Automated Analysis of Integrity Constraints in Multi-level Models

Esther Guerra*, Juan de Lara

Computer Science Department
Universidad Autónoma de Madrid (Spain)

Abstract

Multi-level modelling is a technology for model-based development that enables the incremental refinement of models in successive meta-levels, which results in simpler and more intentional system descriptions in some scenarios. In this approach, integrity constraints can be placed at any meta-level, and need to indicate the meta-level below at which they should hold. This requires a careful design, as constraints defined at different meta-levels may interact in unexpected ways. Unfortunately, current techniques for the analysis of the satisfiability of constraints are designed for two meta-levels only. Hence, nowadays, the analysis of multi-level solutions is performed by hand, which is tedious and error-prone.

In this paper, we define an automated procedure to check the satisfiability of integrity constraints in a multi-level setting, leveraging on "*off-the-shelf*" model finders. This procedure is supported by our multi-level modelling tool MetaDepth, which has been extended to reason on the satisfiability of constraints in multi-level models, and to perform automated model completion.

Keywords: Multi-level modelling, Deep meta-modelling, Conceptual modelling, Management of integrity constraints, Meta-level flattening, Model finders, Constraint solving, MetaDepth

1. Introduction

Modelling is at the core of every model-based development approach [11]. In this context, multi-level modelling [7] is a promising technology which promotes a flexible way of modelling by allowing the use of multiple meta-levels at a time, instead of just two, as it is customary in mainstream modelling architectures nowadays [44]. This extra flexibility results in simpler models [9, 20], typically

*Corresponding author: Esther Guerra, Department of Computer Science, Escuela Politécnica Superior, Universidad Autónoma de Madrid, Ciudad Universitaria de Cantoblanco, Calle Francisco Tomás y Valiente 11, 28049 Madrid, Spain.

Email addresses: Esther.Guerra@uam.es (Esther Guerra), Juan.deLara@uam.es (Juan de Lara)

in scenarios where the type-object pattern [33] or some of its variants arise. Recent works [20] show that this pattern appears frequently in the software architecture and enterprise/process modelling domains. Furthermore, multi-level modelling makes available native meta-modelling facilities at each meta-level, like type definition, type inheritance, or feature definition. This enables the creation of domain-specific modelling languages through successive refinements in different meta-levels, the reuse of language definitions, as well as the dynamic addition of new types, relations, attributes and constraints at any meta-level in a straightforward way [20].

While multi-level modelling has benefits, it also poses some challenges that need to be addressed in order to foster a wider adoption of this technology [18]. One of these challenges is the definition and analysis of constraints in multi-level models. In a two-level setting, constraints are placed in the meta-models and evaluated in the models one meta-level below. This enables the use of "off-the-shelf" model finders [1, 13, 23, 31, 43, 45] to reason about correctness properties, like satisfiability (*is there a valid instance model that satisfies all constraints?*). However, constraints in multi-level models can be placed at any meta-level and be evaluated several meta-levels below, which may cause unanticipated effects. This makes the design and reasoning on the validity of constraints more intricate.

In this paper, we present a systematic method for the analysis of a basic quality property in multi-level modelling: the satisfiability of integrity constraints. Our method relies on the use of "off-the-shelf" model finders which are able to perform a bounded search of models conforming to a given meta-model and satisfying a set of OCL constraints. Since the state-of-the-art model finders only work in a two-level setting, the method needs to "flatten" the multiple levels in a multi-level model to be able to use the finders for our purposes. This process has two orthogonal dimensions, which account for the number of meta-levels provided to, and searched by, the finder. In this paper, we discuss alternative flattening algorithms for different analysis scenarios involving combinations of these two dimensions. Our analysis method is directed to level-agnostic multi-level modelling approaches, supporting a notion of potency or level, as e.g. advocated in [8] and supported by tools like MetaDepth [17] or Melanee [5].

This work extends a previous workshop paper [25] with the following contributions: it provides a more rigorous and comprehensive discussion of the flattening alternatives for each analysis scenario; it expands the related works; it generalizes the satisfiability checking procedure for an arbitrary number of meta-levels (instead of just 3); and it demonstrates the feasibility of the proposal through a working implementation atop the MetaDepth [17] multi-level tool and the USE Validator model finder [31]. This implementation enables the analysis of the satisfiability of multi-level models and the dynamic completion of model instances satisfying a number of integrity constraints.

The remainder of the paper is organised as follows. First, Section 2 introduces multi-level modelling using a running example, and Section 3 presents properties and scenarios in the analysis of multi-level models. Next, Sections 4 and 5 discuss strategies for flattening multi-level models for their analysis with standard model finders. Then, Section 6 describes the integration of a model

finder into MetaDepth to analyse multi-level models. Finally, Section 7 reviews related works, and Section 8 draws some conclusions and lines of future work. The paper includes an appendix with the general algorithm for flattening multi-level models.

2. Multi-level modelling

Multi-level modelling was introduced by Atkinson and Kühne [8] as a way to overcome the rigidity imposed by the dominant praxis in meta-modelling, where only two meta-levels are considered at a time: one defining types and the other defining their instances[1]. This is problematic, especially when the type-object pattern comes into play [20]. This pattern arises when there is the need to define new types and their features dynamically at the instance level (e.g., new kinds of tasks, like a coding task with a start day), as well as creating instances of these types (e.g., a particular coding task starting the 11^{th} of May). Since instances in a two-level meta-modelling architecture do not exhibit properties of types, like the possibility of being instantiated or define attributes, these meta-modelling facilities must be explicitly modelled at the type-level, adding accidental complexity to the definition of a language or problem.

Multi-level modelling overcomes this problem by enabling the use of an arbitrary number of meta-levels, and providing a dual type/instance facet to the elements within a model, so that they are instances with respect to the meta-level above, and types with respect to the meta-level below. For this reason, they are sometimes called *clabjects*, from the union of the words *cla*ss and o*bject* [8].

As an example, Fig. 1 shows the multi-level definition and usage of a very simple language for domain-specific process modelling. The upper model, which corresponds to the language definition, contains the clabject TaskKind representing types of tasks. This generic language can be refined at subsequent meta-levels to create domain-specific process modelling languages. For example, the intermediate meta-level in the figure specializes the language for software engineering process models; hence, the TaskKind clabject is instantiated to create the new task kinds Coding and Testing. These elements are clabjects, being both types (they can be instantiated to create specific tasks, like cd1 and cd2) and instances (of TaskKind, and thus they can assign a value to its feature name).

Since clabjects in the intermediate level have a type facet, they can define new features (e.g., final in clabject SoftwareEngineeringTask) and inheritance relationships. Technically, this is possible by adhering to an orthogonal classification architecture (OCA) [8] which distinguishes two orthogonal typings for model elements: *ontological* and *linguistic*. The ontological typing of an element expresses the instantiation within a domain. For example, the ontological type of Coding is TaskKind, and the one of nextPhase is next. The linguistic typing refers

[1] Although the OMG defines a four-layer meta-modelling architecture [36], only two consecutive layers are considered at a time, in the sense that a layer only influences the one immediately below.

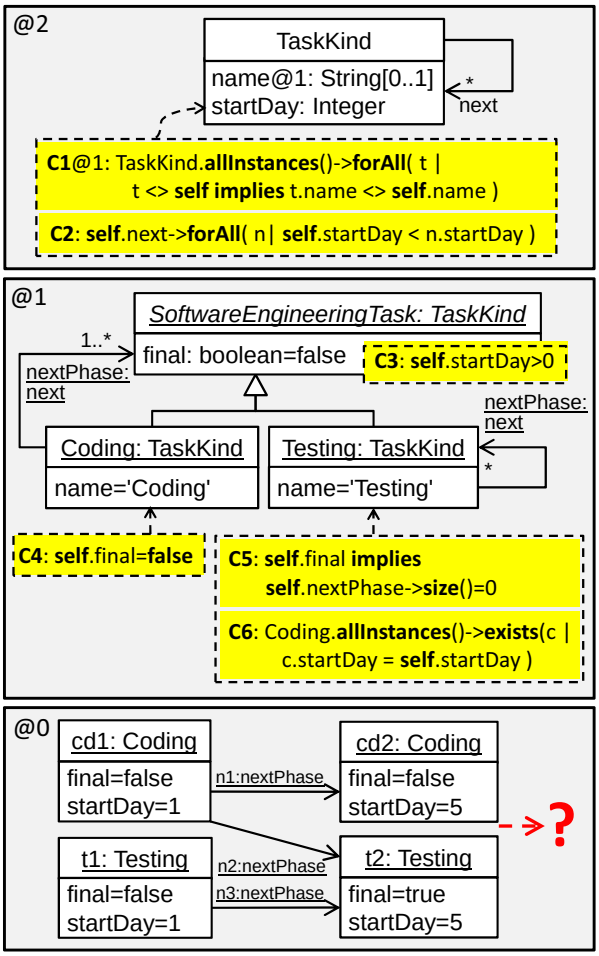

Figure 1: Running example: simplified domain-specific process modelling using multi-level modelling.

to the concept of the meta-modelling language used to define an element. For example, the linguistic type of Coding is Clabject, whereas the linguistic type of final is Field. All model elements have a linguistic type, whereas they may lack an ontological one (e.g., field final has no ontological type). The elements without ontological type are called *linguistic extensions* in [17].

Fig. 2 shows the OCA applied to Fig. 1. The simplified linguistic meta-model to the left pertains to a multi-level modelling architecture. It reflects the fact that most modelling elements (models, clabjects and fields) have a dual type/instance facet. The multi-level model to the right, which is an instance of the linguistic meta-model, shows the ontological typing of elements explicitly using the ontInstanceOf relation. The linguistic meta-model, also called pan-level model in other approaches [29], enables uniform modelling at every meta-level.

This is why this modelling approach is sometimes called *level-agnostic* [4].

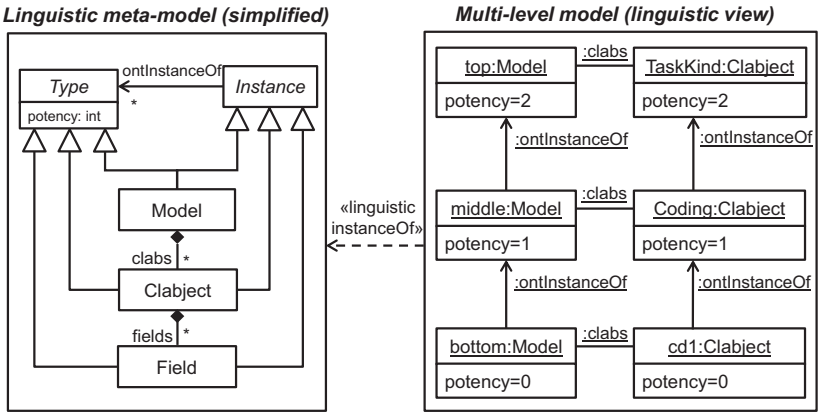

Figure 2: Dual typing in the OCA: simplified linguistic view of the multi-level model in Fig. 1.

References have the same type/instance duality as clabjects. In the example, nextPhase is an instance of next, as well as the type of link n1 in the lowest meta-level. In this paper, we assume that the instantiation of references is mediated. This means that it is necessary to create a reference at level 1 (like nextPhase) to be able to create an instance of next at level 0 (like n1). Moreover, the cardinality of references constrains the number of instances at the meta-level right below, but not lower meta-levels. Thus, the cardinality of next controls the instantiations at level 1, and the cardinality of nextPhase the ones at level 0.

As multi-level modelling spans several meta-levels, it is useful to control the instantiation depth of elements beyond the immediate meta-level below. This can be achieved using the notion of *potency* [8], which is a positive number (or zero) that can be attached to models, clabjects, fields, references and constraints. The potency specifies in how many meta-levels an element can be instantiated. It gets automatically decremented at each deeper meta-level, and when it reaches zero, the element cannot be instantiated in lower meta-levels. If the potency is assigned to a field, then this can be assigned a value only in the deepest meta-level allowed for the field. In the example, this is useful to characterize the instances of instances of TaskKind, for which we know that they have a startDay. If an element does not define a potency, it receives the potency from its enclosing container, and ultimately from the model. The potency of a model is similar to the notion of *level* in some multi-level approaches [7], i.e., it defines the relative meta-level at which the elements within a model reside. In this paper, we use *model potency* and *level* interchangeably, and specifically assume a level-agnostic, potency-based multi-level modelling approach.

