse attributes helps retain the flexible level stratification of SLICER while introducing level alignments in an intuitive fashion, i.e., when values are assigned with explicit inverse attributes specified. Therefore, instances of Money would align with other objects only if an explicit inverse attribute were specified, while the type can still be at a higher-level without violating any constraints. However, in an interoperability scenario, there may be some cases where an explicit inverse is necessary that would cause this constraint to be violated. Therefore, it can be relaxed on a case-by-case basis at the discretion of the interoperability designer.

Cardinality Constraints. Another important aspect of attributes in conceptual models is their cardinality constraints (or multiplicity). Up until now it has been assumed that attributes may have any number of values for an object, i.e., a cardinality of $\langle 0, \infty \rangle$. Similar to inverse attributes, the cardinality of an attribute is specific to the attribute label for a particular object. The number of values an object has for an attribute must be within the defined range of the cardinality constraint (axiom 87). Specialisations can refine the cardinality of an attribute to a narrower range (axiom 88), while the cardinality does not change when an attribute is instantiated (axiom 89). This is analogous to the refinement and instantiation of the type, see axioms 43 and 46. For example, a Vehicle may have an attribute *wheel* with cardinality constraint $\langle 1, \infty \rangle$, i.e., all vehicles have at least one wheel with no upper limit. A specialisation of Vehicle, say Car, may refine the cardinality constraint of *wheel* to $\langle 3, 4 \rangle$ to allow 3-wheeled and 4-wheeled cars, while a specific type of car may refine the cardinality further to specify the number of *wheels* to be exactly 4. When the particular type of car is instantiated, the value of its *wheel* attribute must be a set containing 4 distinct Wheel objects: any less or any more would violate the cardinality constraint.

In the presence of meta-attributes, cardinality constraints are propagated to the point where a value is assigned. Therefore, the cardinality is always applied consistently to attribute values. We believe this avoids confusion in the handling of cardinality across multiple levels (which other MLM approaches do not consider adequately), especially at intermediate levels involving specialisation. For example, if a cardinality is intended to mean the selection of a *single* type and an object with subclasses is selected, does this violate the cardinality constraint? In DDM, the specification of such cardinalities is supported to allow for processes at higher-levels of abstraction to require a certain number of types be selected. However, we consider such constraints to be part of the process specification, not the static model, as there may be many processes with different constraints. Moreover, in the design context, the actual selection is more likely to be constrained by other factors such as material compatibility, dimensions, etc.

⁸From the perspective of the implementation language the navigability of an attribute is always bidirectional

Subsetting Attributes. Similar to other conceptual modelling languages [34, 11], we introduce *subsetting* of attributes.⁹ This is simply the introduction of an inclusion constraint between a sub-attribute and a parent attribute (axiom 91), where *Subsets* is a relation between attribute labels in the context of an object (axiom 90) and is denoted by *sub-attribute/parent attribute* in diagrams. This has several implications that are consistent with what is expected, including that: the parent attribute is defined on the object or is inherited from another (axiom 92); the type of the sub-attribute must be the same as, or a specialisation of, the type of the parent attribute; and the maximum cardinality of the sub-attribute cannot exceed that of the parent attribute. Moreover, in the presence of inverse attributes, subset attributes are symmetric: i.e., the inverse of the parent attribute must be the parent of the inverse of the sub-attribute (axiom 94). Finally, an attribute may have any number of parent attributes and sub-attributes.

Derived Attributes. Another useful feature of conceptual models is derived attributes. These are easily handled in SLICER by the incorporation of constraints in the description of an object (see Section 4.1). (Remember that a key property of SLICER is to distinguish the role of descriptions in a conceptual model). We do not impose a particular language via the framework, so first-order logic, OCL, or others can be used assuming that access to the primitives of SLICER is maintained, i.e. the constraints can be written such that the values, instances, specialisations, etc. of an object can be accessed. For example, *CarModel* may have a derived attribute */enginePower* (the leading slash indicates a derived attribute) based on one or more attributes of an associated engine (type), assuming only a single engine, as in example (1). Moreover, derived attributes can be defined over attributes with multiple values, possibly aggregating the values in some fashion (e.g. min, max, average, and sum) as in example (2). The ease with which such constraints can be included in the description of an object will depend on the language being used.

```
context CarModel::enginePower : Integer
derive: self.engine.power
```

(1)

```
context CarModel::avgSalePrice : Integer
derive: self.allInstances().salePrice->sum() / self.allInstances()->size()
```

(2)

4.4. Relationships

While the (primitive) attributes of SLICER allow objects to be associated with other objects, they are not objects themselves. In contrast, a *relationship* is a connection between objects that is reified in the domain of discourse. Therefore, relationships can have attributes, specialise other relationships, have instances, categories, and specifications like standard objects. As such, relationships are not linguistic in nature (as they are in [42, 11]) but are ontological in nature (more like the *relators* of [43]). For example, while the *Connections* of [42] can have their own attributes, they are defined as a special type of *Clobject* that has property ends, etc. in a similar fashion to UML [34]. This linguistic treatment of relationships unnecessarily complicates the meta-model. In contrast to the use of relators when specifying reference ontologies [43], we do not enforce the identification of the reified relationship; instead, we allow them to be defined when it is conceptually relevant to do so as advocated by [40]. This approach is beneficial for developing conceptual models that integrate different perspectives in an inter-organisational setting as the difference between a relationship and object in one organisation vs. another is a matter of perspective that can be replicated via user views [40]. Another benefit is that n-ary relationships are represented without requiring any modifications to the framework.

An example relationship object is illustrated in Figure 7, where a *Sale* object *mediates* (using the terminology of [43]) the sale of *Cars* by *Organisations*. The direct attribute (and its inverse) between *Car* and *Organisation* is then a derived attribute based on attribute path *inSale.soldBy* through the *Sale* (illustrated using a grey dashed line). Due to the reciprocal nature of attribute values and those of their inverse, it is sufficient to define the derivation rule in only one direction: that is, defining the derivation through either *inSale.soldBy* or *makes.soldItem* is sufficient. The relationship could easily be extended to include the purchaser, e.g. by adding a *soldTo* attribute between *Sale* and *Organisation*. Moreover,

⁹Referred to as *specialisation* in [11]; however, we use the term subsetting to preserve the ontological distinction introduced in [41], i.e. subsetting can hold between attributes that are of different *kinds* of relationship.

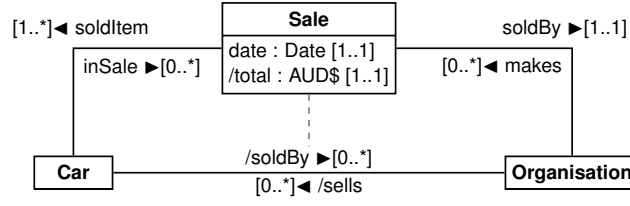


Figure 7: Reified Relationships Example

different types of sale can be represented, e.g. *CashSale*, as a specialisation of *Sale* per normal SLICER representation. While specialising a relationship is similar to subsetting [41] an attribute, there is not a one-to-one correspondence: subsetting attributes establishes a generic inclusion constraint, possibly between semantically different attributes (e.g. for *sellsTo/dealsWith* and *buysFrom/dealsWith*, the semantics of *dealsWith* is neither the semantics of *sellsTo* nor *buysFrom*, but a combination of the two), while a specialisation of a relation is still of the same type as the specialised object (i.e. a *CashSale* is still a *Sale*).

Use of reified relationships has also been proposed in the Object-Role Modelling (ORM) [44] approach. In ORM, attributes of any arity (i.e. unary, binary, ternary, etc.) are represented as reified relationships called *fact types*. Like UML, ORM is based on strict level separation, with schema and (example) instances represented separately. More importantly, SLICER allows choosing relationships or object/attribute representation, based on ontological considerations related to the intent and perspective of the model, permitting a more concise presentation as conditions warrant. For example, the relationship *Sale* as a quarternary fact type in ORM ‘*Organisation sells Car on Date for AUD\$*’ would be a concise verbal representation but abstracts from the concept of a *Sale*, which may be an important distinction in the models for which an interoperability solution is being developed. In contrast, considering *Sale* as an ORM object type (effectively reifying a reified relationship) would require no less than four fact types: *Organisation makes Sale*; *Sale has sold item Car*; *Sale is made on Date*; *Sale has price AUD\$*.

5. Comparison with Dual Deep Modelling

To illustrate the benefits of SLICER we perform a comparative evaluation against DDM [11] using a common example model. While we compare against the DDM approach, as it is the most flexible and complete of the potency-based MLM frameworks (see Section 7), we limit the comparison to aspects that are applicable across all potency-based methods. The comparison expands on SLICER core (refer Figure 4) by incorporating a more detailed view of attribute usage in the framework. Figures 8 and 9 illustrate a DDM model and a SLICER model, respectively, on an example adapted from the larger one used in [11]. The main differences between the two approaches include the:

- explicit distinctions of semantic relationships (e.g. specialisation with extension, specification) and types of object (e.g. specifications, and categories) in SLICER vs. no differentiation in DDM;
- the separation of related concepts as their own entities in SLICER vs. representing related concepts as hidden facets of a single entity using potencies;
- support for explicit complex constraints in SLICER vs. implicit constraints through attributes and their integrity constraints;

These differences have clear ramifications when considering the design of an interoperability solution for transforming between models of different origins. The remainder of this section discusses these differences in turn. In addition, we consider the possibility of a direct DDM-to-SLICER mapping to elaborate some of the distinctions.

