# Execute OCL Constraints

## 1.1 Execution

## Explanation

In this OCL-Expression we first iterate through all the **ManufacturingSystemElement** instances, which are referenced by our **ManufacturingSystem**’s **consistsOf** reference. Then for each of these we iterate through the **ManufacturingSystemElement** instances referenced by their **transition** reference. In the end we don’t check anything resulting in a Boolean, so the execution just returns all the **ManufacturingSystemElement** instances we iterated over. The final output lists all the elements which are directly **transition**ed into by elements directly referenced through **consistsOf** by our system.

## Execution

**Evaluating:** self.consistsOf->forAll(m:ManufacturingSystemElement | m.name <> null)

**Results:** true

## Explanation

Like in the previous expression we iterate through all the elements referenced by **consistsOf** from our system. Then we check for all of them, if their **name** property is not null. This will only return **true**, if every element referenced by **consistsOf** has a non-null **name** property. Unfortunately, an empty string is also not null, so this will return true even if elements are not properly named.

## Execution

**Evaluating:** self.consistsOf->select(oclIsKindOf(Step)).oclAsType(Step).speed->sum()

**Results:** 70

## Explanation

We first iterate through all the elements our system references with **consistsOf**. Then we limit the collection to only those which are based on the **Step** class. Then we cast these elements to the **Step** class, which has a **speed** property, for which we return the sum of. So basically, the sum of the speeds of all steps directly referenced by **consistsOf** from our system.

## Execution

**Evaluating:** self.output.input->forAll(i:InputCondition | self.input->includes(i))

**Results:** true

## Explanation

We get all the inputs (*InputCondition*) used by all the possible outputs *(OutputDecision)* of the step in our context and we check if all these inputs are also included in the step’s input references. In our example, the TransportStep CuttingCompositeTransport has 2 outputs. One requires the “final\_wood” and the other the “final\_metal” input condition. Since both inputs are also referenced by the step itself, the evaluation results in **true.**

## Execution

**Evaluating:** self.transition->exists(m:ManufacturingSystemElement | m.oclIsTypeOf(QualityAssuranceStep)) implies self.oclIsKindOf(ManufacturingStep)

**Results:** false

## Explanation

First, we iterate through all **transition** references of element. Then we check if at least one of the elements we **transition** into is an instance of the **QualityAssuranceStep** class. If that is true, we also need to check if we are an instance of **ManufacturingStep** or any of its subclasses. So in our example, when the expression is executed for an instance of **CompositeManufacturingStep**, which has a **transition** reference to a **QualityAssuranceStep** instance, it will return false, because 1. It transitions into a **QualityAssuranceStep** and 2. It’s not an instance of **ManufacturingStep** *(or any of it’s subclasses, even though we don’t have subclasses of it).*

## Execution

**Evaluating:** ManufacturingSystem.allInstances()

->forAll(ms:ManufacturingSystem|ms.transforms.hasType

->includes(self) implies ms.uses->includes(self))

**Results:** true

## Explanation

We iterate through all instances of **ManufacturingSystem** and check for all of them, if at least one of their **WorkPiece** instances referenced by **transform** references the **WorkPieceType** instance, which we are running this expression on, via its **hasType** reference. If that is the case, we also need to check if this system’s **uses** reference collection includes our **WorkPieceType**. So this would return true for any system, which does not **transform** any workpiece of our type and for those which do but also includes our workpiece in their **uses** reference.

# Extend the Meta-Model

## Each Step shall have exactly one Responsible.

**Invariant:**   


**Context:** abstract class Step extends ManufacturingSystemElement

**Validation:**

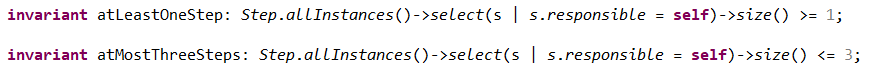
Constraint violated in:

* Transport Step StartTransport
* Manufacturing Step Cutting
* Transport Step CutTransport
* Manufacturing Step Drilling
* Transport Step EndTransport

**Explanation:**

We get the collection of people responsible for the Step in our context and simply check if the size of the collection is exactly 1.

## Each Responsible shall be responsible for between one and three (inclusive) Steps.

**Invariants:**   


**Context:** class Responsible

**Validation:**

Constraint violated in:

* Responsible Peter

**Explanation:**

We created two invariants for this constraint. One for the upper and one for the lower limit. In both, we first get a collection of all Step instances. Then we limit this collection to only those which the responsible in our context is responsible for. We then get the size of this collection and check if it’s within the lower and upper limit.

## There shall not exist a single Step in a ManufacturingSystem which has the same Responsible as the ManufacturingSystem itself.

**Invariant:**   


**Context:** class ManufacturingSystem

**Validation:**

Constraint violated in:

* Manufacturing System FurnitureSystem

**Explanation:**

We start in the context of ManufacturingSystem and first check if the system even has a responsible. If not, the constraint is fulfilled. If it does have a responsible, we get a collection of all the elements it consists of. Then we limit this collection to only those which are steps. Then we check for all of those steps, if their responsible is different from the system’s responsible. So, if a system has a responsible and at least one of it’s steps also has the same responsible, the constraint will not be fulfilled.