In Fig. 1, the potency of elements is indicated with the @ symbol. In this way, the upper model has potency 2, which is also the potency of all the elements it contains except those defining a potency explicitly. The clabject TaskKind has potency 2, which allows the creation of types of tasks in the next meta-

level (e.g., Coding), and their subsequent instantiation into concrete tasks in the bottom meta-level (e.g., cd1). TaskKind defines two fields: name has potency 1 and therefore it receives values in the intermediate level, while startDay has potency 2 and is used to set the start day of specific tasks in the lowest meta-level.

Constraints can be declared at any meta-level. They are OCL [35] boolean expressions defined in the context of a model or clabject, expressing additional requirements for its instances. The potency of a constraint states how many meta-levels below the constraint will be evaluated. In Fig. 1, clabject TaskKind defines two constraints named C1 and C2. Constraint C1, which ensures uniqueness of task kind names, has potency 1; hence, it will be evaluated one meta-level below, ensuring that all direct instances of TaskKind have a different name. Constraint C2 has potency 2, and hence it states that two meta-levels below, the start day of a task must be less than the start day of any task related to it by next references. C2 needs to refer to the instances of the instances of the reference next, two levels below, but the (direct) type of the instances two levels below is unknown beforehand because it depends on the elements created at level 1. In the example, it means that C2 cannot make use of nextPhase to constrain the models at level 0. In prior work [25], we solved this problem by using reflective operations within the constraint. In this paper, we provide a more natural solution: the constraint uses the reference next, but when this is evaluated at level 0, it is interpreted as the union of the instances of next at level 1 (in general, of the instances defined one meta-level above the level where the constraint is evaluated). For example, when self.next is evaluated on clabject cd1, it will consider the reference nextPhase that the type of cd1 declares. Should Coding have declared another instance of next, say named nextOther, then cd1.next would be evaluated to cd1.nextPhase–>union(cd1.nextOther).

Similarly, a constraint with potency 2 may use types defined at its same level, like TaskKind.allInstances(). When evaluated at level 0, this constraint will consider the set of instances of instances of TaskKind ({cd1, cd2, t1, t2} in the example).

Constraints can also be defined in intermediate levels. In the example, constraints C3, C4, C5 and C6 are defined at level 1, and thus they need to be satisfied by models at the subsequent meta-level 0. The purpose of C3 is to enforce positive starting days, where note that the feature startDay is defined one meta-level above. C4 ensures that Coding tasks are not final, while C5 requires final Testing tasks to have no subsequent tasks. Finally, C6 is an attempt to enable some degree of parallelization of tasks, where the modeller wanted to express that any coding task should have a testing task starting the same day.

The lower meta-level in Fig. 1 is an attempt (with no success) to instantiate the model with potency 1. The modeller started adding the coding tasks cd1 and cd2 at days 1 and 5. Then, to satisfy the constraint C6, he added two testing tasks starting at days 1 and 5 as well. Coding tasks must be followed by some other task to avoid they are left untested (controlled by the cardinality 1..* of nextPhase), and the start day of consecutive tasks must be increasing (controlled by constraint C2). Thus, the modeller connected the tasks as shown in the figure to satisfy these constraints. However, then, he realised that the coding task cd2 needed to be followed by some other task. Connecting cd2 with t2 is not a valid

solution because this would violate constraint C2, which demands increasing start days for any two connected tasks.

The next section explains how model finders can help developers during the modelling phase to solve scenarios like the one described, as well as language designers when refining language definitions at higher meta-levels.

3. Analysis of multi-level models: properties and scenarios

A meta-model should satisfy some basic correctness properties, like the possibility of creating a non-empty instance model that does not violate the meta-model integrity constraints. Several works [14, 31, 43] use model finding techniques to check correctness properties of meta-models in a two meta-level setting, like:

- **Strong satisfiability:** There exists a valid instance model that contains at least one instance of every class and association in the meta-model.
- **Weak satisfiability:** There exists a non-empty valid instance of the meta-model.
- **Liveliness of a class c:** There exists some valid instance of the meta-model that contains at least one instance of class c.

We claim that model finding can be helpful in a multi-level setting as well. If we consider level 0 in Fig. 1, a model finder can help model developers by providing a suitable model completion, or indicating that no such completion exists. At level 1, it can help language designers to check the consistency of the integrity constraints at levels 1 and 2 through the analysis of the above-mentioned correctness properties. At level 2, it can provide example instantiations for levels 1 and 0, to ensure that potencies of the different elements at the top-most level work as expected.

However, existing model finders only work in a two-level setting: they receive a meta-model and produce an instance of it, if it exists within the search bounds. In a multi-level setting, a solver would need to receive several models at different meta-levels, and produce a set of models at different meta-levels as well. This could be emulated by *flattening* operations that:

(a) merge several models at different meta-levels into a single model, which can be input to the finder.

(b) extend models with linguistic concepts to emulate several meta-levels within a single model snapshot, which can be produced by the finder.

In this way, these operations must take into account the number of meta-levels to be used in the analysis (*depth of model*, item (a) in the previous list), as well as the number of meta-levels in the generated snapshot (*height of snapshot*, item (b) of the list). Fig. 3 shows the analysis scenarios that arise in multi-level modelling when combining these two dimensions. The background colour

of the meta-levels indicates whether the analysis uses an existing model of the meta-level (shaded), generates a model of the meta-level (blank), or ignores the meta-level (striped).

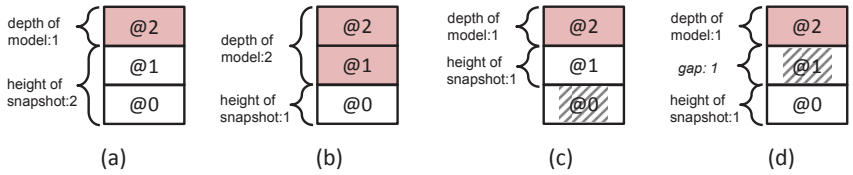

Figure 3: Different scenarios in the analysis of a multi-level model.

In Fig. 3(a), only the definition of the top-most model is available, and we want to check whether this model can be instantiated at each possible meta-level below (2 in the case of having an upper model with potency 2). Thus, in the figure, the depth of the model to be used in the analysis is 1, while the height of the sought snapshot is 2. As standard model finders only provide snapshots of models residing in one meta-level, we need to emulate the generation of models at several meta-levels (1 and 0 in the figure) within one model.

In Fig. 3(b), several models at successive meta-levels are given, and the goal is checking whether there is an instance at the next meta-level (with potency 0) satisfying all integrity constraints in the provided models. This situation arises when there is the need to check the correctness of the constraints in a meta-level (e.g., `@1`) with respect to those in the meta-levels above (e.g., `@2`). Thus, this scenario would help in the analysis of the constraints at levels 2 and 1 in Fig. 1. A particular case of this scenario is model completion, where a fragment of a model at level 0 is given, and the goal is adding the necessary elements to the model to make it a valid instance according to the provided models at higher meta-levels. In Fig. 3(b), the depth of the model to be used in the analysis is 2. Thus, we will need to flatten these two models into a single one, which can be fed into a model finder for standard snapshot generation.

Fig. 3(c) corresponds to the scenario that standard model finders are able to deal with, where a model is given, and its satisfiability is checked by generating an instance of it. However, here the difference is that the generated model contains elements with type facet (e.g., instances of reference next in Fig. 1 have cardinalities, and SoftwareEngineeringTask declares a new field), while model finders normally produce models whose elements have only the instance facet. Moreover, the meta-model to be fed into the solver still needs to be adjusted, removing constraints with potency bigger than 1.

Finally, in Fig. 3(d), only the top-most model is available, and the designer is just interested in the analysis of the lowest meta-level. This can be seen as a particular case of scenario (a), where after the snapshot generation, the intermediate levels are removed. This scenario is of particular interest to verify the existence of instances at the bottom level with certain characteristics, like a given number of objects of a certain type, or to assess whether the designed potencies for attributes work as expected.

Once we have seen the different scenarios, the next two sections describe how to flatten the depth of models to be used in the search, and how to deal with the height of the searched snapshot. Scenarios where both the height and depth are bigger than one are also possible, being resolved by combining the flattenings we present next. The general algorithm for these flattenings, dealing with arbitrary depths and heights, is outlined in the Appendix.

4. *Static* flattening to deal with the depth of the analysed models

To analyse a model in an intermediate meta-level (like in Fig. 3(b), where the goal is analysing level 1), we need to merge it with its type models at all meta-levels above in order to consider all constraints and attributes defined at higher meta-levels. We call this flattening *static* because it "freezes" the models to be analysed and eliminates the instance facet of their elements, i.e., the merged models become fixed and cannot be modified (e.g., the model at level 1 cannot be increased with instances of the model at level 2). We do so as our unique concern for the flattening operation is to ensure that the instances of the resulting flattened model can be seen as instances of the original multi-level model, and vice versa. In other words, the flattening operation should produce a plain (meta-)model that accepts and rejects the same instances as the multi-level model.

For simplicity of presentation, we illustrate the merging of just two meta-levels, while the Appendix shows the general algorithm to deal with the merging of any number of meta-levels.

4.1. The static flattening operation

Next, we describe the flattening of each kind of modelling element separately. Every case is illustrated with figures showing the original levels 2 and 1, and the result of the flattening. The flattened model has potency 1, as it becomes a standard meta-model which can be fed into a model finder. Since the flattening should preserve the set of instances of the original model, we will illustrate the equivalence of the original and flattened models by showing examples of valid and invalid instances at level 0.

Flattening of clabjects. Clabjects at level 2 are set to abstract to prevent their instantiation at level 0. Moreover, the flattening replaces instantiation by inheritance relationships. To avoid redundancies, such inheritance relationships between a clabject at level 1 and its type are only added when the clabject has no parents. The existing inheritance relations between clabjects at the same level are kept.

Fig. 4 illustrates the two flattening cases that we distinguish: (a) clabjects with potency 2 at level 2, and (b) clabjects with potency 1 at level 2. More in detail, Fig. 4(a) shows on the left a multi-level model made of a clabject `A` with potency 2, and an instance of it (`A1`) with potency 1. In its flattened version to the right, `A` becomes abstract and `A1` inherits from it. In the

multi-level model of Fig. 4(b), the clabject A1 at level 1 has potency 0, and therefore, the flattening makes it abstract to avoid its instantiation.

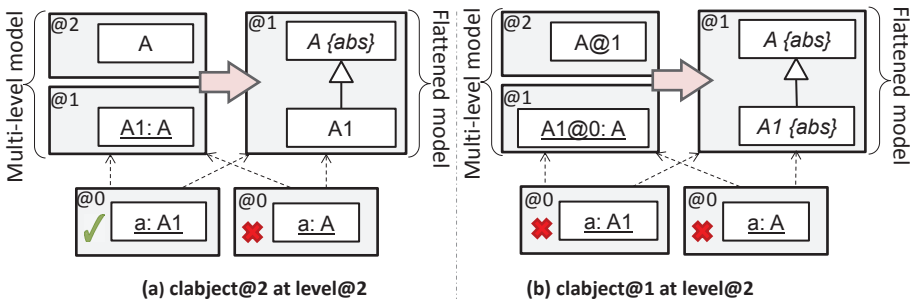

Figure 4: Static flattening of clabjects.

At level 0, both the multi-level models and their flattened versions accept the instances of clabjects created at level 1 with potency 1 (A1 in case (a)), and reject the instances of clabjects with potency 2 at level 2 (A in case (a)), potency 1 at level 2 (A in case (b)), or potency 0 at level 1 (A1 in case (b)). In the figure, valid model instances at level 0 are marked with a ✓symbol, while invalid instances are shown with a cross.

