A Tool for Debugging Unsatisfiable Integrity Constraints in UML/OCL Class Diagrams

Juan Antonio Gómez-Gutiérrez[1], Robert Clarisó[1], and Jordi Cabot[2]

[1] Universitat Oberta de Catalunya, Barcelona, Spain,
{juanto,rclariso}@uoc.edu
[2] ICREA, Barcelona, Spain,
jordi.cabot@icrea.cat

Abstract. Software models are the basis of the Model-Driven Engineering paradigm. The most popular modeling notation is UML class diagrams, which can be annotated with OCL predicates to describe complex integrity constraints.
When creating and managing UML/OCL models, a challenge for domain engineers is diagnosing faults. Problems like inconsistencies among integrity constraints can render a model useless. While existing verification tools provide ample support for detecting faults, users have less support when trying to understand and fix them. In this paper, we present a tool aimed at helping domain engineers locate, understand and fix faults in UML/OCL class diagrams. This tool is built as a plug-in within an existing UML modeling tool, the UML Specification Environment (USE).

Key words: UML, OCL, class diagram, verification, integrity constraint, USE, model debugging

1 Introduction

In the software development process, the relevant characteristics of a system can be captured using a *model*, *e.g.*, a UML diagram. Software models are powerful tools for communication among stakeholders and documentation of design decisions. Moreover, in the Model-Driven Development paradigm, models are the central asset of the software development process, from which other assets like source code are (semi)automatically derived. As a result, the correctness of software models affects the quality of the final software product.

UML class diagrams are a popular notation for modeling structural features. This formalism can be enriched by defining complex integrity constraints using the Object Constraint Language (OCL). OCL is a textual notation that enables the definition of class invariants and pre/postconditions for operations.

As UML/OCL models grow more complex, it may be necessary to check that there are no inconsistencies, *e.g.*, constraints that become unsatisfiable due to the interactions with other constraints. Detecting such errors is a complex task. Furthermore, it is even harder to understand their causes in order to repair the model, rewriting the incorrect constraints in a proper way.

In this paper, we present MVM (Model Validator Mixer)[1] , a modeling tool for domain engineers that helps them locate, understand and fix consistency problems in UML/OCL class diagrams. To this end, MVM computes and organizes information about groups of inconsistent constraints and sample instances that satisfy most (but not all) integrity constraints. MVM is implemented as a plug-in for the UML Specification Environment (USE) [1], a modeling environment offering advanced features for the verification and validation of UML/OCL models.

2 Related Work

Several works have considered the formal verification of UML class diagrams annotated with OCL constraints, *e.g.*, PLEDGE [2], USE Model Validator [3], UMLtoCSP/EMFtoCSP [4, 5] or AuRUS [6], among others. These tools can check correctness properties like *finite satisfiability*, *i.e.*, whether there is a finite instance of the class diagram that satisfies all UML and OCL integrity constraints simultaneously. If the property holds, an example instance is computed as output, otherwise the method warns about the lack of satisfying instances. Other tools such as the Alloy Analyzer [7] or VIATRA [8] can check equivalent properties for closely related conceptual modeling formalisms.

Nevertheless, these tools focus on either detecting faults efficiently or generating high quality example instances (realistic, diverse, . . .) [2, 9]. Thus, once a fault has been detected there is little support to help the designer locate, understand and fix the problem(s). In the following, we discuss three approaches that work in this direction: *unsatisfiable cores*, *max-satisfiabilty* and *model repair*.

An unsatisfiable core is a subset of integrity constraints that is unsatisfiable. If the class diagram is inconsistent, it is possible to compute a small (or even minimal) unsatisfiable core of integrity constraints, helping to locate the fault [10, 6, 11]. These techniques can be used for *model debugging* (also called *fault localization*): identifying the fragment(s) of a specification that are causing the fault [12, 13]. Conversely, using a maximum satisfiability algorithm it is possible to compute the largest set of constraints that can be satisfied simultaneously [14, 15].

A final set of techniques aim to generate repairs, *i.e.* small (or even minimal) changes to a model that fix a particular fault. Some of these methods have targeted Alloy specifications [12, 13, 16] and UML/OCL class diagrams [17]. As a drawback, the catalog of fixes (mutation operators) has to be established a priori and, in some cases, additional predicates to validate the fix must be defined. Instead, our approach aims to help the designer locate, understand and fix faults. That is, it does not assume that the model is almost correct and can be fixed with small updates. Information such as unsatisfiable cores and max-satisfiability is combined with examples and useful feedback to help the designer understand the fault.