5.1. Explicit Distinctions and Entity Separation

A goal of SLICER is to maintain related concepts as separate objects in the model, in contrast to DDM (and DI approaches in general) which collapse them into general concepts that hide the specific

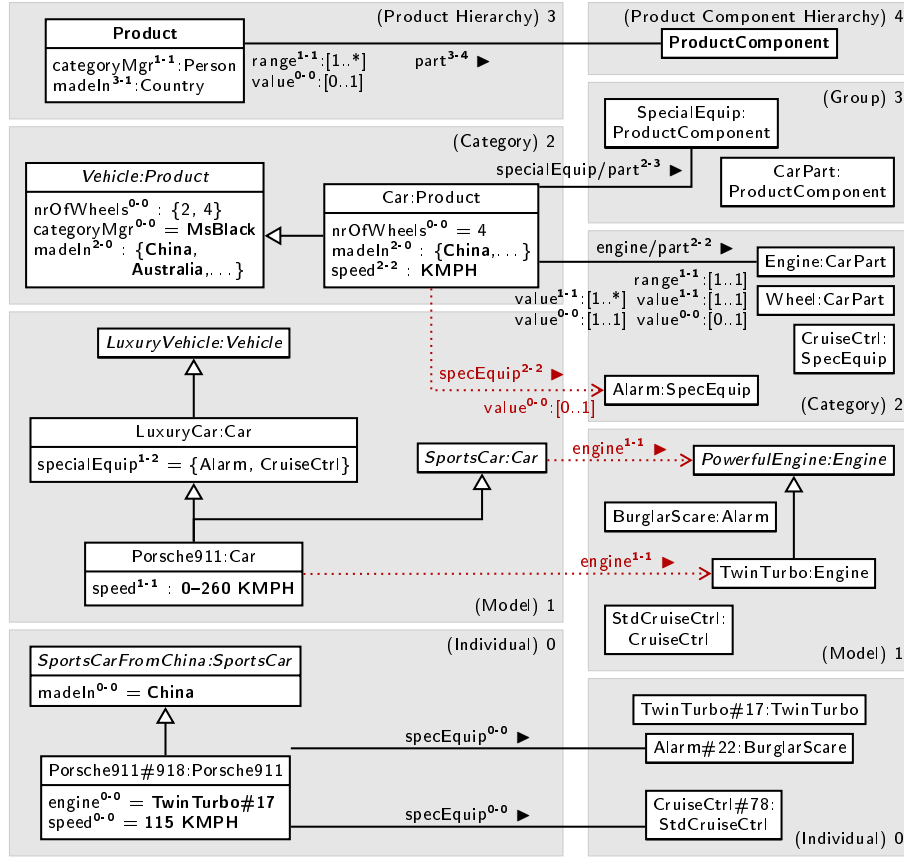


Figure 8: DDM Car Example. Red dotted lines indicate range restrictions.

semantics of the relations, behind potency values. For example, in Figure 8 the concept *Product* combines together the concepts of *Product*, *ProductCategory*, and *ProductModel*, which are illustrated individually in Figure 9 (except *ProductModel* which is illustrated through the more specialised concept *CarModel*). As such DDM assigns the attributes of ontologically distinct entities to the same model element. Moreover, the distinct entities in the SLICER model are of different types, e.g. specification types and categories, which highlights the role of the entity in the model and links them to related concepts with additional, and more fine-grained, semantic relationships rather than the usual instantiation, specialisation, and attributes/relationships.

The benefit of the DDM representation is a slightly more compact visual notation. In exchange, the model requires the specification of numerous textual annotations that are not visually expressed: Figure 8 contains 56 potency annotations. In the context of Ecosystem Interoperability, where different concepts of the model arise from different sources with some overlap, we prefer to maintain distinct entities and link them explicitly through the differentiated relations and constraints. The overall complexity of the model can be reduced by using the finer semantic distinctions between relationships to develop user views as necessary. Similar considerations apply to product lifecycles, where instance information in earlier stages can turn into descriptions at later stages. Since levels in SLICER are derived from the use of relationships, they do not require consistent potency annotations to be predicted at the start of the lifecycle.

5.2. Explicit Complex Constraints

SLICER puts explicit complex constraints in the core of the framework as their use are central and inevitable in the engineering domain within which we work and conceptually provide a framework for the incorporation of business rules in business domains. Moreover, explicit constraints help link aspects of the different models for which we wish to design a joint meta-model. For example, the constraint between the *speed* of an actual car and its *maximum speed* (analogous to the *temp/maxTemp* constraint discussed earlier for pumps) can be defined on *CarModel* with respect to its related type *Car* (refer middle-left of

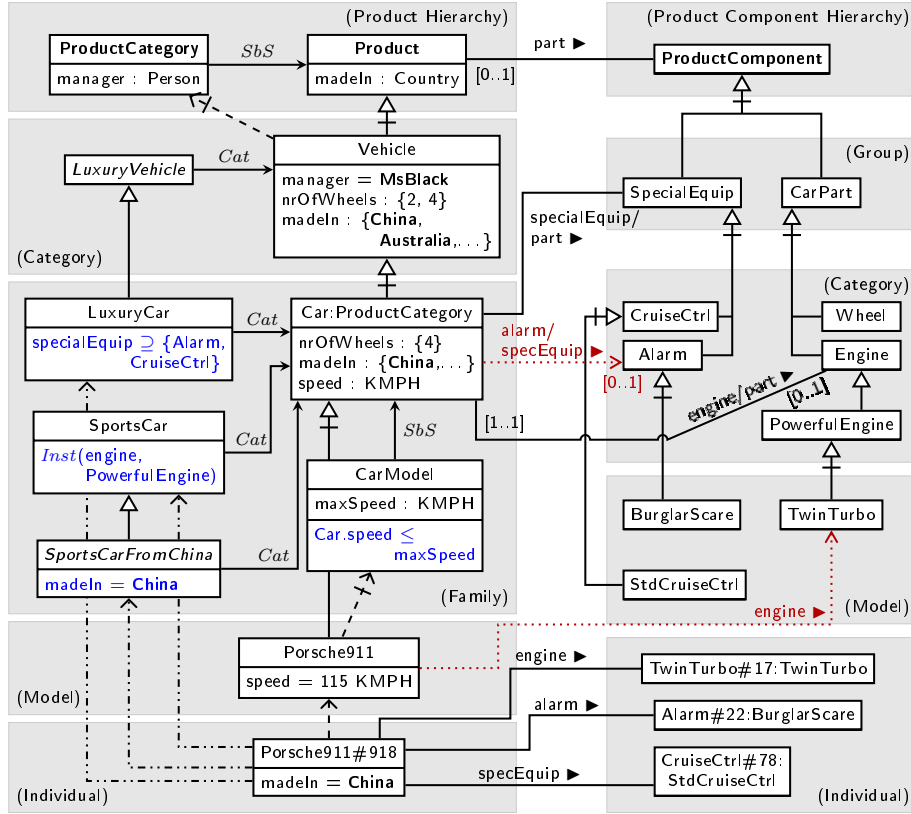


Figure 9: SLICER Car Example. Red dotted lines indicate range restrictions, blue elements represent membership constraints.

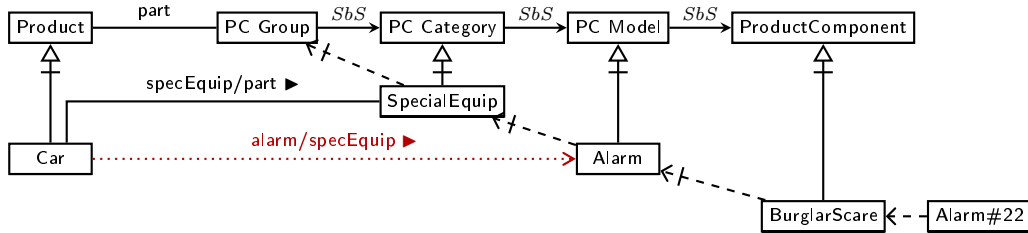


Figure 10: Direct mapping from DD to SLICER *SbS* chain

Figure 9. This constraint will be propagated down to the physical car where it can be evaluated. The same constraint in DDM would be modelled implicitly through refinement of the allowed range of the *speed* attribute of Car (see middle-left of Figure 8).

The DDM approach provides a simpler representation in a top-down designed structure. However, consider the situation where the two attributes originate from different sources and the constraint helps validate the input (or output) data during transformation, e.g. a value for *speed* that exceeds the maximum specified in the other model might suggest a data entry or transformation error. Since neither model individually contains the constraint, such a violation would go undetected. In practice, properties such as *maximum speed* are commonly made explicit on engineering data sheets, but would be hidden in the DDM approach as the top-end of the range constraint on the related attribute *speed*.

