Generating Diverse Exercise Tasks on UML Class and Object Diagrams, Using Formalisations in Alloy

Marcellus Siegburg[1] Janis Voigtländer[2]

Abstract: Class diagrams are used to statically design object-oriented software. Object diagrams are a closely related concept. Understanding class diagrams and their connection to object diagrams is crucial for object-oriented software design. Hence, teaching of these concepts is an important part of many undergraduate CS curricula. We present support for such teaching via automatically generated exercise tasks, of different types, to be used in an e-learning setting. We use the Alloy specification language and analyser as crucial part of the generation process, via modelling the relevant concepts and meta-concepts inside Alloy.

Keywords: UML; E-Learning; Alloy

1 Introduction

Every year we teach large groups of undergraduate students in a course on modelling, with a bend towards software modelling. An important subject in the course is the Unified Modelling Language (UML) [Ob17], from class and object diagrams up to activity and state diagrams. We strive to increase the variety of tasks posed to students in the accompanying exercises, to help their understanding of the various concepts and their interplay. Moreover, we want to enable individualised learning, for example, by providing tasks of different difficulty levels. To achieve these goals, we would like to introduce specific e-learning tasks into the course, starting with exercises on class and object diagrams. In order to automatise the therefore required process of generating individual tasks and correcting submitted answers of learners, we need possibilities to specify tasks and verify answers.

We have previously created a prototypical task generator of the intended kind for exercises on conformance checking of object diagrams (ODs) with respect to class diagrams (CDs), which is available at `https://autotool.fmi.iw.uni-due.de/alloy-cd-od`. Its design, didactic considerations, and technical realisation are detailed in earlier joint work with an additional co-author [KSV20]. That task generator is based on a customisation and extension of the CD2Alloy approach [MRR11], using Alloy [Ja02; Ja11] to generate ODs which in a controlled fashion conform to, and/or do not conform to, specific choices from a given set of CDs. Together with a tailored and configurable random generator for those sets of CDs, it generates per student two somewhat similar CDs along with five ODs and asks

[1] `https://www.uni-due.de/fmi/siegburg_en`, marcellus.siegburg@uni-due.de
[2] `https://www.uni-due.de/fmi/voigtlaender_en`, janis.voigtlaender@uni-due.de

them to decide for each pair of CD and OD whether that OD is a valid instance for that CD. The concrete solution for each task instance is (pre-)determined, but of course kept hidden, a priori. That information can then be used when grading submissions.

The potential of this approach is far from being fully exploited yet. In this paper, we present new ideas for exercise task types on CDs and/or ODs that can be treated in a similar way, automatically generating and grading task instances. This involves both more exhaustive use of our earlier developed setup [KSV20] and completely new work on the Alloy side, in particular modelling the meta-theory of certain CDs (and allowed changes to them) themselves, rather than taking CDs as given and "just" modelling the conformance of ODs with respect to them.

2 New task type: match object links to class relationships

There is a lot of literature about UML [Bo05; Fo04; RJB04], so we will not provide here a complete introduction to UML and its diagram types. In fact, we are only considering simplified CDs and ODs here, eschewing various UML features.

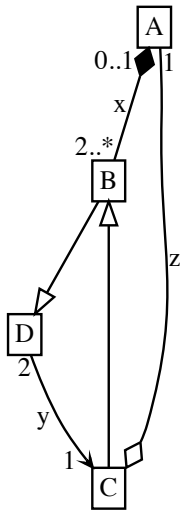


Fig. 1: A class diagram

Fig. 1 shows a CD as we will consider them in the following. Classes are drawn as rectangles containing their names. No attributes and methods are depicted, as they are irrelevant for our current task types. Classes are connected via relationships. This CD includes at least one relationship of each kind we take into account: inheritance, composition, aggregation, and association. Specifically, in this example *B* inherits from *D* and *C* inherits from *B*, *A* is a composition of *B*s, *C* an aggregation of *A*s, and there is an association going from *D* to *C*. Unless it is an inheritance, each relationship has a name which is written next to the edge. The numbers or ranges at the ends of these relationships are called multiplicities and constrain, in a way that students should learn among other things, how many object instances of one class may or have to participate in a given relationship with partners from another class. If no multiplicity is specified, certain default multiplicities apply.

