User Assistance Characteristics of the USE Model Checking Tool

Frank Hilken and Martin Gogolla

University of Bremen, Computer Science Department, 28359 Bremen, Germany
{fhilken,gogolla}@informatik.uni-bremen.de

Abstract. The Unified Modeling Language (UML) is a widely used general purpose modeling language. Together with the Object Constraint Language (OCL), formal models can be described by defining the structure and behavior with UML and additional OCL constraints. In the development process for formal models, it is important to make sure that these models are (a) correct, i.e. consistent and complete, and (b) testable in the sense that the developer is able to interactively check model properties. The USE tool (UML-based Specification Environment) allows both characteristics to be studied. We demonstrate how the tool supports modelers to analyze, validate and verify UML and OCL models via the use of several graphical means that assist the modeler in interpreting and visualizing formal model descriptions. In particular, we discuss how the so-called USE model validator plugin is integrated into the USE environment in order to allow non domain experts to use it and construct object models that help to verify properties like model consistency.

Keywords: UML · OCL · Model Checking · User-Interface integration

1 Introduction

Model-Driven Engineering (MDE) is an approach to software development concentrating on models in contrast to traditional code-centric development approaches. Within MDE, models are frequently formulated in the Unified Modeling Language (UML) with accompanying formal restrictions expressed in the Object Constraint Language (OCL). UML models are visually specified with several diagram kinds emphasizing structural and behavioral system aspects. Visual model descriptions offer a great potential for a user-friendly development process. Naturally, tools must take up the challenge and provide interfaces that support the developer in an easygoing way.

The present paper studies formal system descriptions employing UML class diagrams that are restricted by OCL invariants. The feature set of UML class diagrams that is handled here and the employed OCL elements pose a formal semantics. We regard this combination of UML and OCL as a formal method. The aim of this contribution is to demonstrate how the development of formal UML and OCL models can be supported by a user-friendly interface. Employing this interface it is possible to verify properties like model consistency.

Work concerning user-friendly interfaces shows the necessity of such [3,15] and tool creators are well aware of this and provide the means [2,10,14]. There are many approaches that concentrate on the improvement of prover interfaces [4,1] to make formal techniques more accessible have been recently proposed. Another line of research is the combination of formal techniques and their representation with visual languages in order to improve usability [13,11].

The rest of this paper is structured as follows. Section 2 introduces a running example and the tool USE (UML-based Specification Environment) that is applied here. Section 3 presents a short journey through the graphical user-interface of the USE model validator, a component that is able to build manifestations of a formal UML and OCL model in form of object diagrams. The configuration features of the GUI of the model validator are explained in a systematic way in Sect. 4. Section 5 discusses the USE support for analyzing potential modeling problems. The paper is finished in Sect. 6 with stating conclusions and indicating future work.

2 Preliminaries

2.1 Running Example in UML and OCL

In UML, class diagrams describe the structure of models with classes and class attributes, which are templates for 'things' and their properties, e.g. persons and their personal information. Additionally, associations put the classes in relation with each other. Figure 1 (left) shows the running example model description. It shows a *Car Rental* model with cars that are assigned to branches and their maintenance history. Additionally, there is a categorization for the cars into car groups. Finally, customers can rent cars from the branches that are run by their employees. The model uses a wide variety of UML features.

The class diagram is instantiated to create actual scenarios to describe car rentals. These system states are represented by UML object diagrams. They are restricted by the semantics of the class diagram and must satisfy all model inherent constraints given by, e.g. generalizations, multiplicities or compositions (not present in this model).

In addition to the UML descriptions, OCL invariants are used to employ further restrictions on the model that are not expressible by UML alone. These constraints are pictured in Fig. 1 on the right. They handle further relations between classes and ensure that the model does not allow system states that are not intended. For example, in the car rental model, the categorization of the car groups shall be cycle free and all employees must be connected to a branch, which cannot be handled by multiplicities, because there are multiple ways to represent this relation (`Employment` and `Management`).

The goal is to create a model description that can represent all intended situations but not more. Using model checking tools, the models can then be checked for certain properties. Usually, these properties regard safety, but other concerns can be checked as well. A valid system state must satisfy all model inherent constraints as well as all concrete constraints given by the OCL invariants.

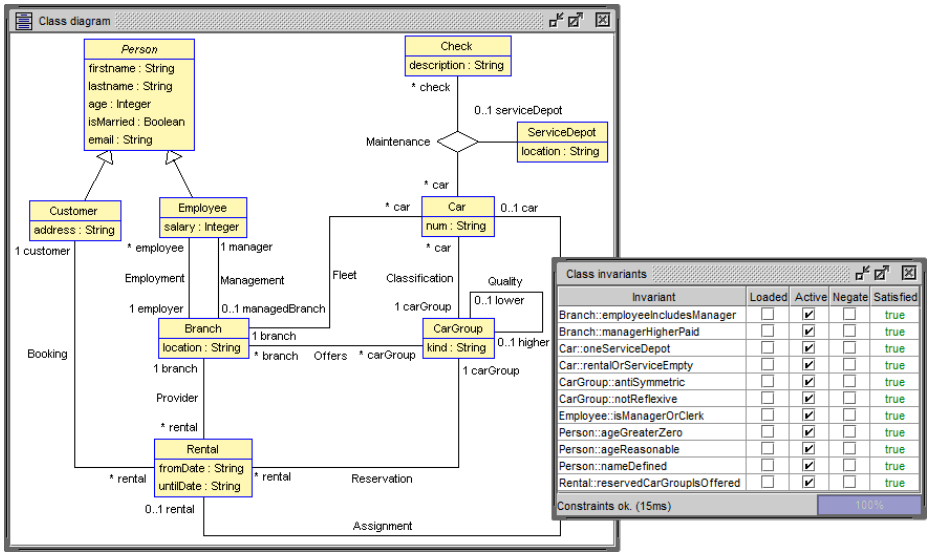

Fig. 1. Car rental running example class diagram (left) and invariants (right).