5.3. Direct DDM to SLICER Mapping

It is possible to perform a more direct mapping between DDM and SLICER models by treating the DDM level hierarchy as a chain of *SbS* relations. For example, Figure 10 illustrates the expansion of the ProductCategory hierarchy as a chain of *SbS* relations. This makes the facets hidden by the DDM model explicit while maintaining the defined level structure. For the SLICER model to be valid it requires the meta-attribute extensions discussed in Section 4.3.

Typically, though, a direct mapping is undesirable as it enforces the DDM level structure on the model, rather than making correct use of the various elements of SLICER. For example, consider a situation where models from different sources overlap on the type and models of product component, i.e. Alarm, BurglarScare, CruiseCtrl, etc., but neither source model has both the product hierarchy (e.g. Car) and the business categories for components (e.g. SpecialEquip). As a result, the concept Car provides a *specEquip* attribute with a range of ProductComponent due to the absence of a SpecialEquip type or category. Using SLICER, the common aspects can be integrated while maintaining the distinctions of the disparate elements. To tie the models together more strongly, a constraint can be incorporated into the joint (meta-)model such that the *specEquip* attribute (of one source model) should have a type from the SpecialEquip category (of the other model).

6. Mapping using SLICER Distinctions

A key factor in this work is in identifying and explicitly representing the distinguishing factors of relationships: extension, refinement, instantiation, specification, and categorisation. These relationships can be considered as *meta-properties* that annotate or enrich the families of relationships existing in conventional models. This can be seen as an extension of the classical meta-properties used in OntoClean [45]: Unity (U), whether a concept has wholes as instances and criteria that apply to its parts; Identity (I), whether a concept carries criteria for determining if two instances are the same; Rigidity (R), whether instances of the concept must always be an instance of the concept; and Dependence (D), whether the concept's instances are dependent on instances of another concept. Rather than applying to concepts, the additional meta-properties are applied to relationships instead.

6.1. Meta-Property Identification

The new meta-properties, Extension (X), Refinement (Rf), Instantiation (Inst), Specification (S), and Categorisation (C), can be used to identify patterns of meaning in models to better map between them. In the case of Extension, it can be identified by examining the property specifications of the related objects. The presence, or absence, of Extension in a relationship can be indicated by +X and -X, respectively. Similarly, Refinement can be identified through the analysis of property specifications; however, the absence of Refinement does not prove much as additional subclasses may be added to allow for the extension of the vocabulary even if the differences (i.e. the refinements) are not modelled. Therefore, we indicate inheritance without explicit refinement as $\pm X \sim Rf$ (extension may or may not be present), whereas no inheritance at all would be indicated by -X-Rf. Instantiation can be identified by the change of a property specification from defining a type to providing a (set of) value(s).

Specification (S) may be identified by, at best, a generic domain association as part of the generic Dependence meta-property, e.g., instances of Pump being dependent on instances of Pump Model through a *model* attribute. At worst, no direct relationship exists between the concepts as is the case in some powertyping approaches where the association is formed by using the same name for the class and instance objects. However, Specification could be inferred from a pattern of relationships carrying the X, Rf, and Inst meta-properties.

The Categorisation (C) meta-property is somewhat harder to identify as it is based on its intended meaning and can be modelled in many ways, such as: (1) multiple inheritance, Figure 11a shows the usage of multiple inheritance to indicate the model C12 Kerosene Pump is in the UNSPC Pumps category (technically this is incorrect as it states that the individual pumps are *in* the category, however it is a two-level work around that occurs in practice); (2) (dynamic) multiple classification, Figure 11b illustrates multiple classification at the model level, although still applied to individual pumps assuming a two-level definition of EntityType and Entity at the meta-model level; or (3) a domain relationship indicating membership to the category possibly defined as aggregation, Figure 11c illustrates such an approach, which may identify the categories explicitly but is often limited to explicit inclusion by a user and may not indicate the type of objects in the category (other than what is defined at the meta-level). These different manifestations may appear at the meta-model and/or model levels.

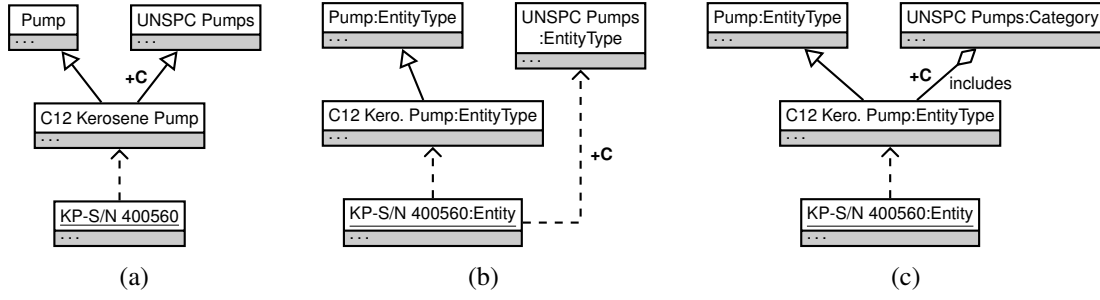


Figure 11: Different manifestations of Categorisation: (a) multiple inheritance, (b) multiple classification, and (c) explicit relationship. A common notation is used to illustrate the structure although the specialisation/instantiation relationships may actually be domain relationships defined in the meta-model.

Automated support for the consistent identification and use of these relationships can be provided through the use of patterns based on these meta-properties. Whereas de Lara et al. [46] illustrate modelling patterns suited to multi-level modelling (e.g. Type-Object, Dynamic Features, Element Classification) and approaches to redesigning two-level models into multi-level models, we aim to utilise meta-property based patterns to aid the development of model transformations. This is necessary as the different models within the ecosystem cannot be redesigned in most cases. Moreover, the primary models of interest in our domain (MIMOSA CCOM [4] and ISO 15926 [3]) contain complex combinations of the multi-level modelling patterns such that instance models contain dynamic types, entities, attributes, and inheritance relationships. Therefore, the identification of meta-properties at both the meta-model and model levels is important for designing model transformations that must take into account both.

6.2. Meta-Property Application to MLM Approaches

Using the semantic distinctions provided by the meta-properties, other models of an ecosystem (both two-level and multi-level modelling approaches) can be analysed to better identify mappings between them. Figure 12 displays a side-by-side comparison of two alternative models with our framework serving as the joint metamodel: a potency-based model on the left, SLICER model in the centre, and a powertype-based model on the right. The relationships are annotated with the identified meta-properties. For brevity, we only discuss the more interesting case of potency-based models due to the greater mismatch caused by the collapse of multiple domain entities into a single (top-level) concept, illustrated by the red frames.

The distinctions between Extension, Refinement, Categories, and Specifications that are made in SLICER are apparent in the potency-based model through:

1. Attributes with potency = 1 indicates +Inst as it must have a value assigned when it is instantiated.
2. An attribute with potency ≥ 2 suggests +X as the potency indicates that the attribute should be introduced to the concept at the level where its potency = 1 (so that it can be given a value at the next instantiation).
3. An object with potency = 0 and attributes with only potencies = 0 is *InstN*.
4. A subclass that introduces new attributes over its parent class suggests +X, while
5. A subclass that refines the domain of an attribute would be assigned +Rf; this is a direct corollary of the same distinction made in SLICER .
6. If all of the instances of an object are specialisations of the same *abstract* class (indicated by a potency = 0 that disallows direct instances) it suggests a relation with +S exists between the instantiated class and the specialised class.¹⁰ In particular, if the specialised class has attributes with potency > 0 or the instantiated class has attributes with potency = 2 it is most likely the case.

¹⁰In the standard DI example no relationship exists between Pump and PumpModel, for example; however, with DDM [11] Pump could be made an *abstract instance* of PumpModel.

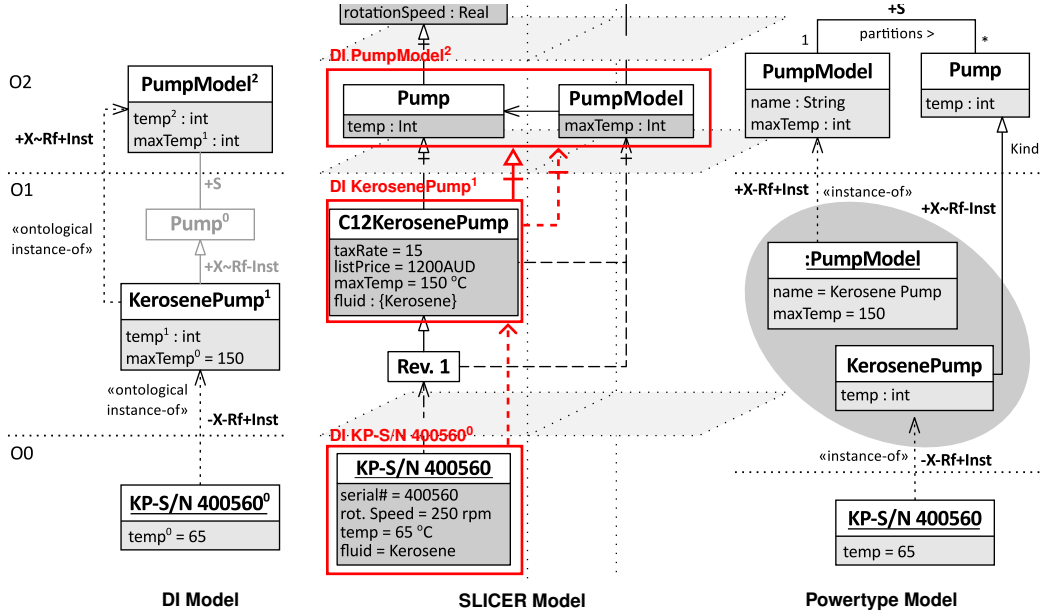


Figure 12: Example identification of relation meta-properties and alignment between multi-level models. Greyed out elements indicate objects/relations that may or may not be present. Red frames illustrate the mapped DI groupings.