Fig. 2 shows an OD that conforms to the CD in Fig. 1. Objects are drawn as rectangles containing the name of the (most specific) class they are an instance of and (unless they are anonymous) their own object name. The edges, called links in ODs, are annotated with the name of the composition, aggregation, or association they belong to. Actually, decisions on what information to include in depictions of each type of diagram can be made under didactic and task specific considerations. For example, the task type from our earlier work [KSV20] does not indicate the navigation directions of links in ODs (and neither does it depict navigation arrows on association relationships in CDs), but for the new task type to

be discussed in this section, we do (see comments further below). In this context, note that for aggregations (and likewise for compositions) our default used is that they are navigable from the "part" towards the "whole". That explains, for example, why the z-labelled link in Fig. 2 is directed from the A object to the C object.

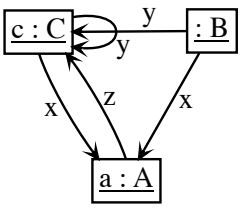


Fig. 2: An object diagram

Now to our first new task type. Similarly to our pre-existing task type, students should develop and demonstrate an understanding of the concept of links in ODs as instances of (certain) relationships in CDs. They have to identify which group of links within a given OD belongs to which relationship in a given CD. In contrast to our pre-existing task type, students do not need to decide whether a given OD is a valid instance of a given CD here. Instead, they are informed that the OD is a valid instance for that CD, and their task is to work out a matching that explains the conformance. Of course, the answer will not be readily available by just inspection of edge labels.

A generator for these matching tasks just described can be based very directly on the existing setup for "decide conformance in the first place" tasks. For those tasks, we already needed to be able to generate random CDs along with conforming ODs, plus non-conforming ones. Now we are only interested in the conforming ODs, not the non-conforming ones, and we invest some extra effort to make the link-relationship-matching task interesting and useful. An example output is the pair of the CD in Fig. 3 and the OD in Fig. 2. The former is in fact simply an edge-relabelled version of Fig. 1. That is also the general idea here: CDs are randomly generated within configurable settings for numbers of classes and relationships of every kind, and with multiplicity annotations drawn from a fixed set of choices. These CDs are checked for structural validity in the sense of UML, i. e. not containing inheritance or composition cycles, and in case of violations are discarded and new CDs generated. Some further restrictions apply, guided by teaching considerations, like that CDs in these tasks should not contain any double relationships between any pair of classes, that there should be no self-relationships, and no multiple inheritances. OD instances for the resulting CDs are generated via Alloy model instance finding, using the CD2Alloy transformation [MRR11] under the hood. And finally, the original CDs are edge-relabelled to obfuscate the exact correspondence of OD links to CD relationships. Actually, it is not exactly as simple as just summarised, for example since we want to guarantee that there is only one correct solution for each task (not, for example, some accidental additional solution matches caused by certain symmetries in a generated CD).

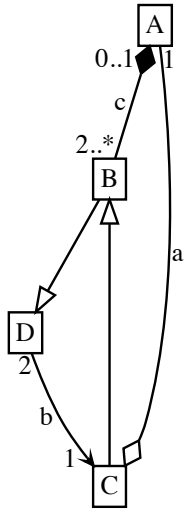


Fig. 3: Class diagram relabelled from Fig. 1

More precisely, we perform the following steps to generate a link-relationship-matching

task, while using CD2Alloy and Alloy's own ability to look for model instances that are selectively valid/invalid for specifically determined CDs.

1. Create a valid CD0 randomly.
2. Create a list of all non-inheritance relationship names in CD0.
3. Replace all names from that list with fresh names.
4. Create all permutations of this new list in order to create all possible mappings from the original names to the fresh names.
5. Create all CD variants CD1, . . . , CDn obtainable by applying these mappings as renamings over CD0. Arrange it so that CD1 stems from a randomly chosen permutation among all of them created in the previous step.
6. Turn all these CDs into Alloy formulas $cd_1, \ldots, cd_n$ using the CD2Alloy transformation, then send them to the Alloy analyser along with the following run command: $cd_1 \wedge \neg cd_2 \wedge \neg cd_3 \wedge \cdots \wedge \neg cd_n$.
7. From the OD instances returned by Alloy (if any exist; otherwise start over with step 1), choose one randomly. Display it along with CD0.

