Extending Deep Meta-Modelling for Practical Model-Driven Engineering

Juan de Lara, Esther Guerra, Ruth Cobos, Jaime Moreno-Llorena

Departamento de Ingeniería Informática,
Escuela Politécnica Superior,
Universidad Autónoma de Madrid (Spain)
Email: {Juan.deLara, Esther.Guerra, Ruth.Cobos, Jaime.Moreno}@uam.es

Meta-modelling is one of the pillars of Model-Driven Engineering (MDE), where it is used for language engineering and domain modelling. Even though the current trend is the use of two-level meta-modelling frameworks, several researchers have pointed out limitations of this scheme for some scenarios and suggested a meta-modelling approach with an arbitrary number of meta-levels in order to obtain more flexible and simpler system descriptions. Unfortunately, such multi-level meta-modelling systems are still in their infancy, lacking for example, integration with model manipulation languages, a characterization of different possibilities for instantiation and inheritance, and primitives for interconnecting multi-level languages in a flexible way.
In this paper, we propose a number of extensions to multi-level (also called *deep*) meta-modelling, based on the needs raised by its use for practical MDE. In particular, we discuss on the issues related to code generation from deep languages, the benefits of allowing inheritance at every meta-level, and patterns and techniques for a fine-grain control of the meta-level of elements. Finally, we provide primitives to control the impedance mismatch when connecting models at different meta-levels.

Keywords: Model-Driven Engineering; Deep Meta-Modelling; Multi-level Meta-Modelling; Deep Characterization; Potency; Code Generation

Received 00 September 2011; revised 00 April 2012

1. INTRODUCTION

Model-Driven Engineering (MDE) [1] is a Software Engineering paradigm that aims for the automated development of software systems by the use of models and their successive refinement into code. Hence, it promotes the use of models as the primary assets in the development, where they are used to specify, simulate, analyse, generate and maintain applications [2]. Models can be expressed in general-purpose modelling languages like the UML [3], but frequently they are built using Domain Specific Languages (DSLs) especially tailored to a specific area of concern. In MDE, DSLs are defined through meta-models, which are models that describe the abstract syntax and main concepts of the DSL.

The OMG has proposed the Meta Object Facility (MOF) [4] as the standard language to describe meta-models, and some popular implementations exist, most notably the Eclipse Modeling Framework (EMF) [5]. In this approach, a system is described using two meta-levels: a meta-model defining allowed types, and a model instantiating these types. While the meta-model is built by the language designer, models are built by the final users. However, this two-level approach has limitations when the meta-model includes the *type-object* pattern [6], which requires an explicit modelling of types and their instances at the same meta-level. Occurrences of this pattern are found in general-purpose languages like the UML, where both *classes* and *objects* are defined at the same meta-level, as well as in domain-specific ones like web modelling languages (node types/node instances), role access control languages (user types/users), process modelling languages (task definition/task enactment, work product kind/work product) or e-commerce models (product type/product instance). When this pattern appears, a description using more than two meta-levels yields simpler models [7]. This organization permits placing types in a meta-level above their instances. We call a language *deep* if it permits its instantiation at more than one meta-level.

Although some works have discussed different aspects of multi-level (also called *deep*) meta-modelling [8, 9, 10, 11, 12], there is still a lack of practical experience

in its application to real case studies involving MDE processes, and on the technical needs arising from its use in large, complex systems. Here we report on our work and experience in this direction and propose some extensions that we incorporated to a multi-level meta-modelling framework in order to enable its use in more complex scenarios in a simpler way.

As deep languages can be instantiated at more than one meta-level, and not just once, our first extension aims at improving the compositionality of DSLs with heterogeneous meta depth (i.e. with different number of meta-levels). This is of paramount importance since complex systems are hardly ever described with a single monolithic model, but using several languages and models to describe the system from different perspectives. However, an impedance mismatch can arise when connecting languages that span a different number of meta-levels, or models that belong to different meta-levels. To solve this problem we propose the notion of *deep reference*, which is a reference to an instance of a type at a certain lower meta-level, not necessarily the immediate one. This is useful to build highly extensible deep languages, where it is not possible to foresee the complete collection of direct instances of a given type, and hence they cannot be referenced explicitly. Additionally, we propose a compact notation to avoid the manual "identity" instantiation of types at every meta-level, useful when conceptually a type needs to "skip" one meta-level.

Other of our proposals aims at enriching and making modelling uniform at the different meta-levels by providing inheritance mechanisms at every meta-level. At the top-most meta-level, the elements are pure types (i.e. they do not have an instance facet, so they are similar to classes in standard meta-modelling) and hence inheritance works as in most object-oriented languages: the fields with type facet are inherited from parent to children elements. At the bottom meta-level, the elements are pure instances (similar to objects in standard meta-modelling) and inheritance is used to obtain default attribute values from the parent, which can be overridden by the children. At intermediate meta-levels (where elements have both class and object facets), inheritance has the combined effect for both types and instances.

The ideas presented in this paper emerged in the daily work with our multi-level meta-modelling tool MetaDepth [12], and have been implemented in this tool recently. The tool supports the definition of characteristics of instances of a given type beyond the immediate meta-level – so called *deep characterization* – through the concept of *potency* proposed by Atkinson and Kühne [6]. The potency is a number attached to the model elements to control the characteristics of their direct and indirect instances (instead of only the direct ones as in standard meta-modelling). In this way, MetaDepth allows the definition of *deep languages* in which the user of the language is able to work with several meta-levels (and not only with one meta-level as in standard DSLs), thus enabling Domain Specific Meta-Modelling [13]. The tool integrates several model management languages of the Epsilon family [14] for an effective support of MDE. In particular, it is possible to define constraints and model manipulations with the Epsilon Object Language [15] (EOL), and build code generators by using the Epsilon Generation Language [16] (EGL). In this way, we realize the MDE vision of generating the final application from models, but with the peculiarity that we are not restricted to two meta-levels only. We will discuss the implications of the availability of multiple meta-levels for MDE throughout the paper.

The paper illustrates the different proposals through a non-trivial running example consisting on the automatic generation of web-based collaborative applications using MDE. For this purpose, we have used MetaDepth to design a family of (deep) languages enabling the description of the application presentation, the allowed users and their roles, and the connection of predefined, heterogeneous Web 2.0 components in a mashup-like style. In this domain, the type-object pattern appeared several times (component types/instances, table data types/values), and hence we found advantages in the use of a description using more than two meta-levels. Please note that for many other domains and scenarios, a description using two meta-levels is satisfactory.

The rest of the paper is organized as follows. We first present our running example in Section 2. Next, Section 3 introduces multi-level meta-modelling and deep languages, whereas Section 4 presents MetaDepth. Section 5 discusses different issues regarding the semantics of potency that we encountered when applying deep modelling to our running example, and the extensions we propose. Section 6 presents further extensions to make inheritance uniform at different meta-levels. Section 7 summarizes benefits and limitations of our approach, proposes patterns to organize the different elements of a deep language across meta-levels, and discusses lessons learnt. Finally, Section 8 compares with related work and Section 9 concludes the paper.

2. RUNNING EXAMPLE: MDE FOR COLLABORATIVE WEB APPLICATIONS

Collaboration-enabled tools are becoming the norm in computer science nowadays, and are pervasive in many facets of our daily life. Popular examples include Facebook, Google+ and Flickr for leisure, as well as applications supporting collaboration in design, project management and learning. The term *groupware* [17, 18] collectively refers to software supporting the actions of people involved in a common task, for the purpose of reaching some goal.

Web 2.0 technologies offer an ideal technological

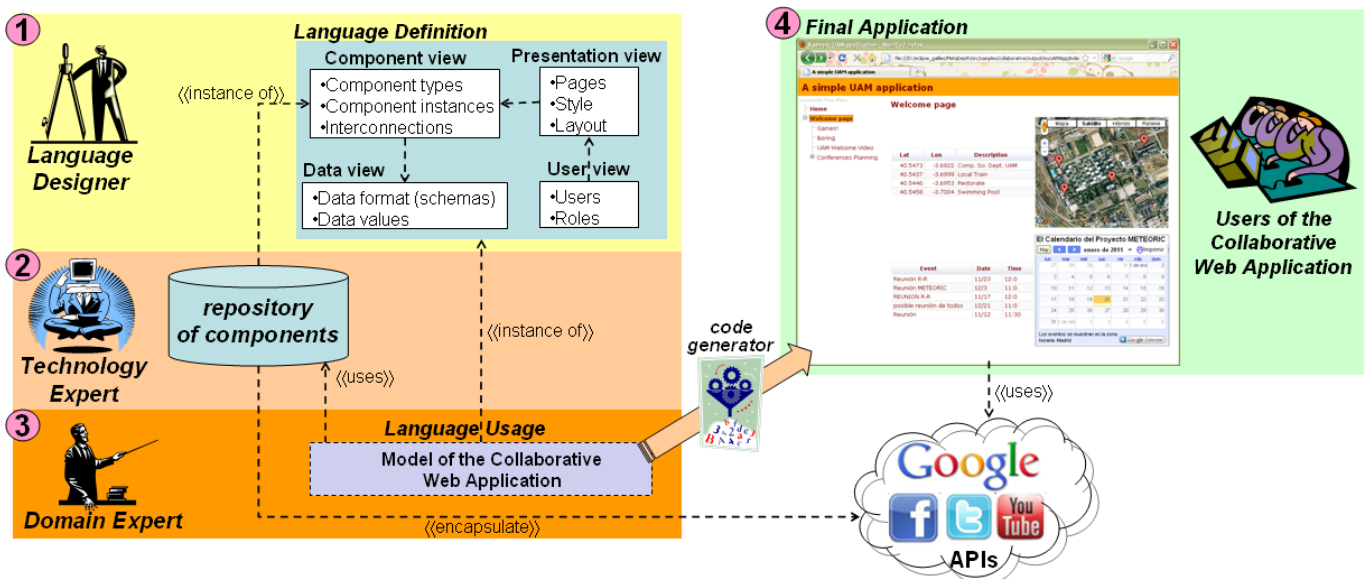


FIGURE 1: Applying MDE to the generation of tailor-made collaborative web applications

space to develop groupware applications. However, the heterogeneity of such technologies (which include HTML, RSS, Flash, AJAX and JavaScript), their continuous change and the ever-growing availability of new services and APIs make Web 2.0 applications difficult to build and maintain. This is so as developers need to be proficient in all these technologies for their combined use in the construction of this kind of applications. Moreover, the increasing need for a rapid release of tailor-made collaborative applications for specific communities of users makes urgent the development of methods and techniques for their automated construction.

In the context of the "Go-Lite" and "e-Madrid" projects, we are investigating ways to generate tailor-made collaborative web applications using MDE. For this purpose, we are developing a number of DSLs to describe different aspects of an application such as its users and their roles, its component-based functionality, and the style and layout of the rendered components. Our rationale is that such languages could be used by non-computer scientists and people without expertise in Web 2.0 technologies, hence facilitating the construction of web applications also by end-users. Then, from models of these aspects, a code generator synthesizes the final application. Figure 1 shows the general working scheme of the approach.

The application functionality is realized through a repository of predefined components which use heterogeneous technologies like Flash, Java Applets and JavaScript, and we also make use of the increasingly number of web-based services offered by companies like Google and Facebook. Nonetheless, in order to overcome their heterogeneity and facilitate their integration, we have abstracted and encapsulated these components and services in the form of models. In this way, our DSLs provide a uniform view of heterogeneous technologies, so that developers do not need to be experts in the low level technicalities and service APIs, but can describe the applications using higher-level models.

As an example, Figure 1 shows a generated application which gathers some Google components from different sources: a map, a table with the coordinates of the locations in the map, a calendar, and a table with the events contained in the calendar. The components communicate by exchanging data, whose format needs to be defined in a data view.

Our approach involves the following four roles in the process: language designer (label 1), technology expert (label 2), domain expert (label 3) and final user (label 4). The language designer is in charge of defining the family of DSLs. The technology expert builds component models that encapsulate technologies like Google maps, or APIs from specific services and vendors like Facebook. The domain expert models web applications for specific user communities by selecting, instantiating and connecting the predefined component models in the repository. Finally, a code generator synthesizes the web application from these models, to be used by the final users.

