Towards Automated Support for Conceptual Model Diagnosis and Repair

Mattia Fumagalli, Tiago Prince Sales, and Giancarlo Guizzardi

Conceptual and Cognitive Modeling Research Group (CORE),
Free University of Bozen-Bolzano, Bolzano, Italy
[mattia.fumagalli, tiago.princesales, giancarlo.guizzardi]@unibz.it

Abstract. Validating and debugging conceptual models is a very time-consuming task. Though separate software tools for *model validation* and *machine learning* are available, their integration for an automated support of the debugging-validation process still needs to be explored. The synergy between model validation for finding intended/unintended conceptual models instances and machine learning for suggesting repairs promises to be a fruitful relationship. This paper provides a preliminary description of a framework for an adequate automatic support to engineers and domain experts in the proper design of a conceptual model. By means of a running example, the analysis will focus on two main aspects: *i)* the process by which formal, tool-supported methods can be effectively used to generate negative and positive examples, given an input conceptual model; *ii)* the key role of a learning system in uncovering error-prone structures and suggesting conceptual modeling repairs.

Keywords: Conceptual Models · Model Simulation · Inductive Learning

1 Introduction

The complexity of building conceptual models is a widely recognized research issue. Works like [9] and [7] underline the limitations of human cognitive capabilities in managing the huge and difficult activities involved in conceptual modeling. This is the main reason why, over the years, multiple solutions aimed at supporting conceptual model design have been provided by different communities. Most of these solutions can be categorized as *complexity management engineering tools*, and they offer semi-automated or fully-automated support facilities for model design, validation, or verification [8].

To adequately support the engineering of complex conceptual models, besides these tools, we have seen, in the last decade, an increasing interest in the use of *ontology-driven conceptual modeling languages* [16]. These languages mainly seek to offer a reference layer for conceptual modeling construction, validation, and code generation. In this spirit, the recent work presented in [14] describes a novel validation strategy using visual *model finding* [10], that can be used for eliciting *anti-patterns* in conceptual models. The empirically-elicited research output in [14] offers a concrete example of how *error-prone modeling decisions* can be uncovered and made explicit, thus offering a methodology to diagnosis and repair of conceptual models.

In approaches such as [14], however, anti-pattern detection as well as the construction of rectification plans is done manually, i.e., the authors have manually validated dozens of models, manually detected these emerging error-prone structures, and have manually proposed effective rectification plans. As shown therein, with this process, they have managed to propose a catalog containing dozens of anti-patterns. Manually conducting this process, however, is a difficult and time-consuming task, which, as consequence, limits the number of models that can be analyzed and, hence, the number of structures that can be discovered. To address this limitation, we are interested in identifying how, by using *model finding* and *machine learning* (ML), the design activities of this approach can be supported. In other words, we want to reduce the effort to uncover error-prone structures in conceptual models and identify repairs suggestion, by automating these tasks as much as possible.

Though separate software tools for model finding and machine learning are available, their integration for automating the debugging-validation process still needs to be explored. Inspired by the work of Alrajeh and colleagues [2, 3], who proposed an approach to automatically diagnose and repair temporal logic software specifications based on the integration of *model checking* and *machine learning*, we seek to develop an approach for conceptual modeling, which in turn, leverages on *model finding*[1] and *machine learning* techniques.

The contributions of this paper are three-fold. Firstly, we propose a framework to implement the aforementioned synergy between model finding and machine learning for conceptual modeling diagnosis and repair. Secondly, we contribute to the identification of how formal, tool-supported methods can be effectively used to generate a data set of negative and positive examples of instances for a given conceptual model. We do this by carrying out an empirical simulation over a simple example conceptual model. In particular, we adopt the *Alloy Analyzer* [10] to generate multiple simulations of the input conceptual model and we propose a series of steps to encode information about intended/unintended models. Thirdly, once the data set of negative and positive examples has been elicited, we show how this data can be given as input to a *learning system*, which can be used to automatically uncover error-prone structures and suggest repairs to the modeler.

The remainder of this paper is organized as follows. In section 2, we briefly introduce our running example. Section 3 introduces the framework, by describing the main steps, agents, and components involved. Section 4 shows how to go from model finding, through annotation, to example set generation. Section 5 describes the role of a learning system in identifying error-prone structures and suggesting repairs. Finally, section 6 presents some final considerations and describes future work.

[1] For a detailed analysis of model checking and model finding see [10].