Unlike in our pre-existing task type [KSV20], we decided to print all navigation directions in CDs and ODs here. Not printing them would allow ambiguous/different renaming mappings in some cases or lead to a search space explosion when searching for task instances. This is because the Alloy library code that constrains the search for OD instances respects the navigation directions. Consider, for example, Fig. 4 under the assumption that navigation directions would not be provided in either CD or OD. Both, matching `b` to `y` and matching `b` to `z` would be correct options, because of symmetry in the graph. If we would like to rule out such occurrences while not displaying navigation direction arrows, we would at least have to additionally consider all flippings of navigation directions and take the resulting CDs/formulas additionally into account in step 6 above.

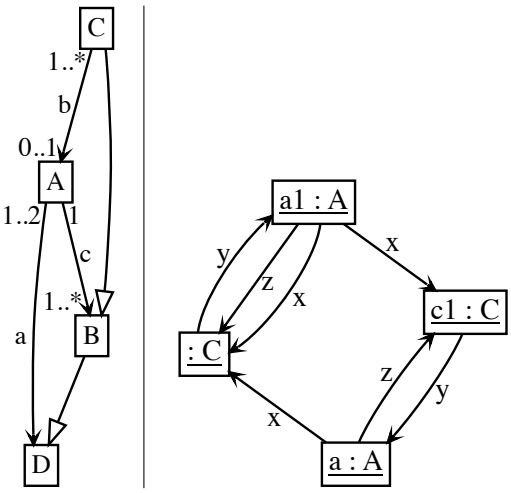


Fig. 4: Matching links from the OD (right) to associations from the CD (left) would be problematic when not providing navigation directions in this case.

3 Technical improvement: generating class diagrams using Alloy

Next, we describe new work on the Alloy side of the approach used so far, an extension that is useful in two ways. On the one hand, it allows us to move more work of task generators from the domain of programming (in our case, Haskell programming) to the domain of modelling (in Alloy). For example, step 1 in the previous section is currently performed programmatically. Basically, a generate-and-test loop is used for generating a single random CD0: throw dices to create some random instance of a data type representing class diagram graphs, then perform all kinds of structural checks (looking for inheritance cycles etc.) via Haskell code we have written, and in case of violations throw the dices anew. Expressing the meta-theory of CDs in Alloy instead, along with the UML standard's constraints as well as our own didactically motivated ones, allows us to avoid that generate-and-test loop, and even gain more confidence in the CDs' adherence to our wishes.

On the other hand, as a second way in which moving the generation of CDs to Alloy is useful, it will allow us to produce two further completely new exercise task generators, for not previously considered task types, in the next two sections.

To shed at least some light on our new CD generator, we have to give up the mostly black box perspective on Alloy employed so far. Alloy is a model checker providing a declarative language with first-order logic and expressions for relations. In List. 1 there is an extract of our new Alloy code. For example, a relationship is defined as a connection from exactly one class to exactly one class. The definition in Alloy actually defines two relations `from` and `to`, which both are relations from `Relationship` to `Class`. Inheritances are a specific kind of relationship. There is a predicate which states that there is no cycle for a given set of inheritances. This predicate holds if there is no class for which it is possible to get back to itself by only following edges of the given set. A detailed explanation of the operations on the relations is as follows: Applying transpose (~) reverses a relation; here the `from`-relation is flipped, i. e. is turned from a `Relationship -> Class` into a `Class -> Relationship` relation. The range (`:>`) and domain (`<:`) restriction operators filter range and domain, respectively. The join operator (`.`) joins two relations into one, thus in this case composing the `Class -> Relationship` and `Relationship -> Class` relations into a `Class -> Class` relation. Finally, applying the `^` operator calculates the transitive closure.

