# Example-based Validation of Domain-Specific Visual Languages

Jesús J. López-Fernández Esther Guerra Juan de Lara
Universidad Autónoma de Madrid
{jesusj.lopez, esther.guerra, juan.delara}@uam.es

## Abstract

The definition of Domain-Specific Languages (DSLs) is a recurrent activity in Model-Driven Engineering. However, their construction is many times an ad-hoc process, partly due to the lack of tools enabling a proper engineering of DSLs and promoting domain experts to play an active role.

The focus of this paper is on the validation of meta-models for visual DSLs. For this purpose, we propose a language and tool support for describing properties that instances of meta-models should (or should not) meet. Then, our system uses a model finder to produce example models, enriched with a graphical concrete syntax, that confirm or refute the assumptions of the meta-model developer.

Our language complements metaBest, a framework for the validation and verification of meta-models that includes two other languages for unit testing and specification-based testing of meta-models. A salient feature of our approach is that it fosters interaction with domain experts by the use, processing and creation of informal drawings constructed in editors liked yED or Dia. We assess the usefulness of the approach in the validation of a DSL for house blueprints, with the participation of 26 $4^{th}$ year computer science students.

***Categories and Subject Descriptors*** I.6.4 [*Computing Methodologies*]: Model Validation and Analysis

***Keywords*** Meta-modelling, Domain-Specific Visual Languages, Meta-model Validation and Verification

## 1. Introduction

Domain-Specific Visual Languages (DSVLs) are used for modelling in a wide range of disciplines, and in particular, their use is pervasive in Model-Driven Engineering (MDE) [8, 21]. Hence, the creation of new DSVLs for particular domains is a recurring activity which at least involves the definition of a meta-model with the relevant concepts and relations in the domain (the *abstract syntax*) and their graphical representation (*concrete syntax*). Typically, domain experts participate in the DSVL creation process by providing requirements for the language, either in natural language or in the form of informal drawings, while language engineers are in charge of developing the DSVL meeting the outlined requirements [16, 21]. Unfortunately, the transition from requirements to DSVL implementations is many times an ad-hoc process, and there are few tools assisting in the validation and verification (V&V) of the resulting DSVLs with respect to both correctness and quality criteria. In particular, tools enabling the assessment of the DSVL by domain experts, who may not be proficient in meta-modelling, are of special interest to obtain useful, effective languages.

To overcome some of these problems, recently, we proposed the metaBest [13] framework for the V&V of meta-models w.r.t specifications or using unit test suites. While specifications state properties that a given meta-model should fulfil (e.g., two meta-classes $x$ and $y$ should be related), test suites require developing test models upfront. However, none of these two mechanisms enable an exploratory way of testing where the system produces instance models to be inspected by the engineer, or take into account the concrete syntax of the DSVL. Moreover, they require some background on meta-modelling, which domain experts may lack.

In this paper, we complement metaBest with a new mechanism to validate DSVLs by the automated generation of example models that take into account both the abstract and concrete syntax of the DSVL. The idea is producing example instantiations of the meta-model being developed, which both domain experts and engineers can inspect more easily to detect possible flaws in the meta-model and reason on the properties that instances should have. The example generation process can be fine-tuned using a textual DSL called mmXtens, which allows constraining the number and type of objects in the example, stating their connectivity, and providing a seed model either in graphical or textual format. The semantics of mmXtens is given in terms of OCL, but it provides a more concise syntax. Moreover, in order to produce the concrete syntax for the generated examples, in addition to the typical node-arch graphical representation, we give support for spatial relations like adjacency or contain-


*SLE '15*, October 25–27, 2015, Pittsburgh, Pennsylvania, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

ment. Such relations can be induced from previous sketches provided by domain experts using simple diagramming tools (currently we provide full coverage for *yED*[1]). To automate the generation of models and their graphical layout, we use constraint solving techniques [10].

Altogether, our example-based validation approach permits an exploratory way of testing as it is possible to analyse whether certain model configurations are allowed or forbidden by a meta-model, or whether the meta-model can be instantiated at all. Moreover, the fact that we generate models in graphical concrete syntax facilitates their assessment by domain experts. Our approach is supported by a tool, and we analyse its usefulness by showing a study where we validate the assignments of 26 fourth-year undergraduate students enrolled in a course on DSLs.

*Organization*. We outline our approach in Section 2, and introduce a running example in Section 3. Next, Section 4 describes mmXtens, and Section 5 shows how to enrich the generated examples with a concrete syntax. Section 6 presents our supporting tool. Section 7 includes an empirical evaluation of our proposal. Section 8 compares with related research, and Section 9 ends with the conclusions.

## 2. Overview of the approach

Fig. 1 outlines our framework. It comprises three complementary approaches and languages for meta-model V&V: the first one is based on unit testing (mmUnit), the second one is based on meta-model property specifications (mmSpec), and the last one is based on the generation of example models (mmXtens). In this paper, we focus on the last one, and refer to [13] for details of the two former.

The goal of our example-based validation approach is to automatically produce interesting example models that can be easily inspected (even by non-meta-modelling experts) to validate the correctness of a meta-model, or counterexamples signalling meta-model flaws. The example generation process can be customized by providing a seed model for the example, as well as a set of properties that the example should fulfil. To make this customization suitable to domain experts, seed models can be sketched in diagramming tools and then imported into our framework (label ①). We also provide a simple textual DSL, called mmXtens, to express model properties in a friendlier way than using e.g. OCL.

Then, our *example generator* uses this information to automatically find a model instance that is conformant to the meta-model under test and meets the defined properties (label ②). If such a model is found within the search bounds, a *layout generator* proceeds with the creation of its concrete syntax (label ③). The arrangement of the different graphical elements in the produced visualization preserves the one in the seed model (if it was provided), and moreover, it may take into account domain-specific layout rules regarding adjacency, containment and (non-)overlapping of graphical elements. These rules might have been induced from previous sketches (label ⓪), or they can be defined as annotations in the meta-model. Since we decouple the abstract and concrete syntax generation processes, it is possible to use the most convenient approach for each step. In our case, we rely on constraint solving in both cases.

[1] http://www.yworks.com/en/products_yed_about.html

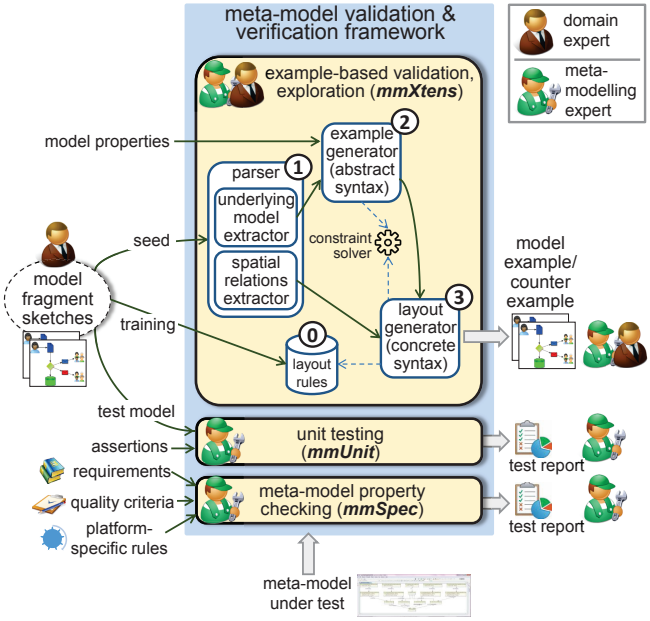


Figure 1: Overview of our framework for meta-model V&V.

