"Mind the Gap!": Learning Missing Constraints from Annotated Conceptual Model Simulations

Mattia Fumagalli[1], Tiago Prince Sales[1], and Giancarlo Guizzardi[1,2]

[1] Conceptual and Cognitive Modeling Research Group (CORE),
Free University of Bozen-Bolzano, Bolzano, Italy
{mattia.fumagalli, tiago.princesales, giancarlo.guizzardi}@unibz.it
[2] Services & Cybersecurity, University of Twente, The Netherlands

Abstract. Conceptual modeling plays a fundamental role to capture information about complex business domains (e.g., finance, healthcare) and enables semantic interoperability. To fulfill their role, conceptual models must contain the exact set of constraints that represent the worldview of the relevant domain stakeholders. However, as empirical results show, modelers are subject to cognitive limitations and biases and, hence, in practice, they produce models that fall short in that respect. Moreover, the process of formally designing conceptual models is notoriously hard and requires expertise that modelers do not always have. This paper falls in the general area concerned with the development of artificial intelligence techniques for the enterprise. In particular, we propose an approach that leverages *model finding* and *inductive logic programming (ILP)* techniques. We aim to move towards supporting modelers in identifying domain constraints that are missing from their models, and thus improving their precision w.r.t. their intended worldviews. Firstly, we describe how to use the results produced by the application of model finding as input to an inductive learning process. Secondly, we test the approach with the goal of demonstrating its feasibility and illustrating some key design issues to be considered while using these techniques.

Keywords: Conceptual Modeling · Model Validation · Inductive Learning · Model Simulation

1 Introduction

Conceptual modeling plays a fundamental role in information systems engineering. In complex and sensitive scenarios (e.g., finance, healthcare), domain models are paramount in supporting critical semantic interoperability tasks. To fulfill this role, modelers must be able to systematically produce models that precisely articulate the worldview of the relevant domain stakeholders [16].

Technically speaking, domain models should only admit *instantiations* (e.g., *model interpretations*, *instance populations*) that correspond to state-of-affairs that are admissible according to the conceptualizations these models are supposed to represent. However, as empirical results show, modelers are subject to cognitive limitations and biases, and, in practice, they are often unable to

create domain models endowed with this property, also due to a lack of expertise [16,18,19,24,31]. In particular, these results show that such models are often *underconstrained*, thus admitting interpretations their designers did not intend.

The issue of repairing underconstrained models has been investigated in the past [12,9,6], however, none has been able to automatically learn complex domain constraints yet. Some of us have developed a validation technique that combines model finding with visual simulation to automatically generate admissible model instances, which one could analyze to manually derive missing constraints [1]. In this paper, we combine this technique with a machine learning algorithm from *inductive logic programming* (ILP) to automate this process.

With our validation technique, one naturally generates a dataset of *allowed (positive) and forbidden (negative) examples* from admissible model interpretations. By combining this dataset with the constraints already embedded in the model and feeding them to an ILP learner [2], we can automatically uncover missing constraints. The main advantage of this approach is that it does not require modelers to formulate the constraints themselves. Instead, they simply need to judge whether model interpretations should be allowed or forbidden.

The remainder of this paper is organized as follows. In section 2, by introducing a running example, we explain how visual model simulation allows us to determine whether a model is *underconstrained*. In section 3, we describe our approach, and, in section 4, we evaluate its capacity to learn complex constraints that would be needed in practice. In section 5, we discuss related work. Finally, in section 6, we make some final considerations, including implications to practice.

2 Model Validation

Conceptual modeling is an error-prone activity [31]. Modelers often dedicate a significant amount of time to testing and debugging their models in order to increase their reliability [7]. To cope with that, research efforts have been devoted to devising engineering tools for model validation, which consists of assessing if a model is: (i) *overconstrained*, namely, if it excludes interpretations intended by the modeler; or (ii) *underconstrained*, namely, if it admits interpretations that are not intended by the modeler.

Checking if a domain model is overconstrained can be easily represented as a classical *model checking* problem, namely, as the activity of *verifying* whether a given state of affairs *holds* in a given model [4]. Take, for instance, the OntoUML[3] model depicted in Figure 1 (a fragment of a model about vehicles and their parts, which could be used to devise a *vehicle dealer* knowledge base). Model-checking allows one to detect whether a given state of affairs like *"x is both a wheel and an engine"* violates the *logical rules* encoded by the model. In this example, it is trivial to see that such a state-of-affairs is not allowed[4], but that is not always the case. Nonetheless, if this state-of-affairs was intended by the author of this

[3] OntoUML is a version of UML designed in accordance with the UFO foundational ontology principles and axiomatization [15,17].

[4] In OntoUML, all *kinds* are mutually disjoint [15].