Further Alloy predicates we defined express all the other properties that our CDs shall satisfy. Moreover, for use in the next two sections (and possibly for improvements to our earlier work [KSV20], where such mutations are also relevant, but performed programmatically), we explicitly model certain change operations/mutations on CDs as conceptual units that may remove or add a relationship. More precisely, these changes may be either

- adding a relationship,
- removing a relationship,
- flipping a relationship,
- changing a relationship kind, e. g. turning a composition into an inheritance,

```
abstract sig Class {}

abstract sig Relationship {
  from : one Class,
  to : one Class
}

...

sig Inheritance extends Relationship {}

...

pred noInheritanceCycles [is : set Inheritance] {
  no c : Class | c in c.^((~ from :> is) . (is <: to))
}

...
```

Listing 1: An extract of the Alloy class diagram generator

- extending or shrinking the range of either a from- or a to-multiplicity,
- or shifting the range of either a from- or a to-multiplicity.

Finally, there is a comprehensive predicate `classDiagram` for use in generating CD instances, which enforces several properties for sets of relationships in a configurable way. Using, for example, `classDiagram[..., 0, 0, 0, False, False, False, False, False, none]` (where `...` is replaced accordingly by sets of relationships), a CD can be created that satisfies our base expectations for valid CDs as mentioned in the previous section. That is by virtue of all the `0`/`False`/`none` parameter values passed above basically turning off all kind of often undesirable properties (for positive CD examples), namely correspondingly meaning:

- the number of relationships with wrong multiplicities for compositions near the diamond,
- the number of relationships with other wrong multiplicities,
- the number of self-relationships,
- whether there are double relationships,
- whether there are reverse relationships,
- whether there are multiple inheritances,
- whether there are inheritance cycles,
- whether there are composition cycles,

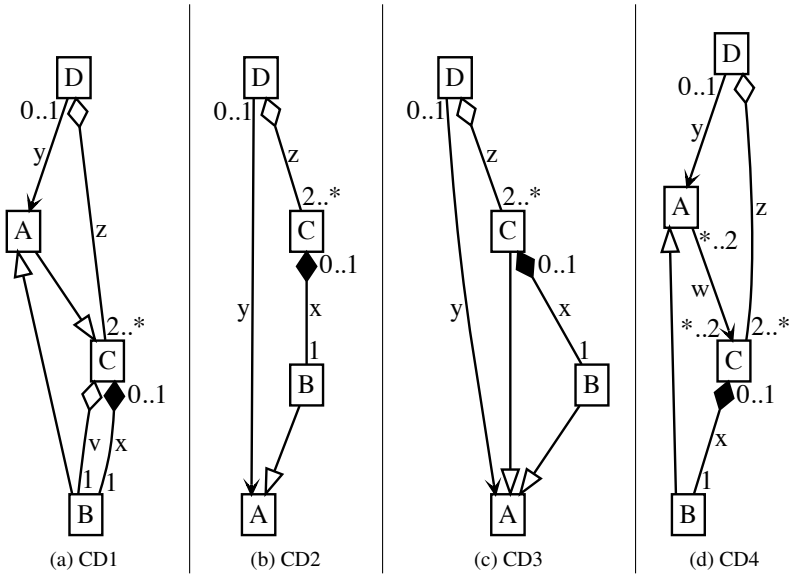
- whether there are "thick edges" (or parameter value `none` instead of either of `True`/`False` if it is irrelevant whether there are such edges).

The so-called "thick edges" are relationships which, due to a subtle interplay with inheritance, possibly lead to surprising multiple links between two objects, or to self-links, within OD instances. They were closely controlled in the task generator of Kafa et al. [KSV20] for didactic reasons discussed in that work.

Use of the new Alloy formalisation work in the context of new task generators is discussed in the next two sections.

4 New task type: decide validity of class diagrams

The new task type here is to focus on the validity of CDs themselves (rather than conformance of ODs to CDs as earlier). The idea is that students should develop an understanding of structural constraints that UML CDs must satisfy. To this end, we present to the students different CDs and let them decide for each of them whether it is valid or invalid, in the UML sense. One instance of such a task is shown in Fig. 5. For these specific CDs, students should declare CD1 and CD4 as being invalid; CD1 as it contains a composition cycle (via inheritance), and CD4 because there is an association having wrongly formed multiplicities. In contrast, both CD2 and CD3 are valid.



