# Four-valued Logic in UML/OCL Models: A "Playground" for the MVL Community
(Tutorial Paper)

Nils Przigoda[1] Judith Przigoda[1] Robert Wille[2]
[1]Siemens Mobility GmbH, Braunschweig, Germany
[2]Institute for Integrated Circuits, Johannes Kepler University Linz, Austria
nils.przigoda@siemens.com, judith.przigoda@siemens.com, robert.wille@jku.at

***Abstract*—The *Unified Modeling Language* (UML) together with the *Object Constraint Language* (OCL) are *the* description means for modeling and specifying, e. g., software systems in early stages of the design. They allow to define components, their relations, and constraints of a system while, at the same time, hide precise implementation details. Despite providing a "blueprint" for the desired systems, UML/OCL descriptions also allow for an early validation and verification of the design. However, an often overseen feature of UML/OCL is that it explicitly allows for the consideration of irregular variables assignments such as `null` and `invalid`—yielding a four-valued logic in the current UML/OCL version. In this tutorial, we provide an overview on this feature and the resulting four-valued UML/OCL logic. More precisely, we are providing a review of the corresponding description means as well as existing methods that allow for a validation and verification of the corresponding models. By this, we are aiming to introduce those UML/OCL descriptions and methods to the MVL community in order to trigger new directions for research and application.**

## I. Introduction

Along with the increasing complexity of modern computer systems, the design, implementation, and verification of corresponding hardware as well as software became more complex as well. As a means of reducing said complexity, more abstraction levels have been introduced to keep the increase of complexity in each level manageable. In the hardware domain, corresponding abstraction levels include the *gate level*, the *Register Transfer Level* (RTL), as well as the *Electronic System Level* (ESL), while the software domain relies on descriptions based on machine code, assembly code, or high level languages. Besides that, modeling languages are used that serve as an additional "bridge" between the initial (textbook) specification as well as a first implementation. Here, the Unified Modeling Language (UML, [16]) together with the *Object Constraint Language* (OCL, [15]) are *the* description means used to define systems from a very abstract point of view.

Moreover, UML/OCL does not only allow a concise modeling and specification of a system to be implemented; it additionally enables the designer to (automatically) conduct validation and verification [2, 4, 10, 12, 21, 27, 28], code-generation [11, 17], as well as other design tasks. To this end, a substantial number of corresponding methods and tools have been proposed in the past years—eventually forming a rather large and active community working on this topic.

However, most of the existing work focused thereby on the consideration of regular values within UML/OCL only, i. e., the consideration of a logic composed of Booleans, integers, etc. and, hence, resting on a two-valued basis with `true` and `false` as elementary information unit. But an often overseen feature of UML/OCL is that, in its newest version, UML/OCL additionally allows for the consideration of irregular variable assignments. Here, the original Boolean values `true` and `false` were no longer deemed sufficient, but are enriched with a third value $\varepsilon$ representing the `null` (pointer)—as known from programming languages—as well as a fourth value $\perp$ representing errors (similar to e. g., the exceptions from Java). This yields a four-valued logic of UML/OCL incorporating `true` and `false` as well as $\varepsilon$ and $\perp$. While this certainly introduces new potential for both, the UML/OCL community but maybe also for the MVL community, very few work towards a more in-depth consideration or exploitation of multiple-valued UML/OCL models has been done thus far.

In this tutorial, we aim at changing that by providing a basis on UML/OCL for the MVL community. Knowing that UML/OCL has hardly been considered by the MVL community thus far, we first start with a simple introduction into the "world" of UML/OCL in Section II—showcasing a small example. Followed by that, we provide a formal definition of the type system and logic behind UML/OCL in Section III—including the multiple-valued cases. Afterwards, we sketch and discuss the potential of the corresponding descriptions with respect to validation and verification of corresponding systems even before first implementations are available in Section IV.

Overall, this may provide the starting point for a more detailed consideration of UML/OCL issues such as validation and verification by the MVL community. In fact, using this tutorial, the formal basis as well as possible directions for further work (e. g., extending validation and verification approaches for the multiple-valued cases) are given—hopefully triggering new directions for research and application.

## II. The UML/OCL "World"

UML together with OCL offers a broad palette for modeling not only technical systems but also, e. g., concepts as well as data schemes. Overall, 14 different diagram types are offered by UML/OCL [16]. The probably most common types are class diagrams and object diagrams, which allow to describe components, attributes and relations of a system to be modeled as well as precise instantiations of all those, respectively. In this section, we will give a brief introduction to the "world" of UML/OCL by means of an example. By this, we provide a glimpse into the corresponding description means, before a more formal review is provided in the next section.

