Exercise Task Generation for UML Class/Object Diagrams, via Alloy Model Instance Finding

Violet Kafa, Marcellus Siegburg, and Janis Voigtländer(✉)

University of Duisburg-Essen, Duisburg, Germany
janis.voigtlaender@uni-due.de

Abstract. The Unified Modelling Language (UML) is the standard for designing and documenting object-oriented software systems. Its most frequent use is for static modelling in the form of class diagrams. A correlated concept is that of object diagrams. An object diagram may or may not adhere to a given class diagram, and the understanding of this connection is key to correctly using class diagrams in practice. We present an approach for automatic generation of verified, non-trivial, conceptually relevant examples and counterexamples of class/object diagram combinations, aimed at providing exercise tasks in a university course setting. The underlying technique is model instance finding using the Alloy specification language and analyser. We provide an implementation of our approach in an e-learning system.

Keywords: E-learning · UML · Alloy

1 Introduction

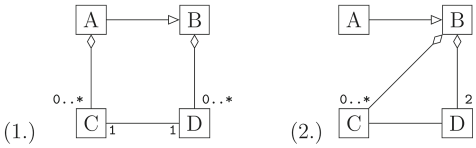
The Unified Modelling Language (UML) [10] is widely used in the software industry and academia. It is a standard for specifying, visualising, constructing and documenting artifacts of object-oriented systems and has a rich set of diagrammatic notations along with their well-formedness rules. The language of class diagrams (CDs) and object diagrams (ODs) is part of the UML standard and supported by many commercial and academic software modelling tools. These specific diagrams are also a typical subject matter in a software modelling course at university.

To facilitate learning and understanding of the CD and OD concepts, it is desirable to confront students with many and diverse examples and counterexamples. For instance, a useful exercise task in a software modelling course is to present a certain number n of CDs and a certain number m of ODs and ask students to determine for each combination of CD and OD whether the latter is a correct instance of the former or not, along with explanation of the reasons (such as possible violations of multiplicities or other constraints). Fig. 1 shows a hand-crafted exercise task of this kind, with $n = 2$ and $m = 5$, used in a concrete course in the past. Our undertaking here is to develop tooling that helps

B. Tait et al. (Eds.): SACLA 2019, CCIS 1136, pp. 112–128, 2020.
https://doi.org/10.1007/978-3-030-35629-3_8

the instructor by systematically and automatically constructing similar exercise tasks.

Let the following class diagrams be given, each of which shows connections between the classes A, B, C and D.



Indicate, for each of the following object diagrams, whether it is valid for the above class diagrams (ten answers altogether). Where that is not the case according to you, explain why and give all reasons.

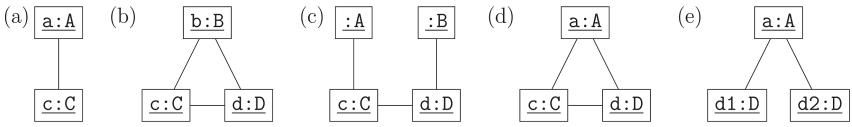


Fig. 1. A sample exercise task.

To that end, formal method techniques from the verification and model generation domain are employed. The basic idea is to randomly generate CDs subject to certain complexity constraints, and with a reduced feature set according to didactic considerations, and to use the Alloy specification language and analyser [5,6] to generate appropriate model instances and non-instances, to be presented as candidate ODs to the students. To make use of Alloy, we employ (and extend/revise) a translation from CDs to Alloy modules that was introduced in related work on CD analysis and OD generation [8]. For the generation of interesting counterexample ODs, which was not a topic of the mentioned CD2Alloy work, we devise a strategy that involves variations of the original CD, such as removing or adding some relationships, manipulating some multiplicities etc.

2 Background on UML's CDs and ODs

There is a vast literature on UML, including many textbooks introducing its various diagram types [2,4,11], so we will not provide another substantial introduction here. But we want to at least give motivating examples of a CD and a conforming OD, to illustrate which model elements these types of diagrams can contain, as well as to shortly discuss the relevant relationships. Moreover, we already briefly delineate what aspects we will *not* cover in our generated exercise tasks (more details then in the next section, about didactic considerations).

Fig. 2 shows a CD, illustrating the static design of a certain object-oriented system. In it, we see classes, some with attributes and operations/methods, an inheritance relationship, and additional relationships in the forms of compositions and general associations with attached multiplicities.

Fig. 3 shows an OD conforming to the CD from Fig. 2. Note that no operations/methods are present in ODs. In our exercise tasks we will actually cover neither attributes nor operations/methods, since we are more interested at this point in teaching in considering the relationships between classes/objects.

