Encapsulation, Operator Overloading, and Error Class Mechanisms in OCL

Vincent Bertram[1], Bernhard Rumpe[1,2], and Michael von Wenckstern[1]

[1] Software Engineering, RWTH Aachen University, Aachen, Germany
[2] Fraunhofer FIT, Sankt Augustin, Germany

Abstract. Checking models for correctness or compatibility using standard formal modeling techniques such as OCL has merits in abstraction and compactness. However, it is inconvenient for developers, since there are no standard mechanisms how to handle large and complex OCL constraints. Therefore, this paper presents an approach how to split complex OCL constraints into multiple ones by defining helper functions and pack these into an OCL/P library with encapsulation mechanisms. Another drawback of using complex OCL constraints at present is the lack of descriptive and user-friendly error messages. Hence, this paper introduces an OCL extension that allows specifying error classes by synthesizing witnesses pointing directly to constraint violations. All approaches are shown on Component & Connector model examples, where OCL/P is used on the meta-level to verify backwards compatibility of interfaces.

1 Introduction

Static software verification is a software engineering discipline, analyzing software against a given specification without running any line of code using formal methods. These checks can identify modeling errors, potential problems, variant and version incompatibilities within a single model or even among different ones.

OCL [19] is a well-known, abstract and compact language to formalize verification properties of models, e.g. consistency or well-formedness. Interface as well as behavioral compatibility and even similarity rules [23,1,20] can be described. This leads to a large set of OCL constraints. However, OCL does not give answers to the following questions, which are needed to handle many OCL constraints:

1. How to logically group OCL constraints?
2. How to split up complex constraints easily into multiple smaller ones?
3. How to use OCL operators for self-defined model structures?
4. How to produce meaningful user error messages?

The contribution of this paper is to answer these four questions, and to make a first step towards the use of OCL in specifying complex constraints in even different aspects, e.g. the verification domain. All concepts presented in this paper will be explained using constraints for interface backward compatibility between two Component & Connector (C&C) models.
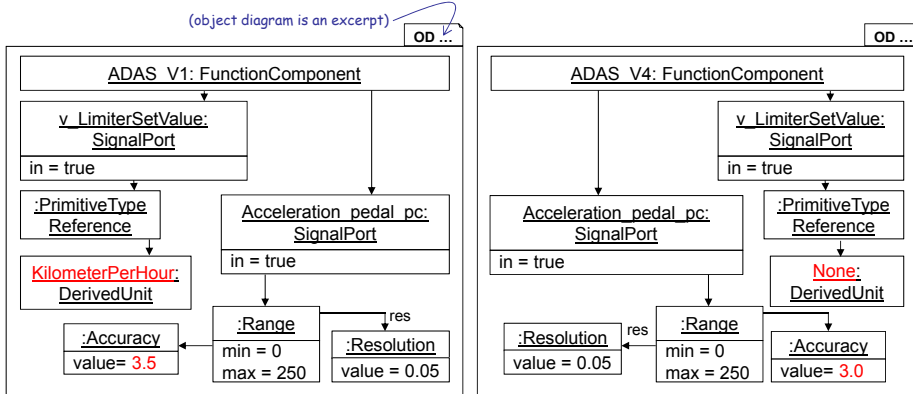
Fig. 1: Object Diagram instantiation of *ADAS_V1* and *ADAS_V4*

The paper is outlined as follows: Section 2 introduces the running example and gives an informal introduction into the used C&C models and their interface compatibility rules. Section 3 presents a formal notation of C&C models, their meta-model, a semi-formal definition of interface-compatibility, as well as a short introduction into OCL. Our first contribution in Section 4, presents solutions for the first three questions by adding a library concept to OCL and by making OCL modeling more convenient with operator overloading. Then Section 5 shows an extension for generating counterexample witnesses based on error classes which are "easy to understand" for the engineer, our second contribution. A brief evaluation and discussion is provided in Section 6. At last, Section 7 compares our approaches with other researches, generating counterexamples for OCL constraint violations.

2 Running Example

Due to the highly competitive automotive market, automobile manufacturers update their vehicles continuously with new features. Since a special single feature does not affect every software part, individual components are updated to successively replace old component versions with new ones.

Figure 1 shows such a scenario where an Advanced Driver Assisted Systems (ADAS) component version (*ADAS_V1*) should be replaced by a more capable version (*ADAS_V4*). The representation as Object Diagram (OD) is done using UML/P [22]. Since such a change can cause incompatibilities, the automotive industry is constantly stating the structural (and/or behavioral) backward compatibility. In this case one has to prove that *ADAS_V4* is backward compatible, at least structural backward compatible to *ADAS_V1*. At this stage static model verification comes into the game.

Hence this paper shows engineering strategies how to formalize complex constraints, e.g. these compatibility constraints, in OCL. Additionally, this paper

shows a mechanism which can be used to generate user friendly and meaningful error messages for violated constraints. This approach is used to provide intuitive feedback for the ones[1] presented in Figure 1 (see red marked parts).

ADAS_V4 is incompatible to *ADAS_V1*, because *ADAS_V1* processes speed limiter values assigned with the unit km/h and *ADAS_V4* processes the values arriving at the port `v_LimiterSetValue` dimensionless as direct bus signals are not assigned to a specific unit. Also, the needed accuracy of the port `Acceleration_pedal_pc` of *ADAS_V4* is less than the one in *ADAS_V1* meaning that *ADAS_V4* cannot process sensor data having a noise of 3.5.