2.2 Model Verification with the USE Tool

In this paper, the *UML-based Specification Environment* tool (USE, [7]) is used together with the so-called *model validator* plugin that allows to generate system states for UML/OCL models based on a relational logic encoding [12]. There is earlier work showing features of the USE tool that help to analyze and debug OCL expressions, namely the evaluation browser [5], but we concentrate on the model validator and its model checking aspects in this work.

In order for the model validator to search for a valid system state, it needs several inputs:

- A description of the *model* in the form of a class diagram optionally enhanced with invariants given as OCL expressions, see Fig. 1.
- A *configuration*, which defines the search space by providing the domain of basic data types (Integer, String and Real) and lower and upper bounds for classes and associations, i.e. specifies how many objects of each class – or links of each association respectively – are required and how many are allowed at most. Furthermore, the configuration contains rules for the assignments of class attributes, e.g. restricting domain values for certain attributes specifically. Finally, it allows to disable or negate invariants.
- Optionally a *partial system state* can be instantiated before the validation process that is used as a base for the task. The model validator adds elements to the system state until it: (a) conforms the bounds given in the configuration; and (b) is a valid system state as defined by the model. This can be seen as a lower bound on the model level.

– Smaller, model independent parameters include the choice of the *SAT solver* and *bitwidth* for the encoding of the model.

The second bullet point, the configuration, previously required the modeler to edit a text file containing key-value pairs to setup values for certain keys. The keys are determined by the model. For example for each class and association a lower and an upper bound is expected. The required values are mostly numbers, but more complex constructs were required to specify, e.g. preexisting links. This process requires a deep understanding of the keys that exist and the syntax to enter the values. Moreover, special values exist for some keys with different meanings, e.g. unlimited upper bounds. Even experienced users regularly required the manual of the configuration.

In the following sections, we explain how a new graphical user interface helps to simplify the configuration process and reduce the necessity for a separate manual to the tool. Furthermore, additional analysis features of the tool are presented to help identify potential problems with the verification process caused by the inputs.

3 Iterative Instantiation of the Running Example

We now study for our running example four use cases corresponding to four model validator configurations that result in UML object diagrams being displayed in Fig. 2, Fig. 3, Fig. 4 and Fig. 5. The configuration names that also appear in the figures are as follows:

```
0_DEF-VALS_NO-OBJ_NO-INVS
1_DEF-VALS_CUS-EMP-BRA_INVS
2_DEF-VALS_CUS-EMP-BRA-REN-CG-CAR_INVS
3_APP-VALS_CUS-EMP-BRA-REN-CG-CAR_INVS
```

`DEF-VALS` means only default data type values are used, `APP-VALS` expresses that application specific values are considered. The `0` configuration generates no objects and switches off all invariants. The list of shortcuts of class names indicates which classes are considered. `INVS` says that the invariants belonging to the respective classes are taken into account.

Figure 2 shows the basic structure of the GUI with three tabs for (1) the datatypes, (2) the classes and associations, and (3) the invariants. For example, the datatype integer is configured to take values from -10 to 10, and at most 10 default strings will be used for the object model that is to be constructed. All classes and associations will not be populated, as required by the '0' entries for the objects and links. All invariants are switched off. The datatype standard values determined here will be used in the following two configurations.

In Fig. 3 the population of the three classes Customer, Employee, and Branch is activated requiring exactly one object in each class. Two of the five associations in which the class Branch participates are required to have exactly one link. The invariants for these classes now become activated. The resulting object diagram utilizing standard datatype values is displayed in the bottom.

