Field Types for Deep Characterization in Multi-Level Modeling

Thomas Kühne*, João Paulo A. Almeida†, Colin Atkinson‡, Manfred A. Jeusfeld§, Gergely Mezei¶
* *Victoria University of Wellington*, Wellington, New Zealand
† *Federal University of Espírito Santo (UFES)*, Vitória, Brazil
‡ *University of Mannheim*, Mannheim, Germany
§ *University of Skövde*, Skövde, Sweden
¶ *Budapest University of Technology and Economics*, Budapest, Hungary

***Abstract*—Traditional two-level modeling approaches distinguish between class- and object features. Using UML parlance, classes have attributes which require their instances to have object slots. Multi-Level Modeling unifies classes and objects to "clabjects", and it has been suggested that attributes and slots can and should be unified to "fields" in a similar way. The notion of deep instantiation for clabjects creates the possibility of "deep fields", i.e., fields that expand on the roles of pure attributes or pure slots. In this paper, we discuss several variants of such a "deep field" notion, pointing out the semantic differences and the various resulting trade-offs. We hope our observations will help clarify the range of options for supporting clabject fields in multi-level modeling and thus aid future MLM development.**

***Index Terms*—multi-level modeling, attribute definition**

I. Introduction

Conventional two-level object-oriented approaches are based on a class-object dichotomy: classes capture invariant aspects of the objects that instantiate them. An integral part of instantiation is that class *attributes* are expressed as object *slots*[1]. For example, if a class Product defines the attribute price : Double, an instance prod1 : Product will then be required to have a slot such as price = 9.95, where the slot value 9.95 must conform to the attribute type Double. The attribute type therefore constrains the set of admissible slot values and often is a datatype, i.e., has instances that, unlike objects, are immutable and lack identity. Datatypes often have an associated set of operations, e.g., arithmetic operations for numeric values.

Multi-level modeling (MLM) unifies the notions of "class" and "object" into a single *clabject* concept [2], [3]. Clabjects therefore may have both an instance (object) role and a type (class) role. This suggests a similar unification of attributes and slots into so-called "fields" [3], since a clabject's fields can in general be expected to both

1) be required to conform to the fields of its classifying clabject, and
2) control the fields of its instances.

As a result, deep (more than two-level) clabject classification hierarchies will give rise to deep "field chains", i.e., sequences of corresponding fields within a clabject instantiation branch. Note that references like "top-level"- or "bottom-level"-field do not necessarily align with the top- and bottom-level of the entire MLM hierarchy, since a field chain may not span the entire multi-level hierarchy.

Naturally, fields, other than the top- or bottom-level fields, may have roles beyond that of a pure attribute or pure slot. Moreover, the top-level field definition may be used to exert control not only over the field directly below it, but instead over more fields, potentially including all fields in the field chain (cf. "deep characterisation" [4]). As with many MLM notions, there is a plurality of different approaches to realize such deep field control over multiple levels. Existing approaches range from the use of traditional shallow fields, in combination with the powertype pattern [5], to elaborate "deep field" mechanisms with widely different semantics.

The goal of this paper is to characterize and document the essential nature of major canonical "deep field" design choices which are embodied in existing approaches, with a view to catalog them and identify their trade-offs. In order to define a realistic scope, we specifically refrain from comparing concrete field designs of existing MLM technologies, but rather focus on the evaluation of the essence of their approaches to using fields to support deep characterization. We hope our observations will help clarify the range of options for supporting clabject fields in MLM and thus aid in fostering consensus in the MLM community regarding relevant concepts, terminology, and the evaluation of design alternatives.

In this paper, we first provide a brief background to how current ways of supporting field evolved (Section II) and then present a deep modeling scenario that we use as a motivation and benchmark for our analysis (Section III). We subsequently present five canonical field variants including appraisals of their respective trade-offs (Section IV). Finally, we discuss the observations made so far – attempting to gain an overview of the design landscape and critiquing our analysis with respect

Gergely Mezei is supported by the Ministry of Culture and Innovation and the National Research, Development and Innovation Office (Grant No. 2022-2.1.1-NL-2022-00012). João Paulo A. Almeida is supported by CNPq (Grant No. 313687/2020–0) and FAPES (Grants No. 281/2021, 1022/2022).

[1]The term "slot", in this context, traces back to Minsky's seminal work on frame-based knowledge representation [1].

to its potential limitations – before concluding with closing remarks (Section VI).

II. A Brief History of Fields

The notion of fields (including their two-level variants) has been relevant in numerous computer science disciplines, specifically in databases, knowledge representation, ontologies, conceptual modeling and object-oriented programming.

In relational databases, a domain type attribute can be represented as a table column. Each such column has a datatype, and values for the column in a given database row must conform to this datatype. The SQL standard for relational databases provides a fixed set of datatypes such as integer, float, date/time, and string. The implied relationship between the datatype of a column and all values in that column can be characterized as a "schema-data" relationship which is compatible with the concept of the values being classified by the datatype.

In the Semantic Web's RDF formalism, triples of the form "subject-predicate-object" are used to represent information about entities of interest. Such information includes the association of values to subject properties. The object position of the respective triple may contain URIs (which denote an entity) or RDF literals, which represent values. Hence, RDF "slots" are not limited to containing values of datatypes. The UML is another example reiterating this generality of "slots", as it also supports classes, next to datatypes, to be used as attribute types. Indeed, while we are only using datatypes as field types in our domain example, we do so without loss of generality.

Note that allowing slots to contain more than just (order-zero) values of (order-one) datatypes, opens up the possibility of letting them contain type-valued (order-one) content, or content of any order, for that matter. The term "value" hence becomes overloaded because it is often used to refer to both an identity-lacking constant, and the content of a slot (order-zero field). We will make use of order-one content for slots, and will occasionally refer to slot contents as "values" without the intent to imply that only identity-lacking constants are referred to.

Another aspect exemplified by RDF and the UML's liberal treatment of "slot" contents is that the boundary to "links" becomes blurred. A slot containing a reference to an object/entity can be regarded as the equivalent of a unidirectional UML "link". Again, our domain example has been chosen to support a straightforward discussion of deep characterization approaches, but is not meant to imply that the observations made are only valid regarding traditional slots that contain datatype values. Note that even operations/methods could be the subject of deep characterization.

In the object-oriented database GemStone, an object was regarded as a "chunk of private memory" [6]. Fields of objects were therefore part of that memory chunk. GemStone inherited this view from SmallTalk [7], and this view influenced a number of current object-oriented programming languages, such as Java. This view emphasises the understanding of "fields" as a representation concept. In other words, there is no claim that fields or slots conceptually occur in a domain, rather they are part of a modeling vocabulary to represent domain properties, analogous to how clabjects represent concepts of domain entities.