A challenge in this approach is a clean separation between component types and instances, enabling the addition of new component types to the repository as they become available, without having to change the DSL meta-model each time a component is added. Hence, first we need to define *component types* encapsulating widgets like the Google calendar, the Google map or several types of data visualizers. Each such component type needs to define its own set of parameter types. Then, such *component types* need to be instantiated giving explicit values to their

parameters, like the owner of the calendar or the locations to be shown in the map. As shown in Figure 1, these two tasks – encapsulating components in the form of a model (i.e. defining a component type), and using them in order to build applications – are normally realized by different people. Moreover, the architecture needs to be extensible as we need to be able to add new component types to our repository at any time. Currently, we have 33 components in the repository including components for polling, web searching, educational simulations, file storage, RSS readers, forms, chats, Facebook posts, Flickr queries and data visualization. However, this number is growing as we find new services and APIs to encapsulate.

The modelling of component types and instances is an example of the type-object pattern. Another occurrence of this pattern arises when modelling the data to be used by the application, as there is the need to define data schemas (e.g. describing the structure of events in a calendar or geo-positioned locations), and then to instantiate them.

As we will show in the next section, resorting to a two meta-level style to define our DSLs implies mixing in the same meta-level both component types and their instances, as well as data schemas and data values. This makes intricate the automatic construction of instances from types, and the conformance testing of instances with respect to types. Instead, we use an alternative solution that separates types and instances in different meta-levels. However, we realized that current multi-level meta-modelling techniques were not capable of handling this complex scenario. Therefore, the realization of this solution raises the need for advanced, novel multi-level modelling techniques and patterns, which we will present in Sections 5 and 6.

3. MULTI-LEVEL META-MODELLING AND DEEP LANGUAGES

Meta-modelling involves building models that describe other models. One of its most common uses is to describe the abstract syntax of modelling languages [19], so that the meta-model defines the set of models considered valid in the language. A prominent approach to meta-modelling is the OMG's *strict* four-level architecture. Strictness refers to the fact that an element at a certain meta-level has to be the instance of exactly one type at the meta-level above. In this architecture, only two adjacent meta-levels are available at a time: the upper one contains meta-models (e.g. describing a language) and the lower one contains instances of these meta-models. While meta-models are built by language developers, models are built by the final users of the language (i.e. the domain experts). This is the mainstream approach nowadays, implemented by frameworks like the EMF [5]. In the following, we refer to this approach as the *two-level* approach.

Even though the two-level approach is the most used one nowadays, some authors have pointed out its limitations when it is used to build languages that incorporate the "type-object" pattern, i.e. that define in the same meta-level a type and its instances [7, 20].

For instance, our case study requires a language for components allowing the definition of component types, which are then instantiated and connected through connectors [21]. The definition of part of this language using two meta-levels is shown in the upper part of Figure 2. The language meta-model defines two types of components, `JavaScriptType` and `AppletType`, together with their instances, `JavaScript` and `Applet`. This is an occurrence of the type-object pattern, where the instantiation relation of each component instance with its type is modelled through the association `type`. Moreover, component types can declare features, to which the component instances should assign a value through the corresponding slot. Again, this is another occurrence of the type-object pattern.

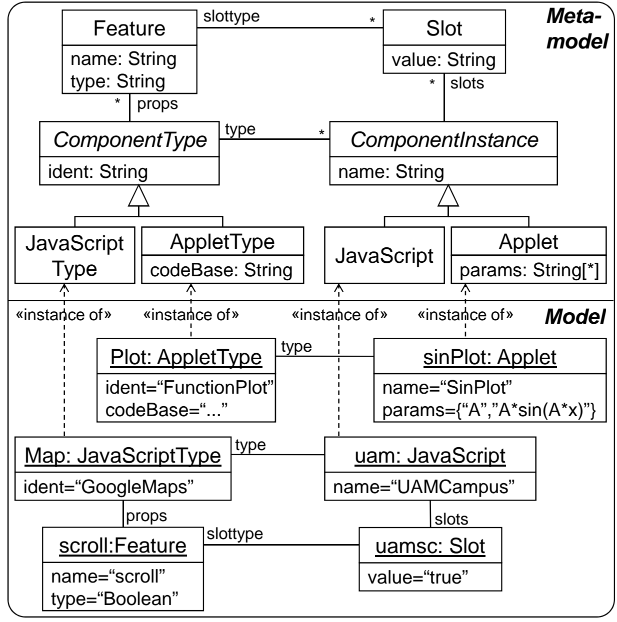


FIGURE 2: A two-level language for components

Below, the figure shows a model example with two component types (`Plot` and `Map`) and one instance of each (`sinPlot` and `uam`). Thus, the steps to create a correct component instance are the following: instantiating a subclass of `ComponentInstance`, selecting its correct `ComponentType` (`Plot` or `Map` in the figure), manually creating one slot in the component instance for each feature declared in the component type, linking the slots to the features through `slottype` links, and assigning the slots a valid value according to the data type declared in the feature. For example, in the model, the `Map` component type declares a feature `scroll`, and therefore the

component instance `uam` has been provided with a slot to assign this feature a value. Thus, the domain expert has to instantiate the appropriate slots manually, which is time-consuming and error-prone. Please note that the language meta-model could include OCL constraints to automatically *check* the correctness of the *typing* links (`type` and `slottype`), but not for the automatic construction of correct component instances declaring appropriate slots for the feature values. To automate such a creation mechanism, a two-level meta-modelling architecture should be extended with extra functionality.

Another limitation of this two-level solution is that it is not possible to define common features and behaviours for several component types at the model level, e.g. a DataVisualizer `JavaScriptType` component that is specialized by several children like PieChartVisualizer and BarDiagramVisualizer. Such an inheritance mechanism is only available at the meta-model level, and therefore it should be implemented ad-hoc at the model level. Moreover, the models of components would become further involved should we need to assign cardinalities to the features declared in the component types. For instance, although it is not shown in Figure 2, the `GoogleMaps` component type declares as a feature zero or more geo-positions to be visualized in the map, which should be assigned a value in the component instances. Altogether, the two meta-level solution for our components language becomes convoluted and hardly usable, or requires many ad-hoc functionalities to be built by hand. In the limit, the needed functionality amounts to building a complete type system emulating the instantiation and type checking mechanisms that standard meta-modelling systems offer between two adjacent meta-levels.

Instead, one can organize the language in a simpler way using three meta-levels, as Figure 3 shows. The resulting models are simpler if we take the size of the models as a measure of their complexity. Thus, the same system is modelled with 7 objects in the three-level solution, but with 14 objects and 9 links in the two-level solution (see Figure 2).

In this multi-level architecture, each element has a *potency* [6] indicated after "@". If an element does not explicitly indicate a potency, then it inherits the potency from its container element, and ultimately from the model where it belongs. The potency is a mechanism for the *deep characterization* of indirect instances of elements, that is, it helps in setting the characteristics of direct and indirect instances of a given element. It is a natural number that indicates in how many consecutive meta-levels a given element can be subsequently instantiated. If it is assigned to a field, then this can be given a value only in the deepest meta-level allowed for the field. At each deeper meta-level, the potency of the instances decreases in one unit. When it reaches zero, we obtain a pure instance that cannot be instantiated further.

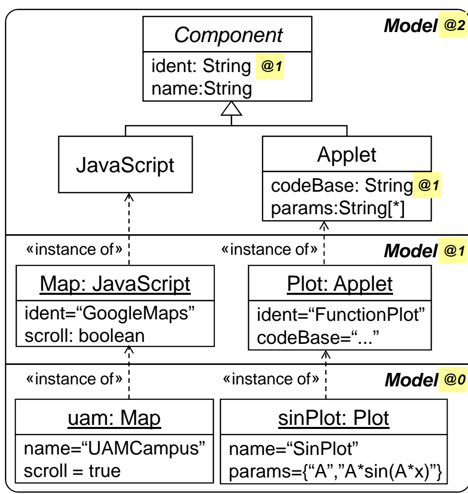


FIGURE 3: A three-level language for components

In the three-level architecture shown in Figure 3, the top compartment contains the definition of our language for components. Its (meta-)model has potency 2, which means that all its elements can be instantiated in the following two meta-levels, except the fields with potency 1, which can be instantiated only in the following meta-level. This mechanism is very useful in the design of deep languages, as the designer can express in the top meta-level the attributes (e.g. `name`) that indirect instances (`uam` and `sinPlot`) should have. For example, the `Applet` type has potency 2, and its instance `Plot` in the next meta-level has potency 1. When `Plot` is created, all fields declared in its type are automatically instantiated as well (`ident`, `name`, `codeBase` and `params`) and their potency gets decreased in one unit. Then, it is possible to assign a value to those with potency 0 (`ident` and `codeBase`), whereas those with potency bigger than 0 are usually kept invisible to the user. As `Plot` has potency 1, it can be instantiated in the next meta-level, as `sinPlot` shows. When `sinPlot` is created, all fields declared in its type with potency bigger than 0 are automatically instantiated (`name` and `params`). These two fields receive potency 0 at this meta-level, and therefore they can be given a value. `sinPlot` has potency 0 and hence cannot be instantiated further.

It can be observed that, as we can instantiate component types like `Map` or `Plot`, it is no longer necessary to include the meta-class `ComponentInstance` and its subclasses in the language definition (like in Figure 2). This is so as each component instance (`uam` and `sinPlot`) is an instance of an instance of `Component` (i.e. instance of `Map` and `Plot` respectively). In this way, the `type` relation that we used in the two-level solution is no longer needed, as the usual *instance of* relation between meta-levels takes care of the typing. As a consequence, in the three meta-level architecture, less modelling elements are needed to model the same system. Moreover, there is no need to build by hand extra machinery to deal with

instantiation and type checking in the same meta-level, because such mechanisms are built-in in the system and available across meta-levels.

In a multi-level setting, the elements in the top meta-level are pure types, the elements in the bottom meta-level are pure instances, and the elements at intermediate meta-levels retain both a type and an instance facet. They all are named *clabjects*, because of the merging of the words class and object [7]. For example, in Figure 3, `Plot` retains both class and object facets: it is an instance of `Applet` and gives value to fields `ident` and `codeBase`, and at the same time is a class enabling the creation of instances like `sinPlot`.

The *instance of* relations depicted in Figure 3 provide an ontological typing to the clabjects, as they represent instantiations within a domain. Hence, *ontological meta-modelling* is concerned with describing the concepts in a certain domain and their properties [22]. In addition, multi-level meta-modelling frameworks usually support an orthogonal linguistic typing [7, 12] which refers to the meta-modelling language elements used to build the models. For instance, the linguistic type of `Component`, `Map` and `uam` is `Clabject`, whereas the linguistic type of `ident` and `name` is `Field`.

Figure 4 shows the dual typing of our deep language for components. The left of the figure contains a small excerpt of the linguistic meta-model over which all models (with any potency) to the right are typed. Actually, one can understand the union of the three models to the right as a normal instance of the linguistic meta-model to the left. The linguistic typing has the additional advantage that one can write generic specifications using the linguistic types instead of the ontological types, so that they become applicable to any model at any meta-level.

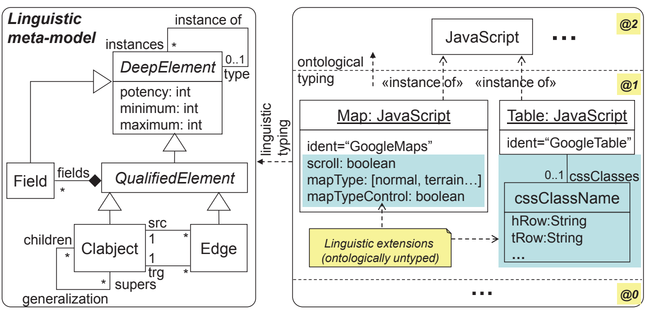


FIGURE 4: Dual linguistic/ontological typing of deep languages, and linguistic extensions

In addition, models at any meta-level can be extended *linguistically* by allowing clabjects to define further fields, as well as by introducing new clabjects without ontological typing [12]. This provides extensibility to deep languages. For instance, clabject `Map` in Figure 3 defines a new field `scroll` at potency 1, which is instantiated at potency 0. Figure 4 shows further linguistic extensions to our example: new fields in clabject `Map`, and a CSS style for `Table` which is defined as a clabject without ontological typing. These linguistic extensions could not be anticipated in the upper meta-level since each JavaScript component may have its own set of features. Moreover, this mechanism makes unnecessary the explicit modelling of classes `Feature` and `Slot` in the upper meta-level as it was required in the two-level solution of Figure 2, and it guarantees that each instance of `Map` will be automatically created with the appropriate features, and type checking will be automatically performed. Thus, the `uam` clabject at potency 0 is created with appropriate fields `name`, `scroll`, `mapType` and `mapTypeControl` because `name` was declared in `Component` with potency 2, and the rest of fields were declared in `Map` with potency 1 (see Figure 4). Then, we can give them a value in `uam` and use the built-in type checking mechanism of the system. We will see in Section 6 that our approach enables using inheritance relations at intermediate meta-levels, as a form of linguistic extension as well. In this way, potency combined with linguistic extensions enables a kind of flexible, extensible domain-specific meta-modelling capability [13].

