# A Feasibility Study on Using Classifying Terms in Alloy*

## A case study

Robert Clarisó[1] and Martin Gogolla[2]

[1] Universitat Oberta de Catalunya, Barcelona, Spain
[2] University of Bremen, Bremen, Germany
rclariso@uoc.edu, gogolla@uni-bremen.de

**Abstract.** To perform the analysis of a structural model of a software system (like a class diagram), it is often necessary to compute sample valid instantiations (like object diagrams), for example, for testing purposes. *Classifying terms* (CTs) provide a technique for improving diversity in the instantiation generation process.
CTs have been proposed and studied in the context of UML class diagrams annotated with OCL invariants. Nevertheless, they can also be employed in other declarative specification languages. This paper explores the feasibility of using CTs in the context of Alloy. The discussion considers both the Alloy notation and the integration with the Alloy Analyzer.

**Keywords:** Software Modeling · Verification and Validation · Testing · Classifying term · OCL · Alloy.

## 1 Introduction

A popular use of models in software engineering is describing the conceptual schema of a software system, e.g., a *UML class diagram* in object-oriented development or an *Entity-Relationship schema* in database design. At this level of abstraction, one defines the concepts that establish the model, the information that is recorded for each concept, the relationships among the concepts, and the integrity constraints that establish what is required or forbidden in our model.

From a quality perspective, a fundamental problem about a conceptual schema is *satisfiability*: finding an instantiation (like an object diagram) populating the concepts, data and relationships that fulfills all integrity constraints in the model. *Model finders*, the tools in charge for finding such valid instantiations, use a variety of techniques: SAT, SMT or constraint solving, theorem proving, search

* This work is partially funded by the H2020 ECSEL Joint Undertaking Project "MegaM@Rt2: MegaModelling at Runtime" (737494) and the Spanish Ministry of Economy and Competitivity through the project "Open Data for All: an API-based infrastructure for exploiting online data sources" (TIN2016-75944-R).

based methods and others have been applied. One of their major concerns is efficiency in finding instantiations. Nevertheless, the output of a model finder can be used for validation (instantiations as (counter-)examples) but also for testing purposes (instantiations as test cases). Hence, it is necessary for outputs of the model finder to be *diverse*, that is, to represent a wide range of scenarios and situations. Otherwise, the outputs of the model finder may fail to include relevant corner cases and mask quality issues that may appear later in the development process.

The premise of this paper is a technique for improving the diversity of model finders: *classifying terms* (CTs) [8]. This approach relies on the software designer to provide a list of expressions that can help us detect if there are meaningful differences between two instantiations. Then, CTs can guide the model finding process to catch a set of solutions that are diverse *by construction*. This method has been used successfully in the context of UML class diagrams annotated with integrity constraints written in the Object Constraint Language (OCL). In this setting, support for classifying terms has been implemented in the tool USE (UML-based Specification Environment) [7]. Nevertheless, there is no support for applying classifying terms in other modeling notations.

Hence, this paper studies the feasibility of using classifying terms in a popular notation for the verification of declarative specifications: Alloy [9]. In addition to describing how CTs can be used in Alloy, we will discuss the benefits and limitations of the language and the toolkit from the point of view of CTs. For instance, a shortcoming of the Alloy textual notation is the lack of support for querying solutions provided by the model finder: query expressions on a solution can only be evaluated interactively through the Alloy GUI or programmatically by calling the Alloy API.

The remainder of the paper is structured as follows. Section 2 describes the running example used throughout the paper. Then, Sect. 3 presents classifying terms and how they are used in the USE tool in the context of UML and OCL. Section 4 describes how to apply classifying terms in Alloy. After that, Sect. 5 presents related work on diversity in Alloy. Finally, Sect. 6 concludes the paper and discusses future lines of work.

## 2 Running example

### 2.1 UML and OCL model

As a running example, we will use a simple UML class diagram describing the relationship between parents and children in a family tree. This class diagram is presented in Fig. 1, depicted utilizing the USE modeling environment.

