Software and Systems Modeling
https://doi.org/10.1007/s10270-025-01272-7

SPECIAL SECTION PAPER

Developing configurations and solutions for logical puzzles with UML and OCL

Martin Gogolla[1] · Jesús Sánchez Cuadrado[2]

Received: 20 March 2024 / Revised: 20 January 2025 / Accepted: 23 January 2025

Abstract
Logical puzzles can be important factors for the development of rational analysis and application capabilities for pupils and students. Therefore, logical puzzles can also take a prominent supporting role in computer science education. This contribution proposes a UML class model with accompanying OCL constraints for developing logical puzzles. The class model acts as a metamodel for the description of the basic puzzle organization and the logical clues presented to the learners. The constraints express, for example, statements about uniqueness of solutions, and the degree of puzzle complexity may be tuned by appropriate model elements. Given a puzzle specification which simply comprises the domain elements of the puzzle plus constraints, our implementation uses a UML and OCL solver to construct a puzzle instance (i.e., a set of clues and solutions) automatically. The puzzle is made playable by a graphical user interface. We have validated this approach for developing puzzles by building several puzzles from the literature of increasing complexity and performed a student survey.

Keywords Logical puzzle · Formal method · UML · OCL · Metamodel

1 Introduction

For teaching mathematics and computer science, puzzles are regarded as valuable tools [2]. They provide a practical setting (i.e., a realistic problem) over which one can practice abstract mathematical concepts. On the one hand, they are useful for improving problem-solving skills. On the other hand, puzzles may improve engagement and motivation since they challenge students and boost their inherent curiosity, and also are a way of training their persistence on tackling difficult tasks.

The goal of this work is to apply modeling techniques to the automated development of logical puzzles which could be used to support teaching methods for computer science and mathematics, but also for other disciplines and rationality in general. We propose an automatic approach for formulating and solving logical puzzles, i.e., logical problems that can be solved through deductive reasoning. The focus is on teaching modeling and formal methods to university students, but we also take the perspective to apply puzzles to teach elementary formal reasoning to school pupils. For expressing models, we employ the unified modeling language (UML [21]) and the object constraint language (OCL [22], [4]). As the underlying modeling tool, we take advantage of the system USE (UML specification environment [8], [9]). As a central component, USE offers a model finder that can automatically construct object models for a class model with OCL constraints (it can also complete partial object models). To give an example for applying the model finder, consider one builds a metamodel (1) for state machines including states and transitions and (2) for running state machines on the basis of transition inputs. Then, the model finder is able to automatically construct, for a given domain, a state machine and one or more example executions of the state machine in the specified domain (see the behavioral example in [13]).

In order to achieve the goal of modeling and automatically deriving logical puzzles, our approach is based on a metamodel, which allows a puzzle developer to formulate the domain of the puzzle and the vocabulary. For example, a puzzle could be about car colors or could be about actors in a fantasy novel. Moreover, through optional OCL con-

Communicated by Javier Troya and Alfonso Pierantonio.

✉ Martin Gogolla
gogolla@uni-bremen.de

Jesús Sánchez Cuadrado
jesusc@um.es

1 University of Bremen, Bremen, Germany

2 Universidad de Murcia, Murcia, Spain

straints the developer can specify relationships between the elements of the domain. The research challenge lies in automatically constructing logical puzzles in a flexible and easy way for various domains, including automatically building clues, for which various degrees of difficulty can be stated. We apply the USE model finder to automatically instantiate concrete puzzles. Instantiation of a puzzle means deriving a consistent set of domain elements and clues. Such clues would be presented to the puzzle player and lead typically to a unique solution. To make the puzzles playable, we have built a simple user interface on top of the USE model. To validate our approach, we have implemented several puzzles proposed by the literature and available in the web, and we have performed a user study with university students.

Our contribution lies in proposing a conceptual model and method for building logical puzzles, providing an implementation and validating the approach with various techniques such as comparing and implementing existing puzzles and performing a user study. By applying the constructed logical puzzles generally in education, not only in Computer Science, one can train rational thinking, analysis and development through using the puzzles in lectures, exercises and examinations. The simplicity of using puzzle notions from a particular domain and the option to tune puzzle difficulty to a particular audience gives the puzzle construction process a low-threshold character.

This contribution is structured as follows. Section 2 gives an overview on our approach by explaining it with a simple example that will be used throughout the paper. Section 3 puts forward the essential model details by discussing how to design, solve and model puzzles. Section 4 turns to practical aspects, and in particular describes an implementation. Section 5 validates the approach by showing in particular known examples from the literature and by reporting on a student survey. Section 6 discusses related work. The paper ends with a conclusion section and also elaborates future work.

2 The basic idea

In this section, we describe our approach to model puzzles using UML and OCL. We use a running example to show the different elements involved. In Sect. 2.1, we discuss how puzzle elements like possible solutions or clues are designed, Sect. 2.2 turns to questions how to solve puzzles, and Sect. 2.3 debates how to generally meta-model and represent puzzles with a UML class model (with classes, attributes, associations and operations) and restricting OCL constraints regarding aspects concerning representation (syntactic aspects) and content (semantic aspects).

2.1 Designing puzzles

A typical logical puzzle, namely a problem that can be solved through deductive reasoning, is shown in Fig. 1. The left upper part visualizes the possible puzzle solutions, and in the left lower part four clues are stated that determine the solution. The right part represents the puzzle in the form of a UML and OCL class model along with OCL invariants that determine exactly one allowed object diagram. The UML and OCL representation is only shown to make the puzzle precise and to show a possible puzzle implementation. One does not need OCL to understand the puzzle task. Often, for better motivation, the puzzle is embedded into a real-life context. In this example, the puzzle could be stated as follows:

> A bank robbery has happened, and the police is consulting four witnesses about the colors of the car that the robbers have used since this information will help them solve the case. The witnesses all tell the truth; however, one cannot expect that all observations are helpful.

Here, observations are stated in the form of conditional statements, as shown in Fig. 1. Statements could also be non-conditional (like `top <> blue`, i.e., *the top of the car was not blue*). In this case, the task for the person solving the puzzle, say the police inspector, is to identify on the basis of the observations the top and body color of the car. Here, this is possible and leads to a unique solution, if done properly. Not all sets of clues do have a solution or do have a single solution. This depends on the stated clues.

To model this puzzle, we use the USE tool [8], [9]) which provides us with facilities to construct UML class models with OCL constraints. USE supports both textual and visual representation of models, enabling the analysis and inspection of model properties through derivation and deduction. The goal in the tool USE is the prediction of system properties from the model before actually building an implementation. USE offers to the developer a graphical user interface (GUI) and a command line interface (CLI). Models that are class-oriented (class and protocol state-machine models) and that are instance-oriented (object models as well as sequence and communication models) can be studied in USE. A model finder for class models including OCL invariants supports object model finding and partial object model completion. Thus, the tool comprises (1) an OCL evaluator that determines the value of an OCL expression in an object model (object diagram), and (2) a model finder, that automatically constructs object models for a given class model and a set of OCL invariants or reports *unsatisfiable*, if contradicting constraints are stated, for example.

The elements from Fig. 1 are formally represented as automatically generated (details will follow later in Sect. 3) USE objects as shown in Fig. 2. The UML and OCL parts from the right of Fig. 1 possess a formal counterpart in Fig. 2. For example, a color is represented as a possible `value` entry of a `Property` object. Each of the nine possible solutions is shown as a `Thing` object and each of the four clues as a `Clue` object. `Property` objects with attributes `name` and `value` describe `Thing` objects, e.g., `thing3` is linked to two `Property` objects, the first one `property3` with `name= 'top'` and `value='red'` and the second one `property5` with `name='body'` and `value='red'`. In order to simplify the diagram, four property objects are hidden. The puzzle solution is represented by the yellow marked `Thing` object `thing3` that is the only `Thing` object satisfying all clues. The remaining `Thing` objects are not solutions to the puzzle and this is indicated by a list of banning clues in the northeast object rectangle corner (details on how to achieve the solution will follow later in Fig. 3).

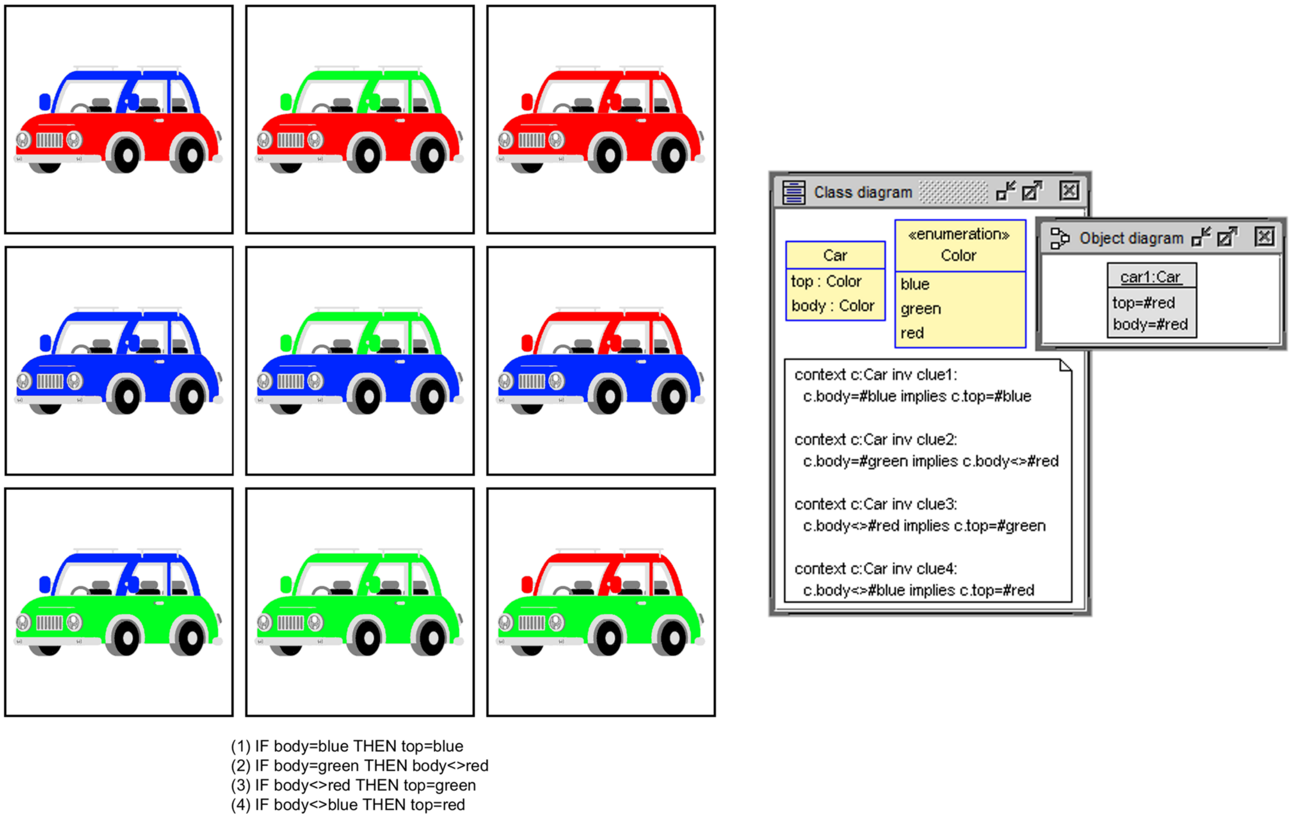
In the lower part of Fig. 2, four OCL queries and their results are shown. The queries discover central aspects of our approach.

1. The first query represents the four clues in human readable form, namely in terms of a string sequence considering the clues (in order of object identity `clue1`, `clue2`, ...) using the derived attribute `toStr`; for example, the third `Clue` object `clue3` is represented as `body <> red => top = green` (In the running text, a sloppy notation for string values without apostrophes is used).
2. The second query selects among the `Thing` objects those in which *all* clues are valid; in this case, there is only a single `Thing` object satisfying all clues, the unique puzzle solution, namely the object `thing3` indicated with yellow color in the upper part of the figure.
3. The third and fourth queries can be understood as constituting a derived association between classes `Thing` and `Clue`. We will use the term *ban* as follows: a clue is *banning* a thing (a possible solution) if violation of the clue excludes the thing as a solution; in the inverse direction, we will say that a thing is *banned by* a clue. The third query indicates for each `Thing` object (in order `thing1`, `thing2`, ...) the set of `Clue` objects that ban the `Thing` object, or in other words, the query yields the set of clues where each single clue disqualifies the considered `Thing` object as a puzzle solution; e.g., (a) `thing7` is banned by `clue3` and `clue4` and (b) `thing3` is banned by no `Clue` object, i.e., the empty set of `Clue` objects indicated by yellow in the query result; the result of this query is also indicated in the upper figure part in the small rectangles drawn northeast next to the `Thing` object rectangles.
4. The fourth query indicates for each `Clue` object the set of `Thing` objects that the `Clue` object is banning; e.g., (a) `clue1` is banning `thing5` and `thing6` and (b) `clue2` is banning no `Thing` object, i.e., the empty set of `Thing` objects, also indicated with yellow in the query result; because `clue2` has this particular characteristic this object is also marked with yellow in the object diagram; thus `clue2` does not give additional restrictions; one would yield the same solution without `clue2`. These four OCL queries show central aspects of the approach insofar that (a) the puzzle clues, (b) the unique puzzle solution and (c) the relationship between the potential solutions (the things) and the clues, which ban them from being the actual solution, are studied.

2.2 Solving puzzles

A systematic method for solving the puzzle is shown in Fig. 3. To illustrate the method, the possible puzzle solutions already shown in Fig. 1 are presented four times but complemented by additional features to indicate each step in the process. In each of the steps, one clue is applied and the result is captured by the black and gray tokens inside one of the nine possible solutions:

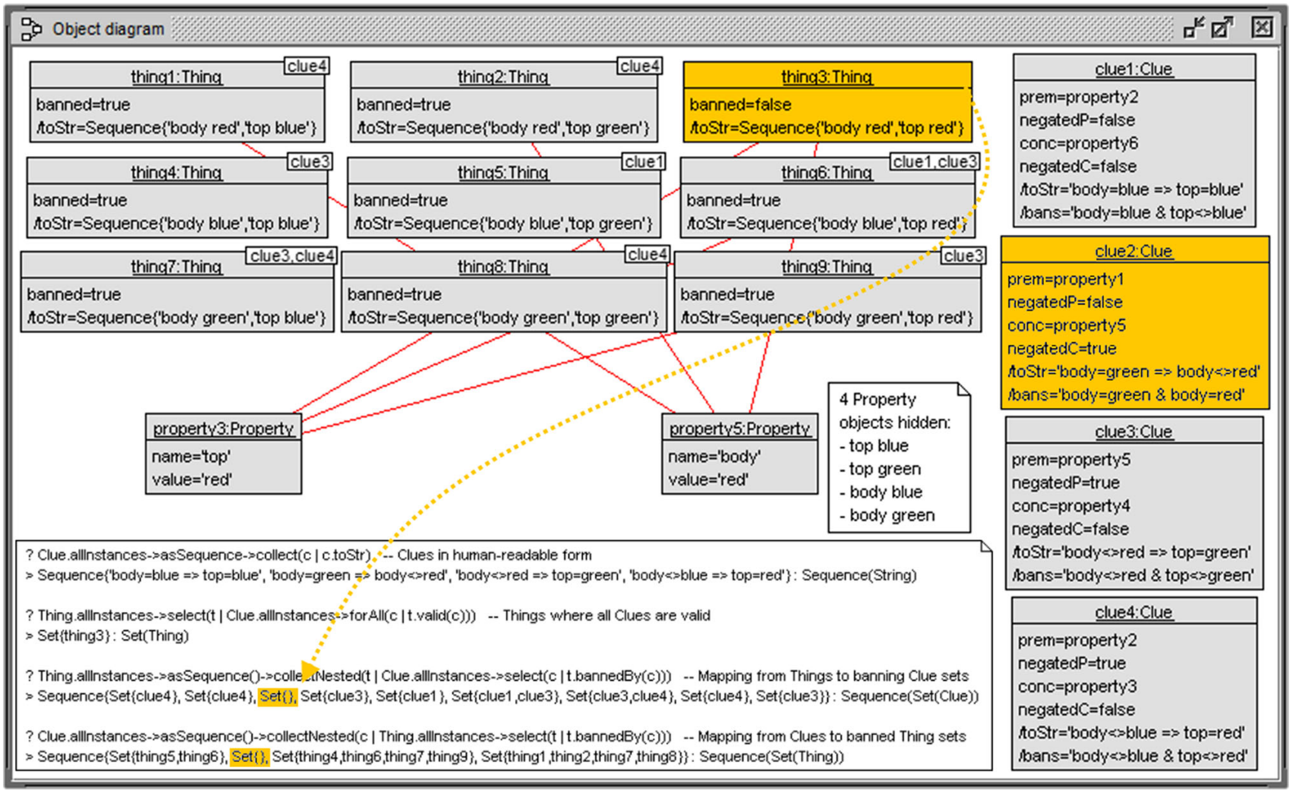
1. In the first step, the first clue `IF body=blue THEN top=blue` is applied. The clue is logically equivalent to `body <> blueORtop = blue`, and its logical negation is `body = blueANDtop <> blue`. All clues are expected to be valid in the solution. Therefore, a possible solution that satisfies the negation of the clue can be ruled out as being the solution (i.e., *the thing is banned*). The ruling out step is indicated by the two black tokens in the possible solutions that satisfy the negation of the first clue. The two black tokens ban two possible solutions and mark them as impossible for becoming the solution.
2. In the second step, the second clue is applied. As a forward-looking remark, we state that some readers might classify this step and this clue as *odd*. (2a) The black marks from the previously detected banned solutions turn its color to gray. (2b) New black marks would be placed for new banned solutions from the second clue; however, in this case, the negation of the second clue says `body=green AND body=red`; there is no possible solution that satisfies this condition; accordingly, no new black mark is introduced (as will be done in the next step).
3. In the third step, the third clue is applied. The negation of the third clue is `body <> redANDtop <> green`. Accordingly, four new black marks for the four possible solutions satisfying the negated clue are introduced; the sixth possible solution was marked gray and now additionally becomes marked black.
4. In the fourth step, the fourth clue is applied. The negation of the last clue additionally bans four possible solutions with `body <> blueANDtop <> red`. This



```
context c:Car inv clue1:
  c.body=#blue implies c.top=#blue

context c:Car inv clue2:
  c.body=#green implies c.body<>#red

context c:Car inv clue3:
  c.body<>#red implies c.top=#green

context c:Car inv clue4:
  c.body<>#blue implies c.top=#red
```