The two ODs [12,4] in Figure 1 represent concrete C&C instances which will in practice be flashed into the automotive software. In our context, a component is a unit executing computations (such as control commands to avoid accidents) and/or storing data (e.g. previous sensor data for interpolation reasons) as well as describing the information flow between components via typed ports [11].

The running example presented in this work is a simplified excerpt of a model, used in the research project SPES_XT[2] and does not represent a real world model, but provides reliable information without being overloaded with irrelevant information. A more detailed version can be found in [1].

3 Preliminaries

This section briefly describes some basics about C&C models, respectively their meta-model including data types, interface compatibility. It also provides some information on OCL.

3.1 Component and Connector Models

C&C models describe components, their interaction and how they are hierarchically composed. Maoz et al. [15] define component models [17] as given in Definition 1, which reflects their essence as formalized by ADLs AADL [6], ACME [7], and MontiArc [11], or used in (commercial) tools Modelica [18] and Simulink [16].

Definition 1 (Component and Connector model [15]). A C&C model is a structure $cncm = \langle Cmps, Ports, Cons, subs, ports, type \rangle$ where

- *Cmps* is a set of named components, $cmp \in Cmps$ has a set of ports $ports(cmp) \subseteq Ports$ and a (possibly empty) set of immediate subcomponents $subs(cmp) \subset Cmps$,
- *Ports* is a disjoint union of input and output ports where each port $p \in Ports$ has a name, a type $type(p) \in Types$, and belongs to exactly one component $p \in ports(cmp)$,

---

[1] Section 6 explains the incompatibilities of both ADAS versions in more detail.
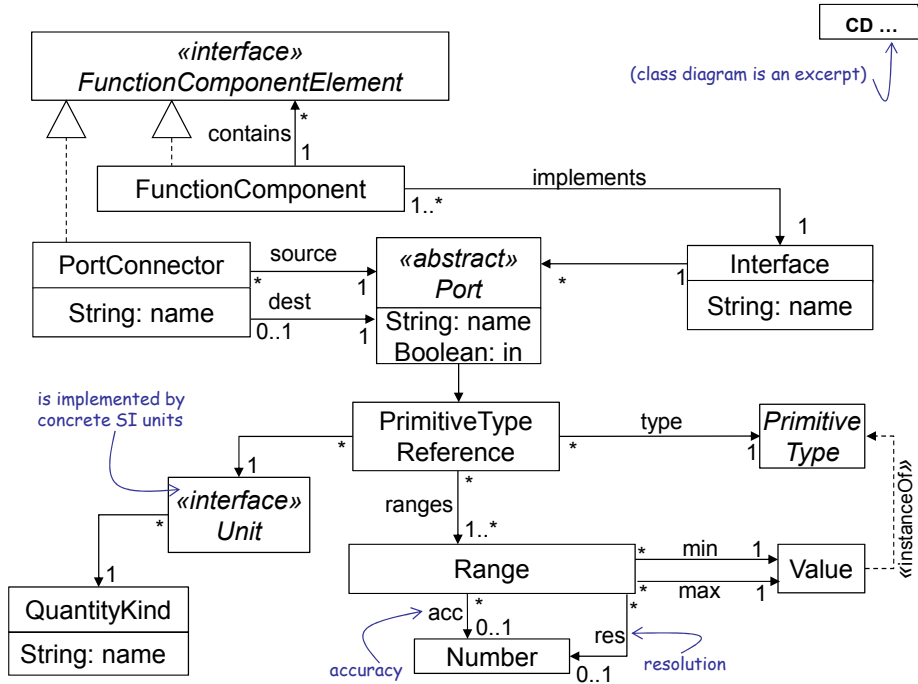[2] `http://spes2020.informatik.tu-muenchen.de/spes_xt-home.html`

19

Fig. 2: C&C meta-model, UML/P excerpt from [1]

- *Cons* is a set of directed connectors $con \in Cons$, each of which connects two ports $con.src, con.tgt \in Ports$ of the same type, which belong to two sibling components or to a parent component and one of its immediate subcomponents, and
- *Types* is a finite set of type names.

A C&C model is valid iff no component is its own (transitive) subcomponent and has at most one direct parent and subcomponents are connected legally with respect to in-/output direction as well as their transmitted data types (see [15] and [21] for complete definitions).

3.2 Component and Connector Meta-Model

Based on Definition 1 and an investigation of all common C&C modeling languages, a complete C&C meta-model has been derived in [1]. This subsection recaps some of this C&C meta-model elements that are necessary to understand the OCL constraints used in this paper. Related names used in Definition 1 are written in brackets after the meta-model elements.