## 3. Running example

To illustrate our approach throughout the paper, we are using as example a DSVL for simple house blueprints. The language requirements have been extracted from a deliverable exercise on meta-modelling, belonging to a course on DSLs for fourth-year undergraduate students in Computer Science. We will use this example to assess our proposal in Section 7.

The DSVL for blueprints should consider several types of rooms, like entries, living rooms, bedrooms, kitchens, baths, gyms and balconies. Any house should have at least one bath and one living room, at most one entry, and there are no restrictions concerning the number of rooms for the rest of types. In addition, all rooms may have any number of plug outlets and switches for lights.

The blueprint should allow designing the disposition of rooms, which for simplicity, are rectangular and have the same size. Rooms can adjoin other rooms in any of the fourth cardinal points (north, east, south and west), and there cannot be isolated rooms (i.e., without any adjacent room). Rooms can adjoin at most one other room in each direction, balconies can be adjacent to one room at most, and entries can be adjacent to three rooms at most.

It should be possible to place between zero and two windows in each exterior wall (i.e., in non-adjacent walls to

other rooms). Balconies are the only exception to this condition, as their windows can only be placed in interior walls (i.e., walls adjacent to other rooms).

Regarding doors, they can only be placed in interior walls. Rooms may have either zero or one door in each wall, and at least one door in total. Additionally, a blueprint should include exactly one entrance door, which in this case, should be in an exterior wall. If the blueprint has an entry room, the entrance door should be there.

Fig. 2 shows a valid blueprint. In this case, using a graphical syntax to represent all spatial relations between the different rooms and the location of windows and doors is helpful to better comprehend the underlying model.

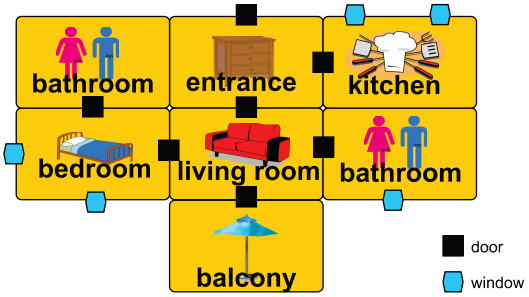


Figure 2: Example of valid house blueprint.

## 4. Meta-model validation with mmXtens

To assist in the construction and validation of DSVLs, we propose the automated generation of models from the DSVL abstract syntax meta-model. The generated models can be used to check that instances with some properties can or cannot be instantiated from the meta-model. To guide the generation process, we provide a language that is compiled into OCL for its use with OCL-based model finders [10].

We organize this section as follows. Subsection 4.1 motivates the need for a model generation facility for meta-model validation. Subsection 4.2 describes the mmXtens language. Subsection 4.3 shows some examples, and Subsection 4.4 describes its OCL-based semantics.

### 4.1 Motivation for an example generation facility

Resuming the running example, assume we build a first draft of its meta-model, as shown in Fig. 3. This meta-model is not complete, but already describes the spatial organization of rooms by means of the references north/south (one inverse of the other), and east/west (one inverse of the other). In addition, class Door allows connecting adjacent rooms. Houses have at least two rooms because, according to the specification, each house has a Bathroom and a LivingRoom. Similarly, houses need at least two doors: one entry door and another connecting the minimum of two rooms.

We define more complex constraints for the DSVL using OCL [17]. Listing 1 shows the set of invariants for our first meta-model draft. They ensure that each House has exactly

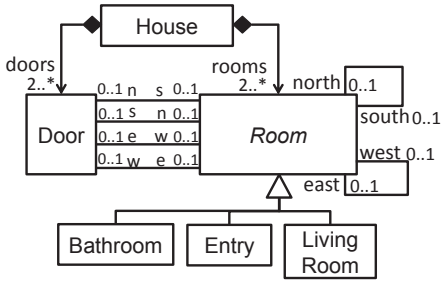


Figure 3: Initial version of the DSVL meta-model.

one entry door (invariant oneEntryDoor); Entry rooms have some entry door (entryDoor); and every Door is either an entry door, or it has values exactly in the n and s references, or in the w and e references (locCoherence). Finally, operation isEntry evaluated on a Door returns if the door is an entry door.

```
1  context House inv oneEntryDoor:
2    self.doors->one(d | d.isEntry())
3
4  context Entry inv entryDoor:
5    Bag{self.n, self.s, self.e, self.w}->exists(d | d.isEntry())
6
7  context Door inv locCoherence:
8    self.isEntry() or (
9      self.s->size() + self.n->size() <> 1 and
10     self.e->size() + self.w->size() <> 1 and
11     self.s->size() + self.n->size() +
12     self.e->size() + self.w->size() = 2 )
13
14 context Door operation isEntry() : Boolean =
15   Bag{self.n, self.s, self.e, self.w}->one(r | not r.oclIsUndefined())
```

Listing 1: Some OCL integrity constraints

While this meta-model captures some intended models, like the one in Fig. 4(a), it also accepts undesired ones, like those in Figs. 4(b-c). We tag each model with a correct or incorrect sign, depending on whether they conform to the specification in Section 3. Model (b) is unacceptable according to the specification because there is a door between non-adjacent rooms, while model (c) is also unacceptable because there is a door connected to the north and south of two rooms that are horizontally adjacent. In both cases, detecting the actual problem is difficult by inspecting the abstract syntax of the model (and we will show how to create a concrete syntax for the models in Section 5). Finally, model (d) complies with the specification, but it describes a house where the entry door leads to a bathroom, which may defeat our expectations for a reasonable house.

Fixing the problems identified in models (b,c) requires adding the OCL constraint shown in Listing 2, which ensures that two rooms connected by a door are adjacent, and their adjacency is compatible with the door location.

```
1  context Door inv roomCoherence:
2    not self.isEntry() implies (
3        not self.s.oclIsUndefined() implies
4            ( self.s.north = self.n and self.n.south = self.s ) and
5        not self.e.oclIsUndefined() implies
6            ( self.e.west = self.w and self.w.east = self.e ) )
```

Listing 2: Invariant ensuring doors and rooms are coherent

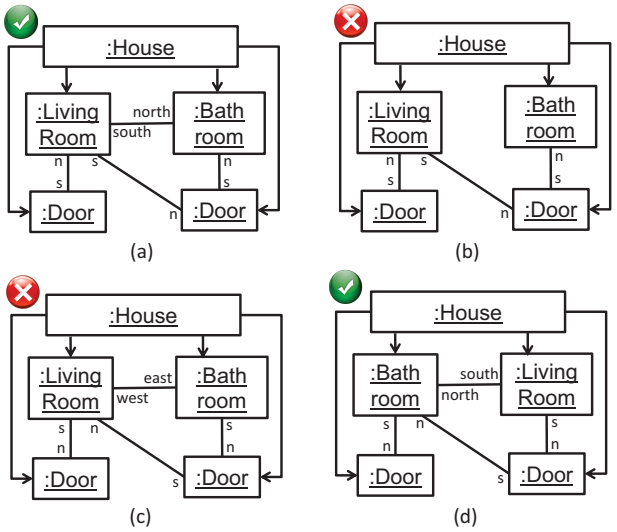


Figure 4: Correct (a,d) and incorrect (b,c) models according to the requirements presented in Section 3.