Flattening of attributes. Fig. 5 illustrates the three different cases for flattening attributes, and example models at level 0 that are accepted by both the multi-level models and their flattenings. The attributes with potency 2 defined at level 2 are just copied to the flattened model (see Fig. 5(a)). This permits assigning them a value at level 0. Similarly, the attributes with potency 1 defined at level 1 only need to be copied (see Fig. 5(c)).

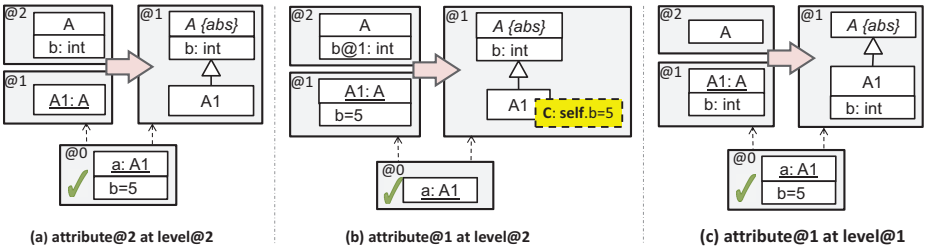

Figure 5: Static flattening of attributes.

The attributes with potency 1 defined at level 2 are also copied to the flattened model, but in addition, their value is emulated by means of a constraint (see Fig. 5(b)). Taking the example of the figure, this approach permits that a query at level 0, like a.b, yields the same result 5 in the multi-level and flattened models. There is a subtle difference though. In the multi-level model, the slot b is not present at level 0, but the query a.b returns the value of the attribute in a's type. In the flattened version, there is a slot for b in

every instance of A1, but the constraint forces all of them to have the value 5. Indeed, this is the desired behaviour if we want to use a constraint solver to instantiate the flattened version of the multi-level model.

Flattening of references. The instantiation semantics of references in multi-level models is as follows. References with potency 2 defined at level 2 can be instantiated at level 1, and in turn, such instances can be instantiated at level 0. Moreover, the cardinality defined by a reference at each level only affects the level immediately below. In Fig. 6(a), reference r with cardinality card at level 2, has the instance r1 with cardinality card1 at level 1. At level 0, there can be instances of reference r1 (as in the left model at level 0, marked with a ✓) but not direct instances of r (as in the right model at level 0, marked with a cross because it is incorrect). Thus, the flattening removes reference r to avoid its instantiation at level 0, and it copies the reference r1 of potency 1 at level 1 so that it can be instantiated one level below.

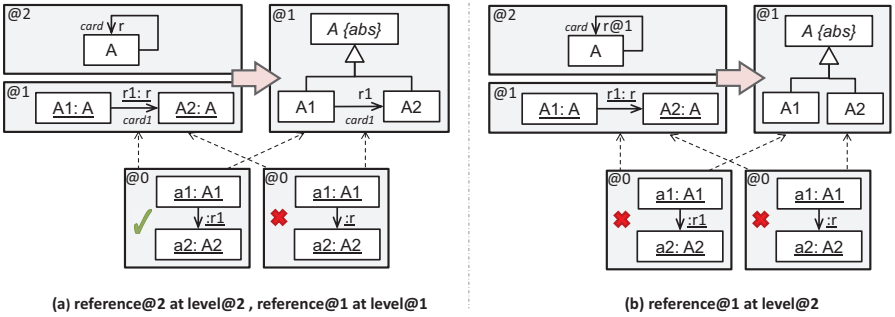

Figure 6: Static flattening of references.

The references of potency 1 at level 2 and their instances at level 1 are removed in the flattening, as they cannot be instantiated at level 0 (see Fig. 6(b)). The bottom of Fig. 6(b) shows two incorrect level 0 models, one attempting to instantiate r1, and the other attempting to instantiate r.

Flattening of constraints. Fig. 7(a) illustrates the flattening of constraints with potency 2 defined at level 2. These constraints are copied to the corresponding class in the flattened model. However, every navigation through a reference with potency 2 (like r) is replaced by an operation call with the same name (e.g., by r()). This is so because, as explained above, the flattening removes all references with potency 2; hence, we need to emulate the removed reference by means of an operation, which is defined in the owner class of the reference (in A). This operation will be overridden in the subclasses to return the union of all instances of the reference the subclasses define. As an example, Fig. 7(a) shows the redefinition of operation r() in A1. In the model at level 0, the navigation a1.r yields {a2, a3} in the multi-level case, while a1.r() yields the same result in the flattened version.

Constraints with potency 1 defined at level 2 are discarded because they

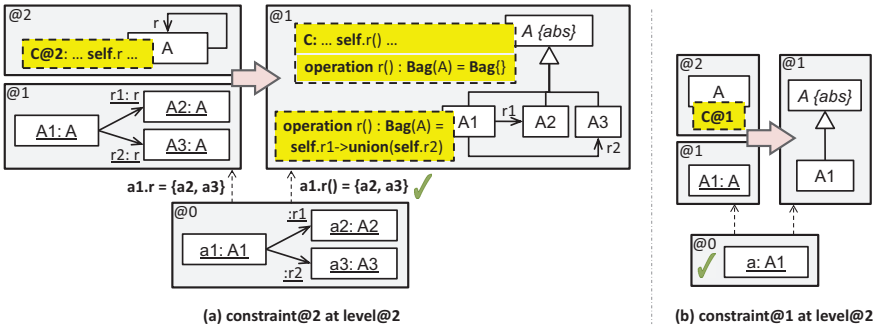

Figure 7: Static flattening of constraints.

do not constrain level 0, as seen in Fig. 7(b). In the figure, constraint C is omitted in the flattened model, because it only affects the model at level 1, which we assume correct. Finally, constraints with potency 1 defined at level 1 are copied without modification, and there is no need to generate operations that emulate references because any reference of potency 1 at level 1 will be copied to the flattened model as well.

While we have used two meta-levels to illustrate the static flattening, the Appendix shows how to generalize this procedure to an arbitrary number of meta-levels. In particular, when merging more than two levels (say m levels, where the potency of the highest model is m), the flattening ensures that only the clabjects with potency 1 at level 1 can be instantiated, as those at higher levels are set to abstract (Fig. 4). The flattening also ensures that, at level 0, the attributes with potency m at level m, with potency m-1 at level m-1, etc. are available (Fig. 5). Attributes with potency p defined at any level n with $p < n$ are also available in clabjects at level 0, but the value they received at level n-p is emulated via constraints. The references with potency 1 at level 1 are copied to the flattened model (Fig. 6), while the rest of references are not copied to disallow their instantiation and their value is emulated with operations if needed. Finally, only the constraints applicable at level 0 are kept (Fig. 7).

4.2. Example

Fig. 8 shows the static flattening applied to the running example. Levels 1 and 2 are merged, as the purpose is creating a regular instance model at level 0, which is accepted both by the multi-level model and the flattened model.

First, the flattening handles the top level. All its clabjects (TaskKind) are set to abstract to disable their instantiation. All references are deleted (next), as only the references at level 1 can be instantiated at level 0. All attributes (name, startDay) are kept. Constraints with potency different from 2 (C1) are deleted as they do not constrain the level we want to instantiate (level 0). Constraints with potency 2 are kept (C2), but any reference navigation expression (next) is substituted by calls to homonym operations that collect the knowledge about the instantiation of the reference. In this example, the generated next operations

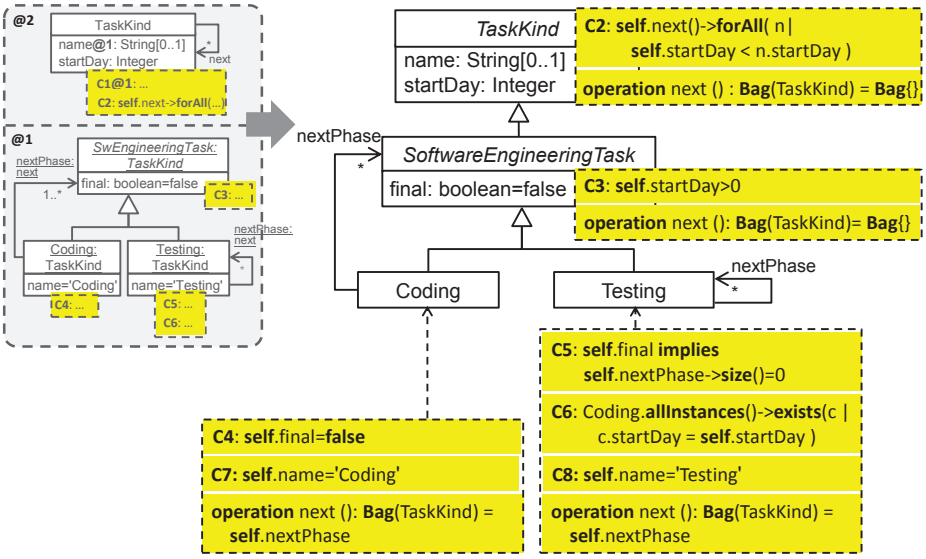

Figure 8: Example of static flattening (for depth 2 and height 1).

encode the fact that nextPhase in Coding and Testing are instances of next, and will be used when evaluating the constraint. The figure does not show the potency of elements, which is interpreted as all elements having potency 1.

Then, the model at level 1 is handled. For clabjects, the instantiation relation is replaced by inheritance. In this way, SoftwareEngineeringTask is set to inherit from TaskKind instead of being an instance of it. This is not required for Coding and Testing, as they already inherit from SoftwareEngineeringTask. This flattening strategy allows clabjects in level 1 to inherit all attributes that were assigned potency 2 at level 2 (startDay), and receive a value at level 0. The value of attributes at level 1 is emulated by constraints. Thus, the slot name in Coding and Testing gets substituted by constraints C7 and C8. All attributes, references and constraints at level 1 are kept. The result of this flattening can be used as input to a model finder to check whether there is a valid instance at level 0.

4.3. Discussion: effects and limitations of the static flattening

From an ontological point of view, replacing inheritance by instantiation does not fully preserve the semantics of the multi-level model. For example, in the multi-level model to the left of Fig. 9, c1 and c2 in the bottom level are not direct instances of TaskKind. However, in the flattened version to the right, c1 and c2 can be seen as instances of TaskKind, as this is a superclass of Coding (their direct type). From a pragmatic point of view, this suffices our purposes as we only need the flattened version of the multi-level model to accept and reject an equivalent set of instances.

In this respect, note that the instances at the bottom level are slightly different in the multi-level and flattened models: in the multi-level model to the

left, the clabjects at level 1 assign a value to the attributes that defined potency 1 (like name), while in the flattened version to its right, the clabjects at level 0 assign these attributes a value which we emulate and fix using constraints. Again, this equivalence of instances is enough for our goals.

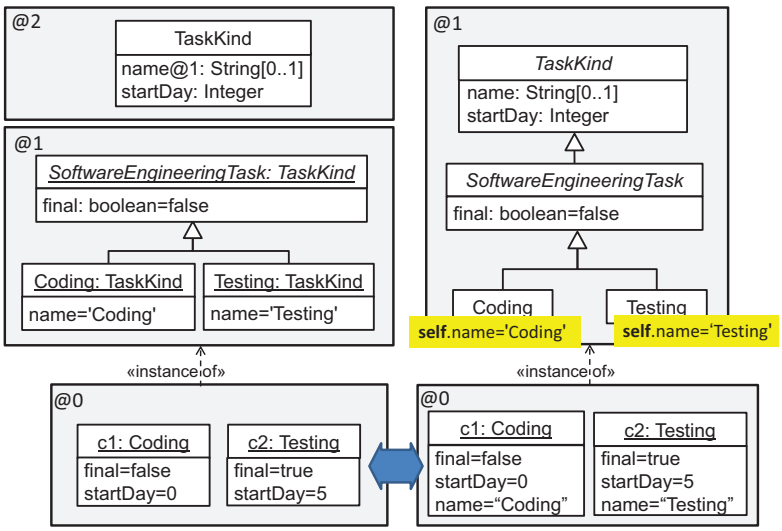

Figure 9: Comparing instances of multi-level and flattened models.