[1] You can download the tool at: https://github.com/juanto2021/MVM#readme

3 Presentation of MVM

3.1 Context

Our goal with MVM is helping domain engineers debug problems with their UML/OCL class diagrams. Rather than offering a stand-alone tool with yet another syntax and GUI for creating model, we have aimed at extending an existing toolkit. Thus, we have integrated MVM inside the UML Specification Environment (USE).

USE already offers several features for the verification and validation of UML/OCL class diagrams. First, it uses a textual syntax for enconding both the class diagram and the OCL constraints. Then, it offers GUIs to visualize the class diagram, instances (object diagrams) and constraints. Moreover, it can check whether an object diagram satisfies all or some integrity constraints. And, finally, it includes a plug-in called Model Validator [3] that can determine if the constraints are satisfiable or not. To this end, it uses a bounded verification solver that constructs a valid instance of the class diagram (satisfiable) or reports its absence within the bounded domains (unsatisfiable). When the result is unsatisfiable, no further feedback is provided by USE (see Section 2 for the feedback provided by other extensions).

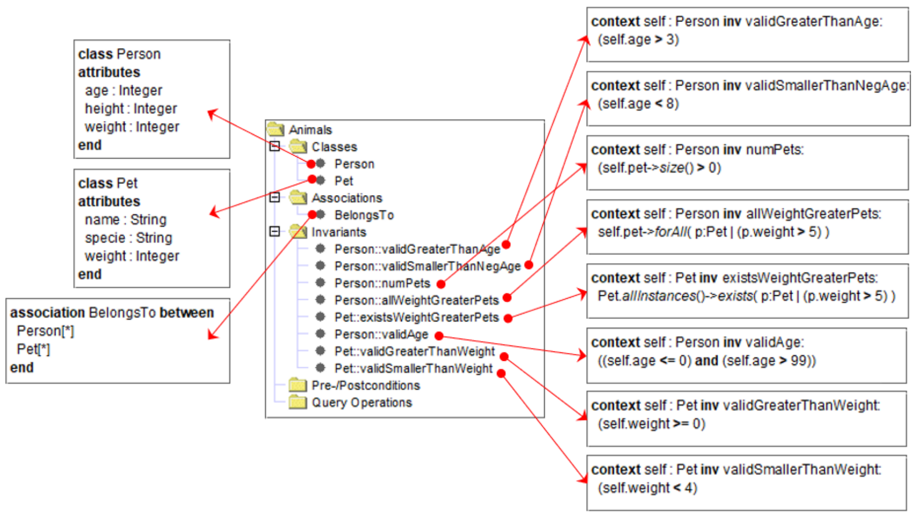
MVM will supply feedback aimed at helping domain engineers diagnose and fix the problem. It will use USE's notation to describe the UML/OCL class diagrams and USE's GUI to visualize sample instances.

3.2 Feedback

A UML/OCL class diagram may contain one or more consistency errors that need to be fixed. All of them will exhibit the same symptom (the model is unsatisfiable), but their causes should be fixed independently.

Each consistency error may be caused by a single incorrect invariant or an unintended interaction between several invariants. To this end, we will provide the following information to the domain engineer:

– *Minimal unsatisfiable cores:* Sets of OCL invariants that cannot be simultaneously satisfied and that become satisfiable if any member of the set is removed. While each unsatisfiable core is potentially an independent error, several cores that share some constraints may indicate a problem in the constraints included in their intersection.
– *Max-satisfiable constraints and instances:* Sets of OCL invariants that can be satisfied as a whole, together with sample instances satisfying only those max-satisfiable constraints. The goal is showing the domain engineer what such instances would look like, in order to help him have a better idea of how the current constraints should be modified.
– *"What-if" scenarios:* Sample instances that would be legal if one constraint in the unsatisfiable core is dropped. Again, the rationale is helping the domain engineer figure out whether such instances should be made valid by



```
class Person
attributes
  age : Integer
  height : Integer
  weight : Integer
end

class Pet
attributes
  name : String
  specie : String
  weight : Integer
end

association BelongsTo between
  Person[*]
  Pet[*]
end
```

```
context self : Person inv validGreaterThanAge:
  (self.age > 3)

context self : Person inv validSmallerThanNegAge:
  (self.age < 8)

context self : Person inv numPets:
  (self.pet->size() > 0)

context self : Person inv allWeightGreaterPets:
  self.pet->forAll( p:Pet | (p.weight > 5) )

context self : Pet inv existsWeightGreaterPets:
  Pet.allInstances()->exists( p:Pet | (p.weight > 5) )

context self : Person inv validAge:
  ((self.age <= 0) and (self.age > 99))

context self : Pet inv validGreaterThanWeight:
  (self.weight >= 0)

context self : Pet inv validSmallerThanWeight:
  (self.weight < 4)
```