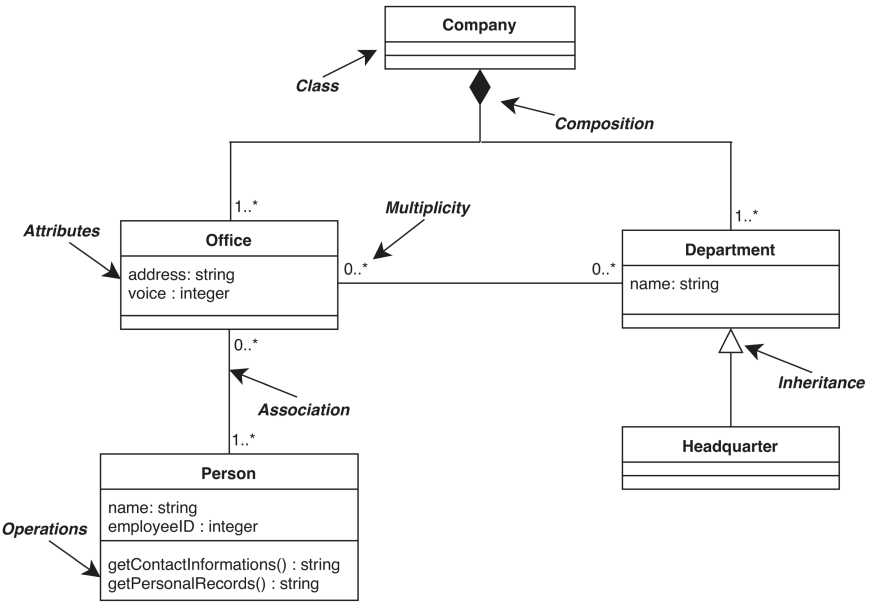


Fig. 2. A CD.

Inheritance between classes is not reflected on the object level by explicit connections/links between corresponding objects in any way. Instead, inheritance expresses that the child class has all the same relationships to other classes as the parent class has, and *that* will be reflected on the object level. For example, according to Fig. 2 every headquarter is a department, so in Fig. 3 we could have replaced **d1** by an object of type **Headquarter** and still let it have the link to **c**.

Association between classes, such as between offices and persons or between offices and departments in Fig. 2, is a broad term that encompasses just about any logical connection between classes. On the object level, associations are represented by links, such as between **o** and **p** in Fig. 3. Multiplicities at the ends of associations in a CD express how many objects of one class can be related to one object of the other class. In our example, each office needs to be linked to at least one person (due to the multiplicity 1..∗ at one end of the relevant association), but not each person needs to be linked to an office (due to the multiplicity 0..∗ at the other end). So in Fig. 3 we could not simply delete **p** (while keeping **o**), but we could add another object of type **Person** without adding any links. We consider 0..∗ the default multiplicity, so it will not always be depicted (see Fig. 1).

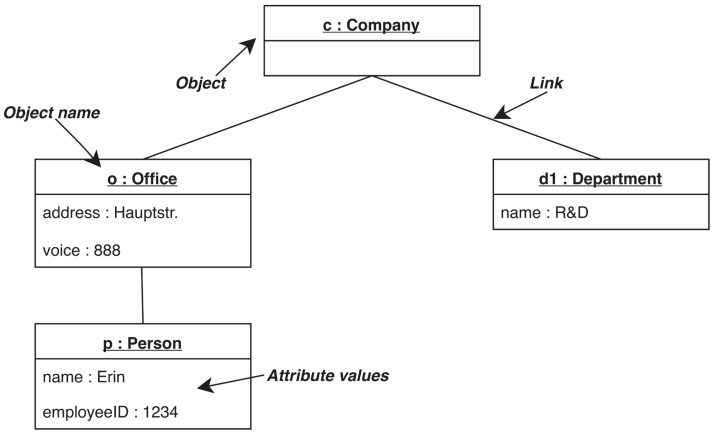


Fig. 3. An OD.

Another kind of relationship is aggregation. It does not appear in Fig. 2, but in Fig. 1, see the hollow diamond shapes there. While from a modelling perspective aggregations are special, in that they are intended to represent whole/part relationships, from a formal perspective concerning conformance of ODs to CDs, aggregations are to be handled just like associations. Their default multiplicities are also as for associations.

Finally, a composition relationship is a stronger form of aggregation which actually comes with additional constraints in considering conformance of an OD to a CD. Namely, every "part" must be linked to at most one "whole". The example in Fig. 2 has a composition relationship between companies and offices, and another one between companies and departments, with companies playing the role of "whole" in both cases (as marked by the filled diamond shape). So in Fig. 3 there could be additional companies, but they could not also be linked to the existing objects **o** and **d1**. Instead, there would have to be additional offices and departments present, due to the multiplicity 1..∗ at the "part" end of both composition relationships. The default multiplicity at those ends would have again been 0..∗, while at the "whole" end of composition relationships the default multiplicity is 1..1 (which would be written simply as 1) and actually only the multiplicities 0..1 and 1..1 are allowed there at all.

Besides the fact that we will not include attributes or operations/methods in our generated examples (manifesting in depictions as in Fig. 1, each class and object being just a simple box with name and/or type inscribed, instead of additional compartments in the boxes as in Figs. 2 and 3), another difference from what we have seen in the current section is that the generated examples will be artificial, with class and object names like **A**, **B**, **c**, **d**, instead of real world notions like **Company** etc. The reason is that we want to use the generated exercise tasks for teaching the formal concepts of the CD and OD language, and the correct interplay between model elements, emphasising the similarities

and differences between the relationship flavours considered. Thus enabled, free modelling set in real world scenarios is part of separate activities in the course.