The meta-model in Figure 2 has a FunctionComponent (*Cmps*) containing a set of subcomponents (realized by the contains association) as well as a set of port connectors (*Cons*). Each FunctionComponent implements at least

one `Interface`, having a set of `Ports` (*Ports*). The component interface can also have (but skipped for simplicity) extra-functional properties such as latency, memory usage, etc. A `Port` can have either `PrimiteTypeReference`s or `CompositeTypeReference`s, representing *struct* or *array* objects (skipped in this figure). The primitive type reference holds all important information on dataflows between ports with a primitive type such as Boolean, Enumeration or Number. The `QuantityKind` represents the physical dimension of the SI basic units. Each `Unit` interface has exactly one `QuantityKind`. `Range`s represent all values a port can send or receive, the values are between `min` and `max` with a step-size given in `res.value`; `Accuracy` is used to express the maximum difference between the acutal value and a given sensor output.

3.3   Interface Compatibility

A component interface (see Figure 2) contains all structural information on how a component communicates with its environment. If a component, e.g. newer version, can replace another one, e.g. older component version, based on their component's interface information, the newer component is interface compatible to the older one, also called backward compatible. In general component A is interface compatible to B iff:

- component A has at least (it may have more) the same input and output port names as component B,
- A's input ports accept the same or more input values than B's input ports, and
- A's output ports produce the same or fewer output values than B's output ports.

It is not complicated to define interface compatibility, but still, one can face a lot of small constraints: (1) primitive data type compatibility (e.g. when are enumerations compatible), (2) unit compatibility such as *km/h* and *m/s*, (3) ranges compatibility considering several ranges each of which may have different minimum, maximum, accuracy as well as resolution values.

3.4   Object Constraint Language

According to Rumpe [22] OCL is a property-oriented modeling language defining queries, model constraints such as invariants as well as pre- and postconditions. This paper uses Rumpe's OCL/Programmable (OCL/P), which is adjusted to Java. Instead of using OCL 2.4's 4-level Boolean [19], OCL/P is using a 2-level binary logic making it more accessible to developers.

Here, we recall shortly the OCL termini introduced by Rumpe (it is incomplete; for a complete list see [22]):

Constraint: Is a Boolean statement about a system.
Context: Is the context in which a constraint is embedded into; e.g. names of classes, attributes and/or properties in Class Diagrams (CDs).
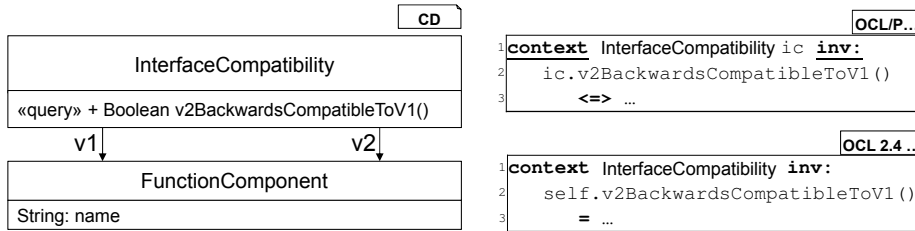
Fig. 3: General schema for interface compatibility constraint.

Invariant: Describes a property that must hold in a system at each point of time.

Query: Is a method whose call does not affect the system's state.

Figure 3 shows how OCL constraints for interface compatibility would look like. The OCL/P *constraint* has a *context* and it describes an *invariant*, meaning the constraint must be satisfied for all `InterfaceCompatiblity` class instances. Line 2 calls on every `InterfaceCompatiblity` object the query method `v2BackwardsCompatibleToV1()`. Compared to the running example in Section 2 *v1* is *ADAS_V1* and *v2* is *ADAS_V4*. It is important that this method call is free of any side-effects and does not modify any system's state. The constraint's last line uses the `<=>`, *if-and-only-if* operator, to define interface compatibility for OCL/P. The equals operator in OCL 2.4 is similar to OCL/P's `<=>`.

4 OCL Extensions

This section presents two syntactic sugars to OCL making it easier to define complex constraints: (1) Defining an OCL library with expressions in a function-like way, and (2) Operator overloading to make OCL constraints more intuitive.

4.1 Definition of Library Expressions

In OCL the *definition* expression defines new attributes and query operations to existing models, which can be used in other constraints. This would allow us to define a query method, e.g. `IsBackwardsCompatibleTo (FunctionComponent v1)`, to the CD model `FunctionComponent`. This approach makes the modeling of an extra class `InterfaceCompatibility` in the CD redundant, which is an advantage.

In order to not create one large interface compatibility constraint, this constraint depends on other compatibility constraints such as data type, range, unit compatibility. But this would result in polluting CD with unnecessary, and probably not reusable, query functions just to make one definition good readable. Hence, we present a method for how to define an OCL library comfortably with public (can be called from outside the library) and private (can only be used inside this library) functions.

```
                                                                    OCL/P...
 1 library InterfaceCompatiblity is:
 2   + def boolean v2BackwardsCompatibleToV1          (code is an excerpt)
 3           (FunctionComponent v1, FunctionComponent v2) is:
 4     result =
 5        (forall Port ports1 in v1.implements.ports,
 6            Port ports2 in v2.implements.ports:
 7                  dataTypeCompatible(ports1, ports2))
 8        && …
 9   - def boolean dataTypeCompatible(Port p1, Port p2) is:
10     result = …
```

public method
private method

```
                                                                    OCL 2.4 ...
11 package InterfaceCompatiblity
12   context FunctionComponent
13     def: v2BackwardsCompatibleToV1(v2: FunctionComponent)
14                                                   :Boolean =
15       self.implements.ports->forAll(ports1: Port |
16         v2.implements.ports->forall(ports2: Port |
17               dataTypeCompatible(ports1, ports2))
18       && …
19   context Port
20     def: dataTypeCompatible(p2: Port) :Boolean = …
21 endpackage
```