7. Specialised classes that have no attributes with a potency ≥ 1 and that are not identified as related to a specification class (see previous rule) suggests possible use of specialisation for categorisation. The introduction of attributes indicates the boundary between categories and categorised object(s).

Applying these rules to the example (left-hand side of Figure 12) we can identify *InstN* by rule (3) between the object at *O0* on the left and KerosenePump, hence it can be aligned with the bottom level of the joint metamodel. At the *O1* level, KerosenePump instantiates PumpModel from the level above which includes an attribute with potency 2; thereby identifying *InstX* by rule (2). Moreover, by rule (4) a *SpecX* relation is identified as KerosenePump adds attributes over the class it specialises (Pump). Finally, all of the instances of PumpModel are subclasses of Pump, matching rule (6) and indicating the relation *SbS*(PumpModel², Pump⁰). As a result, this pattern can be matched in the joint metamodel. Note that the result would be the same if the attribute *temp*² were defined on Pump as *temp*¹. Finally, if the concept Pump were not modelled at all (i.e. it was completely incorporated into PumpModel) then the specialisation to some concept, along with an *SbS* relation, could be inferred due to the combination of specialisation and instantiation embodied by potency. If that concept did not already exist in the joint metamodel, a new concept could be created for each group of attributes with the same potency greater than one and stacked using *SbS* relations as discussed in the direct DDM to SLICER mapping (see Section 5.3). This would make the different facets explicit as desired, simplifying the mapping to similar concepts of the other models in the ecosystem.

6.3. Industry Ecosystem Mapping Example

A more complex example can be demonstrated using one of the models used in our interoperability scenario, MIMOSA CCOM [4]. CCOM is a two-level domain model defined in UML using XML for information exchange, exhibiting some of the multi-level patterns of [46]. Therefore, the identification of the meta-properties involves both the meta-model and some example model instances, or reference data. An example of meta-property identification and the resulting mappings is shown in Figure 13 using a condensed notation for CCOM. In the figure, type names after a colon (':') are from the meta-model. The example focuses on the Assets (representing individual assets) and their types (i.e. AssetType).

As expected, an Asset can have an AssetType (via the *hasType* association) selected from a taxonomy defined through the *hasSuperType* association. Therefore, the meta-model associations are annotated with the +Inst and $\pm X \sim Rf$ meta-properties, respectively. CCOM also utilises a Model concept to represent

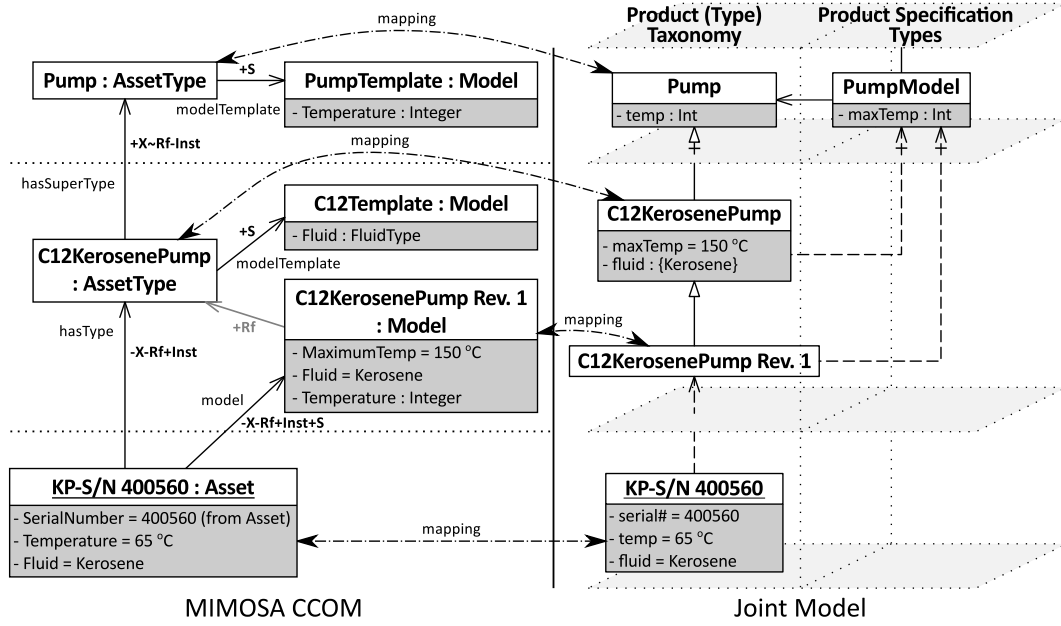


Figure 13: Mapping Example using meta-properties to identify correspondences

specific (revisions of) models of assets. This concept has a dual role in that it is also used to define the attribute specifications of AssetTypes through the *modelTemplate* association. Moreover, the set of attributes on an Asset is considered the merger of the attributes defined on the *modelTemplates* and the specific *model*. This merging of attribute specifications amounts to inheritance between AssetTypes and Model of an Asset, through the *hasType* and *model* associations, respectively. Moreover, the *model* relationship is identified as another instantiation relation along with *hasType*.

The way attribute specifications are merged between AssetTypes also suggests Extension, hence, *hasSuperType* is associated with +X through examination of the specific model instances. In contrast, the attributes of the *model*, excluding those assigned values, are intended to be all of those defined in the *modelTemplates* of the associated Asset, hence, a Refinement (+Rf) relation is inferred: shown in grey in the figure. This relation is inferred only for those Models in the *model* association with an Asset, not those being used as a *modelTemplate*. This is an important aspect of applying meta-properties to the relations, not just concepts as it allows patterns to be identified based on *how a particular instance* is used.

Finally, the AssetType/Model distinction and the pattern of meta-properties implies the existence of Specification (+S). The meta-property assignments result in the mappings illustrated in Figure 13.

7. Evaluation and Comparison

The new relationships and relationship variations in SLICER were developed by examining special cases that are known to cause problems in practice and which obscure modelling concerns with generic domain associations, naming conventions, or both, resulting in highly complex and often inconsistent models [47]. The goal of this approach is to enable development of coherent models in domain areas where persistent and large scale effort has *failed* to produce workable models since the modelling approaches used did not permit expressing the salient features of the domain. With these general considerations in mind, the remainder of this section presents a comparison of SLICER to other MLM approaches, according to the general requirements of MLM languages proposed by Frank [21], and followed by a more detailed comparison using an extension of the the criteria from [6].

7.1. General MLM Language Requirements

Through his work in developing Domain Specific Modelling Languages, Frank [21] arrived at a set of requirements that MLM languages need to fulfil to overcome the limitations imposed by the traditional

two-level modelling paradigm. The requirements are presented below, along with how they are fulfilled by SLICER: requirements denoted by *RLA* are related to language architectures, while *RI* requirements relate to the implementation of tools, and finally *RB* requirements are related to bridging the gap between the traditional paradigm and the multi-level approach.

RLA-1 *Flexible number of classification levels*

As discussed in the MLM literature, there is a need for conceptual models to break away from a fixed number of (classification) levels to support an arbitrary number.

The general definition of objects in SLICER allows arbitrary (classification) levels.

RLA-2 *Relaxing the rigid instantiation/specialisation dichotomy*

The reuse enabled by instantiation and specialisation needs to be allowed to be combined in an MLM language to support reuse across classification levels.

In SLICER, the introduction of extension to instantiation incorporates aspects of specialisation.

RLA-3 *No strict separation of language levels*

Models must be able to contain elements on different levels of classification, rather than separate models representing different levels.

SLICER (meta-)models contain objects at different levels.

RI-1 *Straightforward representation of language architecture*

The models of a MLM architecture should be incorporated into a single tool/modelling environment such that the different levels of classification are represented uniformly.

SLICER's implementation in F-logic provides a common representation of models and code.

RI-2 *Cross-level integrity*

The modelling tool/environment must support maintaining the integrity of models across multiple levels in the presence of updates. Updates should be propagated to all levels automatically.

SLICER models are within a single environment, allowing changes to be propagated automatically. While user intervention is required for deletion, this is desirable to a certain extent. Moreover, cross-level integrity constraints identify inconsistencies caused by deletions, which helps guide the user; therefore, we consider this full support for cross-level integrity.

RI-3 *Cross-level navigation*

The modelling environment of an MLM approach should support navigation across the entire hierarchy of levels in a multi-level model.

All objects in SLICER exist within in the same environment allowing navigation in any direction.