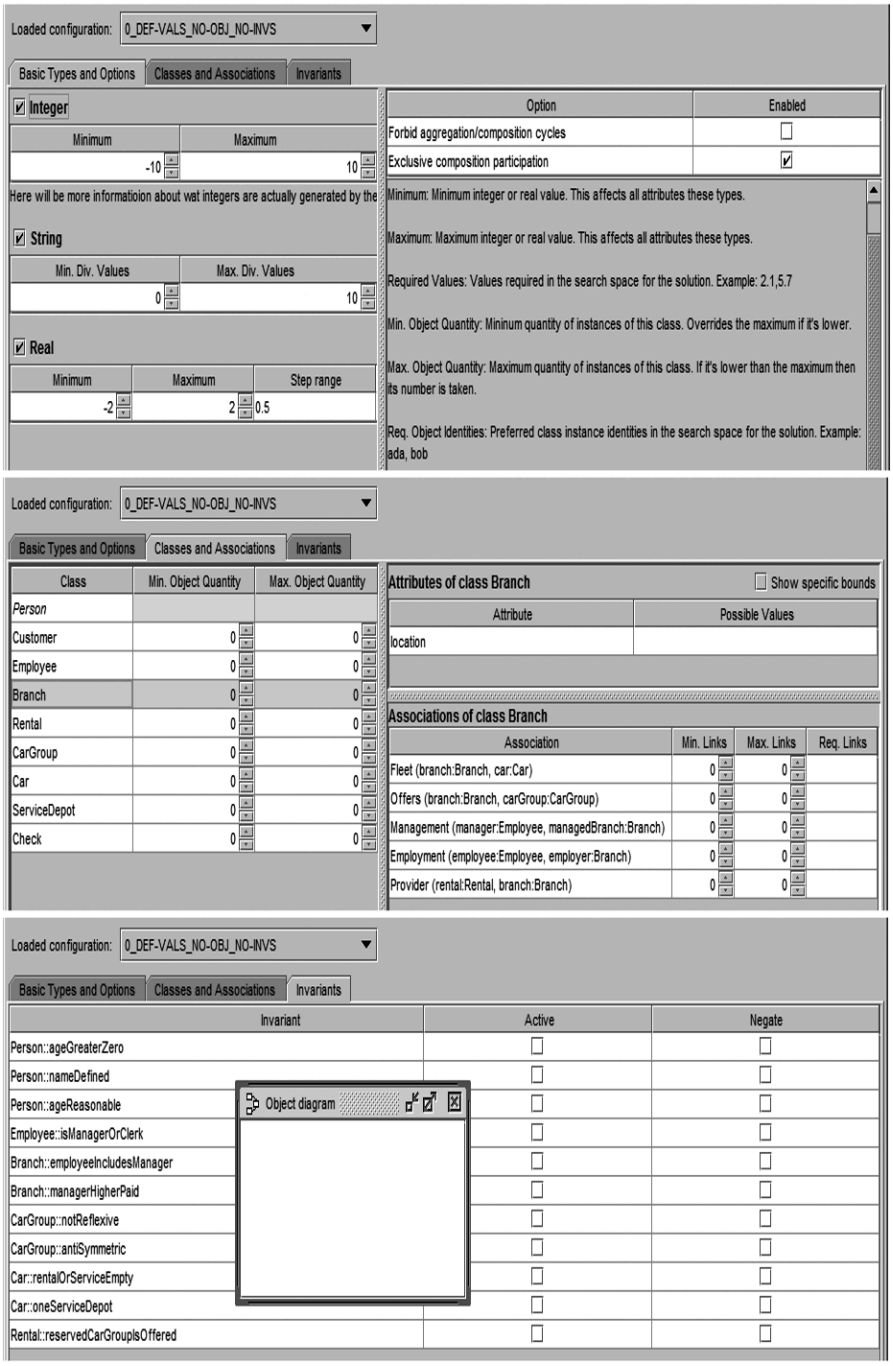

Fig. 2. Configuration GUI with the default domain values, no objects and all invariants deactivated. Since no objects are given, the generated system state is empty.