Animals
Classes
Person
Pet
Associations
BelongsTo
Invariants
Person::validGreaterThanAge
Person::validSmallerThanNegAge
Person::numPets
Person::allWeightGreaterPets
Pet::existsWeightGreaterPets
Person::validAge
Pet::validGreaterThanWeight
Pet::validSmallerThanWeight
Pre-/Postconditions
Query Operations

Fig. 1. Model Animals.

rewriting the corresponding constraint.For example, in Figure 2 you can see in the rightmost tab which combinations are satisfiable if you eliminate this invariant. Also, double-clicking on any of the proposed combinations creates an object diagram with a sample valid instance.

The central idea is presenting this information in a cohesive and usable way that helps the user understand the consistency problems that need to be addressed, their causes and candidate repairs.

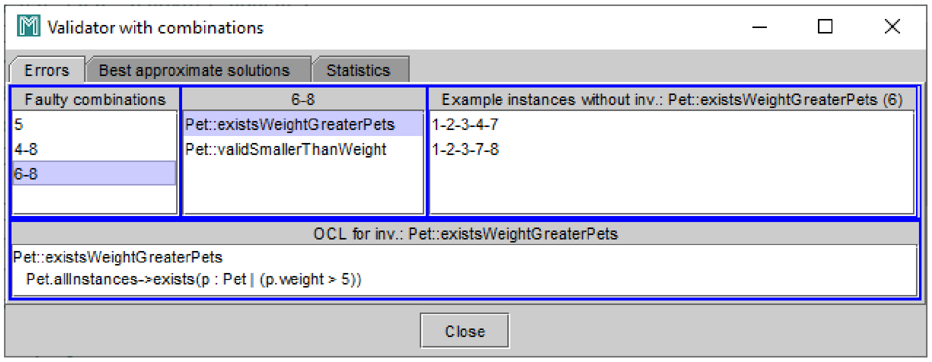
3.3 Running example

In order to illustrate the operation of MVM, we will use the UML class diagram in Figure 1 as our running example. It contains 2 classes (Person and Pet), each with different attributes, several invariants and an association. This model fragment could reflect work in progress to add new features to an existing system.

This class diagram has two separate consistency issues. First, invariant `validAge` cannot be satisfied. Age was probably intended to be in the range from 1 to 99 years, but the relational operators got reversed by mistake:

```
  context Person inv validAge:              -- Invariant 5
     self.age <= 0 and self.age > 99
```

The second problem affects invariants related to the weight of pets: one (`validSmallerThanWeight`) establishes an upper bound of the weight of pets, while two others require the existence of a heavier pet (`existsWeightGreaterPets`) or require the pets owned by some person to be heavier (`allWeightGreaterPets`). Intuitively, the upper bound for the weight should be increased, or the requirements on heavier pets be lowered.



Faulty combinations	6-8	Example instances without inv.: Pet::existsWeightGreaterPets (6)
5	Pet::existsWeightGreaterPets	1-2-3-4-7
4-8	Pet::validSmallerThanWeight	1-2-3-7-8
6-8		

OCL for inv.: Pet::existsWeightGreaterPets

```
Pet::existsWeightGreaterPets
  Pet.allInstances->exists(p : Pet | (p.weight > 5))
```

Fig. 2. Errors tab in the MVM dialog box.

```
context Person inv allWeightGreaterPets: -- Invariant 4
   self.pet→forAll(p|p.weight>5)

context Pet inv existsWeightGreaterPets: -- Invariant 6
   Pet.allInstances()→exists(p:Pet|p.weight>5)

context Pet inv validSmallerThanWeight:  -- Invariant 8
  self.weight < 4
```

In the following, we will show how MVM can be used to identify these issues and help domain engineers come up with potential solutions.

3.4 User interface

MVM displays the information about consistency errors in a dialog box consisting of three tabs: *Errors*, *Best approximate solutions* and *Statistics*.