Hence, a multi-level framework can be seen as an extension of a two-level framework in two ways. First, it extends the instantiation mechanism to enable instantiating elements at more than one meta-level down. Thus, deep languages generalize standard modelling languages in which users work at the same meta-level. Second, as seen in Figure 4, one can view the union of all models in a deep meta-modelling stack as a standard instance of the linguistic meta-model.

Altogether, the three meta-levels in the architecture of our DSL for components establish a neat distinction between the language definition at potency 2, the repository of reusable component types at potency 1, and the instantiation of these predefined components to be customized in particular web applications at potency 0. As shown in Figure 1, each meta-layer is usually developed by different people with different roles, and adopting a deep language neatly reflects this separation of responsibilities, facilitating the generation of customized modelling tools for each role.

4. DEEP META-MODELLING WITH METADEPTH

MetaDepth [12, 23] is a multi-level meta-modelling tool that we started to develop in 2008 in the context of the METEORIC project. The tool supports textual modelling through a uniform textual concrete syntax across meta-levels, and implements deep characterization through potency. This section briefly presents its syntax, as we will use it to illustrate our proposed extensions for deep meta-modelling. The reader is referred to [12] for more details.

As an example, Listing 1 shows the MetaDepth

specification of the three meta-level solution shown in Figure 3. Models, clabjects and fields can be decorated with a potency, specified after the '@' symbol. If no potency is specified for an element, then it inherits the potency of its container element. The potency can be any positive number, or zero. The instances of an element always receive the potency of the element minus 1. An element with potency 0 cannot be instantiated.

For instance, the model `ComponentView` declared in lines 1-11 has potency 2. All clabjects declared inside this model (`Component`, `JavaScript` and `Applet`) receive the same potency because they do not specify a potency on their own. The features declared in lines 3 (`ident`) and 8 (`codeBase`) override the default potency with 1. Once declared, the model `ComponentView` becomes a type and hence can be instantiated. Any instance of this model (like the one in lines 13-22) has potency 1, and this cannot be overridden.

```
1  Model ComponentView@2 {
2    abstract Node Component {
3      ident@1 : String;
4      name    : String {id};
5    }
6    Node JavaScript : Component {}
7    Node Applet : Component {
8      codeBase@1 : String;
9      params : String [*];
10   }
11 }
12
13 ComponentView RepositoryComponents {
14   JavaScript Map {
15     ident = "GoogleMaps";
16     scroll  : boolean = false;
17   }
18   Applet Plot {
19     ident = "FunctionPlot";
20     codeBase="...";
21   }
22 }
23
24 RepositoryComponents myApplication {
25   Map uam {
26     name = "UAMCampus";
27     scroll  = true;
28   }
29   Plot sinPlot {
30     name = "SinPlot";
31     params = ["A", "A*sin(A*x)"];
32   }
33 }
```

Listing 1: Definition of the three-level language for components using MetaDepth.

`Node` is the MetaDepth keyword for top-level clabjects (i.e. clabjects with no ontological typing). Inheritance is declared with a colon (see lines 6 and 7), and the tool supports multiple inheritance. Fields can be declared in the context of `Node`s, `Model`s and `Edge`s. A field has to declare a name and a type, and optionally a potency, initial value, multiplicity (an interval between brackets) and modifiers (between braces). For example, the `name` field in line 4 has type `String` and is declared as an identifier (modifier `id`), therefore its value must be unique for all clabjects at potency zero. Other allowed modifiers include `ordered` (to keep elements in a collection in order of assignment) and `unique` (a collection without repetitions). The field `name` does not declare a potency, and hence it receives the one of its container (`Component`, which has potency 2). It does not declare a multiplicity either, so `[1..1]` is assumed. Line 16 shows the declaration of a field (a linguistic extension) with an initial value. MetaDepth allows controlling whether some model or clabject instance can be linguistically extended or not. References are just a special case of fields with non-primitive type.

Elements with potency bigger than zero become types and can be instantiated. For example, the top-level model `ComponentView` declared at line 1 has potency 2, and hence can be instantiated by using its name as a type (see line 13). This is also the approach to instantiate nodes and edges. For example, the `JavaScript` node declared in line 6 is instantiated in line 14 by clabject `Map`. The latter has potency 1, and hence also becomes a type, which can be instantiated as shown in line 25. When a clabject is instantiated, all its fields with potency bigger than zero get automatically instantiated in the clabject instance, their potency is decreased in one unit, and those that reach potency 0 can be assigned a value. For example, clabject `sinPlot` in lines 29-32 is created with two fields, `name` and `params`, which can be given a value because their potency is zero.

In addition to `Node`s, MetaDepth allows the definition of `Edge`s to declare bidirectional associations between `Node`s. Edges are made of two opposite references and, similar to associative classes in UML, can define fields. As an example, line 10 in Listing 2 shows the definition of an edge named `Produces` modelling a bidirectional association between the reference ends `produces` and `generator` of nodes `Event` and `OutParam` respectively. In this case, the edge does not declare any field. Currently, we restrict the potency of edges to be the same as the opposite references they connect, which therefore should be equal. Lines 14-19 show a model with an instance of edge `Produces` in line 18.

```
1  Model ComponentView@2 {
2    Node Event@1 {
3      produces : OutParam[*];
4    }
5    Node Param { ... }
6    Node OutParam : Param {
7      generator@1 : Event[*];
8    }
9
10   Edge Produces@1(Event.produces, OutParam.generator) {}
11   ...
12 }
13
14 ComponentView RepositoryComponents {
15   ...
16   Event TableClick { ... }
17   OutParam TableData { ... }
18   Produces TableEvent (TableClick, TableData);
19 }
```

Listing 2: Defining edges in MetaDepth.

MetaDepth supports the definition of constraints and derived fields by using Java or the Epsilon Object Language (EOL) [15]. EOL is a language for model manipulation with syntax similar to OCL and

additional constructs to create objects, assign values or iterate, among others. Constraints have a name and a potency specifying the meta-level in which they are to be evaluated. They can be associated to `Nodes`, `Edges` and `Models`.

For example, the constraint shown in Listing 3 ensures that any indirect instance of `ComponentView` at potency 0 contains at least one indirect instance of `Component`. The constraint, named `minComponents`, queries for the set of (indirect) instances of `Component` and ensures that its size is bigger than zero. The method `allInstances` returns the set of all (probably indirect) instances of the given type that exist in the model where the method is evaluated. As the constraint has potency 2, it will be evaluated two meta-levels below, so that the method `allInstances` will retrieve all indirect instances of `Component` with potency 0. If the constraint had potency 1, then it would be evaluated in the next meta-level and would return the instances of `Component` with potency 1. Hence, MetaDepth provides a transparent mechanism to retrieve the indirect instances of types (i.e. the instances of instances – and so on – of a given type).

```
1 Model ComponentView@2 {
2    minComponents : $Component.allInstances().size()>0$
3    ...
4 }
```

Listing 3: Defining constraints in MetaDepth.

In MetaDepth, we also use EOL to define in-place model transformations like simulators and redesigns [23]. Moreover, in order to realize the MDE vision of generating applications from models, the tool includes an innovative code generation facility for deep languages which we describe next.

4.1. Code generation for deep languages

In order to enable code generation from MetaDepth models, we have integrated the Epsilon Generation Language (EGL) [16] into our tool. EGL is a template-based language specifically designed for generating code from models that conform to a meta-model (i.e. EGL assumes two meta-levels). Therefore, integrating EGL required tackling the peculiarities of deep languages, in particular to take care of indirect instances of types. Hence, one can build templates applicable several meta-levels below, not necessarily the immediate one.

As an example, suppose we need to generate HTML code displaying the list of components in an application (i.e. the instances of `Component` at potency 0). This presents problems as we do not know in advance the direct instances of `Component` that should be used in the EGL template, and we do not want to modify the template each time a `Component` instance is created at potency 1. Since our framework needs to be extensible, allowing the dynamic addition of such instances, we need a transparent mechanism to retrieve the indirect instances of types.

The required EGL template is shown in Listing 4. In general, an EGL template writes its content into a file, except the text between the symbols "[%" and "%]", which is EOL code and gets executed on the input model. In the listing, the template iterates on all existing (direct or indirect) instances of `Component` in the input model in line 10, writing their name and identifier in line 11. If we apply this template to our example model at potency 0, we obtain a list with two items that correspond to the clabjects `uam` and `sinPlot`, as they are indirect instances of `Component`. From a clabject we can access its type fields transparently, as expression `c.ident` in line 11 shows. This field was declared with potency 1, so that e.g., all instances of `Map` will retrieve the same value for it, becoming similar to a static attribute in Java from the point of view of the `Map` instances with potency 0. Such code generator can be invoked from the MetaDepth console, using the `load` command, followed by the template name. Additionally, EGL has commands to load and execute a template from another template (as we will see later), hence allowing a modular description of code generators.

```
1  <!DOCTYPE HTML PUBLIC
2     "-//W3C//DTD HTML 4.01 Transitional//EN">
3  <html>
4    <head>
5      <meta http-equiv="content-type"
6              content="text/html;charset=ISO-8859-1">
7    </head>
8    <body>
9      <ul>
10       [% for (c in Component.allInstances()) { %]
11       <li>[%=c.name%] with identifier [%=c.ident%]</li>
12       [% } %]
13     </ul>
14   </body>
15 </html>
```

Listing 4: A simple EGL template.

In our experience working with deep languages, it is often necessary to define generators which do not know beforehand all possible direct and indirect instances of a given clabject. In our language for components, the benefit of our approach is that we define our template once at potency 2, then we create an arbitrary number of component types at potency 1 (maps, plots, games, data tables, etc.), and the generator can be applied to instances at potency 0 independently of the instances at potency 1. This fact enables extensible code generation architectures, where code generators defined at potency 2 are extended with further templates defined at potency 1. The combined generators are then applied to instances at potency 0.

Figure 5 shows the scheme for code generation from our family of DSLs. We have built a code generator that uses types with potency 2, and synthesizes the applications specified at potency 0 independently of the types at potency 1. To include a new component type in our repository (e.g. a gadget of Google or a new RESTful service), the technology expert needs to create the corresponding model at potency 1 (i.e.

an instance of a `Component` subclass), and indicate the specific EGL template to generate code for the component type, hence obtaining extensibility. The generic code generator at potency 2 places the selected component instances in the generated HTML page, and invokes the specific templates of the components whose name is stored in fields `gencodetemplate` and `gencodetemplatecanvas` defined in the `Component` clabject (see lines 5 and 6 of Listing 5). These two fields store the name of the EGL templates that will write code in the generated HTML header and body, respectively. Therefore, defining a new component type like the Google Map in lines 13-24 of Listing 5 requires providing the name of these two EGL templates (lines 18-19), and writing such EGL template, an excerpt of which is shown in Listing 6. The template generates JavaScript code that loads the google map library (lines 1–6), then creates a DataTable with the geopositions to be visualized (lines 10–19) and a function in charge of creating the map with the configuration options taken from the model (lines 21–31), showing the map and synchronizing it with other components via event handlers (not shown).

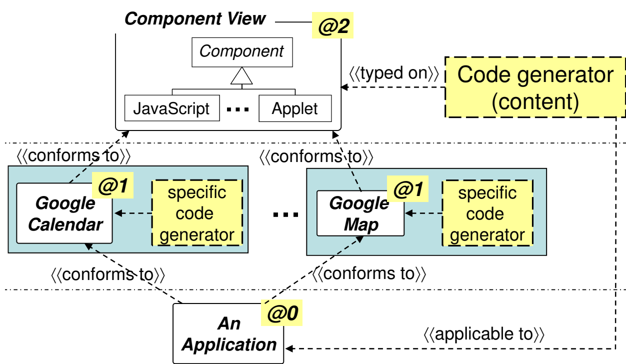


FIGURE 5: Architecture for code generation from our deep languages

```
Model ComponentView@2 {
  abstract Node Component {
    ident@1                  : String;
    name                     : String {id};
    gencodetemplate@1        : String;
    gencodetemplatecanvas@1 : String;
    ...
  }
  Node JavaScript : Component {...}
  ...
}

ComponentView GoogleMapComponent {
  enum mapTypes{ normal, terrain, satellite , hybrid }

  JavaScript Map {
    ident                 = "map";
    gencodetemplate       = "googlemaps.egl";
    gencodetemplatecanvas = "googlemaps_canvas.egl";
    maptype          : mapTypes=normal;
    useMapTypeControl : boolean=false;
    enableScrollWheel  : boolean=false;
  }
}
```

Listing 5: Defining a code generator for the `Map` component type.

```
<script type="text/javascript"
        src="http://www.google.com/jsapi">
</script>
<script type='text/javascript'>
   google.load(' visualization ', '1', {'packages': ['map']});
</script>
<script type="text/javascript">

var [%=component.name%];
[% var dataTable : String :=component.data.table.varid;%]
[% if (not genDataTables.includes(dataTable)){%]

var [%=dataTable%] = new google.visualization.DataTable();
[% for (col in component.data.table.cols){ %]
[%=dataTable%].addColumn('[%=col.dtype%]', '[%=col.label%]');
[%}%]

//... load the geopositions ...
}%]

function drawMap[%=component.name%]() {
 var options={}
 options['showTip']=true;
 options['mapType']='[%=component.maptype%]';
 options['useMapTypeControl']=[%=component.useMapTypeControl%];
 options['enableScrollWheel']=[%=component.enableScrollWheel%];
 [%=component.name%] = new google.visualization.Map(
             document.getElementById('[%=component.name%]'));
 [%=component.name%].draw([%=dataTable%], options);
 //... synchronize with other components...
}

google.setOnLoadCallback(drawMap[%=component.name%]);
</script>
```

Listing 6: Defining a JavaScript EGL template for the `Map` component type (googlemaps.egl).