This class diagram focuses around a single class (`Person`), that stores the first name (`fName`), last name (`lName`) and year of birth (`yearB`) of a person in a family. A relationship (`Parenthood`) links parents and their offspring, with each person having at most two parents and an unlimited number of children.

Moreover, this model also includes integrity constraints describing three well-formedness rules that families in this model should fulfill: the combination of first

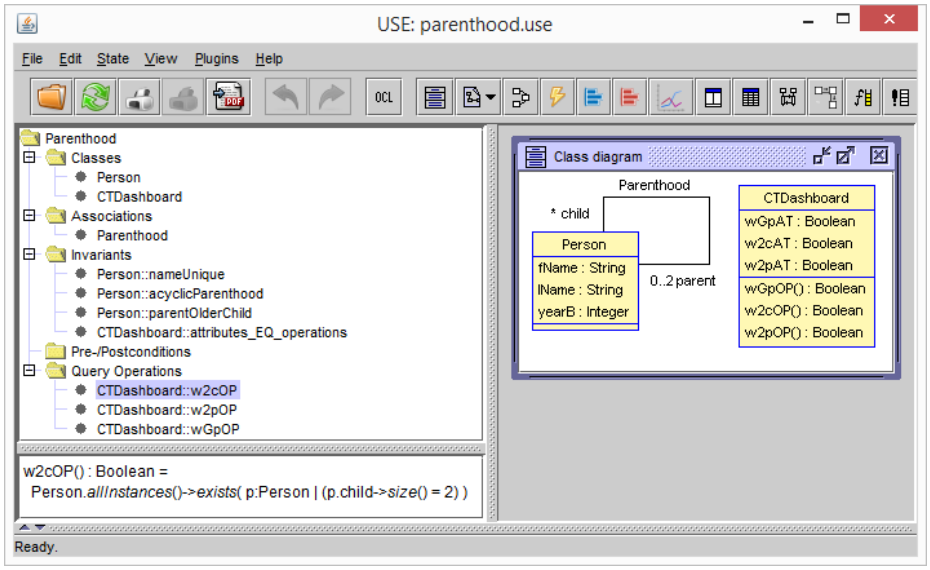


**Fig. 1.** Class diagram used as our running example.

and last name should be unique (`nameUnique`); a person cannot be her or his own ancestor (`acyclicParenthood`); and all parents should be at least 15 years older than their children (`parentOlderChild`). These integrity constraints can be described as invariants in OCL as follows:

```
context p1,p2:Person inv nameUnique:
  p1<>p2 implies (p1.fName<>p2.fName or p1.lName<>p2.lName)

context p:Person inv acyclicParenthood:
  p.parent->closure(p | p.parent)->excludes(p)

context p:Person inv parentOlderChild:
  p.child->forAll(c | p.yearB+15 <= c.yearB)
```

For applying classifying terms in that model, the mentioned model elements are enough. For explaining the CT approach in USE and for comparing it with Alloy, the class diagram also includes another class (`CTDashboard`), several query operations (`w2cOP`, `w2pOP`, `wGpOP`) and an invariant (`attributes_EQ_operations`). These elements will be used to illustrate classifying terms and they will be discussed in Section 3.

### 2.2 Alloy model

The running example can also be described using the Alloy textual notation. In Alloy, the notion of "class" is described as a signature (`sig`). Signatures may have *fields* of a given type, either a basic type like String or Int, a set or a relation.

Well-formedness constraints that should always hold are called *facts* (`fact`). The proposed encoding adds all attributes and association ends as fields of signature `Person`. Notice that two constraints that were implicit in the UML notation must be stated explicitly: the multiplicity of the association end `parent` and the fact that the association end `parent` is the inverse of association end `child`.

```
-- Class "Person"
sig Person {
  -- Attributes
  fName: String,
  lName: String,
  yearB: Int,
  -- Relationship "Parenthood"
  parent: set Person,
  child:  set Person
}
-- Multiplicity of role parent
fact multiplicityParent {
  all p: Person | #(p.parent) <= 2
}
-- Parent is the inverse of child
fact parentChildRelated {
  all p: Person | p.child = p.~parent
}
```