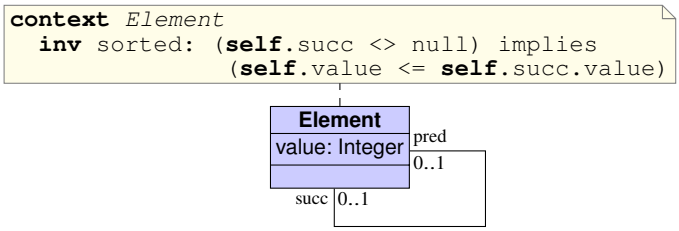



Figure 1: A simple list model

Figure 1 illustrates a model describing a simple list of `Integer` values. It consists of one class `Element` with an attribute to save the entry value and an association to connect elements with each other. Additionally, an invariant `sorted` makes sure that the values of the elements of such lists are in ascending order. Note that, although this example is very small, it will allow us to showcase some interesting aspects with respect to multi-valued logic later.

An instantiation of this model (also called system state) is shown in Figure 2(a) in terms of an object diagram. This system state is composed of two element objects (i. e., instances of the class element). The value of the first instance is 42 and the value of the second 1764. Thus, the invariant `sorted` evaluates to `true` and the system state is valid. Swapping the two values (as shown in Fig. 2(b)), of course, also changes the evaluation of the invariant to `false`—rendering the system state invalid. These examples of system states might give rise to the impression that the UML/OCL only rest on the two atomic values `true` and `false`.[1] However, this impression is wrong.

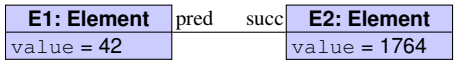
In fact, UML/OCL also allows for the value `null` (symbol: $\varepsilon$), which basically represents a null pointer. All instances of the attribute `value` can not only be assigned a *regular* integer but also be $\varepsilon$. In Fig. 2(c) the attribute of the second element has the value $\varepsilon$ instead of a regular integer value. Having a look at the evaluation of the invariant `sorted` again, the result will be neither `true` nor `false`. Instead the result of the evaluation will be `invalid` (symbol: $\bot$)—a placeholder offered by UML/OCL to represent issues like exceptions. In general, the evaluation of an invariant can be $\varepsilon$ as well, however, not in this simple model.

Note that the OCL specification requires that the evaluation of an invariant returns a Boolean value. But as shown by the simple list model and its system states *Boolean* does not mean either `true` or `false`. Instead the possible values of Boolean are `true`, `false`, $\varepsilon$, and $\bot$—yielding a multiple-valued description.

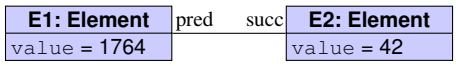
## III. The Logic Behind UML/OCL

While the previous section just briefly sketched the concepts of UML/OCL, a more formal basis is provided now. For a more detailed version refer to [26]. More precisely, this section reviews the logic principles behind UML/OCL models by means of their formalization: A type system, classes, models, and, finally, system states as well as constraints.

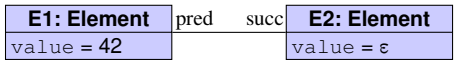
[1] Integers as well as further (even more complex) data types are also not restricted in such an obvious and expected way, details will given in Sect. III.




(a) A system state where the invariant `sorted` evaluates to `true`




(b) A system state where the invariant `sorted` evaluates to `false`




(c) A system state where the invariant `sorted` evaluates to $\bot$

Figure 2: System states for the linked list model

### A. The UML/OCL Type System

A precise type system forms the formal foundation of the logic behind UML/OCL. As UML class diagrams are usually enriched with textual constraints by means of the OCL, it is beneficial to use the type system offered by OCL. However, there is no need to introduce the complete OCL type system, since not all parts of it are considered in this work.