3 Didactic Considerations

What makes a good generator for exercise tasks of the kind considered? We would like to be able to generate many different, but analogous tasks, in order to provide students with ample opportunities for practising without repeated or predictable instances/solutions. We want to be able to control the complexity and difficulty of tasks, for example to enable a transition from simple, preparatory instances to more challenging ones, or to level the field for student groups from different backgrounds (e.g., ones that have already had an object-oriented programming course and thus know some UML concepts at least from a programming language perspective, and ones for which this is not the case). There are additional dimensions along which we would like to be able to parametrise the task generation. For example, at a certain teaching stage we might want to tailor tasks to focus on one specific concept ("give us only tasks in which the correct understanding of aggregation makes the difference between right and wrong, despite other relationships also being present") or to exclude some concept ("do not produce tasks with occurrences of composition relationships, because we have not yet covered that in the lecture"). Generated tasks should be non-trivial and interesting, in the sense that there is really something to discover in them. For example, if the task is to decide for a CD and two ODs which one of the latter two conforms to the CD and which one does not, then it should not be the case that one of the two ODs is obviously far off from possibly having anything to do with the given CD, thus making the question boring and uninstructive. Instead, we should have a degree of control over how far off a counterexample OD is allowed to be from a positive example. And while it might be obvious, it is crucial to ensure that the generated exercise tasks are correct. If we consider a certain OD to be (or to not be) conforming to a certain CD, and use that assessment in feedback to students, or for grading, then it better be the case that it holds true. When hand-crafting artificial tasks that try to emphasise a certain CD/OD language aspect as well as aiming to produce interesting and challenging instances, it can be surprisingly easy to violate this assurance by accidentally not taking into account some subtlety of the UML standard.

Besides the above general considerations, there are more concrete decisions to make about the design of the exercise tasks (generator). For example, in Figs. 1, 2, and 3, we have not annotated names for associations, aggregations, and compositions. UML does allow such annotations, and sometimes they – or some other means of distinguishing links – are needed to properly decide about conformance between CDs and ODs. As a simplistic example, consider the two CDs on the left in Fig. 4 and assume we want to provide an OD that conforms to the first CD, but not to the second CD. The OD on the left in Fig. 5 fits the bill. But if we omit association names, see the right halves of Figs. 4 and 5, this is not discernible anymore, because now the link between objects a and b could be accidentally perceived as stemming from the association between A and C

that B inherits in the second CD. The examples in Fig. 1 were carefully crafted to ensure that no such confusion exists, or put differently, that students always have a chance to puzzle out which association or aggregation a link corresponds to, even in the absence of names. But for our task generator we have decided to always provide those names, thus making the generated tasks more beginner friendly in that respect.

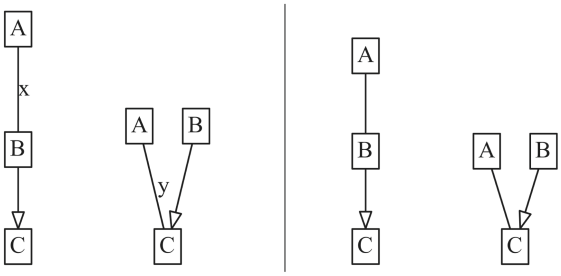


Fig. 4. CD examples with and without association names.

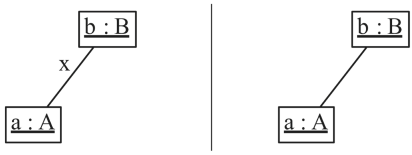


Fig. 5. OD examples with and without association names.

On the other hand, we do omit other annotations on associations, aggregations, and compositions: namely role names and navigation or reading directions. This also has an impact on the level of challenge the checking of conformance poses. For example, if we present the CD shown on the left in Fig. 6, then students are likely to be sceptical about conformance of an OD that has two y links between the same two objects of type A, as in the middle of Fig. 6. After all, even though A inherits from B and is thus allowed (since B is) to be linked with A via y, the double linkage seems strange and at least subtly at odds with the multiplicities written on the ends of y in the CD. If, however, we were to annotate direction arrows on the associations in the CD, and then present the OD also with direction arrows, as on the right in Fig. 6, then the situation would be less surprising (since the revelation "the two y-links go in opposite directions" makes things clearer), and thus possibly less of a challenge. The point here is, even with names provided on associations, aggregations, and compositions, there are still interesting bits to puzzle out by the learner.

We have made further decisions about the CDs and ODs to generate. For example, we do not allow multiple inheritance and we impose structural constraints

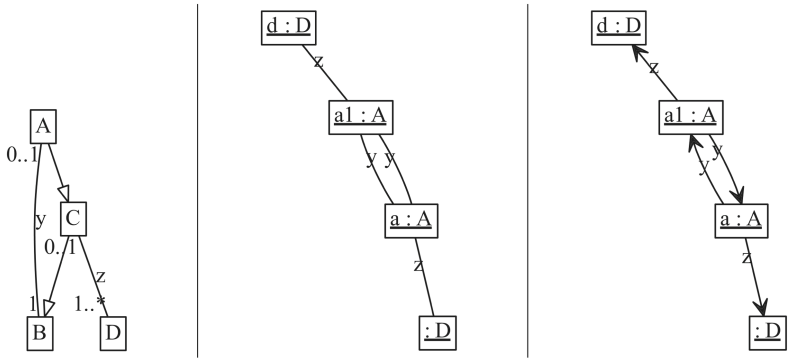


Fig. 6. A somewhat surprising case of conformance.