Besides this description of the structure of the class diagram, the original OCL invariants can also be encoded in Alloy using additional `fact` constraints:

```
-- Invariant uniqueName
fact uniqueName {
  all p1, p2: Person | p1 != p2 implies (
    (p1.fName != p2.fName) or (p1.lName != p2.lName))
}
-- Invariant acyclicParenthood
fact acyclicParenthood {
  no p: Person | p in p.^parent
}
-- Invariant parentOlderChild
fact parentOlderChild {
  all p: Person | all c: p.child | p.yearB + 15 <= c.yearB
}
```

There are many potential ways to model our running UML and OCL example. In this paper, we will consider the encoding presented in this Section to illustrate the proposed approach.

## 3 Classifying terms

Classifying terms are boolean or integer expressions that describe relevant properties about instantiations of a model. The goal is defining classifying terms in

such a way that instantiations with different values for the CT expressions will be *diverse*. That is, they have a characteristic structure or exhibit distinct properties that are considered significant from the point of view of the domain. Such classifying terms are proposed by the designer using domain knowledge.

For example, in our running example we may be interested in considering family trees with the following properties: (a) at least three levels of depth (a grandparent, a parent and a child) are present (with grandparent: wGp); (b) a parent with two children is present (with 2 children: w2c); (c) a child with two parents is present (with 2 parent: w2p). To this end we define three named boolean expressions that check each of these conditions and use them as classifying terms. The expressions do not have any free variables, i.e., classifying terms are closed OCL expressions of type boolean or integer.

```
wGp
  Person.allInstances->exists(g,p,c |
    g.child->includes(p) and p.child->includes(c))
w2c
  Person.allInstances->exists(p | p.child->size=2)
w2p
  Person.allInstances->exists(p | p.parent->size=2)
```

Considering our running example, we will show how the CTs can be used to compute a set of diverse instantiations with USE. Given that USE employs the model finder Kodkod that performs bounded verification, it is necessary to define the bounds of the search space that will be used for verification. The USE model validator, the component in USE responsible for generating instantiations, is called with additionally providing a so-called configuration that specifies the search space and defines lower and upper bounds for the number of objects in a class, the number of links in an association and possible attribute values:

```
Person: 1..3
Parenthood: 1..3
Person::fName: Set{'Ada','Bob','Cyd'}
Person::lName: Set{'Alewife','Baker','Cook'}
Person::yearB: Set{15,30,45,60,75,90}
```

As shown in Fig. 2, the USE model validator finds five instantiations, i.e., object diagrams, that satisfy the model under the stated configuration. One observes that the five instantiations show different shapes. In technical terms, each two instantiations show at least one classifying term that is evaluated differently in the two instantiations.

For better explaining and handling classifying terms in Alloy, we have slightly extended the above sketched USE model with a (singleton) class `CTDashboard` and add the classifying terms to our model as operations: `wGpOP`, `w2pOP` and `w2cOP`. However, for simply applying classifying terms `CTDashboard` is not needed. We also define in the class `CTDashboard` one boolean attribute per classifying term. An auxiliary invariant `attributes_EQ_operations`, which was not part of the original model, forces the value of these attributes to be equal to the result of the corresponding operations that evaluate the classifying term on the current