III. Domain Scenario

In order to simultaneously motivate our discussion of the design space of alternative field approaches for deep characterisation (see Section IV) and define a benchmark for evaluation, we use the example of a software system to support the management of cars from various manufacturers. In this section, we describe salient concepts occurring in the domain and establish several domain requirements that respective models need to adhere to and/or must enforce. The subject domain not only includes individual cars (e.g., myC5III and myCLK200), but also their models (e.g., the Citroën-designed C5III or the Mercedes-Benz-designed CLK200). Cars have properties such as a vehicle identification number (→ vin) and a paint color (→ color) (cf. Figure 1). Note the absence of VIN values at the middle level (e.g., at C5III); respective values at this level do not make sense ontologically, and therefore should not exist. Regardless of which car models may be added to the car management system in the future, these properties must always be available for a particular car. Hence, it is a domain requirement that the respective properties must be stipulated at the level of CarModel, i.e., a model element CarModel needs to *deeply characterize* the instances of its instances so that they feature color and vin fields.

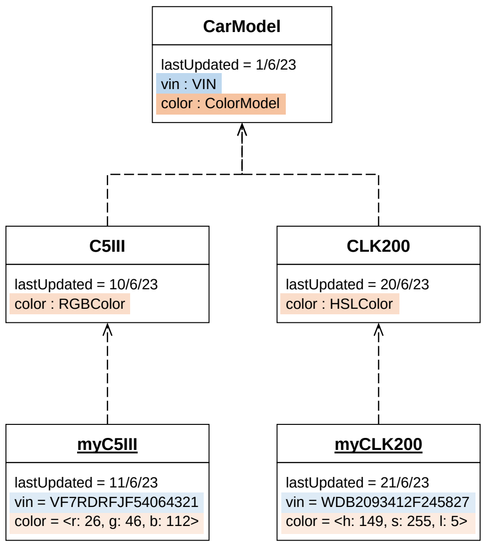

Fig. 1. Car Sales System

The domain requirements furthermore prescribe that all system model elements, i.e., CarModel, its instances, as well as all instances representing individual cars, must have a lastUpdated property since they are used in the car management software system and need to be appropriately version-managed.

Several property types that govern the allowed values for concept properties naturally arise in our sample domain:

VIN, defining the form of "Vehicle Identification Number" values, RGBColor and HSLColor, defining the form of RGB and HSL color representation values, respectively, and ColorModel, classifying these color models (see Table I). Datatypes, like RGBColor typically support operations such as brighten() and impose well-formedness rules on their instances, e.g. that the value range for the red, green, and blue components is [0. .255].

Note that ColorModel can be recognized as a second-order type since its instances RGBColor and HSLColor are (first-order) datatypes that have their own (zero-order) datatype values. In other words, one can observe that property types and their corresponding type definitions in models can form a classification hierarchy, analogous to concept hierarchies (cf. Figure 2).

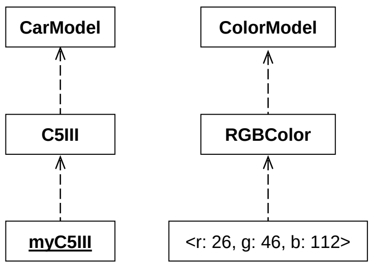

Fig. 2. Concept and Property Hierarchies

TABLE I
PROPERTY DOMAIN TYPES

Property Type	Description
VIN	characterizes vehicle identification numbers
RGBColor	characterizes colors represented in an RGB color model
HSLColor	characterizes colors represented in an HSL color model
ColorModel	characterizes color models (e.g, RGBColor)

A. Deep Property Kinds

In order to examine the strengths and weaknesses of the deep characterization variants, we have chosen each property in our domain scenario to represent a particular kind of deep property. We identified three main kinds of properties that occur in the context of deep domain properties:

1) Single type, Single value (STSV): A property like "vin" is commonly found in domains; it only manifests with an associated value once at the bottom level of its associated instantiation branch. There are no respective values above that relative bottom level as they would lack an ontological justification. All instances at this bottom level have values of exactly the same type (here datatype VIN). Hence, in case one wants to guarantee a "vin" property for each car, regardless of future car model additions, this shared type must be fixed at the top level of the hierarchy (here at CarModel). See Figure 1 for how the top-level "vin : VIN" declaration is meant to ensure that all car identification values at the bottom-level must conform to the VIN datatype.

2) Single type, Multiple values (STMV): A variation of the above property kind is a property kind that still has only one type associated with it, but for which it makes sense to associate a value to each element in an instantiation chain, as opposed to just the bottom concept. In our domain scenario, for example, each element in the car management software system has, and must have, a value for the lastUpdated property (cf. Figure 1).

3) Multiple types, Single value (MTSV): Unlike the VIN values, the color values at the bottom level in Figure 1 are not of exactly the same type. Here, we assume that cars made by Citroën need to have their colors specified using an RGB color model. Hence the Citroën-designed car model C5III specifies a color field "color : RGBColor". In contrast, we assume that Mercedes-Benz uses the HSL color model to specify car colors, which is why the Mercedes-Benz-designed car model CLK200, specifies a "color : HSLColor" field. Note that these types, i.e., the types characterizing the bottom-level values cannot be chosen arbitrarily. Both must be characterized by a common type ColorModel. In other words, the top-level ColorModel type ensures that while car manufacturers are given freedom over their color model choice, the values at the bottom-level are all instances of *some* color model, i.e., exhibit conceptual affinity. The top-level ColorModel type could, for example, deeply specify the existence of a luminance() operation which is supposed to return the brightness level of a color. An HSL color instance could just return its l (luminance) value, whereas an RGB color instance would have to calculate the perceived brightness from its red, green, and blue components.

Note that at each intermediate level, an MTSV-property plays a dual role: it is both a "value" that conforms to the property at the level above, and a "type" that characterizes the property "values" below it. See Figure 1, where the Color properties at the middle level are instances of ColorModel but also characterize fields below them. Although one could conceptualize the respective presence of many (non-order-zero) "values" – as many as the respective concept hierarchy is deep – as "multi-value", we characterize this property kind as *SV ("single value") since there is only a single order-zero value, at the very bottom of an instantiation chain. All other "values" have at least order one and can be regarded as characterizing the bottom-level values in a hierarchical manner.