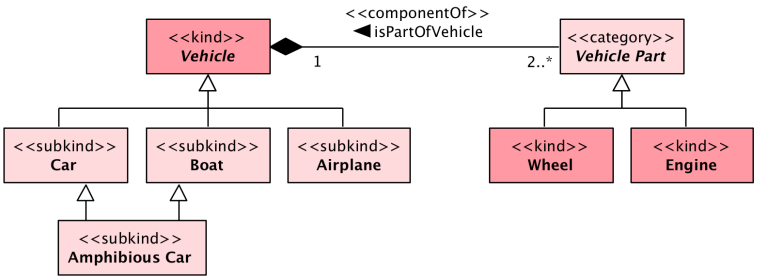

Fig. 1: An OntoUML model on (a subset of) the vehicle dealership domain.

model, and yet not allowed by it, the model would be considered *overconstrained*. To adjust it, one would need to "relax" it by removing or "weakening" some of its constraints [33].

Checking whether a conceptual model is *underconstrained* is another model validation task, but one that cannot be handled via model checking. It can be informally expressed as follows: *"check if the model only allows instantiations representing the state of affairs intended by the modeler"*. Let us come back to the example in Figure 1. Suppose we have a state of affairs like *"x is both a car and a boat, but it is not an amphibious car"*. While running model checking, this statement does not violate the model, but still, the modeler may consider it a violation of her domain conceptualization.

From now on we use the terms "admitted state of affair", "configuration", and "simulation run" interchangeably, where a simulation run is the result of *an interpretation function satisfying the conceptual model*. In other words: if we take the (Onto)UML diagram of Fig.1 as a M1-model (in OMG's MDA sense), a configuration is a M0-model that could instantiate that M1-model; if we take the UML diagram as a logical specification, then a configuration is a logical model of that specification. Finding these valid configurations given a specification is the classical task performed by a *model finder*.

While the two aforementioned tasks are both important when validating conceptual models, there are important differences between them. On the one hand, the task of checking whether a given state of affairs holds in a given model can be algorithmically addressed by *satisfiability solvers* [20]. On the other hand, as anticipated in [11], the task of identifying *what* in a conceptual model allows for an unintended state of affairs, implies that the intended model, which is assumed to be implicit in the mind of the modeler, is involved in the validation phase. This enables an empirical process where humans cannot be left outside the loop. The latter challenge is what we focus on in this paper.

3 From Model Finding to Inductive Learning

An overview of the proposed approach is summarized in Figure 2 below. Besides the domain model M, the ILP process takes as input a list of negative (neg) and positive (pos) examples, which are elicited by applying model finding. The final

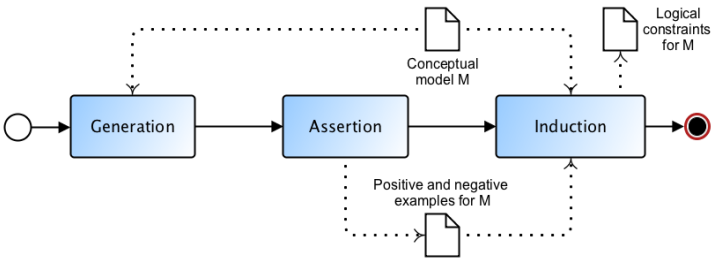

Fig. 2: Approach overview.

output is a set of logical constraints that can be used by the modeler to complete the input domain model.

The core steps involved in the approach we envision are like from the pseudocode represented in Listing 1. These steps can be grouped into three main phases, namely the *generation*, the *assertion* and the *induction* phase.

Listing 1: Model Finding and ILP combination process.

```
   Result: Set of logical constraints L^M
1  get input conceptual model M;
2  for conceptual model M do
       /* (1) Generation */
3      convert conceptual model into model finder specifications M^F;
4      execute model finding;
5      store simulations files into S^M;
6      for unintended and intended simulations S^M admitted by M^F do
           /* (2) Assertion */
7          combine S^M with M^F;
8          elicit positive and negative examples E^{-/+};
           /* (3) Induction */
9          run ILP with E^{-/+} and M as inputs;
10         store ILP outputs into set L^M;
11     end
12 end
```

(1) Generation. The first task here is to take a domain model [15] as input and convert it into a format (see step 3 in Listing 1) so that it can be validated through model finding. We achieved this by using a *compiler* that runs a transformation on the input (OntoUML) model, and relies on the mappings proposed by previous work [1]. Here the model is fully converted into a neutral logical layer[5], which is then converted into Alloy [21], an expressive language for specifying and analyzing structures based on relational logic, which includes existential and universal quantifiers from first-order logic, operators from set theory (e.g., union, intersection), and relational calculus (e.g., relational join). Alloy is equipped with a powerful model analysis service that,

[5] Currently, the logical layer can be encoded by First Order Logic FOL syntax or by Description Logic (DL) syntax, covering ALC, SHOIQ, and SROIQ expressivity.

given a context, generates possible instances for a given specification (it can also allow model checking and counterexamples generation). An example of the model showed in Figure 1 converted into an Alloy specification is available in `https://github.com/unibz-core/Mind-the-Gap/blob/main/CarDL.als`. After converting the input conceptual model, the Alloy Analyzer APIs are applied to validate the model (step 4 in Listing 1). The analysis is performed to simulate arbitrary instances that conform to the model constraints. This step requires a definition of the *scope* of the analyzer, which consists of the *type* of concepts to be analyzed and the *number* of instances to be produced.