The type system $\mathfrak{T}$ will recursively be defined starting with the basic data types, `Boolean` (symbol: $\mathbb{B}$) and `Integer` (symbol: $\mathbb{Z}$):

$$\mathfrak{T} ::= \texttt{Boolean} \mid \texttt{Integer}.$$

While *regular* values belonging to these two types are identical with the mathematic sets, i. e., $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ and $\mathbb{Z} = \{\ldots, -1, 0, 1, 2, \ldots\}$, the OCL type system also allows for two additional *irregular* values: `invalid` (symbol: $\bot$, from OCL type `OclInvalid`) and `null` (symbol: $\varepsilon$, from OCL type `OclVoid`).[2]

**Example 1.** *The $\varepsilon$ value has a similar semantic as null pointers in classical programming languages and would, e. g., be returned by the OCL query `OrderedSet{ε}->at(1)` which asks for the first member of an ordered set (which is $\varepsilon$ in this case). In contrast, $\bot$ is used to indicate exceptions that may occur when evaluating OCL constraints, e. g., when trying to access the second element of an ordered set with just one element as in `OrderedSet{ε}->at(2)`. This is similar to an `OutOfBoundsException`, e. g., in Java. Especially the $\bot$ value only arises in this context and is, unlike the $\varepsilon$ value, not a valid assignment for class attributes.*

More precisely, $\varepsilon$ and $\bot$ are included in the *universe*, i. e., the set of all possible values of each type t—including classes and collections. In order to consider the corresponding type/universe without $\varepsilon$ and/or $\bot$, we use the notation $\mathrm{t}_{\not\varepsilon,\not\bot}$ (universe of regular values) and $\mathrm{t}_{\not\varepsilon}$ or $\mathrm{t}_{\not\bot}$, respectively.

**Example 2.** *For the type `Boolean`, `true` and `false` are the only regular values in the universe. Together with the two irregular values $\varepsilon$ and $\bot$, the universe of `Boolean` is complete and we essentially obtain what is commonly called a four-valued logic in the modeling community—even if there is no named counterpart in literature as, e. g., the Belnap logic.*

Special care has to be taken for collections: the collection `Collection(t)` may not contain the element $\bot$ (invalid),

[2] For more details about the OCL type system, interested readers are referred to the OCL specification [15, p. 211].

even though $\perp$ is an element of the (complete) universe of the type $\mathfrak{t}$. However, $\perp$ as well as $\varepsilon$ themselves are considered valid collections of any type—and are different from the empty collection.

**Example 3.** *The complete universe of* `Set(Boolean)` *is given by:* `Set{}` *(empty set),* `Set{true}`, `Set{false}`, `Set{true, false}`, `Set{ε}`, `Set{true, ε}`, `Set{false, ε}`, `Set{true, false, ε}`, $\varepsilon$, *and* $\perp$.

Another basic type are so-called enumerations. An enumeration allows for a specific and finite set of values. The (infinite) set of all possible enumerations is denoted by $\mathcal{E}$ and, for every element $e \in \mathcal{E}$, a new type is added to the type system as follows:

$$\mathfrak{T} ::= \ldots \mid e.$$

Additionally, all classes $C$ are also types—their precise concept is introduced later in this work. Nevertheless, for every class $c \in C$, the type system is extended by:

$$\mathfrak{T} ::= \ldots \mid c.$$

In addition to the already introduced types, the *collections* mentioned before are also valid types. More precisely, the OCL offers four different collection types, which are inherited from UML: `Set`, `Bag`, `OrderedSet`, and `Sequence`. They can be distinguished by the properties *ordered* and *unique*. An overview is given in Table I. Each collection is a so-called generic type, i. e., it needs a type as argument and all elements of the collection must be of this type. Formally, the type system is extended by:

$$\begin{aligned}\mathfrak{T} ::= \ldots &\mid \texttt{Set}(\mathfrak{T}) \\ &\mid \texttt{Bag}(\mathfrak{T}) \\ &\mid \texttt{OrderedSet}(\mathfrak{T}) \\ &\mid \texttt{Sequence}(\mathfrak{T}).\end{aligned}$$

Obviously, it is possible to nest collection types. This, e. g., makes it possible to have types like `Set(Bag(Integer))` or `Sequence(OrderedSet(Boolean))`.

On top of this type system, variables can be defined: A variable is a tuple $(v, \mathfrak{t})$ consisting of an identifier/name $v$ of type $\texttt{String}_{\emptyset,\notin,\perp}$, i. e., neither empty nor $\varepsilon$ nor $\perp$, and a type $\mathfrak{t}$ and is usually denoted as $v\colon \mathfrak{t}$. It can be seen as an actual instance of a type $\mathfrak{t}$ which represents a precise assignment of any value from the (complete) universe of $\mathfrak{t}$ to $v$.