With the unexpected model (d), we might want to revise the DSL requirements to forbid entry doors in bathrooms. The reader may have noticed additional problems with the meta-model, which we will discuss in the next subsection.

Altogether, this example shows the usefulness that a facility to create meta-model instances has for validation. This is so as we have detected incorrect models according to the requirements (models (b,c)), but which are valid according to the meta-model. Building such models by hand is time-consuming, and one may be biased towards building correct models and miss less obvious faults. Generating random instances is not enough either, because we may end up with many uninteresting models. Instead, we need a mechanism to describe properties of the instances we may like to inspect. This is what the mmXtens language is directed to.

### 4.2 The mmXtens language

Fig. 5 shows our approach to model generation, where for the moment, we neglect the concrete syntax of models, to be explained in Section 5. In step 1, meta-modelling and domain experts may provide a seed model fragment. This is optional, as it is always possible to start from an empty model. In step 2, they use mmXtens to specify rules for extending the seed fragment. These rules encode properties that the generated model should or should not have. Then, in step 3, our engine translates the mmXtens specification into OCL and uses an OCL-based model finder to produce a model satisfying the specification and conformant to the meta-model under test. Finally, in step 4, the produced model (if any exists that satisfies the extension rules) can be inspected to validate whether it complies with the requirements, the expectations about the meta-model, or the intuitions about the domain.

As an example, Fig. 6 shows an mmXtens specification declaring a seed fragment and some simple rules, as well as

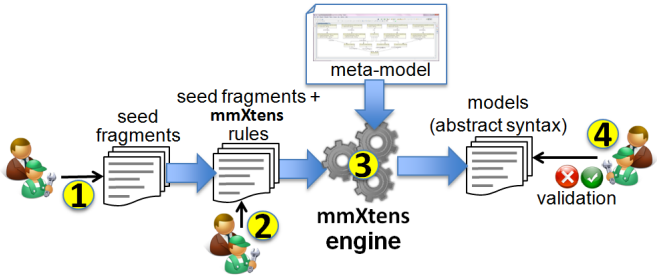


Figure 5: Example-based validation in mmXtens.

a model generated from this specification. The seed contains a House with a Bathroom and a LivingRoom. The constraints demand both rooms to be adjacent, and forbid the produced model to have more rooms than those declared in the fragment. The specification is used to produce the model in the bottom-right, which is actually the one shown in Fig. 4(a). On inspection, we validate the model as correct.

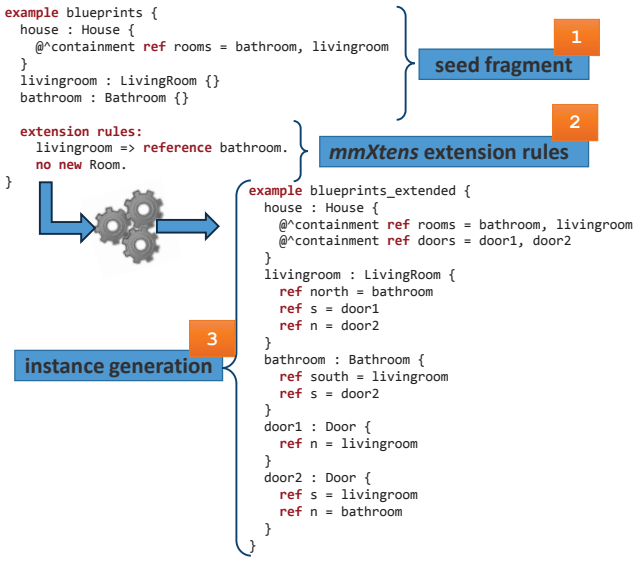


Figure 6: Example of instance generation with mmXtens.

mmXtens specifications have the structure shown in Listing 3 (lines 1–5). They can be tagged as positive (by default) or negative, depending on whether we expect the specification to be satisfiable or not. In the former case, we expect the system to produce a model that contains the given seed model fragment and satisfies the extension rules, while in the latter, we expect that no such model exists. Providing a seed fragment is optional; if given, it does not need to be a full model, but it may contain just a set of initial objects together with their features of interest (i.e., the fragment may break some lower cardinality and OCL constraints, and objects may not specify values for uninteresting attributes). It is possible to define a list of extension rules with conditions that the model to be generated should satisfy (line 4). These rules are expressed using a simple syntax, and then internally translated

into more complex OCL expressions that, transparently to the user, are used for generating the sought example model.

```
1  SPECIFICATION ::= [ positive | negative ]? example <name> {
2     SEED_FRAGMENT
3     extension rules:
4        EXTENSION_RULES
5  }
6
7  SEED_FRAGMENT ::= [ <object-id> : <type> {
8     [ <slot-id> = value ]*
9  } ]*
10
11 EXTENSION_RULES ::= [
12    <object-id> => CONDITIONER . |
13    QUANTIFIER [new]? <type> [FILTER]? [=> CONDITIONER ]? .
14 ]*
15
16 CONDITIONER ::=
17    [ reference | contain ] [ {via <refer-name>} ]? SELECTOR |
18    attr <att-name> = value
19
20 QUANTIFIER ::= <n>..<n> | strictly <n> | <n> | every | no | some
21
22 SELECTOR ::= <object-id> |
23    QUANTIFIER [ new ]? <type> [ FILTER ]?
```

Listing 3: Simplified grammar of mmXtens specifications

Extension rules may refer either to specific objects in the seed fragment (line 12), or to arbitrary objects which can exist in the seed fragment or may need to be created new (line 13). In both cases, we can use a *CONDITIONER* stating required properties for the object. For example, the extension rule used in Fig. 6:

livingroom => **reference** bathroom.

requires the existing livingroom object to be extended with a reference to the bathroom object. As we see in this example, a *CONDITIONER* may require an object to be connected with some other using the keyword **reference**. If we use the keyword **contain** instead, then, the reference should be a composition. Optionally, we may specify a reference name using the keyword **via**. Finally, we can use a *SELECTOR* (lines 22-23) to choose the target object of the reference. In the simplest case, it will be an object present in the seed fragment (e.g., bathroom in the example), though it can also be a new object created in the extended model, or a set of objects of a given type (we give examples below).

Alternatively, a *CONDITIONER* may define requirements on the attribute values of the selected object (line 18 in the listing). In the current version of mmXtens, these requirements must be concrete values. Finally, conditioners can be combined using the logical primitives **and**, **or** and **not** (omitted in the listing for simplicity).

Rules can also require properties on sets of objects of a certain type, without referring explicitly to an existing object (line 13). In such a case, we use a *QUANTIFIER* to select the objects. Valid quantifiers include intervals, **strictly** a given number, at least a given number, **every** object of a given type, **no** object of a given type, or **some** (i.e., at least one) object of a given type. If the type name is preceded by the keyword **new**, then, the selected objects must not belong to the fragment but they must be new in the extended model. If **new** is omitted, the selection is among both existing and new objects.

For instance, valid object selections include **every new** Room (for all newly created Room objects, where Room is an abstract class), **some** Bathroom, **no new** Gym or **0..4 Door**.

In this latter rule type, it is also possible to define a *FILTER* to indicate required properties of the selected objects. The definition of filters is similar to the one for conditioners, and may include conditions over attributes and references.

An mmXtens specification may have zero or more extension rules, and the extended model to produce must fulfil all of them. Moreover, we can customize the minimum and maximum number of objects in the model extension. As we will see in Section 6, this is done through a wizard.