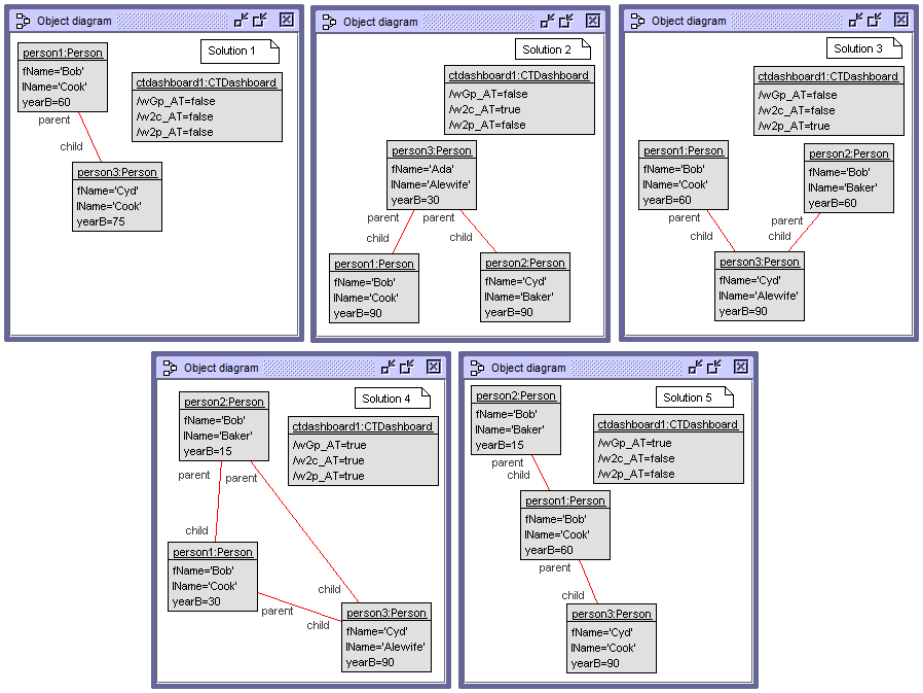


**Fig. 2.** Instantiations found for the CTs in our running example.

instantiation. The class `CTDashboard`, the operations for the classifying terms and the invariant `attributes_EQ_operations` are defined as follows:

```
class CTDashboard -- singleton
attributes -- AT attribute - OP operation
  wGpAT:Boolean -- with grandparent
  w2cAT:Boolean -- with 2 children
  w2pAT:Boolean -- with 2 parents
operations
  wGpOP(): Boolean =
    Person.allinstantiations->exists(g,p,c |
      g.child->includes(p) and p.child->includes(c))
  w2cOP(): Boolean = ...
  w2pOP(): Boolean = ...
end

constraints
  context CTDashboard inv attributes_EQ_operations:
    wGpAT=wGpOP() and w2cAT=w2cOP() and w2pAT=w2pOP()
```

When there are several classifying terms, we are interested in the interactions among them. Our goal will be finding instantiations that have different combinations of values for the classifying terms. For example, with $n$ boolean classifying

```
1 function findDiverseInstantiations(model, invariants, ⟨CT_1, ..., CT_n⟩)
    input  : A model constrained by invariants; CT_1, ..., CT_n: Classifying Terms
    output: A set of diverse instantiations of model that satisfy invariants
2   solutions ← ∅;
3   while instantiation of model satisfying invariants exists do
4     instantiation ← findValidInstantiation (model, invariants);
5     solutions ← solutions ∪ instantiation;
6     ⟨v_1, ..., v_n⟩ ← evaluate (instantiation, ⟨CT_1, ..., CT_n⟩);
7     newInvariant ← ¬(CT_1 = v_1 ∧ ... ∧ CT_n = v_n);
8     invariants ← invariants ∪ newInvariant;
9   return solutions
```

**Algorithm 1:** Generating diverse instantiations using classifying terms.

terms we would be interested in finding up to $2^n$ instantiations, each for each combination of values. Nevertheless, some combinations may be impossible as they cause a contradiction among themselves or with the invariants in the model. Again, deciding the suitable number and structure of classifying terms is up to the designer.

Once we have defined the set of classifying terms, they can be used in order to generate diverse instantiations. Algorithm 1 describes this iterative process, which runs internally in USE and invokes the model finder in each iteration to find a valid instantiation (line 4). Then, each classifying term is evaluated in the new instantiation (line 6) and a new auxiliary constraint is defined (line 7) to forbid future instantiations to have the same combination of values for the classifying term. This new constraint is added as an invariant (line 8) and the process is repeated until the model finder is unable to find new instantiations (line 3). In this way, all instantiations computed by the algorithm differ in at least the value of one classifying term.