**Example 4.** $p$ : `Integer` *defines the variable* $p$ *of the type* `Integer` *and, with* $p \leftarrow 17$*, an explicit assignment is given.*

The set of all variables of a type $\mathfrak{t} \in \mathfrak{T}$ is denoted by $\mathcal{V}_\mathfrak{t}$. Moreover, the short-hand $\mathcal{V}_\mathfrak{T} = \bigcup_{\mathfrak{t}\in\mathfrak{T}} \mathcal{V}_\mathfrak{t}$ is used to denote the set of all variables whose type is in $\mathfrak{T}$ (likewise for $\mathfrak{T}_\perp$). Having the notion of variables, we can now precisely define class types.

| Collection Type | Ordered | Unique |
|---|---|---|
| Set | no | yes |
| Bag | no | no |
| OrderedSet | yes | yes |
| Sequence | yes | no |

Table I: Differences of the four different collections

### B. Classes and Models

This subsection provides the basic notation for classes and models using the previously defined type system $\mathfrak{T}$ and variables $\mathfrak{V}$. Models are mainly represented as so-called class diagrams enriched with textual constraints in OCL.

**Definition 5** (Class)**.** *A* class $c = (n\colon \texttt{String}_{\emptyset,\notin,\perp}, A, O)$ *is a* 3*-tuple composed of a name* $n$ *and finite sets of* attributes $A \subset \mathcal{V}_{\mathfrak{T}_\perp}$ *(i. e., attributes may never be assigned the irregular value* $\perp$*) as well as* operations $O$.[3] *The identifiers of the attributes are unique which means that for* $a_1 = (v_1\colon \mathfrak{t}_1)$, $a_2 = (v_2\colon \mathfrak{t}_2) \in A$ *we have* $(v_1 = v_2) \Rightarrow (a_1 = a_2)$.

As class attributes may not assume the value $\perp$, we additionally consider the derived type system $\mathfrak{T}_\perp$ which contains all types from $\mathfrak{T}$ whose universe does not contain $\perp$.

Classes can be linked to other classes or to themselves using associations.

**Definition 6** (Association)**.** *An* association $r$ *(also called* relation*) is an element of the Cartesian product* $\mathcal{R} := \mathfrak{V}_C \times \mathfrak{V}_C \times (\mathbb{N} \times (\mathbb{N}_{\geq 1} \cup \{\infty\})) \times (\mathbb{N} \times (\mathbb{N}_{\geq 1} \cup \{\infty\}))$*, it is formally denoted by* $r = (role_{c_1} : c_1, role_{c_2} : c_2, (l_1, u_1), (l_2, u_2)) \in \mathcal{R}$*. The first two elements are variables with a precise class instance. The two classes, namely* $c_1$ *and* $c_2$*, are not necessarily different. In fact, if they are equal, the association is called* reflexive*. The identifiers of the variables are given by* $role_{c_i}$*. With these role names belonging to the association, OCL is able to navigate to another end. The third and fourth element are representing the multiplicities between the classes, i. e., its lower and upper bounds. More precisely, any instance of a class* $c_1$ *needs to be connected to at least* $l_1$ *as well as to at most* $u_1$ *instances of class* $c_2$ *and vice versa.*

Note that restricting the multiplicities to one interval per relation end instead of a set of intervals as originally done in the UML yields to a simpler formalization without decreasing the expressiveness as other multiplicities can be expressed in terms of additional OCL constraints.

For the sake of simplicity, we further restrict ourselves to binary associations only, even if the UML allows for $n$-ary associations with $n \geq 2$. However, also this restriction does not decrease expressiveness, since it has been shown that a model containing $n$-ary associations can be mapped into a semantically equivalent model solely composed of binary associations by adding a helping class and some invariants to the affected classes [9].

Additionally, it should be mentioned that there can be more than one relation between the same classes with identical multiplicities as long as the role names are different and, equally to classes, $\mathcal{R}$ contains an infinite number of relations with the same variables (i. e., the role names and the types are equal) but with different multiplicities. Again, there will be some meaningful restrictions for models.

**Definition 7** (Model)**.** *A* model $m = (C, R)$ *is a tuple of* classes $C \subset \mathcal{C}$ *and* relations $R \subset \mathcal{R}$ *where*

- *both elements are finite (sub)sets of the infinite universal sets,*
- *all class names are unique in* $C$*,*

[3] Operations and behavioral aspects of UML/OCL models are not explicitly covered here due to page limitations, but the concepts presented in the following can easily be extended accordingly.