Finally, there may also be *reflective constraints* [19] that navigate to upper ontological meta-levels using meta-modelling facilities, e.g., to obtain the set of ancestors of a clabject. As our flattening does not preserve this ancestors set, our method currently does not support the analysis of this kind of constraints. For example, suppose clabject TaskKind in the multi-level model defines the following constraint with potency 2:

self.type.ancestors→excludes(TaskKind)

The tasks in the bottom-left level of Fig. 9 fulfil this constraint, but the ones in the bottom-right level do not. This is so as, to the right, c1.type is Coding, and Coding.ancestors = Set{SoftwareEngineeringTask, TaskKind}. Emulating the behaviour of reflective operations in the presented flattening is future work. Nonetheless, in our experience, reflective constraints are much less frequent than ontological constraints like the ones we have shown so far, which only access ontological information [19].

5. *Linguistic* flattening to deal with the height of generated snapshots

To analyse whether a model can be consecutively instantiated several meta-levels below (like in Fig. 3(a), where the goal is finding instance models at levels 1 and 0), we need to emulate a set of models spawning several meta-levels within a single one. For this purpose, we use an operation called *linguistic flattening*

that enriches a model with linguistic information, such as the potency of its elements and other artefacts, emulating a multi-level model within a flat one.

For clarity of presentation, we assume the scenario in Fig. 3(a), aimed at generating two models at consecutive levels from a given model at level 2. The algorithm in the Appendix generalises this scenario to any number of levels.

5.1. The linguistic flattening operation

Fig. 10 shows the schema of the linguistic flattening operation. In this case, we want to generate instances of MM at levels 1 and 0. For this purpose, we define MM as an extension of the linguistic meta-model (see Fig. 2), performing a kind of static flattening. The result of merging MM and the linguistic meta-model can then be used as the input to a model finder in order to produce instances at levels 1 and 0 encoded within a single model.

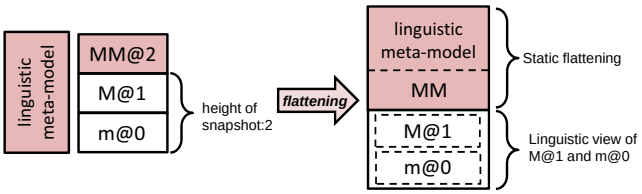

Figure 10: Schema of linguistic flattening.

In the simplest case, the linguistic meta-model only contains a class named Clabject that defines typical clabject features like ontological typing and potency (see Fig. 11(a)). This class is equipped with the following integrity constraints:

C1: This constraint regulates the allowed values for the potency. In Fig. 10, the potency must be in the interval [0..1] because we consider the generation of models at levels 1 and 0. In general, the upper bound for the potency is the potency of the analysed model minus one.

C2: This constraint ensures that the potency of an element is one less than the potency of its type (if it has one), and both the element and its type are ontological instances of the same clabject.

C3: This constraint ensures that the elements with potency 0 have no instances.

This linguistic meta-model does not include explicitly the notion of *Model* because it is possible to deduce the clabjects that belong to the same model given an initial multi-level model.

The flattening operation will merge this linguistic meta-model with the model under analysis, adding the field potency and the relation modelling instantiation to every clabject in the model. This is enough for the purpose of satisfiability checking if the sought models at level 1 do no need to include linguistic extensions, i.e., elements without an ontological type like new fields

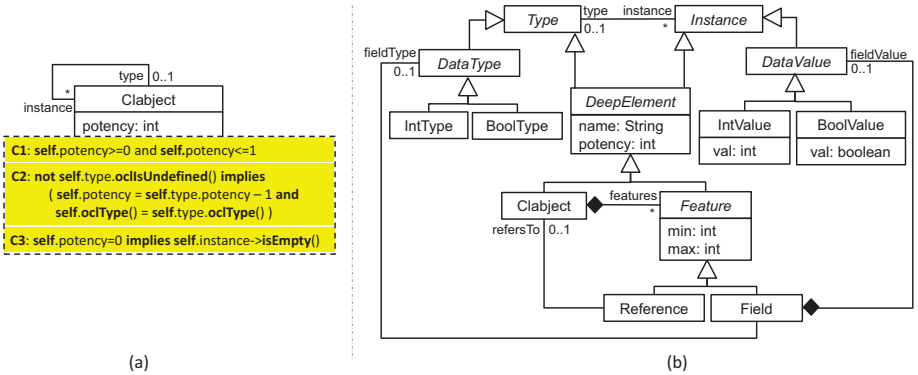

Figure 11: (a) Basic linguistic meta-model. (b) Linguistic meta-model enabling linguistic extensions.

or new clabjects. To support linguistic extensions, one would need to use a more complete linguistic meta-model like the one in Fig. 11(b) (in which OCL integrity constraints are omitted). This meta-model is inspired by the one of MetaDepth [17]. Using this linguistic meta-model, a model finder could produce clabjects with new fields and references by instantiating classes Field and Reference. For simplicity, we will use the simpler meta-model in Fig. 11(a) in the remaining of this section.

Next, we describe the linguistic flattening of each element kind in detail.

Flattening of clabjects. Fig. 12(a) illustrates the flattening of clabjects with potency 2. They are copied to the flattened model, and have a constraint (labelled Ainst) attached to them controlling that their instances at level 0 have an ontological type. In addition, if they have no superclasses, they are set to inherit from class Clabject from the linguistic meta-model, which becomes abstract to avoid its instantiation. At the bottom, the figure shows a multi-level instance to the left, and an equivalent flattened model to the right. This flattened model provides a linguistic view of the multi-level instance, reifying the potency as a field of the objects and the *instance of* relation as a link. In this way, the flattened model can be produced by a standard model finder as the two meta-levels are embedded within one.

The flattening of clabjects with potency 1 is very similar to the flattening of clabjects with potency 2. The only difference is the constraint that gets generated, which in this case forces the instances of the clabject to have potency 0. Fig. 12(b) shows an example. At the bottom, the instance of the flattened model contains an instantiation of A with potency 0. Since A was originally defined at level 2, its direct instances belong to level 1 (i.e., to the level of A minus its potency).

Flattening of attributes. Attributes of potency 2 defined at level 2 are copied to the flattened model, as Fig. 13(a) shows. However, they are set to optional

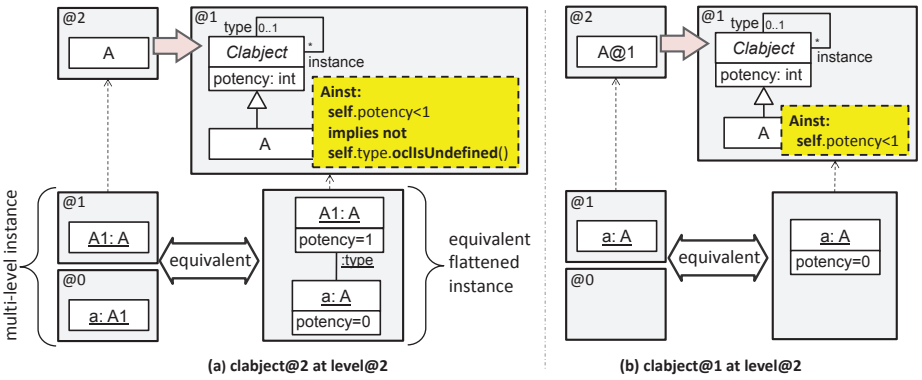

Figure 12: Linguistic flattening of clabjects.

because their value is mandatory at level 0, but optional at level 1. This behaviour is enforced by a constraint (bval) that is added to the owner clabject of the attribute. Fig. 13(a) shows at the bottom a valid multi-level instance and its equivalent flattened model. The flattened model makes explicit the *instance of* relation and the potency, while the object a needs to provide a value for attribute b, emulating the potency 2 of the attribute in the multi-level model.

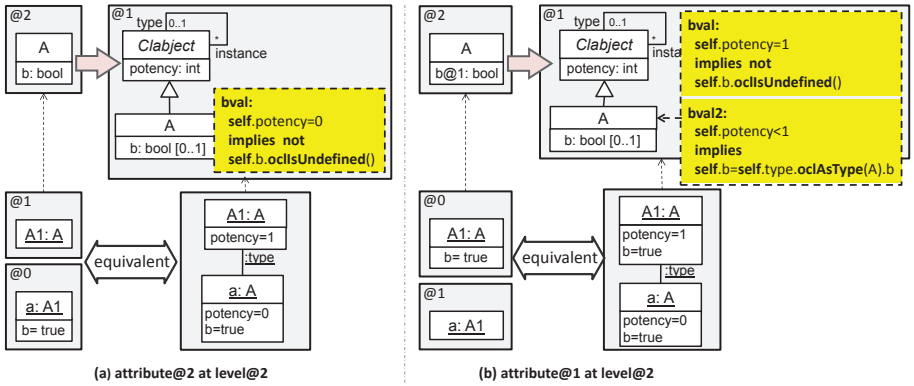

Figure 13: Linguistic flattening of attributes.

Attributes of potency 1 are also copied and set to optional, and a constraint bval is added to enforce that they receive a value in the clabjects with potency 1 (i.e., at level 1). In addition, we add an extra constraint bval2 to enforce that the value of the attribute at level 0 is the same as the value of the attribute at level 1 (see Fig. 13(b)). This is so as the attribute assigned at level 1 is interpreted as a class attribute (in contrast to an object attribute) and hence it defines the same value for all instances of the class. This means that, in the original multi-level model, the query a.b evaluated at level 0

yields true. Constraint bval2 ensures that the same result is obtained when the query is evaluated in the flattened model.

Flattening of references. References with potency 2 can connect clabjects with potency 2 or higher; for simplicity, we will assume they connect clabjects of potency 2. This case is illustrated in Fig. 14(a). The flattening copies the reference r, relaxing its cardinality [m..M] to *, because this reference will be used to create instances at level 1 (where the cardinality constraint applies) and at level 0 (where it does not). The two generated constraints rmin and rmax control the number of instances of the reference at level 1. The first constraint is not generated if the minimum cardinality is 0, while the latter is not generated if the maximum cardinality is * (i.e., the reference is unbounded). The remaining constraint rdef is added to ensure that, given a reference r connecting two clabjects: (a) the clabjects are at the same level (self.potency=n.potency), and (b) either the clabjects are at the top-level (self.type.oclIsUndefined()), or their types are connected by a reference as well.

The figure shows a valid instance of the flattened model that emulates levels 1 and 0. The constraint rdef allows connecting a with b because they have the same potency, and their types (A1 and B1 respectively) are connected. On the contrary, a cannot be connected with B1 because their potency is different. Similarly, rdef allows connecting A1 and B1 because they have the same potency and are at the top-level (1 in this case).

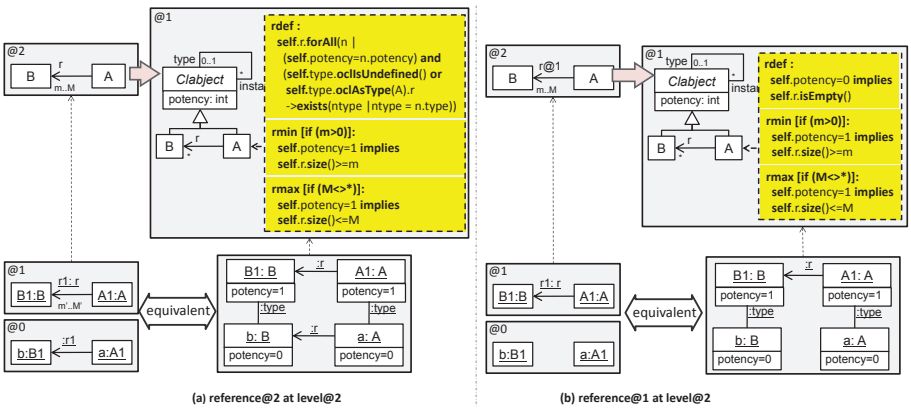

Figure 14: Linguistic flattening of references.

