Towards Automating the Analysis of Integrity Constraints in Multi-level Models

Esther Guerra and Juan de Lara

Universidad Autónoma de Madrid (Spain)
{Esther.Guerra, Juan.deLara}@uam.es

Abstract. Multi-level modelling is a technology that promotes an incremental refinement of meta-models in successive meta-levels. This enables a flexible way of modelling, which results in simpler and more intensional models in some scenarios. In this context, integrity constraints can be placed at any meta-level, and need to indicate at which meta-level below they should hold. This requires a very careful design of constraints, as constraints defined at different meta-levels may interact in unexpected ways. Unfortunately, current techniques for the analysis of the satisfiability of constraints only work in two meta-levels.
In this paper, we give the first steps towards the automation of mechanisms to check the satisfiability of integrity constraints in a multi-level setting, leveraging on "off-the-shelf" model finders.

1 Introduction

Multi-level modelling [3] is a promising technology that enables a flexible way of modelling by allowing the use of an arbitrary number of meta-levels, instead of just two. This results in simpler models [4], typically in scenarios where the type-object pattern or some variant of it arises.

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 [10]. 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, 6, 12, 13, 16] to reason about correctness properties, like satisfiability (*is there a valid model that satisfies all constraints?*). However, constraints in multi-level models can be placed at any meta-level and be evaluated any number of 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 the first steps towards a systematic method for the analysis of a basic quality property in multi-level modelling: the satisfiability of integrity constraints. We base our approach 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, we need 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. Thus, we discuss alternative flattening algorithms for different analysis scenarios. As a proof of concept, we illustrate our method through the analysis of MetaDepth multi-level models [9] using the USE Validator [13] for model finding.

Paper organization. Section 2 introduces multi-level modelling as designed in the MetaDepth tool. Section 3 presents properties and scenarios in the analysis of multi-level models. Section 4 discusses strategies for flattening multi-level models for their analysis with standard model finders. Section 5 describes the use of a model finder to analyse MetaDepth models. Last, Section 6 reviews related research and Section 7 draws some conclusions and future works.

2 Multi-level modelling

We will illustrate our proposal using a running example in the area of domain-specific process modelling, while the introduced multi-level concepts are those provided by the MetaDepth tool [9]. The example is shown in Fig. 1 using MetaDepth syntax (left) and a graphical representation (right). For the textual syntax, we only show the two upper meta-levels of the solution.

The main elements in a multi-level solution are models, clabjects, fields, references and constraints. All of them have a potency, indicated with the @ symbol. The potency is a positive number (or zero) that specifies in how many meta-levels an element can be instantiated. It is automatically decremented at each deeper meta-level, and when it reaches zero, the element cannot be instantiated in the meta-levels below. If an element does not define a potency, it receives the potency from its enclosing container, and ultimately from the model. Hence, the potency of a model is similar to the notion of *level* in other multi-level approaches [3].

As an example, the upper model in Fig. 1 contains a clabject Task with potency 2, thus allowing 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., c1). Task 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.

In MetaDepth, references with potency 2 (like next) need to be instantiated at potency 1 (e.g., nextPhase), to be able to instantiate these latter instances at level 0. The cardinality of a reference constrains the number of instances at the meta-level right below. Thus, the cardinality of next controls the instantiations at level 1, and the cardinality of nextPhase the ones at level 0.

Constraints can be declared at any meta-level. In this case, the potency states how many meta-levels below the constraint will be evaluated. Constraint C1, which ensures uniqueness of task names, has potency 1, and therefore it will be evaluated one meta-level below. 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)

```
1  Model ProcessModel@2 {
2    Node Task {
3      name@1 : String[0..1];
4      startDay : int;
5      next : Task[*];
6
7      C1@1: $ Task.allInstances()->excluding(self)->
8                forAll( t | t.name<>self.name ) $
9      C2: $ self.references('next')->forAll( r |
10               self.value(r)->forAll( n |
11                 self.startDay < n.startDay )) $
12   }
13 }
14
15 ProcessModel SEProcessModel {
16   abstract Task SoftwareEngineeringTask {
17       final : boolean = false;
18       C3: $ self.startDay>0 $
19   }
20
21   Task Coding : SoftwareEngineeringTask {
22     name = 'Coding';
23     nextPhase : SoftwareEngineeringTask[1..*]{next};
24     C4: $ self.final = false $
25   }
26
27   Task Testing : SoftwareEngineeringTask {
28     name = 'Testing';
29     nextPhase : Testing[*]{next};
30     C5: $ self.final implies self.nextPhase->size()=0 $
31     C6: $ Coding.allInstances()->exists( c |
32               c.startDay = self.startDay ) $
33   }
34 }
```