- *the role names belonging to a class $c \in C$ are unique, and*
- *for each class $c \in C$ the set of all identifiers of the corresponding attributes of c and the set of all identifiers of relations where one relation end belongs to the class c are disjoint.*

**Definition 8** (Model elements)**.** *All relations of a model together with the union of all attributes (of all its classes) form the set of* model elements*.*

Further aspects such as class inheritance can be modeled using the type system as introduced so far but will be omitted in this tutorial for the sake of brevity.

### C. Objects and System States

The universe of a class type is given by the set of corresponding objects which can be defined as follows.

**Definition 9** (Class Instance/Object)**.** *An instance of a class $c = (n\colon \mathtt{String}_{\emptyset,\neq,\perp}, A, O)$ is given by a precise assignment of values to n (object name) as well as to all attributes of the class c. In the following, it will be called* object *or* object instance*. The universe of objects of a class c is written as $\Upsilon_c$.*

Based on the type system given so far, now the modeling structures can be defined.

Note that the name of an object is used to refer to it.

**Definition 10** (Links)**.** *Let $m = (C, R)$ be a given model and $r = (role_{c_1} : c_1, role_{c_2} : c_2, (l_1, u_1), (l_2, u_2)) \in R$ be a relation. Then, an instance of a relation is a so-called* link *between two object instances $v_{c_1}$ and $v_{c_2}$ (derived from the classes $c_1$ and $c_2$, respectively). More precisely, it is an element $\lambda = (role_{c_1} \leftarrow v_{c_1}, role_{c_2} \leftarrow v_{c_2})$ of the Cartesian product $\Upsilon_{c_1} \times \Upsilon_{c_2}$. The identifiers are, in general, necessary in order to differentiate between possibly various relations between the classes $c_1$ and $c_2$. A set of links is denoted by $\Lambda$.*

Based on objects and links, system states are derived.

**Definition 11** (System State)**.** *Let $m = (C, R)$ be a given model. Then, $\sigma = (\Upsilon, \Lambda)$ is called a* system state *of m, if*

- *both elements, $\Upsilon$ and $\Lambda$, are finite sets,*
- *the object names for the objects in $\Upsilon$ are unique, and*
- *all objects used anywhere in any link $\lambda \in \Lambda$ are contained in $\Upsilon$.*

*The set of all possible system states for m is denoted by $\Sigma_m$.*

Within system states, the instantiated model elements are formally denoted as follows.

**Definition 12** (Instantiated Model elements)**.** *The set containing all instances of all model elements (cf. Def. 8) in a system state $\sigma$, or instantiated model elements, is denoted by $m(\sigma)$. A single element is normally denoted by $\mu \in m(\sigma)$ in this work.*

Furthermore, the validity of an association has to be determined.

**Definition 13** (Validity of an Association in a System State)**.** *Let $\sigma = (\Upsilon, \Lambda)$ be a system state of a model $m = (C, R)$. Then, the relations in R are* valid *or* satisfied *in the system state $\sigma$, iff*

$$\forall\, r = (role_{c_1} : c_1, role_{c_2} : c_2, (l_1, u_1), (l_2, u_2)) \in \mathcal{R} :$$
$$\left(\forall\, v \in \Upsilon_{c_1} : l_1 \leq \left|\left\{(role_{c_1} \leftarrow v, role_{c_2} \leftarrow v') \in \Lambda\right\}\right| \leq u_1\right)$$
$$\wedge\ \left(\forall\, v \in \Upsilon_{c_2} : l_2 \leq \left|\left\{(role_{c_1} \leftarrow v', role_{c_2} \leftarrow v) \in \Lambda\right\}\right| \leq u_2\right)$$

### D. Class Diagrams with OCL Constraints

So far, we have considered the formalism for pure UML class diagrams. However, in Section II we have already presented a model with textual constraints using OCL. In the following, we discuss how OCL constraints can additionally be taken into account.

The *Object Constraint Language* (OCL) is a declarative language which mainly consists of

- navigation expressions to access attributes and association ends of a particular object (`self`) or related objects that can be reached using navigable association ends,
- arithmetic operations (i. e., addition, subtraction, multiplication, division, etc.),
- collection operations (i. e., intersection, union, element containment, etc.), and
- logic operations (i. e., conjunction, disjunction, negation, etc.) as well as quantifiers (universal and existential).[4]

For the remainder of this work, it is sufficient to know that OCL constraints can be annotated to classes (in terms of so-called *invariants*) in order to express constraints that shall be satisfied by any object instance of that class.[5] In order to refer to the particular object on which an OCL expression is evaluated, the keyword `self` is employed.