Note that this flattening does not create an explicit *instance of* relation between a reference at level 0 and its type at level 1, and does not emulate the possibility of setting cardinality values in the references at level 1. These aspects would require the use of a more detailed linguistic meta-model like the one in Fig. 11(b), where references are represented as classes. Nonetheless, this simpler flattening we propose is enough for our analysis purposes.

Fig. 14(b) illustrates the flattening of references with potency 1. A reference with potency 1 can connect clabjects with potency 1 or bigger. In the figure,

we show a reference that connects clabjects of potency 2. The flattening copies the reference, relaxing its cardinality to *, and generating three constraints. Constraints rmin and rmax control the number of instances of the reference at level 1, as in the previous case. The remaining constraint rdef forbids connecting clabjects of potency 0, as a reference with potency 1 can be instantiated at the next level (1) but not two levels below (0). In case the potency of the connected clabjects is different from 2, the potencies checked by this constraint should be adjusted accordingly.

The bottom of Fig. 14(b) shows a multi-level instance and the equivalent flattened one. In both cases, connecting a with b is forbidden due to the potency of r in the original multi-level model to the left, and due to the rdef constraint in the flattened instance to the right.

Flattening of constraints. The flattening of constraints with potencies 2 and 1 is similar, as Figs. 15(a) and 15(b) illustrate. In both cases, the constraint is added a precondition to ensure the constraint is checked only if the object has the appropriate potency. Hence, let C be the body of any constraint of potency 2 defined on a clabject with potency 2 at level 2. Then, the constraint is modified in the flattened model to self.potency=0 implies C, reflecting the fact that the constraint should be evaluated two levels below.

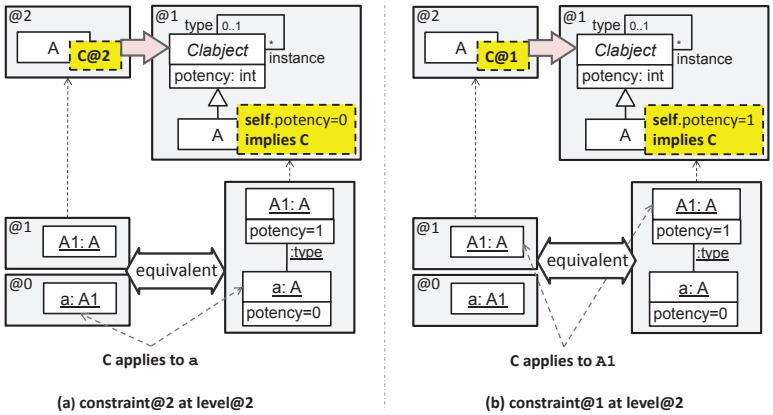

Figure 15: Linguistic flattening of constraints.

Similarly, any constraint C of potency 1 defined on a clabject with potency 2 at level 2 is rewritten to self.potency=1 implies C. In addition, if the constraint includes the operation allInstances() to retrieve all instances of a type, this must be modified to select only the instances with the appropriate potency. In this way, a query like A.allInstances() is replaced by A.allInstances()->select(a | a.potency=0) in constraints with potency 2, and by A.allInstances()->select(a | a.potency=1) in constraints with potency 1.

Although we have explained the linguistic flattening for two meta-levels, the Appendix generalizes this flattening to any number of meta-levels. For example,

to emulate three meta-levels within a single model, we need to ensure that the objects in the model have potencies 2, 1 and 0, and define correct *instance of* relations. This is checked by OCL constraints similar to those in Fig. 12. In general, given a clabject A with potency m, the constraint Ainst shown in Fig. 12(a) can be generalized to: **self**.potency < (m-1) **implies not self**.type.oclIsUndefined(). Similarly, we need to generate constraints like those in Fig. 13 to emulate potencies 2, 1 and 0 for attributes. In particular, given an attribute f with potency n, owned by a clabject A with potency m (where n ≤ m), we generate the following two constraints attached to A:

fval: self.potency = m-n implies not self.f.oclIsUndefined()

fval2: self.potency < m-n implies self.f=self.type.oclAsType(A).f

Constraint fval2 does not need to be generated if m-n=0. The generalization of the flattening for the rest of elements is available in the Appendix.

5.2. Example

Fig. 16 shows the result of applying the linguistic flattening to the top level of the running example. The flattening adds a parent abstract class Clabject that makes explicit the ontological typing and potency, as well as the integrity constraints C9 to C12 to ensure correct potency values and typing for types and instances (cf. Fig. 11(a) and Fig. 12(a)).

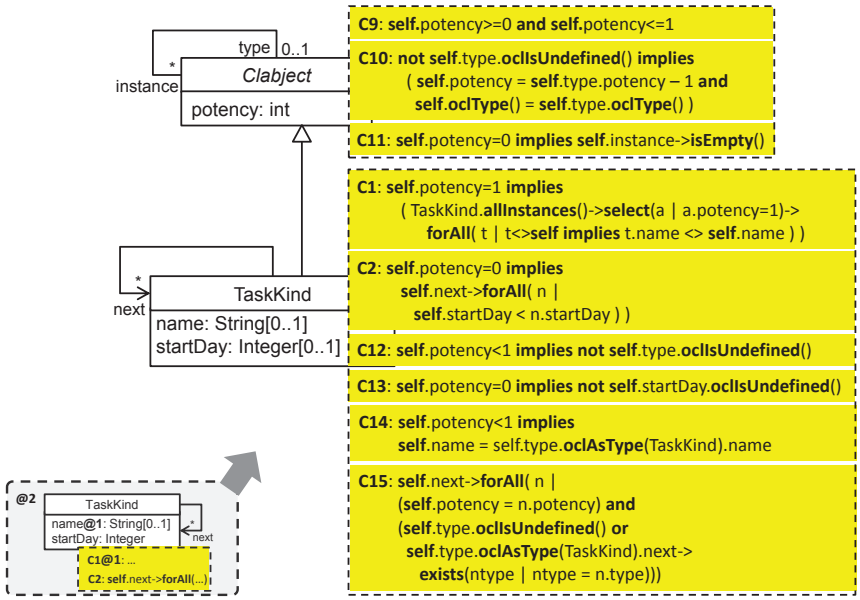

Figure 16: Linguistic flattening for the example (depth 1 and height 2).

All attributes are set to optional (cardinality [0..1]), and we add constraints C13 and C14 to ensure that they receive a value in the appropriate level (cf. Fig. 13). Attribute startDay has originally potency 2, and hence it has to receive

a value in tasks with potency 0 (constraint C13). Attribute name has potency 1, and hence objects at level 0 should have the same value as their types (constraint C14). As the minimum cardinality of name was originally 0, we do not add a constraint demanding a value at potency 1 (cf. constraint bval in Fig. 13(b)).

Regarding references, we add constraints ensuring that they are instantiated in the appropriate levels according to their potency (cf. Fig. 14(a)). In particular, constraint C15 ensures that reference next does not cross meta-levels and is correctly instantiated at every meta-level. The latter means that, if two tasks with potency 0 are related by a next reference, then their types must be related via a next reference as well. As previously stated, this does not fully capture the multi-level instantiation semantics as there is no explicit *instance of* relation between references with different potencies, but it suffices our purposes.

Finally, the constraints in the original model are modified to take into account their potency. Thus, C1 is added the premise self.potency=1 implies... so that it gets applicable only to tasks with potency 1, and similar for constraint C2 for tasks at potency 0. Additionally, the expression TaskKind.allInstances() in C1 needs to be replaced by TaskKind.allInstances()->select(a|a.potency=1) to select only TaskKind instances at level 1.

The resulting flattened model can be used to search valid models at levels 1 and 0 using a model finder. As an example, the left of Fig. 17 shows a snapshot with height 2 generated by the USE Validator from the definition in Fig. 16. The right of the same figure shows the equivalent multi-level model resulting from the "unflattening" of the generated snapshot.

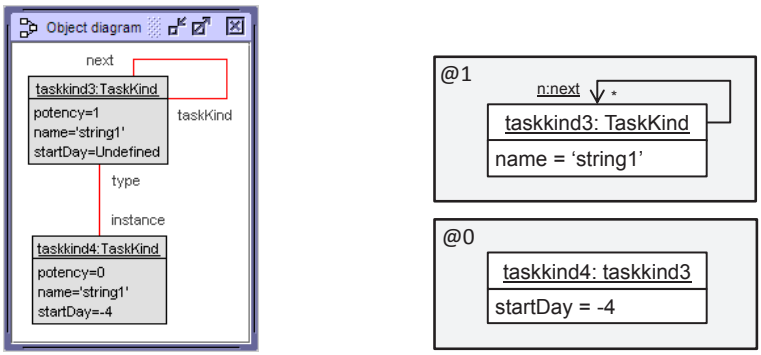

Figure 17: Snapshot with height 2 and explicit potency and instantiation relationships, generated by the USE Validator tool (left). Equivalent multi-level model (right).

5.3. Discussion: alternative flattenings

Next, we discuss the advantages and disadvantages of some flattening alternatives that could be used to deal with the analysis of snapshots with height bigger than 1, and compare those solutions with the one we have proposed.

Type-object flattening. The first alternative consists in making explicit the type/instance facet of clabjects with potency 2. Hence, each clabject C with potency 2 is split into two classes CType and CInstance holding the attributes, references and constraints with potency 1 and 2, respectively, and related by a reference type. For instance, Fig. 18 shows the result of applying this flattening to the running example, where the original clabject TaskKind becomes split in classes C_TaskKind and I_TaskKind. These two classes together can be seen as an instance of the *type-object* design pattern [20, 33] or the *materialization* relation [16]. Similarly, each reference with potency 2 (like next in the example) is split in two. Though a faithful representation of typing would need a class holding the type of a reference, we refrain to do so for comparison with our approach.

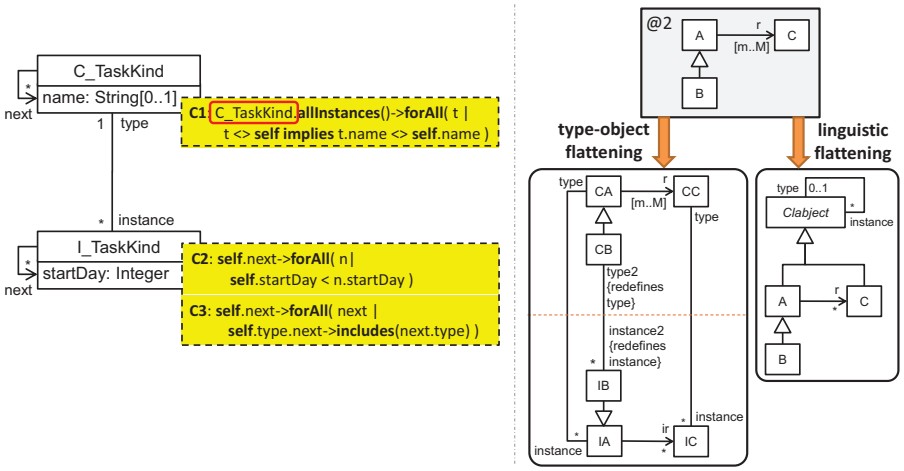

Figure 18: Type-object flattening: clabjects and references with potency 2 are split in two, which makes their type and instance facets explicit.

This flattening is valid and it has the advantage that it adds less constraints to the result than the linguistic flattening. However, it may require rewriting existing constraints in terms of the introduced types and relations (see for instance constraint C1, where type TaskKind was replaced by C_TaskKind). Constraints that consider attributes or references with potencies 1 and 2 in the same expression also need to be rewritten, as we need to navigate the type relation in order to access features with potency 2 from the class with instance facet, and to access features with potency 1 from the class with instance facet. Moreover, this flattening duplicates the number of classes and relations (see right of Fig. 18), or even more if the height of the sought snapshot has more than 2 levels, as this would require splitting each clabject in more than two classes.