Errors In this tab, we show the minimal combinations of invariants that are unsatisfiable (minimal unsatisfiable cores). It consists of the following panels:

– Faulty combinations: The leftmost panel shows the minimal unsatisfiable core. When a combination is selected in this list, the following two views are synchronized.
– Example instances without the selected invariant: This panel shows examples of satisfiable combinations that do not contain one invariant from the core. Double-clicking a combination (each excludes one invariant from the core) creates an object diagram that satisfies the invariant in that combination.
– OCL for inv: For convenience, this panel displays the OCL definition of the selected invariant.

Figure 2 depicts this user interface for the running example. In this dialog box you can see in the upper left part, a list that shows three faulty combinations of invariants that cannot be satisfied:

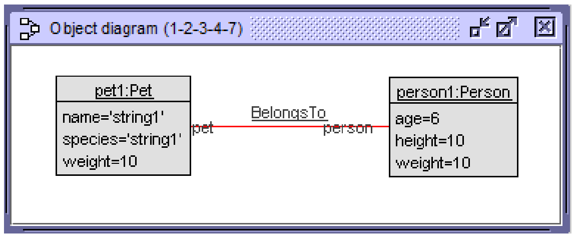


Fig. 3. Object diagram depicting a scenario where invariants 5 and 7 are violated.

- 5 (validAge): This single invariant is unsatisfiable on its own.
- 4-8 (allWeightGreaterPets, validSmallerThanWeight): Even though each invariant is satisfiable on its own, the combination of both is unsatisfiable.
- 6-8 (existsWeightGreaterPets, validSmallerThanWeight): Same as before.

The first core is disjoint from the rest, so this is a separate error that should be repaired independently. Regarding the last two cores, their intersection suggests a potentially shared cause within invariant 8 (`validSmallerThanWeight`). In order to understand these faults, the domain engineer can inspect instances that violate only one of the constraints in this unsatisfiable core, using the rightmost panel. For instance, if we are studying invariant 5 (`validAge`), we can inspect an instance that satisfies the combination of invariants 1-2-3-4-6-7 (which excludes 5). Figure 3 shows the object diagram depicting such instance, as shown in USE.

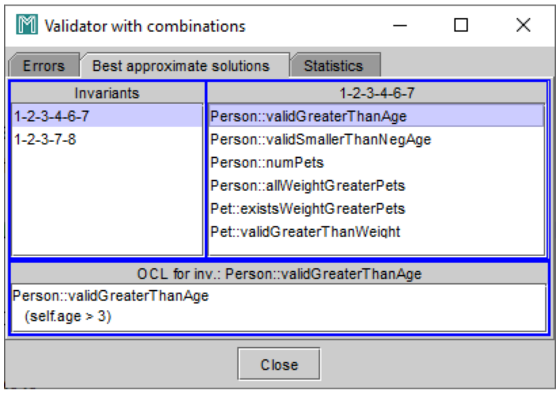
Best approximate solutions This tab shows the satisfiable combinations with the highest number of invariants:

- Invariants: The leftmost panel shows the list of satisfiable combinations with the highest number of invariants.
- Combination panel: When clicking on a combination, the invariants that compose it are shown in the upper right panel.
- OCL for inv: When clicking on a specific invariant, the definition of that invariant is shown in the lower panel.

Figure 4 shows the information this tab. In a similar way to that described for the "Errors" tab, when clicking on a combination, the invariants that compose it will be shown in the upper right list and, when clicking on a specific invariant, its definition will be shown in the lower part.

Statistics The computation of unsatisfiable cores relies on USE's Model Validator to check if a given combination of invariants is satisfiable or not. If a combination of invariants is deemed unsatisfiable, supersets of this combination will also be unsatisfiable. Similarly, if a combination is found to be satisfiable, it is not necessary to explore subsets of this combination. Thus, it is not necessary to invoke the Model Validator for *each* combination: many calls can be pruned.