**Example 14.** *The running example from Fig. 1 has one invariant* `sorted` *which can evaluate to true, false, or $\perp$ (but not $\varepsilon$). Fig. 2 shows how the evaluation of the invariant changes with invalid system states and irregular values.*

The above description leads to the following definition of UML/OCL models:

**Definition 15** (UML/OCL model)**.** *A 3-tuple $m = (C, R, I)$ is called* UML/OCL model *if, and only if, $(C, R)$ is a model and $I = \coprod_{c \in C}(I_c)$ is the disjoint union of sets of invariants $I_c$ for each class $c \in C$. All these sets are finite, possibly empty sets of OCL expressions.*

System states of UML/OCL models are simply the system states of the underlying UML models extended by the validity of the invariants. Thus, the notion of validity is extended as follows:

**Definition 16** (Valid System State)**.** *Let $m = (C, R, I)$ be a UML/OCL model and $\sigma = (\Upsilon, \Lambda)$ be a system state of the model $(C, R)$. Moreover, let $I_c$ denote the set of invariants from I that are associated with a class $c \in C$.*

*Then, $\sigma$ is called a* valid system state *if, and only if,*

[4] A comprehensive overview on all OCL expressions and as keywords as well as a precise semantic definition can be obtained from [15].

[5] In general, OCL constraints can also be annotated to (class) operations in terms of so-called pre- and postconditions, but the translation of the corresponding expressions is essentially identical to invariants.

- *the relations in R are satisfied in $\sigma$ (cf. Definition 13) and*
- *for each class $c \in C$ and each object $\upsilon$ of this class that is instantiated in $\sigma$ (i.e. $\upsilon \in \Upsilon_c \cap \Upsilon$), all invariants from $I_c$ evaluate to true when* `self` *is referring to $\upsilon$.*

*The set of all valid system states of m is denoted by $\Sigma_m^{\checkmark}$.*

## IV. Verification and Validation of UML/OCL Models

Having a formal definition of UML/OCL as given in the previous section, it is now possible to formulate verification and validation tasks with respect to a model. In this section, we provide examples of such tasks and refer to approaches for the (automatic) execution of those tasks—providing a starting point for interested readers that plan to work in this area.

First, *consistency checking* is discussed. The preceding definitions included a broad variety of constraints which all have to be satisfied by a valid system state. However, the complexity of the description may lead to over-constrained (or inconsistent) models, i. e., models from which no valid system state can be derived. As this would not allow a valid instantiation of the system to be realized, such models obviously are considered erroneous. Hence, checking for an inconsistent model is considered as one of the most important structural verification tasks. More precisely, a consistency check proves whether the model is free of contradictions by determining a valid system state of the given model. If it is shown that no valid system state exists, the model has been proven to be inconsistent. Automatic methods addressing this task have been proposed, e. g., in [1–6, 10, 12, 21, 28].[6]

In case of an inconsistent model, the reasons for the inconsistency are an issue of further investigation. Finding errors in complex models proves to be a very cumbersome task, hence, pinpointing the designer to faults saves a lot of effort. Therefore, different approaches have been published which can pinpoint the designer to contradictory parts of the model [8, 22, 25].

In case of a consistent model, the chosen modeling still might not be optimal and lead to problems in later stages of design. As models provide an abstraction they should be as small as possible while at the same time complete and clear. If different persons are contributing to a model or even with only one designer if the system to be modeled is relatively complex, the designer might overlook implications of constraints. A common effect thereby is, that some constraints of the model imply another constraint to be fulfilled such that the implied constraint is, in fact, obsolete and can be removed from the model. Methods that aid the designer in this task are available at [24, 29].