So far, we have described the textual concrete syntax of mmXtens. Its abstract syntax has been defined through a meta-model, which is partly shown in Fig. 7. mmXtens is integrated with our two other languages for meta-model V&V. It shares the notion of seed fragment with mmUnit, but while mmUnit assertions are contained in class TestAssertionSet, mmXtens assertions are contained in ExtensionAssertionSet. Both mmXtens and mmSpec specifications follow a similar selector/filter/condition style, because our goal is to supply users with a homogeneous family of DSLs.

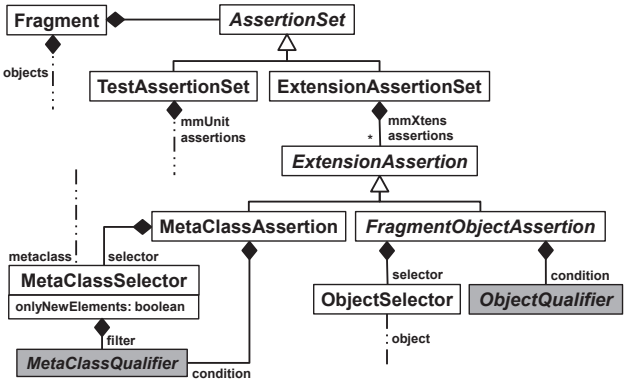


Figure 7: Excerpt of the mmXtens meta-model.

### 4.3 Examples

Next, we show examples of extension rules and illustrate how validation with mmXtens is performed in practice. The aim of the specification in Listing 4 is testing whether two bathrooms can be connected through a door. It is tagged as negative as we deem this house disposition as inadequate, and it uses the seed model fragment shown in Fig. 6.

```
1  negative example can_two_bathrooms_be_connected {
2    house : House {
3      @^containment ref rooms = bathroom, livingroom
4    }
5    bathroom : Bathroom {}
6    livingroom : LivingRoom {}
7  extension rules:
8    1 new Bathroom.
9    some Door => reference 2 Bathroom.
10 }
```

Listing 4: Can bathrooms be connected through a door?

Our system is able to extend the seed fragment to the model in Fig. 8, showing that houses may have bathrooms connected through a door. While this is unusual, both the meta-model and the requirements allow this disposition.

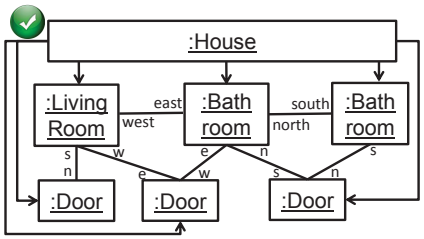


Figure 8: Counterexample model generated from Listing 4.

In this section, we depict models as object diagrams for better understanding. Nonetheless, our tool returns a textual representation like the one used to define seed fragments. This is handy, as the returned example models can become the seed in new specifications. As an example, Listing 5 tries to extend the previous model to check whether an entry door can be placed in a bathroom. To ensure the new Door is an entry door, the extension rules require that it references strictly one Room, which should be the bathroom1 object.

```
negative example enter_through_bathroom {
  house : House {
    @^containment ref rooms = bathroom, bathroom1, livingroom
    @^containment ref doors = door1, door2, door3
  }
  bathroom1 : Bathroom {
    ref north = bathroom
    ref n = door3
  }
  ...
extension rules:
  strictly 1 new Door.
  some new Door => and { reference strictly 1 Room,
                         reference bathroom1}.
}
```

Listing 5: Can the house be entered through a bathroom?

Evaluating this specification returns an extension of the seed model where bathroom1 is added an entry door, while the original entry door in the seed fragment gets connected to another room and stops being entry door. This counterexample breaks our expectations about an acceptable blueprint.

Next, we show some specifications directed to test the meta-model conformance w.r.t. the requirements. Listing 6 validates whether there may be houses without entry door, using an empty seed model. Thus, the extension rules require exactly one new House, with no door connected to exactly one room (i.e., without entry door). In this case, no model is found satisfying the defined properties, and hence, we conclude that the meta-model ensures that every house has an entry door. Similarly, we can check whether more than one entry door is allowed, which is also not possible.

```
negative example no_houses_without_entry_door {
  extension rules:
    strictly 1 new House.
    no Door => reference strictly 1 Room.
}
```

Listing 6: Can any house lack entry door?

Another interesting meta-model validation is whether houses always have at least one bathroom, which can be checked using the specification in Listing 7.

```
negative example no_houses_without_bathroom {
  extension rules:
    strictly 1 new House.
    no new Bathroom.
}
```

Listing 7: Can any house lack bathroom?

In this case, the system finds a model, proving that the meta-model is incorrect according to the requirements. Similarly, the meta-model does not enforce at least a LivingRoom. Listing 8 shows the OCL constraint needed for this. Once added to the meta-model, Listing 7 returns no model.

```
context House inv bathroomAndLivingRoom:
  self.rooms->exists( r | r.oclIsTypeOf(LivingRoom) ) and
  self.rooms->exists( r | r.oclIsTypeOf(Bathroom) )
```

Listing 8: Invariant requiring Bathroom and LivingRoom

Listing 9 tests the semantics of entry doors, in particular, whether they connect rooms to the "outside" (i.e., there is no adjacent room in the direction where the door is located).

```
negative example entry_door_leads_to_outside {
  house : House {
    @^containment ref rooms = livingroom
    @^containment ref doors = entryDoor
  }
  entryDoor: Door {
     ref s = livingroom
  }
  livingroom : LivingRoom {
     ref n = entryDoor
  }
  extension rules:
    livingroom => reference{via north} 1 new Room.
}
```

Listing 9: Can entry doors be between adjacent rooms?

The system finds a model where livingroom adjoins a room to the north, which signals an error in the meta-model. As the invariant roomCoherence in Listing 2 only considered coherence of non-entry doors with room dispositions, we need to add the new constraint in Listing 10.

```
context Door inv entryCoherence:
 self.isEntry() implies (
   not self.n.oclIsUndefined() implies self.n.south.oclIsUndefined() and
   not self.s.oclIsUndefined() implies self.s.north.oclIsUndefined() and
   not self.e.oclIsUndefined() implies self.e.west.oclIsUndefined() and
   not self.w.oclIsUndefined() implies self.w.east.oclIsUndefined() )
```

Listing 10: Invariant ensuring entry doors lead to outside

We can also test the requirement stating that Entry rooms should be adjacent to less than 4 rooms, as otherwise they would not be entries. The test is shown in Listing 11.

```
negative example entry_not_surrounded_by_rooms {
  extension rules:
    strictly 1 new House.
    some new Entry => reference 4 new Room.
}
```

Listing 11: Can entries by adjacent to 4 rooms?

Though we have not explicitly defined an OCL constraint dealing with this requirement, our system does not find any model fulfilling the mmXtens specification. Upon reflection, we can realise that this is because Entry rooms should have an entry door, and the constraint entryCoherence in Listing 10 already ensures that entry doors lead to outside.

Finally, we can test for different house layouts. For example, Listing 12 tries to generate an example of house where all rooms have an adjacent room to the east.

```
1 negative example east_adjacent_rooms {
2   extension rules:
3     1 new Room.
4     every new Room => reference{via east} 1 Room.
5 }
```

Listing 12: Can all rooms have an east-adjacent room?