(such as no two associations between the same pair of classes in a CD) that are not mandated by the UML standard. These decisions are also driven by what aspects we want the learning to focus on, by experience with hand-crafted exercise tasks, and by trying to balance challenge and approachability of tasks. We might revise them, or the decisions about providing association names and/or other annotations, as we gain experience with automatically generated tasks. In any case, even within the frame of the decisions as currently set, we have enough means to produce tasks of varying difficulty, such as by parametrisation over the numbers of classes, inheritances, associations, . . . , objects, links.

4 Background on Alloy Usage

It is important that the exercise tasks we will generate automatically, and present to students without prior inspection of every instance by an instructor, are one hundred percent semantically correct. If the e-learning system assesses that a certain OD does, or does not, conform to a certain CD, then that should be guaranteed. Simply programming a random generator for CD/OD pairs that should serve as examples and counterexamples in an exercise task would risk introducing errors, even under best effort to truly implement what the UML standard implies. Moreover, using such a hand-written coupled generator, it would probably be difficult to tailor generated tasks for special, possibly changing needs, such as emphasis on certain model elements and diagram features. So instead we decided to take a more declarative, and at the same time verifiable, route. We build on the CD2Alloy work [8] that translates CDs to modules in Alloy, a specification language based on mathematical sets and relations and backed by SAT solving [6].

Compared to other translations from the UML domain to Alloy [1,9,12] (and in reverse, to interpret models found by the SAT solver back into UML as object diagrams), the CD2Alloy translation performs a deeper embedding of CD concepts into the Alloy logic. That is, instead of rather directly mapping CD concepts (classes, associations, . . .) to quite-similar-but-not-really-equivalent Alloy

concepts (signatures, fields, ...), the translation in some sense explicitly programs out the UML semantics as functions and logical predicates. That allows more accurate representation, capture of more UML features, and very importantly, analysis over more than one CD at a time. In particular the last aspect will be crucial for us here. The details of the embedding/translation are not of superior importance for the current work, but to at least provide a flavour, Fig. 7 shows excerpts of the Alloy module obtained for the CD shown on the left in Fig. 6. For example, the `fun` definitions essentially express that in any model the set of objects of type A is exactly the set of objects directly belonging to A (since A has no subclasses in the CD), while for example the set of objects of type B is the union of the sets of objects of B, C, and A (due to the inheritance relationships in the CD). And the lines involving `ObjLUAttrib` and `ObjLU` express the multiplicities at the two ends of association y in the CD, using predicates whose definitions are not shown in the excerpt. The translation rules are described in a technical report [7]. For our purposes here we use a subset of them (due to our restricted set of CD features used), and actually had to perform a few revisions/adaptations (to more exactly express some desired constraints). But the fundamental principles are as in the previous work.

```
...

one sig y extends FName {}

...

fun ASubsCD : set Obj {
  A
}
fun BSubsCD : set Obj {
  B + CSubsCD
}
fun CSubsCD : set Obj {
  C + ASubsCD
}

...

ObjLUAttrib[ASubsCD, y, BSubsCD, 1, 1]
ObjLU[BSubsCD, y, ASubsCD, 0, 1]

...
```

Fig. 7. A glimpse of Alloy code for the CD from Fig. 6.

An Alloy module as in Fig. 7 (completed) can be given to the Alloy analyser, which will try to find a model/instance, in a finite scope, and will return any ones existing in a textual form. That textual form can be interpreted as

describing ODs. In fact the ODs in the middle and right (depending on whether navigation directions are to be depicted or not) of Fig. 6 are thus obtained. An Eclipse plug-in exists that implements this whole workflow, thus for example allowing a software engineer to check whether a certain software design is meaningfully populated at all (by letting ODs be created for inspection, from the CD of a software system/component under development). Since our objective here is different, we need a different workflow/approach (see next section), but the CD2Alloy translation is a crucial ingredient.

It is worth pointing out that we are still using a hand-written random generator, but not for CD/OD pairs, just for CDs. That is, CDs are not generated by the Alloy analyser from some meta model. Instead, we have programmed a CD generator (see description in Sect. 6) that is driven by our didactic considerations about which features to support, which structural constraints (beyond those mandated by the UML standard) to adhere to, etc., see also relevant discussion in the previous section. That was a pragmatic decision and does not incur the risk painted in the first paragraph of the current section, concerning semantic correctness. After all, creating CDs essentially just means we need to get the syntax right. The semantics will be covered (and verified!) via Alloy.

5 Approach to Counterexample Generation

Being able to provide positive instances of ODs for a given CD is nice, but only at most half (actually much less) of what we need for our exercise task generation. We also need to be able to provide negative instances, counterexamples. One tempting approach, given the deep embedding of CD concepts into Alloy logic that CD2Alloy performs and which allows first-class use of all of Alloy's logic expressivity on top, would be to simply make use of logical negation. That is, given that the OD in Fig. 6 was obtained (along with many others) by calling the Alloy analyser on the module from Fig. 7 with command:

```
run { cd } for 4 Obj
```

where `cd` is the name of a predicate that builds on all the things introduced by the translation for the given CD (such as the `fun ASubsCD` definitions etc.), and where the `4` stands for at most how many objects should be present in the OD, a naive attempt at counterexample generation would be to instead call the Alloy analyser on the same module but with command:

```
run { not cd } for 4 Obj
```