Fig. 4: Example for an OCL library with public and private query functions

There exists no equivalent translation in OCL 2.4
("Definition constraint must be attached to a Classifier" [19])

```
                                                                    OCL/P
1 def boolean v2BackwardsCompatibleToV1
2     (FunctionComponent v1, FunctionComponent v2) is:
```

Semantically
equivalent

```
                                                                    OCL/P
1 context FunctionComponent v1:
2   def boolean v2BackwardsCompatibleToV1(FunctionComponent v2) is:
```

```
                                                                    OCL 2.4
1 context FunctionComponent
2   def: v2BackwardsCompatibleToV1(v2: FunctionComponent) :Boolean =
```

Fig. 5: Example for OCL def operator in non-member syntax

Figure 4 shows an excerpt of how to define a library using either OCL/P or OCL 2.4. The `result =` keyword in line 4 has the same semantic as line 3 `ic.v2BackwardsCompatible <=>` in Figure 3. The - (private keyword) in line

23

Table 1: All OCL/P operators grouped by their priority

9 is not necessary, because all query functions inside the library are private by default.

Similar to C++'s mechanism that define new functions and operators as member and non-member, new operations can also be defined without an explicit given context for syntactical convenience. Exemplary, the top part in Figure 5 shows the convenience definition from Line 2-3 in Figure 4, which is semantically equivalent to the definition inside the classifier `FunctionComponent` in the bottom part in Figure 5 extending the class `FunctionComponent` with an extra query function `v2BackwardsCompatibleToV1`.

4.2 Operator Overloading

For better readability OCL/P also supports prefix and infix operator overloading; whereas it is *not* possible to change the operator precedence nor to define a new operator symbol. It is also forbidden to overload operators with predefined semantics, e.g. `Number + Number`. Table 1 lists all available OCL operators grouped by their priority. Operators in the same group have the same precedence and are bounded from left to right.

Figure 6 shows an example of how to overload infix operators. Line 1 overloads the equivalence operator ∼: Two units are equivalence iff they have the same unit kind, e.g. `Length`, `Velocity`, and so on.

Lines 3-7 specify whether a `Number` belongs to a specific `Range`; the `Range`'s optional `res`olution is only considered if it is specified. The expression ∼`r.res` becomes true if the `Range r` has no optional association `res` to an instance of the class `Resolution`; if this is the case, the left part of the *or* (||) expression is true and due to OCL/P's short-circuit evaluation strategy the right part will not be evaluated. The number 4.0 belongs to the range [3.0; 5.0], but 4.0 is *not* part of the range [3.0; 5.0] with a resolution 0.6 containing only the values {3.0; 3.6; 4.2; 4.8}.

Line 8 shows that operator overloading is sensitive to its type arguments. OCL resolves the overloaded functions or operators by matching first the ones

```
                                                                    ┌─────────┐
                                                                    │ OCL/P...│
 1 def boolean infix (Unit u1) ~ (Unit u2) is:                      └─────────┘
 2   result = u1.quantityKind == u2.quantityKind         (code is an excerpt)
 3 def boolean infix (Number v) in (Range r) is:
 4   result =
 5     v >= r.min &&          (Range r has no optional association res to Resolution)
 6     v <= r.max &&
 7     (~r.res || (v - r.min) % r.res == 0)
 8 def boolean infix (Number v) in (List<Range> ranges) is:
 9   result = exists Range r in ranges: v in r
10 def boolean typeReferenceCompatible(PrimitiveTypeReference tR1,
11                                     PrimitiveTypeReference tR2) is:
12   let
13     PrimitiveTypeReference tR1c = tR1.convert(tR2.unit)
14   in                          (converts e.g. tR1 from 1 m in 100 cm)
15     result =
16       tR1.unit ~ tR2.unit &&
17       forall Number v in tR1c.ranges:
18         v in tR2.ranges &&
19           …
```

Fig. 6: Example for OCL operator overloading

changed operator '~' to '=', because OCL 2.4 can only overlaod this operators:
'+', '-', '*', '/', '=', '<>', '<', '>', '<=', '>=' [19]

```
                                                                    ┌─────────┐
                                                                    │ OCL 2.4 ...│
 1 context Unit                                                     └─────────┘
 2   def: _'='(u2: Unit) :Boolean = self.quantityKind == u2.quantityKind
 3 context Number
 4   def: isInRange(r: Range) :Boolean =
 5     self >= r.min &&
 6     self <= r.max &&
 7     (r.res->notEmpty() || (self - r.min) % r.res == 0)
 8   def: isInOneRange(ranges: Sequence(Range)) :Boolean =
 9     ranges->exists(r: Range | self.isInRange(r))
10 context PrimitiveTypeReference
11   def: typeReferenceCompatible(tR2: PrimitiveTypeReference) :Boolean =
12     let
13       tR1c: PrimitiveTypeReference = tR1.convert(tR2.unit)
14     in
16       tR1.unit = tR2.unit &&
17       Number.allInstances()->forall(v: Number |
18         v.isInOneRange(tR1c.ranges) implies
19           v.isInOneRange(tR2.ranges) &&
20             …
```

Fig. 7: Equivalent OCL 2.4 code for Figure 6

with the instances' exact types and then it tries to match the ones of the instances' supertypes.

