Introduction to **ADORE Evaluation Interfaces**

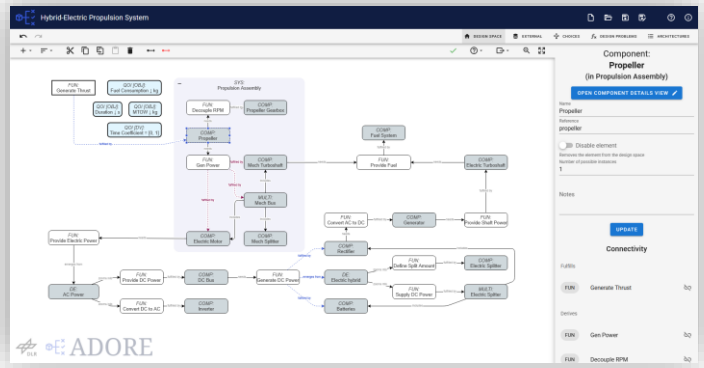
ADORE Evaluation Interfaces | 2026-01-22

Jasper Bussemaker (jasper.bussemaker@dlr.de)

DLR Institute of System Architectures in Aeronautics (SL), Hamburg, Germany

https://adore.mbse-env.com/docs/

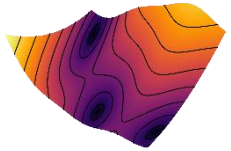
The System Architecture Optimization (SAO) Loop



The ADORE Model

Defines architectural choices and metrics.

formalizes the optimization problem

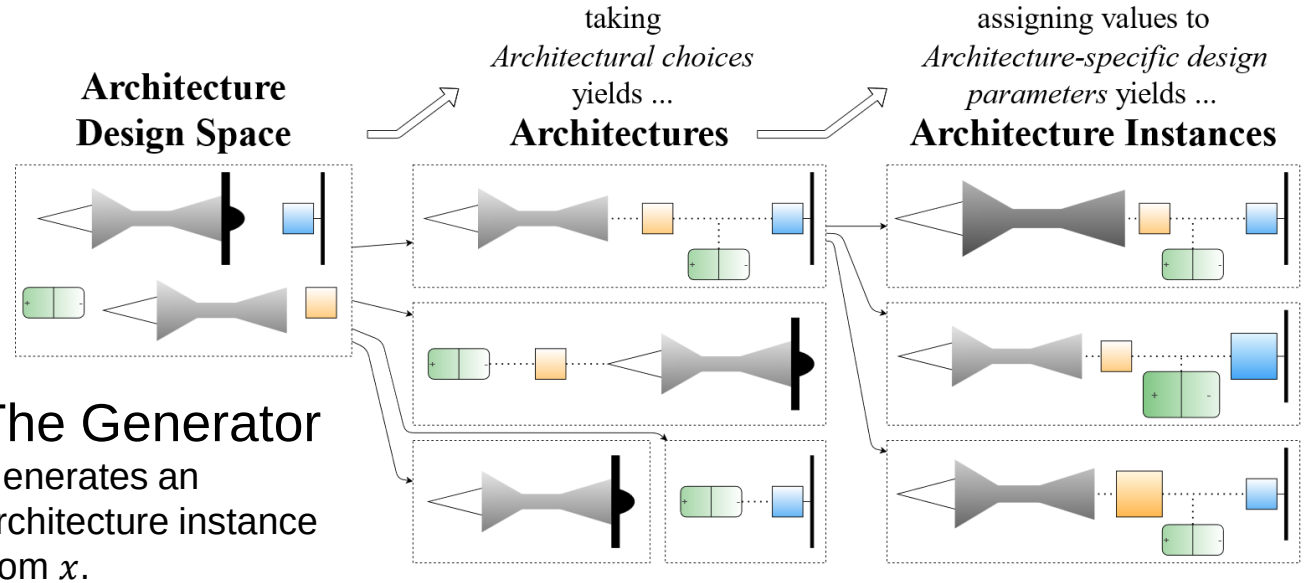


The Optimizer

Uses machine learning algorithms.

generates design vector x

provides



The Generator

Generates an architecture instance from x.

