Encapsulation, Operator Overloading, and Error Class Mechanisms in OCL

OCL @ MODELS 2016

Vincent Bertram, Bernhard Rumpe,
Michael von Wenckstern

Software Engineering
RWTH Aachen
http://www.se-rwth.de/

Outline

Motivation - General

- Due to the highly competitive automotive market, manufacturers update their vehicles continuously with new features.
- A special single feature does not necessary affect every software part, therefore individual components are updated to successively replace old component versions with new ones.
- A feature change can cause incompatibilities, the automotive industry is constantly stating the structural (and/or behavioral) backward compatibility.

Motivation – Static software verification

- Static software verification is a software engineering discipline, analyzing software against a given specification without running any line of code by using formal methods.
- Checking models for correctness or compatibility using standard formal modeling techniques (such as OCL) has merits in abstraction and compactness.
- It is inconvenient for developers, since there are no standard mechanisms how to handle large and complex OCL constraints.

Motivation - Questions

- The contribution is to answer the following questions:

(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 error messages to the user?

Outline

Interface compatibility

- Interface compatibility (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
  - A's output ports produce the same or fewer output values than B's output ports

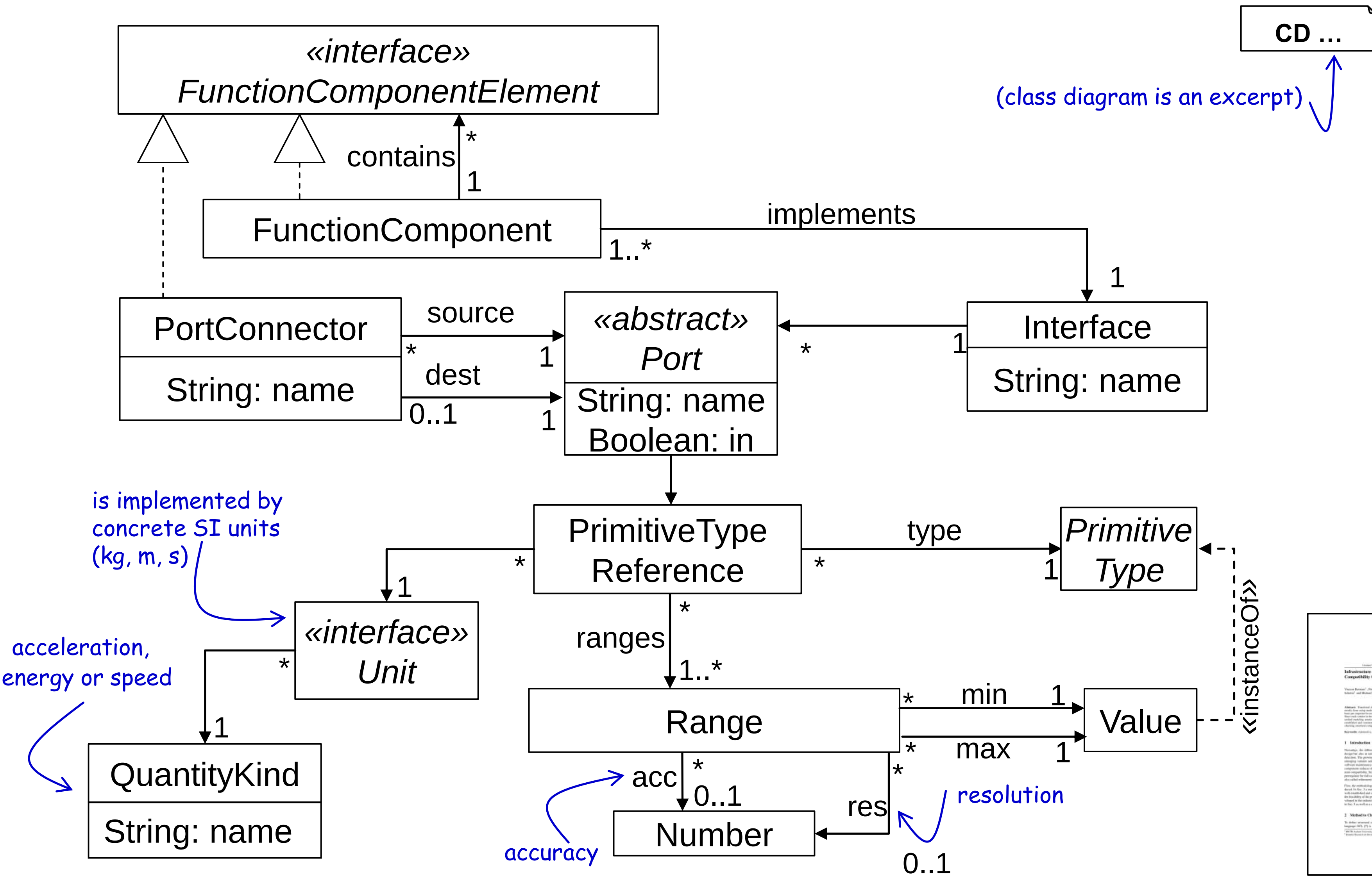
- It is not complicated to define interface compatibility, but still, you can face a lot of small constraints:
  - (1) primitive data type compatibility (e.g. when are enumerations compatible)
  - (2) unit compatibility such as km/h and m/s
  - (3) range compatibility considering several ranges each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