(1) IF body=blue THEN top=blue
(2) IF body=green THEN body<>red
(3) IF body<>red THEN top=green
(4) IF body<>blue THEN top=red

Fig. 1 Things and clues



thing1:Thing [clue4]
banned=true
/toStr=Sequence{'body red','top blue'}

thing2:Thing [clue4]
banned=true
/toStr=Sequence{'body red','top green'}

thing3:Thing
banned=false
/toStr=Sequence{'body red','top red'}

thing4:Thing [clue3]
banned=true
/toStr=Sequence{'body blue','top blue'}

thing5:Thing [clue1]
banned=true
/toStr=Sequence{'body blue','top green'}

thing6:Thing [clue1,clue3]
banned=true
/toStr=Sequence{'body blue','top red'}

thing7:Thing [clue3,clue4]
banned=true
/toStr=Sequence{'body green','top blue'}

thing8:Thing [clue4]
banned=true
/toStr=Sequence{'body green','top green'}

thing9:Thing [clue3]
banned=true
/toStr=Sequence{'body green','top red'}

property3:Property
name='top'
value='red'

property5:Property
name='body'
value='red'

4 Property
objects hidden:
- top blue
- top green
- body blue
- body green

clue1:Clue
prem=property2
negatedP=false
conc=property6
negatedC=false
/toStr='body=blue => top=blue'
/bans='body=blue & top<>blue'

clue2:Clue
prem=property1
negatedP=false
conc=property5
negatedC=true
/toStr='body=green => body<>red'
/bans='body=green & body=red'

clue3:Clue
prem=property5
negatedP=true
conc=property4
negatedC=false
/toStr='body<>red => top=green'
/bans='body<>red & top<>green'

clue4:Clue
prem=property2
negatedP=true
conc=property3
negatedC=false
/toStr='body<>blue => top=red'
/bans='body<>blue & top<>red'

```
? Clue.allInstances->asSequence->collect(c | c.toStr)   -- Clues in human-readable form
> Sequence{'body=blue => top=blue', 'body=green => body<>red', 'body<>red => top=green', 'body<>blue => top=red'} : Sequence(String)

? Thing.allInstances->select(t | Clue.allInstances->forAll(c | t.valid(c)))   -- Things where all Clues are valid
> Set{thing3} : Set(Thing)

? Thing.allInstances->asSequence()->collectNested(t | Clue.allInstances->select(c | t.bannedBy(c)))   -- Mapping from Things to banning Clue sets
> Sequence{Set{clue4}, Set{clue4}, Set{}, Set{clue3}, Set{clue1 }, Set{clue1 ,clue3}, Set{clue3,clue4}, Set{clue4}, Set{clue3}} : Sequence(Set(Clue))

? Clue.allInstances->asSequence()->collectNested(c | Thing.allInstances->select(t | t.bannedBy(c)))   -- Mapping from Clues to banned Thing sets
> Sequence{Set{thing5,thing6}, Set{}, Set{thing4,thing6,thing7,thing9}, Set{thing1 ,thing2,thing7,thing8}} : Sequence(Set(Thing))
```

Fig. 2 Object model representation of ‘Things and clues’ and four central OCL queries

leaves exactly one possible solution not banned, i.e., one solution that is neither marked with a black or a gray token. This unmarked possible solution (car with red top and red body) becomes now the solution.

All elements in Fig. 3 taken together give a systematic, substantial overview on the development of the solution. Colors of the tokens indicate the impact of applying the current clue and the previous clues. The fact that the second clue does not ban any possible solution is reflected in Fig. 2 by the circumstance that the fourth OCL query gives for `clue2` as the result the empty set `Set{}`. The fact that some clues are banning more than one possible solution is reflected by the circumstance that two clues are occuring in the northeast corners of the `Thing` rectangles. Finally, the property of being definitively a solution is reflected in the figure by the circumstance that the `Thing` object `thing3` does not have any banning `Clue` list in its northeast corner; the property of being a unique solution is reflected by the circumstance that object `thing3` is the *only* `Thing` object without a banning `Clue` list in its northeast corner. Here, we have applied the clues in the order of their object identity (`clue1`, `clue2`,...). However, one could apply the clues in any order, leading to a different order and color for the marks, but leading to the same solution `thing3` and banning the same possible solutions.

2.3 Modeling puzzles

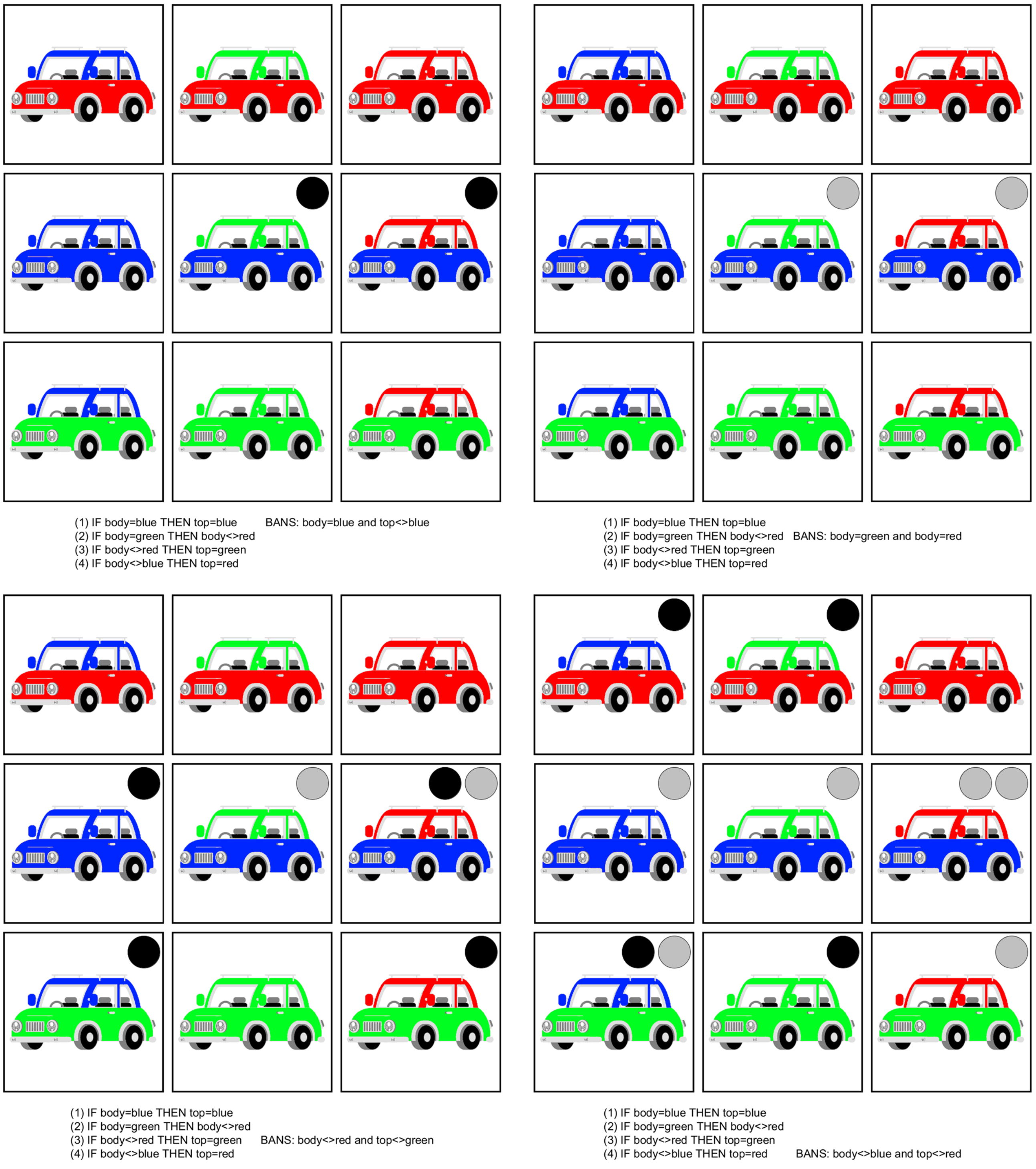
Our approach relies on a UML class model (metamodel), and a puzzle is modeled explicitly with an accompanying object model. In Fig. 4, the USE class model and informal explanations for model elements are presented. These elements have already appeared in the object model in Fig. 2 in the form of examples. Now, we discuss the classes `Thing`, `Property` and `Clue` in a systematic way, mostly independent from an example.

A `Thing` object (e.g., a car with particular top and body color) represents a possible solution for a modeled logical puzzle. A thing can be linked through the association `ThingProperty` to `Property` objects which describe with name-value pairs relevant, puzzle domain-specific aspects of a thing (e.g., a car with top-red and body-blue). A `Property` can state that a linked thing has for the property `name` the shape `value` and can be a universal attribute in the considered domain. In order to give a fresh example different from the used car domain, in a puzzle about potentially poisoned dinner courses there could be course properties like `name = 'Starter'` with `value='Soup'` representing the course `Starter=Soup` or properties like `name='Dessert'` with `value='Fruit'` representing the course `Dessert=Fruit`. A set of `Clue` objects give hints in order to identify a unique solution among the possible solutions, i.e., the set of things constituting the puzzle (e.g., (1) clue: if the car top is not blue, then the car body is green; (2) clue: the car body is not red). Clues refer to properties and can be (1) conditional with a premise `prem` and a conclusion `conc` (e.g., `top =′ red′ => body =′ green′`) or (2) unconditional (e.g., `body = 'red'`) in which case premise and conclusion are required to coincide in the formal object representation. Both premise and conclusion can be indicated as being negated with Boolean flags.

In Fig. 4, the model operations and the needed model invariants are sketched. The model operations allow to express central model functionalities in a compact way:

- `Thing::value(propertyP)`. For a `Thing`, the `value` of the `propertyP` in the `Thing`
- `Thing::valid(Clue)`. Whether the `Clue` is valid (i.e., true) in the `Thing`
- `Thing::bannedBy(Clue)`. Whether the `Thing` is banned by (or, in other words rejected by) the `Clue`.
- `Thing::cluesValid():Set(Thing)`. The set of all valid things, independent from the self-thing.
- `Clue::bansOCL()`. The OCL expression (in the form of a `String`) retrieving the things that are banned by all clues.
- `Clue::bansThing():Set(Thing)`. For a `clue`, the set of things that are banned by the `clue`.

The invariants are listed in the order of their context class and their name, additionally classifying them as belonging to (1) syntax, (2) semantics or (3) the loaded invariants (e.g., user defined). Although all invariants have formally the same shape as universally quantified formulas over the context class, we are distinguishing between syntax and semantics because the invariants serve different roles: Invariants about *syntax* make restrictions about the formal *representation* of entities from the puzzle and clue domain (e.g., all properties are syntactically different in the sense that no two different properties have the same attribute values for `name` and `value`); invariants about *semantics* restrict *evaluations* and properties of entities from the problem domain (e.g., how clues are evaluated in the possible solutions and how that evaluation is related to the `Thing` attribute `banned`; a central semantic invariant requires that there is only one valid solution, i.e., one `Thing` in which all `Clue` objects evaluate to true). The seven syntax invariants handle: (1) clue representation, (2) uniqueness of clues, (3) linking properties to things, (4) unique name-value pairs for properties, (5) linking things to properties, (6) things having unique properties and (7) thing properties referring to properties with different names. The two semantic invariants handle: (1) things where all clues are valid being not banned, and (2) existence of a unique solution in which all clues are valid. Finally, loaded invariants allow the user to add puzzle-specific constraints.



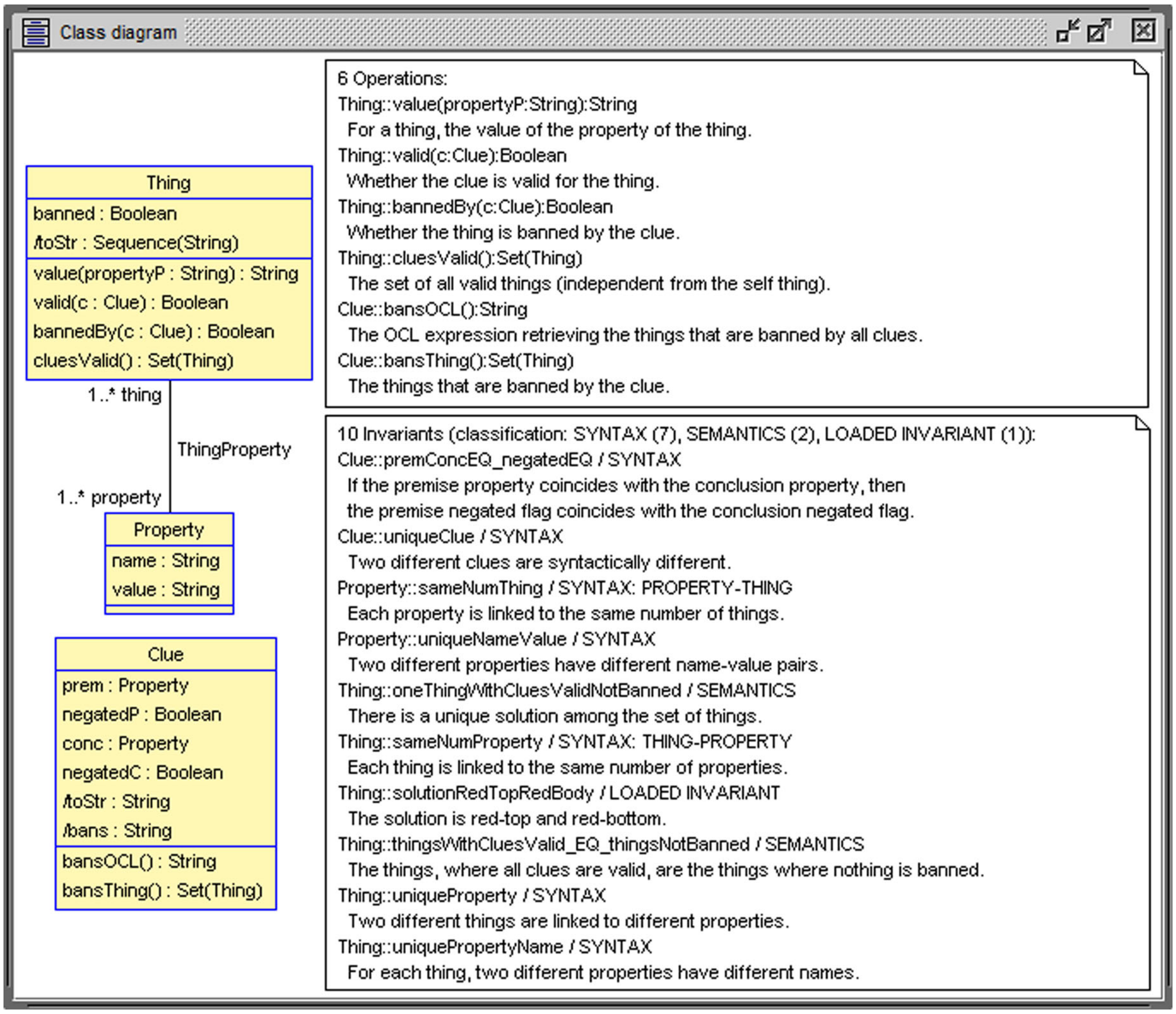
(1) IF body=blue THEN top=blue
(2) IF body=green THEN body<>red
(3) IF body<>red THEN top=green
(4) IF body<>blue THEN top=red
BANS: body=blue and top<>blue

(1) IF body=blue THEN top=blue
(2) IF body=green THEN body<>red
(3) IF body<>red THEN top=green
(4) IF body<>blue THEN top=red
BANS: body=green and body=red

(1) IF body=blue THEN top=blue
(2) IF body=green THEN body<>red
(3) IF body<>red THEN top=green
(4) IF body<>blue THEN top=red
BANS: body<>red and top<>green

(1) IF body=blue THEN top=blue
(2) IF body=green THEN body<>red
(3) IF body<>red THEN top=green
(4) IF body<>blue THEN top=red
BANS: body<>blue and top<>red

Fig. 3 Steps for achieving the puzzle solution



6 Operations:
Thing::value(propertyP:String):String
For a thing, the value of the property of the thing.
Thing::valid(c:Clue):Boolean
Whether the clue is valid for the thing.
Thing::bannedBy(c:Clue):Boolean
Whether the thing is banned by the clue.
Thing::cluesValid():Set(Thing)
The set of all valid things (independent from the self thing).
Clue::bansOCL():String
The OCL expression retrieving the things that are banned by all clues.
Clue::bansThing():Set(Thing)
The things that are banned by the clue.