The function defined in lines 10-19 defines the compatibility of primitive type references. Line 13 introduces a help variable to simplify the latter-on specification using the *let* construct. The expression `tR1.convert(tR2.unit)` will not

25

be evaluated in line 13; it will be lazy evaluated when the variable `tR1c` is used the first time in line 17 and only if line 16 becomes true. The overloaded operators defined in lines 1 and 8 are called in lines 16, 17 and 18. The syntax of overloading a prefix operator is shown in Figure 8.

Figure 7 shows the OCL 2.4 equivalent. Due to the official OCL 2.4 documentation it is not possible to overload the ~ operator, therefore it has been changed to the = operator. Also OCL 2.4 does not allow to define operators in a non-member syntax, therefore intuitive operator definitions such as `infix (Unit u1) = (Unit u2)` must be defined as a member function of the first operand `Unit::_'='(u2: Unit)` as shown in the first two lines.

The expression `tR1.unit = tR2.unit` in line 16 maps OCL 2.4 to the function `tR1.unit._'='(tR2.unit)`. Since OCL 2.4 does not has a `in` operator as OCL/P, this operator cannot be overloaded, and therefore, Figure 7 defines the two functions `isInRange` and `isInOneRange` in lines 4 and 8, which are used in line 17 till 19.

```OCL/P
1 def boolean prefix ~ (Association a) is:
2   result = a.size > 0
```

Fig. 8: Example OCL prefix

Since OCL has no C-like postfix operators, such as `++`[3], there is no syntax for overloading postfix operators. The operator `++` is not supported, since it modifies the data structure; and this is not allowed in OCL.

5 OCL Error Classes for Intuitive Feedback

This section shows a mechanism how to generate user-friendly error messages if OCL constraints fail. These error messages can be domain-specific and hence can give users all the information needed to trace down existing errors.

The previous section has shown how compatibility constraints for instantiations of `PrimitiveTypeReference` can be defined. One drawback of this OCL definition is its restriction to a Boolean result for the user. The answer *satisfied* (*ADAS_V4* is compatible to *ADAS_V1*) or *non-satisfied* makes it hard to understand where exactly the constraints failed in case of a negative answer.

A definition of error classes overcomes this drawback by providing easy to understand witness instantiations of an OCL error class. The top-left part in Figure 9 shows a CD of *UnitWitness*, including the `portName` of the two compared components that contain the two different units. The *query* stereotype in this class facilitates all its methods to be side-effect free [22] which then can be

---

[3] C-like pre- and postfix operators; and **no** method pre and post conditions.
For more details see 3.4.3 in [22].

```
                                                    CD                                                                                        OCL/P...
                                                                1 context UnitWitness uw inv:
                «query»                                         2     let
              UnitWitness                                       3         SignalPort p1 = uw.getUnit1().
 - readonly String portName                                     4           primitiveTypeReference.signalPort;
 - readonly Unit unit1                                          5         SignalPort p2 = uw.getUnit2().
 - readonly Unit unit2                                          6           primitiveTypeReference.signalPort;
 + String getPortName()                                         7     in
 + Unit getUnit1()                                              8         p1.interface.name != p2.interface.name &&
 + Unit getUnit2()                                              9         uw.getPortName() == p1.name &&
                                                               10         uw.getPortName() == p2.name &&
   OCL/P can navigate against                                  11         !(uw.getUnit1() ~ uw.getUnit2())
   navigation direction

   ports belong to different function
   components (different interface names)                         CD + OCL/P context is replaced by
   ports have the same name                                             OCL/P counterexample
   units of the ports are not compatible                                                                               OCL/P...
 1 counterexample String portName, Unit unit1, Unit unit2 inv UnitWitness:
 2     let
 3         SignalPort p1 = unit1.primitiveTypeReference.signalPort;
 4         SignalPort p2 = unit2.primitiveTypeReference.signalPort;
 5     in
 6         p1.interface.name != p2.interface.name &&
 7         portName == p1.name && portName == p2.name &&
 8         !(unit1 ~ unit2)
```
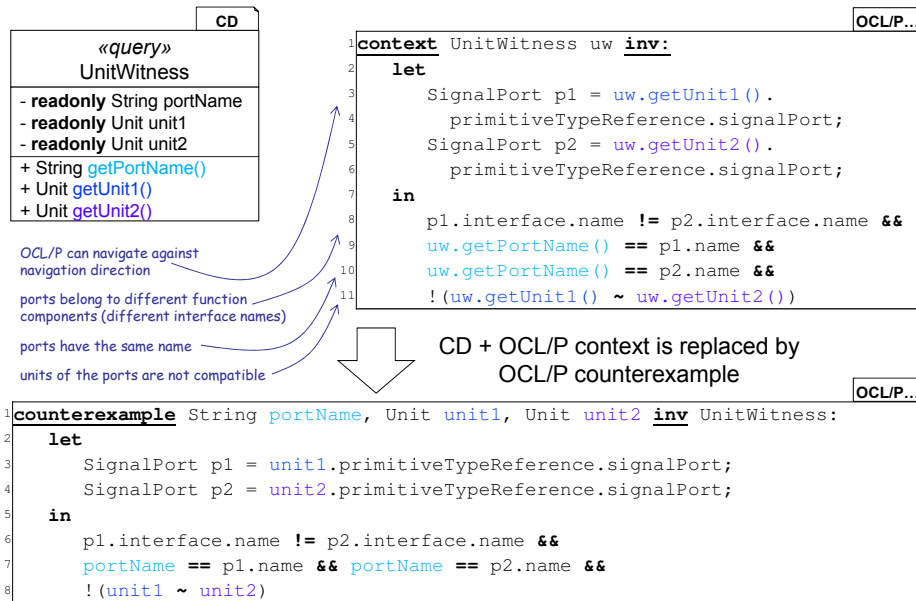
Fig. 9: Defining error classes producing counterexamples