Vincent Bertram

Software Engineering
Chair (Bernhard Rumpe)

RWTH Aachen University

Derived MetaModel



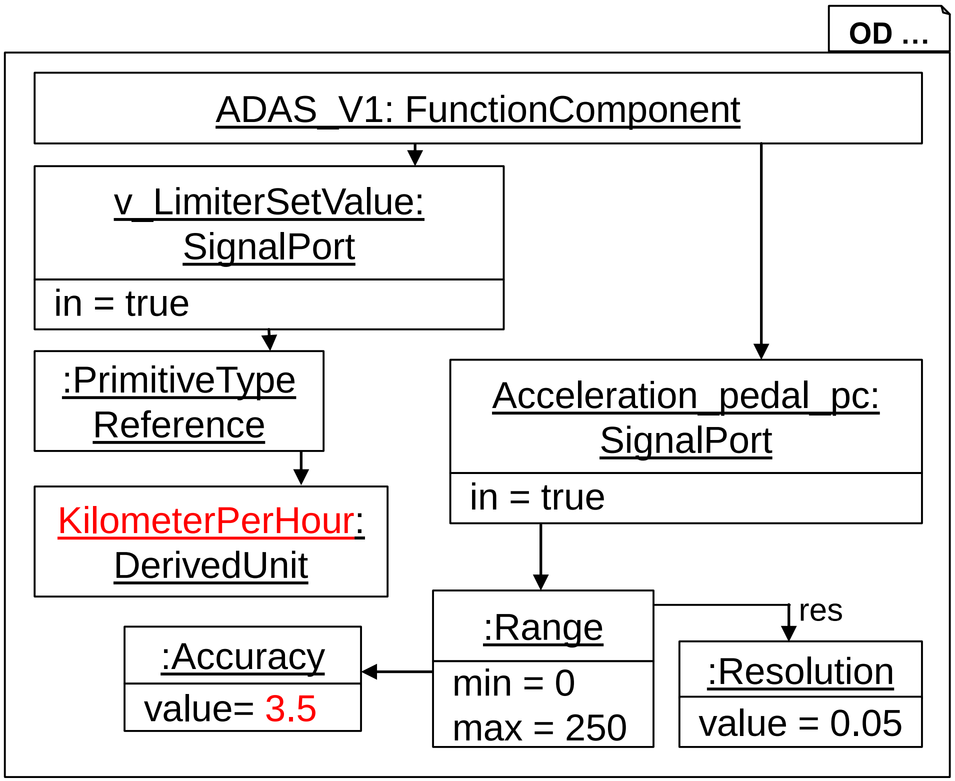
Bertram, V., Manhart, P., Plotnikov, D., Rumpe, B., Schulze, C., von Wenckstern, M.: Infrastructure to Use OCL for Runtime Structural Compatibility Checks of Simulink Models. In: Modellierung (2016)

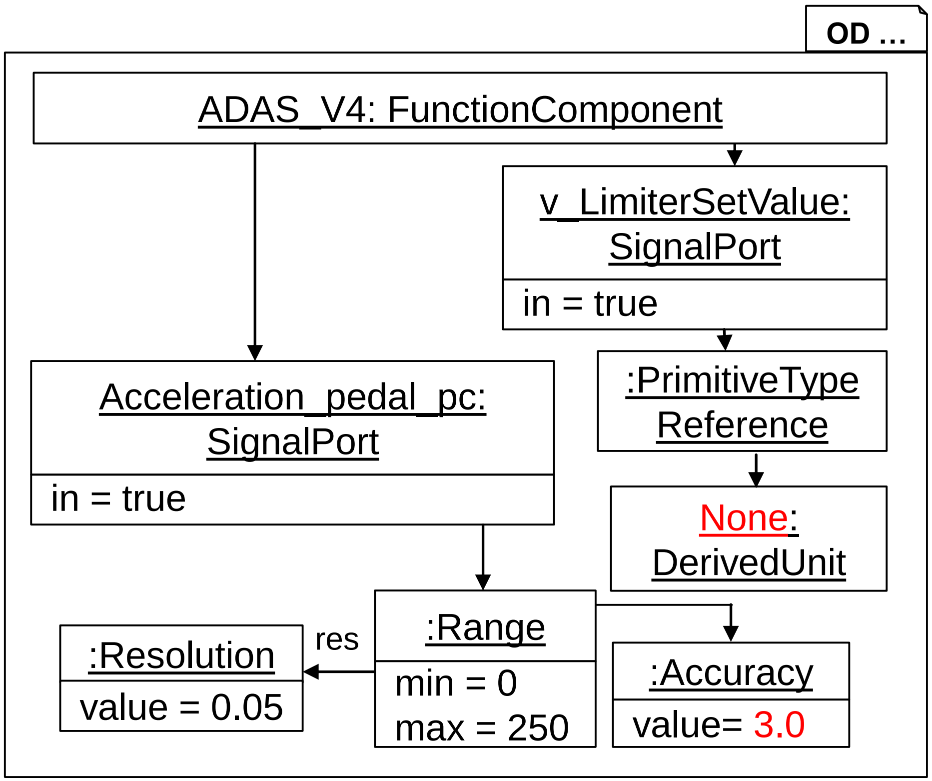
Vincent Bertram

Software Engineering
Chair (Bernhard Rumpe)