Once a set of configurations is produced (S^M in Listing 1), the modeler can classify them into *intended* or *unintended*. For the validation of the model configurations, we followed the strategy in [31]. If some unintended configurations are found the process continues, otherwise it terminates (meaning by this that the input conceptual model is correct according to the modeler scope).

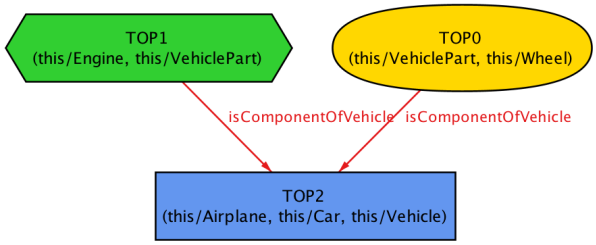

Fig. 3: Example of vehicle parts model simulation.

Figure 3 presents an example of configuration generated out of the model represented in Figure 1, with three instances. The colors of the boxes represent the different kinds of objects involved in the simulation. Notice that "`this/...`" refers to a class, and the values "*TOPx*" refer to its generated instances. So if `TOP2` is marked with `this/Car` and `this/Airplane` then this individual is both a 'Car' and a 'Airplane' at the same time. This simulation could be, for instance, annotated as unintended, since we may do not want to allow for a "Car `(TOP2)` to be also an Airplane". Notice that all the output configurations collected in S^M are saved in a file collecting all the information generated through the Alloy analyzer visualization tool.

(2) Assertion. Once the set of intended and unintended configurations is generated, we apply another conversion step. Here we use the trace of the original domain model as input and all the files of the annotated simulations to create a new output. The file generated out of this step is the domain model and all the instances coming from the intended and unintended simulations. This is what can be used by the modeler to elicit negative and positive examples. Each imported configuration, indeed, involves a mix of allowed and proscribed relations (i.e., particular individuals that instantiate a class in the model). For example, in Figure 3, the instance 'TOP2' may be proscribed, while 'TOP1' may be allowed. The assertion step allows the modelers to mark which instances in

the unintended simulations represent negative (proscribed) or positive (allowed) examples. Notice that the plan is to use an *ad hoc* editor to support the annotation process together with the example set generated in the assertion step. In particular, we will employ the capabilities embedded in the OntoUML editor [10] with some additional features, such as i) exploration of Alloy simulations; ii) simulations annotation; iii) neg/pos example set generation.

(3) Induction. In this phase, the elicited negative and positive examples, along with the structure of the original conceptual model, are given as input to a learning system. Considering the scope of this paper, the learning process we set up must accounts for the ability of *identifying missing formal constraints* in a way that is easily accessible to the modeler, which should be able, then, to process the suggested output and repair the input source model. For this particular goal, we adopted the CELOE algorithm, an extended version of the OCEL [25] algorithm. This is considered one of the best available state-of-the-art *Inductive Logic Programming* (ILP)[28] options for *Class Expression Learning*, and has been applied to a large number of cases [2]. Notice that, multiple ILP algorithms are available. Accurate analysis and benchmark of the existing options for our task is out of the scope of this paper, and is part of immediate future work.

An illustrative case of a populated model transformation for this particular task can be represented as follows (this case reuses the car simulation output of Fig.3, along with elicited neg/pos examples):

* $E^+ = \{$*Engine(TOP1), Wheel(TOP0), VehiclePart(TOP1), VehiclePart(TOP0)*$\}$
* $E^- = \{$*Car(TOP2), Airplane(TOP2), Vehicle(TOP2)*$\}$
* $PCM = \{E^+, E^-, \forall x.(Airplane(x) \rightarrow Vehicle(x)), \forall x.(Car(x) \rightarrow Vehicle(x)), \forall x.(Engine(x) \rightarrow VehiclePart(x)), \forall x.(Wheel(x) \rightarrow VehiclePart(x)), \forall x.(VehiclePart(x) \rightarrow \exists y.(Vehicle(y) \wedge isComponentOfVehicle(x, y)))\}$

where E^+ represents the set of positive examples, namely the examples that are admitted by the intended models as well as (e.g., in this case) allowed fragments in unintended models; where E^- represents the set of negative examples, namely those instances that are proscribed in unintended models; and where PCM (Populated Conceptual Model) represents the model file, with the model axioms and all its instances along with negative and positive examples. By using the model, the positive examples and the negative examples, the algorithm is able to highlight the rule(s) describing the non-admitted instance(s):

* *Structural error* $= \exists x.(Car(x) \wedge Airplane(x))$
* *Suggested constraint* $= \forall x.(Car(x) \rightarrow \neg Airplane(x))$

The axiom identified by the algorithm uncovers that the problem is due to an overlap between the class 'Car' and the class 'Airplane'. Once possible underconstraing problems are uncovered, in order to forbid the unintended instance(s), the axioms are then simply negated.