(a) CD1 (b) CD2 (c) CD3 (d) CD4

Fig. 5: A set of valid and invalid class diagrams

The CDs of Fig. 5 look structurally similar, and that is not by chance. Rather, they were deliberately created to be similar, by obtaining them via small changes applied to one original

```
...

abstract sig Change {
  add : lone Relationship,
  remove : lone Relationship
}

...
```

Listing 2: Definition of changes within the Alloy class diagram generator code

CD0, which is depicted in Fig. 6. The motivation here is that by presenting somewhat similar CDs of which some are valid and some are not, students are encouraged to think about what fine-grained characteristics *make* a CD valid or not, instead of performing a kind of guessing game on one specific CD as a whole. That is also the reason for the task being designed as presenting a whole set of CDs at once in the first place, instead of presenting only one CD in isolation, then another one (possibly completely unrelated to the first one) in another task instance, etc.

Creating such small changes to be applied to a CD is actually part of our Alloy CD generator that we provided some glimpses of in Sect. 3. Specifically, List. 2 presents another extract of our Alloy code. There, a change is defined as consisting of at most one relationship which will be added and at most one relationship which will be removed. We have predicates constraining the changes, for instance we prohibit no-operations (i. e. removing and adding the same relationship) and two non-identical changes being equal. Taken together, our definitions and constraints cover exactly all the possible mutations listed in Sect. 3 before the discussion of predicate `classDiagram`.

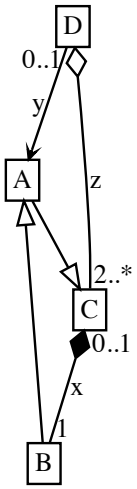


Fig. 6: An invalid class diagram

Of course, just applying changes to a CD is only half of the story. We also need to determine which CDs are valid and which are not. To that end, said predicate `classDiagram` from the previous section can be used. While we classify the CDs accordingly, we ignore the property concerning "thick edges", thus keeping it as `none`. CD1 is classified as `classDiagram[..., 0, 0, 0, True, False, False, False, True, none]` because it contains double relationships (from *B* to *C*, one aggregation and one composition) and a composition cycle (which arises because *B* inherits from *A* which inherits from *C* and thus allows any *B* being composed of itself or another *B*). But all the multiplicities are correct, there are no self-relationships, reverse relationships, multiple inheritances or inheritance cycles. CD2 and CD3 are classified as `classDiagram[..., 0, 0, 0, False, False, False, False,`

`False, none]`; they are valid. Note that the inheritances in CD3 cannot lead to a composition cycle, as no class is inheriting the composition. CD4 contains a single relationship with wrong multiplicities (`*..2`). Valid multiplicities are either a single integer value or a range. A range requires an absolute value as lower bound (first value) and a higher number as upper bound (second value). Only the upper bound may be set to unbounded (`*`). Besides the multiplicities, CD4 is valid, and it is classified as `classDiagram[..., 0, 1, 0, False, False, False, False, False, none]`.

Of course, our aim is not really to use the `classDiagram` predicate to classify CDs after the fact. Instead, we want to generate CDs with the express purpose of being valid or invalid in specific ways, i. e., due to specific properties or non-properties. That is easy to achieve with Alloy, by imposing `classDiagram` with specific parameter value combinations during the generation of (changes to) CDs. To provide appropriate control, we explicitly model differences between two possible choices of parameter value combinations for `classDiagram`, that is, not only properties but property modifications, which we call *structural weakenings*. Some details are provided in the next section, which introduces another new task type, with which the task type from the current section shares most of its implementation (but makes different choices about what is displayed to students and the kind of expected answer).