Unfortunately, the results are of questionable value. Fig. 8 shows the OD-style rendition of the first of a multitude of instances found. That is indeed not an OD conforming to the CD from Fig. 6, but it is utterly useless for an exercise task, because it is off at first look. In fact, even if we were to somehow structurally constrain or filter out instances that seem too far off from being worth showing as solution candidates in an exercise task, for example by discarding everything

with perceivedly too many parallel edges, we would not necessarily be better served. Specifically, without additional precaution there lurks a potential deeper problem here, because the predicate `cd` conceptually expresses:

"[what Alloy finds as model] is an OD instance of the given CD"

and the logical negation of that (in formal or in natural language) happens to not be:

"is an OD that is not an instance of the given CD"

but instead simply:

"is not an OD instance of the given CD"

which actually means:

"is not an OD or is an OD that is not an instance of the given CD".

So it is not ruled out up front that `not cd` might produce structures that are not even proper ODs. We could dispel this concern by additional analysis, but that does still not necessarily, generally give us instances that would serve as *useful* counterexamples in an exercise task.

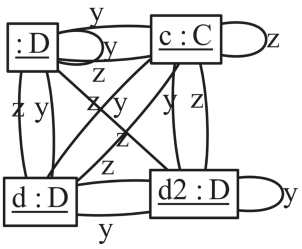


Fig. 8. Result of a naive attempt at counterexample generation.

Instead, we make use of a crucial advantage of the CD2Alloy translation/embedding over competing approaches, namely its ability to perform analysis over more than one CD at once. This ability was already briefly mentioned in the previous section and comes in handy now. The basic idea is that if we have a CD and want to produce a counterexample OD for it, we could look for a structure that is not just *not* an OD for *that* CD, but at the same time actually *is* an OD for a CD *similar* to, but somewhat mutated from, the original CD. That way, we can guarantee that we are not getting something wildly off, and we can even control in some sense how much and which kind of conceptual distance there will be between OD examples and counterexamples we present, by controlling the degree and character of mutation applied on the CD level. For example, if we want to emphasise/train the semantics of composition relationships, we could take an original CD that contains a composition, produce a new CD via a

mutation that specifically affects this composition (moving one of its endpoints to a different class, changing the multiplicity at one of its ends, or even turning the composition into an aggregation or other relationship), and then look for an OD that is an instance of the mutated CD, but not of the original CD. What we will get is an OD that is a counterexample for the original CD, and is so not by having nothing to do with that original CD at all, but instead more targetedly by something having to do with the composition relationship we touched. Of course, we would not tell the students that touching this composition relationship in the way we did is what happened in the background, in order to not disclose too much about what they should actually discover in the example/counterexample.

So our approach in its simplest form can be described as follows. We have a CD (randomly generated), and translate it into an Alloy module. The relevant all-encompassing predicate resulting from that could be called `cd1`.[1] We mutate the original CD to a similar one, and put that one through the CD2Alloy translation as well, resulting in an overall predicate `cd2`. By combining Alloy modules appropriately (there is some overlap between modules due to common definitions, so combining does not simply mean concatenating), we obtain a single one for which we can call verification commands like:

```
run { (not cd1) and cd2 }
```

or:

```
run { cd1 and (not cd2) }
```

Instances found by the former call correspond to relevant OD counterexamples for the original CD, as outlined in the previous paragraph. And instances found by the second call can also be used: as interesting positive examples for the original CD, or of course as counterexamples for the mutated CD, or indeed as part of cross check exercise tasks like the one in Fig. 1.

The approach as described above is sketched in Fig. 9. Actually, we use a more elaborate strategy, to be discussed in the next section. Here, let us just note that it is not hard to imagine that the combination approach generalises well to more than two CDs, so that we can consider instances that have certain positive or negative inhabitation properties regarding three or more CDs. That will be used to create more interesting exercise tasks.

As a side note, an alternative approach could conceivably have consisted of not mutating a CD, for which one or more conforming ODs are already given and counterexample ODs are still being sought, but instead performing mutation on the OD level, that is, turning a conforming OD into a counterexample OD by deleting a link or similar changes. However, we would then still have to check whether the obtained OD has all the desired characteristics relative to the given CD(s). After all, deleting a link may or may not turn an example into a

[1] That involves variants of the stuff shown and elided in Fig. 7, e.g., definitions `fun ASubsCD1` etc., while for the predicate `cd2` considered in a moment, separate variants `fun ASubsCD2` etc. would be produced.

counterexample. So after applying OD mutations we would have to separately establish that what we got is an OD that conforms to the CD(s) we want it to conform to and does not conform to the CD(s) we want it to not conform to. And if it turned out that we did not get what we wanted, we would have to try some other changes. Instead of such a search-and-check procedure, our declarative approach (expressing the desired characteristics in Alloy verification commands) is more targeted and directly gives us appropriate ODs, if they exist.