Notice that Algorithm 1 is general in the sense that it supports both integer and boolean classifying terms. For instance, in line 7 the values $v_1, \ldots, v_n$ for each classifying term may be boolean or integer. For an integer classifying term $CT_I$ we would write constraints like $CT_i = v$ with an integer value $v$. For boolean classifying terms $CT_i$, we may prefer to write $CT_i$ and $\neg CT_i$ for clarity rather than $CT_i = true$ or $CT_i = false$, but the algorithm operates in the same way.

Figure 3 depicts the search space defined in USE for finding instantiations of our running example in graphical form. The procedure presented in Algorithm 1 can then be applied: by invoking a command called `mv -scrollingAllCT` in the USE shell, that instructs the model finder to find all instantiations considering a set of CTs provided as a parameter.

The output of this procedure will be a set of diverse instantiations. For example, Fig. 2 presents the 5 instantiations computed for the running example using the 3 classifying terms introduced in this section and the verification bounds presented in Fig. 3. Notice that out of 8 potential partitions, only 5 include a valid instantiation. This is due to the maximum bound of 3 objects of class `Person` set in the verification bounds. These bounds make it impossible, for example,

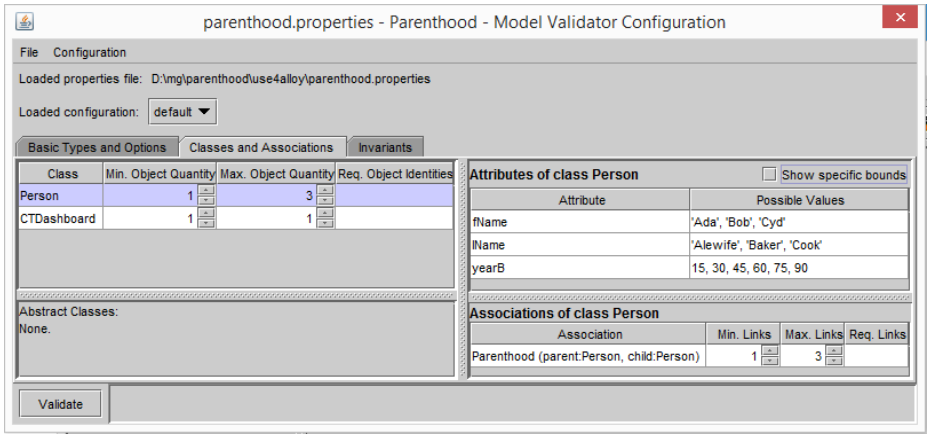


**Fig. 3.** Search space used for model finding in USE.

to have two (different) parents and two (different) children (so four different objects) in one solution. However, please be aware of the fact that in solution 4 there is one person with two children and one person with two parents, but there are only three persons in total.

## 4 Using classifying terms in Alloy

In the following, we describe a strategy to apply classifying terms in Alloy. Our goal is providing the same capabilities available in USE in the context of the Alloy notation and Analyzer. To this end, we need to implement the different steps in the procedure described in Algorithm 1. In particular, the following tasks should be supported:

1. Defining classifying terms.
2. Finding a valid instantiation.
3. Evaluating classifying terms on a given instantiation.
4. Defining a new auxiliary invariant for our model.

**Defining classifying terms.** Alloy allows the definition of *predicates* (named boolean expressions with parameters) and *functions* (named expressions with parameters). Predicates can be used to define boolean classifying terms while functions can describe integer classifying terms.

For example, the three classifying terms in our running example can be defined as the following three Alloy predicates:

```
pred wGp() {
  some g, p, c: Person | (c in p.child) and (p in g.child) }
pred w2c() { some p: Person | #(p.child) = 2 }
pred w2p() { some p: Person | #(p.parent) = 2 }
```

**Finding a valid instantiation.** Analysis in Alloy is based on commands like `run` (find a valid example instantiation of a predicate) or `check` (find a counterexample of an assertion). Similar to configurations in USE, the command *scope* defines the bounds of our search space.