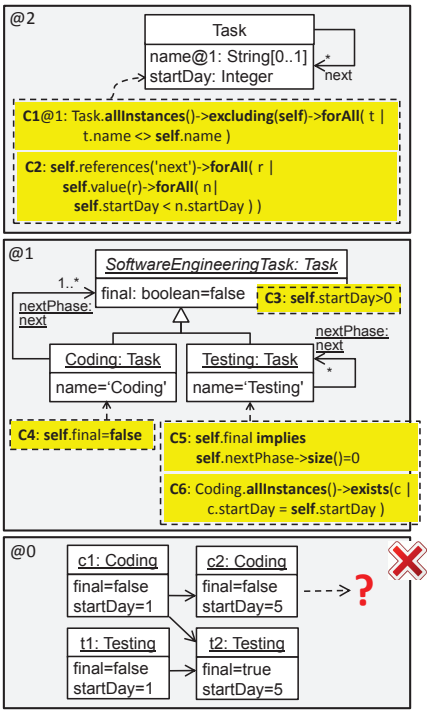


Fig. 1. Running example in MetaDepth syntax (left) and diagram (right).

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. Instead, to allow the access to indirect instances of a given reference, MetaDepth offers two operations:

- references returns the name of the references that instantiate a given one. For example, self.references('next') evaluated at clabject c1 yields Set{'nextPhase'}, as the type of c1 defines the reference nextPhase as an instance of next.
- value returns the content of a reference with the given name. For example, self.value('nextPhase') evaluated in clabject c1 yields Set{c2,t2}.

In the intermediate meta-level, constraints C3, C4, C5 and C6 have potency 1, and thus they need to be satisfied by models at the subsequent meta-level. 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 the figure is an attempt (with no success) to instantiate the model with potency 1. The modeller started adding the coding tasks c1 and c2 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 c2 needed to be followed by some other task. Connecting c2 with t2 is not a valid solution because this would violate constraint C2. Therefore, how can he connect the testing tasks to the coding tasks to satisfy all constraints? The next section explains how model finders can help in this situation.

3 Analysis of multi-level models: properties and scenarios

A meta-model should satisfy some basic properties, like the possibility of creating a non-empty instance that does not violate the integrity constraints. Several works [7] rely on model finding techniques to check correctness properties of meta-models in a standard two meta-level setting, like:

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

Model finding techniques can also be helpful in a multi-level setting. 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 to check the consistency of the integrity constraints at levels 1 and 2 through the analysis of the abovementioned 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, model finders work in a two-level setting (i.e., they receive a meta-model and produce an instance). To enable their use with several meta-levels, we need flattening operations that merge several meta-levels into one, which can then be the input to the finder. The flattening operations must take into account how many meta-levels are going to be used in the analysis (*depth of model*), as well as the number of meta-levels in the generated snapshot (*height of snapshot*).

Fig. 2 shows the different scenarios we need to solve for the analysis of multi-level models. Fig. 2(a) is the most usual case, where only the definition of the top-most model is available, and we want to check whether this 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 searched snapshot is 2. As standard model finders

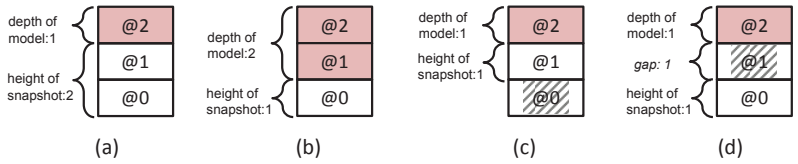


Fig. 2. Different scenarios in the analysis of a multi-level model.

only provide snapshots of models residing in one meta-level, we will need to emulate the generation of several meta-levels within one.

In Fig. 2(b), the models of several successive meta-levels are given, and the purpose 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 introduced at a meta-level (e.g., `@1`) with respect to those in the meta-levels above (e.g., `@2`). This scenario would have helped in the analysis of the constraints at levels 2 and 1 in Fig. 1. In Fig. 2(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. 2(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, the meta-model to be fed into the solver still needs to be adjusted, removing constraints with potency bigger than 1.