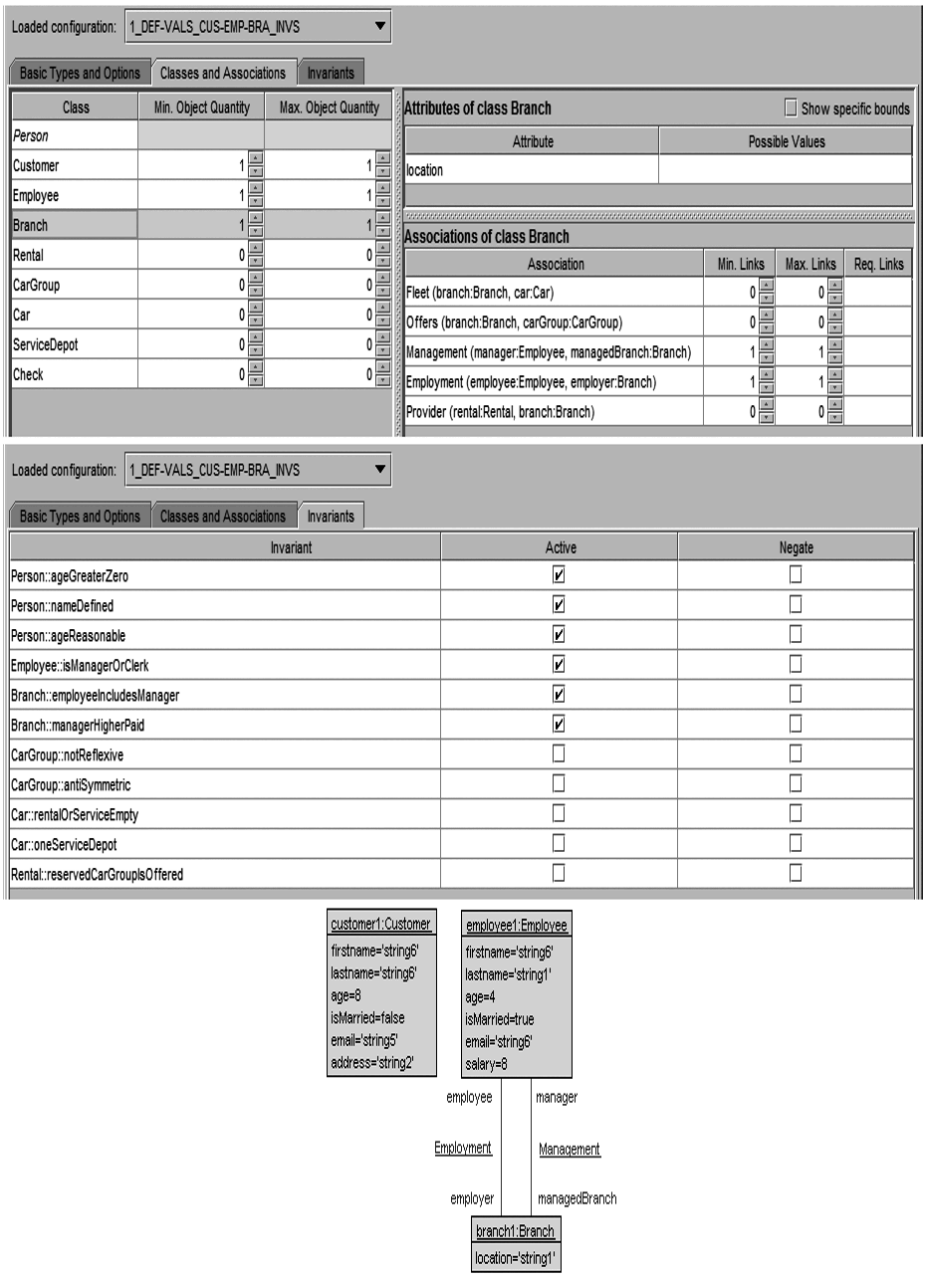

Fig. 3. Configuration of parts of the model considering only a few classes and associations. Only invariants of the considered classes are active. The object diagram shows a simple system state with a branch, an employee and a customer.

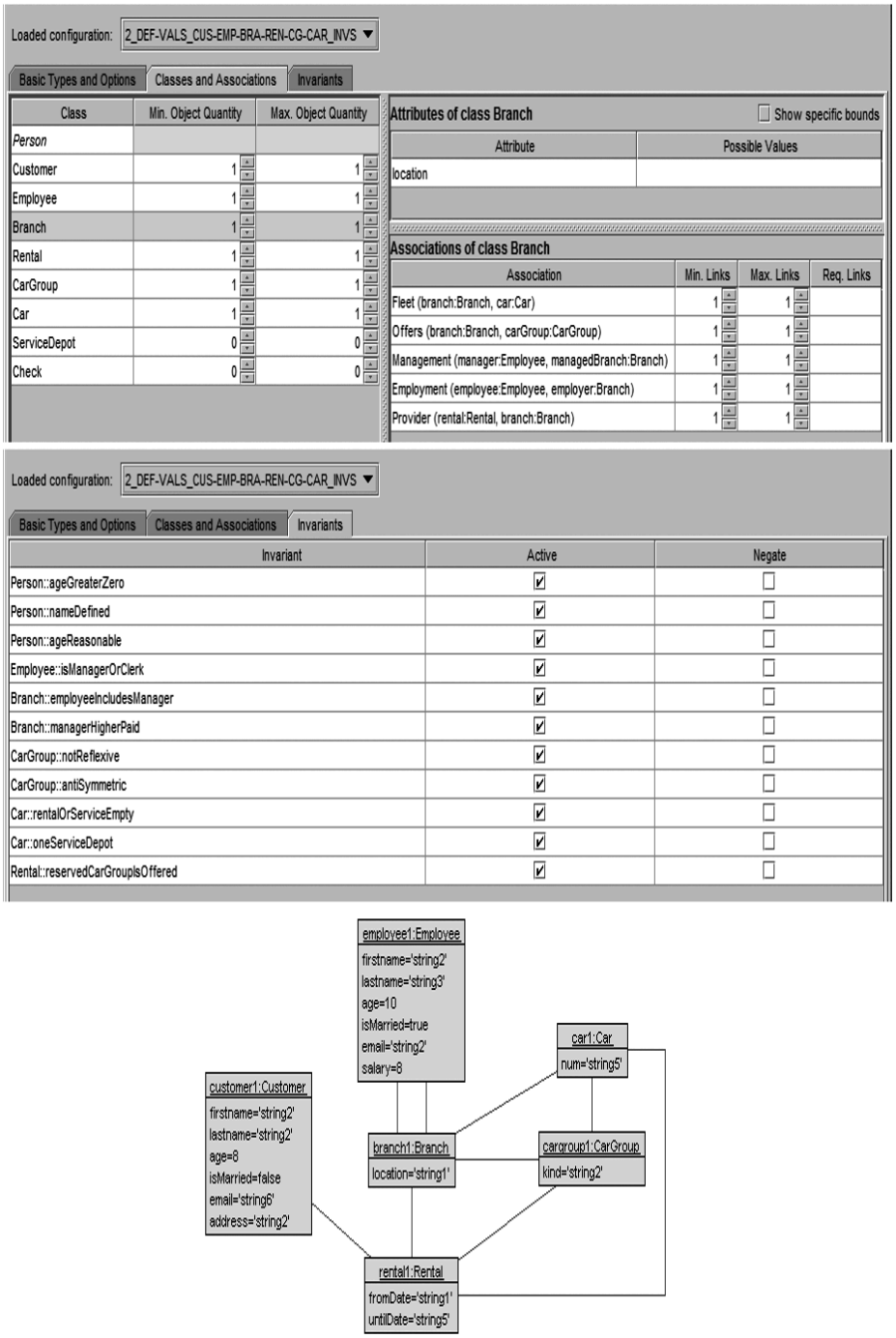

Fig. 4. The second iteration of the configuration considers more classes and all invariants. The resulting system state shows a complete rental process.