For example, the following command asks the Alloy Analyzer to find a valid instantiation with at most 3 elements in signature `Person`, using at most 3 different strings and encoding integers using 8 bits:

```
pred show() {}
run show for 3 Person, 8 Int, exactly 3 String
```

**Evaluating classifying terms on an instantiation.** The Alloy textual notation does not provide any mechanism to access or query the instantiations computed by the model finder. Nevertheless, the GUI for the Alloy tool offers an "Evaluator" view within the instantiation viewer where it is possible to evaluate expressions, including predicates or functions. Furthermore, it is possible to evaluate an expression over an instantiation using the Alloy API.

Using the Evaluator view, we can check the predicates on the instantiation computed by the Alloy Analyzer. Figure 4 shows a sample evaluation, displaying an instantiation where all three classifying terms evaluate to false (no grandparent, no parent with two children and no children with two parents).

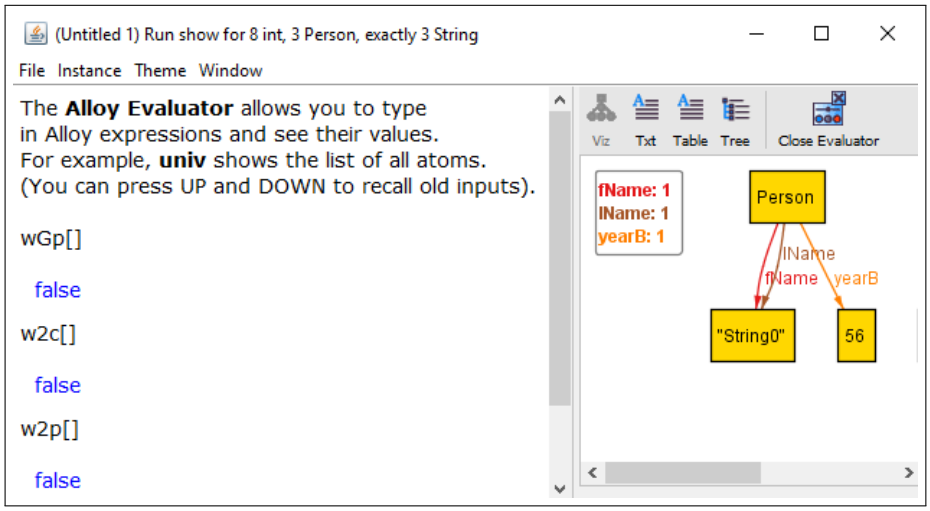


**Fig. 4.** Evaluating classifying terms using the Alloy GUI.

**Defining a new auxiliary invariant for our model.** After computing an instantiation, further instantiations should avoid previously visited combinations of values for the classifying terms. This means to define a new predicate like the following one, that forbids having all three classifying terms evaluating to false:

```
pred forbidFFF() {
  not (not wGp[] and not w2c[] and not w2p[]) }
```

This predicate can be included in our analysis by setting a new target predicate for our next Alloy command:

```
pred ct1() { forbidFFF[] }
run ct1 for 3 Person, 8 Int, exactly 3 String
```

This leads to the new instantiation in Fig. 5 where now the classifying term `w2p` evaluates to true: there is a person (`Person2`) with two parents. We have taken the freedom to manually modify the Alloy representation of the instantiation for better comprehension and added our view in terms of a UML object diagram.