Conceptually, this is topologically impossible in the 2D plane, as it would require an infinite row of rooms. However, the finder returns the topologically unsound model in Fig. 9(a), which has a circular dependency. This reveals a deeper problem, exemplified by the unsound models in Figs. 9(b,c). In reality, we overlook some of these incorrect configurations as it is difficult to mentally render these models in abstract syntax, especially those similar to Fig. 9(c). We only realized they were wrong when trying to automatically assigning them a concrete syntax (and failed!).

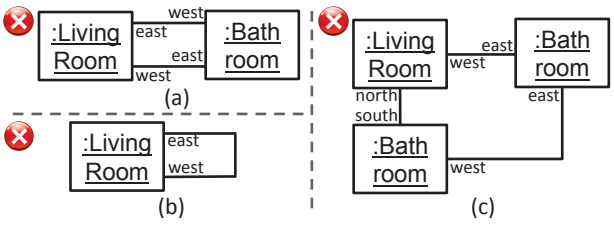


Figure 9: Topologically unsound models.

We can fix the meta-model by adding further invariants to Room ensuring that the adjacency relationships in the meta-model conform to topological adjacency. This is intricate and requires heavy use of the closure OCL operator. Instead, if we realise that a house layouts its rooms in a grid, we can incorporate integer attributes x, y to Room, and make the adjacency relationships derived. Thus, when two rooms r1, r2 exist such that r1.x+1 = r2.x and r1.y = r2.y, then both are adjacent and r1 is to the east of r2. Section 5 will show that, by asserting that both adjacency relations in the meta-model are to model (horizontal and vertical) topological adjacency, we can automatically produce this solution.

### 4.4 From mmXtens to OCL for model finding

The semantics of mmXtens is given in terms of OCL. This allows the use of an OCL-based model finder [10] for searching models satisfying the mmXtens specification.

Seed fragments are translated into OCL by nesting existential quantifiers requiring the presence of the objects in the fragment with the specified attribute values. Schematically, a fragment with n objects (o_11,...,o_1n) of type T_1, and p objects of type T_m, is translated as shown in Listing 13. Assignments of primitive attributes are translated into equalities (o_11.a_11=v_11), while references of the form o_11:T_1{ r_1 = o_12, o_13...} are translated into o_11.r_1→includesAll(Set{o_12, o_13,...}).

```
1 T_1.allInstances()->exists(o_11, ..., o_1n |
2   o_11<>o_12 and ... and o_1n-1<>o_1n and
3   T_m.allInstances()->exists(o_m1, ..., o_mp |
4     o_m1<>o_m2 and ... and o_mp-1<>o_mp and
5     o_11.a_11 = v_11 and ... and o_1n.a_11 = v_1n and
6     o_m1.a_m1 = v_m1 and ... and o_mp.a_m1 = v_mp and
7     <mmXtens-extension-rules> ))
```

Listing 13: OCL translation schema for seed fragment

Extension rules are also translated into OCL, and added to the inner-most existential quantifier of the seed fragment translation (line 7 in Listing 13). Table 1 shows the translation schema of the main mmXtens primitives. Every rule starts by either an existing object or a possibly quantified class c. In the first case, the translation is just the object identifier, while in the second, it is an expression starting by c.allInstances()→, and ending by the OCL translation of the quantifier. If the class is preceded by **new**, then we need to exclude all existing objects in the fragment. If the class is followed by a filter, then we add a select clause containing the filter expressions. The expression **reference** ⟨q⟩ c, with q a quantifier and c a class, is translated into the set of references of the selector reaching c or a superclass of c, and concatenating at the end a condition due to the translation of the quantifier q. Typically, such a condition performs checks on the size of the set. The translation of **reference** o, with o a concrete object, is similar, but we just need to check that o is included in some reference of the selector targeting o's class. In both cases, the modifier {**via** r} restricts the checking to reference r only. The translation of **contain** is similar but considering compositions only. Conditions on attributes are checked either on the selected object (for expressions starting with object selectors), or on every selected object of a given class (if starting from class selectors). In both cases they lead to the same translation. In case they are conditioners (i.e., after the =>), they are translated into a select. Finally, the last 5 rows of the table show the translation of quantifiers, where between is an operation that returns whether a given number is in an interval, and exp the generated OCL expression.

The resulting invariant is assigned to an artificial class created with the sole purpose of holding the invariant and some auxiliary operations. The idea is to require the model finder to create exactly one instance of this class.

Listing 14 shows the translation of the mmXtens specification in Listing 4. Lines 2–5 contain the translation of the seed fragment, made of nested existentials requiring the existence of every object and reference in the fragment. Line 6 contains the translation of the extension rule 1 **new** Bathroom. This is checked by demanding that the number of all Bathroom instances, excluding the ones in the seed fragment, is

| mmXtens | OCL translation |
|---|---|
| **Selectors** | |
| *Fragment object* o | o |
| *Class* c | c.allInstances() |
| **Filters/modifiers** | |
| **new** c<br>(with c a class) | → excludingAll(Set{$o_1$,...,$o_n$})<br>with {$o_1$, ..., $o_n$} all c's instances in seed fragment |
| filter or conditioner | → select(s \| ⟨filter/conditioner⟩) |
| **Filters/Conditioners** | |
| **reference**<br>⟨quantifier⟩ c<br>(with c a class) | Set{s.$r_1$, s.$r_2$, ...} → flatten()<br>→ select(x \| x.oclIsKindOf(c))<br>→ ⟨quantifier⟩<br>with {$r_1$, $r_2$...} all refs. targeting c or a superclass |
| **reference** o<br>(with o an object) | Set{s.$r_1$, s.$r_2$, ...} → flatten() → includes(o)<br>with {$r_1$, $r_2$...} all refs. targeting o's class or a superclass |
| attr = ⟨value⟩ | If conditioner of object selector o, or filter:<br>o.attr = ⟨value⟩<br>If conditioner of a class selector:<br>→ select(c \| c.attr = ⟨value⟩) → ⟨quantifier⟩ |
| attr = ⟨value⟩ | s.attr = ⟨value⟩ |
| **Quantifiers** | |
| **no** | size() = 0 |
| **some** | size() > 0 |
| **every** | size() = c.allInstances() → size()<br>c is the class **every** quantifies |
| **strictly** n | size() = n |
| $n_1..n_2$ | self.between(⟨exp⟩ → size(), $n_1$, $n_2$) |

Table 1: Translation of mmXtens extension rules to OCL

1. Here, we optimize the code and use excluding instead of excludingAll as just one object of the given kind exists. Finally, lines 7-10 hold the translation of the rule **some** Door => **reference** 2 Bathroom. The OCL subexpression selects the set of doors that reference a minimum of two objects of kind Bathroom (through the n, s, e, w references), and then, it checks that the set is not empty. As an optimization, we omit the flattening of the set in line 8 as it is unnecessary in this case. As we can see, the original mmXtens specification is more concise and easier to understand than its OCL equivalent.

```
1  context DummyClass inv can_two_bathrooms_be_connected:
2   House.allInstances()->exists(house |
3    Bathroom.allInstances()->exists(bathroom |
4     LivingRoom.allInstances()->exists(livingroom |
5       house.rooms->includesAll(Set{bathroom, livingroom}) and
6       Bathroom.allInstances()->excluding(bathroom)->size()=1 and
7       Door.allInstances()->select(s |
8         Set{s.n, s.s, s.e, s.w}->select(x |
9           x.oclIsKindOf(Bathroom))->size() >= 2)
10      ->size() > 0 )))
```

Listing 14: OCL translation of Listing 4.