5 New task type: correct a class diagram via selecting changes

The new task type here is again about the validity of CDs themselves. But instead of posing students basically Yes/No questions (as in Sect. 4), we want them to figure out more tangible information (as in Sect. 2). In the current case, they will be asked to explain how an invalid CD could be turned valid. For example, given a composition cycle, removing one composition relationship would provide such a fix. Generally, the task will be to select, from a given set of suggested changes, all applicable options that would actually lead to a valid CD from a given invalid one. When solving such a task, students have to identify what is wrong in the given CD and whether the changes that are offered eliminate the error. As a small trap, we will sometimes offer changes which eliminate the existing but introduce a new error.

As a concrete example, Fig. 6 from the previous section presents an invalid CD as created by our new generator.[3] The change operations that are offered, and may be chosen by students for fixing the given CD, are:

1. add an aggregation for *C* of *B*s where *C* participates `0..*` times and *B* participates exactly once
2. remove an inheritance where *A* inherits from C
3. replace an inheritance where *A* inherits from *C* by an inheritance where *C* inherits from *A*

[3] Indeed, that CD0 would be classified as `classDiagram[..., 0, 0, 0, False, False, False, False, True, none]`; it contains the same composition cycle as CD1 discussed in the previous section.

4. replace an inheritance where *A* inherits from *C* by an association from *A* to *C* where *A* participates `*..2` times and *C* participates `*..2` times

Of course, these change operations were automatically generated alongside the CD. Applying these changes would result in the CDs shown in Fig. 5, but for this task type here we do not provide those to students.

Considerations the students thus have to perform now go somewhat along the following lines: The CD here contains an invalid composition cycle, which arises because *B* inherits from *A* which inherits from *C* and thus would allow any *B* being composed of itself or another B. This cycle may only be broken by changing any of the edges involved, i. e. the composition or one of the inheritances. Changes 2 and 3 both make this CD valid; the former because it removes the inheritance and the latter because flipping the inheritance between *A* and *C* breaks the cycle as well. Flipping an inheritance could have introduced a new error; for instance flipping the inheritance between *B* and *A* would have created a composition cycle again, in this case enabling any *A* to be composed of itself or another *A*; but flipping the inheritance between *A* and *C* is actually a good move. Change 4 also introduces another error, because the newly created association would contain an invalid range for its upper and lower multiplicities (`*..2`). Change 1 does not address any of the problematic edges. That is why it does not contribute to making the CD valid. So changes 2 and 3 are what students should select here. These correct answers are also produced by our generator along with the task instance, and can thus be used for comparing them against what a student answers.

Now about the implementation. In the previous section we already talked about properties and non-properties of CDs and how they are captured by parameter values of the `classDiagram` predicate. In order to generate these parameters, we define a set of default properties *dp* as `classDiagram[..., 0, 0, 0, False, False, False, False, False, none]` and *structural weakenings*, which we categorise into legal and illegal weakenings (see Tab. 1). Applying any illegal weakening on a property set results in an illegal CD. By applying possibly multiple structural weakenings on *dp*, we are therefore able to constrain CDs while also being able to decide if the resulting CD is valid. Having the structural weakening functions 2. for adding a relationship with wrong multiplicities, 4. for forcing composition cycles, and c. for forcing double relationships, we are able to express the properties of our example CDs (discussed above and in the previous section) based on *dp* as follows: CD1 has the properties gained by applying c. and 4. on *dp*, CD2 and CD3 have the properties *dp*, CD4 has the properties gained by applying 2. on *dp*, and CD0 has the properties gained by applying 4. on *dp*.

Overall, our new task generator here performs the following steps each time it is asked to create a task instance.