Altogether, we have opted for the linguistic flattening because the result is more compact, and it is very easy to configure the linguistic meta-model to accommodate semantic variants or to support the generation of linguistic extensions (i.e., new clabjects and features without an ontological type).

Promotion-based flattening. Another possibility is to proceed in two steps: first, a model at level 1 is generated, and then, this model is promoted into a meta-model that can be instantiated at level 0. A disadvantage of this solution is that it may require rewriting the constraints with potency 2 in terms of the types and relations generated at level 1. Moreover, it does not consider all constraints at a time, which may result in different attempts before two valid models at levels 1 and 0 are obtained. Fig. 19 shows an example of this situation. The model defined at level 2 forbids task instances at level 0 to be disconnected (constraint C1), or be followed by another task of the same type (constraint C2). In the first attempt, the model built at level 1 cannot be instantiated at level 0 (or more precisely, the only valid instance model at level 0 is the empty model) because coding tasks can only be connected to other coding tasks, thus violating constraint C2. Thus, it is necessary a second attempt, where the model at level 1 can be instantiated at level 0.

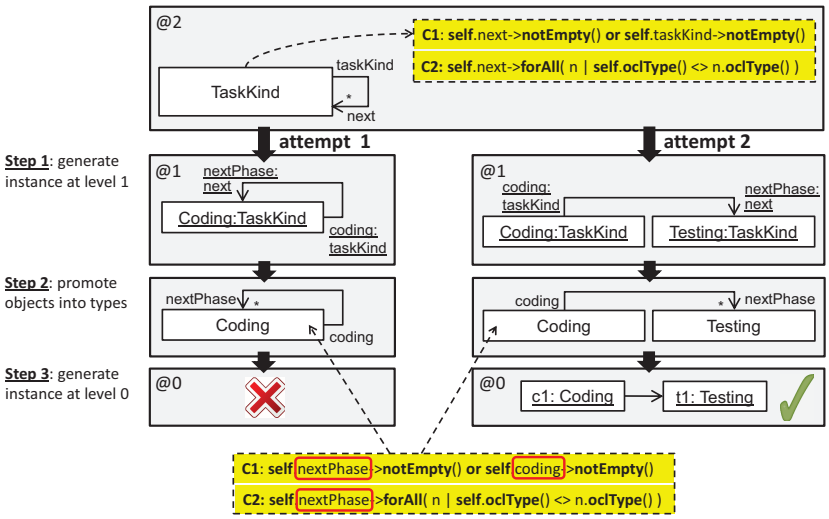

Figure 19: Promotion-based flattening: models at different levels are generated separately in successive steps.

Linguistic embedding. This flattening represents the elements of the model to analyse as instances of a linguistic meta-model. As an example, Fig. 20 shows the top level of the running example as an instance of the linguistic meta-model in Fig. 11(b). Checking for instantiability at levels 1 and 0 is equivalent to completing this model while requiring the existence of clabjects with potency 1 and 0.

Although this approach provides much flexibility, the disadvantage is that it requires a heavy rewriting of the OCL constraints, because ontological types like TaskKind do not exist in the linguistic meta-model, and features are explicitly modelled. For instance, constraint C1 in Fig. 1 should be rewritten as shown in Listing 1.

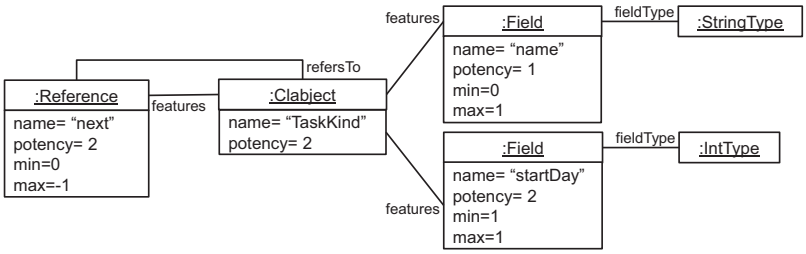

Figure 20: Linguistic embedding: the model elements are represented as instances of a linguistic meta-model.

```
1 Clabject.allInstances()->forAll( c | c.name = 'TaskKind' implies -- sets the context to TaskKind
2     c.instance->forAll ( _self | c.instance->forAll ( t | t <> _self implies
3             t.oclAsType(Clabject).features->select( f | f.name='name')->asSequence()->first().
4                                 oclAsType(Field).value.oclAsType(StringValue).val -- emulates t.name
5             <>
6             _self.oclAsType(Clabject).features->select( f | f.name='name')->asSequence()->first().
7                                 oclAsType(Field).value.oclAsType(StringValue).val -- emulates self.name
8     ) ) )
```

Listing 1: Rewriting an ontological constraint into a linguistic constraint.

6. Automated analysis of constraints in MetaDepth

This section presents the support of our analysis method by the MetaDepth tool. MetaDepth [17] is a potency-based multi-level modelling tool developed by our research group. It provides a simple textual syntax to define multi-level models, which can be exported into PlantUML[2] for their graphical visualization. In addition, it is possible to query and manipulate MetaDepth models using the Epsilon languages [37] to specify constraints, in-place model transformations, model-to-model transformations and code generators.

As an example, Listings 2 and 3 show the definition of the two upper meta-levels in Fig. 1, using MetaDepth's syntax. As in the figures of this paper, the potency of elements is specified after the '@' symbol; if an element does not specify any potency, it takes the one of its immediate container. `Node` is MetaDepth's keyword for clabject (line 2 in Listing 2), and inheritance is declared with a colon (lines 7 and 13 of Listing 3). Constraints in MetaDepth can be defined using Java or the Epsilon Object Language (EOL) [30]. EOL is a variant of OCL extended with model manipulation primitives, like assignments and loops. Constraints have a name and a potency, and can be defined in the context of nodes (i.e., clabjects), edges and models. In Listing 2, the `TaskKind` node defines a constraint named `C1` with potency 1 (line 7), and a constraint named `C2` with potency 2 (line 11).

Recently, we have extended MetaDepth to automate the analysis of EOL/

[2] `http://plantuml.sourceforge.net/`

```
Model ProcessModel@2 {
  Node TaskKind {
    name@1 : String[0..1];
    startDay : int;
    next : TaskKind[*];

    C1@1:
      $ TaskKind.allInstances()->forAll(t |
          t <> self implies
          t.name <> self.name) $
    C2:
      $ self.next->forAll(n |
          self.startDay < n.startDay) $
  }
}
```

Listing 2: Definition of level 2 in Fig. 1, using MetaDepth.

```
ProcessModel SEProcessModel {
  abstract TaskKind SoftwareEngineeringTask {
      final : boolean = false;
      C3: $ self.startDay>0 $
  }

  TaskKind Coding : SoftwareEngineeringTask {
    name = 'Coding';
    nextPhase : SoftwareEngineeringTask[1..*]{next};
    C4: $ self.final = false $
  }

  TaskKind Testing : SoftwareEngineeringTask {
    name = 'Testing';
    nextPhase : Testing[*]{next};
    C5: $ self.final implies self.nextPhase->size()=0 $
    C6: $ Coding.allInstances()->exists(c |
          c.startDay = self.startDay) $
  }
}
```

Listing 3: Definition of level 1 in Fig. 1, using MetaDepth.

OCL integrity constraints in multi-level models. For this purpose, we have integrated the USE Validator [31] model finder, and have made available the following two new commands in MetaDepth:

- complete: it completes a model at a certain level, to make it satisfy the constraints of all its model types at higher levels. This corresponds to the analysis scenario in Fig. 3(b), but the depth is not constrained to be 2. This command can receive additional constraints that the result of the completion should satisfy, or a model fragment that should be part of the completion.

- sat: it checks the strong or weak satisfiability of a given model, and generates a witness model if the model is satisfiable. This corresponds to the analysis scenario in Fig. 3(a). In this case, it is possible to indicate the range of levels where satisfiability has to be analysed, that is, the height of the output snapshot.

Fig. 21 shows the working scheme of the tool to tackle these two commands. It consists of the following steps:

1. Provision of input for the analysis, using the commands provided by MetaDepth. Here, the user selects the multi-level model to be analysed and the analysis scenario (completion, additional constraints, weak/strong satisfiability).

2. Flattening of multi-level model according to the selected analysis scenario. This step is internally encoded as a model-to-model transformation.

3. Compilation of the flattened model into the input format of the USE Validator.

4. Automated analysis of the flattened model by the generation of a snapshot with the USE Validator. If no model satisfying the scenario is found, the user is warned.

5. Translation of results back to MetaDepth.

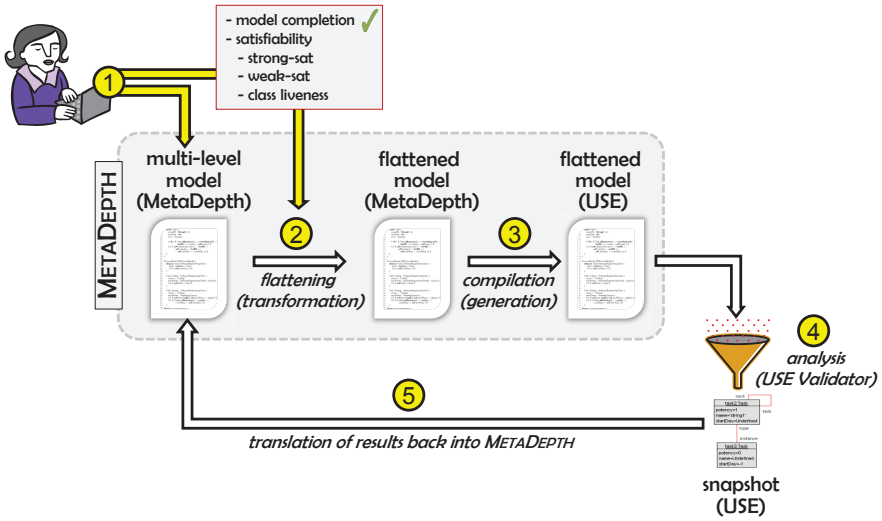

Figure 21: Automated analysis of integrity constraints in MetaDepth.

Steps 2 to 5 are automatic, triggered by the selection of the analysed scenario in step 1. In this way, the user is unaware of the usage of the USE Validator in the background, or the translation between different formats.

We will demonstrate these steps for the running example, considering the scenario in Fig. 3(b), i.e., we start from models at levels 2 and 1, and check their instantiability at level 0. Fig. 8 shows the merging of levels 1 and 2 for the running example, while part of its translation into the input format of USE is shown in Listing 4. The USE Validator does not currently support solving with arbitrary strings. All strings must adhere to the format 'string<number>'. Thus, our code generator converts any string in the original OCL constraints and operations into this format. This is why the string 'Coding' in constraint C7 gets substituted by 'string0' in the USE file (line 26 in the listing). Another limitation of the USE Validator is that it only supports collections of type Set. Hence, the return type of the operations introduced by our compilation is Set instead of Bag (see lines 11, 20 and 28). As a consequence, no reference in the generated snapshot will contain the same object twice.

```
1  -- generated by MetaDepth
2  model ProcessModelFlattened
3
4  abstract class TaskKind
5    attributes
6      name : String
7      startDay : Integer
```

```
8    constraints
9      inv C2 : self.next()->forAll( n | self.startDay < n.startDay )
10   operations
11     next() : Set (TaskKind) = Set{}
12 end
13
14 abstract class SoftwareEngineeringTask < TaskKind
15   attributes
16     final : Boolean
17   constraints
18     inv C3 : self.startDay>0
19   operations
20     next() : Set (TaskKind) = Set{}
21 end
22
23 class Coding < SoftwareEngineeringTask
24   constraints
25     inv C4 : self.final=false
26     inv C7 : self.name='string0'
27   operations
28     next() : Set (TaskKind) = self.nextPhase
29 end
30 ...
```

Listing 4: Flattening in Fig. 8 expressed in the input format of USE.