As Figure 5 shows, the templates for code generation of the different components are called from another template in charge of generating the content of the application HTML pages. Listing 7 shows an excerpt of this latter template, which is invoked for each page in the presentation view. Line 6 iterates on each component in the page content. For each component, line 7 loads the corresponding template, line 8 adds the component as parameter to the template, and line 10 invokes the template, writing the resulting text in that position. This is the standard way to load, add parameters and invoke a template from another one in EGL.

```
<!DOCTYPE HTML PUBLIC
    "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
 <head>
  ....
  [% for (c in page.content) {
       var t : Template := TemplateFactory.load(c.gencodetemplate);
       t.populate('component', c);
  %]
      [%=t.process()%]
  [%}%]
 </head>
 <body>
   ....
```

Listing 7: Excerpt of the template for generating the content code.

Once we have introduced the main ingredients of deep meta-modelling and their realization in METADEPTH, the following sections discuss extensions and new deep meta-modelling features that we needed to include in our framework in order to be able to apply deep meta-modelling in practice. However, the proposed

extensions are general and applicable to any multi-level meta-modelling framework supporting potency.

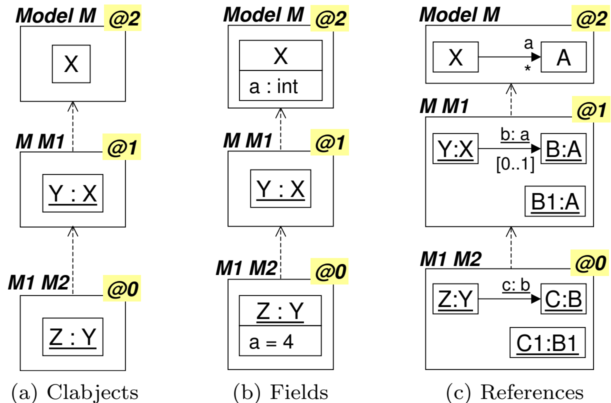
5. RICHER SEMANTICS FOR POTENCY

In this section we discuss some issues regarding the semantics of potency, introducing a novel extension to the standard semantics [7] aimed at providing flexibility in modelling and facilitating the composition of deep languages.

5.1. Instantiation semantics

A key issue when building a multi-level meta-modelling framework is the instantiation semantics of potency for the different modelling elements (clabjects, fields, references and edges). Even though there is a consensus in the literature concerning the semantics of clabjects and fields [8, 11], it is not so for references, constraints and edges [11, 24].

The instantiation semantics for clabjects is the following: a clabject `X` with potency `n>0` can be instantiated, leading to a clabject with potency `n-1`. If such an instantiation of `X` is not performed, then one cannot continue instantiating `X` at level `n-2`. This situation is illustrated in Figure 6(a), where a clabject `X` with potency 2 is instantiated once at level 1 (clabject `Y`) and then once again at level 0 (clabject `Z`). If `X` is not instantiated at potency 1, then we cannot obtain an indirect instance of `X` at potency 0. Hence, we say that this kind of instantiation is *mediated*.



(a) Clabjects (b) Fields (c) References

FIGURE 6: Semantics of the instantiation of potency

The standard instantiation semantics for fields with a primitive type is different. A field with potency `n` can only receive a value exactly `n` meta-levels below. As a consequence, at intermediate meta-levels, clabjects do not need to have a slot for it. Figure 6(b) illustrates this issue, where the field `a` with potency 2 receives a value at level 0. We therefore say that the instantiation of fields is *not mediated*. In MetaDepth, we relax this restriction, and the tool internally instantiates fields at intermediate meta-levels (e.g. at level 1 in the figure) in order to allow redefining their default value. In any case, the point is that the instantiation of fields at intermediate meta-levels *is not mandatory*, being the slot for the field strictly necessary at potency 0, but not at intermediate meta-levels.

The instantiation of references (i.e. fields with non-primitive type) is similar to that of clabjects, as a reference with potency `n` must be instantiated at potency `n-1` to enable further instantiations at potency `n-2`, and so on. Hence, the instantiation of references is also mediated. For example, Figure 6(c) shows a reference `a` with potency 2 that is instantiated once at level 1 (reference `b` of type `a`), and then reference `b` is instantiated at meta-level 0 (reference `c` of type `b`). The cardinality of a reference only restricts the number of instances in the immediate meta-level below. In the figure, `a` has cardinality `[0..*]`, therefore clabject `Y` at level 1 can define any number of `a` instances. In its turn, each instance can define its own cardinality to be evaluated in the meta-level below.

Note that the semantics of potency given to references does not allow connecting clabjects `Z` and `C1` in Figure 6(c), because the model at level 0 must be typed by the model at level 1, and this specifies that references of type `b` must point to `B` clabjects and not to `B1` clabjects. In contrast, one could adopt an alternative field-like instantiation for references enabling their instantiation only with potency 0. In this case, reference `a` could only be instantiated at level 0, and one could connect any indirect instance of `X` with any indirect instance of `A`, and in particular connecting `Z` and `C1`. As a drawback, with this second semantics it is not possible to fine-tune the cardinality of references at intermediate meta-levels as we have done in Figure 6(c), where we have assigned cardinality `[0..1]` to the reference at level 1.

In MetaDepth, edges have a potency and follow a clabject-like instantiation like in Figure 6(a). Constraints are also decorated with a potency and follow a field-like instantiation. In this way, a constraint with potency 2 will be evaluated exactly 2 meta-levels below, but not at the immediate level. Should we like the constraint to be evaluated at the immediate meta-level below, we just have to assign it potency 1.

Figure 7 shows an application of the previous discussion to our running example. Node `Component` declares a reference to `InParam` modelling the input parameters of the component. Each instance of the reference at potency 1 defines the type of one or more input parameters for a particular component type, whereas an instantiation at potency 0 specifies the actual value for a parameter. The cardinality of the `inParams` reference at potency 2 allows defining any number of parameter types at potency 1. The reference `data`, which is an instance of `inParams`, indicates that the `TableVisualizer` component has exactly one `DTable` as input parameter. If the visualizer had two tables as parameters, it would be enough to change the cardinality of this reference. This cardinality will be

evaluated in the models of potency 0.

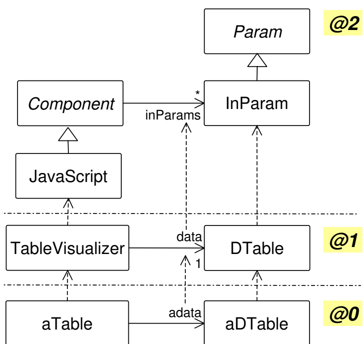


FIGURE 7: Example of instantiation of references

Listing 8 shows the METADEPTH definition of the two upper meta-levels in Figure 7. The reference `inParams` is declared in line 3 and instantiated in line 16. The name between braces in line 16 refers to the ontological type of `data`, whereas the number between brackets indicates its cardinality. As reference `inParams` is declared with cardinality `[*]`, clabject `TableVisualizer` could instantiate it any number of times, and not just once as in this example. Regarding the instantiation of primitive fields (like `name` or `ident`), it is automatically handled by METADEPTH when the clabject that defines them is instantiated.

```
1  Model ComponentView@2 {
2    abstract Node Component {
3      inParams : InParam[*];
4      name     : String;
5      ident@1  : String;
6      ...
7    }
8    Node JavaScript : Component { ... }
9    abstract Node Param { ... }
10   Node InParam : Param { ... }
11   ...
12 }
13
14 ComponentView RepositoryComponents {
15   JavaScript  TableVisualizer  {
16     data : DTable[1] {inParams};
17     ident = "TVisualizer";
18     ...
19   }
20   InParam DTable { ... }
21 }
```

Listing 8: Example of instantiation of references.

In conclusion, the presented instantiation mechanisms are extensions of those found in standard two-level meta-modelling, where all elements in meta-models would have potency 1. Moreover, the different instantiation styles shown in Figure 6 clarify the proposals currently found in the literature of multi-level meta-modelling [8, 11, 24].

5.2. Deep references

Complex systems are usually modelled from different viewpoints, each one of them focusing on a different aspect of the system. Hence, a system is described by a collection of interconnected models.

When using deep languages, sometimes a view needs to refer to the instances of a clabject defined in another view. This is problematic if the direct type of these "external" instances is unknown beforehand, and we only know their indirect type two or more meta-levels above. This is the case of our Presentation view. This view defines a clabject `Page` that stores a collection of `Component` indirect instances with potency 0, which are declared in the Component view. However, we cannot know in advance the direct type of the components in this collection, as it depends on the instances of `Component` with potency 1 (like `Map`, `Plot` and `TableVisualizer`). The declaration of this collection becomes problematic, since we want our language to be extensible so that new component types can be added at any time, but it is unfeasible to know the type of all possible component types of potency 1 in advance.

This issue does not arise in two-level meta-modelling approaches, as all references declared in a meta-model always refer to the next meta-level below. Thus, in a two-level setting, it would be enough to declare a collection of type `Component`. In our case, if we define in `Page` a collection of type `Component`, then the instances of `Page` will store direct instances of component with potency 1, but not indirect ones with potency 0 as it is needed.

To solve this problem we extend deep languages with the possibility of referencing clabjects by its top-level type, together with a potency indicating the depth of the instance that is to be referenced. This idea is shown in Figure 8, which depicts the relation between the Presentation and the Component views. In particular, `Page` (with potency 1) declares a reference to `Component` (with potency 2). However this reference is assigned potency 0, which means that it will store indirect instances of `Component` with potency 0.

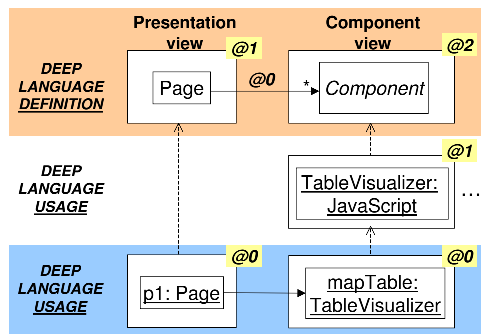


FIGURE 8: Presentation view: reference to deep clabject

In our experience, the languages used to model complex systems usually include views with heterogeneous number of meta-levels. As Figure 8 shows, our deep reference extension is needed to overcome the impedance mismatch due to the different depth of the languages to be interconnected: from the Presentation view perspective, the potency 1 of the Component view is irrelevant

and therefore skipped.

Listing 9 contains an excerpt of the definition and instantiation of the Presentation view in MetaDepth syntax. Lines 1-16 define the `PresentationView` model with potency 1. The view imports the `ComponentView` model, which has potency 2, in line 1. The `Page` node in lines 8-14 declares a collection of `Component` clabjects of potency 0, which is specified after the symbol "@" in line 11. The listing also shows part of one instantiation of the Presentation view in lines 18-27. The instance model defines the appearance of the simple web application shown in Figure 1, which is made of six pages although the listing only shows the declaration of one (`p1`). The field `content` of this page (line 21) holds a collection of four indirect instances of `Component` with potency 0: a map, a table storing the locations in the map, a calendar and a table storing the events in the calendar.

```
1  Model PresentationView@1 imports ComponentView {
2    Node Presentation {
3      content      : Page[*];
4      styleHead    : StyleHead;     // head
5      styleTree    : StyleTree;     // tree
6      styleContent : StyleContent;  // content
7    }
8    Node Page {
9      name    : String;
10     parent  : Page [0..1];
11     content : Component@0[*];
12     style   : StyleContent;
13     layout  : Layout;
14   }
15   ...
16 }
17
18 PresentationView UAMPresentation imports UAMComponents {
19   Page p1 {
20     name    = "Welcome page";
21     content = [map, mapTable, calendar, calendarTable];
22   }
23   ...
24   Presentation presen {
25     content = [p1,p2,p3,p4,p5,p6];
26   }
27 }
```

Listing 9: Reference to deep clabject.