2 Conceptual Modeling: Learning by Feedback

We take here the general methodological practice employed in natural sciences [5] of starting with simple models to explore a fuller extent of the ideas at hand before making progress to complex ones. In that spirit, although the ultimate goal of this research program is to develop a framework target at ontology-driven conceptual modeling languages (in particular, OntoUML [7]), we start here with standard UML and with the toy model depicted in Figure 1 below.

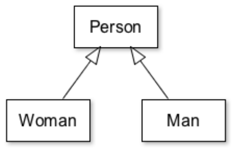

Fig. 1: A toy example in UML.

Now suppose that we can run simulations (or configurations) of the given example model with at most 2 instances per configuration[2]. The list of possible configurations of this model is depicted in Figure 2, in which solid arrows mean direct instantiation and dashed arrows indirect instantiation.

By looking at these possible outputs, the modeler may identify some *unintended* configurations, namely instances that she does not want her model to allow. Now suppose that by looking at these outputs, the modeler can annotate what are the intended/unintended configurations. From these annotated configurations, what can we learn as the most general rules? Looking at the super-simple model above the modeler may want to avoid all the cases in which 'Person' has direct instances (e.g, 'c' and 'e' in Figure 2) and where an instance is both a 'Man' and a 'Woman' (e.g, 'i' and 'm' in Figure 2). If this is the case, the simple rule to be inferred can be informally expressed as *"Every person is either a man or a woman and no person is both a man and a woman"*. To repair the input conceptual model, a knowledge engineer would simply have to add a constraint that forbids these two generic configurations represented in Figure 3. In UML, this could be achieved with a generalization set that is complete (`isCovering = true`) and disjoint (`isDisjoint = true`).

From this example, we make two main observations. Firstly, consider a much more complex model than the one in Figure 1. The activity of debugging the model by checking all the intended/unintended configurations is very time consuming and it may not be easy for the modeler to understand *where* the errors come from, *how* to repair the model, and *what* rules need to be added (if any).

[2] From now on we use the terms "simulation run" and "configuration" interchangeably, where a simulation run is the result of *an interpretation function satisfying the conceptual model*. In other words: if we take the UML diagram as a M1-model (in the 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*.

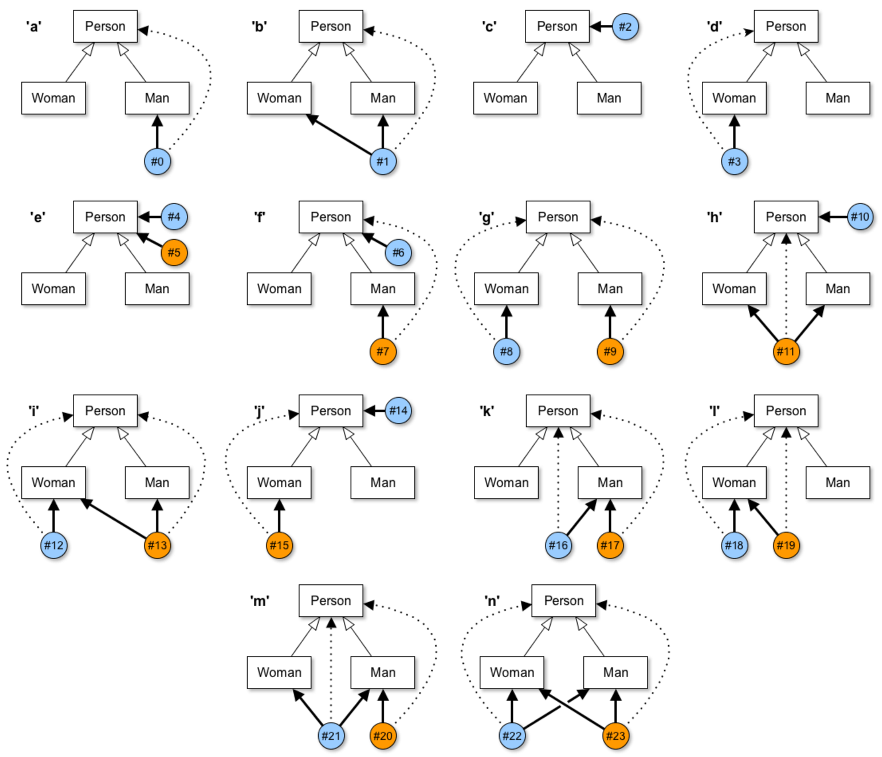

Fig. 2: List of simulations for the model of Figure 1.