10 Invariants (classification: SYNTAX (7), SEMANTICS (2), LOADED INVARIANT (1)):
Clue::premConcEQ_negatedEQ / SYNTAX
If the premise property coincides with the conclusion property, then the premise negated flag coincides with the conclusion negated flag.
Clue::uniqueClue / SYNTAX
Two different clues are syntactically different.
Property::sameNumThing / SYNTAX: PROPERTY-THING
Each property is linked to the same number of things.
Property::uniqueNameValue / SYNTAX
Two different properties have different name-value pairs.
Thing::oneThingWithCluesValidNotBanned / SEMANTICS
There is a unique solution among the set of things.
Thing::sameNumProperty / SYNTAX: THING-PROPERTY
Each thing is linked to the same number of properties.
Thing::solutionRedTopRedBody / LOADED INVARIANT
The solution is red-top and red-bottom.
Thing::thingsWithCluesValid_EQ_thingsNotBanned / SEMANTICS
The things, where all clues are valid, are the things where nothing is banned.
Thing::uniqueProperty / SYNTAX
Two different things are linked to different properties.
Thing::uniquePropertyName / SYNTAX
For each thing, two different properties have different names.

Fig. 4 Class model with attributes and associations as well as informal explanation for operations and invariants

In this case, the single loaded invariant requires that one particular solution among the nine possible solutions is chosen: Top and body of the car must be red. If this invariant is not given, the system will just pick one solution arbitrarily.

3 Automatic generation of puzzle games

Our formal UML and OCL model allows the construction of logical puzzles. This could be done by manually creating the objects explicitly, and the USE tool would check the validity of the designed puzzle. However, as the puzzle becomes larger and more complex, building the puzzle manually could be time-consuming. Moreover, each possible variant of the puzzle must be encoded explicitly. In this section, we describe how we take advantage of the capabilities of an OCL solver, the USE model finder, in order to automatically generate puzzle instances from our puzzle metamodel as given in Fig. 4.

3.1 Focusing on operations, clues and invariants

One central model functionality is realized by the operation `Thing::valid(c:Clue):Boolean`. In the context of a potential puzzle solution, in formal terms a `Thing` object, the operation checks whether a parameter `Clue`, which is basically a primitive formula over a potential solution, is valid in the potential solution or is not valid, where the operation result is determined by the `Boolean` return type. The operation must consider the different syntactical options for building a `Clue`, where the syntactical options are determined by choices between (a) conditional or non-conditional clue, (b) premise negation or premise affirmation and (c) conclusion negation or conclusion affirmation.

In Fig. 5, all six options for building and representing puzzle clues are shown. In the left column, the two options for expressing unconditional clues are pictured, and in the right column, the four options for conditional clues are displayed. In each of the two columns, the clues are first shown in tex-

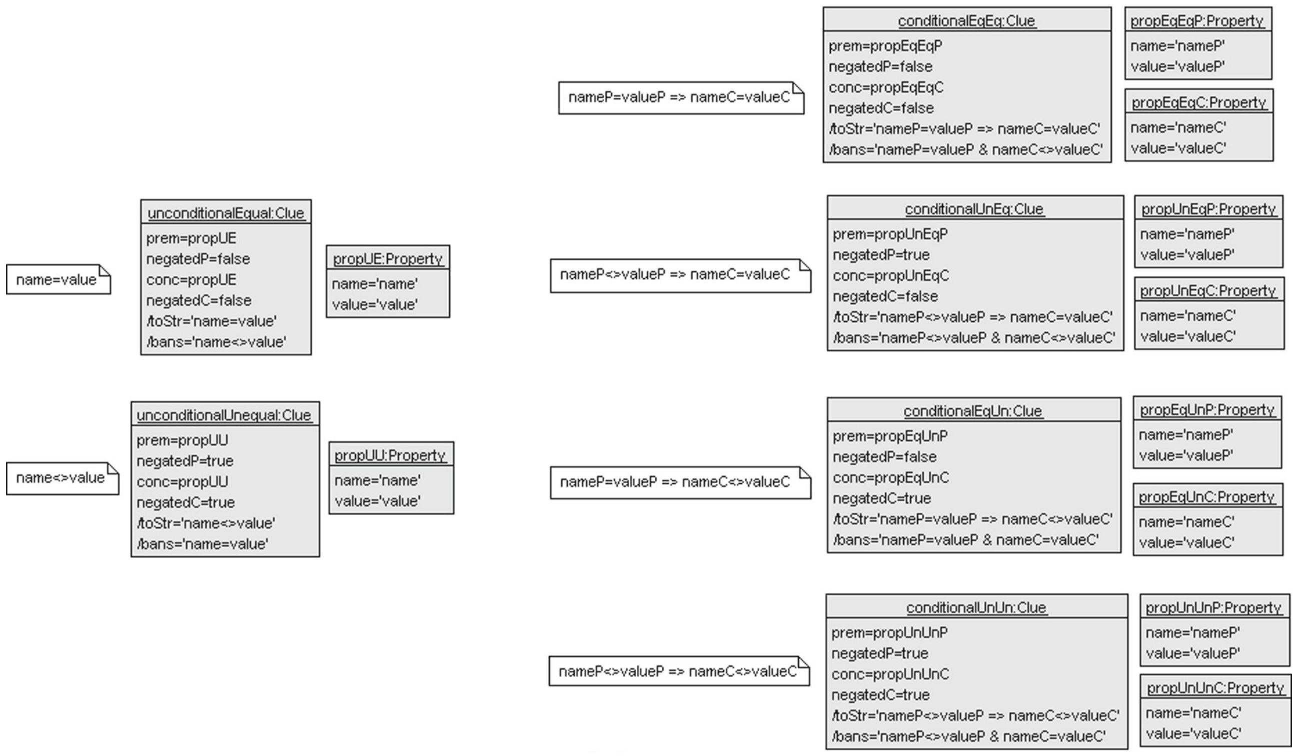


Fig. 5 Six options for building and formally representing puzzle clues

tual form and then in the form of the corresponding object model representation. Basically, the left textual representation coincides with the value of the derived attribute `toStr` in a `Clue` object. The different options in both columns arise due to choices for negation or affirmation of the `Clue` condition with respect to the `Clue` premise and conclusion.

In Fig. 6, the implementation of the operation `Thing::valid(...)` is shown. As indicated within comments, the implementation distinguishes the six cases for clues as presented in Fig. 5. Let us consider in more detail one prototypical example case, a conditional clue with equality in the premise and in the conclusion: `nameP` = `valueP` => *nameC* = *valueC*. This case is implemented lines 10–14. (1) In line 12, the premise property name `c.prem.name` is used to obtain the value of the property in the current thing (`nameP`). Then, it is checked whether the premise property is equal to the premise value obtained with `c.prem.value` (`valueP`). (2) For the conclusion, a similar procedure is followed (line 14). The conclusion property `nameC` and the conclusion value `valueC` are determined (using the expressions `t.value(c.conc.name` and `c.conc.value`, respectively). Then, it is checked whether the conclusion property is in the current `Thing` equal to the conclusion value. (3) Last, the implication expressed in the clue is checked by evaluating the left and right hand side (i.e., `implies` in OCL). The other five cases work in an analogous way with negation or affirmation of clue conditions respected by applying equality or inequality checks at appropriate spots.

As is detailed in Fig. 6, the operation `Thing::cluesValid():Set(Thing)` returns the set of all `Thing` objects in which *all* clues are valid. The operation does not use or rely on a self-object and is thus independent from the `Thing` object on which it is called. It can be applied for characterizing solutions, because all clues are assumed to be valid in a solution. If the puzzle is required to have a unique solution, this unique solution must be included in the operation result because it must be valid for all clues. Two further, central invariants, which apply `cluesValid()` in an essential way, contribute to the core of the model, namely the two invariants that we have called *semantic* invariants before, because they restrict the way attribute values and links have to be *evaluated*. Both invariants use the `Boolean`-valued `Thing` attribute `banned`: The purpose of this attribute is to mark a `Thing` object, i.e., to mark a possible solution, as being banned or, in other words, as being unusable as a solution. (1) The invariant `Thing::thingsWithCluesValid_EQ_thingsNotBanned` asserts that the set of valid `Thing` objects is exactly the set of `Thing` objects that are not banned, i.e., things `t` with `t.banned=false`. (2) The invariant `Thing::oneThingWithCluesValidNotBanned` guarantees the existence of a unique solution by requiring that among all `Thing` objects, in which *all* clues are valid,

```
Thing::valid(c:Clue):Boolean = -- Thing self regards Clue c as valid
  let t:Thing = self in -- Premise property nameP; Conclusion property nameC;
  if c.prem=c.conc and -- non-conditional
     c.negatedC=false then -- name=value - E.g., top=red
     t.value(c.conc.name)=c.conc.value else
  if c.prem=c.conc and -- non-conditional
     c.negatedC=true then -- name<>value - E.g., top<>red
     t.value(c.conc.name)<>c.conc.value else
  if c.prem<>c.conc and -- conditional
     c.negatedP=false and c.negatedC=false then -- nameP=valueP => nameC=valueC
       -- nameP=valueP
       t.value(c.prem.name)=c.prem.value implies -- E.g., top=blue => body=red
       -- nameC=valueC
       t.value(c.conc.name)=c.conc.value else
  if c.prem<>c.conc and -- conditional
     c.negatedP=false and c.negatedC=true then -- nameP=valueP => nameC<>valueC
       t.value(c.prem.name)=c.prem.value implies -- E.g., top=blue => body<>red
       t.value(c.conc.name)<>c.conc.value else
  if c.prem<>c.conc and -- conditional
     c.negatedP=true and c.negatedC=false then -- nameP<>valueP => nameC=valueC
       t.value(c.prem.name)<>c.prem.value implies -- E.g., top<>blue => body=red
       t.value(c.conc.name)=c.conc.value else
  if c.prem<>c.conc and -- conditional
     c.negatedP=true and c.negatedC=true then -- nameP<>valueP => nameC<>valueC
       t.value(c.prem.name)<>c.prem.value implies -- E.g., top<>blue => body<>red
       t.value(c.conc.name)<>c.conc.value
  else false endif endif endif endif endif endif


Thing::cluesValid():Set(Thing) =
  Thing.allInstances->select(t |
    Clue.allInstances->forAll(c | t.valid(c)))


context Thing inv thingsWithCluesValid_EQ_thingsNotBanned:
  cluesValid()=Thing.allInstances->select(t | t.banned=false)


context Thing inv oneThingWithCluesValidNotBanned: -- one solution
  cluesValid()->select(t | t.banned=false)->size=1
```

Fig. 6 Implementation details: operations `Thing::valid(c:Clue):Boolean` and `Thing::cluesValid():Set(Thing)` as well as invariants `Thing::thingsWithCluesValid_EQ_thingsNotBanned` and `Thing::oneThingWithCluesValidNotBanned`

there is exactly *one* `Thing` object that is not banned, namely the unique solution.

3.2 Model configuration

Our puzzle model is a general model to build logical puzzles and we use an OCL solver, the USE model finder, to construct puzzle instances automatically. The solver requires information about which values are permitted in the solution, the number of possible solutions and optional additional constraints about the puzzle not captured by the puzzle by the puzzle model.

In our approach, a puzzle can be configured in a number of ways: One can use (1) the properties file in order to specify the basic underlying puzzle terminology and the number of possible solutions, and (2) additional invariants to restrict possible solutions regarding the puzzle terminology or regarding the form or the extent of puzzle clues.

In the properties file, one can determine for the class `Property` the items that are used for the attributes `name` and `value`. In our running example, we have used `top` and `body` for the attribute `name`, and `red`, `blue` and `green` for the attribute `value`. One can also specify bounds for the number of objects in a class and the number of links in an association. In the example, we have restricted the number of `Thing` objects to `9`, the number of `Property` objects to `6` and the number of `ThingProperty` links to `18`, resulting in `2` links per `Thing` object and `3` links per `Property` object. Additionally, in order to restrict the puzzle options, one can use particular invariants to be loaded that regulate name-value dependencies, similar to introducing value types for properties. For example, one could have an invariant that restricts the `body` color to `black` and `white` and the top color to `gold` and `silver`, if desired.

Puzzle configuration with additional invariants might also handle puzzle features which are independent from the puzzle domain at hand. Invariants can restrict the features of the puzzle solution by considering the form and number of used clues: As clues can be conditional or unconditional, invariants might require a certain number of clues in each category. For example, it is possible to demand that all clues are conditional. We have seen in examples, clues do not necessarily ban things, i.e., a clue might not exclude any possible solution at all. For instance, the clue `body = green => body <> red` in the running example was such a clue. Such clues can make the puzzle purposefully more complicated, if this is desired, and the identification of such clues is part of the teaching concept. Additional invariants might require to exclude or to include such clues. Lastly, our invariant that insists on a unique solution is debatable: Instead of requiring exactly one solution, a replacing invariant could require that two or three solutions are allowed.

3.3 Classifying our model

Our model is a not only a model, but we classify it as a metamodel. The assessment that we are working with a metamodel is already present in Figs. 1 and 2: the UML and OCL class model from the right of Fig. 1 is represented in Fig. 2 as an object model that instantiates the meta class model from Fig. 4. Basically, the object model in Fig. 2 and the class model in Fig. 1 are different views on the same matter, on the same circumstances.

Our model can be regarded as a multi-level model, but it is implemented in a two-level framework (with class and object model) as it is available in USE. Some background on multi-level and two-level approaches can be found in [15]. In this setting, the `Property` class behaves as an *ontological* element (i.e., it describes elements from the application domain), and invariants optionally added by the user allow to restrict allowed values in the application domain for each property. As an example, if one would extend our car example with a property `engine` allowing values like `petrol` and `electric`, one could guarantee by restriction through an OCL invariant, the `top` property takes only values like `red` and the `engine` property values like `electric`, but the `engine` property will never take a value like `green`; an invariant realizes the restriction of values per property type. Please note adding OCL invariants is optional, and many puzzles can be built without the need for OCL. The `Clue` class behaves like a *linguistic* element which allows our system to impose 'universal puzzle constraints' over the application objects. For the distinction between ontological and linguistic elements, see [1].

As an example for the nature of our metamodel, when the class `Clue` is instantiated, one does not obtain directly a clue in the shape of a formula like `forAllc : Carc.body <> (red`. But, in a well-formed puzzle model, one obtains two objects `c:Clue` and `p:Property` with `c.prem=p`, `c.negatedP=true`, `c.conc=p`, `c.negatedC=true`, `p.name='body'`, and `p.value='red'`. The object `p:Property` belongs to the application domain (describes a car), whereas `c:Clue` models actually a constraint over the properties which has been automatically derived according to the puzzle rules encoded in our model invariants and especially in the operation `Thing::valid(Clue)`.

In Fig. 7, you see an overview on our approach showing the central parts and their connections. Basically, there is a class model, namely our metamodel, including metamodel invariants, and an object model that instantiates the metamodel. This object model is generated by the USE model finder based on a puzzle configuration designed by the puzzle developer. Optionally, domain invariants can added. The object model may be viewed also as a class model in which invariants correspond to the clues generated.

4 Building and playing games

The previous sections have described our formal model and how it is integrated with the USE tool to design and build puzzles automatically. However, the puzzles are not directly playable in USE. Therefore, in this section we describe our prototype that allows to build and play puzzle games in practice and the implementation that supports our approach.