5.3. Leap semantics for potency

As discussed in Section 5.1, in deep meta-modelling, it is only possible to create an element at potency `n` if the meta-level at potency `n+1` contains its clabject type. This is in line with the strict meta-modelling approach [25], where each element is the instance of exactly one element in the upper meta-level. However, we are sometimes interested only in the instances with potency 0. In this case we have to create clabjects at each intermediate meta-level – with a so-called *identity* instantiation that does not introduce new information – only for the purpose of obtaining suitable types to instantiate clabjects at potency 0.

For example, our Component view must allow connecting any two components at level 0 by means of connectors. For this purpose we should define a clabject `Connector` with potency 2, and then one instance of it for each pair of instances of `Component` with potency 1, since each connector must store a reference to the concrete parameters of the connected component types. In this way, it is possible to connect any two component instances at level 0. However, this solution is hardly usable in practice as, whenever a new component type is added with potency 1, we need to add also new instances of connector with potency 1 to enable the connection of the new component type with all existing ones.

To alleviate this situation, we introduce a semantics for potency named *leap* which is similar to the semantics of potency for fields (see Figure 6(b)), in the sense that instantiation is *not mediated*. Thus, clabjects and references with a leap potency of `n` can be instantiated exactly `n` meta-levels below, but not at intermediate meta-levels. Conceptually, this is equivalent to "skip" the instantiation at the intermediate meta-levels. Technically, it is equivalent to let the meta-modelling framework to create "phantom" instances of the element with leap potency at each intermediate meta-level.

Figure 9 makes use of the leap semantics to define the Component view in the upper part. `Connector` is assigned potency 2, which is indicated after the symbol "@" as usual, but the enclosing parenthesis indicate that the potency has leap semantics and therefore we can instantiate the clabject only at level 0 (2 meta-levels below), saving us from its manual instantiation at level 1. The connector defines deep references to the clabjects `InParam` and `OutParam`. In this way, the instances of `Connector` can be connected to any indirect type of these two clabjects with potency 0, thus solving the problem described before.

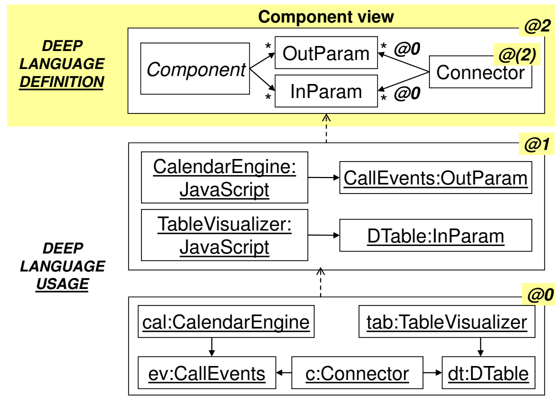


FIGURE 9: Component view: leap potency for clabject

As an example, there are two instances of `JavaScript` with potency 1: `CalendarEngine`, which is used to retrieve calendar events for a given user from the Google Calendar and store the events in a table, and `TableVisualizer`, which displays a table visually. Both component types are instantiated again at level zero, and can be connected through a connector. In this way, the retrieved events from the calendar are shown in a table. Should `Connector` be assigned a normal

potency (instead of *leap*) we would need to instantiate it at level 1.

Listing 10 shows the definition of the previous Component view in METADEPTH. The notation for leap potency is shown in line 10. Each component type has input and output parameters, and the clabject `Connector` in lines 10-13 allows connecting these parameters at level 0.

```
1  Model ComponentView@2 {
2    abstract Node Component {
3      inParams  : InParam[*];
4      outParams : OutParam[*];
5      ...
6    }
7    abstract Node Param { ... }
8    Node InParam : Param { ... }
9    Node OutParam : Param { ... }
10   Node Connector@(2) {
11     from : OutParam@0[*];
12     to   : InParam@0[*];
13   }
14   ...
15 }
```

Listing 10: Leap potency for clabject.

There is a constraint between the leap potency of an element and the deep references it can declare, though. In particular, a clabject `C` with leap potency `n` can only declare a deep reference of potency `k` to a clabject `D` with potency `n+k` (assuming that both clabjects are defined in the same model). This is so because clabject `C` can only be instantiated `n` meta-levels below, where the indirect instances of `D` have potency `k`, as required by the deep reference. In addition, if the potency of `D` is also leap, then `k` must be 0. In the example of Listing 10, the deep references `from` and `to` point to `Component` instances of potency 0. Hence we have `n=2` and `k=0`, which fulfils 2=2+0.

References can also be assigned a potency with leap semantics. For illustrative purposes, Listing 11 shows an example where `Connector` has standard potency 2, and its two owned references have leap potency 2. In this way, the connector needs to be instantiated at level 1, whereas the references can only be instantiated at level 0. As a difference from the previous example, here it is possible to define different types of `Connector` with potency 1, if we want to model different kinds of communication semantics between components.

```
1  Model ComponentView@2 {
2    ...
3    Node Connector {
4      from@(2) : OutParam@0[*];
5      to@(2)   : InParam@0[*];
6    }
7    ...
8  }
```

Listing 11: Leap potency for references.

In summary, we have shown that the different semantics of potency proposed in the literature (clabject-like and field-like, see Figure 6) can be captured either by standard potency or leap potency. Hence, our proposal is the first one to recognise and implement these two kinds of instantiation semantics.

5.4. Leap semantics vs. language fragmentation

As an alternative to the use of leap potency to "skip" meta-levels, one could decide to define `Connector` with potency 1, and include deep references to `Component` with potency 0. However, this would not have the desired effect because `Connector` could only be instantiated in the next meta-level (i.e. at a model with potency 1), but not two meta-levels below, where the indirect instances of `Component` to be referred reside (i.e. at a model with potency 0).

To be able to instantiate connectors only once, but two meta-levels below (i.e. at a model with potency 0), we proposed leap potency in the previous section. Another possibility would be to fragment the language definition by creating a new view for connectors with potency 1, as Figure 10 shows. Hence, a general solution to simulate leap semantics is to move the clabjects with leap semantics to a separate model with potency 1. The moved clabjects would declare the necessary deep references to the clabjects in the original models.

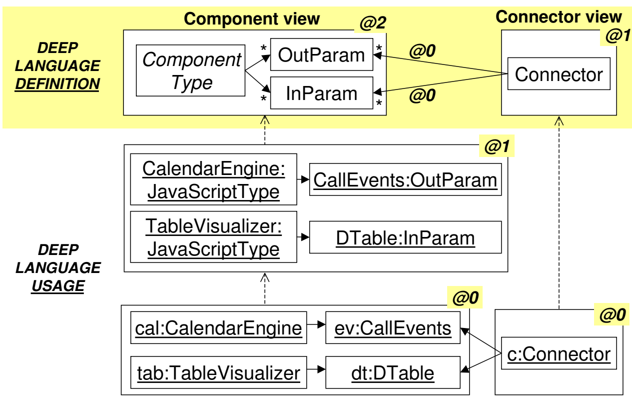


FIGURE 10: Splitting connectors and components

Note however that this solution may sometimes lead to an excessive fragmentation of models, and therefore to unnecessary complexity. This is the situation in Figure 10, where both components and connectors conceptually belong to the same view of the system, but need to be kept in separate models.

Other times this fragmentation is natural, like in the case of our Presentation view, where `Page` has potency 1 and declares a reference of potency 0 to `Component` (see Figure 8). In that case, `Page` and `Component` belong to two different system views and can be naturally kept apart in different models. Should we like to declare pages and components in the same view of the system, then `Page` should be given a leap potency of 2 in the Component view.

6. UNIFORM HANDLING OF INHERITANCE

Clabjects may have both a type facet and an instance facet. Clabjects having a type facet naturally support

inheritance, therefore it is natural to use inheritance at any potency greater than zero because clabjects retain a type facet. In addition, we have also found useful the inheritance between clabjects that are pure instances at potency zero, to enable value inheritance and overriding. This permits handling inheritance at every meta-level uniformly.

6.1. Inheritance between pure types

Inheritance acts as usual for clabjects with a type facet: field types declared in a parent clabject are inherited by its children, which can redefine the initial value of the inherited fields. For example, in lines 3-4 of Listing 1, the `Component` clabject declares the fields `ident` and `name`, which are inherited by the children clabjects `JavaScript` and `Applet`.

A child clabject can declare a potency equal or higher than the potency of its parent, being most common the equality of both potencies. If the child declares a higher potency, then its instantiation depth is bigger than the one declared by the parent, which does not pose any problem. On the contrary, declaring a lower potency in the child is problematic because the substitutability of supertypes by subtypes would get compromised. In such a case, the subtype could not be instantiated as many times as the supertype, which may break the expectations of a client that relies on all subtypes of the supertype to be instantiable in as many meta-levels as the potency of the parent indicates (like, e.g., a deep code generator like the one defined in Listing 4). As a consequence, this situation is not allowed.

Moreover, declaring a clabject with lower potency than its parent has the additional problem that it might prevent the instantiation of some of the fields declared in the parent. Figure 11 illustrates this issue. The left part shows a compatible inheritance relation where clabject `B` has higher potency than clabject `A`. This situation is not problematic because the instance of `B` at potency 1 can assign a value to the inherited field `a`. The right part of the figure shows an incompatible inheritance relation because the clabject `B` reduces the potency of its supertype `A`. As a result, `B` can only be instantiated one meta-level below and hence the `a` field never receives a value.

6.2. Inheritance between pure instances

For fields of potency zero, inheritance is interpreted as value overriding, similar to the case of initial values for field definitions. This is feasible when both the parent and the child clabjects define the same field, in order to enable overriding. Hence, we require the child clabject to have the same ontological type as the parent, or to be a subtype (the next subsection will show that this is indeed required at any meta-level, not only at potency 0). In this way, if the child clabject does not explicitly assign a value to a field with potency 0, it takes the

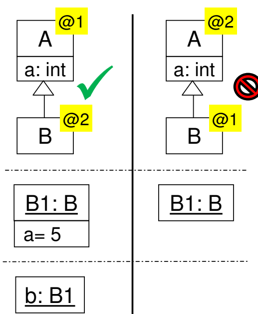


FIGURE 11: Compatible (left) and incompatible potency (right) in inheritance relations

value from the parent. This is useful to enable a kind of prototype-instance [26] way of modelling, where the instances of a prototype clabject can extend and override the value of its properties.

In our example, we have used inheritance of pure instances to reuse predefined presentation styles when modelling new web applications, as Figure 12 shows. In particular, we have built a repository of presentation models with potency 0, which declare predefined styles for the header, navigation tree and content of the applications. The figure shows the definition of one of such predefined styles, named `Predef_ORANGE`, which renders elements in orange colour. The style has potency 0 as it is an instance of the Presentation view model with potency 1. If a new application wants to use the predefined style, its Presentation view (named `UAMPresentation` in the figure) only has to inherit from it to get all the style values. In this way, the presentation view of the new user application does not have to give value to the fields taking care of the style unless it wants to override them – the values are inherited from the predefined presentation – but only to the fields dealing with the content.