This tab shows information about the computation of unsatisfiable cores and sample instances. It describes the CPU time spent searching for combinations,


Validator with combinations

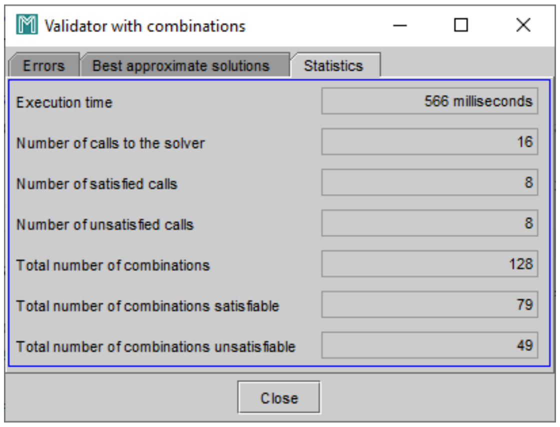
Invariants
1-2-3-4-6-7
1-2-3-7-8

1-2-3-4-6-7
Person::validGreaterThanAge
Person::validSmallerThanNegAge
Person::numPets
Person::allWeightGreaterPets
Pet::existsWeightGreaterPets
Pet::validGreaterThanWeight

OCL for inv.: Person::validGreaterThanAge

```
Person::validGreaterThanAge
  (self.age > 3)
```

Fig. 4. Best approximate solutions.


Validator with combinations

Execution time	566 milliseconds
Number of calls to the solver	16
Number of satisfied calls	8
Number of unsatisfied calls	8
Total number of combinations	128
Total number of combinations satisfiable	79
Total number of combinations unsatisfiable	49

Fig. 5. Statistics

the number of calls to the solver, and the number of calls that produced a satisfiable/unsatisfiable result. Figure 5 depicts this panel.

4 Conclusions And Future Work

In this paper we have presented MVM, a tool for debugging consistency errors in UML/OCL class diagrams. This tool complements existing UML/OCL tools such as USE by detailing which are the unsatisfiable invariants and the best possible combinations, giving feedback to the user in the form of potential instances for relevant scenarios.

As future work, we will improve the usability of the user interface, *e.g.*, providing step-by-step suggestions to fix the model; and improve the efficiency of the calculation, by introducing different strategies for enumerating unsatisfiable cores and proposing heuristics tailored for OCL invariants.

Acknowledgements. This work is partially funded by the Spanish Agencia Estatal de Investigación through the project "LOw-COde development of Smart Software" (LOCOSS, PID2020-114615RB-I00/AEI/10.13039/501100011033).

References

1. Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.
2. Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C Briand. Practical constraint solving for generating system test data. *ACM TOSEM*, 29(2):1–48, 2020.
3. Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In *TOOLS'11*, pages 290–306. Springer, 2011.
4. Carlos A. González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. EMFtoCSP: A tool for the lightweight verification of EMF models. In *FormSERA'12*, pages 44–50, 2012.
5. Jordi Cabot, Robert Clarisó, and Daniel Riera. On the verification of UML/OCL class diagrams using constraint programming. *JSS*, 93:1–23, 2014.
6. Guillem Rull, Carles Farré, Anna Queralt, Ernest Teniente, and Toni Urpí. AuRUS: explaining the validation of UML/OCL conceptual schemas. *SoSyM*, 14(2):953–980, 2015.
7. Daniel Jackson. *Software abstractions: logic, language, and analysis.* MIT Press, 2012.
8. Oszkár Semeráth, András Szabolcs Nagy, and Dániel Varró. A graph solver for the automated generation of consistent domain-specific models. In *ICSE'18*, pages 969–980, 2018.
9. Oszkár Semeráth and Dániel Varró. Iterative generation of diverse models for testing specifications of DSL tools. In *FASE'18*, volume 18, pages 227–245, 2018.
10. Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM'08*, pages 326–341. Springer, 2008.
11. Nils Przigoda, Robert Wille, and Rolf Drechsler. Analyzing inconsistencies in UML/OCL models. *J. of Circuits, Systems and Computers*, 25(03):1640021, 2016.
12. Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Automated model repair for Alloy. In *ASE'18*, pages 577–588. IEEE, 2018.
13. Guolong Zheng, Hamid Bagheri, and ThanhVu Nguyen. Debugging declarative models in Alloy. In *ICSME'20*, pages 844–848. IEEE, 2020.
14. Hao Wu. MaxUSE: a tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In *iFM'17*, pages 348–356. Springer, 2017.
15. Changjian Zhang, Ryan Wagner, Pedro Orvalho, David Garlan, Vasco Manquinho, Ruben Martins, and Eunsuk Kang. AlloyMax: bringing maximum satisfaction to relational specifications. In *ESEC-FSE'21*, pages 155–167, 2021.
16. Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo Frias. Bounded exhaustive search of Alloy specification repairs. In *ICSE'21*, pages 1135–1147. IEEE, 2021.
17. Robert Clarisó and Jordi Cabot. Fixing defects in integrity constraints via constraint mutation. In *QUATIC'18*, pages 74–82. IEEE, 2018.