4.1 Architecture

Figure 8 shows the architecture of the approach, showcasing the different elements involved. The metamodel that supports our approach and has been presented in Sect. 2 is made avail-

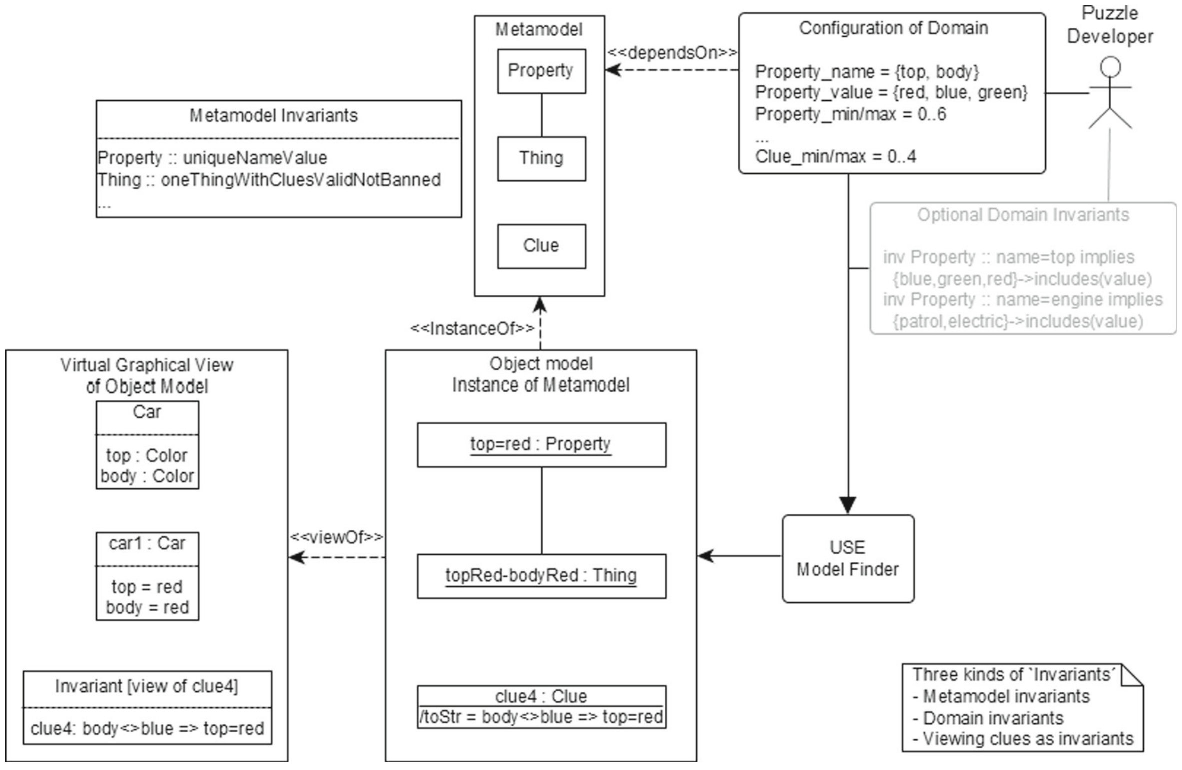

Metamodel
Property
Thing
Clue
Metamodel Invariants
Property :: uniqueNameValue
Thing :: oneThingWithCluesValidNotBanned
...
<<dependsOn>>
Configuration of Domain
Property_name = {top, body}
Property_value = {red, blue, green}
Property_min/max = 0..6
...
Clue_min/max = 0..4
Puzzle Developer
Optional Domain Invariants
inv Property :: name=top implies {blue,green,red}->includes(value)
inv Property :: name=engine implies {patrol,electric}->includes(value)
<<InstanceOf>>
Virtual Graphical View of Object Model
Car
top : Color
body : Color
car1 : Car
top = red
body = red
Invariant [view of clue4]
clue4: body<>blue => top=red
<<viewOf>>
Object model Instance of Metamodel
top=red : Property
topRed-bodyRed : Thing
clue4 : Clue
/toStr = body<>blue => top=red
USE Model Finder
Three kinds of 'Invariants'
- Metamodel invariants
- Domain invariants
- Viewing clues as invariants

Fig. 7 Classifying our approach: Connection metamodel, instantiation, model finder

able to the USE tool and becomes the basis around which the rest of the elements are built (label 1). If a modeler wants to use the metamodel to make experiments or play within the USE tool, he or she must write a properties file and additional constraints (label 2). This can be loaded into the USE model finder to generate puzzle instances and can be inspected and manipulated using the regular USE interface. This is the approach used in the previous sections to illustrate the internals of the approach.

To make the puzzles playable through a user interface that hides the complexity of manipulating USE, we have built two components (label 3). A game generator which takes a property specification and OCL constraints and interacts with the USE model finder to generate the corresponding object diagram. Then, a game engine component interprets the object diagram to make it interactive (e.g., check if a selected 'thing' violates some clue). On top of this, a user interface (label 4) has been built, which provides a way for the user to write the property specification and the constraints, and then a game zone where the puzzle is shown and can be played.

4.2 Specifying puzzles

To specify puzzles, our system requires two main elements. First, the puzzle design has to declare the allowed properties of the puzzle in the form of property names and property

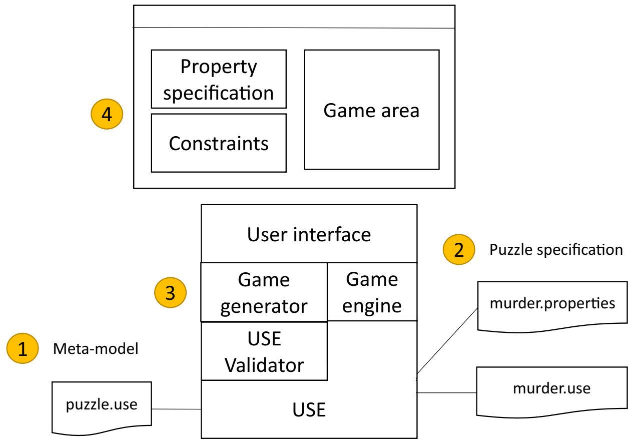


Fig. 8 Architecture of the implementation

values. This is currently done by using property files in the format expected by the USE model finder in order to declare the bounds of the constraint solver. In the example, this will be specified with the following syntax:

Listing 1 Property specification for the running example.

```
Property_name = Set{'top','body'}
Property_value = Set{'red','green',
                  'blue'}
```

In addition, we need to set the bounds for the number of objects of each type in our puzzle model. The computation of these bounds needs to be done currently manually as explained in the previous section.

Listing 2 Model element bounds for the running example.

```
Thing_min = 9
Thing_max = 9

ThingProperty_min = 18
ThingProperty_max = 18

Property_min = 6
Property_max = 6

Clue_min = 4
Clue_max = 4
```

Using this configuration, the USE model finder will construct an instance of the puzzle which will consist of a valid combination of the property values. To enable the construction of more interesting puzzles, we support the specification of additional OCL invariants which guarantee that the puzzle has the required shape (i.e., loaded invariant in Fig. 4). In general, a puzzle may have several solutions. The USE model finder is able to find them (using its scrolling feature). However, a user might be interested on specifying exactly one, target solution. We can use OCL invariants to specify it. For instance, let us assume that we want to make sure that the solution is a car that is fully red.

Listing 3 OCL invariants to restrict the values of the puzzle.

```
context Thing inv solution:
  Thing.allInstances→exists
          (t | t.banned=false
    and t.value('top')='red'
    and t.value('body')='red'
```

4.3 Storing and loading puzzles

A puzzle description is loaded into the USE model finder to generate a concrete puzzle which involves things, clues and a particular solution. This process may take from a few seconds to a few minutes depending on the complexity of puzzle. Therefore, we allow the user to store (and later load) the puzzle so that it can be played repeatedly without the burden of recomputing it.

Our tool generates puzzles using the following YAML format,[1] which is also human readable so that she or he can easily tweak the puzzle (e.g., typically slightly modify the clue description).

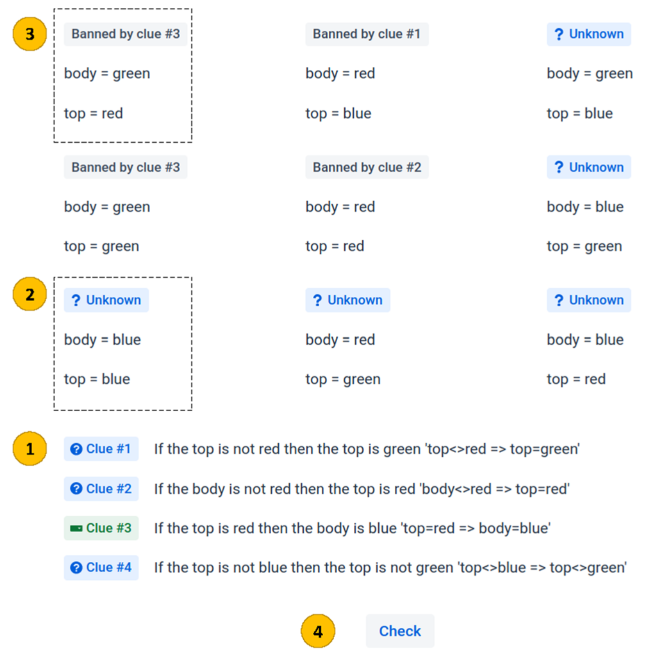
[1] YAML (YAML Ain't Markup Language) is a user-friendly data serialization language. The specification is available at http://yaml.org.

Listing 4 Excerpt of the YAML format.

```
things:
- id: "thing1"
  description:
    elements:
    - body: "green"
    - top: "red"
  banned: true
- id: "thing2"
  description:
    elements:
    - body: "red"
    - top: "blue"
  banned: true
- id: "thing3"
  description:
    elements:
    - body: "red"
    - top: "red"
  banned: false
# ... more things
clues:
- id: "clue1"
  description: |
     If the body is not red
     then the top is red
  bannedThings:
  - "thing3"
  - "thing5"
  - "thing8"
- id: "clue4"
  description: |
     If the top is red
     then the body is blue
  bannedThings:
  - "thing1"
  - "thing6"
# ... more clues
```

4.4 Playing games

To play a game, we have built a user interface (UI) which shows the different elements of the game and allows the user to interactively try to solve the puzzle. Figure 9 shows the user interface for playing the running example puzzle about robbery. The UI shows the list of clues ❶ that the user must take into consideration to figure out the solution. At any time in the game, there is an active clue which is the one that the player is using to reason about which solutions must be discarded according to the clue. The possible solutions are



Banned by clue #3
body = green
top = red

Banned by clue #1
body = red
top = blue

? Unknown
body = green
top = blue

Banned by clue #3
body = green
top = green

Banned by clue #2
body = red
top = red

? Unknown
body = blue
top = green

? Unknown
body = blue
top = blue

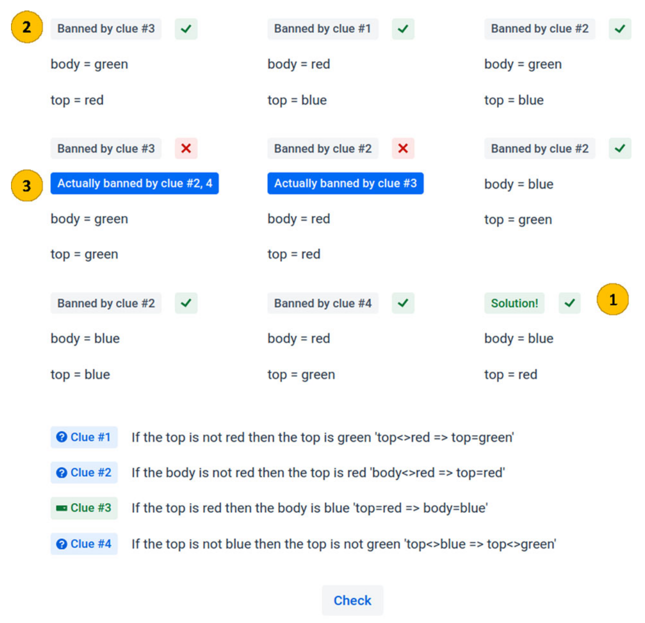
? Unknown
body = red
top = green

? Unknown
body = blue
top = red

Clue #1	If the top is not red then the top is green 'top<>red => top=green'
Clue #2	If the body is not red then the top is red 'body<>red => top=red'
Clue #3	If the top is red then the body is blue 'top=red => body=blue'
Clue #4	If the top is not blue then the top is not green 'top<>blue => top<>green'

Check

Fig. 9 User interface for playing a game



Banned by clue #3 ✓
body = green
top = red

Banned by clue #1 ✓
body = red
top = blue

Banned by clue #2 ✓
body = green
top = blue

Banned by clue #3 ✗
Actually banned by clue #2, 4
body = green
top = green

Banned by clue #2 ✗
Actually banned by clue #3
body = red
top = red

Banned by clue #2 ✓
body = blue
top = green

Banned by clue #2 ✓
body = blue
top = blue

Banned by clue #4 ✓
body = red
top = green

Solution! ✓
body = blue
top = red

Clue #1	If the top is not red then the top is green 'top<>red => top=green'
Clue #2	If the body is not red then the top is red 'body<>red => top=red'
Clue #3	If the top is red then the body is blue 'top=red => body=blue'
Clue #4	If the top is not blue then the top is not green 'top<>blue => top<>green'

Check

Fig. 10 User interface showing the game solution

shown in the form of *cards* (❷ and ❸). A card has a state which can be *Unknown* if it has not been banned as a solution (❷), *Banned* when it has been marked as an invalid solution to the puzzle according to the current clue, or *Solution* which is given to the card that the player thinks is the solution to the puzzle (i.e., typically because it is the only one not banned).

The behavior of the UI follows the resolution strategy described in Sect. 2.2. We consider three modes or difficulty levels for playing the game.

– *Manual solving*. In this mode, the player solves the puzzle completely by herself and only checks the correctness of the solution at the end. Figure 10 shows how the puzzle in Fig. 9 has evolved into the final solution proposed by the player. When the player clicks on 'Check' the system uses the underlying USE model to check whether the things banned by the player are correct or not. In case of a mistake, the thing is marked with a red cross and the correct 'banning clues' are shown, otherwise a green tick is shown. At the same time, if the user has selected the correct solution it is indicated with a green tick.
– *Computer-assisted solving*. In this mode, the user selects a clue, then bans some of the solutions and the system automatically provides corrections. This mode is intended to help the user gain an understanding of how to play.
– *Computer-driven solving*. In this fully automated mode, the system demonstrates, for each clue, which things are banned. This can be useful, for instance, to explore whether a puzzle proposed by a third-party (like the ones in Sect. 5) is correctly encoded or not.

4.5 Implementation

To implement the system, we have integrated the USE model presented in Sect. 3 into a Java web application. The application invokes the USE model finder programmatically to compute the actual puzzle. To facilitate the integration, we use the options offered by EFinder [5] since it provides a simple API for invoking the USE model finder.

When the puzzle instance is computed, our application processes the generated USE objects and present them in the user interface and make the puzzle playable using the UI shown before.

The prototype is available at https://github.com/jesusc/puzzle-builder.

5 Validation

In this section, we aim at demonstrating that our approach is flexible and can be applied for various domains and different kinds of underlying puzzle information and different kinds of clue formulation. To this end, we develop five puzzles of different complexity. In the first two subsections, we develop two puzzles built by ourselves. In the third subsection, we design three more puzzles proposed by an external party.

5.1 A simple, tribute puzzle

We propose a puzzle similar to the running example but focused on researchers. The puzzle goal is stated as follows:

> Your task as PI in a strong modeling group is to find a good candidate to join your team as a postdoc. Three candidates are proposed to you by means of clues about research interests and city of birth. Only one of them is a really bright and young researcher who perfectly fits your modeling team.

To design this puzzle, we have three properties: author, topic and origin (city of birth), and we use as property values the data of Antonio Vallecillo and the authors of this paper. This is specified as follows:

Listing 5 Description of the properties of the puzzle

```
Property_name =
 Set{'Author', 'Topic', 'Origin'}
Property_value = Set{
  'Antonio', 'Jesus', 'Martin',
  'Uncertainty', 'Tooling', 'Validation',
  'Malaga', 'Murcia', 'Dortmund'}
```

The goal is to have *thing* elements like `Martin, Validation, Dortmund` and `Jesus, Tooling, Murcia`, but also `Antonio, Uncertainty, Malaga` as the actual solution for the puzzle.

Using this configuration, the solver may select as a solution an unintended combination of property names and values like `Author = Antonio`, `Topic = Martin`, `Origin = Validation`. Moreover, we are interested in forcing a particular solution, in this case the one involving Antonio. To this end, we must specify additional OCL invariants which guarantee that the puzzle has the required shape. In this case, they will be as follows:

Listing 6 OCL invariants to restrict the values of the puzzle.

```
context Property inv nameFitsValue:
(name='Author' implies
  Set{'Antonio','Martin','Jesus'}
    →includes(value)) and
(name='Topic' implies
  Set{'Uncertainty','Tooling',
             'Validation'}
    →includes(value)) and
(name='Origin' implies
  Set{'Malaga','Murcia','Dortmund'}
    →includes(value))

context Thing inv solutionAntonio:
 Thing.allInstances→exists
            (t | t.banned=false
     and t.value('Author')='Antonio'
     and t.value('Topic')='Uncertainty'
     and t.value('Origin')='Malaga')
```

5.2 Complex puzzle with involved solution derivation process

The context of the puzzle as shown in Fig. 11 comes from a well-known fantasy novel.