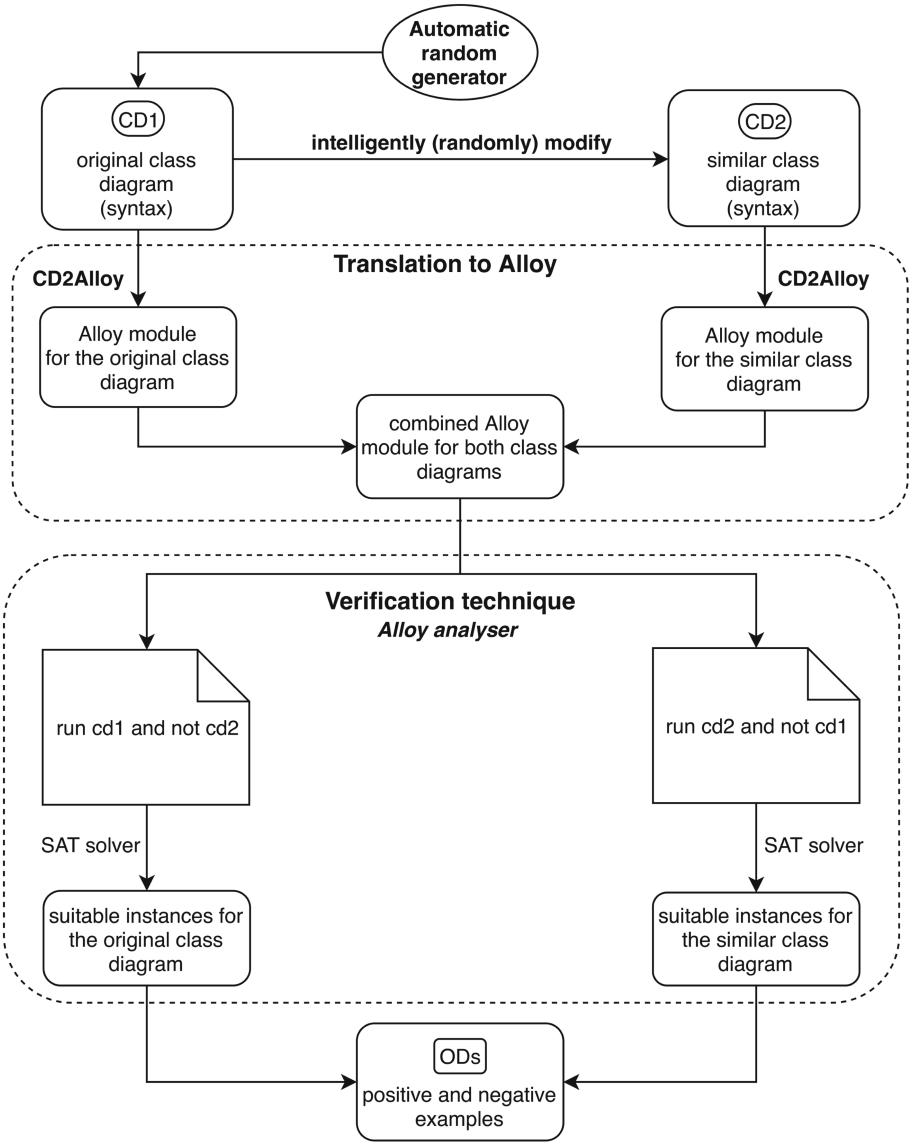


Fig. 9. Sketch of the approach in its simplest form.

6 Strategy and Implementation

As already mentioned, we are using a hand-written random generator for CD syntax without attributes or methods. The generation is parameterised by a configuration comprised of a minimum and maximum number each for classes, inheritance relationships, associations, aggregations, and compositions. The default configuration is: exactly four classes, between one and two inheritances, at most two associations, at most two aggregations, and at most one composition. The multiplicities used in the CD are drawn from a small set of choices: the default multiplicities and some special cases like 0..1, 0..2, 1..∗, but not arbitrary $n..m$. We do not create CDs with multiple inheritance (one class having two outgoing inheritance arrows) and impose some additional structural constraints:

- Classes have no self-relationships of any kind. That is, a class cannot inherit from itself, cannot have an association relationship to itself, etc. Note that this does not mean that objects cannot have self-links, in fact we will see the opposite at the end of this section.
- There is at most one relationship between the same pair of classes.
- There are no inheritance cycles and no directed composition cycles.

Some of these are actually already imposed by the UML standard, which we adhere to anyway and also beyond what is listed above.

When it comes to mutating a CD to another one, we randomly choose from the following operations (while preserving all constraints mentioned above):

- adding a new relationship between any two classes,
- removing an existing relationship,
- flipping the direction of an existing inheritance, aggregation, or composition relationship,
- exchanging an existing relationship with another relationship, e.g., turning an inheritance into an aggregation,
- changing an existing multiplicity by increasing, decreasing, or shifting its range.

Note that some of these have the character that a CD2 obtained from a CD1 can only ever have more (or only ever have fewer) OD instances than originally. In such cases, simply using `run {cd1 and not cd2}` and `run {cd2 and not cd1}` as sketched in Fig. 9 would not result in interesting exercise tasks, since at most one of these commands would actually produce instances. Possibilities to counter this include to perform more than one mutation and/or to involve more than two CDs. Another reason to involve at least a third CD is that we would also like to present ODs in an exercise task that do not conform to either of two given CDs.[2] But simply calling `run {not cd1 and not cd2}` is not a good idea, due to the same issues that led to Fig. 8. So a third predicate `cd3` should

[2] For example, can you spot which of the five ODs in Fig. 1 do not conform to either of the two CDs given there?

be involved, for yet another mutated CD. Actually, our exercise task generation strategy uses overall four CDs, and is described next.