RB-1 *Clear specification of classification levels*

Since models may contain elements at different levels of classification, the language must provide means to identify/specify the level to which an element belongs.

Every SLICER object has an associated level.

RB-2 *Backward compatibility*

It is important that the MLM approach be able to integrate existing (meta-)models as there are many already in existence.

Traditional conceptual modelling features are a subset of SLICER . Furthermore, many of the finer distinctions made in SLICER can be identified in existing (meta-)models.

These requirements outline the general needs of an MLM approach and its associated implementation and can be used to compare the different approaches. However, in [21], only a comparison between FMML^X and traditional, two-level, approaches is performed. In Table 2, we expand on the comparison to include SLICER as well as other MLM approaches. In general, we found that the current crop of MLM approaches adequately fulfil all of the requirements: the exceptions being Materialization, which lacks

Table 2: Summary of MLM approaches fulfilment of general MLM requirements

Approach	Requirement	RLA-1	RLA-2	RLA-3	RI-1	RI-2	RI-3	RB-1	RB-2
FMML ^x [21]		+	+	+	+	~	+	+	+
MOF/UML		–	~	–	–	–	–	+	n/a
SLICER		+	+	+	+	+	+	+	+
Deep Instantiation [8], Melanee [9], MetaDepth [48]		+	+	+	+	~	+	+	+
DDI [20, 49], DDM [11]		+	+	+	+	+	+	+	+
Powertypes [19, 33]		+	~	~	–	–	–	~	+
Materialization [27]		+	+	+	n/a	n/a	n/a	–	+
M-Objects [15]		+	+	+	~	~	+	+	–

Legend: (+) Full Support, (~) Partial Support, (–) No Support

an implementation, and Powertypes, which typically inherit the limitations of the traditional modelling approach in which they are embedded.

7.2. Detailed Comparison Criteria

A previous comparison of MLM approaches [6] provides specific criteria for making domain models more concise, flexible, and simple. Meeting these criteria aids in reducing accidental complexity in the sense of using a minimal set of elements to model a domain [12]. However, the criteria do not cover certain aspects important for defining joint metamodels in an interoperability scenario as in the OGI Pilot—except for criterion #2, which is necessary for multiple classification hierarchies. We therefore extend the comparison with additional criteria (#5–7) to better capture these requirements.

1. **Compactness** encompasses *modularity* and *absence of redundancy*.

- (a) An approach is considered *modular* if all of the aspects related to the representation of a domain concept can be treated as a unit. For example, product category, (product) model, and (product) physical entity are grouped together to as a representation of the concept ‘Product’. In the context of the Ecosystem Management Scenario and interoperability such modularity is undesirable as it encourages a modelling approach that confounds aspects of multiple concepts into a single concept. While this may be beneficial when enforcing a top-down structure on an information system, it does not support the integration of domain concepts that are considered disparate when viewed from the perspectives of different users or when integrating models that differ in granularity or scope. In such a situation it is desirable to be able to maintain the distinctions made between related, but different, concepts of the models being integrated. Therefore, we introduce the *locality* criterion as an alternative (see #5).
- (b) *Redundancy-free* means that information (e.g. concepts, attributes) is represented only once.

SLICER supports redundancy-free modelling (and partly modular modelling) using a range of relations that include various attribute propagation, inheritance, and assignment semantics.

2. **Query Flexibility** means that queries can be performed to access the model elements at the different levels of abstraction (e.g. querying for all of the product categories, models, physical entities, and their specialisations).

Query flexibility in SLICER is supported across concepts and attributes in a domain model.

3. **Heterogeneous Level-Hierarchies** allow an approach to introduce “new” levels of abstraction for one (sub-)hierarchy without affecting another (e.g. introducing a PumpSeries level of abstraction in between PumpModel and PumpPhysicalEntity without causing changes to any other hierarchy).

SLICER allows heterogeneous level-hierarchies through specific primitive semantic relations and its flexible, dynamic approach to level stratification.

4. **Multiple Relationship-Abstractions** include classification of relationships, querying and specialisation of relationships (and their classes and meta-classes). Relationship-Abstractions include: (a) the classification of relationships by relationship classes and meta-classes, (b) the ability to address and query relationships, relationship classes and relationship meta-classes at different levels, and (c) the specialisation and/or instantiation of domain and range of relationships, either through classification or constraints.

SLICER supports specialisation and instantiation of domain and range relationships, while meta-classification of relationships is supported at the attribute level, through the small extensions discussed in Section 4.3, and at the relationship level, as a natural consequence of supporting reified relationships as any other concept.

5. **Locality** is supported by an approach that can define attributes, relations, and constraints on the model element closest to where they are used (e.g. an attribute most relevant to models or designs themselves should be situated on the concept `ProductModel` rather than some related concept such as `Product`, or vice versa).

Locally specified attributes and relations with intuitive constraint propagation is supported by SLICER's design.

6. **Decoupling of Relationship Semantics** occurs in an approach if they have clearly delineated semantics from one another rather than combining the semantics of multiple, commonly understood relations. For example, the combination of common relation semantics, such as instantiation and specialisation, into a single relation may cause confusion for the interoperability designer when dealing with the model as a whole (as opposed to a specific view) as the relationship must be interpreted in a multiple ways.

SLICER extends the set of common key relations between domain concepts by several novel relation types. Note that while instantiation with extension incorporates some aspect of specialisation, SLICER clearly distinguishes it as necessary relation to support multi-level models. Having a larger set of semantically differentiated relations makes it easier to perform “*sanity checks* regarding the integrity of metamodeling hierarchies” as described in [50].

7. **Multiple Categorisation** considers whether or not the modelling approach can support an element being placed in multiple (disjoint) categories. Since an item could simultaneously belong to multiple categories, it is particularly important for business level classification. For example, a pump model being certified by two different standards would place it in the categories for both.

SLICER supports multiple categorisations through explicit representation of categories for which objects can belong to multiple concurrently.

A summary of the comparison of the MLM techniques with respect to the 7 criteria defined above is provided in Table 3. The comparison extends that of [6] by incorporating a couple of specific potency-based approaches (MetaDepth, Dual Deep Instantiation, Dual Deep Modelling), an ontological power typing approach (Powertype (Onto.)), and FMML^X. While we focus on the approaches proposed as MLM solution, the table incorporates other approaches (such as Materialization, HERM, and Component Model) for completeness with respect to the original comparison.

8. Conclusion

Effective exchange of information about processes and industrial plants, their design, construction, operation, and maintenance requires sophisticated information modelling and exchange mechanisms that enable the transfer of semantically meaningful information between a vast pool of heterogeneous information systems. This need increases with the growing tendency for direct interaction of information systems from the sensor level to corporate boardroom level. One way to address this challenge is to provide more powerful means of information handling, including the definition of proper conceptual models for industry standards and their use in semantic information management and transformation.

Table 3: Summary of comparison in the interoperability context (extended and adapted from [6])

Approach	Criterion #	1a	1b	2	3	4a	4b	4c	5	6	7
MOF/UML		+	-	+	$\sim/+^1$	-	-	$+^1$	+	\sim	+
Deep Instantiation [8]		+	+	+	-	-	-	+	\sim	\sim	\sim
Melanee [9]		+	+	+	-	\sim	\sim	+	\sim	\sim	\sim
MetaDepth [48]		+	+	+	+	\sim	\sim	+	\sim	\sim	\sim
Dual-Deep Instantiation [20, 49]		+	+	+	+	+	+	+	\sim	-	-
Dual-Deep Modelling [11]		+	+	+	+	+	+	+	\sim	-	\sim
Powertypes (Simple) [19]		\sim	-	\sim	-	-	-	$+^1$	+	\sim	\sim
Powertypes (Ext.)		\sim	-	+	+	-	-	$+^1$	+	\sim	\sim
Powertype (Onto.) [33]		\sim	-	\sim	+	-	-	\sim^1	+	+	-
Materialization [27]		+	+	\sim	-	\sim	-	\sim	-	\sim	-
M-Objects [15]		+	+	+	+	+	+	+	-	-	-
HERM		\sim	+	$-^2/+^3$	$+^2/-^3$	\sim	+	\sim	+	$-^2/\sim^3$	-
Component Model		\sim	+	\sim	+	+	+	+	+	\sim	-
FMML ^X [21]		+	+	+	-	\sim	\sim	+	\sim	\sim	-
SLICER		\sim	+	+	+	+	+	+	+	+	+

Legend: (+) Full Support, (\sim) Partial Support, (-) No Support¹using OCL; ²specialization schema; ³overlay schema

In this paper we have described the SLICER framework for large scale ecosystem handling. It is based on the notion that a model-driven framework for creating mappings between models of an ecosystem must identify its underlying semantics based on the relationship between an entity and its description, most clearly embodied by artefacts subject to their specifications. This led to the introduction of a set of primitive relationship types not so far separately identified in the literature. For one, they reflect the fundamental conceptual modelling distinction between extension and refinement [25], thus allowing to succinctly express distinct semantics encountered in multilevel and multi-classification modelling scenarios. On the other hand, they represent the first time elevation of the *specification* relationship to full “citizenship” status among the relationships in the conceptual model. Together these relationships capture underlying design assumptions encountered in various potency and powertype approaches and enable their separate examination and study. This allows the expression of complex mappings for large scale interoperability tasks in a consistent and coherent manner. The identification of extension and specification/description as ontologically relevant meta-properties offers as a connection to our work on artifact and specification ontologies [23]. Future work includes the extension of the underlying formal framework, investigation of the above mentioned connections to formal ontological analysis, an ontological analysis of meta-attributes, and evaluation of the framework with external users with particular focus on automated meta-property identification in the development of model transformations.