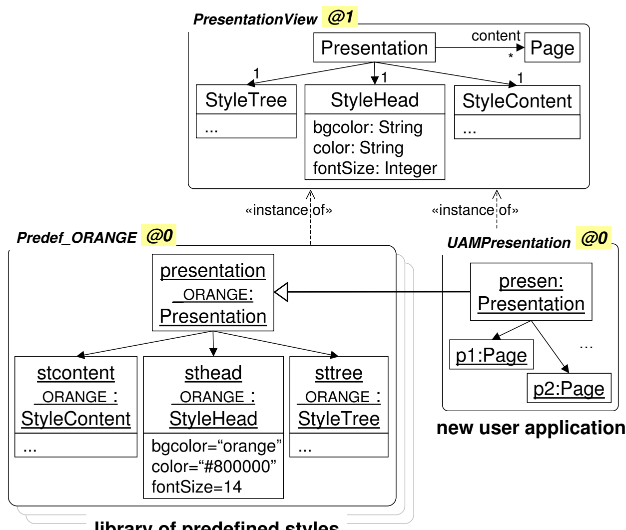


FIGURE 12: Presentation view: using inheritance at potency 0 to reuse predefined presentations

Listing 12 shows the realization in MetaDepth. Lines 1-14 define the predefined presentation style. Lines 16-24 create the presentation view of a particular application that reuses the predefined style. The way of using the predefined style is by importing its model (`Predef_ORANGE` in line 17), and creating a presentation object that inherits from the presentation object in the imported model (line 20). Changing the style of an application amounts to importing and inheriting from a different predefined presentation. The defined presentation `presen` can override values inherited from the parent, like for example it is done in line 21, where the `fontSize` is changed to 12. In addition to inheriting and overriding values, the created clabject `presen` points to the actual content of the application, as done in line 22.

```
1  PresentationView Predef_ORANGE { // Predefined style  definition
2    StyleHead sthead_ORANGE {
3      bgcolor   = "orange";
4      color     = "#800000";
5      fontSize  = 14;
6    }
7    StyleTree     sttree_ORANGE  { ... }
8    StyleContent stcontent_ORANGE { ... }
9    Presentation  presentation_ORANGE {
10     styleHead     = sthead_ORANGE;
11     styleTree     = sttree_ORANGE;
12     styleContent  = stcontent_ORANGE;
13   }
14 }
15
16 PresentationView UAMPresentation
17   imports UAMComponents, Predef_ORANGE {
18   Page p1 {  ...  }
19   Page p2 {  ...  }
20   Presentation  presen  :  presentation_ORANGE { // Using style
21     fontSize  = 12;
22     content  = [p1,p2];
23   }
24 }
```

Listing 12: Using inheritance at potency 0.

Another way to support predefined styles without resorting to inheritance would have been to define two different languages to model the presentation view: one to define the content and layout of pages, and another one for the style. The instances of the former would store a reference to a (predefined) instance of the latter, and changing the style would consist on updating the reference to a different style. In general, each set of features that we may want to make reusable should be declared in a separate clabject, and then use references instead of inheritance. Our approach based on inheritance of instances is more flexible, yields more cohesive language definitions and fewer clabjects.

6.3. Inheritance at intermediate meta-levels

At intermediate meta-levels, clabjects exhibit both type and instance facets. In this case, inheritance works as overriding for fields with potency 0, whereas the rest of fields are inherited from parents to children. Moreover, the type of the clabjects in an inheritance relation must be compatible, i.e. a clabject `B` can only inherit from `A` if both have the same (direct) type, or if the type of `B` is a subtype of the type of `A`. In the remaining of this section we discuss why other cases are problematic, and therefore they are not allowed.

The first forbidden situation, shown in Figure 13, is the definition of an inheritance relation between two clabjects `A1` and `B1`, if their types are different and there is no inheritance relation between the types. The rationale is that this may break possible assumptions on design decisions taken at the top-level model. For example, allowing the inheritance depicted in the figure would break the expectations from clients of such clabjects, as they would expect that any direct or indirect instance of `B` is not an instance of `A`. Hence, if the designer of the top-level model adds a constraint with potency 2 that iterates on all `B` instances, he does not expect finding indirect instances of `A` at the bottom-level model. As we will see later, access to fields declared with potency 2 can also become problematic. Similar constraints were also posed in [27] in order to define a *refinement* relation between elements at different meta-levels.

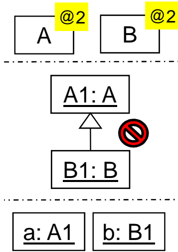


FIGURE 13: Incompatible inheritance relation

Figure 14 explains the problem using Venn diagrams. In Figure 14(a), the set `A` represents all possible instances of clabject `A`, and similar for set `B`. Both sets are disjoint because no clabject at potency 1 can be an instance of `A` and `B` simultaneously. Two elements of the sets (the `A1` and `B1` clabjects at potency 1) are explicitly represented with dots. This reflects the instance facet of `A1` and `B1`, which are direct instances of `A` and `B`. However, `A1` and `A2` have also a type facet and hence can be seen as sets themselves, as Figure 14(b) shows. Instances of `A1` (resp. `B1`) are indirect instances of `A` (resp. `B`) and therefore we have the set inclusions shown in the figure.

Figure 14(c) shows the sets of possible instances of clabjects `A1` and `B1` at potency 1, taking now into account the inheritance relation between `B1` and `A1`. The inheritance relation requires all instances of `B1` to be also instances of `A1` (therefore the set inclusion $B1 \subseteq A1$), but there may be instances of `A1` (like `a`) that are not instances of `B1`. At potency 0, any instance of `B1` (e.g. `b` in Figure 13) is an indirect instance of both `B` and `A` (the latter due to the inheritance relation at potency 1). In order to represent such indirect instances, we have to merge or *flatten* the diagrams in Figures 14(a) and (c). This flattening cannot be done satisfactorily, preserving the constraints imposed by the

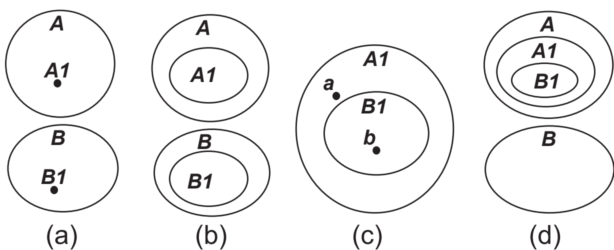


FIGURE 14: Set-based visualization of subtyping for Figure 13. (a) Direct instance sets of unrelated clabjects. (b) Including instances of `A1` and `B1` as (indirect) instances of `A` and `B`. (c) Declaring `A1` as supertype of `B1`. (d) Impossible flattening: Set `B1` cannot belong to both `A` and `B`

two diagrams. An attempt to such flattening is shown in Figure 14(d). As shown in the figure, any instance of `B1` is also an instance of `A1`, and transitively of `A`. Hence, the set `A` includes all direct and indirect instances of `A`. However, `B1` instances cannot be also indirect instances of `B`, because sets `A` and `B` are demanded to be disjoint (see Figure 14(a)), and therefore no element can belong to both.

Please note that, in MetaDepth, an instance of an instance of `A` is an indirect instance of `A`, and we treat direct and indirect instances uniformly. In the example, evaluating the expression `A.allInstances()` at potency 1 yields all direct instances of `A`, while the same expression evaluated at potency 0 yields all indirect instances of `A`.

A second forbidden situation is illustrated in Figure 15. In this case, `A1` is not allowed to inherit from `B1` at potency 1 because `A` is a supertype of `B` at potency 2 (i.e. the subtyping relation at the top meta-level is reversed in the instances at the intermediate meta-level). As before, we forbid this situation to meet the expectations from clients of the language [27].

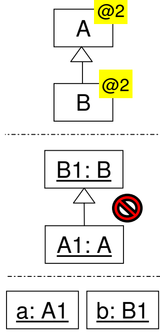


FIGURE 15: Incompatible inheritance relation

The fact that `B` is a subtype of `A` in this example can be seen as subset inclusion, as shown in Figure 16(a). The figure shows two instances `A1` and `B1` of clabjects `A` and `B`, respectively. `A1` is not an instance of `B`, and therefore is not included in set `B`. Figure 16(b) shows the sets of instances of `A1` and `B1`, once the former is declared a subtype of the latter at potency 1. At potency 0, one may ask for the indirect instances of `A` and `B`, for which we need to *flatten* the sets in Figures 16(a) and (b). Hence, in Figure 16(c) the set `A` contains the direct and indirect instances of `A`, and similarly for set `B`. The flattening in Figure 16(c) reveals a contradiction as it requires all `A1` instances to be indirect instances of `B` (there is a subset inclusion $A1 \subseteq B$), while `A1` itself is not an instance of `B` (see Figure 16(a)). Although `A1` should belong to set `B` because all its instances are also `B` instances (see Figure 16(b)), it does not. Thus, the problem comes from the fact that the property "not being an instance of `B`" is not preserved for the instances of `A` (like `A1`) due to the incompatible inheritance at potency 1.

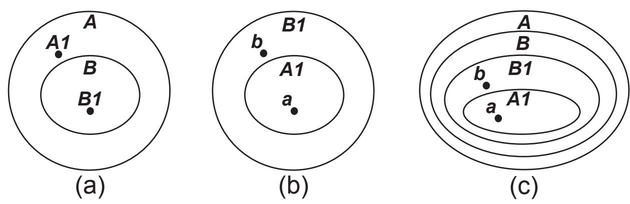


FIGURE 16: Set-based visualization of subtyping for Figure 15. (a) `A` is a supertype of `B`. (b) `B1` is a supertype of `A1`. (c) Flattening of (a) and (b) yields a contradiction

An additional problem of allowing reversing inheritance relations is related to field access, as shown in Figure 17. The figure shows the declaration of an attribute `at` of potency 2 in clabject `B`, and a constraint with potency 2 that checks that some indirect instance of `B` has a value bigger than zero. At level 1, the left model includes a compatible inheritance, whereas the right model shows an incompatible one because clabject `a` at level 0 does not have a field `at`, even though it is an indirect instance of `B`. This is so as, in level 1, `A1` inherits the fields with type facet declared in `B1`, but not `at` which was declared at level 2.

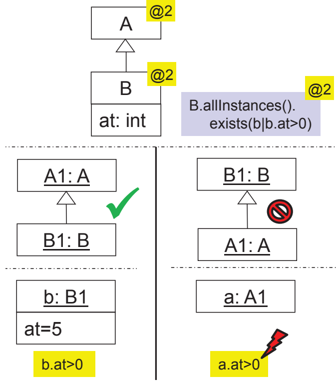


FIGURE 17: Field access issues when using inheritance at intermediate meta-levels

On the contrary, there is no problem if the inheritance relation is compatible, as shown in Figure 18. In this case, `B1` is allowed to inherit from `A1` because `A` is a supertype of `B`.

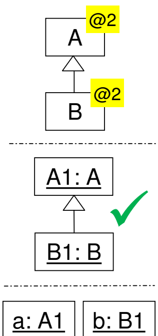


FIGURE 18: Compatible inheritance relation

Figure 19 shows the set-based representation of the example: the sets of instances of clabjects `A` and `B` in Figure 19(a), the sets of instances of clabjects `A1` and `B1` in Figure 19(b), and the flattening of all these sets in Figure 19(c). In particular, the flattening shows that the defined inheritance relation does not lead to a contradiction, as `b` is a direct instance of `B1` and an indirect instance of `B` and `A`.

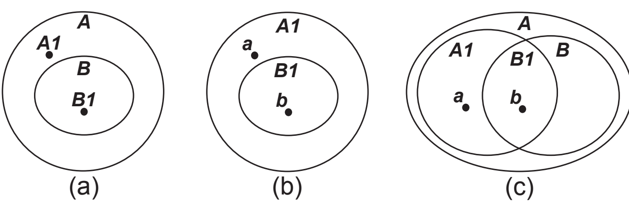


FIGURE 19: Set-based visualization of subtyping for Figure 18. (a) `A` is a supertype of `B`. (b) `A1` is a supertype of `B1`. (c) Compatible flattening of (a) and (b)

Coming back to the example, our language for components makes use of inheritance at intermediate meta-levels, as the Figure 20 shows. In particular, the Google suite of gadgets contains several visualizers for JavaScript data tables, like PieCharts, Tables and GeoMaps. Hence, our library of components at potency 1 contains instances of the `JavaScript` component type for the different visualizers, and we have defined a parent abstract clabject `TableVisualizer` for all of them with the definition of their common features. Thus, `TableVisualizer` declares one input parameter of type `DTable`, which is inherited by the children components. The clabject is abstract so that it cannot be instantiated. Note how clabjects at potency 1 can be extended linguistically with new fields, like e.g. the field `allowHTML` in clabject `Table`.

Listing 13 shows the METADEPTH definition of this example. The Component view (with potency 2) is declared in lines 1-10, and an instance of this view

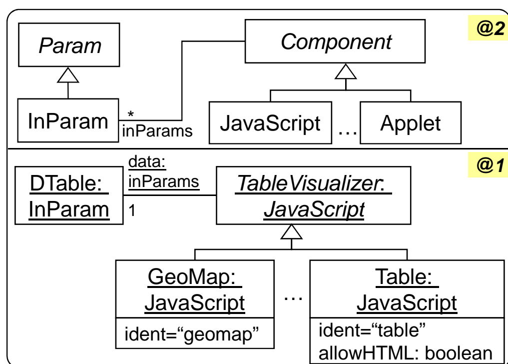


FIGURE 20: Component view: inheritance at potency 1 to encapsulate Google visualizers for tables

named `RepositoryComponents` is created in lines 12-24. The repository declares `TableVisualizer` as an instance of `JavaScript` in line 13. Two clabjects, corresponding to two concrete visualizers, inherit from `TableVisualizer` in lines 17 and 20.

```
1  Model ComponentView@2 {
2    abstract Node Component {
3      inParams : InParam[*];
4      ...
5    }
6    Node JavaScript : Component { ... }
7    Node Applet : Component { ... }
8    Node InParam { ... }
9    ...
10 }
11
12 ComponentView RepositoryComponents {
13   abstract JavaScript TableVisualizer {
14     data : DTable[1] {inParams};
15     ...
16   }
17   JavaScript GeoMap : TableVisualizer {
18     ident = "geomap";
19   }
20   JavaScript Table : TableVisualizer {
21     ident = "table";
22     allowHTML : boolean;
23   }
24 }
```

Listing 13: Using inheritance at potency 1.