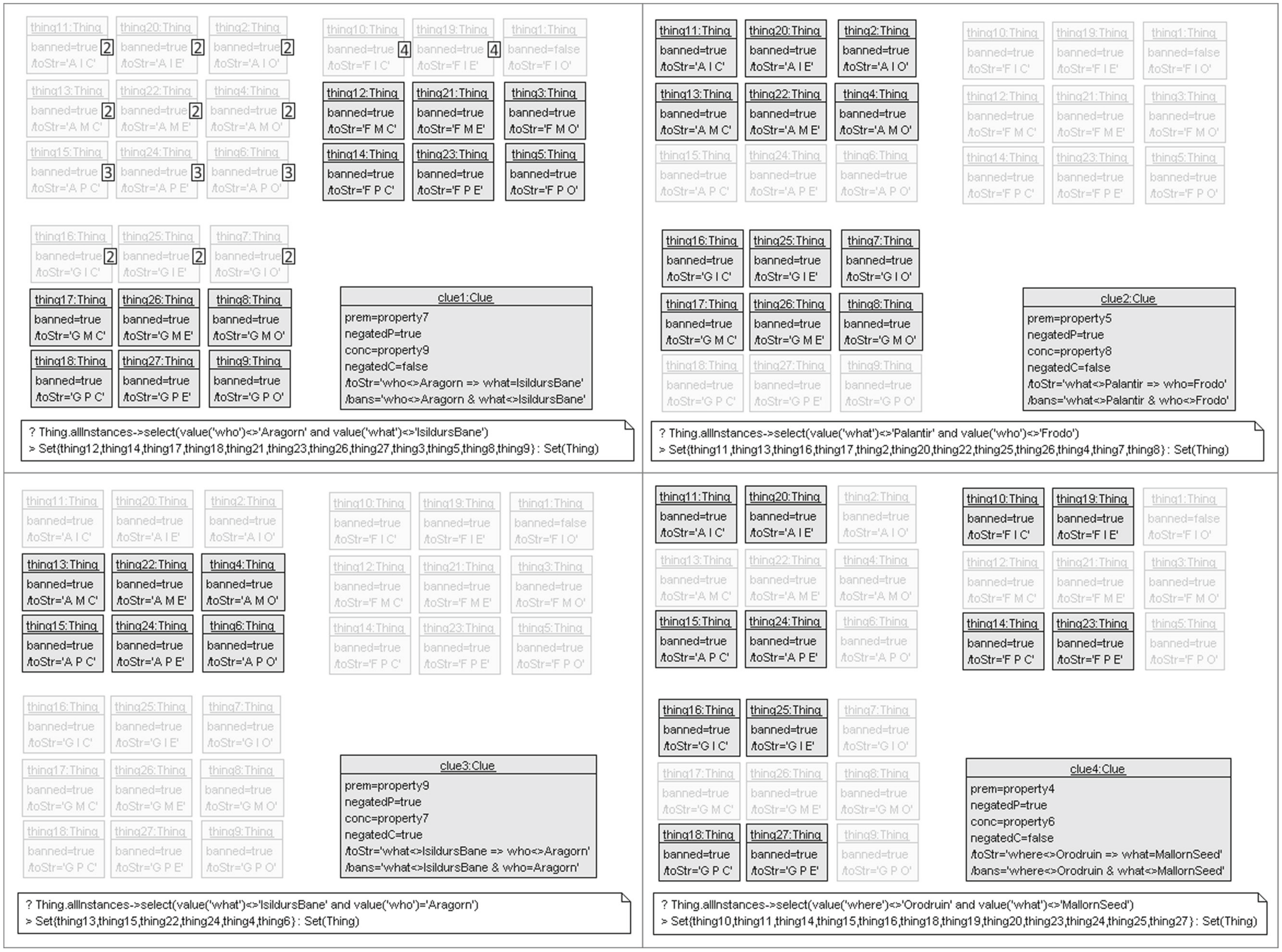
> Three actors are involved in a dangerous journey. The task is to find out *who* of them brings a specific *object* to a specific *place*.

Although one might know the solution (or parts of the solution) by inspecting the offered choices ('who': Aragorn, Frodo, Gandalf; 'what': IsildursBane, MallornSeed, Palantir; 'where': Caradhras, Erebor, Orodruin), the solution, namely the right choice for 'who,' 'where' and 'what,' can be formally derived from the clues in Fig. 11.

In the figure, basically a sequence of USE object diagrams shows (1) the formal representation of puzzle choices with `Property` objects, possible solutions with `Thing` objects, and used clues with `Clue` objects and (2) one derivation of the solution. The upper left quarter shows the application of `clue1`, the upper right of `clue2`, the lower left of `clue3`, and the lower right of `clue4`. In each of the four quarters, the `Thing` objects banned by the respective clue are indicated as dark objects, whereas the light `Thing` objects are not reached by the clue. The respective OCL query and its result are shown as well. In the first quarter for the first clue, the essential results for the last three clues are indicated in short form, as well. Here, dark objects are those banned by `clue1`, the `Thing` objects that have on its right border a rectangle with 2 are those newly banned by `clue2`, `Thing` objects with a 3 rectangle are newly banned by `clue3`, and `Thing` objects with a 4 rectangle are newly banned by `clue4`. The only light `Thing` object being not marked is the only object with `banned=false`, and this object represents the puzzle solution.

The puzzle configuration defines in the properties file the items for the `name` and `value` of `Property` objects as well as appropriate sizes for the number of objects and links for the classes and the association. The configuration includes one invariant that determines the solution:

```
context Thing inv fixSolution
 cluesValid()->exists(t |
   t.value('who')='Frodo' and
   t.value('what')='IsildursBane' and
   t.value('where')='Orodruin')
```



```
clue1:Clue
prem=property7
negatedP=true
conc=property9
negatedC=false
/toStr='who<>Aragorn => what=IsildursBane'
/bans='who<>Aragorn & what<>IsildursBane'
```

```
? Thing.allInstances->select(value('who')<>'Aragorn' and value('what')<>'IsildursBane')
> Set{thing12,thing14,thing17,thing18,thing21,thing23,thing26,thing27,thing3,thing5,thing8,thing9} : Set(Thing)
```

```
clue2:Clue
prem=property5
negatedP=true
conc=property8
negatedC=false
/toStr='what<>Palantir => who=Frodo'
/bans='what<>Palantir & who<>Frodo'
```

```
? Thing.allInstances->select(value('what')<>'Palantir' and value('who')<>'Frodo')
> Set{thing11,thing13,thing16,thing17,thing2,thing20,thing22,thing25,thing26,thing4,thing7,thing8} : Set(Thing)
```

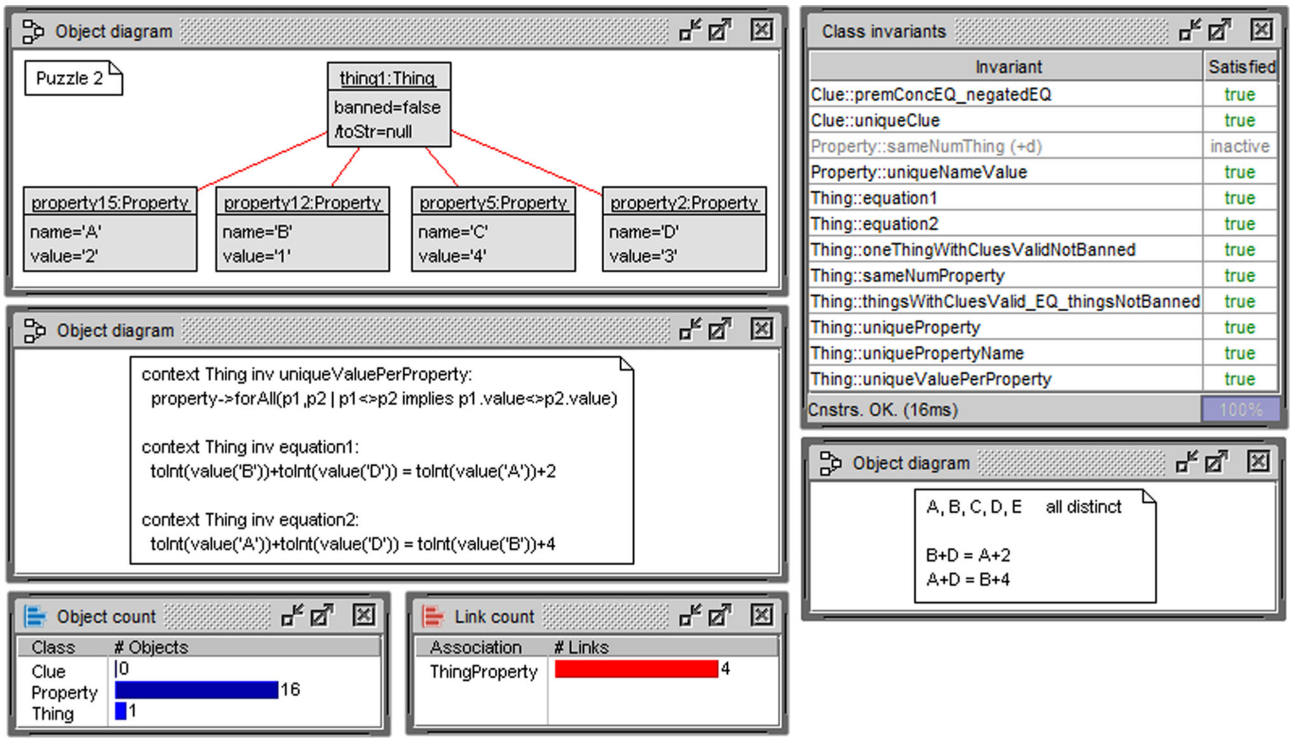
```
clue3:Clue
prem=property9
negatedP=true
conc=property7
negatedC=true
/toStr='what<>IsildursBane => who<>Aragorn'
/bans='what<>IsildursBane & who=Aragorn'
```

```
? Thing.allInstances->select(value('what')<>'IsildursBane' and value('who')='Aragorn')
> Set{thing13,thing15,thing22,thing24,thing4,thing6} : Set(Thing)
```

```
clue4:Clue
prem=property4
negatedP=true
conc=property6
negatedC=false
/toStr='where<>Orodruin => what=MallornSeed'
/bans='where<>Orodruin & what<>MallornSeed'
```

```
? Thing.allInstances->select(value('where')<>'Orodruin' and value('what')<>'MallornSeed')
> Set{thing10,thing11,thing14,thing15,thing16,thing18,thing19,thing20,thing23,thing24,thing25,thing27} : Set(Thing)
```

Fig. 11 Puzzle configuration and solution for fantasy novel



Puzzle 2

thing1:Thing
banned=false
/toStr=null

property15:Property	property12:Property	property5:Property	property2:Property
name='A'	name='B'	name='C'	name='D'
value='2'	value='1'	value='4'	value='3'

Invariant	Satisfied
Clue::premConcEQ_negatedEQ	true
Clue::uniqueClue	true
Property::sameNumThing (+d)	inactive
Property::uniqueNameValue	true
Thing::equation1	true
Thing::equation2	true
Thing::oneThingWithCluesValidNotBanned	true
Thing::sameNumProperty	true
Thing::thingsWithCluesValid_EQ_thingsNotBanned	true
Thing::uniqueProperty	true
Thing::uniquePropertyName	true
Thing::uniqueValuePerProperty	true

Cnstrs. OK. (16ms) 100%

```
context Thing inv uniqueValuePerProperty:
  property->forAll(p1,p2 | p1<>p2 implies p1.value<>p2.value)

context Thing inv equation1:
  toInt(value('B'))+toInt(value('D')) = toInt(value('A'))+2

context Thing inv equation2:
  toInt(value('A'))+toInt(value('D')) = toInt(value('B'))+4
```

```
A, B, C, D, E    all distinct

B+D = A+2
A+D = B+4
```

Class	# Objects
Clue	0
Property	16
Thing	1

Association	# Links
ThingProperty	4

Fig. 12 Puzzle with four integer variables and two equations

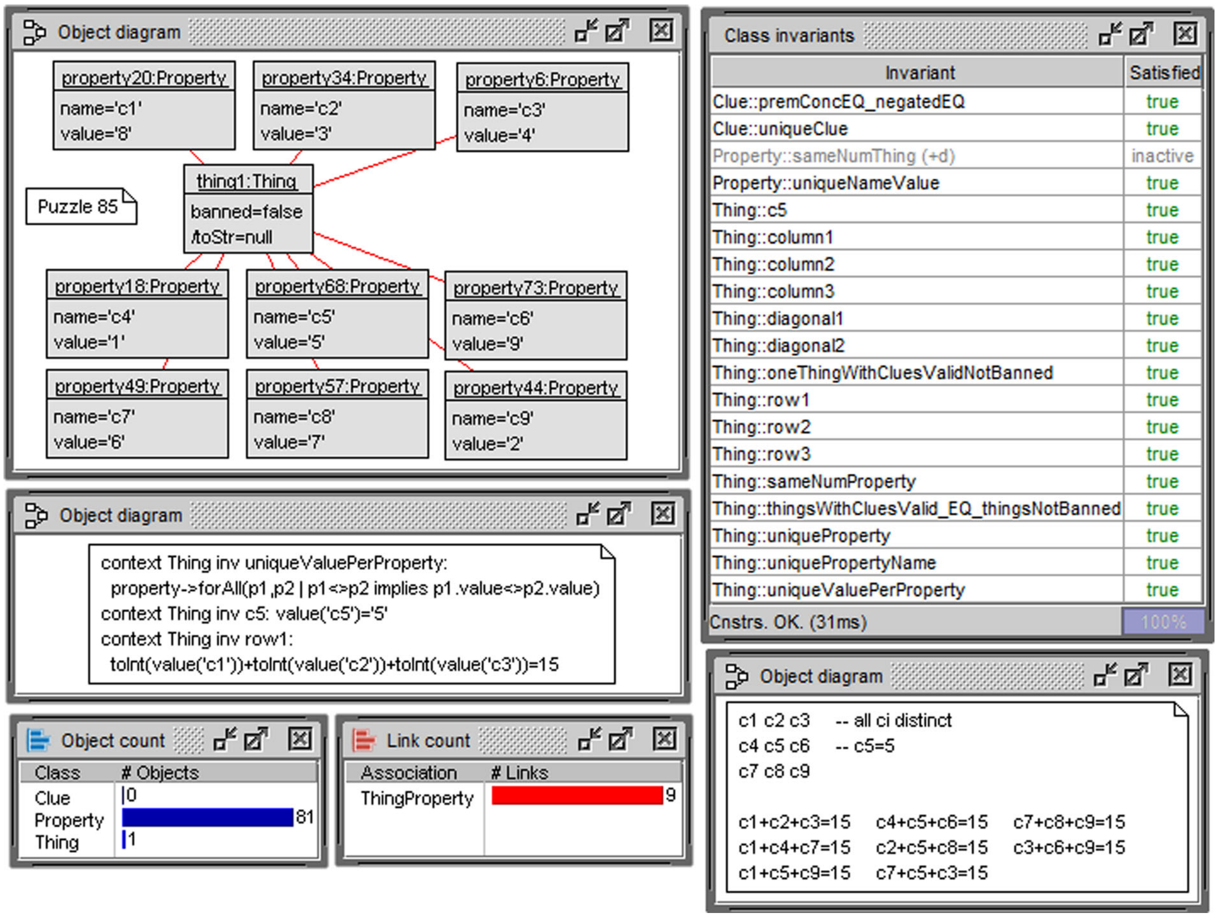
5.3 Puzzles stemming from literature

In the book [10], a large collection of puzzles is modeled in first-order logic. In order to show that our approach for puzzle modeling is capable of expressing puzzles like the ones in [10] and to further validate our approach, we have chosen three typical puzzle examples and explain how they can be formulated in our approach. The examples come from three different book chapters (Micro-arithmetic Puzzles, Love and Marriage, Grid Puzzles). These examples illustrate how analogous puzzles could be effectively modeled using our methodology.

The first puzzle from the book is displayed basically in the form of an object diagram in Fig. 12. It uses four integer variables and requires to find substitutions for them such that particular relationships between the variables values are satisfied. All substituted values have to be distinct, have to come from the interval `1..4`, and the two equations shown in the lower right of Fig. 12 have to hold. In order to handle this puzzle, we had to slightly extend our class model to be able to express numerical conditions, since values are expressed in the original model as strings: we extended the class model with the operation `toInt` that converts a string to an integer, provided the string represents an integer. The four variables are expressed as four properties, the integer interval as attribute restrictions in the properties file, and the distinctness requirement and the two equations are stated as invariants. As can be seen from the found solution, the book puzzle parts could be successfully stated, and the proper solution was found.

In this model and in the following two, there is only one `Thing` object, i.e., only one possible solution, that must be found. Formally, there are no `Clue` objects, since the clues are given in the puzzle formulation and are not expected to be constructed. The already formulated puzzle clues are expressed as invariants that the `Thing` object, which is to be constructed and that possesses attribute values and links, has to satisfy. We emphasize that the puzzle formulations that we have developed for the book puzzles are instantiations of our puzzle metamodel, whereas in the book each puzzle is newly formulated with an ad-hoc model, not as a metamodel instantiation as done in our approach.

The second puzzle from the book is displayed basically in the form of an object diagram in Fig. 13. It shows a `3x3` matrix of Integer variables, requires the value `5` to occur in the center, and each of the rows, columns and diagonals to have the sum `15`. As before, the variables from the matrix are expressed as properties, integer ranges as attribute restrictions in the properties file, and the eight equations for rows, columns and diagonals are stated as invariants, where in Fig. 13 from the eight equations only one equation is shown in an exemplary way. From the found solution, one learns that the book puzzle parts could be properly expressed, and again the right solution was achieved.