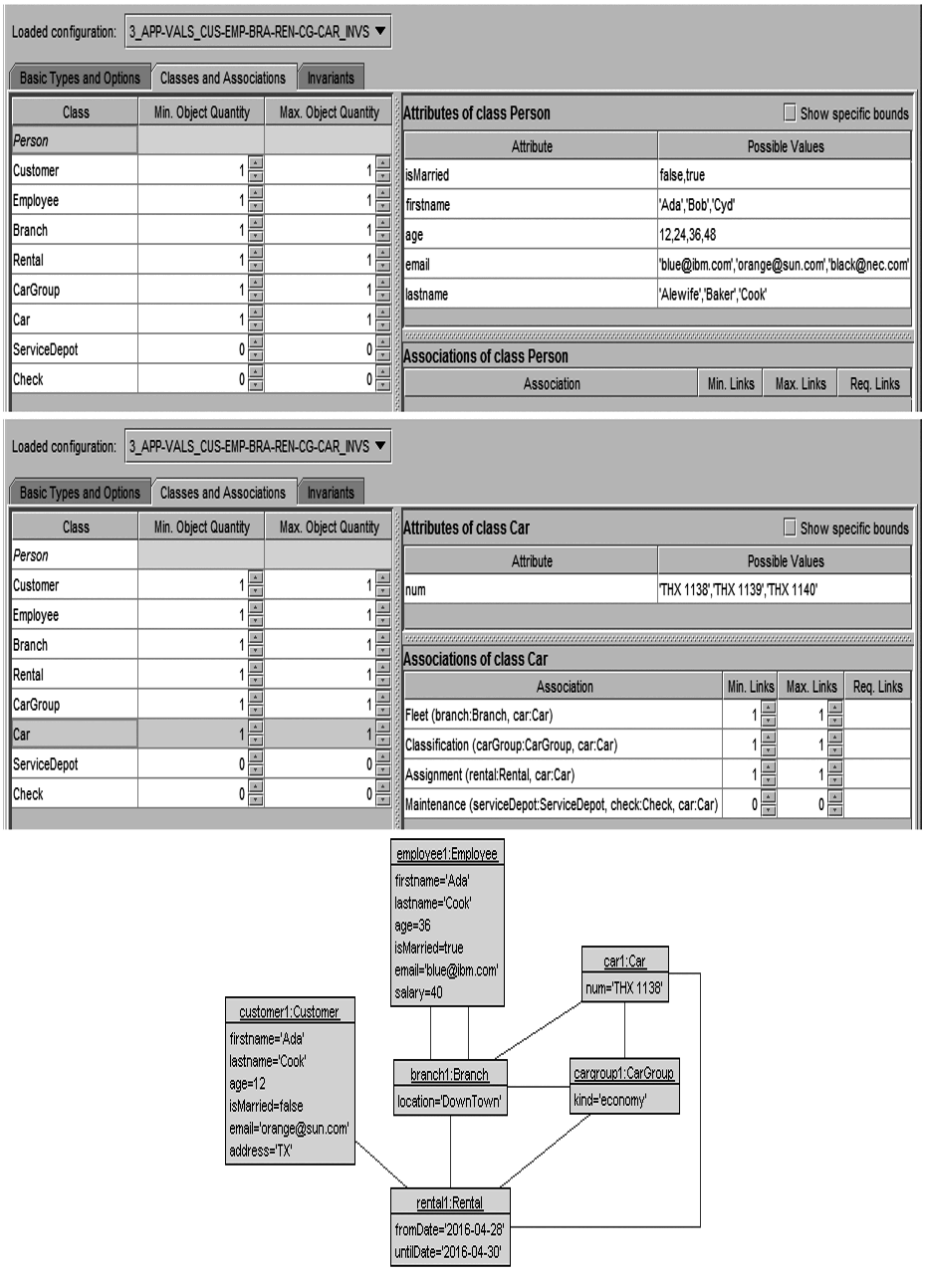

Fig. 5. The final configuration is cloned from the previous, but all attribute domains have been defined with relevant values. The resulting system state represents real situations much more closely and is easier to understand with less abstract values.

Figure 4 concentrates on the classes Rental, Car, and CarGroup, and all associations with participants in the classes and all invariants become active. Again, exactly one object per class and one link per association is constructed. Because we have employed the general, default datatype values for different purposes, rather uncommon attribute values show up, e.g. the string value 'string2' is displayed as the CarGroup kind and the first name of both person objects.

Lastly, in Fig. 5 application specific datatype values are employed. They are entered in the respective line for the attribute belonging to a particular class. They lead in the constructed object diagram to a state that seems more realistic, more domain specific than the previous object diagram. Not all classes and all associations are activated yet, as the '0' entries in the classes and association tables indicate.

4 Configuration GUI

In this section, we describe the visible and behind-the-scenes features of the configuration GUI that assist the modeler in setting up the model domains and bounds for their verification tasks. The features are a collection of basic GUI concepts enhanced with experience from personal and student work as well as support questions to provide assistance for common use cases.