Finally, in Fig. 2(d), only the top-most model is available, and the designer is interested just 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.

4 Flattening multi-level models for analysis

In this section, we use the running example to show 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.

4.1 Depth of analysed model

To analyse a model that is not at the top-most meta-level (like in Fig. 2(b), where the goal is analysing level 1), we need to merge it with all its type models at higher levels. This permits considering all constraints and attributes defined at such higher levels. For simplicity, we assume the merging of just two meta-levels. Fig. 3(a) shows this flattening applied to the running example: levels 1 and 2 are merged, as the purpose is creating a regular instance at level 0.

First, the flattening handles the top level. All its clabjects (Task in the example) are set to abstract to disable their instantiation. All references are deleted (next), as only the references defined at level 1 (nextPhase) can be instantiated at level 0. All attributes are kept, as if their potency is bigger or equal than the depth (2) plus the height (1), they still can receive a default value. Constraints with potency different from 2 (C1) are deleted as they do not constrain the level we want to instantiate (level 0). As the figure shows, the notion of potency does not appear in the flattened model. This can be interpreted as all elements having potency 1.

Then, the model at potency 1 is handled. For clabjects, the instantiation relation is changed by inheritance. In this way, SoftwareEngineeringTask is set to inherit from Task 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 naturally define all attributes that were assigned potency 2 at level 2 (startDay), and receive a value at level 0. For attributes that receive a value at level 1, we need to emulate their value using constraints. Thus, in the example, we substitute the slot name from Coding and Testing, by constraints C7 and C8. The attributes, references, and constraints defined by the clabjects at level 1 are kept. Finally, we generate two operations references and value to emulate the homonym MetaDepth built-in operations by collecting the knowledge about reference instantiation statically. That is, they encode that nextPhase in Coding and Testing are instances of next. As a result of this flattening, we can use a model finder to check whether there is a valid instance at level 0.

4.2 Height of snapshot generation

In this scenario, we need to emulate the search of a set of models spawning several meta-levels. For simplicity, we assume the scenario in Fig. 2(a), aimed at generating two models in consecutive levels from a meta-model with potency 2.

A possible strategy is to split each clabject C with potency 2 into two classes CType and CInstance holding the attributes, references and constraints with potency 1 and 2, respectively, and related by a reference type. However, this solution gets cumbersome if C is related or inherits from other clabjects, and requires rewriting the constraints in terms of the newly introduced types and relations.

Another possibility is to proceed in two steps: first a model of potency 1 is generated, which is promoted into a meta-model that can be instantiated into a model of potency 0. However, this solution may require rewriting the constraints with potency 2 in terms of the types generated at potency 1. Moreover, it does not consider all constraints at a time, which may result in different attempts before two valid models at potencies 1 and 0 are obtained.

Instead, we propose the flattening in Fig. 3(b), which adds a parent abstract class Clabject that makes explicit typical clabject features, like ontological typing and potency. All constraints are kept, but they need to be changed slightly to take into account their potency. Thus, C1 is added the premise self.potency=1 implies... so that it gets only applicable to tasks with potency 1, and similar for constraint C2 for tasks at potency 0. To emulate the potency of attributes, they are

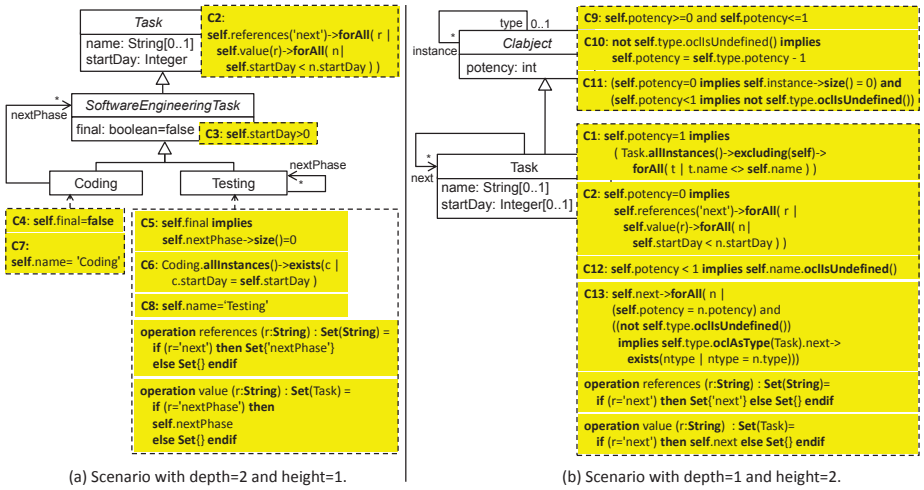
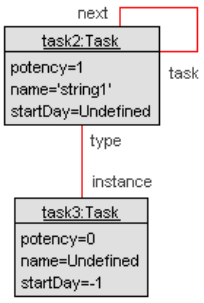