4 Evaluation

We evaluate the proposed approach by addressing the two following research questions: (1) *To which extent the proposed combination between model finding and ILP is able to discover constraints that can be used to avoid unintended configurations in practice?* (2) *To which extent the process we propose is able to produce constraints that are expected?*

The first question aims at assessing the feasibility of the approach, namely if the two technologies together can be used to find useful constraints. The second question aims at assessing performance issues from a qualitative perspective, namely if the design choices we adopted in the presented process allow us to identify the constraints that are expected.

4.1 Setup

Method. To address both questions we ran an experiment involving a simple example model.[6] We take here the general methodological practice employed in natural sciences of starting with simple models to explore a fuller extent of the ideas at hand [3]. In this particular case, irrespective of its size, the conceptual model used in this experiment allows for the investigation of relevant constraints, which are likely to be needed in practice. For this goal, we used, as a preliminary "litmus test"[7] recurrent modeling issues that appear in models of all sizes, namely: i) a possible occurrence of *completeness error*, usually caused by the difficulty of balancing the *flexibility* and *consistency* of the model [26,24]; ii) a possible occurrence of *"or" operator misuse*, usually caused by the *overlap between the linguistic and logical usage of "and" and "or"* [30] ; iii) a possible occurrence of *imprecise association*, usually caused by the fact that the range of an association is too broad, thus allowing to miss some domain-specific constraints [31]. The evaluation is performed in a controlled environment, in which we know in advance the constraints to be learned. The configurations to be assessed are generated randomly through Alloy, and, to force the analyzer in creating the unintended configurations, the target constraints were negated.

To answer question (1) we checked whether the approach can learn constraints able to avoid occurrences of issues like *i)*, *ii)* and *iii)*. To answer the research question (2) we analyzed the process we followed to derive the expected constraint with 100% *accuracy*, as the sum of all the *true* elicited positive examples and all the *true* elicited negative examples divided by all the examples.

Data. The model we used in the experiment is the one depicted in Figure 1. We firstly highlighted three kinds of error (e) that can be hosted by the input model and. Secondly, we identified possible target constraints (c). Selected errors and constraints were paired as follows:

[6] All data used for the case study described in this section are available for research purposes at `https://github.com/unibz-core/Mind-the-Gap`.

[7] A litmus test is "a critical indicator of future success or failure". A is a litmus test for B if A can be effectively used to measure some property of B [5].

(1.e) *"some vehicles are neither cars, boats nor airplanes"*
(2.e) *"some vehicles are both cars and boats, but not amphibious cars"*
(3.e) *"some vehicles have no engines"*
(1.c) *"all vehicles are either cars, boats or airplanes"*
(2.c) *"all vehicles which are both cars and boats are amphibious cars"*
(3.c) *"all vehicles have at least one engine"*

(1.e) is generated by the fact that the specialization of the class "Vehicle" is not complete. (2.e) occurs because the class "AmphibiousCars" is taken as a sub-class of "Car" *"or"* "Boat" instead of being equivalent to the intersection between "Car" *"and"* "Boat". (3.e) is generated by the fact that the "componentOf" relation is used at a very abstract level, namely between "Vehicle" and "VehiclePart", thus missing the specific constraint between "Vehicle" and "Engine". To check whether we are able to learn constraints for avoiding errors (1.e-2.e-3.e), we ran *15 simulations* and collected a total of *17 examples*. For each error, we then created a populated model by selecting the negative examples highlighting each error and the related positive example. This was in order to test if we are able to learn the constraints (1.c-2.c-3.c).

4.2 Results

Question 1. We firstly highlighted the error (1) by selecting instances of the concept 'Vehicle' that are neither cars, nor boats, and nor airplanes. The output constraint was the following:

$$\forall x.(Vehicle(x) \rightarrow (Airplane(x) \vee Boat(x) \vee Car(x))) \tag{1}$$

The rule was straightforwardly derived with 100% of *accuracy*.[8] Secondly, we selected negative examples highlighting error (2) where the elicited negative examples were both 'Cars' and 'Boats', but not 'Amphibious Cars'. The derived rule was the following:

$$\forall x.((Boat(x) \wedge Car(x)) \rightarrow AmphibiousCar(x)) \tag{2}$$

The output constraint was straightforwardly derived with 100% of *accuracy*. Finally, we learned a constraint for defining a target class through a target relation. Here we highlighted vehicles without an engine as negative examples. The final output axiom was the following:

$$\forall x.(Vehicle(x) \rightarrow \exists y.(Engine(y) \wedge isComponentOfVehicle(y, x))) \tag{3}$$

[8] Notice that the output provided by the applied algorithm can be taken as a *rule* composed by axioms encoded in Description Logic (DL) or *manchester owl syntax* (`www.w3.org/owl2-manchester-syntax/`), and in order to map the output into FOL language, a further mapping must be applied. For instance, the output resulting from the conjunction of the first three axioms provided as solution by the algorithm applied for the rule (1) above was: $(Vehicle \sqsubseteq ((Airplane) \sqcup (Boat) \sqcup (Car)))$.