used in the top-right OCL context expression. Since this OCL code specifies only valid *UnitWitness* elements, it is allowed to navigate against CD's navigation direction [22]; this is used to receive the ports to which the two units (`getUnit1` and `getUnit2`) of the `UnitWitness` object `uw` belong to. Line 8 in Figure 9 specifies both ports as members of two different component *interface*s and lines 9-10 constraint that the two ports have the same name as the one given in the witness `cw`. Line 11 specifies the real witness condition, meaning both units are not compatible to each other.

A valid `UnitWitness` instantiation related to the OD in Figure 1 would have the attribute values:

 – `portName="v_LimiterSetValue"`,
 – `unit1="KilometerPerHour"`, and
 – `unit2="None"`.

This witness instance can be used in templates for generating user friendly text messages. In order to maintain only one kind of artifact (in this case OCL), OCL has been extended by the `counterexample` keyword as demonstrated in the bottom part of Figure 9. This code is semantically equivalent to the top part; but can be read as a function specification that returns three values: a port name and the two incompatible units. Apart from this example, each `counterexample` OCL code can have **n** possible *return* values, but it must include an error class name, specified after the `inv` keyword.
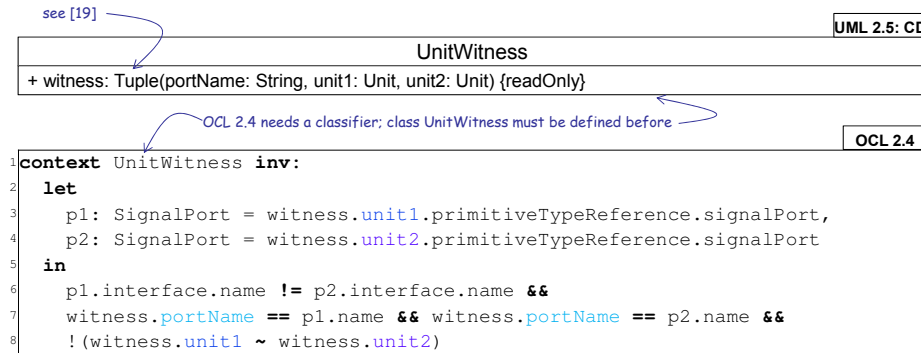
UML 2.5: CD

UnitWitness

+ witness: Tuple(portName: String, unit1: Unit, unit2: Unit) {readOnly}

OCL 2.4 needs a classifier; class UnitWitness must be defined before

OCL 2.4

```
1 context UnitWitness inv:
2   let
3     p1: SignalPort = witness.unit1.primitiveTypeReference.signalPort,
4     p2: SignalPort = witness.unit2.primitiveTypeReference.signalPort
5   in
6     p1.interface.name != p2.interface.name &&
7     witness.portName == p1.name && witness.portName == p2.name &&
8     !(witness.unit1 ~ witness.unit2)
```

Fig. 10: Equivalent OCL 2.4 code for Figure 9

Figure 10 shows the OCL 2.4 counterpart of Figure 9. In OCL 2.4 it is not possible to omit the CD definition, since new witness attributes can only be defined within a classifier. In this example, it is also not possible to just define the class `UnitWitness` and add the attribute `witness`, using OCL 2.4 `def:` expression (see 7.4.4 [19]). In that case the concrete algorithm for how to define this `Tuple` (see 7.5.15 [19]) must explicitly be given. As a result, we constrained the properties of the `witness` attribute with OCL 2.4's invariant expression.

Error classes can be prioritized in order to avoid the generation of many witness instantiations which are all implied by only one major error. TO give an example from the SPES_XT project, unit compatibility is higher prioritized than range compatibility (all range incompatibility witnesses must have compatible units), as making a unit dimensionless, often results in changing its value ranges and its accuracies.

In this paper, and in most of our OCL examples, this mechanism was used in order to find errors; but it can also be used to produce a positive user model for satisfied constraint checks. In the compatibility example it can return all matched ports and their values in its ranges, because backward compatibility only requires the match of a subset of ports and of values - whether it is an input or output port. Thus, the output can be used for documentation purposes.

6 Evaluation and Discussion

OCL constraints have been used to define *ContextCondtions* in MontiArc, an architectural description language for C&C models developed at our chair. Table 2 shows an excerpt of these context conditions. See [10] for a full list and expressive for examples.