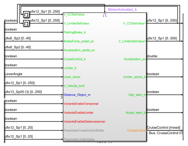
RWTH Aachen University

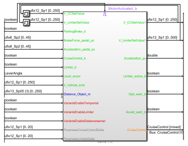
ADAS Object Diagram instantiation





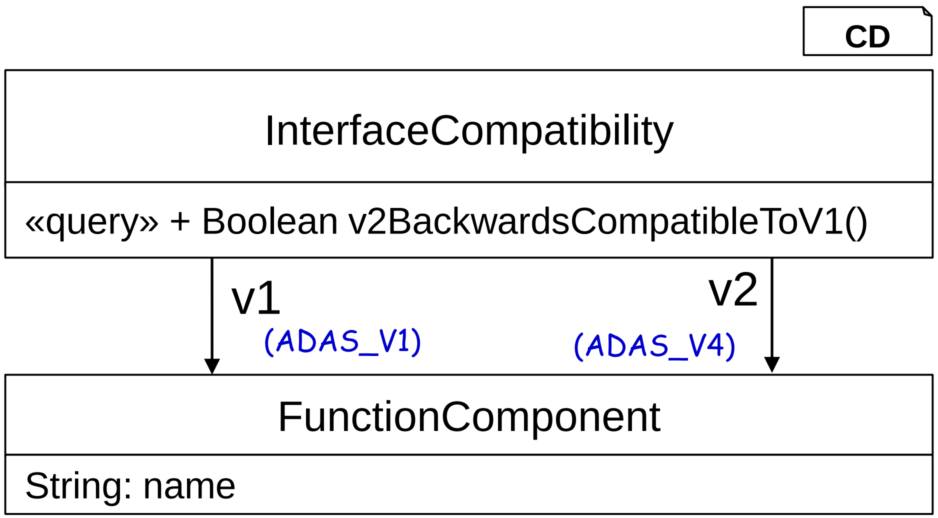
Interface backward compatibility between two Component & Connector (C&C) models. Given if ADAS_V4 can replace ADAS_V1.





Vincent Bertram
Software Engineering
Chair (Bernhard Rumpe)
RWTH Aachen University

General interface compatibility constraint



OCL 2.4 …

```
context InterfaceCompatibility inv:
    self.v2BackwardsCompatibleToV1()
      = ...
```

the equals operator in OCL 2.4 is similar to OCL/P's <=>.

OCL/P…

```
context InterfaceCompatibility ic inv:
    ic.v2BackwardsCompatibleToV1()
      <=> ...
```

calls for every InterfaceCompatiblity object the query method `v2BackwardsCompatibleToV1()`.

The constraint has a context and describes an invariant, meaning the constraint must be satisfied for all InterfaceCompatiblity class instances.

the context in which a constraint is embedded into; e.g. names of classes, attributes and/or properties in class diagrams

a boolean statement about a system

describes a property that must hold in a system at each point of time

Definition of Library Expressions

- Defining an OCL library with expressions in a function-like way

- The definition expression defines new attributes and query operations to existing models, which can be used in other constraints. Define a new query method: `IsBackwardsCompatibleTo(FunctionComponent v1)` to the CD model FunctionComponent and makes the modeling of an extra class like `InterfaceCompatibility` in the CD redundant.

- This function depends on other compatibility constraints such as data type, range, unit compatibility. This would result in polluting CD with unnecessary, and probably not reusable, query functions just to make one definition good readable.
  - 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

Vincent Bertram

Software Engineering
Chair (Bernhard Rumpe)

RWTH Aachen University

Library definition in OCL/P and OCL

OCL/P…

```
library InterfaceCompatiblity is:
  + def boolean v2BackwardsCompatibleToV1
           (FunctionComponent v1, FunctionComponent v2) is:
    result =
      (forall Port ports1 in v1.implements.ports,
              Port ports2 in v2.implements.ports:
                dataTypeCompatible(ports1, ports2))
      && …
  - def boolean dataTypeCompatible(Port p1, Port p2) is:
    result = …
```

public

private

OCL 2.4 …

```
package InterfaceCompatiblity
  context FunctionComponent
    def: v2BackwardsCompatibleToV1(v2: FunctionComponent)
                                                  :Boolean =
      self.implements.ports->forAll(ports1: Port |
        v2.implements.ports->forall(ports2: Port |
                dataTypeCompatible(ports1, ports2))
      && …
  context Port
    def: dataTypeCompatible(p2: Port) :Boolean = …
endpackage
```

Example for an OCL library with public and private query functions

Define new functions and operators

there exists no equivalent translation in OCL 2.4
("Definition constraint must be attached to a Classifier")

OCL/P

```
def boolean v2BackwardsCompatibleToV1
     (FunctionComponent v1, FunctionComponent v2) is:
```

⇕ semantically equivalent

OCL/P

```
context FunctionComponent v1:
  def boolean v2BackwardsCompatibleToV1(FunctionComponent v2) is:
```

OCL 2.4

```
context FunctionComponent
  def: v2BackwardsCompatibleToV1(v2: FunctionComponent) :Boolean =
```

- The top part shows the previous convenience definition. FunctionComponent extending the class FunctionComponent with an extra query function v2BackwardsCompatibleToV1().
- 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.