Again, the output constraint was derived with 100% of *accuracy*. The selection of the target classes for the above trials worked as a scope restriction to focus on the part of the model we wanted to analyze and repair.

Question 2. The amount of data we generated to identify the target constraints was relatively small. As anticipated before, we just needed to generate 15 simulations and 17 example instances. The amount of time used to run the transformation steps and the induction of the constraints was trifling, in the order of few milliseconds. However, in order to avoid not useful, i.e., noisy, simulations (e.g., simulations with concepts, such as 'Wheel', which are not related to the errors), during the model finding step, we had to manage the scope of the analyzer. Moreover, before learning the expected constraints with 100% accuracy we had to go through each input populated model and identify possibly conflicting negative/positive examples. For instance, in order to learn rule (1), we had to exclude negative examples highlighting other possible errors. By running the first trial we got indeed 66% for the target constraint rule. This was because we firstly selected a total of 6 examples, which, with the provided annotation, returned 4 true positive examples, two false-positive examples, and, accordingly, 2 false-negative examples and 4 true negative examples. The two 'outliers' (annotated as negative examples) in this case, were selected to avoid overlapping issues (e.g., a 'Vehicle' which is both an 'Airplane' and a 'Boat', thus allowing for a negative and a positive example for both 'Airplane' and a 'Boat'), which were not directly connected to error 1.e.

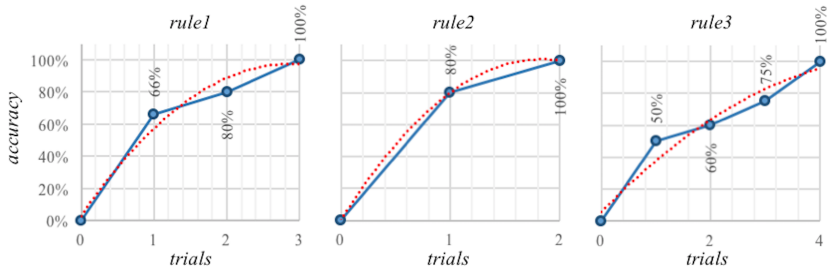

Fig. 4: Number of trials and accuracy trends.

Figure 4 shows the trials we run, for each rule, to get 100% accuracy. The chart shows how the deletion of the given outliers improves the accuracy of the learning task (for sake of clarity we showed the improvement by deleting an example per trial). The conclusion we can draw from this experience is that, in order to get the best from the learning task and allowing the modeler to decrease the effort of trial-and-error activities, a set of formal guidelines for the annotation and the ILP set-up must be provided (for instance, processing subfragments of the model, focusing on associations and generalization relations separately, deleting irrelevant instances, minimizing the set of examples). The definition of these guidelines will be part of the immediate future work.

4.3 Threats to Validity

As for any experimental evaluation, some threats can affect the validity of our findings. These threats are mainly concerned with the generalization of the observed results to a real-case scenario.

Firstly, we did not account for random annotation. In this experiment, indeed, the errors we found in the configurations were generated by knowing in advance the rules to be learned. The main focus here is on assessing whether the given approach is valid in learning the selected repairing solutions. In order to check the types of rules that can be learned when the modeler does not know in advance the required output, an experiment considering a larger population of modelers and, possibly, a larger set of conceptual models should be conducted. Moreover, to make the approach usable by teams of engineers that may vary in size and complexity, the proposal must be evaluated across a breadth of annotation data sets, both varying in size and complexity, to also provide a precise assessment of the limits of the new technique. Secondly, in the experiment we did not compare the observed results with what could be obtained by using alternative ILP algorithms. We recognize that the approach could potentially benefit from assessing different algorithms in terms of what types of rules can be learned (or not) and *how* efficiently they can be learned. We intend to do this as future work after gathering more data from concrete models in different domains.

5 Related Work

Model validation. Our work builds primarily upon the large amount of work done in the recent years on *conceptual model validation*. [1] proposed a solution to assess conceptual models defined in OntoUML by transforming these models into Alloy specifications. Similarly, [32] devised a methodology to test the conceptual models semantic quality, in terms of *correctness* and *completeness*, which is based on the generation of *automated conceptual test cases* [23]. The approach presented in [13] is also related to what we propose. Here the solution is based on the application of a system, namely the USE system, that allows for a model-driven validation of UML models and OCL constraints, by generating multiple instances of the model, or snapshots. These snapshots, like the Alloy simulations, can be then assessed by the modeler to detect overconstraining and underconstraining issues. The recent work described in [14], even if from a higher-level perspective, is related to our proposal as well. There, the main goal is to formalize the process by which modelers analyze and modify the model as a sort of "dialogue", which must be iterated in order to better identify and capture the final intended worldview.