One way to appreciate the difference between STSV- and MTSV-properties is to observe that the former allows values of one single type only, whereas the latter supports the use of subtypes of a single common type. This becomes evident by observing that RGBColor and HSLColor can be generalized to Color. This common supertype Color would have to be used to accommodate all desired values with a "single type" at the top level. The relationship between such a common supertype of the flexible type choices at the middle level and the top-level type is the so-called "powertype" relationship [5]. Every instance of the top-level type (here ColorModel) must be a subtype of the common supertype (here Color).

4) Multiple types, Multiple values (MTMV): Our categorization scheme for property kinds suggests that a fourth MTMV-property kind may exist. Properties of this kind would have to be compound in nature as they would have to, at each non-bottom-level, simultaneously have an order-zero value, and a higher-order type. We are unaware of a respective domain example and most modeling technologies do not accommodate this property kind (DMLA being a notable exception [8]). We therefore do not cover this property kind in our comparisons.

IV. Field Variants

In this section we apply five different deep characterization approaches to the domain scenario, to analyze how they handle the three different property kinds of the domain scenario.

All five approaches are inspired by existing field semantics designs but we deliberately present and discuss the essential deep characterization aspects of those designs only, to avoid distractions and biases that could potentially be caused by other design decisions coupled to the respective existing field designs. For all five distilled design choices we adhered to the convention of linking fields within a field chain to each other via field name equality. For instance, a price = 9.95 field is known to be controlled by a price : Double field due to the name equality between those fields.

Note that the first four approaches use a hierarchy of classification levels as their foundation, whereas the last approach is based on refinement.

A. Variant 1: Shallow Fields

"Shallow fields" (e.g., the combination of UML attributes and UML slots), are characterized by the fact that their control depth does not exceed that of an attribute from a two-level approach, i.e., they only control fields one level below them. In contrast to two-level approaches, however, shallow fields can be defined for clabjects at any level, i.e., may give rise to class-level slots, for example, if defined at the level above the class level. Initially, it may appear to be futile to attempt the deep characterization we referred to earlier with fields that are per definition not "deep". It seems necessary, in order to control even only the second level further down, to either target it directly, or somehow control the type facet of elements at the adjacent level lower down. Fortunately, the latter effect can be achieved by using the well-known concept of a "powertype", specifically in this case the "Odell variant" [5], [9], [10]. Figure 3 shows a solution for our domain using shallow fields only.

Odell powertypes were originally not conceived to achieve deep characterization, but rather to clarify and explain class-level slots, partitioning generalization sets, and the latter's relationship to metatypes [9]. However, since powertypes link a supertype to a metatype, the latter can use the former to influence the class facet of its instances. The Car class in Figure 3 can impart its vin and color fields on the instances of CarModel since the powertype-relationship between Car and

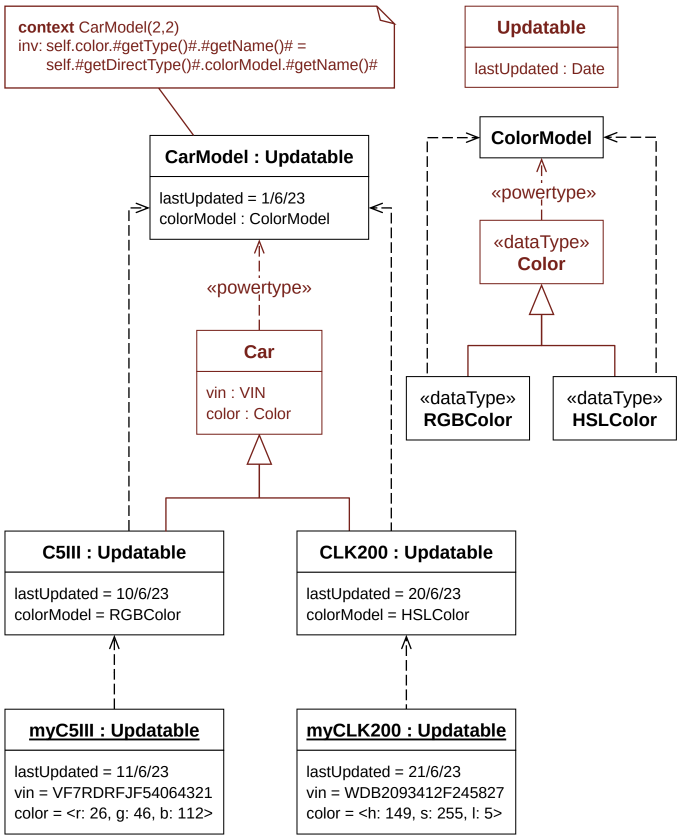

Fig. 3. Shallow Fields Model

CarModel ensures that every CarModel instance must also be a subtype of Car. Hence, all instances of CarModel inherit the vin and color fields, thus effectively achieving deep control over the order-zero vin and color fields of particular cars at the bottom level. In some approaches, supertype Car is referred to as the "most general instance" [11]. The above approach works very well for field vin, since all cars are guaranteed to have vin fields of type VIN. Note that an order-one vin field is introduced at the middle level, which is technically not required by the example domain. However, since no values can be assigned to such a field in a "shallow" fields scheme, no harm is caused.

Arguably, the necessity to explicitly introduce a supertype (Car for all car models) could be regarded as adding accidental complexity. In many cases, such a supertype is desirable anyway, but when its only purpose is to serve as a type facet template for car model types then it could be considered to be a workaround artefact (compare Figure 3 with Figure 4 to see the respective difference). Note that the powertype approach shown here is capable of supporting deep fields with an unbounded depth control by chaining respective powertype configuration in a hierarchical sequence, potentially compounding the aforementioned concern regarding accidental complexity.

Regarding field color, shallow fields are not as good a match, as they are for field vin. Field color requires a more precise specification of object properties – one that allows subtypes (here RGBColor and HSLColor) to constrain the color model used by bottom-level color fields to the manufacturer-stipulated one. Since the color field is defined once in Car with a fixed type, all car model types have to share the same fixed type, i.e., lack the required precision regarding the choice of color model (cf. Figure 1). In Figure 3 we illustrate one of many possible workarounds. Here, we introduce the supertype Color so that both value types can be supported at the bottom-level color fields. Note the second use of a powertype-relationship; in this case it is necessary to ensure that instances of ColorModel are also subtypes of Color, so that their respective instances can be assigned the color field of type Color in Car.