Vincent Bertram
Software Engineering Chair (Bernhard Rumpe)
RWTH Aachen University

Overload infix / prefix operators using OCL/P

OCL/P…

```
def boolean infix (Unit u1) ~ (Unit u2) is:
  result = u1.quantityKind == u2.quantityKind
def boolean infix (Number v) in (Range r) is:
  result =
    v >= r.min &&
    v <= r.max &&
    (~r.res || (v - r.min) % r.res == 0)
def boolean infix (Number v) in (List<Range> ranges) is:
  result = exists Range r in ranges: v in r
def boolean typeReferenceCompatible(PrimitiveTypeReference tR1,
                                    PrimitiveTypeReference tR2) is:
  let
    PrimitiveTypeReference tR1c = tR1.convert(tR2.unit)
  in
    result =
      tR1.unit ~ tR2.unit &&
      forall Number v in tR1c.ranges:
        v in tR2.ranges &&
        …
def boolean prefix ~ (Association a) is:
  result = a.size > 0
```

(Range r has no optional association res to Resolution)

(converts e.g. tR1 from eg. 1 m in 100 cm)

syntax of overloading a prefix operator

Overload operators using OCL

changed from '~' to '=', because OCL 2.4 can only overlaod:
'+', '-', '*', '/', '=', '<>', '<', '>', '<=', '>='

OCL 2.4 …

```
context Unit
  def: _'='(u2: Unit) :Boolean = self.quantityKind == u2.quantityKind
context Number
  def: isInRange(r: Range) :Boolean =
    self >= r.min &&
    self <= r.max &&
    (r.res->notEmpty() || (self - r.min) % r.res == 0)
  def: isInOneRange(ranges: Sequence(Range)) :Boolean =
    ranges->exists(r: Range | self.isInRange(r))
context PrimitiveTypeReference
  def: typeReferenceCompatible(tR2: PrimitiveTypeReference) :Boolean =
    let
      tR1c: PrimitiveTypeReference = tR1.convert(tR2.unit)
    in
      tR1.unit = tR2.unit &&
      Number.allInstances()->forall(v: Number |
        v.isInOneRange(tR1c.ranges) implies
          v.isInOneRange(tR2.ranges) &&
          …
```

OCL Error Classes for Intuitive Feedback

- 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.

- One drawback of a 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.

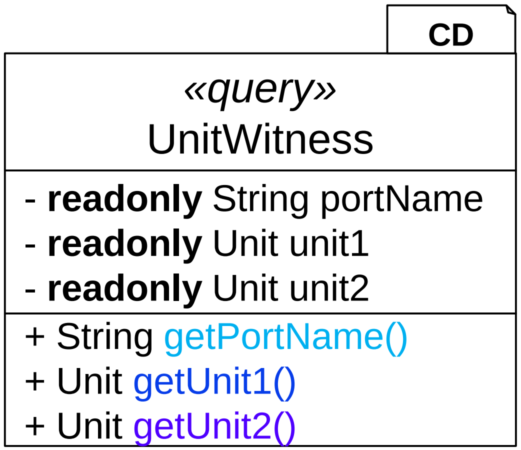
- A definition of error classes overcomes this drawback by providing easy to understand witness instantiations of an OCL error class.

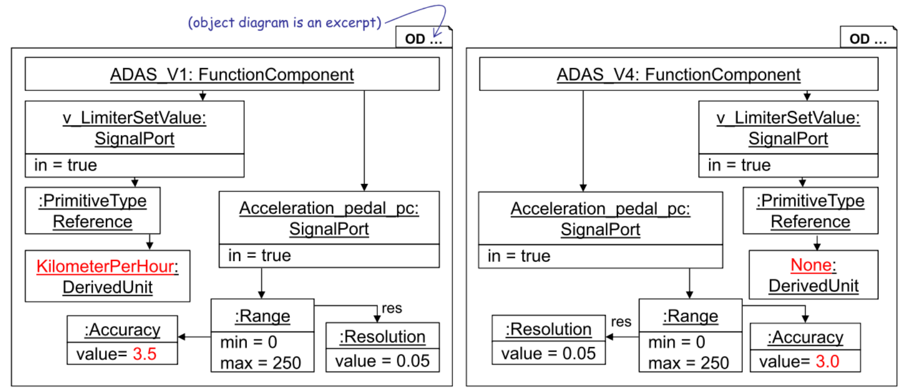
Vincent Bertram

Software Engineering
Chair (Bernhard Rumpe)

RWTH Aachen University

Example error class instantiation





A valid UnitWitness instantiation related to the previos OD would have the attribute values:

```
portName = "v_LimiterSetValue"
unit1 = "KilometerPerHour"
unit2 = "None"
```

A witness instance can be used in templates for generating user friendly text messages. In order to maintain only one kind of artifact, OCL has been extended by the counterexample keyword as demonstrated:

```
counterexample String portName, Unit unit1, Unit unit2 inv UnitWitness:
```

Vincent Bertram
Software Engineering
Chair (Bernhard Rumpe)
RWTH Aachen University

Defining error classes producing counterexamples

CD

«query»
UnitWitness

- **readonly** String portName
- **readonly** Unit unit1
- **readonly** Unit unit2

+ String getPortName()
+ Unit getUnit1()
+ Unit getUnit2()