The USE file is given as input to the model finder. In addition, the USE Validator permits configuring the search scope by setting the minimum and maximum number of instances of each class and association allowed in the generated model. To analyse strong satisfiability, the minimum bound of all classes and associations is set to 1, to enforce there is at least one instance of each one of them. To analyse weak satisfiability, we set the minimum of all classes and associations to 0, and introduce a *dummy* class with minimum and maximum instantiation scope of 1, and which defines an OCL invariant demanding a non-empty model as result of the model search. All these settings are automatically performed by MetaDepth.

If we check the satisfiability of the running example, we discover that the model is neither weak nor strong satisfiable. Revising the constraints at level 1, we realise that C6 does not express what the designer had in mind (that for any coding task, there should be a testing task starting the same day), but it expresses the converse (that for each testing class, a coding class exists). One solution is moving constraint C6 from class Testing to Coding, modified to iterate on all instances of Testing (i.e., Testing.allInstances()...). If we perform this change, the resulting model becomes satisfiable, and the USE Validator generates snapshots that demonstrate this satisfiability. Such snapshots are translated back into MetaDepth as feedback to the user, and can be visualized graphically using PlantUML. Fig. 22 shows to the left the model found by the USE Validator when checking weak satisfiability, and to the right the one generated in case of strong satisfiability. As it can be noted, each clabject is decorated with its potency.

If the scenario to solve is completing a partial model, like the one at the bottom level of Fig. 1, we need to provide a seed model for the search. For this purpose, MetaDepth internally encodes the partial model as an additional OCL constraint demanding the existence of the starting model structure, and

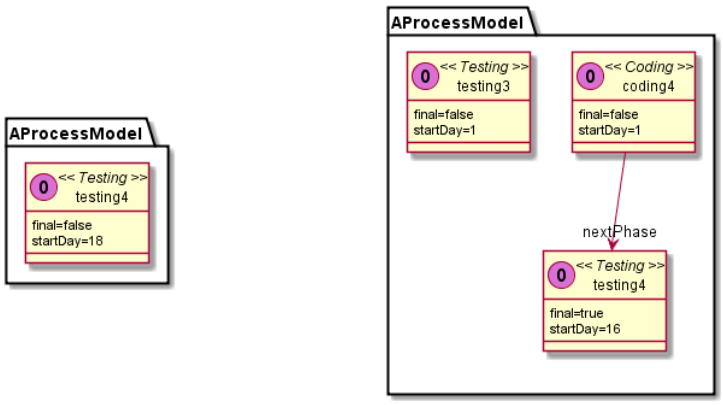

Figure 22: Showing weak (left) and strong (right) satisfiability of the running example.

which is defined in the context of the abovementioned *dummy* class. Fig. 23 shows the constraint representing our example model in Fig. 1 at level 0 (left), as well as the found complete valid model (right).

```
Coding.allInstances()->exists(cd1,cd2|
  Testing.allInstances()->exists(t1,t2|
    cd1 <> cd2 and
    t1 <> t2 and
    cd1.final = false and
    cd2.final = false and
    t1.final = false and
    t2.final = true and
    cd1.startDay = 1 and
    cd2.startDay = 5 and
    t1.startDay = 1 and
    t2.startDay = 5 and
    cd1.nextPhase->includes(cd2,t2)
    and
    t1.nextPhase->includes(t2) ))
```

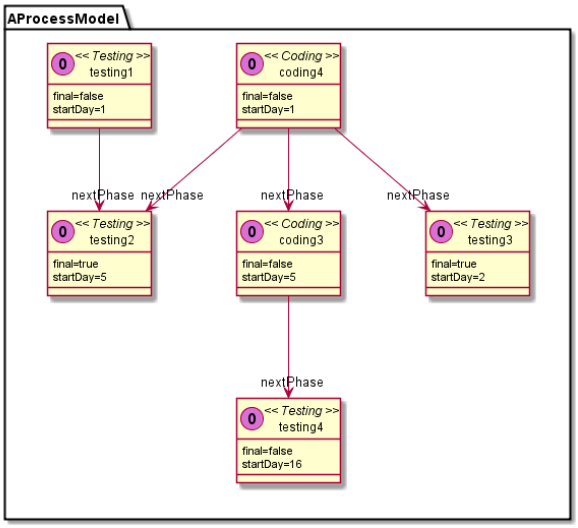

Figure 23: Encoding of incomplete model at level 0 (left). Complete valid instance (right).

By inspecting the generated model, one can realise that it contains a non-final Testing task without output tasks. This can be forbidden by modifying constraint C5 to require non-final Testing tasks to have some task in its collection

nextPhase. In this way, the generation of models with different characteristics can also serve to validate multi-level models.

7. Related work

In this section, we review the main lines of related research. First, we focus on multi-level approaches with support for expressing constraints. Then, we review approaches for the analysis of integrity constraints and model completion. Next, we study works on flattening several meta-levels. Last, we discuss other multi-level modelling variants and how to adapt our analysis method for them.

Support of constraints in multi-level modelling. There are a few multi-level approaches that provide support for constraints. Some of them define an underlying semantics based on constraints, like NIVEL [3], which is based on the Weight Constraint Rule Language (WCRL). This allows some decidable, automated reasoning procedures on NIVEL models, but they lack support for integrity constraints beyond multiplicities. Other tools like Melanee [5], which is a graphical multi-level modelling tool based on Eclipse, support the definition of OCL constraints but not their analysis as we have done in this paper.

Analysis of integrity constraints. There are several tools able to analyse the satisfiability of integrity constraints in a two-level setting. We have illustrated our method with the USE Validator [31], which translates a UML model and its OCL constraints into relational logic, and uses a SAT solver to check its satisfiability. UML2Alloy [2] follows a similar approach. Instead, UMLtoCSP [13] and EMFtoCSP [23] transform the model into a constraint satisfaction problem (CSP) to check its satisfiability, and ocl2smt [43] translates it into a set of operations on bit-vectors which can be solved by SMT solvers. The approach of Queralt [38, 39] uses resolution, and Clavel [15] maps a subset of OCL into first-order logic and employs SMT solvers to check unsatisfiability. HOL-OCL [12] is a theorem proving environment for OCL; in contrast to the previous tools, it does not rely on bounded model finding, but it is able of proving complex properties of UML/OCL specifications, though sometimes it requires user guidance to complete the proofs. In [24], a thorough review of state-of-the-art tools devoted to the verification of static software models is presented. All these tools consider two meta-levels and could be used to solve the multi-level scenarios in Section 3, once they have been translated into a two-level setting. That is, they are not directly applicable to a multi-level setting.

There are also works that rely on constraint solving for model completion in different contexts, but always in a two-level setting. For instance, in [41], the authors synthesize model editors with model completion capabilities which take into account the meta-model well-formedness constraints to automatically build correct models from partial models. For this purpose, the partial model is transformed into Alloy, and a SAT solver is used to complete it. Thus, in this approach, meta-model constraints must be expressed as Alloy facts, while constraints in METADEPTH can be expressed in EOL, which is much closer to the

standard constraint language OCL. Moreover, we allow the user to fine-tune the search by providing features of the desired model by means of constraints. Other applications of constraint solving for model completion include the synthesis of input test models from model fragments for transformation testing [26, 42].

Altogether, the use of model finders to verify properties in models is not novel. However, to the best of our knowledge, ours is the first work targeting the analysis of integrity constraints in multi-level models.

Flattening of multiple meta-levels. In [28], the authors describe a method to generate promotion transformations (i.e., model-to-metamodel transformations) by performing a kind of static flattening. Their goal is enabling deep meta-modelling atop standard two-level meta-modelling tools. The proposed flattening has some commonalities with our static flattening, like the substitution of instantiation by inheritance, or the removal of elements which cannot be instantiated due to their potency. However, the approach does not deal with integrity constraints, and only two consecutive meta-levels are considered in each promotion. In fact, the flattening in Fig. 19 uses this same idea of promoting models into meta-models, but as we discussed then, this may be inefficient in our scenario (satisfiability checking) as it may require several model generation attempts before a valid model at each meta-level is obtained.

In [21], the authors describe a way to represent in a single model a collection of models at different meta-levels. The approach is similar to our *linguistic embedding* in Section 5.3; however, the authors do not discuss the representation of OCL constraints. One goal of that work is to be able to precisely define and compare different meta-modelling concepts. Instead, our primary goal is to flatten existing multi-level models for analysing integrity constraints, for which we propose several flattenings. Such flattenings could also serve as a means to compare different variations of multi-level modelling concepts.

In [10], partial instance models are represented as class diagrams. Moreover, the authors also discuss the usefulness of partial object diagrams to encode model uncertainty, model variability or underspecification. For this purpose, they propose a technique similar to our static flattening.

In [32], the authors propose designing UML profiles by first creating a (multi-level) domain model which yields a more comprehensible design, and then to convert this domain model into a profile. They represent multi-level models by means of an UML profile, too. As our goal is to analyse multi-level models with model finders, we do not flatten into stereotypes. Encodings of multi-level models using stereotypes were discussed in our previous work [20].

Multi-level modelling variants. In this paper, we have followed a potency-based, level-agnostic approach to multi-level modelling, as implemented by our tool MetaDepth. Other approaches, called level-blind [4, 27], abstract away the notion of level. While our approach is founded on assigning a potency to constraints to allow their application to the clabjects in a certain level, we believe our analysis mechanisms could be adapted to level-blind approaches as well.

Some level-agnostic approaches provide variations of certain multi-level features, which requires fine-tuning the proposed flattening algorithm. For instance, in *dual deep instantiation* [34] (DDI) – a variant of deep instantiation – references can relate clabjects at different meta-levels, and it is possible to indicate the depth of characterization of the source and target of relationships. Instead, our algorithm considers that references always connect clabjects at the same level (although perhaps with different potency). It would be possible to consider this particular behaviour in our approach, by modifying the generated constraints (rdef in Fig. 14).

Another source of variability concerns the semantics of deep characterization. This paper assumed that instantiation is always mediated (i.e., in order to create an instance, its type must be defined in the meta-level immediately above). In [40], the so-called *single-potency* assigned to an element constrains its instantiation to only the n^{th} meta-level below (if the specified single-potency is n). To take this semantics into account, the flattening should not include a clabject if its *single-potency* is outside the desired snapshot height, and additional constraints should ensure that instances only have the specified single-potency.

The concepts of multi-level modelling are similar to the materialization pattern [16]. For instance, the work in [16] explains the usefulness of multi-level constraints, and our method could be used as an analysis mechanism.

Altogether, our approach is novel because the few multi-level modelling tools that permit specifying constraints (like Melanee) do not permit their analysis, and the approaches the permit analysing constraints do not support multiple meta-levels. Finally, the flattenings we have proposed compile multi-level models into two-level models that accept an equivalent set of instance models. We believe these flattenings could serve as an effective means to compare different multi-level modelling approaches.

8. Conclusions and future work

In this paper, we have proposed a method to check the satisfiability of constraints in multi-level models using "off-the-shelf" model finders. To this aim, the method proposes two flattenings that depend on the number of levels fed to the finder and the height of the generated snapshot. The method is supported by the MetaDepth multi-level modelling tool, which now permits checking the satisfiability of a multi-level model, as well as to complete a given model to make it satisfy all integrity constraints defined at the meta-levels above.

The analysed scenarios consider a top-down or constructive meta-modelling approach, i.e., one or several instance models are built from a given meta-model. In the future, we intend to tackle other scenarios, like the completion of models at level 1 and 2 given a model at level 0, so that they accept the model provided at level 0. This would be useful to rearchitect meta-models upon introduction of new model requirements, and in exploratory modelling [6]. For that purpose, it is not enough to use a simplified linguistic meta-model such as the one in Fig. 11(a), but a more complete meta-model like the one in Fig. 11(b) would be

needed instead. We also plan to analyse other correctness properties in multi-level models, like independence of constraints [22]. Finally, another goal is to use the proposed flattenings to clarify the semantics of the different multi-level modelling variants proposed in the literature.