Note that a deep constraint (attached to CarModel in Figure 3), expressed using a deep constraint language [12] and targeting the bottom level (see the "(2,2)" notation), is required to link the color subtype choice made at the bottom level (to represent a color in a certain color model) to the color model choice made at the middle level, since all instances from the same branch must share the same color model respectively. A functionally equivalent constraint using OCL would be longer and would exhibit bad maintainability due to the need to alter it each time a new car model is added to the system. Although the aforementioned constraint ensures the integrity between car models and their instances, the color fields at the bottom of the hierarchy are still typed with the unspecific Color type. Therefore, a comparison in a query, or an assignment in a transformation, along the lines of myC5III.color == myCLK200.color would pass a static typecheck (since both expression types are of the general type Color) which is not ideal. The constraint would flag a respective inconsistency as soon as its evaluation is triggered somehow at some point in time, but this corresponds to runtime error checking, i.e., is not equivalent to static typechecking. The constraint could be avoided altogether by employing covariant redefinitions of the color field in subclasses RGBColor or HSLColor. This solution would even support static type checking of color value assignments. However, it would require language support for covariant redefinitions and entail all the associated benefits and downsides of respective approaches.

It should be noted that the workaround of using a Color supertype as shown in Figure 3 only works under the assumption that the provided domain types RGBColor and HSLColor can be placed into respective subtype roles. In case this is not possible (e.g., due to lack of control over the domain datatypes) then a union-type variant that uses a custom Color type definition like a variant record by explicitly recording the chosen color model and delegating operations to internally stored color values, would have to be employed.

The solution chosen for STMV-properties (→ field lastUpdated) in the model in Figure 3 uses multiple classification as supported by DeepTelos [11] or *Orthogonal Ontological Classification* [13]. Note the definition of a secondary-classification type Updatable and that every element is declared to be an instance of this type using the textual ":" notation. Since there are not only ":" classifications but also "instance-of" arrows, this means that each element has two ontological types. Hence, this solution would not be available in every modeling language. Furthermore, to ensure the requirement that all elements in a system must have a lastUpdated field, some mechanism (e.g., a constraint, or allowing system elements to be constructed as instances of Updatable only) is required to ensure that each element is indeed specified as an instance of Updatable.

Without support for multiple classification, workarounds may be required. For example, one could introduce various lastUpdated declarations, distributing them over all relevant elements. This redundancy would be detrimental in that it would make it impossible to change the field's type at one place only and/or to access all lastUpdated values in a uniform manner.

An advantage of Variant 1 is the close match to the traditional use of shallow fields in UML. A disadvantage is the introduction of additional specialization relations to the powertype instance, which adds to the complexity of the model and may negatively impact the substitutability of operation arguments.

B. Variant 2: Single Fields

Single fields, introduced in [3], extrapolate the semantics of shallow fields by repeating the field type of the first-order field (which is comparable to a UML attribute) up to the top level. They align very well with STSV-properties, since they support one value at the bottom-level (order-zero field) and fix the type of the latter at the very top (and at every level in between). In Figure 4, the field vin is straightforwardly represented with a "single field", thus enabling the top-level deep characterization of the value type at the bottom, and prohibiting any intermediate values in elements at the middle level. Single fields imply an order-one vin field at the middle level analogous to the shallow fields case. This time, the intermediate field is the result of an intentionally simple clabject instantiation semantics in which positive field potencies are reduced by one upon each instantiation [3]. An alternative scheme could feature single fields at the very top and the very bottom of a field chain only. However, this would compromise "*characterization locality*", i.e., the ability of a modeler to look up the type of a field by moving up by at most one level. Without the intermediate field occurrences, a modeller would have to scan the hierarchy upwards until the top-level field declaration is found.

If "object primacy" is assumed, i.e., if all higher levels are primarily thought of as scaffolding to define desired object scenarios, then single fields seem to be ideally suited for that due to their ability to shape objects directly from any level.

However, single fields suffer the same limitations as shallow fields plus powertypes when it comes to representing STMV- or MTSV-properties. As a result, single fields yield a very similar outcome to shallow fields, i.e., entail very similar workarounds (compare Figures 4 and 3), they do however, not require the introduction of a Car supertype. In particular, deeper control (exceeding the two levels covered here) is much more concisely obtained using single fields.