Appendix A. Framework modelling primitives, predicates, and rules

Primitive Signatures

Domains:

- | | | |
|-----|--------------|--|
| (1) | O | Set of objects |
| (2) | L | Set of attribute labels |
| (3) | \mathbb{N} | Natural numbers |
| (4) | S | Constraint language over attributes in L plus the relations <i>InstN</i> , <i>InstX</i> , <i>SpecR</i> , <i>SpecX</i> , <i>Member</i> , <i>Cat</i> , <i>SbS</i> , and their generalisations (defined below). |

Functions:

- | | | | |
|------|----------------|--------------------------|--|
| (5) | <i>level</i> : | $O \mapsto \mathbb{N}$ | The level at which an object is defined (zero denotes the top level) |
| (6) | <i>attr</i> : | $O \mapsto 2^L$ | The set of attribute labels for an object |
| (7) | <i>type</i> : | $O \times L \mapsto O$ | The type of an attribute of an object |
| (8) | <i>val</i> : | $O \times L \mapsto 2^O$ | The value(s) of an attribute of an object |
| (9) | <i>desc</i> : | $O \mapsto 2^S$ | The set of constraint expressions an object must satisfy |
| (10) | <i>names</i> : | $S \mapsto 2^L$ | The attribute labels used in a constraint expression |

- (11) $names : 2^S \mapsto 2^L$ The attribute labels used in a description (for convenience)
 $\phi = desc(x) \rightarrow names(\phi) = \bigcup_{\psi \in \phi} names(\psi)$

Relations:

- (12) $InstN \subseteq O \times O$ $InstN(x, c)$: x is a standard instance of c
(13) $InstX \subseteq O \times O$ $InstX(x, c)$: x is an instance-with-extension of c
(14) $SpecR \subseteq O \times O$ $SpecR(c, c')$: c is a specialization-by-refinement of c'
(15) $SpecX \subseteq O \times O$ $SpecX(c, c')$: c is a specialization-by-extension of c'
(16) $Member \subseteq O \times O$ $Member(x, c)$: x is in a *Member* relation with c
(17) $Cat \subseteq O \times O$ $Cat(c, c')$: c is a *category* for c'
(18) $SbS \subseteq O \times O$ $SbS(c, c')$: c is in a *Subset-by-Specification* relation with c'
(19) $Covers \subseteq O \times O$ $Covers(c, c')$: c is a specification type whose instances define subsets that cover the referred type c'
(20) $Disjoint \subseteq O \times O$ $Disjoint(c, c')$: c is a specification type whose instances define subsets that are disjoint from one another in c'
(21) $Partitions \subseteq O \times O$ $Partitions(c, c')$: c is a specification type whose instances define subsets that both cover c' and are disjoint from one another
(22) $From \subseteq O \times L \times O$ $From(x, a, c)$: the attribute a of x is inherited from c
(23) $From \subseteq O \times S \times O$ $From(x, \phi, c)$: the constraint ϕ of x is inherited from c

We use $(\rho^*) \rho^+$ to denote the (reflexive-)transitive closure of relation ρ .

Definitions

- (24) $Spec(c, c') \leftrightarrow SpecR(c, c') \vee SpecX(c, c')$ (The *Spec* relation generalises the two forms of specialisation)
(25) $Inst(x, c) \leftrightarrow InstN(x, c) \vee InstX(x, c)$ (The *Inst* relation generalises the two forms of instantiation)
Object x is a general instance of c iff it is (or specialises) an instance of (a specialisation of) c
(26) $GenInst(x, c) \leftrightarrow \exists x' \exists c' : Spec^*(x, x') \wedge Inst(x', c') \wedge Spec^*(c', c)$
An object is *Directly Instantiable* iff it has no specialisations and it is not a standard instance of an object
(27) $DirectInstbl(c) \leftrightarrow \nexists x : Spec(x, c) \wedge \nexists y : InstN(c, y)$

Axioms for specialization and instantiation

- (28) $(Inst \cup Spec \cup Member)^+(x, c) \rightarrow x \neq c$ (*Inst*, *Spec*, *Member* are jointly acyclic)
(29) $\rho(x, c) \rightarrow \nexists c' : \sigma(x, c')$, and $\sigma(x, c) \rightarrow \nexists c' : \rho(x, c')$ (*InstN* and *InstX*, *SpecR* and *SpecX*, *SbS* and *Cat* are mutually exclusive)
(30) $Spec(x, c) \wedge Spec(x, c') \rightarrow c = c'$ (*Spec* restricted to a unique parent object)

Inst restricted to a unique parent object, except where there are multiple specifications on the same type

- (31) $Inst(x, c) \wedge Inst(x, c') \wedge \nexists a, b, t : Inst(c, a) \wedge SbS(a, t) \wedge Inst(c', b) \wedge SbS(b, t) \wedge a \neq b \rightarrow c = c'$
(32) $Inst(x, c) \rightarrow DirectInstbl(c)$ (Only *Directly Instantiable* objects can have instances)

Only *Directly Instantiable* objects and subset specification types can have *InstX* instances

- (33) $InstX(x, c) \rightarrow DirectInstbl(c) \vee \exists c' : SbS(c, c')$
(34) $Member(x, c) \rightarrow \exists c' : Cat(c, c')$ (Only categorisations can have members)

Level consistency:

- (35) $Inst(x, c) \rightarrow level(x) < level(c)$ (Level consistent with *Inst*, *Spec*)
(36) $SpecR(x, c) \rightarrow level(c) \leq level(x)$ (*SpecR* must remain consistent with level order)
(37) $SpecX(x, c) \rightarrow level(c) < level(x)$ (*SpecX* introduces a new level)

Axioms for schema consistency

- (38) $a \in attr(x) \wedge t = type(x, a) \rightarrow level(t) \leq level(x)$ (Attribute type must be consistent with level order)
(39) $a \in attr(x) \wedge v \in val(x, a) \rightarrow level(v) \leq level(x)$ (Value must be consistent with level order)
(40) $t = type(x, a) \rightarrow \exists c : Cat(t, c) \vee \nexists c' : InstN(t, c')$ (A type must be able to have members or instances)
(41) $SpecR(x, c) \rightarrow attr(c) = attr(x) \wedge (\forall a \in attr(c) \rightarrow From(x, a, c))$ (*SpecR* does not change attribute set)
(42) $SpecX(x, c) \rightarrow attr(c) \subset attr(x) \wedge (\forall a \in attr(c) \rightarrow From(x, a, c))$ (*SpecX* extends attribute set)

Spec may specialise attribute types including categories whose base type specialises the type

- (43) $Spec(x, c) \wedge a \in attr(x) \cap attr(c) \wedge t = type(x, a) \wedge t' = type(c, a) \rightarrow Spec^*(t, t') \vee (Cat(t, t') \rightarrow Spec^*(t', t'))$

InstN does not change attribute set, excluding attributes with values

- (44) $InstN(x, c) \rightarrow \{a \mid a \in attr(c) \wedge \nexists v : v = val(c, a)\} = attr(x) \wedge (\forall a \in attr(x) \rightarrow From(x, a, c))$

InstX extends attribute set, excluding attributes with values

- (45) $InstX(x, c) \rightarrow A = \{a \mid a \in attr(c) \wedge \nexists v : v = val(c, a)\} \wedge A \subset attr(x) \wedge (\forall a \in A \rightarrow From(x, a, c))$
(46) $Inst(x, c) \wedge a \in attr(x) \cap attr(c) \rightarrow type(x, a) = type(c, a)$ (*Inst* does not change attribute type)
(47) $Inst(x, c) \wedge a \in attr(c) \wedge t = type(x, a) \wedge val(c, a) = \emptyset \rightarrow val(x, a) \neq \emptyset \wedge (v \in val(x, a) \rightarrow GenInst(v, t) \vee Member(v, t))$ (*Inst* instantiates all (unassigned) attributes of the type)

- (48) $Spec(c, c') \wedge GenInst(c, t) \wedge GenInst(c', t) \wedge From(c, a, t) \wedge From(c, a, c') \wedge V = val(c', a) \wedge V \neq \emptyset$
 $\rightarrow (V = val(c, a) \oplus (\exists O : O = val(c, a) \wedge O \neq V))$
 (Sub-types can inherit values from super-types or they can have their own values.)