For each formalized *ContextCondition* also a `counterexample` class was given to produce meaningful user feedback. A more complex example, has been realized, forbids component type clones using OCL constraints to avoid inconsistencies later. There, we forced that it is not allowed for two C&C type definitions

B1	All names of model elements within a component namespace have to be unique.
B1	Top-level component type definitions do not have instance names.
CO1	Connectors may not pierce through component interfaces.
R1	Each outgoing port of a component type definition is used at most once as target of a connector.
R2	Each incoming port of a subcomponent is used at most once as target of a connector.
R8	The target port in a connection has to be compatible to the source port, i.e., the type of the target port is identical or a supertype of the source port type.
R11 ...	Inheritance cycles of component types are forbidden.

Table 2: Context conditions

to have the same structural interface as well as the same internal structure. This was later extended to match also structural similar components, e.g. `Gain(2)` block, multiplying the input with two, and a `Sum` block connecting with the same source, with two input ports.

The interface compatibility check had similar constraints as the ones detecting clones. Therefore we outsourced the type reference constraints, primitive as well as complex ones, to an OCL library used by both checks.

Introducing the OCL library concept with private and public constraints, made it easier for us to organize this amount of constraints. Operator overloading would not be necessary, but it made it easier to read OCL constraints, especially for non-symmetric infix operators as the `in` one. The most important concept was the introduction of error classes, because otherwise we were not able to use OCL for MontiArc *ContextConditions*, since the user needs to know which C&C element causes a constraint to fail.

To create even better user feedback, we prioritized error classes as it is shown in Figure 11. There, every witness is an instances of exactly one error class. The error prioritization is done by defining disjunct error classes, which ensures that solely one error is causing the incompatibility for exactly one port of the C&C model instead of many different errors that are implied by the one main error (e.g. unit compatibility is higher prioritized than range compatibility, because making a unit dimensionless results in changing its value ranges and its accuracies).

Figure 11 is the result which automotive engineers get when they ask for our tool in cases where *ADAS_V4* can replace *ADAS_V1* from the running example in Section 2. This figure shows further that error class witnesses can be transformed to other models presenting constraint violations in an even more user-friendly way. This example illustrated C&C incompatibilities in *Simulink*, since this is the de facto standard modeling tool used by automotive engineers. The
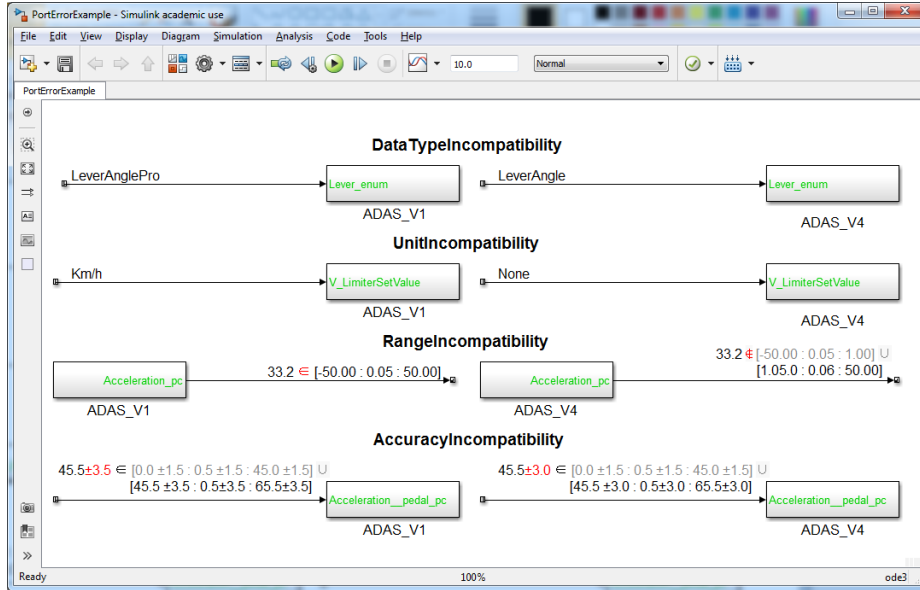
Fig. 11: Structural compatibility errors are illustrated as *Simulink* model

top left component *ADAS_V1* shows the ADAS in version 1, which is checked for compatibility with *ADAS_V4* in version 4. Both form a witness for the incompatibility between the two versions, since the enumeration types `LeverAnglePro` and `LeverAngle` represent different data types. Another incompatibility is identified by the second witness *UnitIncompatibility*. It shows that *V_LimiterSetValue*'s units are *Km/h* for *ADAS_V1* but *None* for *ADAS_V4*, which are not compatible. Besides unit incompatibilities also range incompatibilities, as shown by witness *RangeIncompatibility*, and can be detected too. Furthermore, accuracy incompatibilities, e.g. the accuracy of the `Acceleration_pedal_pc`'s value *45.5* is *±3.5* in *ADAS_V1* and is only *±3.0* in *ADAS_V4*, shown by the witness *AccuracyIncompatibility* in Figure 11, can be identified.

7 Related Work and Conclusion

OCL is often used to define static verification criteria for UML models, e.g. CDs [14], behavior diagrams [9], or even source code [24]; however, plain OCL code as shown in [24] is hard to read; OCL/P with its infix support and *Java* based notation, as it is more familiar to most programmers, tackles this issue.