We step through an explanation of our generation strategy in the remainder of this section, along with a concrete example. First of all, we create a random CD, see CD0 in Fig. 10. In this case, it so happens that CD0 does not contain any non-inheritance relationship. That limits which mutations are applicable in the next step here, since for example there are no multiplicities to change, but in general the next step chooses, twice, from the whole assortment of mutations listed above. We mutate CD0 into CD1, and CD0 into CD2, see again in Fig. 10. In this case, CD1 was obtained by adding an association, and CD2 by adding an aggregation elsewhere. The thick edge in CD1 will be explained in a short while; to students it will be shown as normal edge. In general, we now have CD1 and CD2 that are one or two mutations apart from each other.[3] We only continue if at least one of them contains a non-inheritance relationship; otherwise we start over with generating a new CD0. The motivation is that CD1 and CD2 will be the CDs included in the exercise task, and having them contain only inheritance relationships would not give interesting tasks. Next we mutate CD0 yet again, into CD3, see Fig. 10. This time, an inheritance was turned into an aggregation.

Now, Alloy is asked to find instances for all combinations of CD1/CD2 satisfied positively/negatively, with CD3 involved as a "safeguard" in the otherwise all negative case:

- `cd1 and not cd2`
- `cd2 and not cd1`
- `cd1 and cd2`
- `not cd1 and not cd2 and cd3`

The search is limited by a maximum number of objects allowed, our default being four (see earlier remarks on configurability). In addition, structural constraints on the ODs are possible. At the moment, we use the following Alloy code:

```
fact LimitIsolatedObjects {
  #Obj > mul[2, #{o : Obj | no o.get and no get.o}]
}
```

to prevent generation of instances in which half or more of the objects are not linked to other objects.

In the concrete example, the numbers of OD instances found for the calls listed above are: 23, 9, 0, 5. So here there exists no OD (within the search constraints) that is an instance of both CD1 and CD2. From all the ODs we now randomly choose five, while ensuring that not more than two are taken from the same "bucket", i.e., not three or more ODs that satisfy `cd1 and not cd2`, etc.

[3] By happenstance, they could also be identical, but that would be detected and rejected in a later step.

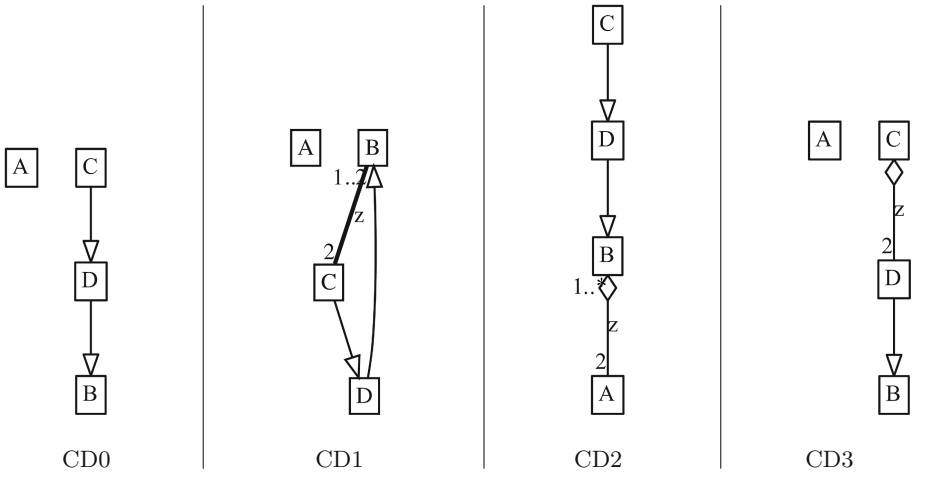


Fig. 10. CDs created in the concrete example.

If that is not possible, we start over with generating a new CD0.[4] Fig. 11 shows the five ODs obtained in our example run of the generator, each annotated with the bucket it came from.

We have implemented the strategy explained above, along with visualisation etc., and made it available through integration into an instance of the Autotool [13] e-learning system at https://autotool.fmi.iw.uni-due.de/alloy-cd-od. What is shown to students for each task are CD1 and CD2 (but thick edges turned normal) and OD1–OD5, of course without the bucket annotations. Immediate feedback is provided on student answers by checking them against what the system knows about the origin (buckets) of the presented ODs. Seeds for the random task generator would be derived from student identification numbers in an actual course.

What remains to be done here in the paper is to explain the role of thick edges in CDs. These are associations, aggregations, or compositions that interact in a somewhat subtle way with inheritance. For example, the thick association in CD1 in Fig. 10 will be inherited at one end from B to D to C, with the consequence that C objects can have links to themselves (see OD3 in Fig. 11). Ultimately, the "puzzle" concerning Fig. 6 in Sect. 3 was also caused by a "thick edge" (though it was not depicted thick there). So treatment of these specific constellations of relationships in CDs is one way of making exercise tasks less or more challenging. We currently permit them for CD0, for at most one of CD1 and CD2, but not for CD3. We do not disclose their presence or

[4] This is also the step where we would reject the case that CD1 and CD2 happened to be identical. For if they were, then the first two buckets, `cd1 and not cd2` as well as `cd2 and not cd1`, would be empty, and it would be impossible to choose five ODs from the remaining two buckets while not taking more than two from one bucket.