## 5. Applying geometry to concrete syntax

The active participation of domain experts during the DSL construction is crucial for its success [7]. To foster their participation, our system supports interaction via drawings in two ways. First, drawings can be imported for their use as seed fragments, and it is possible to infer spatial relations between elements in the drawings. Second, generated examples can be rendered as graphical drawings, where the layout of elements may take into account spatial constraints.

### 5.1 Parsing informal drawings

Domain experts can provide drawings made with yED, which are parsed to extract the underlying model processable by our tool. The extracted model can be materialized using our textual concrete syntax for model fragments, and then included in mmXtens specifications as seed. Once parsed, objects retain as annotations some of their graphical features, like position and size. Moreover, our parser infers spatial relationships between the graphical elements in the imported drawings, which can be used to generate suitable graphical visualizations of the models obtained with mmXtens. Currently, we identify the following spatial relationships:

- *Containment*. This relationship appears when the bounds of a graphical object are within the bounds of another. An object might be contained within several others.
- *Adjacency*. Graphical objects can be adjacent horizontally (left/right) or vertically (up/down).
- *Overlapping*. This relationship is made explicit when two graphical objects are not in a containment relation, but they overlap. We distinguish left/right/up/down overlapping of the bounding box of the two objects.
- *Alignment*. We distinguish left/right/up/down alignment of graphical objects. Heuristically, this relationship is only signalled if objects are adjacent as well.

As an example, Fig. 10(a) shows a drawing made with the yED editor. Drawings are enriched with legends that assign a type name to graphical elements. Fig. 10(b) shows the legend for this example. From the drawing and the legend, our parser produces the model in Fig. 10(c), where the attributes starting with "@" are annotations storing the original graphical position (@x, @y) and size (@width, @height).

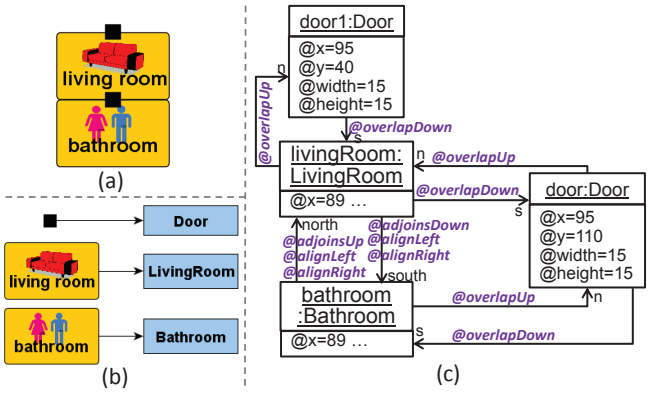


Figure 10: (a) Drawing. (b) Legend. (c) Extracted model with annotations for the graphical concrete syntax.

In the parsing process, associations in the abstract syntax (e.g., n/s) may become annotated with layout information derived from spatial relationships between the objects participating in the association. If two objects are in a certain spatial relation, it may mean they should be connected through a given association kind. In the example, as the top of the LivingRoom overlaps with the bottom of a Door, we connect both objects via n/s roles, and the connections get annotated as

@overlapUp/@overlapDown. The correspondence between spatial relations and associations can be done in two ways. The first one is by building the meta-model via example models [14]. In this case, the meta-model is induced from drawings provided by domain experts, and the detected spatial relations trigger the creation of meta-model associations annotated with the spatial relation. The second way is by annotating manually the meta-model associations with the spatial relations they should fulfil. When such spatial relations are found between two objects, the association gets instantiated.

### 5.2 Layouting models

In Section 4, we generated example models considering only their abstract syntax, which may be difficult to assess by domain experts. For this reason, we render the generated models as yED diagrams. Moreover, being able to validate examples using their concrete syntax is crucial for some DSVLs, like our running example, where spatial relations play a key role and some meta-model associations represent just these relations.

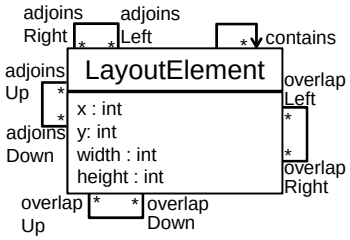


Figure 11: Simplified conceptual meta-model for layouting.

Fig. 11 shows the conceptual meta-model for layouting (alignment relations are supressed for simplicity). When our system finds an example model (in abstract syntax), we represent it as an instance of the layout meta-model, where the necessary spatial relationships are instantiated as well. The semantics of the spatial relations is expressed in OCL. Then, the challenge is computing a value for the attributes x, y, width and height of the different objects, compatible with the spatial constraints. Moreover, if part of the model to be layouted is a seed fragment extracted from a drawing, then, the size and position of the objects in the drawing should be preserved. In order to perform this computation, we use a model finder again. The resulting model is rendered as a yED file using the computed layout.

For the sake of illustration, Listing 15 shows the OCL-based semantics of the contains spatial relation.

```
1 context LayoutElement inv contains:
2   self.contains->forAll ( c |
3       self.x <= c.x and self.y >= c.x
4       and self.x+self.width >= c.x+c.width
5       and self.y+self.height >= c.y+c.height )
```

Listing 15: OCL semantics of containment.

Using this approach, unsound models like those in Fig. 9 do not get any visualization, thus signalling they have some problem. However, they are still valid according to the meta-model.

### 5.3 Inducing abstract syntax constraints from the concrete syntax

By inspecting models in concrete syntax, it may be easier to detect errors. For example, by trying to assign a layout to any model containing the fragments in Fig. 9, we realize that the spatial constrains they require are unsatisfiable. This means that the meta-model of the DSVL lacks extra constraints. In this section, we show that some of these constraints can be automatically derived from the spatial relations.

For example, assume the association roles north and south in the meta-model of Fig. 3 are tagged as adjoinsUp and adjoinsDown. Topologically, this means that the association needs to be acyclic. Similarly, assume roles east and west are tagged as adjoinsRight and adjoinsLeft, so they should be acyclic as well. Finally, these associations are also aligned vertically and horizontally, and annotated accordingly. Having this configuration means that Rooms can be placed in a grid. In such a case, we produce a compilation adding x, y attributes to class Room, as described at the end of Section 4.3.

Similarly, for a reference tagged as contains, we can produce OCL constraints analogous to those of *containment* references. However, the spatial containment we detect is *weaker* than meta-modelling containment, as graphically, an object can have two overlapping containers (i.e., two containers, one not contained in the other).

In our current implementation, this synthesis of OCL constraints for the meta-model is optional.

## 6. Tool support

mmXtens is integrated into metaBest [13], a tool for the integral V&V of meta-models, which also includes the mmSpec and mmUnit languages. The tool permits importing drawings into a textual format (Fig. 12(1)). These are seed fragments, serving as starting point for example model generation. Once imported, extension rules can be added.

The tool integrates the USE validator [10] for model finding. Hence, given an mmXtens specification, the tool compiles it into OCL, and passes the meta-model and the OCL to USE. metaBest permits setting a number of preferences for the solver, as shown in label 2 of the figure. When the solver is invoked (label 3), the user may select whether to generate: only the abstract syntax (MBF checkbox), only the concrete syntax (the abstract syntax is generated but discarded) or both. If a concrete syntax is requested, a legend like the one shown in Fig. 10(b) must be provided, identifying each object type with a graphical representation. Label 4 in the figure shows the generated model using concrete syntax and visualized in the yED editor.