OCL/P…

```
context UnitWitness uw inv:
  let
    SignalPort p1 = uw.getUnit1().
      primitiveTypeReference.signalPort;
    SignalPort p2 = uw.getUnit2().
      primitiveTypeReference.signalPort;
  in
    p1.interface.name != p2.interface.name &&
    uw.getPortName() == p1.name &&
    uw.getPortName() == p2.name &&
    !(uw.getUnit1() ~ uw.getUnit2())
```

OCL/P can navigate against navigation direction

ports belong to different function components (different interface names)

ports have the same name

units of the ports are not compatible

semantically equivalent

CD + OCL/P `context` is replaced by OCL/P `counterexample`

OCL/P…

```
counterexample String portName, Unit unit1, Unit unit2 inv UnitWitness:
  let
    SignalPort p1 = unit1.primitiveTypeReference.signalPort;
    SignalPort p2 = unit2.primitiveTypeReference.signalPort;
  in
    p1.interface.name != p2.interface.name &&
    portName == p1.name && portName == p2.name &&
    !(unit1 ~ unit2)
```

Vincent Bertram
Software Engineering
Chair (Bernhard Rumpe)
RWTH Aachen University

OCL Conditions to define Error Classes

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.
…	

For each formalized ContextCondition also a counterexample class is given to produce meaningful user feedback.

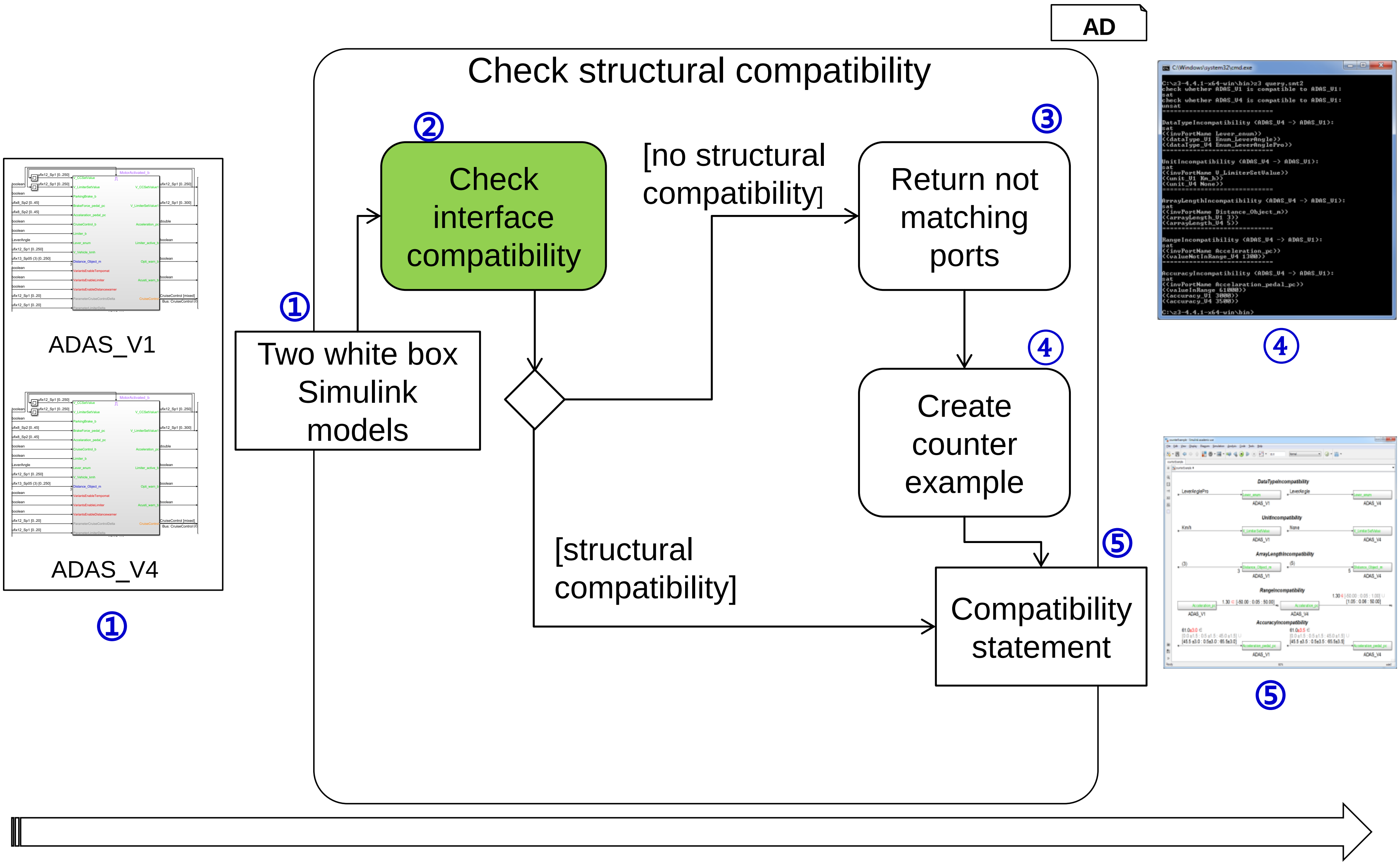
Vincent Bertram

Software Engineering
Chair (Bernhard Rumpe)