Object diagram

property20:Property	property34:Property	property6:Property
name='c1' value='8'	name='c2' value='3'	name='c3' value='4'

Puzzle 85

thing1:Thing
banned=false
/toStr=null

property18:Property	property68:Property	property73:Property
name='c4' value='1'	name='c5' value='5'	name='c6' value='9'

property49:Property	property57:Property	property44:Property
name='c7' value='6'	name='c8' value='7'	name='c9' value='2'

Object diagram

```
context Thing inv uniqueValuePerProperty:
  property->forAll(p1,p2 | p1<>p2 implies p1.value<>p2.value)
context Thing inv c5: value('c5')='5'
context Thing inv row1:
  toInt(value('c1'))+toInt(value('c2'))+toInt(value('c3'))=15
```

Object count

Class	# Objects
Clue	0
Property	81
Thing	1

Link count

Association	# Links
ThingProperty	9

Class invariants

Invariant	Satisfied
Clue::premConcEQ_negatedEQ	true
Clue::uniqueClue	true
Property::sameNumThing (+d)	inactive
Property::uniqueNameValue	true
Thing::c5	true
Thing::column1	true
Thing::column2	true
Thing::column3	true
Thing::diagonal1	true
Thing::diagonal2	true
Thing::oneThingWithCluesValidNotBanned	true
Thing::row1	true
Thing::row2	true
Thing::row3	true
Thing::sameNumProperty	true
Thing::thingsWithCluesValid_EQ_thingsNotBanned	true
Thing::uniqueProperty	true
Thing::uniquePropertyName	true
Thing::uniqueValuePerProperty	true

Cnstrs. OK. (31ms) 100%

Object diagram

```
c1 c2 c3   -- all ci distinct
c4 c5 c6   -- c5=5
c7 c8 c9

c1+c2+c3=15   c4+c5+c6=15   c7+c8+c9=15
c1+c4+c7=15   c2+c5+c8=15   c3+c6+c9=15
c1+c5+c9=15   c7+c5+c3=15
```

Fig. 13 Puzzle with matrix of integer variables and eight equations

The third puzzle from the book is pictured basically in the form of an object diagram in Fig. 14. It is about guessing the name of a particular person at a dinner meeting where hints are given regarding the order that persons sit at a round table. Opposed to the previous two models with focus on numbers, properties express characteristics of persons, namely the gender of the person and the person sitting left to the respective person. In the properties file, the person names and values for gender as well as the number of `Property` objects and links are stated. In order to capture the cyclic nature of the underlying information, i.e., every person has a left neighbor, some `Property` objects show the same domain (namely, the names of the persons) for the `name` and `value` attributes. Invariants formulate the puzzle conditions as (1) proper specification for the gender characteristic, (2) the requirement that women and men alternate, and (3) particular neighbor requirements for two persons (namely `Anna` and `Roger`). The puzzle answer is here formulated as an OCL query which yields a particular result. From the found formulation, we see that the puzzle parts from the book could be expressed, and also the solution from the book was reached.

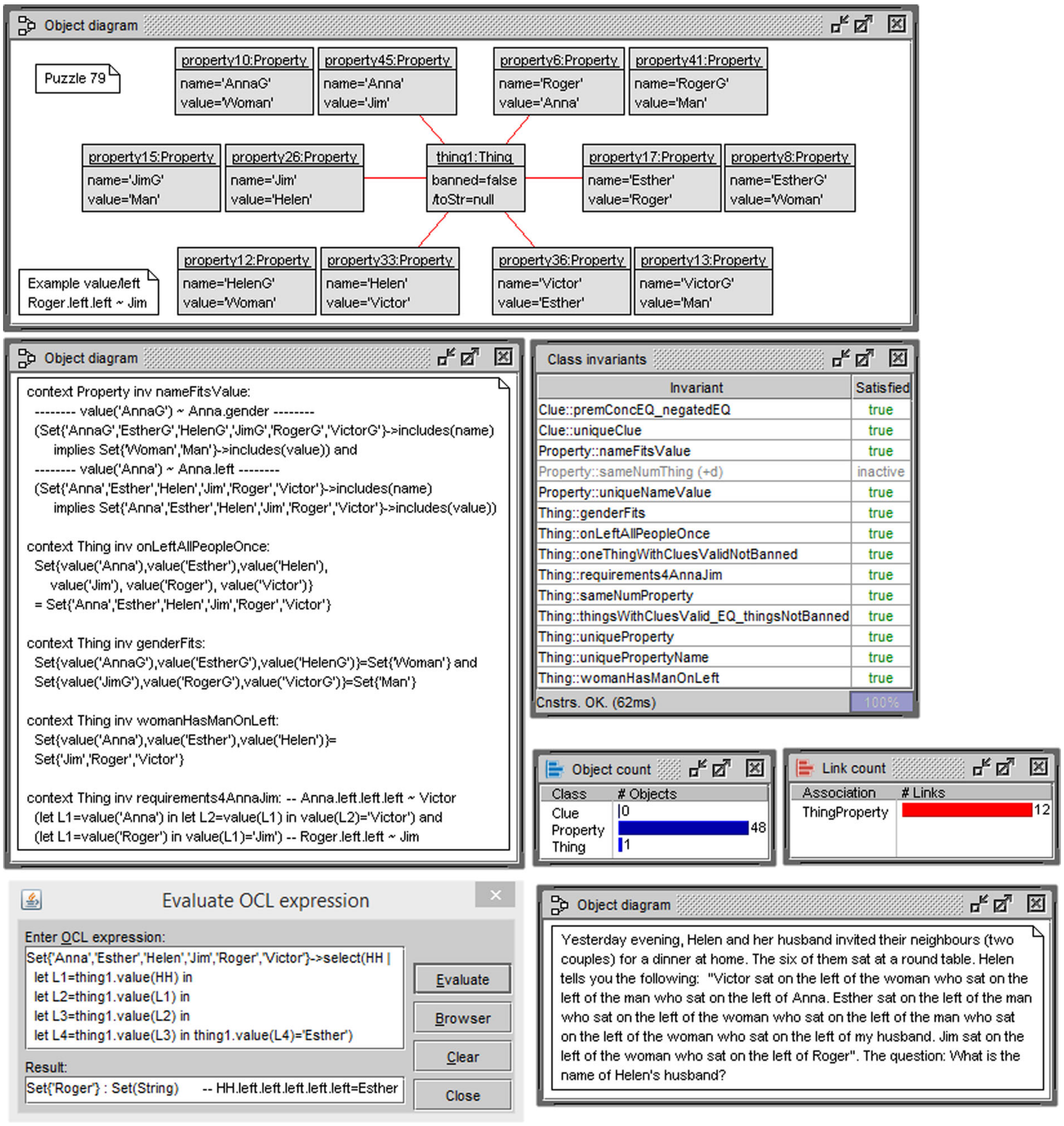
In Fig. 15, we show how the cyclic information structure for the example could be represented in a USE model as an extension of our puzzle metamodel. As the persons are expressed with `Property` objects, one could introduce a derived, reflexive association on the class `Property` for the person sitting left to another person. The object model displays the achieved solution using this additional association.

5.4 Puzzles stemming from the web

In this section, we show that our approach can also be used to model puzzles available in the web and consumed by actual users. In particular, we focus on the type of puzzles of https://www.zebrapuzzles.com (puzzles in the style of the Einstein Riddle [2]).

By using an example, Fig. 16 gives an impression how puzzles of the `zebrapuzzles` website look like and how they would be modeled in our approach. Four parts are

[2] The Einstein's Riddle or Zebra Puzzle is logical puzzle whose creation is attributed to Albert Einstein (https://en.wikipedia.org/wiki/Zebra_Puzzle



Object	name	value
property10:Property	'AnnaG'	'Woman'
property45:Property	'Anna'	'Jim'
property6:Property	'Roger'	'Anna'
property41:Property	'RogerG'	'Man'
property15:Property	'JimG'	'Man'
property26:Property	'Jim'	'Helen'
property17:Property	'Esther'	'Roger'
property8:Property	'EstherG'	'Woman'
property12:Property	'HelenG'	'Woman'
property33:Property	'Helen'	'Victor'
property36:Property	'Victor'	'Esther'
property13:Property	'VictorG'	'Man'

thing1:Thing
banned=false
/toStr=null

```
context Property inv nameFitsValue:
  -------- value('AnnaG') ~ Anna.gender --------
  (Set{'AnnaG','EstherG','HelenG','JimG','RogerG','VictorG'}->includes(name)
      implies Set{'Woman','Man'}->includes(value)) and
  -------- value('Anna') ~ Anna.left --------
  (Set{'Anna','Esther','Helen','Jim','Roger','Victor'}->includes(name)
      implies Set{'Anna','Esther','Helen','Jim','Roger','Victor'}->includes(value))

context Thing inv onLeftAllPeopleOnce:
  Set{value('Anna'),value('Esther'),value('Helen'),
     value('Jim'), value('Roger'), value('Victor')}
  = Set{'Anna','Esther','Helen','Jim','Roger','Victor'}

context Thing inv genderFits:
  Set{value('AnnaG'),value('EstherG'),value('HelenG')}=Set{'Woman'} and
  Set{value('JimG'),value('RogerG'),value('VictorG')}=Set{'Man'}

context Thing inv womanHasManOnLeft:
  Set{value('Anna'),value('Esther'),value('Helen')}=
  Set{'Jim','Roger','Victor'}

context Thing inv requirements4AnnaJim: -- Anna.left.left.left ~ Victor
  (let L1=value('Anna') in let L2=value(L1) in value(L2)='Victor') and
  (let L1=value('Roger') in value(L1)='Jim') -- Roger.left.left ~ Jim
```

Invariant	Satisfied
Clue::premConcEQ_negatedEQ	true
Clue::uniqueClue	true
Property::nameFitsValue	true
Property::sameNumThing (+d)	inactive
Property::uniqueNameValue	true
Thing::genderFits	true
Thing::onLeftAllPeopleOnce	true
Thing::oneThingWithCluesValidNotBanned	true
Thing::requirements4AnnaJim	true
Thing::sameNumProperty	true
Thing::thingsWithCluesValid_EQ_thingsNotBanned	true
Thing::uniqueProperty	true
Thing::uniquePropertyName	true
Thing::womanHasManOnLeft	true

Cnstrs. OK. (62ms) 100%

Class	# Objects
Clue	0
Property	48
Thing	1

Association	# Links
ThingProperty	12

Enter OCL expression:

```
Set{'Anna','Esther','Helen','Jim','Roger','Victor'}->select(HH |
  let L1=thing1.value(HH) in
  let L2=thing1.value(L1) in
  let L3=thing1.value(L2) in
  let L4=thing1.value(L3) in thing1.value(L4)='Esther')
```

Result:

```
Set{'Roger'} : Set(String)     -- HH.left.left.left.left.left=Esther
```

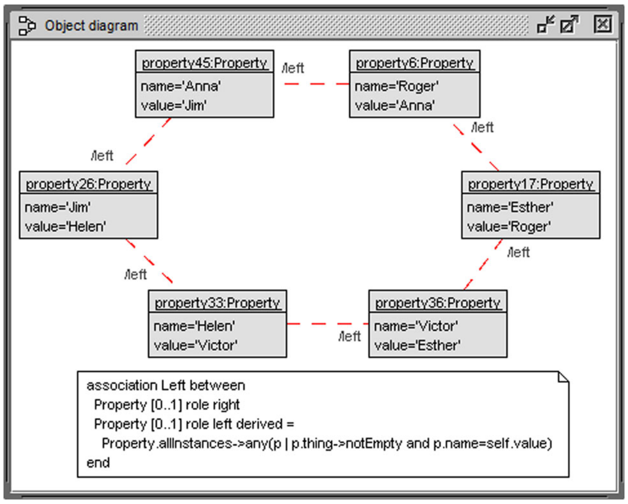
Yesterday evening, Helen and her husband invited their neighbours (two couples) for a dinner at home. The six of them sat at a round table. Helen tells you the following: "Victor sat on the left of the woman who sat on the left of the man who sat on the left of Anna. Esther sat on the left of the man who sat on the left of the woman who sat on the left of the man who sat on the left of the woman who sat on the left of my husband. Jim sat on the left of the woman who sat on the left of Roger". The question: What is the name of Helen's husband?

Fig. 14 Puzzle with three couples sitting at round table

shown: The website puzzle structure in the form of a table to be filled by the user, the representation of the solution table as a UML object diagram in our model, the verbal clues given to the user and the representation of the clues in the form of a complex OCL invariant with each of the 13 clues formalized as a conjunctional part in an existentially (`one`) quantified OCL formula.

The puzzle involves four different men characterized by four properties: a *shirt* with a unique color, a unique first *name* and a food order for a unique *soup* and a unique *meat* dish. The men are arranged from left to right, and, accordingly, one can think of the properties being arranged in line, e.g., there is the soup in the third position. The property values are available through drop-down menus. The table already shows the puzzle solution except the second soup. However, this soup is already determined as the other three soups have already been selected. This second soup is represented in the object diagram in the second part of the figure through the property object `property37` with `name` value `Soup2`. The object diagram layout follows the table.

The UML object diagram shows how the puzzle solution is represented as a `Thing` object with links to sixteen property



```
association Left between
  Property [0..1] role right
  Property [0..1] role left derived =
    Property.allInstances->any(p | p.thing->notEmpty and p.name=self.value)
end
```

Fig. 15 Domain-specific model extension for capturing example characteristics

objects. Thus, for example, there are four *Shirt* `Property` objects, namely for *Shirt1*, *Shirt2*, *Shirt3* and *Shirt4* holding appropriate values. In our model, there are 64 `Property` objects, as there are 16 properties (Shirt 1..4, Name 1..4, Soup 1..4, Meat 1..4) each capable of holding 4 values: $16 * 4 = 64$. In our first puzzle example for a car with three different top and body colors, we have explicitly shown all possible nine solutions in Fig. 1. In this example, we would roughly obtain $4! * 4! * 4! * 4! = 331.776$ different solutions, a number unmanageable to display and to construct in our approach.

The third part of Fig. 16 displays the verbal clues presented to the user. The clues directly constrain (1) single fields in the table (e.g., using formulations as 'in the last position'), (2) lines by particular restrictions (e.g., 'at one of the ends'), (3) the table by requiring particular patterns to occur (e.g., 'immediately after') or (4) neighbor or successor properties and values (e.g., 'somewhere after'). These verbal formulations are made precise by stating them in OCL.

The fourth part shows the clues as parts of a complex OCL invariant. The clues partly directly refer to single properties, e.g., 'The man who ordered beef is in the last position' roughly corresponds to 'Meat4=beef.' The clues partly refer to property combinations, e.g., 'The person who ordered corn soup is immediately after Charles' roughly corresponds to '(Name1 = Charles and Soup2 = CornS) or (Name2 = Charles and Soup3 = CornS) or (Name3 = Charles and Soup4 = CornS).' We have marked three of the 13 clues in the verbal part and correspondingly marked them in the formal representation as an OCL invariant for easy identification. The OCL formulation nicely points out that there are six used clue patterns. Note that some of the clues cannot be expressed in our clue metamodel as there we only consider simple implications, but not general first-order formulas (e.g., a series of disjunctions of conjunctions as in the above 'Charles' and 'CornS' example).

In conclusion, we can say that we are able to model puzzles like the ones from `zebrapuzzles` and express the clues as constraints. Proceeding this way, we can check whether among a number of existing `Thing` objects a solution exists, but we currently cannot generate clues like the ones occurring in the example as this would require an extension of our current (clue) metamodel. We emphasize, however, that the basic logic behind `zebrapuzzles` can be formulated in our approach, but some approach limits become apparent. However on `zebrapuzzles`, manually constructed puzzles and solutions are presented whereas we automatically can construct puzzles, solutions and clues all restricted with constraints (e.g., single solution or multiple solution).

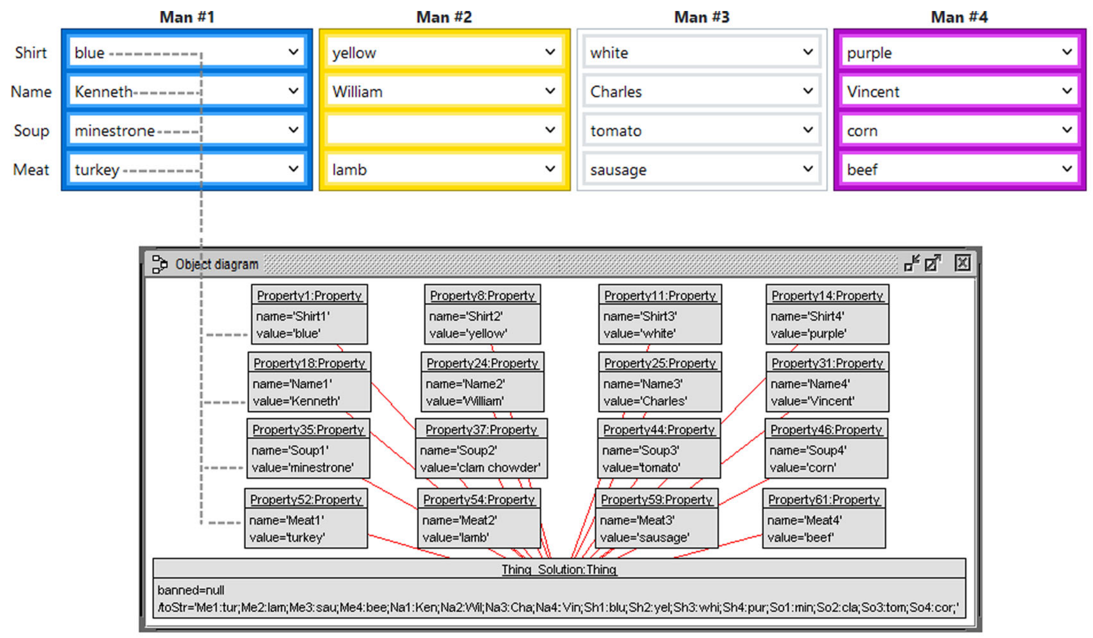
5.5 Student survey on logical puzzles

In order to obtain feedback on our puzzle approach, we performed a survey among students participating in a UML and OCL course. The survey was performed after the end of the course and before the oral examinations. The survey with nine questions presented two puzzles and was asking about (1) the solutions and derivations of solutions [4 questions for puzzle A and 2 questions for puzzle B], (2) the applied logical laws in the solution finding process [1 question] and (3) the difficulty and the usefulness of logical puzzles [2 questions].