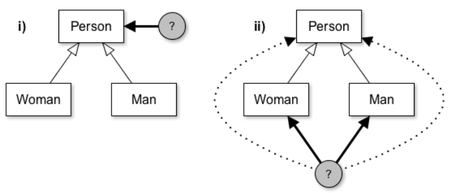

Fig. 3: Simulations of the model in Figure 1 allowing for unintended instances.

Secondly, consider a scenario where several people simulate the same model and people diverge on what they assign as intended and unintended configurations. We can then offer to the modelers possible options giving them an indication of how often people chose each of the options. This is about repairing a particular model by learning from a *collective judgment* (in this case, a type of *meaning negotiation* activity).

In summary, from the marriage between model validation, for finding faults, and machine learning, for suggesting repairs, a fruitful synergy emerges, which can support knowledge engineers in understanding how to design and refine rigorous models.

3 From Model Validation to Repairs Suggestion

The framework we envision should be able to produce, from a given conceptual model, a set of rules that forbid the occurrence of configurations marked as unintended by knowledge engineers. The key idea here is to combine and exploit model validation and learning technologies in order to: *i)* automatically generate a set of configurations of the input conceptual model and identify unexpected outputs; *ii)* carry out diagnosis and repair tasks by learning from the identified errors and suggesting rules to adjust the model accordingly.

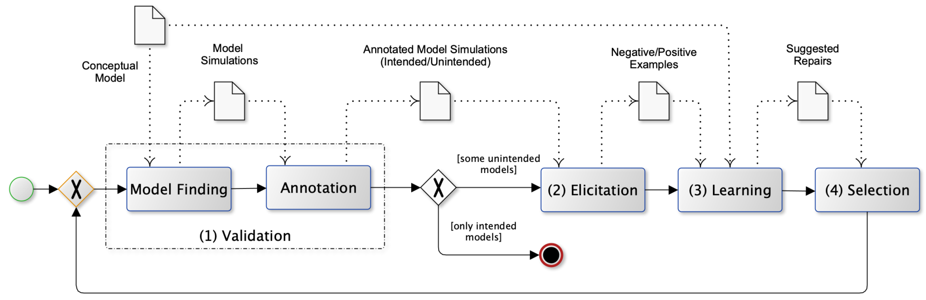

Fig. 4: Automated support for conceptual modeling diagnosis and repair: the proposed framework.

This framework comprises four steps explained in the sequel (see Figure 4), which can be executed iteratively:

Step 1. Validation. This step consists in automatically generating possible configurations from an input conceptual model and asserting whether these should indeed be allowed by it. If no unintended configurations are identified, the process terminates, otherwise it proceeds to step 2. The generational part of this step requires feeding the input model to a *model finder*.[3] The assertional part requires one or more knowledge engineers to decide on its validity. Considering the example in Figure 2, an unintended configuration could be represented by 'i', where an instance is both 'Man' and 'Woman'). If unintended models are found, the process continues to the next step.

Step 2. Elicitation. At this point, the model configurations generated by the model finder and annotated by knowledge engineers do not specify *why* they are intended or unintended. In an unintended configuration, indeed, we may have both allowed and forbidden instances (i.e. particular individuals that instantiate a class in the model). For example, in Figure 2, configuration 'f', the instance '6' is forbidden, while '7' is allowed. Step 2 allows the modelers to mark which instances represent negative or positive examples. Once negative and positive examples are produced, they are ready to be given as input, along with the structure of the original conceptual model, to the learning system.

[3] This step may require a previous conversion step, from the language used to design the conceptual model (e.g. UML, OntoUML) to the model finder specifications as in, e.g., [4]

Step 3. Learning. Having identified the negative and positive examples that make the configurations unintended or intended, a learning system software carries out the diagnosis process automatically. The goal of this step is to identify the structures of the model that are “error-prone” [14]. The output of the learning process, considering the super-simple model introduced above, can be exactly the negation of the two configurations represented in Figure 3. This, of course, depends on the information collected with the modeler annotation.

Step 4. Selection. The learning system may produce multiple examples of “error-prone” structures for the same model. For the repair task, a selection step for deciding among the possible repairs options is required. This selection step is always application dependent (i.e., it depends on the final purpose of the conceptual model) and requires inputs from the knowledge engineer. Once the selection is made, the update of the original input conceptual model can be addressed. Considering the two examples in Figure 3, if both of them are selected, the suggested repairs would be a negation of direct instantiation of ‘Person’ and the disjointness between ‘Woman’ and ‘Man’.