Acknowledgements. We are grateful to the reviewers for their comments, which helped in improving previous versions of the paper. This work was supported by the Spanish Ministry of Economy and Competitivity with project Flexor (TIN2014-52129-R), and the Madrid Region with project SICOMORO (S2013/ ICE-3006).

References

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: a challenging model transformation. In *MoDELS*, volume 4735 of *LNCS*, pages 436–450. Springer, 2007.

[2] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 9(1):69–86, 2010.

[3] T. Asikainen and T. Männistö. Nivel: a metamodelling language with a formal semantics. *Software and Systems Modeling*, 8(4):521–549, 2009.

[4] C. Atkinson, R. Gerbig, and T. Kühne. Comparing multi-level modeling approaches. In *MULTI@MODELS*, volume 1286 of *CEUR Workshop Proceedings*, pages 53–61. CEUR-WS.org, 2014.

[5] C. Atkinson, M. Gutheil, and B. Kennel. A flexible infrastructure for multilevel language engineering. *IEEE Trans. Soft. Eng.*, 35(6):742–755, 2009.

[6] C. Atkinson, B. Kennel, and B. Goß. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In *SWESE*, 2011.

[7] C. Atkinson and T. Kühne. The essence of multilevel metamodeling. In *UML*, volume 2185 of *LNCS*, pages 19–33. Springer, 2001.

[8] C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.

[9] C. Atkinson and T. Kühne. Reducing accidental complexity in domain models. *Software and Systems Modeling*, 7(3):345–359, 2008.

[10] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski. Partial instances via subclassing. In *SLE*, volume 8225 of *LNCS*, pages 344–364, 2013.

[11] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[12] A. D. Brucker and B. Wolff. HOL-OCL: a formal proof environment for UML/OCL. In *FASE*, volume 4961 of *LNCS*, pages 97–100. Springer, 2008.

[13] J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE*, pages 547–548, 2007.

[14] J. Cabot, R. Clarisó, and D. Riera. On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software*, 93:1–23, 2014.

[15] M. Clavel, M. Egea, and M. A. G. de Dios. Checking unsatisfiability for OCL constraints. *Electronic Communications of the EASST*, 24:1–13, 2009.

[16] M. Dahchour, A. Pirotte, and E. Zimányi. Materialization and its metaclass implementation. *IEEE Trans. on Knowl. and Data Eng.*, 14(5):1078–1094, Sept. 2002.

[17] J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *TOOLS*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010. See also `http://miso.es/tools/metaDepth.html`.

[18] J. de Lara, E. Guerra, R. Cobos, and J. Moreno-Llorena. Extending deep meta-modelling for practical model-driven engineering. *Comput. J.*, 57(1):36–58, 2014.

[19] J. de Lara, E. Guerra, and J. S. Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Software and System Modeling*, 14(1):429–459, 2015.

[20] J. de Lara, E. Guerra, and J. Sánchez Cuadrado. When and how to use multi-level modelling. *ACM Transactions on Software Engineering and Methodology*, 24(2):12, 2014.

[21] M. Gogolla, J.-M. Favre, and F. Büttner. On squeezing M0, M1, M2, and M3 into a single object diagram. In *MoDELS'2005 Workshop Tool Support for OCL and Related Formalisms*, LGL-REPORT-2005-001, EPFL (Switzerland), 2005.

[22] M. Gogolla, L. Hamann, and M. Kuhlmann. Proving and visualizing OCL invariant independence by automatically generated test cases. In *TAP*, volume 6143 of *LNCS*, pages 38–54. Springer, 2010.

[23] C. A. González, F. Büttner, R. Clarisó, and J. Cabot. EMFtoCSP: a tool for the lightweight verification of EMF models. In *FormSERA*, pages 44–50. IEEE, 2012.

[24] C. A. González and J. Cabot. Formal verification of static software models in MDE: A systematic review. *Information & Software Technology*, 56(8):821–838, 2014.

[25] E. Guerra and J. de Lara. Towards automating the analysis of integrity constraints in multi-level models. In *MULTI*, volume 1286 of *CEUR Workshop Proceedings*, pages 63–72. CEUR-WS.org, 2014.

[26] E. Guerra and M. Soeken. Specification-driven model transformation testing. *Software and Systems Modeling*, 14(2):623–644, 2015.

[27] B. Henderson-Sellers, T. Clark, and C. Gonzalez-Perez. On the search for a level-agnostic modelling language. In *CAiSE*, volume 7908 of *LNCS*, pages 240–255. Springer, 2013.

[28] G. Kainz, C. Buckl, and A. Knoll. Automated model-to-metamodel transformations based on the concepts of deep instantiation. In *MODELS*, volume 6981 of *LNCS*, pages 17–31. Springer, 2011.

[29] B. Kennel. *A unified framework for multi-level modeling.* PhD thesis, U. Mannheim, 2012.

[30] D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Object Language (EOL). In *ECMDA-FA*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.

[31] M. Kuhlmann, L. Hamann, and M. Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In *TOOLS (49)*, volume 6705 of *LNCS*, pages 290–306. Springer, 2011.

[32] F. Mallet, F. Lagarde, C. André, S. Gérard, and F. Terrier. An automated process for implementing multilevel domain models. In *SLE*, volume 5969 of *LNCS*, pages 314–333. Springer, 2009.

[33] R. C. Martin, D. Riehle, and F. Buschmann. *Pattern Languages of Program Design 3.* Addison-Wesley, 1997.

[34] B. Neumayr, M. A. Jeusfeld, M. Schrefl, and C. Schütz. Dual deep instantiation and its ConceptBase implementation. In *CAiSE*, volume 8484 of *LNCS*, pages 503–517. Springer, 2014.

[35] OMG. Object Constraint Language (OCL) 2.4. `http://www.omg.org/spec/OCL/`, 2014.

[36] OMG. OMG's meta-object facility (MOF). `http://www.omg.org/mof/`, 2016.

[37] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *ICECCS*, pages 162–171. IEEE Computer Society, 2009.

[38] A. Queralt, A. Artale, D. Calvanese, and E. Teniente. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering*, 73:1–22, 2012.

[39] A. Queralt and E. Teniente. Verification and validation of UML conceptual schemas with OCL constraints. *ACM Transactions on Software Engineering and Methodology*, 21(2):13, 2012.

[40] A. Rossini, J. de Lara, E. Guerra, A. Rutle, and U. Wolter. A formalisation of deep metamodelling. *Formal Asp. Comput.*, 26(6):1115–1152, 2014.

[41] S. Sen, B. Baudry, and H. Vangheluwe. Towards domain-specific model editors with automatic model completion. *Simulation*, 86(2):109–126, 2010.

[42] S. Sen, J.-M. Mottu, M. Tisi, and J. Cabot. Using models of partial knowledge to test model transformations. In *ICMT*, volume 7307 of *LNCS*, pages 24–39. Springer, 2012.

[43] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using boolean satisfiability. In *DATE*, pages 1341–1344. IEEE, 2010.

[44] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008. See also `http://www.eclipse.org/modeling/emf/`.

[45] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.

Appendix

For understandability, sections 4 and 5 presented our flattening strategies for three meta-levels. Next, we provide a general algorithm that implements such strategies in a systematic way for any depth of multi-level model and any height of snapshot. Listing 5 presents the pseudocode of a procedure flattening implementing this algorithm. It has the following four parameters:

- mlm: multi-level model to analyse
- t: potency of the top-most level in the multi-level model
- d: depth, that is, number of levels to analyse
- h: height, that is, number of levels to be generated in the snapshot

The procedure returns a new model with the flattened version of the multi-level model (i.e., it does not modify the multi-level model in-place, but it performs a model-to-model transformation). The function target(elem) that is used in the pseudocode receives an element of the input multi-level model, and returns the element into which it was transformed in the output flattened model.

```
proc flattening (mlm: MultiLevelModel, t: int, d: int, h: int)
   result = create model

   for level=t to (t−d+1) step −1 do /* process each meta−level for the considered depth */

   /* step 1: copy all clabjects; non−top clabjects are copied as abstract */
      for each clabject in mlm.at(level) do
         var class = clabject.clone
         if level > (t−d+1) then // Fig. 4(a), the level is not the top
            set class to abstract
         else if clabject.potency = 0 // Clabject cannot be instantiated, set to abstract (cf. Fig 4(b))
            set class to abstract
         endif
         add class to result

   /* step 2: copy all attribute definitions; if height of snapshot is bigger than 1, the attributes are set to
   optional, and forced to have a value according to their potency (see constraints C13−C14 in Fig. 16) */
         for each attribute in clabject do
            var att = attribute.clone
            if h > 1 then
               set att.mincardinality to 0
               var cr1 = "self.potency = " + (clabject.potency−attribute.potency) +
                         " implies not self." + attribute.name + ".oclIsUndefined()"
               var cr2 = "self.potency < " + (clabject.potency−attribute.potency) +
                         " implies self." + attribute.name + " = " +
                         " self.type.oclAsType(" + clabject.name + ")." + attribute.name
               add cr1 to class
               add cr2 to class
            endif
            add att to class
         endfor

   /* step 3: attribute values (i.e., slots) become constraints in the owner class (see C7 in Fig. 8) */
         for each slot in clabject do
            var cr = "self." + slot.name + " = " + slot.value
            add cr to class
         endfor

   /* step 4: replace instantiation between levels by inheritance (only for clabjects without ancestors) */
         if level <> t and clabject.superclasses is empty then
            add target(clabject.type) to class.superclasses
         endif
      endfor

   /* step 5: copy all inheritance relationships */
      for each clabject in mlm.at(level) do
         for each superclass of clabject do
            add target(superclass) to target(clabjet).superclasses
         endfor
      endfor

   /* step 6: copy all constraints evaluated in levels of the snapshot to generate; if height of snapshot
   is bigger than 1, modify constraints to consider their potency (see constraints C1−C2 in Fig. 16) */
      for each constraint in mlm.at(level) do
         if (level−constraint.potency) >= (t−d) and
            (level−constraint.potency) <= (t−d−h+1) then
            var cr = constraint.clone
            if h > 1 then
               var execLevel = level−constraint.potency // execution level of constraint
               cr = "self.potency=" + execLevel + " implies " + cr
               /* replaces T.allInsances() by T.allInsaces()−>select ( x | x.potency=execLevel) */
               cr = replaceAllInstancesOfType ( cr, execLevel )
            endif
            add cr to target(constraint.context)
         endif

   /* step 7: for constraints which are not in the lowest depth, obtain navigated references, and add
```

```
    homonym operations in the reference's owner class and its subclasses (see operations next in Fig. 8) */
        if level > (t−d+1) then
            for each reference in constraint do
                for each subclass of reference.sourceClass (including itself) do
                    var op = create operation
                    set op.name to reference.name
                    set op.type to Bag of target(reference.sourceClass)
                    set op.body to union of instances of reference defined in subclass
                    add op to target(subclass)
                endfor
            endfor
        endif
    endfor
endfor

/* step 8: copy all references in the lowest depth; if height of snapshot is bigger than 1, add
constraint to control depth of instantiation */
for each clabject in mlm.at(t−d+1)
    for each reference in clabject do
        var ref = reference.clone
        set ref.cardinality to *
        if h > 1 then
            var cr = "self.potency < " + (clabject.potency−reference.potency) +
                     " implies self." + reference.name + "−>isEmpty()"
            add cr to target(clabject)
        endif
        add ref to target(clabject)
    endfor
endfor

/* step 9: if height of snapshot is bigger than 1, add superclass Clabject (see Fig. 16) and integrity
constraints ensuring correct values for the potency (see constraints C9 to C12 in Fig. 16) */
if h > 1 then
    var class−clabject = create class "Clabject"
    add integrity−constraints to class−clabject
    add class−clabject to result
    for each clabject in mlm do
        if clabject.superclasses is empty then
            add class−clabject to target(clabject).superclasses
        endif
    endfor
endif

return result
endproc
```

Listing 5: Flattening algorithm for any depth (d) and height (h).