Both puzzles were about a scenario where a murder crime had to be solved. The murderer was either the Butler, the Duke or the Countess. The murder happened either in the Library, the Parlour or the Garden. The murder either used a Rope, a Gun or a Knife. Two different sets of clues both possessing a unique solution and presented in the form of object diagrams with `Clue` objects from our metamodel were given. Both puzzles are displayed in Fig. 17 with the derived attribute `/toStr` showing the clue in compact form. In the figure, the first puzzle also presents on the right side three derivations for a potential solution that is shown in the bottom line. Two of the three derivations are erroneous and make unacceptable conclusions. The second derivation is the correct one. Results and intermediate steps of the derivations are pictured in comment notes on the right side.

All questions were presented along with multiple-choice questions and answers. The questions asked were:

1. Which solution is correct for puzzle A?
2. Which is the first incorrect step in derivation 1?
3. Which is the first incorrect step in derivation 2?
4. Which is the first incorrect step in derivation 3?
5. Which clue order is good for solving puzzle B?
6. What is the solution for the puzzle B?
7. Which logic law must be applied for puzzle solving?
8. Rank the difficulty of the logical puzzles.



	Man #1	Man #2	Man #3	Man #4
Shirt	blue	yellow	white	purple
Name	Kenneth	William	Charles	Vincent
Soup	minestrone		tomato	corn
Meat	turkey	lamb	sausage	beef

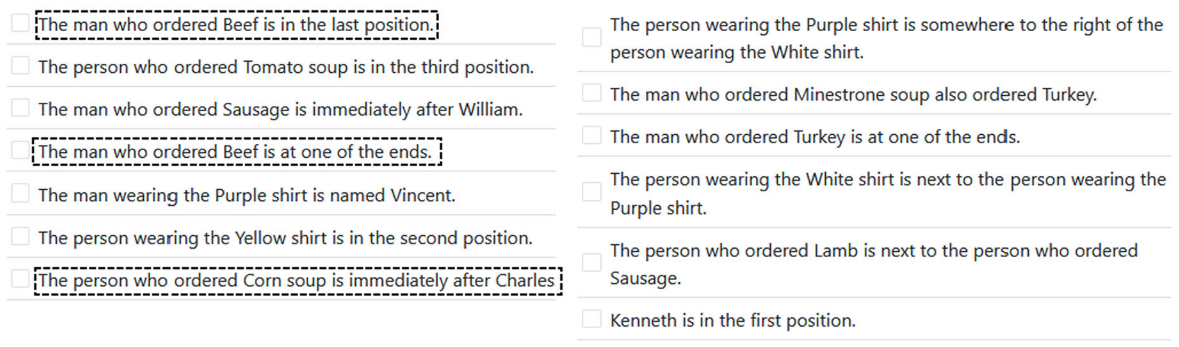
Property1:Property
name='Shirt1'
value='blue'
	Property8:Property
	name='Shirt2'
	value='yellow'
		Property11:Property
		name='Shirt3'
		value='white'
			Property14:Property
			name='Shirt4'
			value='purple'

Property18:Property
name='Name1'
value='Kenneth'
	Property24:Property
	name='Name2'
	value='William'
		Property25:Property
		name='Name3'
		value='Charles'
			Property31:Property
			name='Name4'
			value='Vincent'

Property35:Property
name='Soup1'
value='minestrone'
	Property37:Property
	name='Soup2'
	value='clam chowder'
		Property44:Property
		name='Soup3'
		value='tomato'
			Property46:Property
			name='Soup4'
			value='corn'

Property52:Property
name='Meat1'
value='turkey'
	Property54:Property
	name='Meat2'
	value='lamb'
		Property59:Property
		name='Meat3'
		value='sausage'
			Property61:Property
			name='Meat4'
			value='beef'

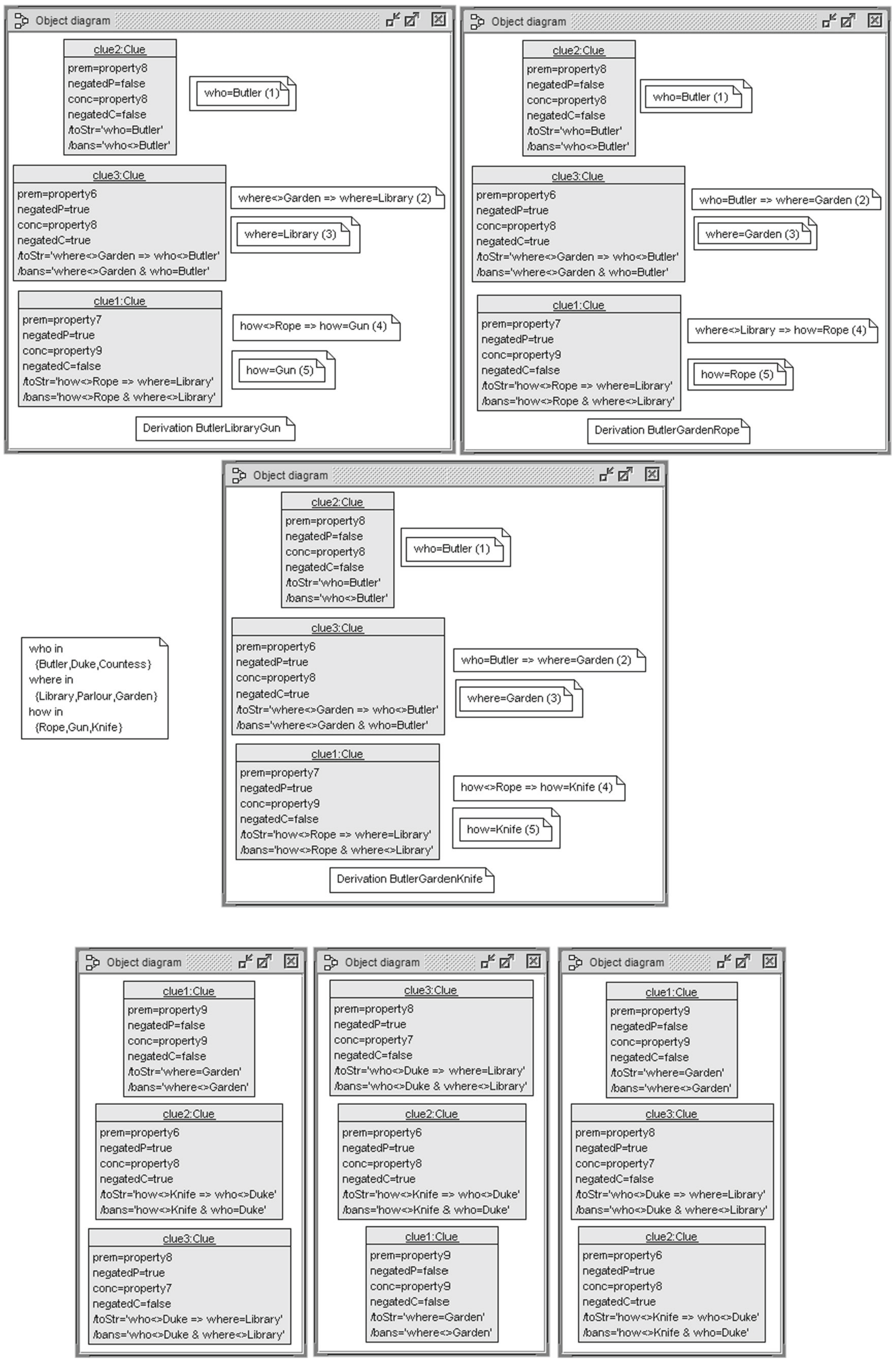
Thing_Solution:Thing
banned=null
/toStr='Me1:tur;Me2:lam;Me3:sau;Me4:bee;Na1:Ken;Na2:Wil;Na3:Cha;Na4:Vin;Sh1:blu;Sh2:yel;Sh3:whi;Sh4:pur;So1:min;So2:cla;So3:tom;So4:cor;'



- ☐ The man who ordered Beef is in the last position.
- ☐ The person who ordered Tomato soup is in the third position.
- ☐ The man who ordered Sausage is immediately after William.
- ☐ The man who ordered Beef is at one of the ends.
- ☐ The man wearing the Purple shirt is named Vincent.
- ☐ The person wearing the Yellow shirt is in the second position.
- ☐ The person who ordered Corn soup is immediately after Charles
- ☐ The person wearing the Purple shirt is somewhere to the right of the person wearing the White shirt.
- ☐ The man who ordered Minestrone soup also ordered Turkey.
- ☐ The man who ordered Turkey is at one of the ends.
- ☐ The person wearing the White shirt is next to the person wearing the Purple shirt.
- ☐ The person who ordered Lamb is next to the person who ordered Sausage.
- ☐ Kenneth is in the first position.

```
Thing.allInstances->one(t|
( t.value('Meat4')='beef' ) and
( t.value('Soup3')='tomato' ) and
( (t.value('Name1')='William' and t.value('Meat2')='sausage') or
  (t.value('Name2')='William' and t.value('Meat3')='sausage') or
  (t.value('Name3')='William' and t.value('Meat4')='sausage') ) and
( t.value('Meat1')='beef' or t.value('Meat4')='beef' ) and
( (t.value('Shirt1')='purple' and t.value('Name1')='Vincent') or
  (t.value('Shirt2')='purple' and t.value('Name2')='Vincent') or
  (t.value('Shirt3')='purple' and t.value('Name3')='Vincent') or
  (t.value('Shirt4')='purple' and t.value('Name4')='Vincent') ) and
( t.value('Shirt2')='yellow' ) and
( (t.value('Name1')='Charles' and t.value('Soup2')='corn') or
  (t.value('Name2')='Charles' and t.value('Soup3')='corn') or
  (t.value('Name3')='Charles' and t.value('Soup4')='corn') ) and
( (t.value('Shirt1')='white' implies Set{t.value('Shirt2'),t.value('Shirt3'),t.value('Shirt4')}->
                                     includes('purple')) and
  (t.value('Shirt2')='white' implies Set{t.value('Shirt3'),t.value('Shirt4')}->includes('purple')) and
   t.value('Shirt3')='white' implies t.value('Shirt4')='purple' ) and
( (t.value('Soup1')='minestrone' and t.value('Meat1')='turkey') or
  (t.value('Soup2')='minestrone' and t.value('Meat2')='turkey') or
  (t.value('Soup3')='minestrone' and t.value('Meat3')='turkey') or
  (t.value('Soup4')='minestrone' and t.value('Meat4')='turkey') ) and
( t.value('Meat1')='turkey' or t.value('Meat4')='turkey' ) and
( (Set{t.value('Shirt1'),t.value('Shirt2')}=Set{'white','purple'}) or
  (Set{t.value('Shirt2'),t.value('Shirt3')}=Set{'white','purple'}) or
  (Set{t.value('Shirt3'),t.value('Shirt4')}=Set{'white','purple'}) ) and
( (Set{t.value('Meat1'),t.value('Meat2')}=Set{'lamb','sausage'}) or
  (Set{t.value('Meat2'),t.value('Meat3')}=Set{'lamb','sausage'}) or
  (Set{t.value('Meat3'),t.value('Meat4')}=Set{'lamb','sausage'}) ) and
( t.value('Name1')='Kenneth' ) )
```

Fig. 16 Zebra Puzzle structure and clues on UI and in our internal metamodel representation



```
Object diagram

clue2:Clue
prem=property8
negatedP=false
conc=property8
negatedC=false
/toStr='who=Butler'
/bans='who<>Butler'

who=Butler (1)

clue3:Clue
prem=property6
negatedP=true
conc=property8
negatedC=true
/toStr='where<>Garden => who<>Butler'
/bans='where<>Garden & who=Butler'

where<>Garden => where=Library (2)
where=Library (3)

clue1:Clue
prem=property7
negatedP=true
conc=property9
negatedC=false
/toStr='how<>Rope => where=Library'
/bans='how<>Rope & where<>Library'

how<>Rope => how=Gun (4)
how=Gun (5)

Derivation ButlerLibraryGun
```

```
Object diagram

clue2:Clue
prem=property8
negatedP=false
conc=property8
negatedC=false
/toStr='who=Butler'
/bans='who<>Butler'

who=Butler (1)

clue3:Clue
prem=property6
negatedP=true
conc=property8
negatedC=true
/toStr='where<>Garden => who<>Butler'
/bans='where<>Garden & who=Butler'

who=Butler => where=Garden (2)
where=Garden (3)

clue1:Clue
prem=property7
negatedP=true
conc=property9
negatedC=false
/toStr='how<>Rope => where=Library'
/bans='how<>Rope & where<>Library'

where<>Library => how=Rope (4)
how=Rope (5)

Derivation ButlerGardenRope
```

```
who in
  {Butler,Duke,Countess}
where in
  {Library,Parlour,Garden}
how in
  {Rope,Gun,Knife}
```

```
Object diagram

clue2:Clue
prem=property8
negatedP=false
conc=property8
negatedC=false
/toStr='who=Butler'
/bans='who<>Butler'

who=Butler (1)

clue3:Clue
prem=property6
negatedP=true
conc=property8
negatedC=true
/toStr='where<>Garden => who<>Butler'
/bans='where<>Garden & who=Butler'

who=Butler => where=Garden (2)
where=Garden (3)

clue1:Clue
prem=property7
negatedP=true
conc=property9
negatedC=false
/toStr='how<>Rope => where=Library'
/bans='how<>Rope & where<>Library'

how<>Rope => how=Knife (4)
how=Knife (5)

Derivation ButlerGardenKnife
```

```
Object diagram

clue1:Clue
prem=property9
negatedP=false
conc=property9
negatedC=false
/toStr='where=Garden'
/bans='where<>Garden'

clue2:Clue
prem=property6
negatedP=true
conc=property8
negatedC=true
/toStr='how<>Knife => who<>Duke'
/bans='how<>Knife & who=Duke'

clue3:Clue
prem=property8
negatedP=true
conc=property7
negatedC=false
/toStr='who<>Duke => where=Library'
/bans='who<>Duke & where<>Library'
```

```
Object diagram

clue3:Clue
prem=property8
negatedP=true
conc=property7
negatedC=false
/toStr='who<>Duke => where=Library'
/bans='who<>Duke & where<>Library'

clue2:Clue
prem=property6
negatedP=true
conc=property8
negatedC=true
/toStr='how<>Knife => who<>Duke'
/bans='how<>Knife & who=Duke'

clue1:Clue
prem=property9
negatedP=false
conc=property9
negatedC=false
/toStr='where=Garden'
/bans='where<>Garden'
```

```
Object diagram

clue1:Clue
prem=property9
negatedP=false
conc=property9
negatedC=false
/toStr='where=Garden'
/bans='where<>Garden'

clue3:Clue
prem=property8
negatedP=true
conc=property7
negatedC=false
/toStr='who<>Duke => where=Library'
/bans='who<>Duke & where<>Library'

clue2:Clue
prem=property6
negatedP=true
conc=property8
negatedC=true
/toStr='how<>Knife => who<>Duke'
/bans='how<>Knife & who=Duke'
```

Fig. 17 Two clue sets both with a unique solution for a murder puzzle: Puzzle A and Puzzle B

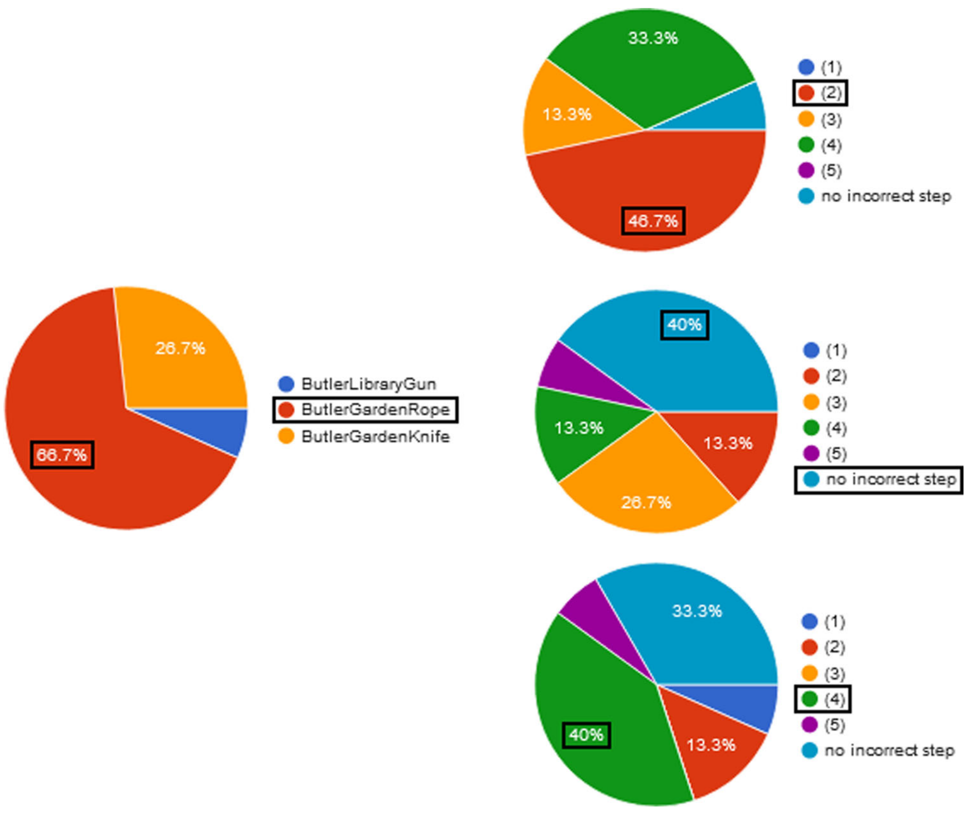


Fig. 18 Overview student responses to questions 1, 2, 3, 4 with correct answers marked

9. Rank the usefulness of logical puzzles for teaching.

For questions 2–4, the multiple-choice answers included an option 'no incorrect step.' For question 7, a choice between three important logical laws was offered. Although the survey is based on multiple-choice questions, we believe that the questions are formulated on a high technical level, and it seems unlikely that by chance only correct answers are picked.