The presented combination of model finding and (logic-based) learning is intended to support an iterative process for evolving and repairing conceptual models by adding constraints that prevent unintended configurations. The iterative aspect of the process is relevant because there is no guarantee that a single application of the four steps will ensure the correctness of the model. Thus, it should be repeated until no unintended configurations can be found.

4 Highlighting Possibly Erroneous Decisions

Let us now consider model validation more formally. In the proposed framework, following the strategy in [14], the input conceptual model is translated into Alloy [10], a logic language based on set theory, which offers a powerful model analysis service that, given a context, generates possible instances for a given specification (it can also allow model checking and counterexamples generation). For example, once the conceptual model of Figure 1 is converted into an Alloy specification, multiple configurations of the model (for two instances) can be produced. Figure 5 below presents the full list of possible configurations, covering also the example diagrams provided in Figure 2.[4] Notice that “`this/...`” refers to a class, and the values within curly brackets refer to its generated instances. So if `this/Person` contains `Person3` and `this/Woman` contains `Person3`, it means the individual `Person3` is a ‘Person’ and a ‘Woman’ at the same time.

At this point, as a first task, the modeler should annotate those configurations that are intended or unintended. Following the super-simple model example the annotation can be represented as from Figure 6 below, where the red cross marks the unintended simulation.

[4] Notice that Alloy produces ‘0’ and ‘1’ instances only, we numbered the instances considering the full list of possible configurations.

```
#  'Toy' model Alloy configurations
'a' this/Person={Person0}, this/Man={Person0}, this/Woman={}
'b' this/Person={Person1}, this/Man={Person1}, this/Woman={Person1}
'c' this/Person={Person2}, this/Man={}, this/Woman={}
'd' this/Person={Person3}, this/Man={}, this/Woman={Person3}
'e' this/Person={Person4, Person5}, this/Man={}, this/Woman={}
'f' this/Person={Person6, Person7}, this/Man={Person7}, this/Woman={}
'g' this/Person={Person8, Person9}, this/Man={Person9}, this/Woman={Person8}
'h' this/Person={Person10, Person11}, this/Man={Person11}, this/Woman={Person11}
'i' this/Person={Person12, Person13}, this/Man={Person13}, this/Woman={Person12, Person13}
'j' this/Person={Person14, Person15}, this/Man={}, this/Woman={Person15}
'k' this/Person={Person16, Person17}, this/Man={Person16, Person17}, this/Woman={}
'l' this/Person={Person18, Person19}, this/Man={}, this/Woman={Person18, Person19}
'm' this/Person={Person20, Person21}, this/Man={Person20, Person21}, this/Woman={Person21}
'n' this/Person={Person22, Person23}, this/Man={Person22, Person23}, this/Woman={Person22,
↪  Person23}
```

Fig. 5: Configurations generated by Alloy (empty model excluded). Each individual, e.g., *Person0*, maps into the corresponding instance in Fig. 2, e.g., *#0*.

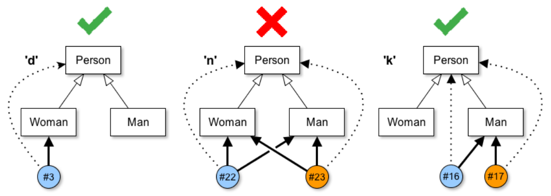

Fig. 6: Examples of simulations annotated as intended/unintended.

The further step here is to convert the above annotation into an example set collecting information about the input conceptual model and the annotation provided by the modeler. This will involve an additional input from the modeler, namely the annotation of what instance in the simulation makes the configuration intended or unintended. For instance, looking at Figure 6 the two instances to be marked as "negative" are '#22' and '#23'.

Notice that, the plan is to use an *ad hoc* editor to support the annotation process and the example set generation step. In particular, we will employ the capabilities embedded in the OntoUML editor [6], which will play a key role along the process, with some additional features (some of them already implemented), such as: a) exploration of Alloy simulations; b) simulations annotation; c) negative/positive example set generation.

The overall phase from the input conceptual model, through the generation of multiple simulations, to the annotation and the generation of the negative/-positive examples set, can be formalized as a composed function f_a, where:

$$f_a : f_b \circ f_c \tag{1}$$

$$f_b : M \rightarrow (A_M \times \mathcal{S}^{+/-}_{A_M}) \tag{2}$$

$$f_c : (A_M \times \mathcal{S}^{+/-}_{A_M}) \rightarrow E^{+/-} \tag{3}$$

With M being the conceptual model. A_M being the conceptual model converted into Alloy specifications. $\mathcal{S}^{+/-}_{A_M}$ being a set of simulations generated through

Table 1: Embedding knowledge engineers input into neg/pos examples matrix.