Compared to these key works, the main difference in our approach is the exploitation and the integration of model finding and statistical relational learning techniques, like ILP, in order to identify refactoring solutions.

Constraint Learning. Our work can be also placed in the general research area of *constraint learning for conceptual models*. In this respect, of direct relevance

to our effort on using the validation process to then infer model constraints, is also the work in [29]. This approach aims at identifying and solving possible (UML) model design flaws, by exploiting model finding and adopting *constraint logic* [22]. Similarly, the work in [9] proposes a genetic algorithm [27] in order to generate Object Constraint Language (OCL) invariants from examples. Moreover, the work in [6] propose an approach to infer OCL invariants from valid/invalid snapshot by checking the relevance of the generated outputs. Another work that is close in spirit to ours is discussed in [18,31], where the main goal is to efficiently identify recurrent error-prone structures across conceptual models and then manually uncover possible missing constraints. Compared to these pivotal works, the main difference in our approach are the following: i) we employ ILP (as opposed to CSP as, e.g., in [6]) to address the constraint learning problem. As discussed in [8], there might be some relevant advantages in addressing CSP problems from an ILP perspective; ii) we support multiple representational languages for the input conceptual model to be assessed and repaired, namely, OWL, UML and OntoUML (approaches like the ones present in [6,27], for instance account for UML models only). In fact, our aim is to build an infrastructure for learning: a) *ontological constraints* such as the ones expressible in OntoUML [15]; b) anti-patterns for that language [31]; iii) we provide a set-up of the model finding facility that can be used to curate a database of positive and negative examples for conceptual models, covering different application domains. In fact, our approach seamlessly leverages and extends on the existing methodological and computational support for model validation via visual simulation (visual model finding) provided by the OntoUML toolset [1,10]. In that sense, it requires less intrusive manual interventions for eliciting and curating examples and counterexamples than, for example, [6].

6 Discussion

In the sequel, we discuss the implications of this novel approach. Moreover, by identifying the limitations of our current setting, we also discuss opportunities for future work.

Implications. A first central implication is that the proposed approach can support the modelers in better exploiting the analysis they run with the model finding model simulator. Indeed, by collecting and annotating the simulations generated through the application of the model finder, a large data set of intended/unintended simulations can be generated, and then (re-)used to derive possible repair options. This can be taken as a backup facility that allows storing and keeping track of information that may be lost during the run-time analysis.

Secondly, another key point is that, by enabling modelers to generate populated models with a related set of elicited negative/positive examples, the proposed framework paves the way to a large number of case studies, especially on the application of different learning approaches, to identify constraints, or to address other tasks that can support the conceptual modeling activities (like for instance the identification of unintended instances across models, exploiting

the annotation of previous activities). In the current setting, we adopted one single algorithm, but the same algorithm can be tuned in multiple ways and can produce different kinds of outputs. A benchmark of the available technologies and the possible configurations is out of the scope of this paper, but it is still a pivotal research issue, especially to check *what* kind of constraints can be learned and *how* easily they can be understood and reused to repair the model.

Thirdly, by adopting ILP the constraint learning task can leverage on a very small amount of data. Similarly, relying on ILP allows inducing human-understandable logical rules. Still, this does not prevent us to exploit a much larger amount of data. Even just by applying the CELOE algorithm, we selected, indeed, we can measure the *accuracy* of each derived rule, thus leveraging the feedback provided by multiple annotations and/or users.

Fourthly, the application of the automated steps of the framework, see for instance the conversion of the conceptual model into Alloy specifications, or the automatic identification of possible erroneous axioms, along with the suggestion of possible repair solutions, return to be useful allies in the conceptual modeling process. While they cannot be seen as alternatives to most of the modelers' activities, still they can be seen as useful means for improving the efficiency of some key steps. For instance, the modeler does not need to know how to encode its model into Alloy specifications, or she can start from a set of multiple repair suggestions, before deciding how to change the source model, this being particularly helpful in a scenario with very complex models, or involving non-expert (i.e., novice) modelers.

Limitations. Applying the presented framework over OntoUML configurations, and then learning more complex constraints is our long-term objective and it triggers the agenda for the immediate future work. In order to achieve this goal, with the current set-up, there is still a gap that needs to be bridged.