The student responses are shown in overview form in Figs. 18, 19 and 20. For the first seven questions, there is one correct answer, the other answers are wrong. Basically, and as expected, there is a relative majority in the student answers for the right answers. However, there is one exception: in question (7) about which logical law has to be applied in the puzzle solution process there is a majority for the De Morgan law, not for the law stating implication equivalence. Of course, this question is not directly about a concrete puzzle, but it is about abstract principles standing behind formal reasoning. As we know, a large number of students in media informatics being not trained in formal matters attended the course, we suspect that this is the reason for this fact. Otherwise the student answers are basically as expected and do not present serious surprises. The two ranking questions reveal that the students found the puzzles relatively hard to solve. And there is a majority in regarding such logical puzzles as important or at least as useful for Computer Science educa-

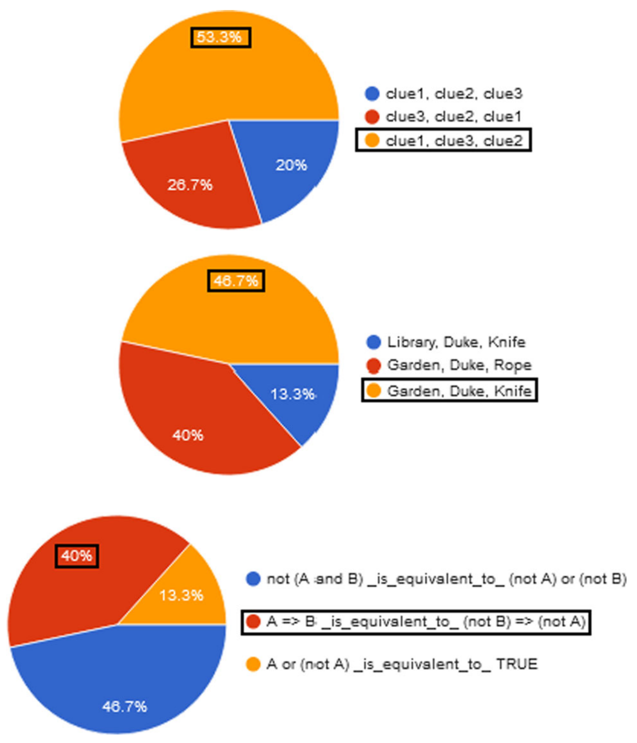


Fig. 19 Overview student responses to questions 5, 6, 7 with correct answers marked

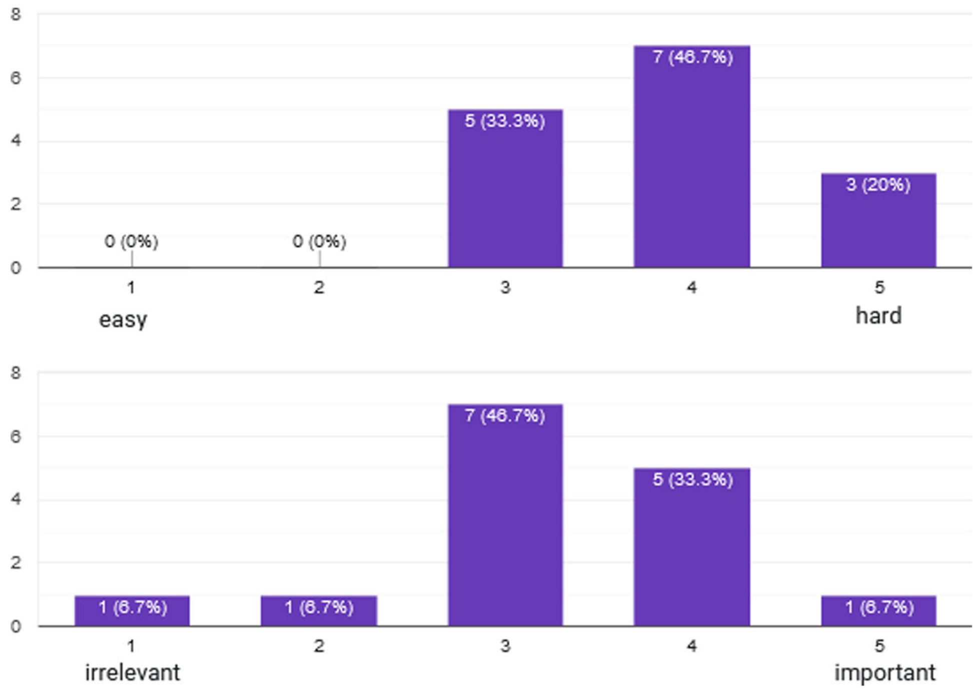


Fig. 20 Overview student responses to questions 8, 9

tion. In the respective course, which covers a wide range of topics from basic UML to operation details of OCL, we did not put a high emphasis on using puzzles. Probably a better approval of puzzles would result, if teaching with puzzles would be better emphasized.

Survey summary: Fifteen students participated in the survey. For the single questions and stated percentages, one can derive the concrete numbers. The educational context for the survey was that the survey took place on a voluntary basis for the students after the lectures and before the oral examination. Twenty-five students were enrolled in the course, which means that ten enrolled students did not participate in the survey. The participating ones probably expected further hints for the oral examination that was already scheduled.

5.6 Closing remarks on validation

The examples from this section on validation show that the approach is capable of formulating a large spectrum of puzzle situations, from numerical problems to discrete graph-like structures with cyclic information reference. In particular, nontrivial puzzles that come from the literature or from the web could be realized or at least be modeled with the approach.

The student survey indicates that students with background on UML/OCL are, in general, able to interpret our puzzle models. Moreover, the survey indicates that teaching with puzzles is accepted well by students.

6 Related work

In this section, we present related work. It is organized in four categories, namely, works about puzzles in education (Sect. 2.1), works about automatically solving puzzles (Sect. 2.2), approaches to (automatically) develop puzzles (Sect. 2.3) and finally we describe how logical puzzles are used as part of businesses (Sect. 2.4).

6.1 Puzzles in education

The use of puzzles is a well-known approach to the teaching of problem-solving skills and athematical thinking [17], [7]. The nature and difficulty of the puzzles vary depending on the target players: from school grade students to engineering students.

For instance, Parsons puzzles are targeted to computer science students since the task is to reorder a set of statements to form a target programming assignment. For this kind of puzzles, both solvers [14] and builder [6] tools have been proposed.

Puzzles can also be used to introduce complex topics to students. For instance, quantum gates are introduced to middle-school and high-school students through a dedicated puzzle game [16].

Puzzles have been used to teach first-order logic. For instance, the 'Introduction to logic' course at Stanford uses a variant of the MineField game .[3] In [11], a variant of this

[3] http://logic.stanford.edu/intrologic/extras/minefield.html.

approach in which the system also generates new instances of the game automatically is put forward.

In our case, our goal is to automatically build logic-based puzzles which could be played by students (e.g., high-school students). These puzzles can also be targeted to children if the complexity of the solution and clues is small and the appropriate gaming experience (e.g., physical cards) is provided.

6.2 Solving puzzles

The topic of solving puzzles has traditionally been addressed by using solvers of different types (e.g., SAT or SMT solvers). In this case, given a puzzle the complexity lies on properly encoding the puzzle using the underlying formalism. For instance, in [10] several puzzles along with their encoding in a logic formalism are discussed. In this work, we use some of these puzzles to show the applicability of our approach.

The topic of solving puzzles has recently gained interest due to the new AI models, since puzzles along with solutions can be used to test the reasoning capabilities of large language models (LLMs). Also, LLMs can be used as an interface to interact with the puzzle [12]. In our case, our system could be used as a test generator (by generating different puzzle instances) to analyze the capabilities of LLMs for solving logic puzzles.

6.3 Developing puzzles

The design and development of new puzzles is a concern complementary to solving them. In this respect, some DSLs have been proposed to specify puzzles and their gaming context. For instance, EGGG [19] is DSL which allows the user to encode games, and it generates an interface to play with them (e.g., the rock–paper–scissors). However, the system is primarily a code generator, and it does not really 'understand' the games.

In [3], a DSL is proposed to encode human strategies for solving logical puzzles like Sudoku or Nonograms. The goal in this case is to understand how players address complex games. This is done through program synthesis techniques. In our case, our generation of `Clue` objects can also be seen as program synthesis procedure.

Ortiz et al. [18] proposed a system to generate Nonograms puzzles based on color images.

To the best of our knowledge, our approach is the first approach based on software modeling foundations to build logical puzzles, which incorporates the automatic generation of puzzle instances via constraint solving.

6.4 The industry of logical puzzles

Although logical puzzles have a tradition dating back many centuries [20] and many well-known logical puzzles have been created (e.g., the *zebra puzzle* also known as Einstein's Riddle), there is still an active industry which produces new puzzles and publishes them in magazines, books and dedicated websites.

As a concrete example, the website https://www.brainzilla.com/ offers a large number of puzzles of different kinds. Several of them can be classified within the category of *logical puzzles* since they are built on top of clues and logical relationships. It has the so-called 'zebra puzzles' (the type of puzzles shown in Sect. 5.4), logic grid puzzles, Suduku-like puzzles and logical equations.

Our approach is particularly relevant for logic grid puzzles which we can model and generate automatically with our tool. According to personal communication with the owner of the website, the logic grid puzzles are constructed manually by skilled game player. This means that our approach could be an alternative to build this type of puzzles. On the other hand, we have shown that our approach is expressive enough to model Zebra Puzzles and the question of how to tune an OCL solver to automatically construct a concrete instance could be become a relevant benchmark to move this type of tools forward.

7 Conclusion and future work

This contribution has proposed an approach for teaching formal reasoning with logical puzzles. The puzzle, which comprises its vocabulary, its structure and its clues, has been described with a UML model including OCL constraints. The approach can be used in various ways, for example, for formulating a given puzzle and finding a solution, and it can be used to automatically construct a new puzzle and clues.

Future work can be performed in a number of directions. Different sets of clues possess different properties with regard to finding a solution: (a) we have applied an exclusion approach by banning possible solutions until only one solution is left; (b) we would like to study in the future a forward reasoning approach in which 'new' facts are constructed until a solution is found. Currently, we do not support clues in *full* first-order logic, but clues are restricted to equations, inequations or simple conditionals. We would like to extend this to cover also negation, conjunction and disjunction over more than two ground formulas. Instead of having untyped properties, one could provide typed properties making certain domain invariants dispensable. The user interface of the current implementation could be improved by advanced interaction options. For example, one could generate visual representation for puzzle models. More case and in particular user studies must be performed in order to consolidate and to bring the approach to a satisfying degree of maturity.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. IEEE Softw. **20**(5), 36–41 (2003)
2. Bell, T., Rosamond, F., Casey, N.: Computer science unplugged and related projects in math and computer science popularization. The multivariate algorithmic revolution and beyond: Essays dedicated to Michael R. Fellows on the occasion of his 60th Birthday pp. 398–456 (2012)
3. Butler, E., Torlak, E., Popović, Z.: Synthesizing interpretable strategies for solving puzzle games. In: Proceedings of the 12th International Conference on the Foundations of Digital Games, pp. 1–10 (2017)
4. Cabot, J., Gogolla, M.: Object Constraint Language (OCL): A definitive guide. In: M. Bernardo, V. Cortellessa, A. Pierantonio (eds.) Proc. 12th Int. School Formal Methods for the Design of Computer, Communication and Software Systems: Model-Driven Engineering, pp. 58–90. Springer, Berlin, LNCS 7320 (2012)
5. Cuadrado, J.S., Gogolla, M.: Model finding in the EMF ecosystem. J. Object Technol. **19**(2), 10–1 (2020)
6. Deitrick, E.: Graphical parsons puzzle creator: Sharing the full power of 2d parsons problems through a graphical, open-source online tool. In: Sherriff M., Merkle L.D., Cutter P.A., Monge A.E., Sheard J.(eds.) SIGCSE '21: The 52nd ACM Technical Symposium on Computer Science Education, Virtual Event, USA, March 13-20, 2021, p. 1379. ACM (2021). https://doi.org/10.1145/3408877.3439548
7. Falkner, N., Sooriamurthi, R., Michalewicz, Z.: Teaching puzzle-based learning: development of basic concepts. Teach. Math. Comput. Sci. **10**(1), 183–204 (2012)
8. Gogolla, M., Büttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. J. Sci. Comput. Program. **69**, 27–34 (2007)
9. Gogolla, M., Hilken, F., Doan, K.: Achieving model quality through model validation, verification and exploration. Comput. Lang. Syst. Struct. **54**, 474–511 (2018). https://doi.org/10.1016/J.CL.2017.10.001
10. Groza, A.: Modelling Puzzles in First Order Logic. Springer Nature, Cham, Switzerland (2021)
11. Groza, A., Baltatescu, M.M., Pomarlan, M.: Minefol: A game for learning first order logic. In: 2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP), pp. 153–160. IEEE (2020)
12. Groza, A., Nitu, C.: Natural language understanding for logical games. arXiv preprint arXiv:2110.00558 (2021)
13. Kästner, A., Gogolla, M., Doan, K.H., Desai, N.: Sketching a Model-Based Technique for Integrated Design and Run Time Description. In: Proc. STAF 2018 Workshops, Workshop Model-Driven Engineering for Design-Runtime Interaction in Complex Systems (DeRun 2018), p. 529–535 (2018)
14. Kumar, A.N.: Epplets: A tool for solving parsons puzzles. In: Barnes T., Garcia D.D., Hawthorne E.K., Pérez-Quiñones M.A. (eds.) Proceedings of the 49th ACM Technical Symposium on Computer Science Education, SIGCSE 2018, Baltimore, MD, USA, February 21-24, 2018, pp. 527–532. ACM (2018). https://doi.org/10.1145/3159450.3159576
15. Lara, J.D., Guerra, E., Cuadrado, J.S.: When and how to use multilevel modelling. ACM Transact. Softw. Eng. Methodol. (TOSEM) **24**(2), 1–46 (2014)
16. Liu, T., Gonzalez-Maldonado, D., Harlow, D.B., Edwards, E.E., Franklin, D.: Qupcakery: A puzzle game that introduces quantum gates to young learners. In: Doyle M., Stephenson B., Dorn B., Soh L., Battestilli L. (eds.) Proceedings of the 54th ACM Technical Symposium on Computer Science Education, Volume 1, SIGCSE 2023, Toronto, ON, Canada, March 15-18, 2023, pp. 1143–1149. ACM (2023). https://doi.org/10.1145/3545945.3569837
17. Meyer, E.F., Falkner, N., Sooriamurthi, R., Michalewicz, Z.: Guide to teaching puzzle-based learning. Springer (2014)
18. Ortíz-García, E.G., Salcedo-Sanz, S., Leiva-Murillo, J.M., Pérez-Bellido, Á.M., Portilla-Figueras, J.A.: Automated generation and visualization of picture-logic puzzles. Comput. Gra. **31**(5), 750–760 (2007)
19. Orwant, J.: EGGG: automated programming for game generation. IBM Syst. J. **39**(3.4), 782–794 (2000)
20. Rosenhouse, J.: Games for your mind: the history and future of logic puzzles. Princeton University Press, New Jersey (2022)
21. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, 2nd Edition (2004)
22. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, 2nd Edition (2004)

Martin Gogolla was professor for Computer Science at University of Bremen, Germany and head of the Research Group Database Systems. He has retired in 2020 but is still teaching and doing research. His interests include software development with object-oriented approaches and formal methods in system design. He is still working on concepts and tools for languages like UML and OCL.

Jesús Sánchez Cuadrado is an Associate Professor at the Languages and Systems Department of the University of Murcia. His research is focused on Model Driven Engineering (MDE) topics, notably model transformation languages, meta-modelling and domain specific languages and recently in the application of AI to software modeling. On these topics, he has published several articles in journals and peer-reviewed conferences, and developed several open source tools. His e-mail address is jesusc@um.es and his web-page is http://sanchezcuadrado.es.