In case a valid, minimal and clear model and system state have been found, this still only proves that, at one point in time, it is possible to have a valid system state. However, this does not necessarily mean that this system state does not lead to catastrophic behavior, if the operations defined in the classes are executed. Hence, while so far this work focused on static aspects, it is also beneficial to analyze sequences of (valid) system states connected by operations. One of the most obvious tasks is to check whether there are two (or more) valid succeeding system states between which an operation can be called such that the aforementioned pre- and postconditions of this operation are satisfied. This is referred to as executability of a model. Another question might be the executability of a single operation—there is no need of an operation which can be not invoked at all or which always results in an invalid system state. Reachability of a certain desired system state is also a verification issue which leads to another important aspect, namely the determination of deadlocks, i. e., system states from which no operation can be called anymore. The designer should check if the system can be trapped into such a deadlock system state from one or more starting system states. This is equivalent to defining such a state and then checking its reachability. Methods for that are available at [21, 27]. They additionally may require the proper definition of frame conditions [13, 14] as well as their consideration during reasoning [18, 23]. Also concurrency for models has been considered [19].

Overall, a huge stack of methods which tackle those verification/validation tasks has been developed over the last decades (see references from above). Many of them are also publicly available in corresponding tools such as the *UML-based Specification Environment* (USE) [7]. Here, the designer has to write an ASSL script for a dedicated task which, then, enumeratively executes all possible solutions (even for small models, this can make the runtime escalate). Hence, in newer versions of USE, relation logic based on Alloy and the constraint solver KodKod are combined. This also replaces the ASSL idea [12]. Other approaches propose to use CSP [10] or theorem provers like Isabell/HOL [2]. Solutions using SAT solvers are available at https://github.com/przigoda/model-finder.

## V. Conclusions

In this tutorial, we reviewed the basic concepts of UML/OCL—*the* description means for modeling and specifying systems in early stages of their design. Besides a brief sketch of how UML/OCL can be used, this particularly includes a formal definition of the type system and logic behind UML/OCL as well as a review on the possible validation and verification tasks that can be conducted using these descriptions. Our review explicitly included the consideration of irregular values such as `null` and `invalid` which yields descriptions in a four-valued logic.

Using this review, we hope to provide the starting point for a more detailed consideration of this often overseen feature of UML/OCL. In fact, the existing validation and verification approaches discussed above already nicely showcases how UML/OCL can be used in early stages of the design flow to check the plausibility or correctness of a model. Together with the formal basis reviewed in Section III, this could easily be extended for solutions additionally covering multiple-valued models as well. A first approach in this direction has recently been proposed in [20]. However, we see much more potential here. Particularly for a community such as gathered at ISMVL, this will provide a new and exciting "playground" for further work that may trigger new directions for research and application.

[6] Note that the number of objects even in a minimal valid system state might be very huge, thus, so-called problem bounds are normally considered which restrict the number of objects per class and, by this, making the verification task decidable. However, a closer look at problem bounds is out of scope of this tutorial and, therefore, this issue will be ignored in the following.

## References


[1] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. "UML2Alloy: A Challenging Model Transformation". In: *Model Driven Engineering Languages and Systems (MoDELS)*. 2007, pp. 436–450.

[2] Achim D. Brucker and Burkhart Wolff. "HOL-OCL: A Formal Proof Environment for UML/OCL". In: *Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*. 2008, pp. 97–100.

[3] Jordi Cabot, Robert Clarisó, and Daniel Riera. "Verification of UML/OCL Class Diagrams using Constraint Programming". In: *Model Driven Engineering, Verification, and Validation (MoDeVVa)*. 2008, pp. 73–80.

[4] Carolina Dania and Manuel Clavel. "OCL2MSFOL: A Mapping to Many-sorted First-order Logic for Efficiently Checking the Satisfiability of OCL Constraints". In: *Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS)*. 2016, pp. 65–75.

[5] Martin Gogolla, Jörn Bohling, and Mark Richters. "Validating UML and OCL models in USE by automatic snapshot generation". In: *Software and System Modeling* 4.4 (2005), pp. 386–398.

[6] Martin Gogolla, Jörn Bohling, and Mark Richters. "Validation of UML and OCL Models by Automatic Snapshot Generation". In: *Int'l Conf. on The Unified Modeling Language (UML)*. 2003, pp. 265–279.

[7] Martin Gogolla, Fabian Büttner, and Mark Richters. "USE: A UML-based specification environment for validating UML and OCL". In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 27–34.

[8] Martin Gogolla, Mirco Kuhlmann, and Lars Hamann. "Consistency, Independence and Consequences in UML and OCL Models". In: *Int'l Conf. Tests and Proofs (TAP)*. 2009, pp. 90–104.

[9] Martin Gogolla and Mark Richters. "Expressing UML Class Diagrams Properties with OCL". In: *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*. Vol. 2263. Lecture Notes in Computer Science. 2002, pp. 85–114.