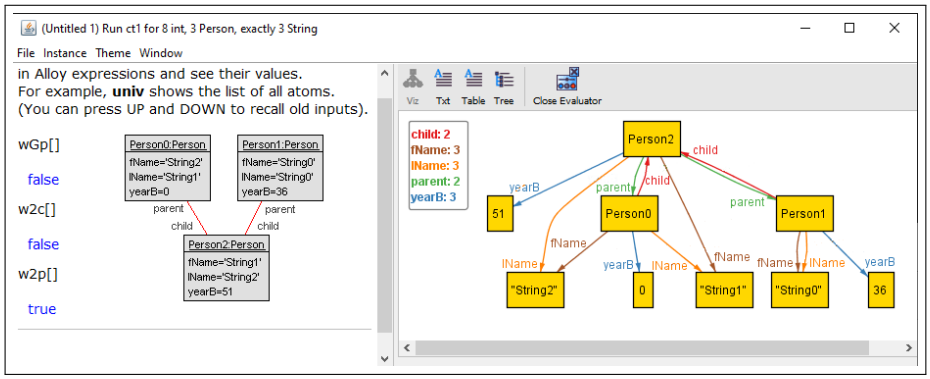


**Fig. 5.** New instantiation after forbidding the found combination of values for CTs.

The next iteration would forbid this combination of values (as well as the previous ones) and continue searching for an instantiation in a similar way. The process would continue until the Alloy command failed to find a valid instantiation within the defined scope:

```
pred forbidFFT() { not (not wGp[] and not w2c[] and w2p[]) }

pred ct2() { forbidFFF[] and forbidFFT[] }
run ct2 for 3 Person, 8 Int, exactly 3 String
```

**Discussion.** The proposed strategy allows the implementation of classifying terms in Alloy to improve diversity in the set of generated instantiations. As a result, a set of diverse instantiations of the model is computed. It supports both boolean or integer classifying terms and it does not require *changing* the original conceptual schema, only *adding* predicates that describe the classifying terms using the Alloy textual notation. An alternative approach could define a

"dashboard" as in the second part of the USE example. The dashboard allows to observe the values of the classifying terms in the current solution.

Nevertheless, the Alloy toolkit is designed for running individual analysis: after running a command, the textual Alloy notation does not support accessing the computed instantiation or to invoke further commands. These tasks can either be implemented using the GUI (which requires user intervention) or by invoking the Alloy API from a Java program.

In our experiment, both the USE Model Validator and Alloy used the KodKod constraint solver to find relevant instantiations. Given that both tools are using the same underlying solver, it does not make sense to compare the efficiency of both approaches. Also, as the description of the constraint is very similar but not exactly equal, comparing the size or shape of the resulting instances does not provide interesting insights.

However, there is one difference between Alloy and USE in their usage of KodKod: the Alloy language offers limited control of the problem bounds. In Alloy it is possible to specify an upper bound or an exact value for the population of each signature in the model or for the values of integers. While it is possible to constrain signatures and values by defining additional Alloy facts, the solver does not take advantage of the additional bound information and there may be a performance overhead. Instead, USE allows the definition of sets of potential values for attributes and lower and upper domain bounds for class populations. This makes the definition of the instantiations of interest more usable to the designer as it is possible to define meaningful values for attributes (*e.g.*, require the values of attribute `name` to be in the set "Ada, Bob, Cyd").

## 5 Related work

Classifying terms [8] have been proposed in the context of UML class diagrams annotated with OCL invariants. Support for classifying terms has been implemented in USE (UML-based Specification Environment) [7]. This paper describes the first attempt to implement classifying terms in Alloy. Relationships between UML and OCL on the one hand and Alloy on the other hand have also been studied in [2].

First, we discuss related work on how a user can guide the output of Alloy for a given command. One way to control the output is by *changing the scope* (keyword `for`) defined in the command, *i.e.*, the number of elements in each signature or the bit-width of integers, if the specification includes any. The scope can either be defined as an upper bound (the default behavior) or a specific size (using the keyword `exactly`). By adjusting the scope, diverse outputs can be produced for the same command. Nevertheless, unlike classifying terms, there is no systematic way to generate all relevant configurations, and it is not possible to control the structure of instantiations beyond their size.

Another approach to control Alloy's output for a given command is *changing the predicate*: defining a new predicate that invokes the one used in previous executions and adding additional constraints. This is the approach we have chosen

to implement classifying terms: adding additional predicates in each step that forbid repeating instantiations belonging to previously visited partitions.

Regarding extensions of Alloy, it is possible to require the output instantiation to be as close as possible to a *target instantiation* [3] or to some instantiation of a *target model* [11]. Changing the target allows us to explore diverse types of solutions. Furthermore, different criteria can be used to select which target instantiation should be computed by the solver. For example, *minimality* aims to find instantiations where no elements can be removed without violating the property being checked [12]. In the context of UML/OCL, *coverage* [15] considers a predefined catalog of properties for instantiations such as the multiplicity of association ends in an association or user-defined graph-based properties on the shape of the instantiation (e.g. being acylic) and attempts to compute diverse instances with respect to these criteria.