6.4. Simulating deep references with inheritance

Once we have analysed the semantics of inheritance at different meta-levels, this section discusses the uses of inheritance as an alternative solution to some of the problems we addressed in Section 5.

In some situations, one could use inheritance to replace deep references by a reference to a base clabject from which all other clabjects inherit, as Figure 21 shows. Hence, instead of declaring a deep reference from `Page` to `Component` with potency 0 (see part (a) of the figure), another solution is to declare a normal reference to an instance of `Component`, from which all instances of `Component` at potency 1 should inherit. However, this solution is not adequate in our context for several reasons. First, it requires an extra clabject `BaseComponent` at potency 1, solely for the purpose

of being the base class of all `Component` instances at potency 1. Second, this solution works because of a convention, as it requires all `Components` with potency 1 to inherit from `BaseComponent`. If some `Component` instance does not inherit from it, then it cannot be included in a `Page` instance. Finally, it introduces a dependency between all `Components` at potency 1 and `BaseComponent`.

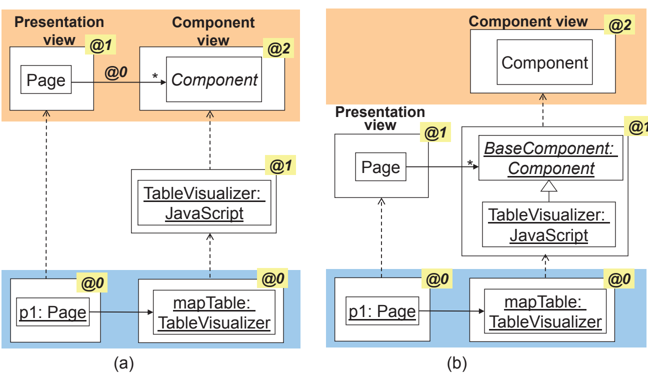


FIGURE 21: (a) Connecting languages with heterogeneous depth using deep references, (b) Replacing deep references by inheritance to common parent

Our solution using deep references does not work by a convention, but by construction, and is simpler as it does not need to introduce an artificial instance of `Component`, together with the inheritance relationship to all `Components` of potency 1.

7. DISCUSSION AND LESSONS LEARNT

In this section we discuss the advantages and limitations of having used a deep meta-modelling approach for the example project, present a summary of the proposed extensions and discuss additional findings in our experience with multi-level meta-modelling for MDE.

7.1. Benefits of using a multi-level approach

Our running example presents several challenges that we found difficult to overcome using a two-level meta-modelling approach. First, our Component view needs to define both component types and instances. The former declare features that must be allocated by their instances. Using only two meta-levels, types and instances have to be defined in the same meta-level, yielding more complex system descriptions, as shown in Figure 2. Moreover, when modelling a particular application, either the designer has to manually instantiate the appropriate features and ports for each component instance according to its type and ensure their correctness – which makes the construction of models tricky – or the DSL developer has to provide ad-hoc mechanisms for instantiation – which makes the development of DSLs difficult. On the contrary, with deep languages the framework itself provides an automatic instantiation mechanism for component types (i.e. component instances are created with all necessary slots) and includes type checking for these, leading to simpler descriptions as shown in Figure 3.

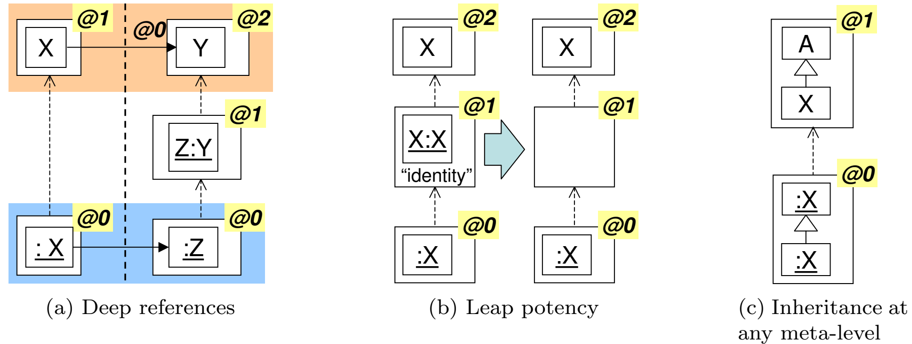
The separation of component types and instances in two different meta-levels has the additional advantage that it cleanly separates specification from usage, tasks that are usually carried out by different people. Thus, technology experts encapsulate reusable components as models with potency 1, together with code generation EGL templates for each component. Then, domain experts select and connect instances of these components to build concrete applications (see Figure 1). While in two-level approaches both types of users work at the same meta-level, in deep languages the technology expert effectively performs domain specific meta-modelling. Our separation in different layers for each role automates the construction of customized environments in a natural way

We obtained other benefits by using a multi-level approach in our scenario. For example, in the Data view, we define data schemas and instantiate them with actual data, which permits type checking of the data. Moreover, our code generator produces JavaScript code that populates the appropriate data structures. Working directly in JavaScript one would populate the tables directly without type checking. Although this double instantiation can be emulated in two meta-levels, we would find the same difficulties as in the Component view. Therefore, multi-level meta-modelling enabled a simpler modelling of system aspects that contained the type-object pattern.

7.2. Lessons learnt

While deep meta-modelling presents advantages when modelling parts of a system that can be naturally expressed in three meta-levels (like the Component and the Data views), when we started to build the deep languages for the running example we realized that additional mechanisms were necessary to obtain simpler descriptions. Deep references (Figure 22(a)) permit establishing connections to clabjects whose direct type is unknown, and they are useful to solve heterogeneities in meta-level depth when models of different potency are related. The leap semantics of potency (Figure 22(b)) is a shortcut to avoid *identity* instantiation at intermediate meta-levels. It is especially useful to define connectors able to link any two indirect instances of `Component` at potency 0. Finally, we have proposed the uniform handling of inheritance at every meta-level (Figure 22(c)). In particular, the inheritance between pure instances at potency 0 permits the definition of libraries with predefined clabject configurations.

In addition to these mechanisms, during our work with deep languages we have also discovered some design patterns for deep meta-modelling. For instance,



(a) Deep references (b) Leap potency (c) Inheritance at any meta-level

FIGURE 22: Novel techniques for deep meta-modelling

the "stratify potency" pattern organizes the fields in a hierarchy of clabjects according to their potency. It can be used when the definition of a deep language includes a hierarchy of clabjects, and one of the children clabjects in the hierarchy requires the potency of a field defined in the parent to be different. For example, in our language for components, a Flash component defines at potency 1 all relevant information needed for its use, therefore it is redundant to instantiate it at potency 0 as this does not add new information. Thus, while `JavaScript` and `Applet` have potency 2, `Flash` should have potency 1. However, all these clabjects share some characteristics. In this case, our pattern suggests breaking the `Component` clabject in a superclass with potency 1 and a subclass with potency 2. While `Flash` inherits from the superclass, both `JavaScript` and `Applet` inherit from the subclass. The result of applying a refactorization towards this pattern is shown in Figure 23. In the figure, `Flash` gives a default initial value to attributes `gencodetemplate` and `gencodetemplatecanvas` (common to all instances unless they explicitly give a value to these fields) and defines a number of additional fields with potency 1.

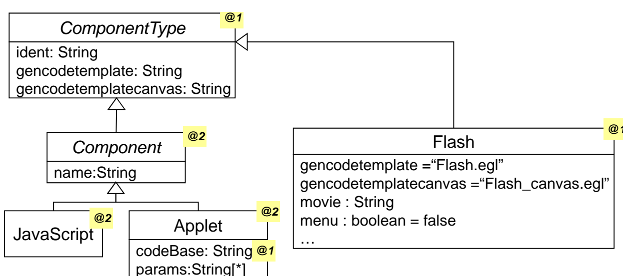


FIGURE 23: Applying the "stratify potency" pattern to accommodate the `Flash` component

As we have seen, deep meta-modelling offers a more flexible solution for defining complex languages that include both types and instances. We are not claiming that two-level approaches cannot deal with them, but would require to manually add ad-hoc functionality to standard two-level meta-modelling frameworks to emulate a type system within a single meta-level. This functionality is built-in in MetaDepth and available across different meta-levels. However, a multi-level framework introduces new problems, most notably concerning the connection of models at different potencies and the heterogeneity in the depth of clabjects with same type. Here we have presented some techniques to solve them.

8. RELATED WORK

In this section we compare our proposals with existing works in the literature. First, we review multi-level systems without deep characterization. Then, we review systems supporting deep characterization. Finally, we analyse works with respect to each extension we proposed in our framework.

Multi-level meta-modelling systems. Multi-level meta-modelling can be traced back to the eighties in knowledge-based systems like Telos [28] and deductive object base managers like ConceptBase [29]. ConceptBase implements the object model of a Datalog-based variant of Telos. It supports instantiation chains of arbitrary length and definition of (Event-Condition-Action) rules and integrity constraints, but not deep characterisation (i.e. the ability to influence meta-levels below the immediate one). Moreover, being from the database tradition, it lacks integration with model manipulation languages enabling its use in MDE, like template languages for code generation, and does not provide advanced modelling mechanisms as the new ones we have proposed in Sections 5 and 6.

A more recent approach to multi-level meta-modelling is the use of powertypes [20]. The instances of powertypes are also subtypes of another type, hence having both type and object facets. Again, this approach does not consider deep characterization either.

The VPM framework [27] formalizes multi-level meta-modelling by a refinement relation. Entities are viewed both as sets (a type that defines the set of its instances) and elements of a set (an instance in the set of instances defined by its type). Thus, as in multi-level

meta-modelling, an element retains both class and instance facets. In VPM, an entity `A` refines another entity `B` if $A.id \in B.set$ and $A.set \subseteq B.set$, where $A.id$ denotes the entity `A` viewed as an element and $A.set$ denotes `A` viewed as a set. This is close to our discussion in Section 6.3, where we treat clabjects as both elements and sets. While VPM is realized in the VIATRA tool, it lacks deep characterization and does not consider attributes or constraints.

Deep characterization. There are some frameworks based on the seminal ideas of Atkinson and Kühne [6] regarding multi-level meta-modelling, but few of them report on the modelling of complex, real life systems and its use with MDE processes. For example, [10] presents an ongoing effort to build a multi-level framework based on *Ecore*, effectively including all ontological meta-levels in one level. This is similar to [30], where MOF is extended for multiple meta-levels to enable XML-based code generation. Nivel [31] is a deep meta-modelling framework based on the weighted constraint rule language (WCRL). OMME [32] is a prototype that implements advanced modelling constructs like a dual ontological/linguistic typing, deep characterization and powertypes. However, none of these works describe primitives to attack scalability of complex, multi-view systems described with deep languages, as we have done in this paper.

The work in [8, 9] is one of the few attempts to use a multi-level framework in practice. The framework is used in the context of test beds for industrial automation systems. While [9] presents a technique to efficiently navigate between several meta-levels, in [8] the authors propose a non-strict meta-modelling approach where the instantiation relation is always allowed to jump arbitrarily between meta-levels. In the present paper we have provided a number of systematic techniques to apply deep meta-modelling in practice and, in particular, our leap semantics for potency can solve some of the problems observed in [8], but adhering to strict meta-modelling. Another difference is that we retain the ability to require an explicit instantiation at every meta-level (standard potency) or allow phantom instantiation (leap potency). Finally, the examples in [8] consider one single model, but do not tackle the problems that arise when connecting languages with arbitrary depth.

Semantics of potency. One of the contributions of this paper is the realization of the different possible semantics for potency: clabject-like or field-like (leap potency). To the best of our knowledge, this difference is not realized in previous works. For example, DeepJava [11] extends the Java language with deep characterization through potency, which can be added to attributes, methods and classes. However, the semantics of potency for attributes with non-primitive type is unclear (either leap or clabject-like semantics). In [24], the authors explain the need to have the same structure to model associations (connectors) and their instances, but they do not discuss the semantics of potency attached to these. Instead, connectors are modelled with clabjects, hence receiving a clabject-like instantiation. None of these frameworks offer the two kinds of semantics.