Axioms for Subset by Specification

- (49) $SbS(c, c') \rightarrow level(c) = level(c')$ (Subset by specification must be consistent with level order)
 The instances of each subset specification must be a specialisation of the partitioned type
 (50) $SbS(c, c') \wedge Inst(x, c) \rightarrow Spec^+(x, c')$
 The specification type may refer only to the attributes of the specification and the partitioned type
 (51) $SbS(c, c') \wedge \phi = desc(c) \rightarrow names(\phi) \subseteq (attr(c) \cup attr(c'))$
 (52) $Covers(c, c') \rightarrow SbS(c, c')$ (Covers and Disjoint are Subset-by-Specification relations)
 (53) $Disjoint(c, c') \rightarrow SbS(c, c')$ (No description)
 (54) $Partitions(c, c') \rightarrow Covers(c, c') \wedge Disjoint(c, c')$ (Partitions is both a Covers and Disjoint relation, and therefore a SbS relation)

The subsets denoted by the instances of a covering type cover the referred type

- (55) $Covers(c, c') \rightarrow (\forall x : Inst(x, c') \rightarrow \exists i : Inst(i, c) \wedge Inst(x, i))$

The subsets denoted by the instances of a disjoint specification type are disjoint

- (56) $Disjoint(c, c') \rightarrow \nexists i_1, i_2 : Inst(x, i_1) \wedge Inst(x, i_2) \wedge i_1 \neq i_2 \wedge Inst(i_1, c) \wedge Inst(i_2, c)$

Each specialisation of a specification type is also a specification type

- (57) $Spec(x, c) \wedge SbS(c, c') \rightarrow \exists x' : SbS(x, c')$
 (58) $Spec(x, c) \wedge SbS(c, c') \rightarrow SpecX(x, c)$ (Each specialisation of a specification type is a SpecX)

The partitioned type of a specification type that specialises another specification type must have consistent parents, i.e., specialise the partitioned type of the parent specification type

- (59) $SbS(c_1, c'_1) \wedge SbS(c_2, c'_2) \wedge Spec(c_1, c_2) \rightarrow Spec^+(c'_1, c'_2)$

Each specialisation of an instance of a specification type is also an instance of the specification type

- (60) $Spec^+(x, c) \wedge Inst(c, y) \wedge SbS(y, c') \rightarrow Inst(x, y)$

Axioms for Member and Categorisation

- (61) $Member(x, c) \rightarrow level(c) < level(x)$ (Member relation must be consistent with level order)
 (62) $Cat(c, t) \rightarrow level(t) = level(c)$ (Categorisation must be consistent with level order)
 (63) $Cat(c, t) \wedge Member(x, c) \rightarrow GenInst(x, t)$ (Categories classify general instances of the type)
 (64) $Spec(c, c') \wedge Cat(c', t) \rightarrow SpecR(c, c')$ (Categories may only be specialised by refinement)
 (65) $Cat(c, c') \rightarrow \nexists x : Inst(x, c)$ (Categories cannot be instantiated)

The constraint of the category may refer only to the attributes of the category and the categorised type

- (66) $Cat(c, c') \wedge \phi = desc(c) \rightarrow names(\phi) \subseteq (attr(c) \cup attr(c'))$
 (67) $Spec(x, c) \wedge Cat(c, c') \rightarrow \exists x' : Cat(x, c')$ (A specialisation of a category is also a category)

Specialisation of categories and their type must have consistent parents

- (68) $Cat(c_1, c'_1) \wedge Cat(c_2, c'_2) \wedge Spec(c_1, c_2) \rightarrow Spec^*(c'_1, c'_2)$
 (69) $Member(x, c) \wedge Spec(c, c') \rightarrow Member(x, c')$ (Members of a category are members of the parent category)

Axioms for Descriptions

Constraints can use only attributes defined in its associated object, except in the case of specifications and categories.

- (70) $\phi = desc(c) \wedge \neg(SbS(c, c') \vee Cat(c, c')) \rightarrow names(\phi) \subseteq attr(c)$
 (71) $\phi = desc(x) \rightarrow \phi_a(x)$ (An object must be valid, i.e., its ground constraints must evaluate as true.)

Validating a description means successfully evaluating all of its constraint expressions that are not missing values.

- (72) $\phi = desc(x) \rightarrow (\phi_a(x) \leftrightarrow (\forall \psi \in \phi : \psi(x) \vee \exists a : a \in names(\psi) \wedge \nexists v : v = val(x, a)))$

Constraints are inherited through the specialisation hierarchy, i.e., the specialisation's description incorporates all of the constraints of the parent's description.

- (73) $Spec(c, c') \rightarrow desc(c') \subset desc(c) \wedge (\forall \phi \in desc(c') \rightarrow From(c, \phi, c'))$

Instances must incorporate the constraints of the class that are missing a value; the attributes with values are substituted by their values.

- (74) $GenInst(x, c) \wedge \phi = desc(c) \wedge \psi \in \phi \wedge a \in names(\psi) \wedge \nexists v : v = val(c, a) \rightarrow$
 $\psi' = (\psi\{a' \mapsto val(c, a') \mid a' \in names(\psi) \wedge \exists v : v = val(c, a')\}) \wedge \psi' \in desc(x) \wedge From(c, \psi', c)$

Members incorporate the category's constraints related to the categorised type

- (75) $Member(x, c) \wedge Cat(c, t) \wedge \phi = desc(c) \wedge \psi \in \phi \wedge a \in names(\psi) \wedge a \in attr(t) \rightarrow$
 $\psi' = (\psi\{a' \mapsto val(c, a') \mid a' \in names(\psi) \wedge \exists v : v = val(c, a')\}) \wedge \psi' \in desc(x) \wedge From(x, \psi', c)$

An object must be consistent with its description, i.e., the conjunction of the expressions is satisfiable.

- (76) $\phi = desc(x) \rightarrow \bigwedge_{\psi \in \phi} \psi(x)$

Axioms for Attributes

(77) $attrs(c, a, t) \leftrightarrow a \in attr(c) \wedge t = type(c, a)$ (Attribute signature: the object c has attribute a with type t)

Axioms for meta-instantiation of attributes

The following four axioms optionally replace axioms 46 and 47.

(78) $InstN(x, c) \wedge a \in attr(x) \cap attr(c) \rightarrow type(x, a) = type(c, a)$ ($InstN$ does not change attribute type)

$InstX$ does not change attribute type if a value is assigned

(79) $InstX(x, c) \wedge a \in attr(x) \cap attr(c) \rightarrow (val(x) \neq \emptyset \rightarrow type(x, a) = type(c, a))$

$InstN$ instantiates all (unassigned) attributes of the type

(80) $InstN(x, c) \wedge a \in attr(c) \wedge t = type(x, a) \wedge val(c, a) = \emptyset$
 $\rightarrow val(x, a) \neq \emptyset \wedge (v \in val(x, a) \rightarrow GenInst(v, t) \vee Member(v, t))$

$InstX$ instantiates all (unassigned) attributes or redefines the type as an instance of the parent type.

(81) $InstX(x, c) \wedge a \in attr(c) \wedge t = type(x, a) \wedge t' = type(c, a) \wedge val(c, a) = \emptyset$
 $\rightarrow ((val(x, a) \neq \emptyset \wedge (v \in val(x, a) \rightarrow GenInst(v, t) \vee Member(v, t))) \vee GenInst(t, t'))$

Inverse Attributes

(82) $inverse : O \times L \rightarrow L$ The label of the inverse attribute of the attribute of an object (partial function).

(83) $Spec^*(c, c') \wedge From(c, a, c') \wedge a' = inverse(c', a) \rightarrow a' = inverse(c, a)$ (Inverse attribute consistency)

Inverse inverses should match with the inverse of the type

(84) $attrs(c, a, c') \wedge attrs(c', a', t) \wedge inverse(c, a) = a' \rightarrow inverse(c', a') = a \wedge Spec^*(c, t)$

Inverse inverses should match with the object of the value

(85) $a \in attr(x) \wedge a' \in attr(x') \wedge x' \in val(x, a) \wedge inverse(x, a) = a' \rightarrow inverse(x', a') = a \wedge x \in val(x', a')$

Cardinality Constraints

(86) $card : O \times L \mapsto \mathbb{N} \times \mathbb{N}$ The min. and max. cardinality governing the values for the attribute of an object. We use $\pi_n(cd)$ to denote accessing the n th element of the tuple cd .

The number of values must be consistent with the cardinality

(87) $cd = card(x, a) \wedge V = val(x, a) \rightarrow \pi_1(cd) \leq |V| \leq \pi_2(cd)$

Cardinality consistency, specialisation can restrict the cardinality

(88) $Spec(c, c') \wedge From(c, a, c') \wedge cd = card(c, a) \wedge cd' = card(c', a) \rightarrow \pi_1(cd) \geq \pi_1(cd') \wedge \pi_2(cd) \leq \pi_2(cd')$

Cardinality consistency, instantiation does not change cardinality

(89) $Inst(x, c) \wedge From(x, a, c) \wedge cd = card(c, a) \rightarrow card(x, a) = cd$

Attribute Subsets

(90) $Subsets \subseteq O \times L \times L$ $Subsets(o, a, a')$: the domain of attribute a of o is a subset of attribute a'

The values of a parent attribute must include the values of the sub-attribute

(91) $a \in attr(x) \wedge a' \in attr(x) \wedge Subsets(x, a, a') \rightarrow val(x, a) \subseteq val(x, a')$

(92) $Subsets(c, a, a') \wedge From(c', a, c) \wedge From(c', a', c) \rightarrow Subsets(c', a, a')$ (Subset attribute inheritance)

(93) $Subsets(c, a, a') \wedge From(c, a, c') \wedge From(c, a', c') \rightarrow Subsets(c', a, a')$ (Subset parent consistency)

(94) $attrs(x, a, t) \wedge Subsets(c, a, a') \wedge i = inverse(x, a) \wedge i' = inverse(x, a') \rightarrow Subsets(t, i, i')$ (Inverse consistency)

Acknowledgements

This research was funded in part by the South Australian Premier's Research and Industry Fund grant no. IRGP 37.