The model in Figure 5 provides an example of one of the problems that we may want to address in the future, but that cannot be solved without human intervention with our current approach. In this model, while 'Purchase' defines a certain *kind* of relationship (i.e., "Relator", according to the OntoUML terminology), 'Buyer' and 'Seller' represents two *roles* that can be played by instances of the *kind* 'Person'. Supposing that we run a simulation by reducing the scope to 'Purchase', 'Buyer', 'Seller' and 'Purchasable Item', it is possible to have an instance that is involved as both a 'Buyer' and 'Seller' in the same 'Purchase' relationship. An example of an unintended simulation that can be generated would be then when an instance of 'Purchase' (suppose `this/TOP0`) is the source of both 'involvesSeller' and 'InvolvesBuyer', and these relations have the same instance (suppose `this/TOP1`) as the target. The constraints to be learned to avoid this problem can be then represented by the formula (4) below.

$$\forall x.y.z((Purchase(x) \wedge involvesBuyer(x,y) \wedge involvesSeller(x,z)) \rightarrow y \neq z) \quad (4)$$

Currently, we are not able to learn the exact constraints for this type of error. This is primarily due to the fact that the algorithm we selected is limited to DL

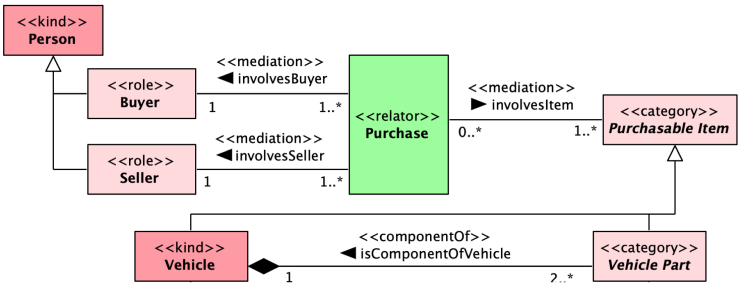

Fig. 5: An extension of the OntoUML model represented in Figure 1.

expressivity. CELOE, indeed, is an efficient solution if the goal is to support modelers in constructing models and resources, used to devise, for instance, reasoning systems, or *Semantic Web* applications. Moreover, it is widely applied in a lot of ontology engineering case studies and it is also implemented as part of the DL-learner framework [2]. Still, learning complex rules, such as (4), is out of the scope of this algorithm and, hence, will require the investigation of complementary ILP approaches. Moreover, this kind of more complex problems brings challenges for the annotation step. We may have cases, indeed, where to highlight a possible problematic structure we may need to annotate as negative multiple instances involved in multiple relations, thus increasing the level of complexity of the learning process, and affecting the accuracy of the output.

Another key observation is that, while the proposed approach aims at supporting the conceptual modelers through the automatization of some steps of the engineering activities, humans still need to be in the loop of the process. Since the approach aims at making explicit unintended models, the only way to collect this information is, indeed, to leverage on the feedback of the modeler, and to manually annotate the simulations. The key aspect here is that we offer a facility to collect data about this (often tacit) information. Moreover, the output provided by the ILP algorithm still needs to be interpreted by the modeler. Depending on the applied set-up, different outputs can be provided, and each output can be used in different ways. Similarly, constraints with different levels of restriction can be learned, and the choice of what axiom to be selected depends on the modeler's goals. For instance, with the current set-up, instead of the constraint presented by formula (4), we can learn the following constraint:

$$\forall x.(Purchase(x) \rightarrow \neg\exists y.z.(involvesBuyer(x,y) \wedge\ Seller(y) \\ \wedge involvesSeller(x,z) \wedge\ Buyer(z)) \tag{5}$$

The above rule implies, indeed, a stronger restriction as it makes the roles 'Seller' and 'Buyer' disjoint (while formula (4) only requires that they are not played by the same person in the scope of the same Purchase). For this reason, for constraints that depart from DL expressivity such as (4) we consider the ILP output as "repair suggestions", i.e., the learned constraints must be checked and

eventually adapted by the modeler. This strategy of providing partial solutions that are then adapted by the modeler was successfully employed for anti-pattern rectification in [31].

7 Conclusion and Perspectives

This paper makes a contribution to the theory and practice of (ontology-driven) conceptual modeling diagnosis and repair by: i) presenting a framework to combine model finding and ILP, in order to support model validation; ii) presenting a practical solution to generate and exploit multiple simulations for any given (Onto)UML conceptual model, thus allowing its analysis and annotation; iii) presenting a practical solution to learn constraints from the annotated simulations output. Adopted data and processes are available at `/unibz-core/Mind-the-Gap` and `/unibz-core/gufo2alloy`, respectively.

Based on the presented results, as future work, we plan to evaluate our approach over different OntoUML models, encoding errors with a higher level of complexity, thus uncovering also *recurrent* errors across models and related constraints. This involves both practical and theoretical research that examines the impact of various algorithms on the learning goal, as well as the generation of a data set of annotated simulations coming from different OntoUML models.