Fig. 3. Flattenings for different depths and heights.

set to optional (cardinality [0..1]), and we add constraints ensuring that the attributes are undefined in the meta-levels where they cannot be instantiated. For example, name has originally potency 1, and hence it can only receive a value in tasks of potency 1 (constraint C12). Constraint C13 ensures that references (like next) do not cross meta-levels and are correctly instantiated at every meta-level. The latter means that, if two tasks at potency 0 are related by a next reference, then their types must be related via a next reference as well. While this does not fully captures the instantiation semantics as there is no explicit "instance-of" relation between references with different potencies, it suffices our purposes. Finally, constraints C9 to C11 ensure correct potency values for types and instances. As an example, the figure to the right shows a snapshot with height 2, generated by USE from the definition in Fig. 3(b).



5 Automating the analysis of constraints

Next, we illustrate our method by checking the satisfiability of MetaDepth multi-level models with the USE Validator [13] for model finding. The checking includes the following steps: (1) flattening of multi-level model according to the selected scenario; (2) translation of flattened model into the input format of USE; (3) generation of snapshot with the USE Validator tool; and (4) translation of the results back to MetaDepth. We will demonstrate these steps for the running example, considering the scenario in Fig. 2(b), i.e., we start from models at potency 2 and 1, and check their instantiability at potency 0.

Fig. 3(a) shows the merging of levels 1 and 2 for the running example, while part of its translation into the input format of USE is listed below. The USE

Validator does not currently support solving with arbitrary strings, but they must adhere to the format 'string<number>'. Thus, we translate strings to this format, where 'next' is substituted by 'string0' (lines 12 and 27), 'nextPhase' by 'string1' (lines 28 and 31), and so on.

```
1  model ProcessModel
2
3  abstract class Task
4    attributes
5      name : String
6      startDay : Integer
7    operations
8      references(r:String) : Set(String) = Set{}
9      value (r:String) : Set(Task) = Set{}
10   constraints
11     inv C2:
12       self.references('string0')->forAll(r |
13         self.value(r)->forAll(n |
14           self.startDay < n.startDay))
15 end
16
17 abstract class SoftwareEngineeringTask < Task
18   attributes
19     final : Boolean
20   constraints
21     inv C3: self.startDay > 0
22 end
23
24 class Coding < SoftwareEngineeringTask
25   operations
26     references(r:String) : Set(String) =
27       if (r='string0')
28       then Set{'string1'}
29       else Set{} endif
30     value(r:String) : Set(Task) =
31       if (r='string1')
32       then self.nextPhase
33       else Set{} endif
34   constraints
35     inv C4: self.final=false
36     inv C7: self.name = 'string2'
37 end
38
39  ...
40
41 association Coding_nextPhase between
42   Coding[*]
43   SoftwareEngineeringTask[1..*] role nextPhase
44 end
```

If we try to find a valid instance of this definition, we discover that the only model satisfying all constraints is the empty model. Thus, the model at level 1 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 (i.e., 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 the snapshots in Fig. 4.

Weak satisfiability is checked by finding a valid non-empty model. The USE Validator allows configuring the minimum and maximum number of objects and references of each type in the generated model. If we set a lower bound 1 for class Testing, we obtain the instance model shown in the left of Fig. 4. Strong

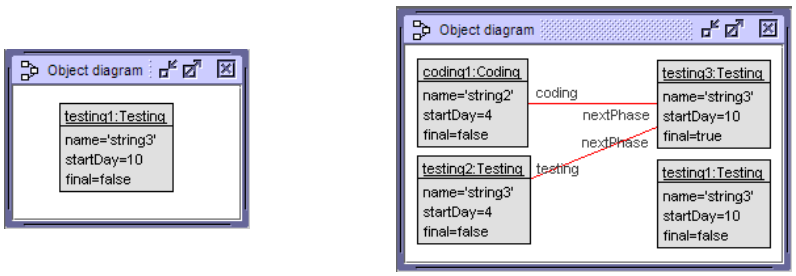


Fig. 4. Showing weak (left) and strong (right) satisfiability of the running example.

```
Coding.allInstances()->exists(c1,c2|
  Testing.allInstances()->exists(t1,t2|
    c1.final = false and
    c2.final = false and
    t1.final = false and
    t2.final = true and
    c1.startDay = 1 and
    c2.startDay = 5 and
    t1.startDay = 1 and
    t2.startDay = 5 and
    c1.nextPhase->includes(c2,t2) and
    t1.nextPhase->includes(t2) ))
```