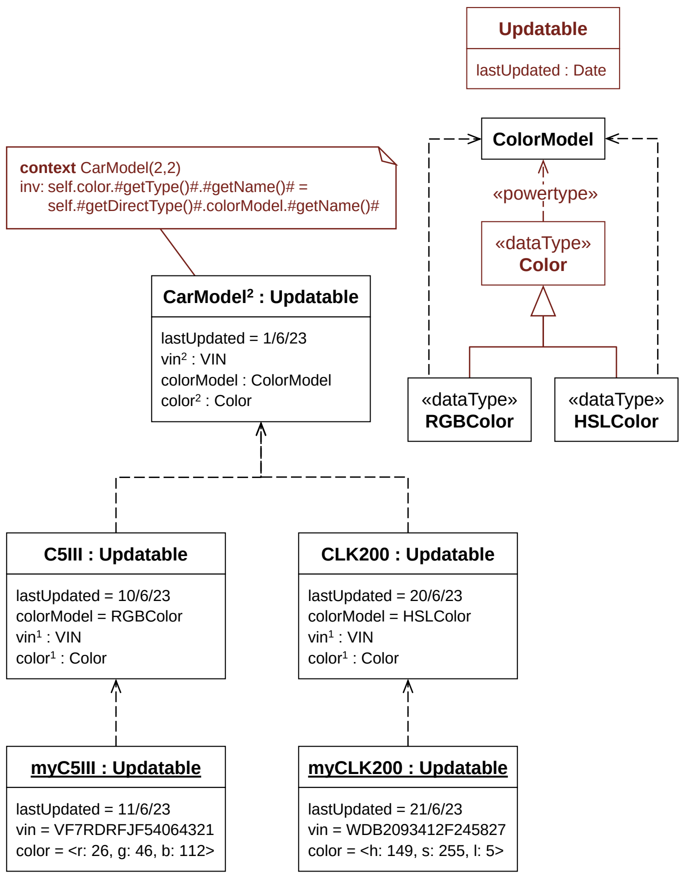

Fig. 4. Single Fields Model

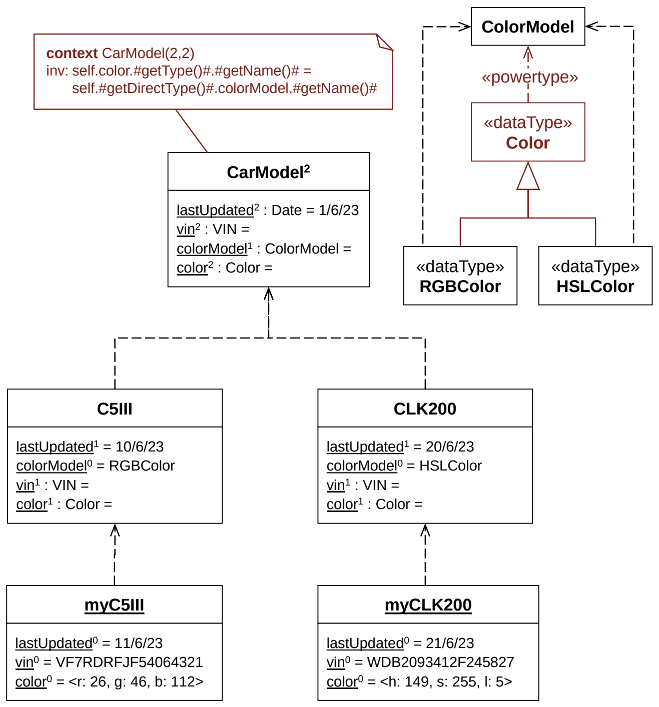

Fig. 5. Dual Fields Model

C. Variant 3: Dual Fields

Dual fields, introduced in [3], are like single fields, except that they also feature field values at every level, as opposed to the bottom-level only.

There are several design options available as to where the type and values of dual fields need to be present and/or shown (see for instance the Melanee tool which aims to leverage the dual field approach in a level-agnostic field design [14]), each of them having their own set of trade-offs. In the interest of supporting a comparison that focuses on the differences regarding deep characterization aspects, we chose to present dual fields as being identical to single fields, except that dual fields always have a value. The latter makes dual fields very well suited to represent STMV-properties, since they determine a single type at the top of the field chain and allow multiple values (of that type) across the field chain. As a result, the STMV-property lastUpdated can be represented as a dual field without the use of multiple classification and an additional Updatable type (see Figure 5 in which dual fields are underlined).

The STSV-property vin is handled less ideally by dual fields, since even though the bottom-level values are supported and correctly typed, the dual field vin technically allows a value to be assigned at both the top- and the middle levels (hence the use of "=" that are not followed by a value in Figure 5). This is incongruent with the "single value" quality of the vin property. While sometimes additional values can be put to good use, e.g., by specifying default values for lower levels, this is not always the case. In short, dual fields essentially turn STSV-properties into STMV-properties, whether that is adequate or not. A potential remedy could be a constraint that restricts intermediate fields to have an undefined value, provided such a field state is supported.

Regarding the MTSV-property Color, dual fields suffer from the same limitation as single fields, i.e., the inability to let the type vary down the field chain. Therefore, the same workaround of postulating a Color supertype and establishing a constraint as employed for single fields, with all the associated downsides, becomes necessary.

D. Variant 4: Hierarchical Fields

Hierarchical fields are potency-based, like single- and dual-fields, but extrapolate the semantics of shallow fields by consistently repeating the same relationship between field contents. As mentioned before, in shallow field semantics the value of a slot (potency-zero field) can be regarded as being an instance of the type of the corresponding higher-level attribute (potency-one field). Unlike single- and dual-fields, hierarchical fields maintain this relationship between higher-level field correspondences. For example, in Figure 6, the field content RGBColor of field color in C5III can be regarded as being an instance of the field content of the field color in CarModel, i.e., as an instance of ColorModel (cf. Figure 2). Hence, hierarchical fields can be seen as having a uniform semantics across an instantiation hierarchy, in contrast to single fields which exhibit a discontinuity at the bottom two levels where the previous repetition of the field type is replaced by a value-type pair, with the value being an instance of the type.

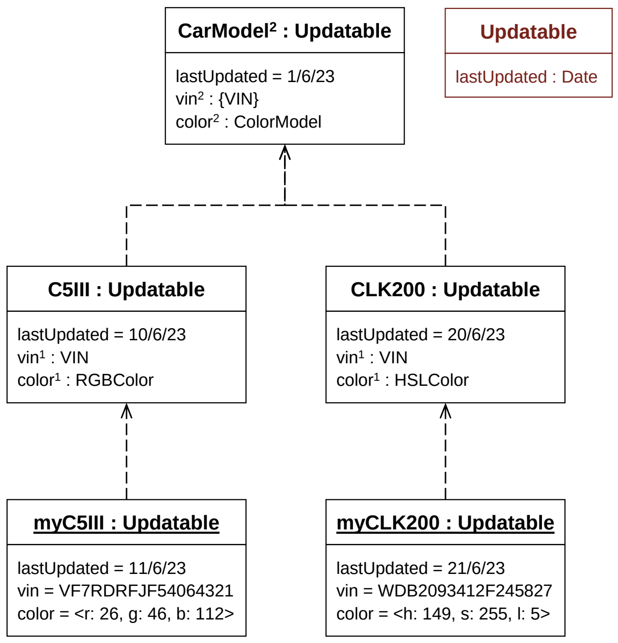

Fig. 6. Hierarchical Fields Model

As Figure 6 illustrates, hierarchical fields are particularly well suited to modeling an MTSV-property like color. The domain relationship between a color, its color model, and the latter's type ColorModel is naturally expressed, achieving a "direct mapping"-quality that mimics the direct mapping between the domain instances, types, and metatypes and their corresponding clabjects in the models shown in Figures 1–2.

Hierarchical fields do not offer the option of using multiple values since the "value" of a hierarchical field at any level (but the bottom-most) is the type that governs the well-formedness of the field content one level further down. As a result, the solution shown here to accommodate the lastUpdated property, uses the same approach as the single fields solution, i.e., an additional Updatable datatype and multiple classification.

At first glance, hierarchical fields appear to be ill-suited to accommodate single type, single value properties like vin for two reasons: First, given that the middle-level type for field vin must be VIN – so that instances at the bottom have vin fields whose values conform to datatype VIN – it appears to be necessary for modellers to use a type abstraction for VIN at the level of CarModel, e.g., declare a field vin^2 : IdentificationType. It could be regarded as burdensome to have to come up with, and define, a metatype like IdentificationType, if the only purpose of the deep declaration for the vin field is to ensure VIN-valued properties at the bottom-most level.

Second, there appears to be a loss of control over the types of the vin fields at the bottom level. At the middle level, any choice for the vin field type is valid, as long as it is an instance of IdentificationType (assuming the aforementioned declaration of vin^2 was made). This appears to allow undesirable choices and, in particular, make it impossible to ensure that all elements at the middle level use the same VIN field type.