## 7. Assessment

We have used mmSpec and mmXtens to assess the usefulness of our framework by validating the meta-models built by 26 undergraduate students as solution to the requirements in Section 3. To be able to use the English word checking

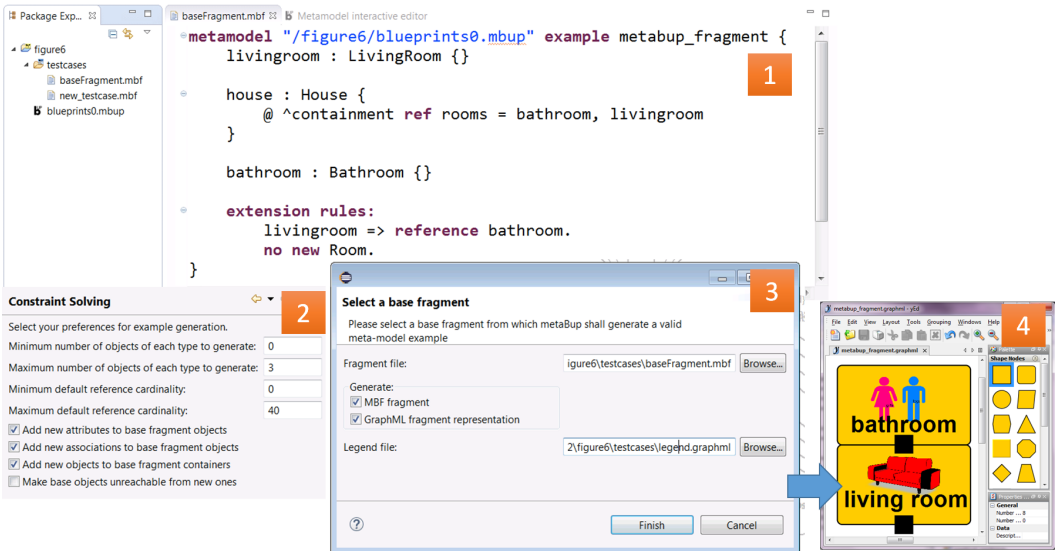


Figure 12: Meta-model example generation with metaBest.

primitives provided by mmSpec, we had to translate the name of the meta-model elements into English (as they were originally in Spanish). The translation preserved expressions and word choices, abbreviations and diacritic mistakes.

### 7.1 Assessing meta-model quality

We have used mmSpec to assess the quality of meta-models and analyse design choices for the DSL. The selected quality properties assess the fulfilment of good habits and patterns in the meta-model, independently of domain requirements. They are introduced in [12], and include design properties, best practices, metrics and naming conventions. The results show good average quality, with fail rates of 0.4% for design properties; 8.2% for best practices; 3.3% for metrics and 56.3% for naming conventions.

The best practice with highest fail rate is having two classes referring to each other with references that should be opposite but are not, most commonly between Wall and Room. This property is failed by 23% students. Reasonable thresholds for metrics are not violated, since meta-models are small. Finally, naming conventions where not consistently followed. For example, all meta-models contain names that do not follow camel-case, and 21 out of 26 do not use a noun-phrase name for non-boolean attributes.

In addition, we have designed specific mmSpec properties to automatically analyse the student solutions for some aspects of the DSL. For most aspects, we expressed possible solutions ranked from best (L1) to worst (L4).

- *How is the house/room containment relation modelled?* L1: Room types are subclasses of an abstract class, and the root house class has a single containment reference pointing to the abstract class. L2: Similar to L1, but the root class has a containment reference to each subclass. L3: Room types are identified with an enumeration. L4: Room types are identified with a string attribute.
- *How is room navigability modelled?* L1: By means of associations. L2: Using *ids* instead of associations. L3: Using compositions instead of associations.
- *Are doors represented (as attributes or classes)?*
- *Are windows represented (as attributes or classes)?*
- *How are light-spots and power outlets modelled?* L1: As attributes. L2: As classes.

Fig. 13 shows the result for each property. The choices for room containment were the most diverse. We were not able to predict a few solutions to room containment and the modelling of doors and windows. These cases were examined individually for determining their correctness.

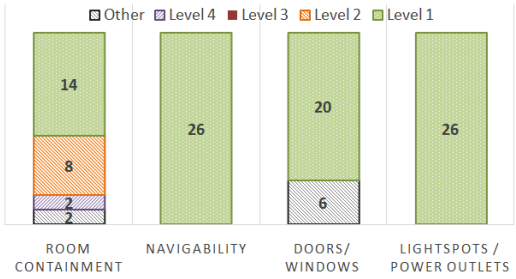


Figure 13: Level of compliance for domain properties.

### 7.2 Validating meta-model instances

We have used mmXtens to test to what extent the evaluated meta-models are able to produce valid models meeting the requirements. In particular, we used mmXtens to generate a model with only one house, and in case of failing, we

checked the reason for failure. Fig. 14 shows the results. 11% meta-models have satisfiability problems, with severe flaws like unfeasible association cardinalities or containment relationships that make impossible having instances, even if their OCL constraints are not considered.

There are two main types of errors for OCL constraints: syntax errors (31%) and unsatisfiable constraints (19%). The former include errors like missing brackets or wrong parameter types. The latter refer to incompatible constraints, so that making satisfiable one would make the other(s) unsatisfiable. These issues are hard to detect in a traditional approach, since one would have to produce models and detect errors manually. A small percentage (12%) of solutions had complex and numerous issues, demanding profound changes for being acceptable. Only 27% were able to produce examples. We have used those 7 solutions to make a series of tests using mmXtens to check their correctness regarding the domain of the problem.

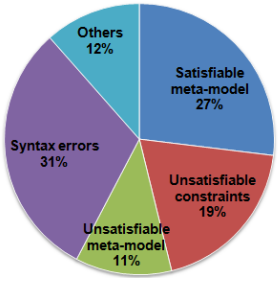


Figure 14: mmXtens results

Fig. 15 shows a student solution with two extension rules seeking the smallest valid house, and the generated model. The model has some flaws: both rooms have walls with the same orientation, while lacking others. Both rooms keep a logical north/south adjacency, but the involved walls hold two doors instead of sharing one. Finally, the living room has two windows on its south wall, and adjoins the bathroom in that direction, hence the windows are incorrectly placed.

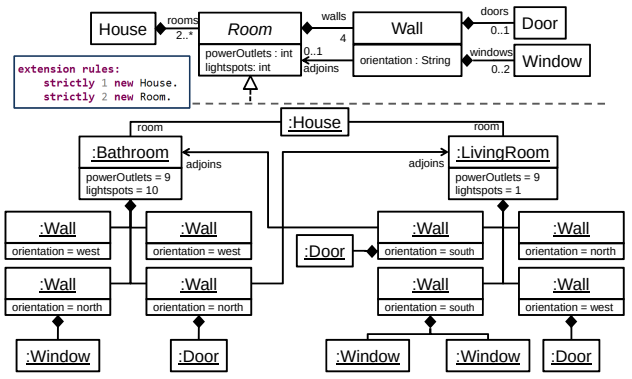


Figure 15: A student solution and a generated model.

We systematically encoded the problem requirements in mmXtens, evaluating the 7 solutions. The results are shown in Table 2. While most achieved the most basic requirement (Rq1), they fail for more complex ones. In all our experiments, mmXtens/USE had good performance, producing models within seconds.