	#Person0	#Person1	#Person2	#Person3	#Person4	#Person5	#Person6	#Person7	#Person8	#Person9	#Person10	#Person11	#Person12	#Person13	#Person14	#Person15	#Person16	#Person17	#Person18	#Person19	#Person20	#Person21	#Person22	#Person23
label	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**
{ ?x <#type> <#Person> }	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
{ ?x <#type> <#Man> }	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**1**
{ ?x <#type> <#Woman> }	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**

Alloy, annotated as intended/unintended, given A_M. $E^{+/-}$ being a data set collecting negative and positive examples. The output of f_b is an Alloy model associated to a list of intended/unintended simulations. The output of f_a is an example set that can be given as input to the learning system.

As a final remark, f_a can be seen as a semi-automatic process. If the conversion steps (e.g., from the conceptual model to the Alloy specifications, or the generation of $E^{+/-}$) can be easily automatized, some manual work from the modelers, which need to provide feedback within the loop, is still required.

5 Uncovering Error-Prone Structures

The output of the phase described in the previous sections should be as represented by Table 1. In order to generate the above matrix we adopted a standard propositionalization process [12], where: *i)* we converted the conceptual model and the related instances (e.g., '#1', '#2', '#3', etc.) into a logical knowledge base specification (KB) (i.e., the combination of the so-called TBOX and ABOX); *ii)* we gave the reference KB as input of a script to generate a matrix of patterns; *iii)* we extended the matrix with the information about positive and negative examples (see the 'label' attribute). From this input, there may be multiple ways to set-up the learning step and automatically extract repairs suggestions. For instances, the learning system can be used to implement an *Association Rule Mining* (ARM) [1] approach, or to implement relational learning based on *Inductive Logic Programming* (ILP)[13].

```
//Rule1
if { ?x <#type> <#Man> } = false and { ?x <#type> <#Woman> } = false then false
//Rule2
if { ?x <#type> <#Man> } = false and { ?x <#type> <#Woman> } = true then true
//Rule3
if { ?x <#type> <#Man> } = true and { ?x <#type> <#Woman> } = false then true
//Rule4
if { ?x <#type> <#Man> } = true and { ?x <#type> <#Woman> } = true then false
```

Fig. 7: Rules extracted from the annotation of the output presented in Fig. 5.

In this paper, we adopted a standard approach. We derived the rules by using a simple *Decision Tree* model, where the attributes for splitting are selected according to the *gain ratio* criterion [11], and we run *subgroups discovery* to induce an exhaustive rule set plus a list of insights to better explain the results. Notice that the role of ML statistical techniques to extract rules and insights from the modelers' feedback becomes more useful as the complexity of the model increases and the number of feedback increases. For instance, having multiple (and possibly inconsistent) feedback for the same simulation, the labels for each instance may have multiple values encoding *weights*, instead of binary values, such as '0' and '1', like in the example of Figure 1.

'Rule 1' and 'Rule 4', in Fig. 7, represent the rules accounting for the unintended configurations, namely: *i)* when instances of 'Person' are neither instances of 'Man' nor 'Woman' ('1'); *ii)* when instances of 'Person' are both 'Woman' and 'Man' ('4'). The following formulas represent a *First Order Logic* (FOL) formalization of the derived 'negative' rules.

$$\exists x\, Person(x) \land \neg(Woman(x) \lor Man(x)) \tag{4}$$

$$\exists x\, Person(x) \land (Woman(x) \land Man(x)) \tag{5}$$

A further analysis can be run by checking the results provided by the subgroup discovery implementation, as from Table 2 below, where we grouped the most 'precise' rules.

Table 2: Extracted rules: some additional insights.