1. Choose randomly an illegal structural weakening *isw*, two legal structural weakenings lsw_1 and lsw_2, and another, either legal or illegal, structural weakening *sw*.

illegal weakenings	legal weakenings
1. have one more relationship with wrong multiplicities for compositions near the diamond	a. change no property
2. have one more relationship with other wrong multiplicities	b. have one more self-relationship
3. force presence of at least one inheritance cycle	c. force presence of at least one double relationship
4. force presence of at least one composition cycle	d. force presence of at least one reverse relationship

Tab. 1: Categorisation of structural weakenings

2. Choose randomly two of the property sets dp, $sw(dp)$, $lsw_2(dp)$, $lsw_2(dp)$, and $isw(dp)$.
3. Call the Alloy library introduced in Sect. 3 with $isw(dp)$ for the CD, and ask it to come up with four changes defined on this CD resulting, respectively, in a CD for which the first and second property set chosen in step 2, and $lsw_1(isw(dp))$, and dp hold.
4. Select any of the (not yet chosen) retrieved instances; if no instances are left, go back to step 1.
5. Check for all relevant valid CDs in the selected instance that there are inhabited ODs available, by using CD2Alloy.
6. If step 5 succeeded, print the original CD and verbal descriptions of all changes in the instance, as well as (hidden to students, but to be used in correcting submissions) Booleans indicating which of the changes result in a valid CD. If step 5 did not succeed, go back to step 4.

Not all combinations of two from dp, $sw(dp)$, $lsw_2(dp)$, $lsw_2(dp)$, and $isw(dp)$ result in an invalid CD with according modifications to the enforced property sets. Providing several options in step 2 therefore broadens the possibilities for valid sets of structural weakenings for the same properties of the basic CD. By performing the check for available ODs (step 5), we ensure that resulting CDs are not only structurally valid, but also have at least one inhabited OD instance.

By configuring the generator to deliver the resulting CDs after applying all offered changes, we generate tasks of the type introduced in the previous section.

6 Conclusion and future work

There are plenty of possibilities for generating exercise tasks of different kinds by using Alloy libraries for checking validity of CDs and generating ODs for (valid) CDs. We demonstrated that the latter alone enabled generating tasks for matching of OD links to CD relationships, in addition to the pre-existing task generator on checking conformance of ODs to CDs. Moreover, switching the generation and modification of CDs themselves over from a programmatic setup to a modelling setup using Alloy, enabled us to come up

with two new task generators addressing the understanding of valid and invalid CDs. More specifically, basically a single implementation resulted in generators for two different task types, as discussed in Sect. 4 and Sect. 5. First, deciding whether given CDs are valid or invalid. Second, choosing from a set of changes to an invalid CD those which result in a valid CD. There are certainly more types of tasks that may be generated by using only the tools provided so far.

The automatic layouting during the generation of graphics has some issues which we need to address. At the moment it happens sometimes that labels or multiplicities are hidden behind edges or are placed in between two edges, which could lead to misinterpretation.

The various task types are not yet systematically categorised by the skill required in order to solve them. We expect such a categorisation to be useful in order to identify skills not yet covered and, if applicable, create suitable new task types to cover those as well. In the current semester, we are preparing an optional exercise segment, using some of our task generators in an e-learning platform to provide students with task instances in order to get feedback (them and us). Eventually, we want to use the tasks during the main part of the next iteration of our modelling course.

References

[Bo05] Booch, G.: The unified modeling language user guide. Pearson Education India, 2005.

[Fo04] Fowler, M.: UML distilled: A brief guide to the standard object modeling language. Addison-Wesley Professional, 2004.

[Ja02] Jackson, D.: Alloy: A Lightweight Object Modelling Notation. ACM Transactions on Software Engineering and Methodology (TOSEM) 11/2, pp. 256–290, 2002.

[Ja11] Jackson, D.: Software Abstractions – Logic, Language, and Analysis, Revised edition. MIT Press, 2011.

[KSV20] Kafa, V.; Siegburg, M.; Voigtländer, J.: Exercise Task Generation for UML Class/Object Diagrams, via Alloy Model Instance Finding. In: ICT Education. Springer, pp. 112–128, 2020.

[MRR11] Maoz, S.; Ringert, J. O.; Rumpe, B.: CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In: Model Driven Engineering Languages and Systems, Proceedings. Vol. 6981. LNCS, Springer, pp. 592–607, 2011.

[Ob17] Object Management Group: Unified Modeling Language (OMG UML), Version 2.5.1, Dec. 2017.

[RJB04] Rumbaugh, J.; Jacobson, I.; Booch, G.: The unified modeling language reference manual. Pearson Higher Education, 2004.