In conclusion, we have seen that mmSpec was useful to evaluate quality properties of the meta-models, as well as to automatically evaluate the design solutions followed. With mmXtens we could evaluate the conformance of meta-models w.r.t. the problem statement. The advantage is that we did not have to create tentative examples manually. As most students failed in some way, we argue that the availability of a tool like mmXtens would have helped in obtaining higher quality solutions. However, this cannot be concluded from the present experiment, and we leave it to future work.

| Requirement | Correct sols. |
|---|---|
| 1. *Houses have at least a living room and a bathroom.* | 6 |
| 2. *Balconies can only be adjacent to a single room.* | 0 |
| 3. *Entries can be adjacent to three rooms at most.* | 1 |
| 4. *Only balconies can have windows in interior walls.* | 1 |
| 5. *A room at north/east of another is at south/west of the other.* | 3 |
| 6. *Houses have an entry door (at the entry if there is one).* | 2 |
| 7. *Rooms are topologically coherent.* | 0 |

Table 2: Solutions that fulfil the original requirements

## 8. Related work

In [15], the authors discuss the need for V&V in MDE, and identify the lack of comprehensive methods. The goal of our metaBest framework is providing such methods for DSL development. An analysis of the literature reveals three main approaches to meta-model V&V: *unit testing*, *specification-based testing* and *reverse testing*.

Works inspired by unit testing support the definition of test suites made of model fragments, and their validation w.r.t. a meta-model. For example, in [18], test models describe instances that the meta-model should accept or reject. Our own language mmUnit belongs to this category.

While unit testing proposals work at the model level, specification-based testing allows expressing desired properties of a meta-model. In this line, [20] presents an approach for checking meta-model integration, relying on OCL. However, as the authors recognise, using OCL to check meta-model properties is cumbersome, leading to complicated assertions and demanding expert technical knowledge. Our mmSpec language follows this approach, but is a DSL especially designed to express meta-model properties.

In reverse testing, the system produces artefacts, to be evaluated by the engineers. Most approaches are based on the automatic generation of instance models from a meta-model, likely using constraint solving [3, 5, 22], as we do in mmXtens. In these approaches, domain experts inspect the generated models to detect invalid ones, in which case the meta-model contains errors. This approach is followed by [4] (where the generated snapshots are targeted to test cardinality boundaries) and [6] (where a language to define object snapshots is proposed). In [11], the authors translate meta-models into ontologies, and then use reasoners to check the instantiability of the meta-models. This method was applied to the analysis of the ATL meta-model zoo. However, they do not use the method in the process of DSL development. On a more different approach, in [1], question-

naires with true/false questions are generated from the meta-model, and the domain experts perform the meta-model validation by answering the questionnaires.

The closest work to mmXtens is *Lightning* [5]. The tool is built atop *Alloy*, and supports defining the syntax and semantics of DSLs. For validation, it uses constraint solving to produce examples, but it does not include a user-friendly language (but Alloy itself) to describe their expected properties. Moreover, it does not support layout inference or graphical parsing, and is not integrated with other testing languages.

Regarding the inference and use of spatial relations in DSVLs, most recent (Eclipse-based) frameworks, like Sirius [19] or Eugenia [9], focus on graph-like languages. Their support for specifying and enforcing more advanced relations, like adjacency, overlapping or containment is limited. However, spatial relations play a crucial role for many visual languages (including our running example), as widely acknowledged in the visual languages community [2].

Hence, while there are approaches covering some aspect of meta-model testing, to our knowledge, metaBest is the most comprehensive one for DSVL V&V.

## 9. Conclusions and future work

We have presented an approach for the example-based validation of meta-models. This is done by means of a DSL, mmXtens, which permits describing seed fragments and extension rules to complete those examples by means of a model finder. The seed examples may come from informal drawings, which are parsed, and where spatial relations are extracted. These relationships are then used to provide a layout for the generated examples. The language has been integrated with two other DSLs for meta-model V&V, and implemented in a tool. The usefulness of the approach has been tested on a set of meta-models built by students.

In the future, we would like to extend mmXtens with more sophisticated primitives, e.g. for expressions regarding attribute values, or conditions on paths between objects. We will also generate visual environments for the DSVL, with the concrete syntax inferred from the example sketches.

## Acknowledgments

Work supported by the Spanish MINECO (TIN2011-24139 and TIN2014-52129-R), the R&D programme of the Madrid Region (S2013/ICE-3006), and the EU commission (FP7-ICT-2013-10, #611125).

## References

[1] A. Bertolino, G. D. Angelis, A. D. Sandro, and A. Sabetta. Is my model right? let me ask the expert. *Journal of Systems and Software*, 84(7):1089–1099, 2011.

[2] P. Bottoni and A. Grau. A suite of metamodels as a basis for a classification of visual languages. In *VL/HCC*, pages 83–90. IEEE Computer Society, 2004.

[3] 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. ACM, 2007.

[4] J. J. Cadavid, B. Baudry, and H. A. Sahraoui. Searching the boundaries of a modeling space to test metamodels. In *ICST*, pages 131–140. IEEE, 2012.

[5] L. Gammaitoni, P. Kelsen, and F. Mathey. Verifying modelling languages using lightning: a case study. In *MoDeVVa@MODELS*, volume 1235 of *CEUR*, pages 19–28, 2014.

[6] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.

[7] S. Kelly and R. Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, 2009.

[8] S. Kelly and J. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008.

[9] D. Kolovos, L. Rose, S. bin Abid, R. Paige, F. Polack, and G. Botterweck. Taming EMF and GMF using model transformation. In *MODELS*, volume 6394 of *LNCS*, pages 211–225. Springer, 2010.

[10] M. Kuhlmann and M. Gogolla. From UML and OCL to relational logic and back. In *MODELS*, volume 7590 of *LNCS*, pages 415–431. Springer, 2012.

[11] Y. Liu, S. Höglund, A. H. Khan, and I. Porres. A feasibility study on the validation of domain specific languages using owl 2 reasoners. In *TWOMDE*, volume 604 of *CEUR*, 2010.

[12] J. J. López-Fernández, E. Guerra, and J. de Lara. Assessing the quality of meta-models. In *MODEVVA*, volume 1235 of *CEUR*, pages 3–12, 2014.

[13] J. J. López-Fernández, E. Guerra, and J. de Lara. Meta-model validation and verification with MetaBest. In *ASE*, pages 831–834. ACM, 2014.

[14] J. J. López-Fernández, J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Example-driven meta-model development. *Software and System Modeling*, in press, 2014.

[15] J. Merilinna and J. Pärssinen. Verification and validation in the context of domain-specific modelling. In *DSM*, pages 9:1–9:6. ACM, 2010.

[16] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[17] OMG. OCL 2.4. `http://www.omg.org/spec/OCL/`, 2014.

[18] D. Sadilek and S. Weißleder. Testing metamodels. In *ECMFA*, volume 5095 of *LNCS*, pages 294–309. Springer, 2008.

[19] Sirius. `https://eclipse.org/sirius/`, 2015.

[20] S. Sobernig, B. Hoisl, and M. Strembeck. Requirements-driven testing of domain-specific core language models using scenarios. In *QSIC*, pages 163–172. IEEE, 2013.

[21] M. Voelter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. CreateSpace, 2013.

[22] H. Wu, R. Monahan, and J. F. Power. Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In *TASE*, pages 175–182. IEEE, 2013.