each architecture instance is evaluated by

The Evaluator

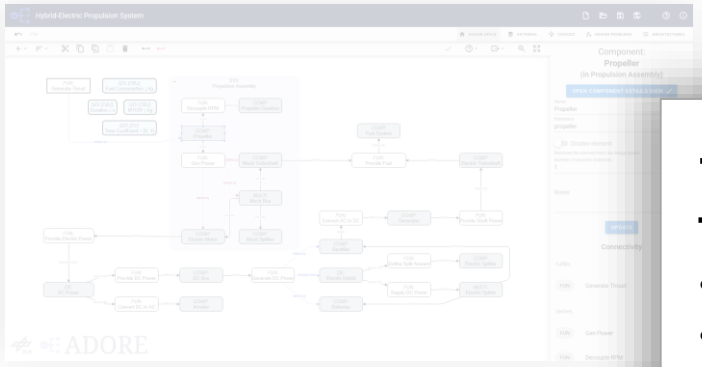
Executes problem-specific multidisciplinary analysis code.

provides objective f and constraint g values to optimizer

ADORE

https://adore.mbse-env.com/docs/

The Architecture Evaluator



The ADORE Model

Defines architectural choices and metrics.

formalizes the optimization problem

The Evaluator

- Is problem-specific.
- Should calculate **all** relevant metrics for **all** architectures with **similar** fidelity.
- Should be **sensitive** to architectural choices.
- Should be **automatically executable** (i.e. without user interaction).

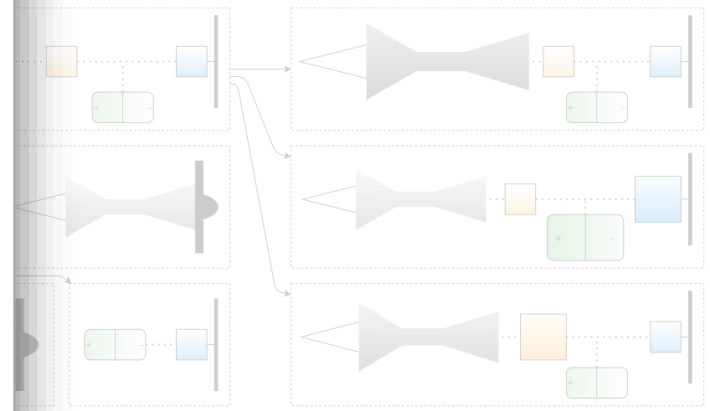
taking *Architectural choices* yields ...

Architecture

ectures

assigning values to *Architecture-specific design parameters* yields ...

Architecture Instances



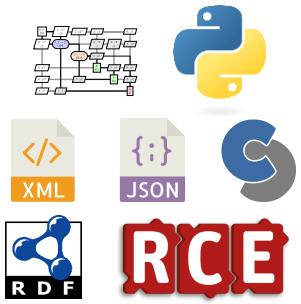
each architecture instance is evaluated by

generates design vector x

The Optimizer

Uses machine learning algorithms.

provides objective f and constraint g values to optimizer



The Evaluator

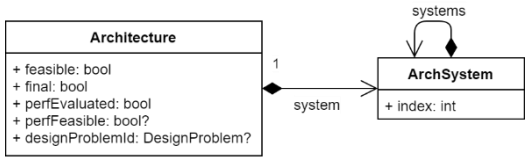
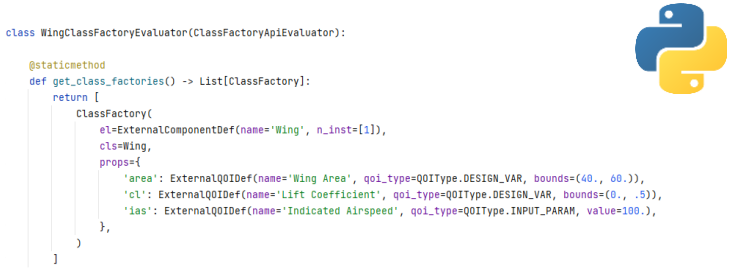
Executes problem-specific multidisciplinary analysis code.