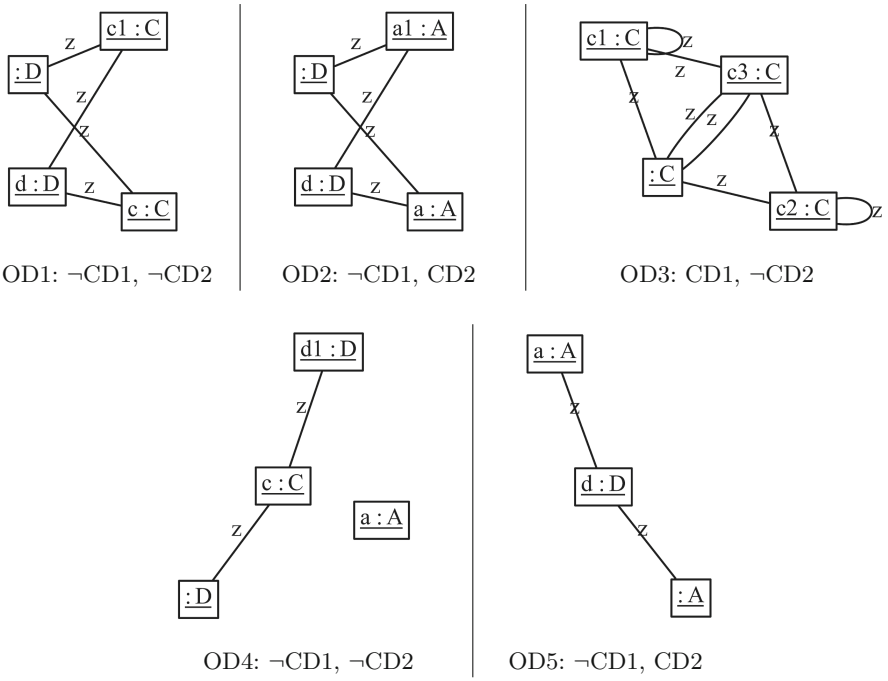


Fig. 11. Randomly chosen ODs in the concrete example run.

absence (showing everything as normal lines instead) to students, just as we do not disclose CD0, CD3, or which mutations have been made between CDs.

7 Related and Future Work

We have already mentioned existing translations from the UML domain to Alloy [1,7–9,12] throughout the paper. Instead of generating ODs using Alloy, instances for a given CD may also be found using an instance generating graph grammar [3]. To do so, the meta model in that context would require essentially two extensions. First, analogously to the customisation of CD2Alloy, support for mutations of CDs would be additionally needed. Second, constraints would have to be put in place to ensure that the ODs generated as instances are (or are not) instances of certain mutated variants.

Besides empirically evaluating our own exercise task generation via use with student cohorts, it would be interesting to further investigate ways of tailoring tasks to specific teaching goals. We have already discussed some possibilities, such as in the last paragraph of the previous section. We could also provide even more control to instructors for variability of tasks, for example not just letting them configure the numbers of classes, relationships etc., but also which CD mutations should be employed in a certain setup (thus allowing generation of

tasks focusing on a specific CD/OD concept, or in which we can actually ask students for the reasons a certain OD does not conform to a certain CD), or allowing a larger mutation distance between the CDs used in a task. Extending the approach in order to go beyond structural aspects of CDs and ODs, for example by including attribute fields and methods, would be feasible since the CD2Alloy translation already supports these features. Of course, we would have to use didactic considerations, such as which kinds of bad examples related to attributes and methods we want to handle, for devising appropriate mutations to employ. On a more technical level, future changes could see us using Alloy for generation of CDs (under a range of structural and possibly other constraints) as well. And in a departure from our current use of completely artificial class and object names, we could aim for generating tasks with more meaningful names, also for attributes etc., instead.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Softw. Syst. Model. **9**(1), 69–86 (2010)
2. Booch, G.: The Unified Modeling Language User Guide. Pearson Education India (2005)
3. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Softw. Syst. Model. **8**(4), 479–500 (2009)
4. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional, Reading (2004)
5. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. (TOSEM) **11**(2), 256–290 (2002)
6. Jackson, D.: Software Abstractions – Logic, Language, and Analysis, Revised edn. MIT Press, Cambridge (2011)
7. Kautz, O., Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: a translation of class diagrams to Alloy. Technical report AIB-2017-06, RWTH Aachen University (2017)
8. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: class diagrams analysis using Alloy revisited. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 592–607. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_44
9. Massoni, T., Gheyi, R., Borba, P.: A UML class diagram analyzer. In: Proceedings of Workshop on Critical Systems Development with UML, pp. 143–153 (2004)
10. Object Management Group: Unified Modeling Language (OMG UML), Version 2.5.1, December 2017
11. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Pearson Higher Education (2004)
12. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 158–171. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12261-3_16
13. Waldmann, J.: Generating and grading exercises on algorithms and data structures automatically. In: Proceedings of Automatische Bewertung von Programmieraufgaben. CEUR Workshop Proceedings, vol. 2015. CEUR-WS.org (2017)