References

1. Benevides, A. B. et al.: Validating modal aspects of OntoUML conceptual models using automatically generated visual world structures. Journal of Universal Computer Science **16**(20), 2904–2933 (2010)
2. Bühmann, L. et al.: DL-Learner—a framework for inductive learning on the semantic web. Journal of Web Semantics **39**, 15–24 (2016)
3. Cairns-Smith, G.: The life puzzle: on crystals and organisms and on the possibility of a crystal as an ancestor. University of Toronto Press (1971)
4. Clarke Jr, E. M. et al.: Model checking. MIT press (2018)
5. Collins: Litmus test. In: Collins Dictionary. `https://www.collinsdictionary.com/dictionary/english/litmus-test`, accessed in 2021-04-02
6. Dang, D.H., Cabot, J.: On automating inference of OC constraints from counterexamples and examples. In: proc. KSE 2015, pp. 219–231. Springer (2015)
7. De Nicola, A., Missikoff, M., Navigli, R.: A software engineering approach to ontology building. Information systems **34**(2), 258–275 (2009)
8. De Raedt, L. et al.: Learning constraint satisfaction problems: An ILP perspective. In: Data Mining and Constraint Programming, pp. 96–112. Springer (2016)
9. Faunes, M. et al: Automatically searching for metamodel well-formedness rules in examples and counter-examples. In: proc. MODELS 2013. pp. 187–202 (2013)
10. Fonseca, C.M., Sales, T.P., Viola, V., Fonseca, L.B.R., Guizzardi, G., Almeida, J.P.A.: Ontology-driven conceptual modeling as a service. In: 11th International Workshop on Formal Ontologies Meet Industry (FOMI'21). CEUR-WS (2021)
11. Fumagalli, M., Sales, T.P., Guizzardi, G.: Towards automated support for conceptual model diagnosis and repair. In: 1st Workshop on Conceptual Modeling Meets Artificial Intelligence and Data-Driven Decision Making (2020)

12. Gogolla, M., Büttner, F., Richters, M.: A UML-based specification environment for validating UML and OCL. Science of Computer Programming (2005)
13. Gogolla, M. et al.: USE: A UML-based specification environment for validating UML and OCL. Science of Computer Programming **69**(1-3), 27–34 (2007)
14. Grüninger, M.: Ontology validation as dialogue (2019)
15. Guizzardi, G.: Ontological foundations for structural conceptual models. Telematica Instituut / CTIT (2005)
16. Guizzardi, G.: Theoretical foundations and engineering tools for building ontologies as reference conceptual models. Semantic Web **1**(1, 2), 3–10 (2010)
17. Guizzardi, G., Fonseca, C.M., Almeida, J.P.A., Sales, T.P., Benevides, A.B., Porello, D.: Types and taxonomic structures in conceptual modeling: A novel ontological theory and engineering support. Data & Knowledge Engineering **134** (2021). https://doi.org/10.1016/j.datak.2021.101891
18. Guizzardi, G., Sales, T.P.: Detection, simulation and elimination of semantic antipatterns in ontology-driven conceptual models. In: proc. ER 2014. pp. 363–376. Springer (2014)
19. van Harmelen, F., ten Teije, A.: Validation and verification of conceptual models of diagnosis. In: proc. EUROVAV 1997. pp. 117–128 (1997)
20. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and reasoning about systems. Cambridge university press (2004)
21. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)
22. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. The journal of logic programming **19**, 503–581 (1994)
23. Janzen, D., Saiedian, H.: Test-driven development concepts, taxonomy, and future direction. Computer **38**(9), 43–50 (2005)
24. Kayama, M. et al.: A practical conceptual modeling teaching method based on quantitative error analyses for novices learning to create error-free simple class diagrams. In: proc. IIAI. pp. 616–622. IEEE (2014)
25. Lehmann, J. et al.: Class expression learning for ontology engineering. Journal of Web Semantics **9**(1), 71–81 (2011)
26. Leung, F., Bolloju, N.: Analyzing the quality of domain models developed by novice systems analysts. In: proc. HICSS. pp. 188b–188b. IEEE (2005)
27. Mirjalili, S.: Genetic algorithm. In: Evolutionary algorithms and neural networks, pp. 43–55. Springer (2019)
28. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. The Journal of Logic Programming **19**, 629–679 (1994)
29. Pérez, B., Porres, I.: Reasoning about UML/OCL class diagrams using constraint logic programming and formula. Information Systems **81**, 152–177 (2019)
30. Roussey, C. et al.: A catalogue of OWL ontology antipatterns. In: Proceedings of the fifth international conference on Knowledge capture. pp. 205–206 (2009)
31. Sales, T.P., Guizzardi, G.: Ontological anti-patterns: Empirically uncovered error-prone structures in ontology-driven conceptual models. Data & Knowledge Engineering **99**, 72–104 (2015)
32. Tort, A., Olivé, A., Sancho, M.R.: An approach to test-driven development of conceptual schemas. Data & Knowledge Engineering **70**(12), 1088–1111 (2011)
33. Troquard N. et al: Repairing ontologies via axiom weakening. In: McIlraith, S.A., Weinberger, K.Q. (eds.) 32nd AAAI Conference on Artificial Intelligence. AAAI Press (2018)