## No start StoragePoint shall have incoming Transitions.

**Invariant:**   


**Context:** class ManufacturingSystemElement

**Validation:**

Constraint violated in:

* Transport Step EndTransport
* Transport Step CuttingCompositeTransport
* Storage Point ProductStorage

**Explanation:**

We start in the context of ManufacturingSystemElement and get a collection of all the elements it transitions into. Then we limit this collection to only include elements of the class or subclass *(in case subclasses of it are created in the future)* of StoragePoint. Finally, we check if the size of this collection is exactly 0 which means that this element does not transition into a storage point.

## No end StoragePoint shall have outgoing Transitions.

**Invariant:**

**Context:** class StoragePoint extends ManufacturingSystemElement

**Validation:**

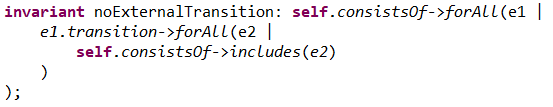
Constraint violated in:

* Storage Point RawMaterialStorage
* Storage Point PreparationStartStorage
* Storage Point ProductStorage

**Explanation:**

We simply start in the context of StoragePoint and check if the number of elements in the transition collection is exactly 0. This ensures that storage points cannot have any outgoing transitions.

## Transitions shall only connect ManufacturingSystemElements within the same ManufacturingSystem.

**Invariant:**   


**Context:** class ManufacturingSystem

**Validation:**

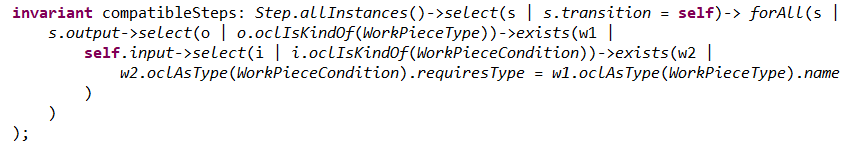
Constraint violated in:

* Manufacturing System FurnitureSystem
  + For example, by “Transport Step StartStorageTransport” transitioning into “Manufacturing Step Cutting”, which is part of a different system.

**Explanation:**

We start in the context of ManufacturingSystem and check for all the elements it consists of, that they do not transition into an element which the system in our context does not consist of.

## Outputs and inputs of connected Steps shall be compatible.

**Invariant:**   


**Context:** abstract class Step extends ManufacturingSystemElement

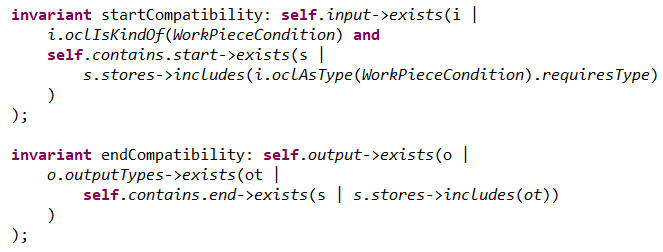
**Validation:**

Constraint does not seem to be violated in our example model.

**Explanation:**

We first get a collection of all Step instances and limit the collection to only those, which transition into the Step in our context. Then we get check for all of them if they have an output WorkPieceType which is required by an input WorkPieceCondition by the Step in our context. To do this we get the WorkPieceType outputs from the other Step and check if at least one exists, which is the required type of an WorkPieceCondition input of the Step in our context.

## For composite steps, the input needs to be compatible (see 7.) with the types stored in one of the start points and the output needs to be compatible with the types stored in one of the end points.

**Invariants:**  


**Context:** class CompositeManufacturingStep extends Step

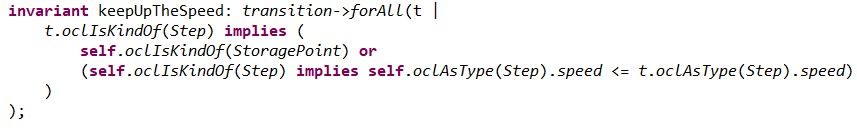
**Validation:**

Invariants do not seem to be violated in our example model.

**Explanation:**

Since the task points to task 7 for understanding what compatible means, we interpreted it as meaning that at least one of the input and output has to be included in the start and end storage of the composite step’s system.  
  
We split this constraint into two invariants.  
  
The first one checks if an input exists, which is a WorkPieceCondition and if the system contained by the composite step in our context has any storage facility which includes the input’s work piece type.  
  
The second invariant checks if an output exists, which has at least one output type, which at least one of the storage facilities of the system contained in the composite step in our context includes.

## For each Step, the previous ManufacturingSystemElement (if existent) shall either be a StoragePoint or the speed of that Step shall be lesser or equal than the speed of the current Step. That is, the speed of the manufacturing system can not suddenly become lower without a storage (bottleneck).

**Invariant:**   


**Context:** class ManufacturingSystemElement

**Validation:**

Constraint violated in:

* Composite Manufacturing Step CuttingComposite
* Manufacturing Step Cutting
* Manufacturing Step Drilling

**Explanation:**

To make the task easier we viewed it from the context of the previous element. We first check for all transitions, if the transition leads to a Step element. If it does not, the transition is fine. If it does, we also need to check if the element in our context is a storage point or if it is a Step and its speed is lesser or equal to the speed of the Step we are transitioning into.