Deep references. Similar to our deep references, DeepJava introduces abstract type declarations [11]. Variables of an abstract type may contain indirect instances of any potency. In contrast, our deep references can specify the potency required for these instances, providing more control when defining a deep language. This is especially useful to interconnect languages with different depth, where it is usual to specify the potency required from some external clabject. For instance, in Listing 9, we need to express the requirement that the `Component` instances should have potency 0. Using DeepJava's abstract types would prevent from specifying such a constraint, allowing the connection with `Component` instances of potency 1 as well.

Inheritance and instantiation. We are not aware of any previous work discussing the uniform treatment of inheritance at any meta-level, nor any practical modelling framework supporting inheritance at potency zero. The proposal of merging classes and objects is present in systems like SELF [26] and its prototype-instance mechanism. Just like VPM, SELF unifies the inheritance and instantiation relations, while their differences are stressed in [33]. This latter work proposes *abstract objects* and a mechanism to apply inheritance at the instance level. An abstract object is a *representative* of a set of instances which are identified through an equivalence relation. While that work gives different semantics of inheritance at the instance and type levels, here we provide a uniform semantics to inheritance at every meta-level. In our case, a parent (pure) instance clabject can be considered as the *representative* of the set of all clabjects that inherit from it. Hence, the field values of the parent clabject effectively define the equivalence relation. In [33] this proposal is given theoretically, but here we provide a working implementation with further mechanisms like a transparent overriding of attribute values defined in the parent by children clabjects.

The GME modelling environment [34] supports the definition of templates, or prototypes, which can be cloned to enable reuse. Modifying one of these templates produces a modification of its clones as well. This is related to our inheritance between pure instances, where parent clabjects act like prototypes for children clabjects. In [35], the authors propose using this cloning mechanism to build libraries of model elements. However, note that cloning is

different from instantiation: while cloning creates a structural copy of a template, a model is not structurally equal to its meta-model because meta-models allow variability in the instantiated models, like the number of references of a given type. For this purpose, the authors of [35] allow the inclusion of scripts to parameterize certain aspects of the clone, like the number of references of a given type. Still, the proposed prototype-cloning mechanism does not emulate fundamental aspects of instantiation. For example, meta-models declare attribute types and models provide values to the attributes; and meta-models may include integrity constraints which are evaluated in the models. Including such aspects in a cloning mechanism would lead to the construction of a type system emulating two meta-levels in only one, which is the strategy followed in [10] to implement a multi-level system in EMF.

Domain-specific meta-modelling. The idea of domain-specific meta-modelling was already suggested in our previous work [12] and by some other authors using a two-level approach [13]. In the present paper, we have provided richer modelling support for domain-specific meta-modelling. We believe that a multi-level meta-modelling framework is more suitable for this purpose, as it does not require transforming models into meta-models to enable instantiation at deeper meta-levels.

Altogether, in this paper we have proposed novel meta-modelling mechanisms enabling the use of deep meta-modelling with MDE processes in complex scenarios. On the conceptual side, the contributions of the paper are a characterization of two types of semantics for potency (clabject-like and field-like or leap), deep references to enable the connection of model elements at different potencies, and a uniform treatment of inheritance at every meta-level. On the practical side, we have implemented these concepts in the MetaDepth tool, augmented it with a template language for code generation, and used the system in a complex project consisting on the automatic generation of web applications from models.

9. CONCLUSIONS AND FUTURE WORK

In this paper we have demonstrated the practical use of deep languages and MetaDepth in a project that applies MDE to the development of collaborative web applications. In this scenario, where the type-object pattern appears, the literature already reported advantages of the use of deep meta-modelling regarding facilities for instantiation. However, we have found that for this meta-modelling paradigm to be usable in practice, some extra features are needed, like references to deep clabjects, inheritance at every meta-level and *phantom* instantiations through a leap, field-like semantics for potency. Moreover we have proposed a design pattern to control the meta depth of clabjects, and shown some issues when integrating a template language for code generation. In our view, the project presents challenges that would have required considerable effort using a two-level framework. However, for other domains and scenarios, a two-level description is enough. Actually, our meta-modelling system supports two levels regarding linguistic instantiation, but an arbitrary number of ontological meta-levels.

As for future work, we would like to extend the domain-specific meta-modelling capabilities of MetaDepth with richer constructs. For example, we would like to investigate less restrictive uses of inheritance at intermediate meta-levels. We will also tackle the construction of an Eclipse plug-in for the development of MetaDepth models and its use for practical MDE. We are also working on an algebraic formalization of multi-level meta-modelling and deep characterization [36].

Regarding the project, the languages we have proposed use concepts of the solution space (component, page, presentation). We are considering building different DSLs for different kinds of collaborative applications so that these can be described from the problem domain point of view. Then, we would need to transform these models into the languages we have presented, and for this purpose we would need to integrate a model-to-model transformation language within MetaDepth. We are also building a collaborative web tool for the construction of collaborative web applications. The tool is based on questionnaires, from which models are automatically generated and fed into MetaDepth, which produces the final web application [37]. The tool, the presented examples and a sampler of generated collaborative applications are available at: `http://astreo.ii.uam.es/~jlara/metaDepth/Collab.html`.

ACKNOWLEDGEMENTS

This work has been partially sponsored by the Spanish Ministry of Science and Innovation, under project "Go Lite" (TIN2011-24139), as well as by the R&D programme of the Community of Madrid, project "e-Madrid" (S2009/TIC-1650). The authors are grateful to the anonymous reviewers that helped in improving the paper.

REFERENCES

[1] Völter, M. and Stahl, T. (2006) *Model-driven software development.* Wiley, England.

[2] Kelly, S. and Tolvanen, J.-P. (2008) *Domain-Specific Modeling: Enabling Full Code Generation.* John Wiley & Sons, Hoboken, New Jersey.

[3] Mellor, S. J., Scott, K., Uhl, A., and Weise, D. (2004) *MDA Distilled.* Addison-Wesley Object Technology Series, USA.

[4] OMG (2009). MOF 2.0. http://www.omg.org/spec/MOF/2.0/.

[5] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008) *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, Upper Saddle River, NJ.

[6] Atkinson, C. and Kühne, T. (2002) Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, **12**, 290–321.

[7] Atkinson, C. and Kühne, T. (2008) Reducing accidental complexity in domain models. *Software and Systems Modeling*, **7**, 345–359.

[8] Aschauer, T., Dauenhauer, G., and Pree, W. (2009) Multi-level modeling for industrial automation systems. *Proceedings of EUROMICRO-SEAA'09*, Patras, Greece, August 27-29, pp. 490–496. IEEE CS.

[9] Aschauer, T., Dauenhauer, G., and Pree, W. (2009) Representation and traversal of large clabject models. *Proceedings of MoDELS'09*, Denver, CO, USA, October 4-9, LNCS, **5795**, pp. 17–31. Springer.

[10] Atkinson, C., Gutheil, M., and Kennel, B. (2009) A flexible infrastructure for multilevel language engineering. *IEEE Trans. Soft. Eng.*, **35**, 742–755.

[11] Kühne, T. and Schreiber, D. (2007) Can programming be liberated from the two-level style? – Multi-level programming with DeepJava. *Proceedings of OOPSLA'07*, Montreal, Quebec, Canada, October 21-25, pp. 229–244. ACM.

[12] de Lara, J. and Guerra, E. (2010) Deep meta-modelling with METADEPTH. *Proceedings of TOOLS'10*, Málaga, Spain, June 28 - July 2, LNCS, **6141**, pp. 1–20. Springer.

[13] Zschaler, S., Kolovos, D. S., Drivalos, N., Paige, R. F., and Rashid, A. (2009) Domain-specific metamodelling languages for software language engineering. *Proceedings of SLE'09*, Denver, CO, USA, October 5-6, LNCS, **5969**, pp. 334–353. Springer.

[14] Epsilon (2009). http://www.eclipse.org/gmt/epsilon/.

[15] Kolovos, D. S., Paige, R. F., and Polack, F. (2006) The Epsilon Object Language (EOL). *Proceedings of ECMDA-FA'06*, Bilbao, Spain, July 10-13, LNCS, **4066**, pp. 128–142. Springer.

[16] Rose, L. M., Paige, R. F., Kolovos, D. S., and Polack, F. (2008) The Epsilon Generation Language. *Proceedings of ECMDA-FA'08*, Berlin, Germany, June 9-13, LNCS, **5095**, pp. 1–16. Springer.

[17] Ellis, C. A., Gibbs, S. J., and Rein, G. L. (1991) Groupware: Some issues and experiences. *Commun. ACM*, **34**, 39–58.

[18] Coleman, D. and Khanna, R. (1995) *Groupware: Technologies and Applications*. Englewood Cliffs NJ: Prentice Hall, Upper Saddle River, NJ.

[19] Bézivin, J. (2005) On the unification power of models. *Software and Systems Modeling*, **4**, 171–188.

[20] González-Pérez, C. and Henderson-Sellers, B. (2006) A powertype-based metamodelling framework. *Software and Systems Modeling*, **5**, 72–90.

[21] Garlan, D., Monroe, R. T., and Wile, D. (2000) ACME: Architectural description of component-based systems. *Foundations of Component-Based Systems*, pp. 47–68. Cambridge University Press, New York, NY, USA.

[22] Atkinson, C. and Kühne, T. (2003) Model-driven development: A metamodeling foundation. *IEEE Software*, **20**, 36–41.

[23] de Lara, J. and Guerra, E. (2012) From types to type requirements: Genericity for model-driven engineering. *Software and Systems Modeling*, **in press**.

[24] Gutheil, M., Kennel, B., and Atkinson, C. (2008) A systematic approach to connectors in a multi-level modeling environment. *Proceedings of MoDELS'08*, Toulouse, France, September 28 - October 3, LNCS, **5301**, pp. 843–857. Springer.

[25] Atkinson, C. (1998) Supporting and applying the UML conceptual framework. *Proceedings of UML'98*, Mulhouse, France, June 3-4, LNCS, **1618**, pp. 21–36. Springer.

[26] Ungar, D. and Smith, R. B. (1987) SELF: The power of simplicity. *Proceedings of OOPSLA'87*, Orlando, Florida, USA, October 4-8, pp. 227–242. ACM.

[27] Varró, D. and Pataricza, A. (2003) VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Software and Systems Modeling*, **2**, 187–210.

[28] Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. (1990) Telos: Representing knowledge about information systems. *ACM Trans. Inf. Syst.*, **8**, 325–362.

[29] Jarke, M., Gallersdörfer, R., Jeusfeld, M. A., and Staudt, M. (1995) ConceptBase - a deductive object base for meta data management. *J. Intell. Inf. Syst.*, **4**, 167–192.

[30] Gitzel, R., Ott, I., and Schader, M. (2007) Ontological extension to the MOF metamodel as a basis for code generation. *Comput. J.*, **50**, 93–115.

[31] Asikainen, T. and Männistö, T. (2009) Nivel: a metamodelling language with a formal semantics. *Software and Systems Modeling*, **8**, 521–549.

[32] Volz, B. and Jablonski, S. (2010) Towards an open meta modeling environment. *Proceedings of DSM'10*, Reno, Nevada, USA, 17-18 October, pp. 1–6. ACM.

[33] Kühne, T. (2009) Contrasting classification with generalisation. *Proceedings of APCCM'09*, Wellington, New Zealand, January 20-23, CRPIT, **96**, pp. 71–78. Australian Computer Society.

[34] Karsai, G., Maroti, M., Ledeczi, A., Gray, J., and Sztipanovits, J. (2004) Composition and cloning in modeling and meta-modeling. *Control Systems Technology, IEEE Transactions on*, **12**, 263 – 278.

[35] Herrmannsdörfer, M. and Hummel, B. (2010) Library concepts for model reuse. *Electron. Notes Theor. Comput. Sci.*, **253**, 121–134.

[36] Rossini, A., de Lara, J., Guerra, E., Rutle, A., and Lamo, Y. (2011) A graph transformation based semantics for deep metamodelling. *Proceedings of AGTIVE'11*, Budapest, Hungary, October 4-7, LNCS, **7233**. Springer.

[37] Cobos, R., Martín, R., Moreno-Llorena, J., Guerra, E., and de Lara, J. (2011) REUSES: Questionnaire-driven design for the automatic generation of web-based collaborative applications. *Proceedings of CollaborateCom'11*, Orlando, FL, USA, 15-18 October, pp. 9–18. IEEE.