Rule	*Conclusion*	*Pos*	*Neg*	*Size*	*Coverage*	*Precision*	*Lift*	*Length*
//Rule1	**false**	**0**	**6**	**6**	**0,250**	**1,000**	**2,182**	**3**
//Rule2	**true**	**6**	**0**	**6**	**0,250**	**1,000**	**1,846**	**3**
//Rule3	**true**	**7**	**0**	**7**	**0,292**	**1,000**	**1,846**	**3**
//Rule4	**false**	**0**	**5**	**5**	**0,208**	**1,000**	**2,182**	**3**

Besides collecting information about the *Size* (i.e., how many instances are involved), the *Length* (i.e., how many predicates are involved) and the *Coverage* (i.e., how many instances covered over the total instances), a ranking of the rules can be provided in terms of, for instance, *Precision* and *Lift*. The *Precision* value explains the ratio of different values ('Pos' and 'Neg', for a certain rule) for the same instance (in the example we have precision '1', meaning that values are only 'Pos' or 'Neg'). The *Lift* value measures the value of a certain rule considering the ratio of premises and consequences in the given data set (see [15] for further details). Given the above derived 'negative' rules, the repairs that can be selected by the modelers would be quite straightforward. The input conceptual model (assuming here a FOL formalization of that model) can be then constrained as follows:

$$M = \{\forall x\, Woman(x) \rightarrow Person(x), \forall x\, Man(x) \rightarrow Person(x)\} \tag{6}$$

$$M^R = \{M, \forall x\, Person(x) \rightarrow (Woman(x) \lor Man(x)), \forall x\, Man(x) \rightarrow \neg Woman(x)\} \tag{7}$$

Where M represents the original conceptual model and M^R represents the new repaired (i.e., constrained) version of the conceptual model.

6 Conclusion and Perspectives

This paper presents preliminary results towards a framework for diagnosing and repairing faulty structures in conceptual models. In particular, our objective is to combine, on one hand, the *model finding* techniques for generating positive (intended) and negative (unintended) model configurations and, on the other hand, use this curated base of positive/negative examples for feeding a *learning process*. Our overall research program, aims at addressing the full ontological semantics of the OntoUML. In that sense, it will leverage on the existing OntoUML support for model validation via visual simulation in Alloy [4, 14]. A further objective is learning from the configurations *structures that are recurrent* in several OntoUML models (i.e., anti-patterns) as well as reusable cross-model rules that could rectify them. Addressing these objectives for a particular model with a subset of the semantics of that language is a first step in that direction.

References

1. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: 1993 ACM SIGMOD. pp. 207–216 (1993)
2. Alrajeh, D., Kramer, J., Russo, A., Uchitel, S.: Elaborating requirements using model checking and inductive learning. IEEE TSE **39**(3), 361–383 (2013)
3. Alrajeh, D., Kramer, J., Russo, A., Uchitel, S.: Automated support for diagnosis and repair. Communications of the ACM **58**(2), 65–72 (2015)
4. Braga, B.F., Almeida, J.P.A., Guizzardi, G., Benevides, A.B.: Transforming OntoUML into Alloy: Towards conceptual model validation using a lightweight formal method. Innovations in Systems and Software Engineering **6**(1-2), 55–63 (2010)
5. Cairns-Smith, A.G.: The life puzzle: On crystals and organisms and on the possibility of a crystal as an ancestor. University of Toronto Press (1971)
6. Guerson, J., Sales, T.P., Guizzardi, G., Almeida, J.P.A.: Ontouml lightweight editor: a model-based environment to build, evaluate and implement reference ontologies. In: 19th IEEE EDOC (2015)
7. Guizzardi, G.: Ontological foundations for structural conceptual models. Telematica Instituut / CTIT (2005)
8. Guizzardi, G.: Theoretical foundations and engineering tools for building ontologies as reference conceptual models. Semantic Web **1**(1, 2), 3–10 (2010)
9. Guizzardi, G., Sales, T.P.: Detection, simulation and elimination of semantic anti-patterns in ontology-driven conceptual models. In: International Conference on Conceptual Modeling. pp. 363–376. Springer (2014)
10. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)
11. Karegowda, A.G., Manjunath, A., Jayaram, M.: Comparative study of attribute selection using gain ratio and correlation based feature selection. Intl. Journal of Information Technology and Knowledge Management **2**(2), 271–277 (2010)
12. Kramer, S., Lavrač, N., Flach, P.: Propositionalization approaches to relational data mining. In: Relational data mining, pp. 262–291. Springer (2001)
13. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. The Journal of Logic Programming **19**, 629–679 (1994)
14. 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)
15. Tufféry, S.: Data mining and statistics for decision making. J.Wiley & Sons (2011)
16. Verdonck, M., Gailly, F.: Insights on the use and application of ontology and conceptual modeling languages in ontology-driven conceptual modeling. In: Conceptual Modeling. pp. 83–97. Springer, Cham (2016)