Finally, there are alternatives to classifying terms in order to improve diversity in solvers and model finders. Among them, *symmetry breaking* [14], *random sampling* [1, 5, 4], *abstract graph shapes* [13] and *distance metrics* [6]. However, a detailed comparison between these methods is out of the scope of this work.

## 6 Conclusions

In this paper, we have proposed a strategy for applying classifying terms in Alloy. Classifying terms can be used as a way to control the output of the Alloy Analyzer and to ensure the diversity of the generated instantiations.

Given a command, our approach works by computing the first output instantiation and then changing the command after each output. In each output instantiation, we assess the values of each classifying term. Then, we define a new predicate that adds a new constraint to those existing in the previous run: the combination of values for the classifying terms obtained by the last command is now forbidden. This predicate will be used as the goal for the next command, ensuring that the next instantiation differs in the value of at least one classifying term from preceding outputs. The process continues until no further (diverse) outputs can be found.

As future work, we plan to automate this approach and to implement it in Alloy, so that the overall process of computing an instantiation, evaluating classifying terms and generating predicates for the next command is performed automatically. Furthermore, other textual modeling approaches like B or Event-B [10] could be explored, checking whether the idea of classifying terms can be applied for them as well.

## References

1. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: On parallel scalable uniform SAT witness generation. In: TACAS'15. pp. 304–319 (2015)

2. Cunha, A., Garis, A.G., Riesco, D.: Translating between alloy specifications and UML class diagrams annotated with OCL. Software and System Modeling **14**(1), 5–25 (2015). https://doi.org/10.1007/s10270-013-0353-5, https://doi.org/10.1007/s10270-013-0353-5
3. Cunha, A., Macedo, N., Guimaraes, T.: Target Oriented Relational Model Finding. In: Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411. pp. 17–31. Springer-Verlag New York, Inc. (2014)
4. Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. In: ICSE'18. pp. 549–559 (2018)
5. Ermon, S., Gomes, C., Selman, B.: Uniform solution sampling using a constraint solver as an oracle. In: UAI'12. pp. 255–264 (2012)
6. Ferdjoukh, A., Galinier, F., Bourreau, E., Chateau, A., Nebut, C.: Measurement and generation of diversity and meaningfulness in model driven engineering. International Journal On Advances in Software **11**(1/2), 131–146 (2018)
7. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. of Comp. Prog. **69**(1-3), 27–34 (2007)
8. Hilken, F., Gogolla, M., Burgueño, L., Vallecillo, A.: Testing models and model transformations using classifying terms. SoSyM **17**(3), 885–912 (2018)
9. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)
10. Körner, P., Leuschel, M., Meijer, J.: State-of-the-art model checking for B and event-b using prob and ltsmin. In: Furia, C.A., Winter, K. (eds.) Integrated Formal Methods - 14th Int. Conf., IFM 2018, Proceedings. Lecture Notes in Computer Science, vol. 11023, pp. 275–295. Springer (2018). https://doi.org/10.1007/978-3-319-98938-9_16, https://doi.org/10.1007/978-3-319-98938-9_16
11. Montaghami, V., Rayside, D.: Bordeaux: A Tool for Thinking Outside the Box. In: Fundamental Approaches to Software Engineering (FASE 2017). pp. 22–39. Springer, Berlin, Heidelberg (2017)
12. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 232–241. IEEE (5 2013). https://doi.org/10.1109/ICSE.2013.6606569, http://ieeexplore.ieee.org/document/6606569/
13. Semeráth, O., Varró, D.: Iterative Generation of Diverse Models for Testing Specifications of DSL Tools. In: FASE'18. pp. 227–245. Springer, Cham (4 2018)
14. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: TACAS'07. pp. 632–647 (2007)
15. Wu, H.: Generating metamodel instances satisfying coverage criteria via smt solving. In: 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD). pp. 40–51. IEEE (2016)