Fortunately, a simple solution addresses both of the above potential issues. Instead of using a general metatype like IdentificationType for the vin^2 field type, one can use a Singleton type, either in the form of a very restrictive type like VINType, or an anonymous type like {VIN}, i.e., a type whose only instance is the one that is desired at the middle level. Using a Singleton type for the top-level field declaration removes any choice for a modeller at the middle level and hence creates a scenario that is equivalent to a single field scenario. Even though the top-level type is not VIN, as it would be in a single field scenario, the allowed options at the middle level are exactly the same. N.B., if specifying the VIN type had been necessary at one level higher up, the respective type would have had to be {{VIN}}.

It should be noted that this way of using hierarchical fields considerably benefits from an ability to create anonymous types by enumerating their instances with a respective syntax. In addition to the above described uses like {VIN} (cf. Figure 6), multi-instance definitions like {RGBColor, HSLColor}, could replace types like ColorModel, if, in this case, the only desired choices for color models are the aforementioned ones. Such a facility would make it possible to forgo the explicit definition of metatypes that are not needed for any other purpose. In case the literal type definition syntax used above is deemed to be insufficiently intuitive and/or concise, one could introduce a convenience notation to create anonymous Singleton types, e.g., using "!VIN", which would be equivalent to "{VIN}".

Note that not using a Singleton type for the top-level field could still suggest that there is a loss of control over the precise types used at the middle levels, due to modellers being given a choice of the field types, while no such choice is available for single- and dual-fields. However, actually the reverse is true. Either by using a narrow metatype such as VINType – whose associated supertype would be VIN, in the sense of VIN's powertype being VINType — or by only listing sufficiently precise types (e.g., subtypes of VIN) in an extensional metatype definition, one can ensure that the field types will be at least as specific as VIN.

E. Variant 5: Level-Blind Fields

Level-blind fields are distinguished from the previously presented field variations in several ways: They –

1) are used in the context of a refinement hierarchy, rather than in a hierarchy of classification levels.
2) can be regarded as combining two notions: dual- and hierarchical fields.
3) have no associated field potency that specifies the depth of their control over other fields.
4) support separate control over field- and value presence.

Since they are based on refinement (as used in [8], for example), the elements containing level-blind fields are not organized in levels and afford much more flexibility. For example, it is possible to add or remove elements in refinement chains without invalidating existing elements (e.g., by affecting their order and thus fundamentally changing their nature). The terminating elements of refinement chains are often "objects", i.e., elements that can no longer be refined. The latter are distinguished by "{Object}" annotations (see Figure 7), and can be specifically targeted via "{concrete}" annotations, i.e., making sure that "object" fields receive concrete (order-zero) values.

Level-blind fields follow strict hierarchical instantiation rules, i.e., "neighbouring" fields in a field chain must be in an "instance-of" relationship, or, alternatively may differ in the value they hold. Field neighbours may be separated by several refinement steps between their enclosing elements, though. In the sense that they, if the field type is changed, must change the field type to an instance of the preceding field type, they resemble hierarchical fields (see Section IV-D).

They resemble dual fields (see Section IV-C) in the sense that they can simultaneously hold a field type and a field value, albeit with the latter being optional. Like the dual fields presented in Section IV-C, they exist across entire field chains, in this case from the top till the very end of refinement chains. That is why the declaration of lastUpdated at CarModel in Figure 7 is sufficient to imply the presence of this field everywhere "below", and including, CarModel. Note the constraint attached to the lastUpdated field which ensures that the field has a date value everywhere, thus overriding the optional nature of level-blind field values.

Unlike dual fields, level-blind fields support the explicit "deactivation" of their value component; see the top right constraint in Figure 7, attached to the vin field, which ensures that no VIN values may appear anywhere, apart from elements at the end of refinement chains (marked with {Object} annotations). The {concrete} annotation in front of the vin field guarantees the presence of a value at the "bottom-level" objects.

The same annotation is applied to field color which is likewise forced to receive an order-zero value in the objects on the "bottom-level". Since the top-level property type ColorModel of field color (in CarModel) is of order two, given the two refinement steps used in the presented solution, its type must be instantiated in the car model elements and then, once again, when transitioning from the car models to the "bottom-level" cars.

While the two refinement steps solution forces the instantiation of the color field type ColorModel to one of the color model choices at the car models C5III and CLK200, a model featuring more refinement steps could not easily establish such a guarantee.

Summarizing, level-blind fields can precisely associate order-zero field values with "bottom-level" objects (through the use of the {concrete} and {Object} annotations). However they lack the depth control that potency-based field declarations afford, i.e., they cannot precisely specify at which level a particular type change (instantiation) or the transition to a value should occur.

V. Discussion

After having observed the trade-offs entailed by the individual approaches (versions 1–5) in isolation, we now take a bird's eye view to gain a broader perspective on the approaches with the aim of characterising them and comparing them to each other.

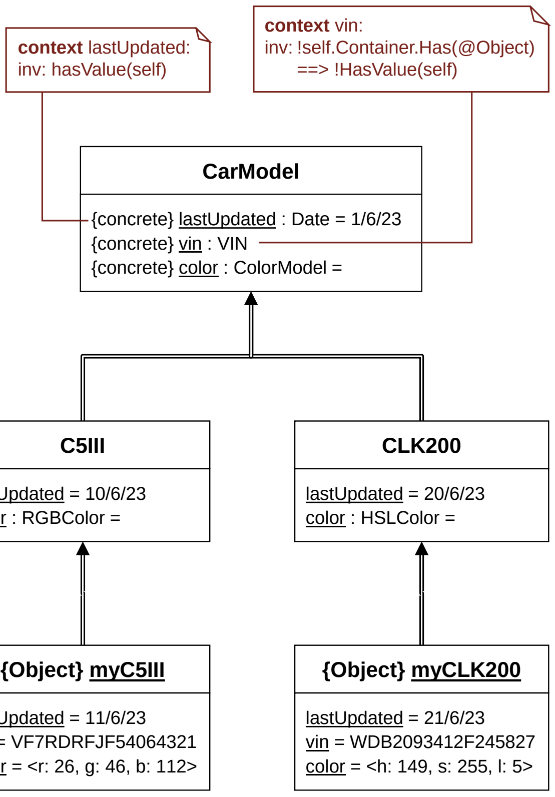

Fig. 7. Level-Blind Fields Model

A. Analysis

"Shallow fields" demonstrate that deep characterization is achievable even with just the traditional notions of "attributes" and "slots", when making use of an additional mechanism like "powertypes". As our trade-off observations revealed, they require solutions that need additional mechanisms and have the hallmarks of workarounds.