First off, with the release of the configuration GUI the interface of configuration files was extended to allow storing multiple configurations in one file. These configurations can be named individually and the configuration GUI offers operations to manage them, e.g. cloning configurations etc. This allows for an easy iterative construction of the configurations as shown in Sect. 3. The GUI also offers common file system operations to deal with the generated configuration files. For convenience, a configuration file with the same base name as the loaded model is automatically opened if one exists.

The GUI is split into three tabs that each cover a part of the configuration and are detailed in the following sections. The first tab is the basic types tab in which the domains for the basic types of OCL are defined. The second tab defines the model dependent bounds and domains and the final, third tab configures the invariants.

4.1 Basic Type Configuration

The *Basic Types and Options* tab offers the possibilities for configuring the default values that attributes can take. The default values become relevant when an attribute in a class is not given a special treatment in the *Classes and Associations* tab as described below. Furthermore in this tab two global options can be set: it is possible (1) to disallow cycles in the constructed object diagram for aggregation and composition relationships (whole-part relationships), and (2) to forbid the possibility that one part object is included two or more whole objects when considering composition relationships.

The default values for three basic UML and OCL data types (Integer, String and Real) can be described here. The values for type Integer are determined by the allowed minimum and maximum value. For example, the setting in Fig. 2 determines the set {-10..10}. The values for type String are fixed basically by an upper bound for the maximum number of diverse values that are used for String-valued attributes. For example, an upper bound of '10' will provide the ten string constants 'string1', ..., 'string10'. The values for type Real are settled by the minimum and the maximum Real value and an additional step range. For example, the configuration in Fig. 2 will establish the set {-2, -1.5, -1, -0.5, 0, 0.5, 1.0, 1.5, 2.0}.

Default attributes values are typically considered in the early usage phase of the model validator. The developer is given the option to employ non-specific values in order get a first impression of possible object diagrams for the model. Later, the default values can be made more application-oriented by providing domain-specific values.

4.2 Model Bound and Domain Configuration

The *Classes and Associations* tab supports the developer in configuring the objects, the links, and particular attribute values.

In this tab each model class can be turned into the focus of editing by clicking on the respective class name in the left part of the tab. The click activates the display of the class attributes and the associations in which the class participates. These attributes and associations are then shown in the right part of the tab. For example, in the upper part of Fig. 5 class Person and its attributes are shown, whereas in the middle part class Car with its attributes and associations are displayed. The minimum and the maximum number of objects that have to be present in the constructed object diagram can be determined. The object identities can be either automatically constructed identities like car1, car2 or the identities can be taken from an explicitly given list like ada, bob, cyd, e.g. for class Customer. For abstract classes that do not posses instances on their own but only through their subclasses, minimum and maximum object numbers cannot be specified.

Only the attributes of the currently selected class are shown in the right part of the tab. The attribute values are settled by the default values if no entry is made for the particular attribute. An important option here is to settle attribute specific values, so that the constructed object diagram becomes closer to the imagination of the developer. For example, in Fig. 5 the attribute 'firstname' is made concrete with the String values 'Ada', 'Bob', 'Cyd' and the attribute 'age' is made specific with the Integer values 12, 24, 36, 48.

The associations of the currently selected class are displayed in the right part of the tab below the attributes. For an association, its name, its participating classes, and the respective role names are stated. The constructed object diagram is determined by a minimum and maximum number of links that have to be present. In addition, a list of mandatory, required links can be entered. For example, one could require the `Management` links (ada,downTown), (bob,northStreet)

provided respective objects have been defined in the participating classes. In general, the settings for a given binary association Assoc(C,D) can be manipulated in two places, namely when the class C or the class D has been selected.

4.3 Invariant Configuration

The *invariants* tab allows to control how model invariants are considered by the model validator. The default behavior renders all invariants active, i.e. the resulting system state must satisfy all invariants. These invariants put constraints on the relations between, e.g. classes, attributes, etc. For partial instantiations this behavior is undesirable because often invariants require that all elements mentioned are present in the system state or the invariants are not satisfied, which prevents partial system states to be found. For the purpose of partial instantiations, we allow to deactivate invariants with the effect that they are ignored by the model validator.

Deactivating invariants has to be done with care, since they enforce properties ensuring valid system states. Partly, this is achieved by the fact that invariants are defined on classes and are only checked for the instances of that particular class. Therefore, if no instance of a class exists, all its invariants are satisfied by definition. We are interested in identifying invariants that relate between multiple classes and associations of which some are elements that are instantiated and some are not. These type of invariants prevent partial system states to be instantiated. To analyze invariants, the USE tool offers a visualization of OCL expressions coverage in the diagrams. Figure 6 shows the running example class diagram with those elements colored that are affected by the invariant `Rental::reservedCarGroupIsOffered`. The darker the color of an element the more prominent it is in the invariant. This visualization allows to quickly determine the scope of an invariant without having to look at the complex textual expressions. Figure 3 shows the configuration of the partial system instantiation with the inactive classes (lower and upper bound equal zero) and invariants.

Finally, the invariants tab allows to negate invariants, i.e. the invariant shall fail for every object. This is useful to setup certain model verification tasks, e.g. invariant independence [8]. The information is then saved in the configuration file along with the information from the other tabs.