RWTH Aachen University

Outline

Workflow: Check structural compatibility

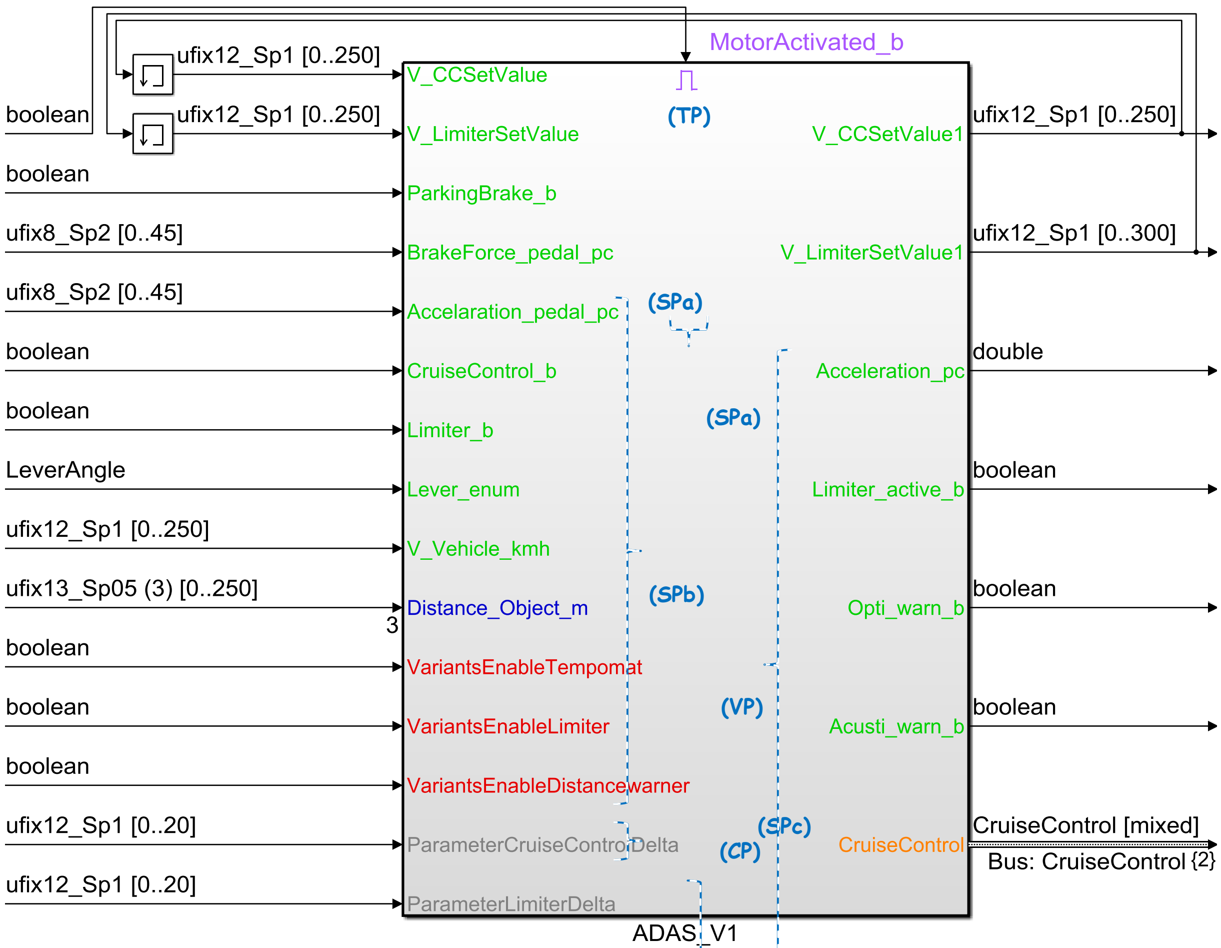


Vincent Bertram

Software Engineering
Chair (Bernhard Rumpe)

RWTH Aachen University

ADAS main component



LeverAngle.m

```
classdef(Enumeration) LeverAngle <
Simulink.IntEnumType
    enumeration
        ZeroDegree(0)
        PlusFiveDegree(1)
        MinusFiveDegree(2)
    end
end
```

LeverAnglePro.m

```
classdef(Enumeration) LeverAngle <
Simulink.IntEnumType
    enumeration
        ZeroDegree(0)
        PlusFiveDegree(1)
        PlusSevenDegree(2)
        MinusFiveDegree(3)
        MinusSevenDegree(4)
    end
end
```

Legend:

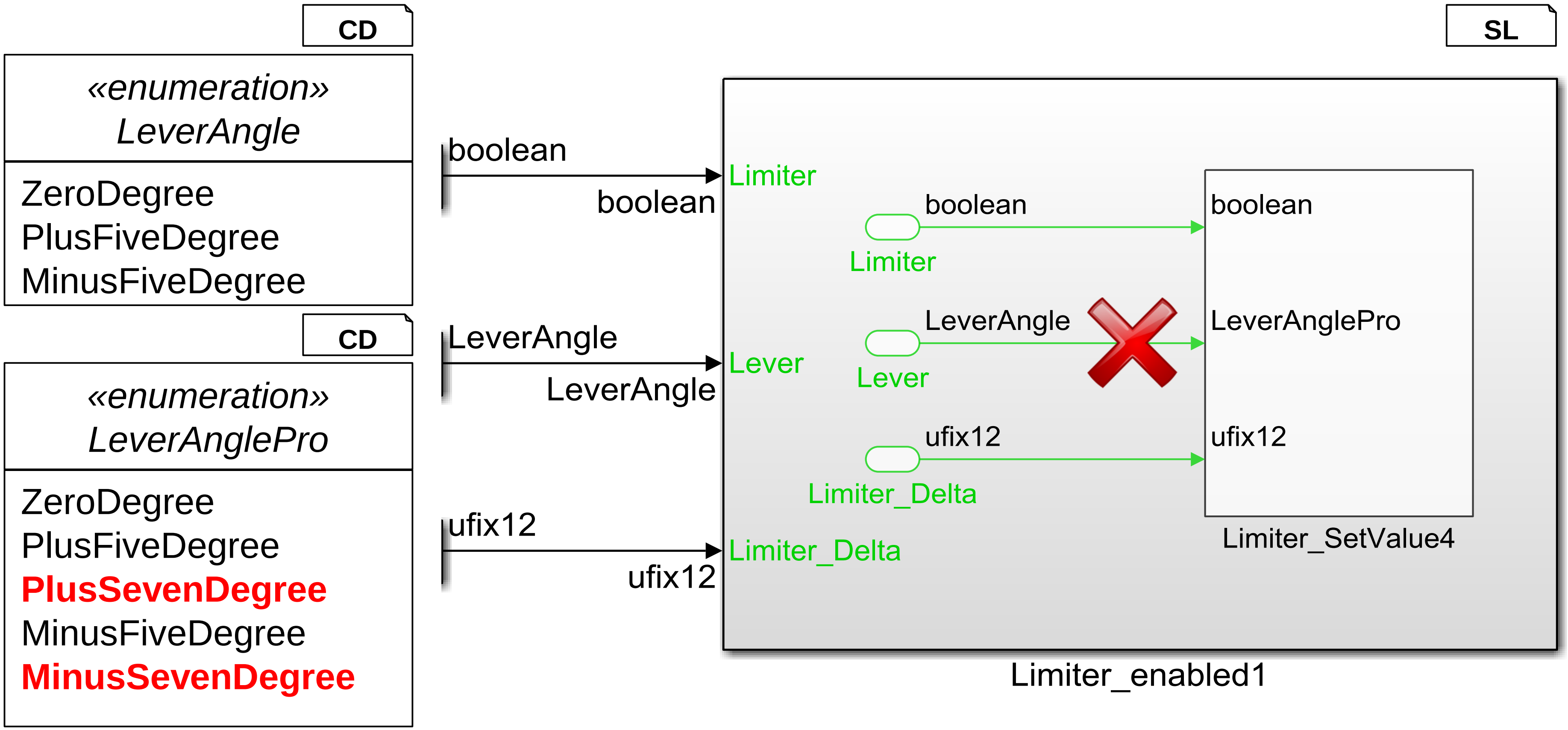
(1)
(SPa) green port names - signal ports with primitve data type
(SPb) blue port names - signal ports with array as data type
(SPc) orange port names - signal ports with bus object (C struct) as data type

(2)
(VP) red port names - variation ports
(CP) grey port names - calibration ports

(3)
(TP) violette port names- trigger ports

Vincent Bertram
Software Engineering
Chair (Bernhard Rumpe)
RWTH Aachen University