References

- [1] N. Young, S. Jones, SmartMarket Report: Interoperability in Construction Industry, Tech. rep., McGraw Hill (2007).
- [2] Fiatch, Advancing Interoperability for the Capital Projects Industry: A Vision Paper, Tech. rep., Fiatch (Feb. 2012).
- [3] ISO, ISO 15926 – Part 2: Data Model, 2003.
- [4] MIMOSA, Open Systems Architecture for Enterprise Application Integration, 2014.
- [5] M. Selway, M. Stumptner, W. Mayer, A. Jordan, G. Grossmann, M. Schrefl, A conceptual framework for large-scale ecosystem interoperability, in: Proc. ER'15, Vol. 9381 of LNCS, Stockholm, 2015, pp. 287–301.
- [6] B. Neumayr, M. Schrefl, B. Thalheim, Modeling techniques for multi-level abstraction, in: The Evolution of Conceptual Modeling, Vol. 6520 of LNCS, Springer Berlin Heidelberg, 2011, pp. 68–92.
- [7] S. Bergamaschi, D. Beneventano, F. Guerra, M. Orsini, Data integration, in: Handbook of Conceptual Modeling, Springer, 2011, pp. 441–476.
- [8] C. Atkinson, T. Kühne, The Essence of Multilevel Metamodeling, in: Proc. of UML 2001, Springer, 2001, pp. 19–33.
- [9] C. Atkinson, R. Gerbig, Flexible deep modeling with Melanee, in: Modellierung (Workshops), Vol. 255 of LNI, GI, 2016, pp. 117–122.

- [10] J. de Lara, E. Guerra, Deep Meta-modelling with MetaDepth, in: Proc. of TOOLS, LNCS 6141, Springer, 2010, pp. 1–20.
- [11] B. Neumayr, C. G. Schuetz, M. A. Jeusfeld, M. Schrefl, Dual deep modeling: multi-level modeling with dual potencies and its formalization in F-Logic, *Software & Systems Modeling* (2016) 1–36.
- [12] C. Atkinson, T. Kühne, Reducing accidental complexity in domain models, *Soft. & Syst. Modeling* 7 (3) (2008) 345–359.
- [13] F. P. J. Brooks, No silver bullet essence and accidents of software engineering, *Computer* 20 (4) (1987) 10–19.
- [14] W. D. Embley, B. Thalheim (Eds.), *Handbook of Conceptual Modeling*, Springer, 2011.
- [15] B. Neumayr, K. Grün, M. Schrefl, Multi-level domain modeling with M-objects and M-relationships, in: *Proceedings APCCM '09*, 2009, pp. 107–116.
- [16] C. Atkinson, T. Kühne, Model-driven development: A metamodeling foundation, *Software, IEEE* 20 (5) (2003) 36–41.
- [17] C. Atkinson, T. Kühne, Demystifying ontological classification in language engineering, in: *Proc. ECMFA 2016*, Springer, 2016, pp. 83–100.
- [18] Y. Wand, R. Weber, On the ontological expressiveness of information systems analysis and design grammars, *Information Systems Journal* 3 (3) (1993) 217–237.
- [19] C. Gonzalez-Perez, B. Henderson-Sellers, A powertype-based metamodeling framework, *Soft. & Syst. Modeling* 5 (1) (2006) 72–90.
- [20] B. Neumayr, M. A. Jeusfeld, M. Schrefl, C. Schütz, Dual deep instantiation and its ConceptBase implementation, in: *Proc. CAISE '14*, Springer, 2014, pp. 503–517.
- [21] U. Frank, Multilevel modeling, *Business & Information Systems Engineering* 6 (6) (2014) 319–337.
- [22] J. J. Odell, Power types, *JOOP* 7 (1994) 8–12.
- [23] A. Jordan, M. Selway, W. Mayer, G. Grossmann, M. Stumptner, An ontological core for conformance checking in the engineering life-cycle, in: *Proc. Formal Ontology in Information Systems (FOIS 2014)*, IOS Press, 2014.
- [24] S. Borgo, M. Franssen, P. Garbacz, Y. Kitamura, R. Mizoguchi, P. Vermaas, Technical artifact: An integrated perspective, in: *FOMI 2011*, Vol. 229 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2011, pp. 3–15.
- [25] M. Schrefl, M. Stumptner, Behavior consistent specialization of object life cycles, *ACM TOSEM* 11 (1) (2002) 92–148.
- [26] W. Klas, M. Schrefl, *Metaclasses and Their Application*, LNCS 943, Springer, 1995.
- [27] R. C. Goldstein, V. C. Storey, Materialization, *IEEE Trans. on Knowl. and Data Eng.* 6 (5) (1994) 835–842.
- [28] G. Mintchell, OpenO&M demonstrates ‘information interoperability’ for oil and gas applications, *Automation World* (2012) 24–25.
- [29] V. A. Carvalho, J. P. A. Almeida, C. M. Fonseca, G. Guizzardi, Extending the foundations of ontology-based conceptual modeling with a multi-level theory, in: *Proc. ER 2015*, 2015, pp. 119–133.
- [30] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, M. Stumptner, Configuring large-scale systems with generative constraint satisfaction, *IEEE Intelligent Systems* 13 (4).
- [31] T. Gjøsaeter, A. Prinz, J. P. Nyttun, MOF-VM: Instantiation revisited, in: *Proc. MODELSWARD'16*, 2016, pp. 137–144.
- [32] OMG, OMG meta object facility (MOF) core specification, version 2.4.1, Ref. no. formal/2013-06-01 (2013).
- [33] O. Eriksson, B. Henderson-Sellers, P. J. Ågerfalk, Ontological and linguistic metamodeling revisited: A language use approach, *Information and Software Technology* 55 (12) (2013) 2099–2124.
- [34] OMG, OMG Unified Modeling Language (OMG UML), version 2.5, Ref. no. formal/2015-03-01 (2015).
- [35] J. Bézivin, O. Gerbé, Towards a precise definition of the OMG/MDA framework, in: *Proceedings of ASE'01*, Vol. 1, San Diego, USA, 2001, pp. 273–282.
- [36] J. Bézivin, On the unification power of models, *Software & Systems Modeling* 4 (2) (2005) 171–188.
- [37] T. Kühne, Matters of (meta-) modeling, *Software & Systems Modeling* 5 (4) (2006) 369–385.
- [38] B. Henderson-Sellers, T. Clark, C. Gonzalez-Perez, On the search for a level-agnostic modelling language, in: *Proc. CAISE'13*, LNCS 7908, 2013, pp. 240–255.
- [39] T. Clark, P. Sammut, J. Willans, *Superlanguages: developing languages and applications with XMF*, Ceteva, 2008.
- [40] M. Hammer, D. Mc Leod, Database description with SDM: A semantic database model, *ACM Trans. Database Syst.* 6 (3) (1981) 351–386.
- [41] D. Costal, C. Gómez, G. Guizzardi, Formal semantics and ontological analysis for understanding subsetting, specialization and redefinition of associations in UML, in: *Proc. ER 2011*, Springer, 2011, pp. 189–203.
- [42] C. Atkinson, R. Gerbig, T. Kühne, A unifying approach to connections for multi-level modeling, in: *Proc. MODELS 2015*, ACM/IEEE, 2015, pp. 216–225.
- [43] N. Guarino, G. Guizzardi, We need to discuss the relationship, in: *Proc. CAiSE 2015*, Springer, 2015, pp. 279–294.
- [44] T. Halpin, *Object-role modeling*, in: *Encyclopedia of Database Systems*, Springer, Boston, MA, 2009, pp. 1941–1946.
- [45] C. A. Welty, N. Guarino, Supporting ontological analysis of taxonomic relationships, *Data Knowl. Eng.* 39 (1) (2001) 51–74.
- [46] J. de Lara, E. Guerra, J. S. Cuadrado, When and how to use multilevel modelling, *ACM Trans. Softw. Eng. Methodol.* 24 (2) (2014) 12:1–12:46.
- [47] B. Smith, Against idiosyncrasy in ontology development, in: *Proc. FOIS 2006*, 2006, pp. 15–26.
- [48] J. de Lara, E. Guerra, R. Cobos, J. Moreno-Llorena, Extending deep meta-modelling for practical model-driven engineering, *Computer* 57 (1) (2014) 36–58.
- [49] B. Neumayr, M. Schrefl, Abstract vs concrete clabjects in dual deep instantiation, in: *Proc. MULTI'14 Workshop*, 2014, pp. 3–12.
- [50] T. Kühne, Contrasting classification with generalisation, in: *Proceedings APCCM '09*, Australia, 2009, pp. 71–78.