In this context, we consider a workaround to mean a solution structure expressed in a language lacking native support for a modeling feature, aimed at mimicking the effects of natural modeling choices when using a language with such native support. Respective solution structures often use a combination of mechanisms, such as extra concepts, additional fields, and constraints in a concerted manner that entails unnatural modeling choices, adds complexity, and does not scale well. Table II illustrates that while "shallow fields" are able to fulfill all domain requirements, they do so at considerable cost. The main reason is that any control over more than the next level, i.e., "control depth", must be pieced together by using two-level building blocks. When used to address an MTSV-property (here in the form of the color field), the "shallow fields" solution is analogous to the "cascading" approach [15] used by two-level technologies to create multi-level hierarchies. Note how the (instance-faceted) colorModel

TABLE II
FIELD DESIGN COMPARISON

	Property Kind		
	ST SV	ST MV	MT SV
shallow fields	✅🩹	✅🩹	✅🔧
single fields	✅	✅🩹	✅🔧
dual fields	✓	✅	✅🔧
hierarchical fields	✅🩹	✅🩹	✅
level-blind fields	✅📋	✅📋	✓

field in the middle-level elements in Figure 3 must be linked to the (type-faceted) color field at the same level via a constraint that insures that the instance-facet value in colorModel gains type control over the color values at the level below. Clearly, though feasible, using two-level technology to create multi-level hierarchies, requires too much additional scaffolding to be entirely convincing. This is why we evaluated the "shallow fields" solutions for addressing the STMV- and MTSV-properties as "workarounds" in Table II. In comparison, STSV-properties can be addressed relatively straightforwardly, but only by relying on an additional mechanism like "powertypes".

"Single fields" and "dual fields" can be considered to be an improvement over "shallow fields" since they support deep characterization of fields in a more direct and concise manner, without requiring an additional mechanism. Both approaches share the property that the order range of the field contents (types and values) does not exceed one. In other words, they do not go beyond employing a single type that controls one or more values. One can regard them as "shallow fields" whose type- and/or value- components are "stretched" over more than two levels. In both field approaches, the type component is "stretched" from the one but last level of the field chain to the top of the field chain. In the dual field approach, the same "stretching up to the top" is applied to the value at the bottom, allowing for different values (of the same type) to be used.

While this serves their "specialist applications" very well (single fields for STSV and dual fields for STMV), both approaches unsurprisingly require a workaround to account for more than one property type in a field chain, i.e., when addressing the MTSV case (cf. Table II).

Postulating the existence of values across the whole field chain, as implied by the version of "dual fields" we evaluated, means that every STSV-property is treated like an STMV-property. The respective potential occurrence of values that have no ontological justification (here, vin values at the middle level of the diagram in Figure 5) lead to dual fields only being able to partially address STSV-properties (cf. Table II).

In contrast to the previously discussed approaches, hierarchical fields are "specialists" for the MTSV case since their field type mechanics match the hierarchical nature of respective domain properties (cf. Figure 2). The use of a "Singleton type" notion, harnesses their hierarchical nature to match the requirements of STSV-properties. We categorized this "trick" as an "additional mechanism" (see Tables II & III).

In contrast to a "workaround", the use of an "additional mechanism" does not involve unnatural modeling choices and does not add complexity proportional to the depth of a hierarchy.

Like all approaches that do not directly support dual fields, hierarchical fields require an additional mechanism to obtain multiple values from an ultimately single-value approach. The solution used here is to view an ostensibly deep hierarchy of values as a flat arrangement of values, by using a "spanning" (i.e., horizontal) perspective on the deep (i.e. vertical) value sequence. Since the multiple classification mechanism needed to support this solution can be regarded as acknowledging multiple classification in the domain and the Updatable type can also be considered to be a natural domain abstraction – some model elements are updatable, others are not – we classified this solution also as "additional mechanism" required.

Level-blind fields, like dual fields, face the challenge of appropriately addressing STSV-properties, since they inherently support multiple values in a field chain, as opposed to just one value. The adequate modeling of an STSV-property requires explicitly not allowing values except at the target elements, which is why simply refraining from using the option to add a value is not sufficient. In other words, adequate field declarations for modeling STSV- and STMV-properties must necessarily be different in order to express the different modeling intent and to prevent undesired value occurrences. The solution presented in Section IV-E addresses this by suppressing undesired values with a constraint. The opposite, i.e., the forcing of a value presence at every element is achieved by the constraint attached to the lastUpdated field, hence the respective table entries in Table II. The inherent flexibility of refinement, in particular, the choice over refinement chains lengths, makes it difficult to make any guarantees as to where, i.e., at which associated concept, fields will received a value. The {concrete} annotation represents a solution regarding bottom-level presence, but there is no respective mechanism to enforce the association of certain field values or field redefinitions (e.g., color : RGBColor) to particular elements (e.g., C5III). In the small domain example we chose, it happens to be the case that C5III is the only possible place to redefine color so that the bottom-level color field will have an order-zero value, but in general (in a model with more refinement steps), that redefinition could appear anywhere in the middle of the respective refinement chain, i.e., there is no way to make sure that color models are associated with car models. As a result, MTSV-properties can be handled by level-blind fields in principle, but not in a way that exactly meets the domain requirements of our sample domain (cf. Table II).

B. Caveats

Several factors suggest that judgements about the presented approaches should be made cautiously:

TABLE III
Rating Categories

rating	description
✅	fulfilled
✅📝	fulfilled, with constraints
✅🩹	fulfilled, with additional mechanism
✓	only partially fulfillable
✅🔧	workaround required

1) Incomplete Evaluation: We focused on a single aspect – the ability to deeply characterize field types – of field semantics only. While identifying differences in addressing this sole aspect is useful, a particular complete field semantics design will address other aspects as well, e.g., control over the mutability of values. Furthermore, we solely considered technical trade-offs and therefore did not cover the immensely important aspect of end user usability.

2) Incomplete Benchmark: The choice of our sample domain scenario impacts on the evaluation of approaches. First, its size – covering only three domain levels – does not explicitly tease out all differences between the approaches. Sometimes we pointed out how deeper structures would impact on approaches but we did not do this comprehensively. Neither did we cover all possible property types. For example, "level-blind fields" would require further annotations, if so-called direct properties [16] had to be modelled, and covering "regularity properties" [16] would uncover more differences between the approaches.

Furthermore the domain requirements are biased towards regulating the properties of specific elements. This favours approaches based on classification level hierarchies, due to their predictable level structure and respective modeling element placement, plus the precise control over field occurrences via potencies. Based on our analysis, it appears that a different set of evaluation criteria that emphasises exploration over rigid specification would most likely play more to the strengths of refinement-based approaches. It is probably reasonable to suspect that these different kinds of multi-level modeling approaches will perform best in different application areas, due to their differences regarding flexibility and ability to precisely associate properties with specific modeling elements.