5 Analysis of Potential Modeling Problems

So far, we have shown how the user is assisted by the graphical user interface to setup the configuration of a verification task. However, besides a bad configuration, other problems can interfere with the checking process, in particular problems that the user is not aware of.

UML and OCL are rich languages filled with features for all kinds of purposes. Trying to support all of them is not only a lot of work, but also reduces the efficiency of the tools, the more features they support [9]. Therefore, it is common practice to restrict UML and OCL verification engines to a subset of

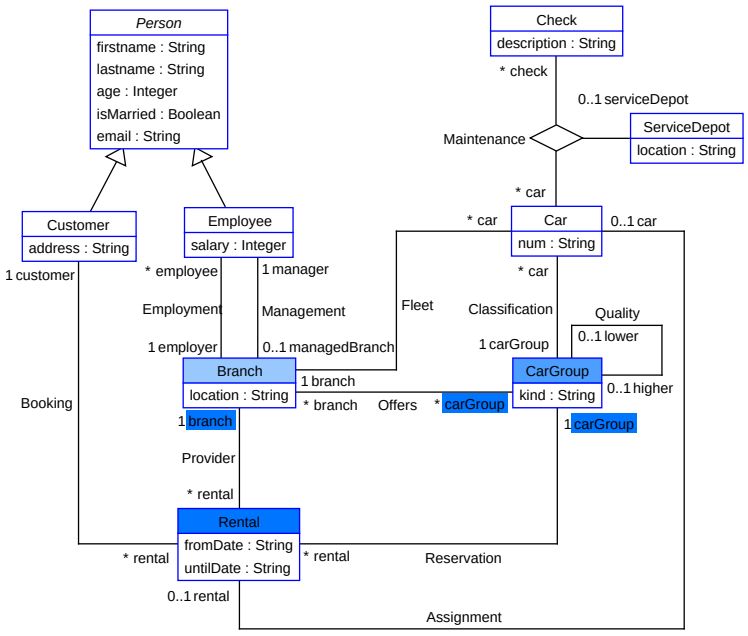

Fig. 6. Class diagram coverage of invariant `Rental::reservedCarGroupIsOffered`.

the languages. This results in some features being completely unsupported and others only having limited support. Both categories pose problems to the users of the tools. If there are no means in place to detect the limitations, the outcome might differ from the user's expectations and it is not feasible to keep track of all limitations from long tool manuals.

The verification engine of the USE tool, the model validator, has good support for UML and OCL, but also has limits. For example the data type `String` is only represented as a token, meaning that many operations like `size`, `substring` or `concat` are not supported – only equality can be checked. When a model with unsupported elements is trying to be used in the model validator, the translation process will fail and inform the user about the exact UML/OCL feature that made the translation fail. In case that the unsupported element is within an invariant, the user can decide whether the invariant can be ignored or rewritten using other expressions.

More obscure problems occur with features that are not fully supported. The underlying solving engine of the model validator is based on relational logic, which has great support for set operations and, thus, the integration of the OCL `Set` collection type is extensive. Adding support for the other collection types `Bag`, `Sequence` and `OrderedSet` would pose a significant overhead though, i.e. will be less efficient. This restriction to the `Set` collection type is particularly problematic for OCL navigation expressions. This expression allows to navigate the classes of the model using associations, more precisely the roles

visible in Fig. 1. Usually a 1-n navigation results in a set of elements, because at most one link is allowed between two objects, but under certain circumstances – namely starting with a set of objects rather than a single one – the navigation results in the duplicate preserving `Bag` type, because the result might contain the same value multiple times after the navigation. For example the expression `Set{-2,0,2}->collect( n | n*n )` to square numbers in a set results in `Bag{4,0,4}` in order to be able to preserve the information that the 4 is in the result twice. Due to the implicit nature of this effect and the strict interpretation of bags as sets in the model validator, simple expressions might already suffer unintended side effects. To help identify those potential problems, the occurrences are made visible to the modeler via a warning.

```
WARNING: Collect operation '[...].employee.age' results in unsupported
    type 'Bag'. It will be interpreted as 'Set'.
```

The implicit type change from `Set` to `Bag` in OCL, which is consequently interpreted as `Set` by the model validator, brings more potential problems with it. Most OCL collection operations are defined on all collection types, but the results are different. Consider the operation `sum()`, which sums all integer elements of a collection returning a single integer. Here, the implicit conversion from `Set` to `Bag` is usually helpful when collecting, for example, the ages of persons to calculate an average. However, the interpretation as a `Set` does not work in this situation. To assist the user, the model validator checks for these situations and warns the modeler about this potential problem, which only the modeler can decide whether it needs to be addressed or not.

```
WARNING: The evaluation of sum expression '[...].employee.age->sum()'
    might be wrong if source contains duplicates (Collection is
    interpreted as Set).
```

Other problems might arise from contradictions in the model itself. Navigation expressions through the model can become quite long and obscure the resulting type. In these situations, typecasts like `oclAsSet()` need to be used to be able to compare differing types, but the textual representation of OCL alone is often insufficient to recognize such disparities. USE is able to structurally analyze the expressions in the model for type contradictions and gives hints about (sub)expressions that were determined to be contradicting, resulting in constant values.

```
WARNING: Expression 'Set{ 1 } = Bag{ 1 }' can never evaluate to true
    because 'Set(Integer)' and 'Bag(Integer)' are unrelated.
WARNING: Expression 'Set{ 'str' }->intersection(Set{ 1 })' can never
    evaluate to more than an empty set, because the element types
    'String' and 'Integer' are unrelated.
```