[10] Carlos A. González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. "EMFtoCSP: a tool for the lightweight verification of EMF models". In: *FormSERA@ICSE*. 2012, pp. 44–50.

[11] Alexander Knapp and Stephan Merz. "Model checking and code generation for UML state machines and collaborations". In: *Workshop on Tools for System Design and Verification (FM-TOOLS)*. 2002, pp. 59–64.

[12] Mirco Kuhlmann and Martin Gogolla. "From UML and OCL to Relational Logic and Back". In: *Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*. 2012, pp. 415–431.

[13] Philipp Niemann, Frank Hilken, Martin Gogolla, and Robert Wille. "Assisted generation of frame conditions for formal models". In: *Design, Automation & Test in Europe Conference (DATE)*. 2015, pp. 309–312.

[14] Philipp Niemann, Frank Hilken, Martin Gogolla, and Robert Wille. "Extracting frame conditions from operation contracts". In: *Model Driven Engineering Languages and Systems (MoDELS)*. 2015, pp. 266–275.

[15] Object Management Group. *Object Constraint Language – Version 2.4*. 230 pp.

[16] Object Management Group. *OMG Unified Modeling Language TM (OMG UML) – Version 2.5*. 230 pp.

[17] Judith Peters, Robert Wille, and Rolf Drechsler. "Generating SystemC Implementations for Clock Constraints Specified in UML/MARTE CCSL". In: *Int'l Conf. on Engineering of Complex Computer Systems*. 2014, pp. 116–125.

[18] Nils Przigoda, Jonas Gomes Filho, Philipp Niemann, Robert Wille, and Rolf Drechsler. "Frame conditions in symbolic representations of UML/OCL models". In: *Int'l Conf. on Formal Methods and Models for System Design (MEMOCODE)*. 2016, pp. 65–70.

[19] Nils Przigoda, Christoph Hilken, Robert Wille, Jan Peleska, and Rolf Drechsler. "Checking concurrent behavior in UML/OCL models". In: *Model Driven Engineering Languages and Systems*. 2015, pp. 176–185.

[20] Nils Przigoda, Philipp Niemann, Judith Peters, Frank Hilken, Robert Wille, and Rolf Drechsler. "More than true or false: native support of irregular values in the automatic validation & verification of UML/OCL models". In: *Int'l Conf. on Formal Methods and Models for System Design (MEMOCODE)*. 2017, pp. 77–86.

[21] Nils Przigoda, Mathias Soeken, Robert Wille, and Rolf Drechsler. "Verifying the structure and behavior in UML/OCL models using satisfiability solvers". In: *IET Cyper-Phys. Syst.: Theory & Appl.* 1.1 (2016), pp. 49–59.

[22] Nils Przigoda, Robert Wille, and Rolf Drechsler. "Contradiction Analysis for Inconsistent Formal Models". In: *Int'l Symposium on Design and Diagnostics of Electronic Circuits & Systems*. 2015, pp. 171–176.

[23] Nils Przigoda, Robert Wille, and Rolf Drechsler. "Ground setting properties for an efficient translation of OCL in SMT-based model finding". In: *Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS)*. 2016, pp. 261–271.

[24] Nils Przigoda, Robert Wille, and Rolf Drechsler. "Leveraging the Analysis for Invariant Independence in Formal System Models". In: *Conference on Digital System Design (DSD)*. 2015, pp. 359–366.

[25] Nils Przigoda, Robert Wille, and Rolf Drechsler. "Verbesserung der Fehlersuche in inkonsistenten formalen Modellen (Erweiterte Zusammenfassung)". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation (MBMV)*. 2015, pp. 165–172.

[26] Nils Przigoda, Robert Wille, Judith Przigoda, and Rolf Drechsler. *Automated Validation & Verification of UML/OCL Models Using Satisfiability Solvers*. 2018. ISBN: 978-3-319-72813-1.

[27] Mathias Soeken, Robert Wille, and Rolf Drechsler. "Verifying dynamic aspects of UML models". In: *Design, Automation & Test in Europe (DATE)*. 2011, pp. 1077–1082.

[28] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. "Verifying UML/OCL models using Boolean satisfiability". In: *Design, Automation & Test in Europe*. 2010, pp. 1341–1344.

[29] Robert Wille, Mathias Soeken, and Rolf Drechsler. "Debugging of inconsistent UML/OCL models". In: *Design, Automation & Test in Europe Conference (DATE)*. 2012, pp. 1078–1083.