3) Incomplete Coverage: We chose deep characterization approaches mostly based on their fundamental characteristics and our expertise. There are many more approaches that could have been included in our analysis. For instance, concretization-based approaches [17]–[19], approaches supporting level-jumping [20], or vitality-based variants of dual fields [14].

In addition, variations of all approaches we have covered and did not cover, could be considered, such as single fields that do not feature intermediate occurrences of the field, etc. In short, we only looked at a subset of existing and conceivable deep characterization designs. Finally, in our analysis, we only covered field multiplicities of exactly one ([1..1]), i.e., we left considering the implications of optional and multi-valued fields as future work.

VI. Conclusion

Although multi-level modeling has reached some maturity and acceptance, it is still hotly debated. For some aspects of MLM, existing languages employ a variety of approaches, sometimes with fundamentally different underpinnings. The semantics of fields are chief among the concepts that lack a common consensus in the community. In this paper we therefore attempted to illuminate the differences between existing field approaches, restricting our focus to field types intended to support deep characterization. While such a relatively narrow focus bears the risk of missing a holistic understanding, we felt it was necessary to start with one aspect of field semantics and to widen the focus later on.

Since our concern was specifically about "deep fields" – i.e., ways to capture domain properties that demand regulation at higher levels but do not necessarily involve the manifestation of (order-zero) values before lower levels are reached – we identified four theoretically possible kinds of deep domain properties: STSV-, STMV-, MTSV-, and MTMV-properties. However, since the last of these has no obvious practical uses, we restricted our analysis to the first three. To the best of our knowledge, this is the first time deep domain properties have been categorized in this manner.

We then applied five canonical approaches that support the deep characterization of fields – shallow fields, single fields, dual fields, hierarchical fields, and level-blind fields – to a domain scenario that embodies the above property kinds. In the presentation of the approaches, as well as in a dedicated discussion section, we identified trade-offs, the nature of necessary workarounds, and how the approaches relate to each other. We acknowledge the limitations of our comparison with respect to approach coverage, breadth of domain requirements, and evaluation aspects considered. We nevertheless hope that our critical surveying of an important part of the MLM design landscape will add clarity to the understanding of which design options for deep fields are available, what their respective merits are, and what a potential future consensus might center around.

References

[1] M. Minsky, "A framework for representing knowledge," 1974.
[2] C. Atkinson, "Meta-modeling for distributed object environments," in *Enterprise Distributed Object Computing*. IEEE, Oct. 1997, pp. 90–101.
[3] C. Atkinson and T. Kühne, "The essence of multilevel metamodeling," in *Proceedings of the 4^{th} International Conference on the UML 2000, Toronto, Canada*, ser. LNCS 2185. Springer, Oct. 2001, pp. 19–33.
[4] ——, "Rearchitecting the UML infrastructure," *ACM Transactions on Modeling and Computer Simulation*, vol. 12, no. 4, pp. 290–321, Oct. 2003.
[5] C. Partridge, S. de Cesare, A. Mitchell, and J. Odell, "Formalization of the classification pattern: survey of classification modeling in information systems engineering," *Software & Systems Modeling*, vol. 17, no. 1, pp. 167–203, February 2018.
[6] J. Stein and D. Maier, *Associative Access Support in GemStone*. Springer, 1991, pp. 323–339. [Online]. Available: https://doi.org/10.1007/978-3-642-84374-7_20

[7] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[8] G. Mezei, Z. Theisz, S. Bácsi, F. A. Somogyi, and D. Palatinszky, “Towards flexible, rigorous refinement in metamodeling,” in *2019 ACM/IEEE 22nd Int. Conf. on Model Driven Engineering Languages and Systems Companion*, 2019, pp. 455–459.

[9] J. Odell, “Power types,” *Journal of Object-Oriented Programming*, vol. 7, no. 2, pp. 8–12, May 1994.

[10] V. A. Carvalho and J. P. A. Almeida, “Toward a well-founded theory for multi-level conceptual modeling,” *Software and Systems Modeling*, vol. 17, pp. 205–231, 2018.

[11] M. A. Jeusfeld and B. Neumayr, “DeepTelos: Multi-level modeling with most general instances,” in *Conceptual Modeling - 35th International Conference, ER 2016*, 2016, pp. 198–211.

[12] A. Lange, “dACL: the deep constraint and action language for static and dynamic semantic definition in Melanee,” Master’s thesis, University of Mannheim, 2016. [Online]. Available: http://ub-madoc.bib.uni-mannheim.de/43490/

[13] T. Kühne, “Multi-dimensional multi-level modeling,” *Software and Systems Modeling*, vol. 21, no. 2, pp. 543–559, 2022.

[14] C. Atkinson and R. Gerbig, “Melanie: Multi-level modeling and ontology engineering environment,” in *Proc. Modeling Wizards’12*. ACM, 2012.

[15] C. Atkinson, R. Gerbig, and T. Kühne, “Comparing multi-level modeling approaches,” in *Proceedings of the 1st International Workshop on Multi-Level Modelling co-located with the 17th ACM/IEEE International Conference MODELS 2014*, ser. CEUR Workshop Proceedings, vol. Vol-1286, 2014, pp. 43–52.

[16] J. P. A. Almeida, V. A. Carvalho, C. M. Fonseca, and G. Guizzardi, “A note on properties in multi-level modeling,” in *2021 ACM/IEEE Int Conf Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE Computer Society Press, 2021, pp. 497–501.

[17] B. Neumayr, M. A. Jeusfeld, M. Schrefl, and C. G. Schütz, “Dual deep instantiation and its ConceptBase implementation,” in *Proceedings CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014*, ser. Lecture Notes in Computer Science, vol. 8484. Springer, 2014, pp. 503–517.

[18] B. Neumayr, C. G. Schuetz, M. A. Jeusfeld, and M. Schrefl, “Dual deep modeling: multi-level modeling with dual potencies and its formalization in F-Logic,” *Software & Systems Modeling*, pp. 1–36, 2016.

[19] T. Clark and U. Frank, “Multi-level modelling with the FMMLx and the XModelerML,” Modellierung 2020, pp. 191–192, 2020.

[20] J. de Lara, E. Guerra, R. Cobos, and J. Moreno-Llorena, “Extending deep meta-modelling for practical model-driven engineering,” *The Computer Journal*, vol. 57, no. 1, pp. 36–58, 2012.