Data type compatibility fault of type enumeration

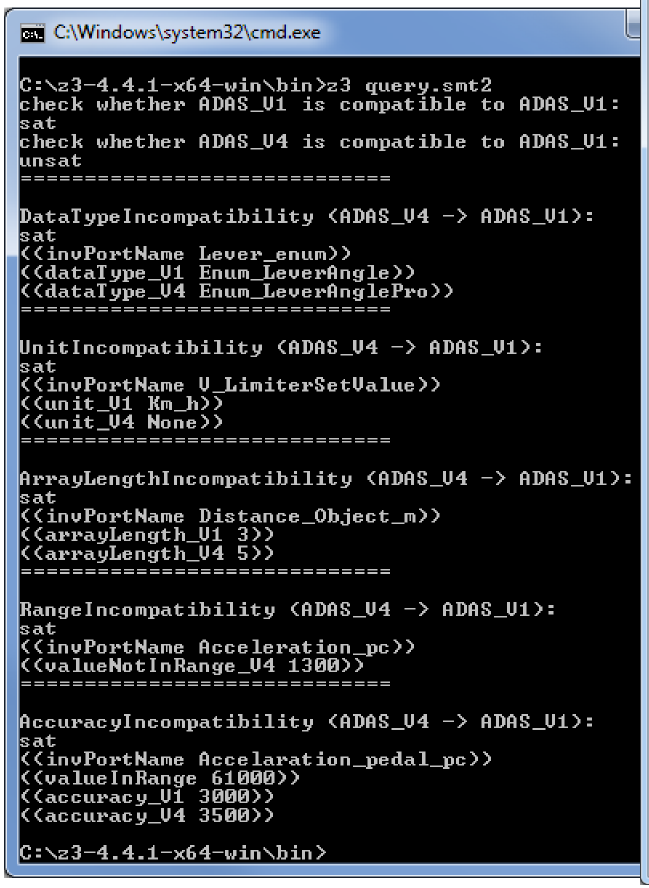


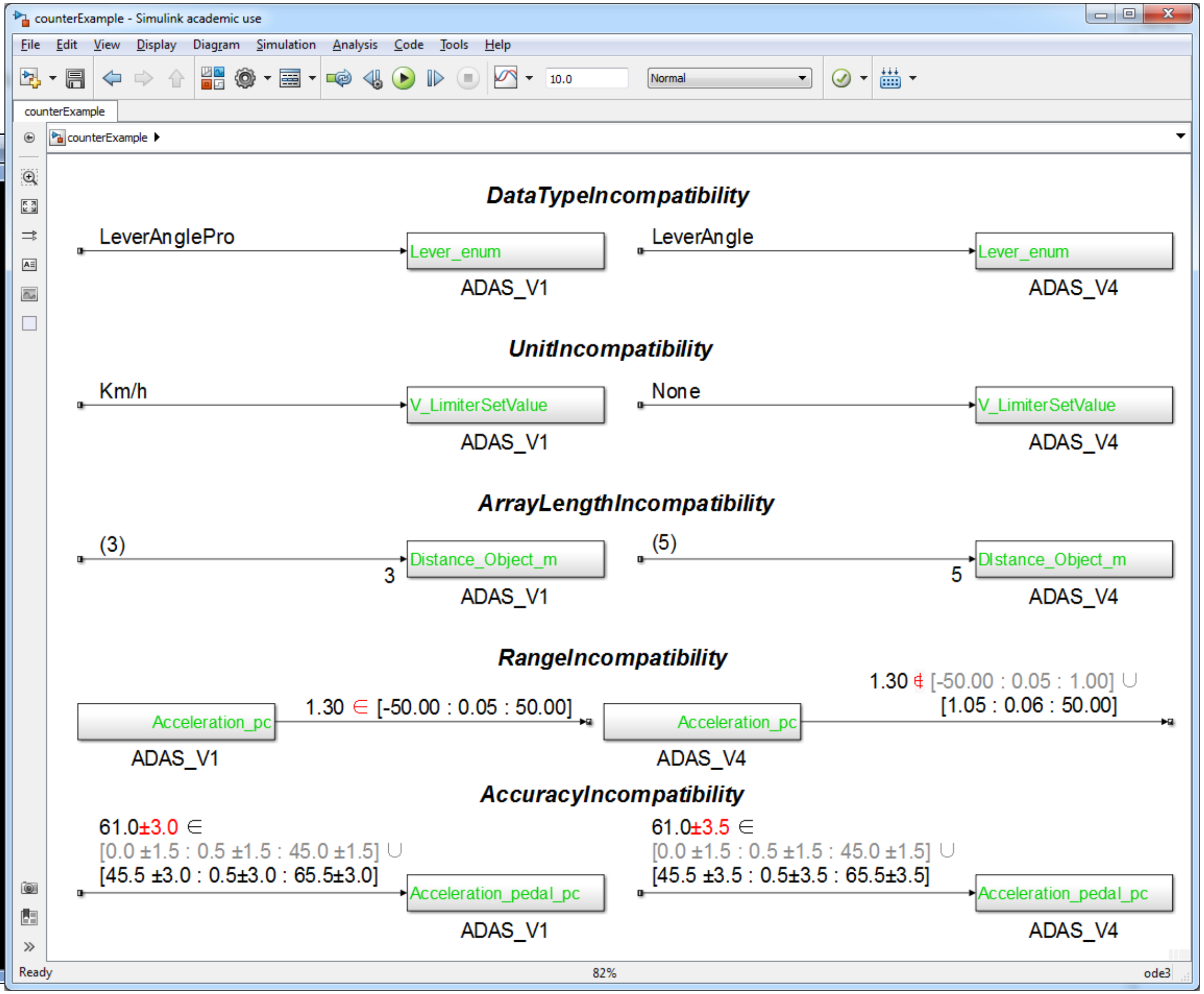
Vincent Bertram

Software Engineering
Chair (Bernhard Rumpe)

RWTH Aachen University

Simulink: structural incompatibility





Structural compatibility errors found (left) are illustrated as Simulink model (right)

Outline

Conclusion (Concepts)

- The OCL library concept with private and public constraints, supports to organize the amount of constraints by hiding unnecessary details for developers.
- Maybe operator overloading would not be necessary, but it made it easier to read OCL constraints.
- The most important concept is the introduction of error classes, other-wise it is not possible to use OCL for ContextConditions, since the user needs to know which C&C element causes a constraint to fail.
- To create even better user feedback, error classes are prioritized. 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).

making a unit dimensionless results in changing its value ranges and its accuracies

Conclusion (Answering the initial questions)

- How to logically group OCL constraints?

  Create OCL libraries to structure the code and use their encapsulation mechanisms with private and public constraint definitions.

- How to split up complex constraints easily into multiple smaller ones?

  Create smaller OCL helper constraints by the OCL `def` operator. It is now very similar to splitting large Java or C function into smaller ones.

- How to use OCL operators for self-defined model structures?

  Thanks to operator overloading, a well-known principle in many languages, self-defined models, e.g. complex numbers defined as a CD, can be accessed as intuitive (e.g. + operator) as the OCL basic types such as integer numbers.

- How to produce meaningful user error messages?

  A methodology on how to specify and prioritize error classes for users to generate intuitive user feedback was developed.

Finish

Thank you for your attention.