Similar to the approach presented in this paper, the checking of compatibility requirements, OCL is used for defining requirement specifications to verify embedded architectures [3]. Specifying TLM 2.0 communication rules for C&C models in OCL allows validating communication compatibility between com-

ponents based on defined communication protocols [13]. OCL is even used for consistency constraint definitions between *AUTOSAR* and *SysML* models [8].

Usage of OCL in templates for *SysML* requirement specifications in embedded software is done by [5] to verify user inputs during requirement specification. Their verification tool even pinpoints, in some cases, to error columns in tables to guide the user and thus avoid inaccurate or wrong instances of requirement specifications. However they do not support guidance for complex OCL constraints and error prioritization.

Today, witnesses are mostly generated to validate the result of constraint checks in order to avoid false positives [2]. None of these approaches allow the specification of how user-friendly error messages should look like, based on witnesses relation to their constraint violations. During our component interface compatibility modeling process for C&C models in the SPES_XT context, where 63 OCL/P constraints were needed to fully specify component interface compatibility, the four key questions for using complex OCL specification models have been identified and can finally be answered:

1. How to logically group OCL constraints?
   ⇒ Create OCL libraries to structure the code and use their encapsulation mechanisms with private and public constraint definition to hide unnecessary details for developers who want to use only the main constraints (see Section 4.1).
2. How to split up complex constraints easily into multiple smaller ones?
   ⇒ Create smaller OCL helper constraints by using the easy to use OCL `def` operator; there is no further need to create `query` classes first. Due to the available non-member `def` syntax, splitting large OCL constraints, it is now very similar to splitting large Java or C function into several smaller ones (see Section 4.1).
3. How to use OCL operators for self-defined model structures?
   ⇒ Thanks to operator overloading, a well-known principle in many programming languages, self-defined models, e.g. *complex numbers* defined as a CD, can be accessed as intuitive (e.g. by using the `+` operator) as the OCL basic types such as integer numbers (see Section 4.2).
4. How to produce meaningful user error messages?
   ⇒ In Section 5 this paper presented a methodology on how to specify and prioritize error classes for users to generate intuitive user feedback.

This paper presented solutions to these above questions, by applying OCL as a user-friendly and modular specification language for formal problems, e.g. static software verification. Section 6 even demonstrated some use-cases where we successfully used OCL with the new introduced concepts as specification language.

References

1. Bertram, V., Manhart, P., Plotnikov, D., Rumpe, B., Schulze, C., von Wenckstern, M.: Infrastructure to Use OCL for Runtime Structural Compatibility Checks of Simulink Models. In: Modellierung (2016)

2. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: FSE (2015)
3. Blouin, D., Senn, E., Turki, S.: Defining an annex language to the architecture analysis and design language for requirements engineering activities support. In: MoDRE (2011)
4. Cengarle, M.V., Grönniger, H., Rumpe, B.: System Model Semantics of Class Diagrams. Tech. rep., TU Braunschweig (2008)
5. Chang, C.H., Lu, C.W., Kao, K.F., Chu, W.C., Yang, C.T., Hsueh, N.L., Hsiung, P.A., Koong, C.S.: A SysML-Based Requirement Supporting Tool for Embedded Software. In: SSIRI-C (2011)
6. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley (2012)
7. Garlan, D., Monroe, R.T., Wile, D.: Acme: An architecture description interchange language. In: CASCON. pp. 169–183 (1997)
8. Giese, H., Hildebrandt, S., Neumann, S.: Graph Transformations and Model-Driven Engineering, chap. Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. Springer (2010)
9. Gogolla, M., Hamann, L., Hilken, F., Sedlmeier, M., Nguyen, Q.D.: Behavior Modeling with Interaction Diagrams in a UML and OCL Tool. In: BM-FA (2014)
10. Haber, A.: MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems. Aachener Informatik-Berichte, Software Engineering, Shaker Verlag (2016)
11. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Tech. rep., RWTH Aachen (2012)
12. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? IEEE Computer (2004)
13. Jain, V., Kumar, A., Panda, P.R.: A SysML Profile for Development and Early Validation of TLM 2.0 Models. In: ECMFA (2011)
14. Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back. In: MODELS (2012)
15. Maoz, S., Ringert, J.O., Rumpe, B.: Synthesis of component and connector models from crosscutting structural views. In: FSE. pp. 444–454. ACM (2013)
16. Mathworks: Simulink User's Guide. Tech. rep. (2015)
17. Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering (2000)
18. Modelica Association: Modelica - A Unified Object-Oriented Language for Systems Modeling. Tech. rep. (2012)
19. OMG: Object Constraint Language, Version 2.4. Tech. rep. (2014)
20. Richenhagen, J., Rumpe, B., Schloßer, A., Schulze, C., Thissen, K., von Wenckstern, M.: Test-Driven Semantic Similarity Analysis for Software Product Line Extraction. In: SPLC (2016)
21. Ringert, J.O.: Analysis and Synthesis of Interactive Component and Connector Systems. Shaker Verlag (2014)
22. Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer International (July 2016)
23. Rumpe, B., Schulze, C., von Wenckstern, M., Ringert, J.O., Manhart, P.: Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In: SPLC (2015)
24. Seifert, M., Samlaus, R.: Static Source Code Analysis using OCL. In: OCL (2008)