Finally, the underlying solving engine is bound to a bitwidth that has to be specified with the verification task. Setting the bitwidth as small as necessary increases the efficiency of the tool, thus we leave the task to the user to choose an appropriate bitwidth. But if the bitwidth is chosen too small, undefined behavior occurs when dealing with arithmetic operations exceeding the bitwidth. Finally,

if the user is not aware of – or forgets – that the bitwidth is specified in two's-complement, off-by-one errors can occur. In order to alleviate the problem, the model validator analyzes the configuration and model description[1] for integer literals and checks them against the given bitwidth. If it is determined that the chosen bitwidth is too small, a warning is displayed including the appropriate bitwidth for the current model and configuration.

```
WARNING: The configured bitwidth is too small for the property Integer
    max value (237). Required bitwidth: 9 or greater.
```

6 Conclusion and Future Work

We have presented the USE tool together with its model validator plugin and have shown the steps necessary to apply model checking to given UML/OCL models. Furthermore, the simplifications of the process by integrating the graphical user interface have been discussed and the possibilities of the configuration GUI and the coverage mode are demonstrated. The translation of the textual OCL expressions into visible, quick and easy to understand visualizations help the user to confidently handle models. Finally, the possibilities of the model validator plugin to detect potential problems have been demonstrated to guide users in finding incompatibilities in their models.

Besides the configuration of the model domains and bounds, we have presented more aspects that have to be setup before a system state can be generated including the verification task itself, e.g. by manipulating the invariants. Future work should concentrate on the simplification of all steps of the setup and provide easy interfaces for each of them. Additionally, the evaluation of the configuration GUI and other presented interfaces is an ongoing process and new assistance features are constantly added and improved.

Acknowledgement. We thank Subi Aili for his contributions to the configuration GUI – ideas and implementation – in his diploma thesis.

References

1. Adams, M.: Refactoring proofs with tactician. In: Bianculli, D., Calinescu, R., Rumpe, B. (eds.) Software Engineering and Formal Methods - SEFM 2015. Lecture Notes in Computer Science, vol. 9509, pp. 53–67. Springer (2015)
2. Arshad, F., Mehmood, H., Raza, F., Hasan, O.: g-hol: A graphical user interface for the HOL proof assistant. In: Artho, C., Ölveczky, P.C. (eds.) Formal Techniques for Safety-Critical Systems, FTSCS 2015. Communications in Computer and Information Science, vol. 596, pp. 265–269. Springer (2015)

[1] Integer literals that occur in OCL expressions in the model need to be within the bitwidth as well.

3. Beckert, B., Grebing, S.: Evaluating the usability of interactive verification systems. In: Klebanov, V., Beckert, B., Biere, A., Sutcliffe, G. (eds.) Comparative Empirical Evaluation of Reasoning Systems. CEUR Workshop Proceedings, vol. 873, pp. 3–17. CEUR-WS.org (2012)
4. Beckert, B., Grebing, S., Böhl, F.: A usability evaluation of interactive theorem provers using focus groups. In: Canal and Idani [6], pp. 3–19
5. Brüning, J., Gogolla, M., Hamann, L., Kuhlmann, M.: Evaluating and Debugging OCL Expressions in UML Models. In: Brucker, A.D., Julliand, J. (eds.) Proc. 6th Int. Conf. Tests and Proofs (TAP 2012). pp. 156–162. Springer, Berlin, LNCS 7305 (2012)
6. Canal, C., Idani, A. (eds.): Software Engineering and Formal Methods - SEFM 2014, Lecture Notes in Computer Science, vol. 8938. Springer (2015)
7. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
8. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, independence and consequences in UML and OCL models. In: Dubois, C. (ed.) Tests and Proofs, TAP. Lecture Notes in Computer Science, vol. 5668, pp. 90–104. Springer (2009)
9. Hilken, F., Niemann, P., Gogolla, M., Wille, R.: Filmstripping and unrolling: A comparison of verification approaches for UML and OCL behavioral models. In: Seidl, M., Tillmann, N. (eds.) Tests and Proofs, TAP. LNCS, vol. 8570, pp. 99–116. Springer (2014)
10. Homik, M., Meier, A.: Designing a GUI for proofs - evaluation of an HCI experiment. CoRR abs/0903.3926 (2009)
11. Idani, A., Stouls, N.: When a formal model rhymes with a graphical notation. In: Canal and Idani [6], pp. 54–68
12. Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back. In: France, R., Kazmeier, J., Breu, R., Atkinson, C. (eds.) Proc. 15th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2012). pp. 415–431. Springer, Berlin, LNCS 7590 (2012)
13. Ladenberger, L., Dobrikov, I., Leuschel, M.: An approach for creating domain specific visualisations of CSP models. In: Canal and Idani [6], pp. 20–35
14. Lapets, A., Kfoury, A.J.: A user-friendly interface for a lightweight verification system. Electr. Notes Theor. Comput. Sci. 285, 29–41 (2012)
15. Lüth, C.: User interfaces for theorem provers: Necessary nuisance or unexplored potential? ECEASST 23 (2009)