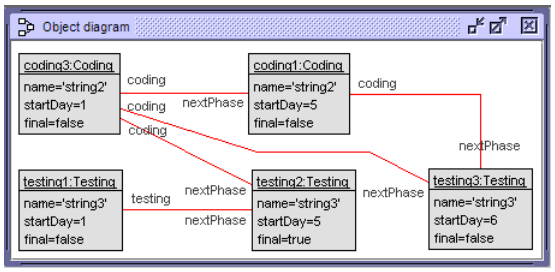


Fig. 5. Encoding of incomplete model at level 0 (left). Complete valid instance (right).

satisfiability is checked by finding a model that contains an instance of every class and reference. By assigning a lower bound 1 to all types, the USE Validator finds the model to the right of Fig. 4.

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. This can be emulated by an additional constraint demanding the existence of the starting model structure. Fig. 5 shows the OCL constraint representing our example model at level 0 (left), as well as the complete valid model found by USE (right).

6 Related work

Some multi-level approaches have an underlying semantics based on constraints, like Nivel [2], which is based on WCRL. This allows some decidable, automated reasoning procedures on Nivel models, but they lack support for integrity constraints beyond multiplicities.

There are several tools to validate the satisfiability of integrity constraints in a two-level setting. We have illustrated our method with the USE Validator [13], which translates a UML model and its OCL constraints into relational logic, and uses a SAT solver to check its satisfiability. UML2Alloy [1] follows a similar approach. Instead, UMLtoCSP [6] and EMFtoCSP [12] transform the model into a constraint satisfaction problem (CSP) to check its satisfiability, and ocl2smt [16] translates it into a set of operations on bit-vectors which can be solved by SMT solvers. The approach of Queralt [14] uses resolution and Clavel [8] maps a subset of OCL into first-order logic and employs SMT solvers to check unsatisfiability. HOL-OCL [5] is a theorem proving environment for OCL. In contrast to the enumerated tools, it does not rely on bounded model finding, but it is able of proving complex properties of UML/OCL specifications. All these works 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. Other works use constraint solving for model completion [15], also in a two-level setting.

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.

7 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 flattenings that depend on the number of levels fed to the finder and the height of the generated snapshot. The method has been illustrated using MetaDepth and the USE Validator.

Currently, we are working towards a tighter integration of the USE validator with MetaDepth, providing commands to e.g., complete a given model. We also plan to analyse other properties, like independence of constraints [11].

Acknowledgements. This work has been funded by the Spanish Ministry of Economy and Competitivity with project "Go Lite" (TIN2011-24139).

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. T. Asikainen and T. Männistö. Nivel: a metamodelling language with a formal semantics. *Software and Systems Modeling*, 8(4):521–549, 2009.
3. C. Atkinson and T. Kühne. The essence of multilevel metamodeling. In *UML*, volume 2185 of *LNCS*, pages 19–33. Springer, 2001.
4. C. Atkinson and T. Kühne. Reducing accidental complexity in domain models. *Software and Systems Modeling*, 7(3):345–359, 2008.
5. 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.
6. 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.
7. 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.
8. M. Clavel, M. Egea, and M. A. G. de Dios. Checking unsatisfiability for OCL constraints. *Electronic Communications of the EASST*, 24:1–13, 2009.
9. J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *TOOLS*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010.
10. 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.
11. 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.
12. C. A. González, F. Büttner, R. Clarisó, and J. Cabot. EMFtoCSP: a tool for the lightweight verification of EMF models. In *FormSERA*, 2012.
13. 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.
14. A. Queralt and E. Teniente. Verification and validation of UML conceptual schemas with OCL constraints. *TOSEM*, 21(2):13, 2012.
15. S. Sen, B. Baudry, and H. Vangheluwe. Towards domain-specific model editors with automatic model completion. *Simulation*, 86(2):109–126, 2010.
16. 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.