ADORE

https://adore.mbse-env.com/docs/

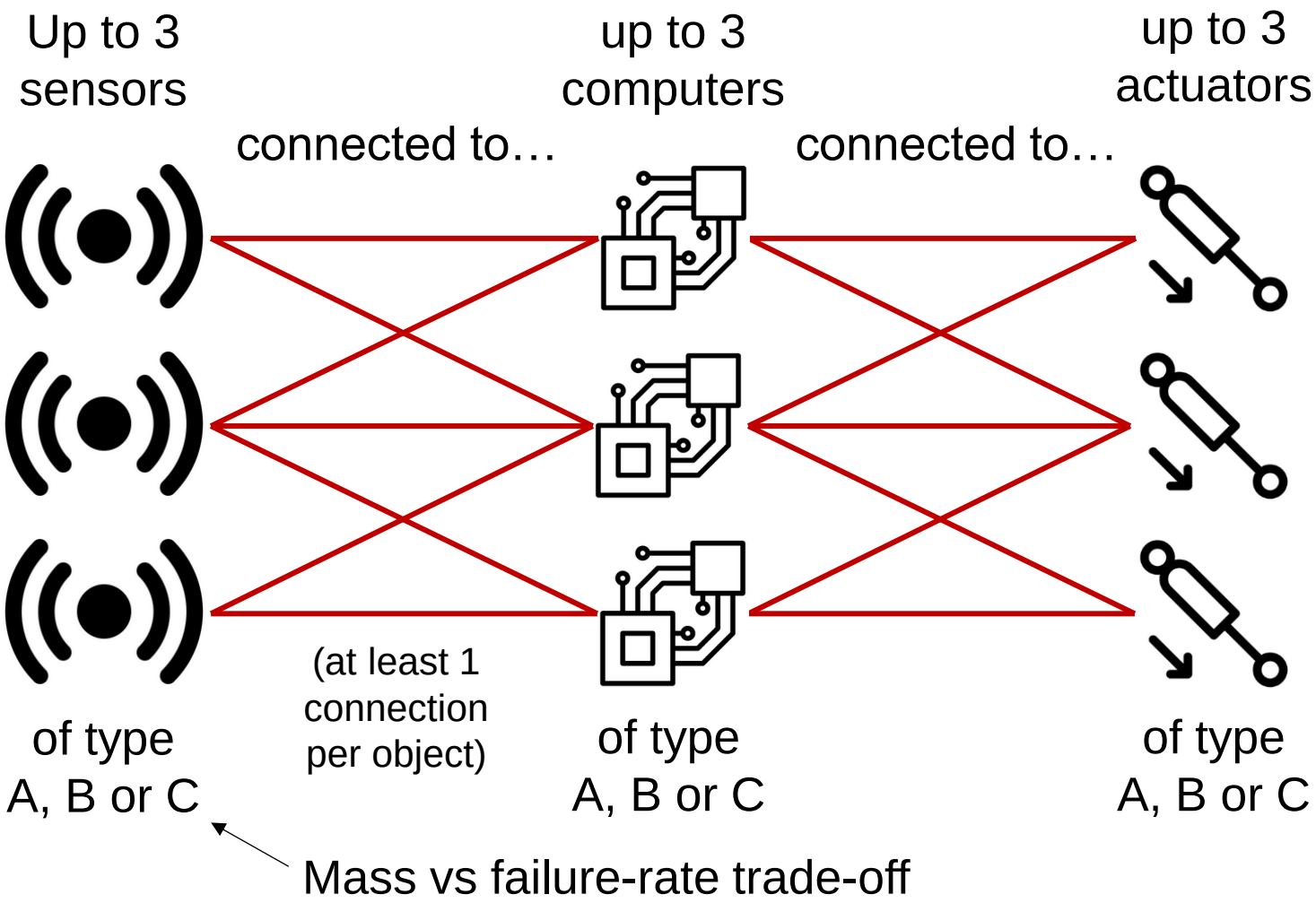
ADORE Evaluation Interfaces

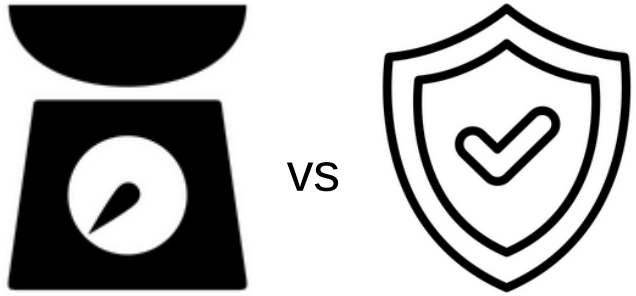
Supports no-code, external optimization
(e.g. in MDO integration platforms).

	Python-based	*File-based*
Direct access to ADORE data model	**Evaluation API** • `Architecture` instance as input • Need to parse data model yourself 	**Architecture Serialization** • Serialization of architecture to file • Placeholders for outputs XML JSON RDF
Rule-based translation	**Class Factory Evaluator** • Define rules to instantiate classes from architecture elements 	**Node Factory Evaluator** • Define rules to create XML nodes from architecture elements • Dedicated CPACS interface available XML

The Guidance, Navigation & Control Problem

DLR





How many architectures are possible?
79 091 323

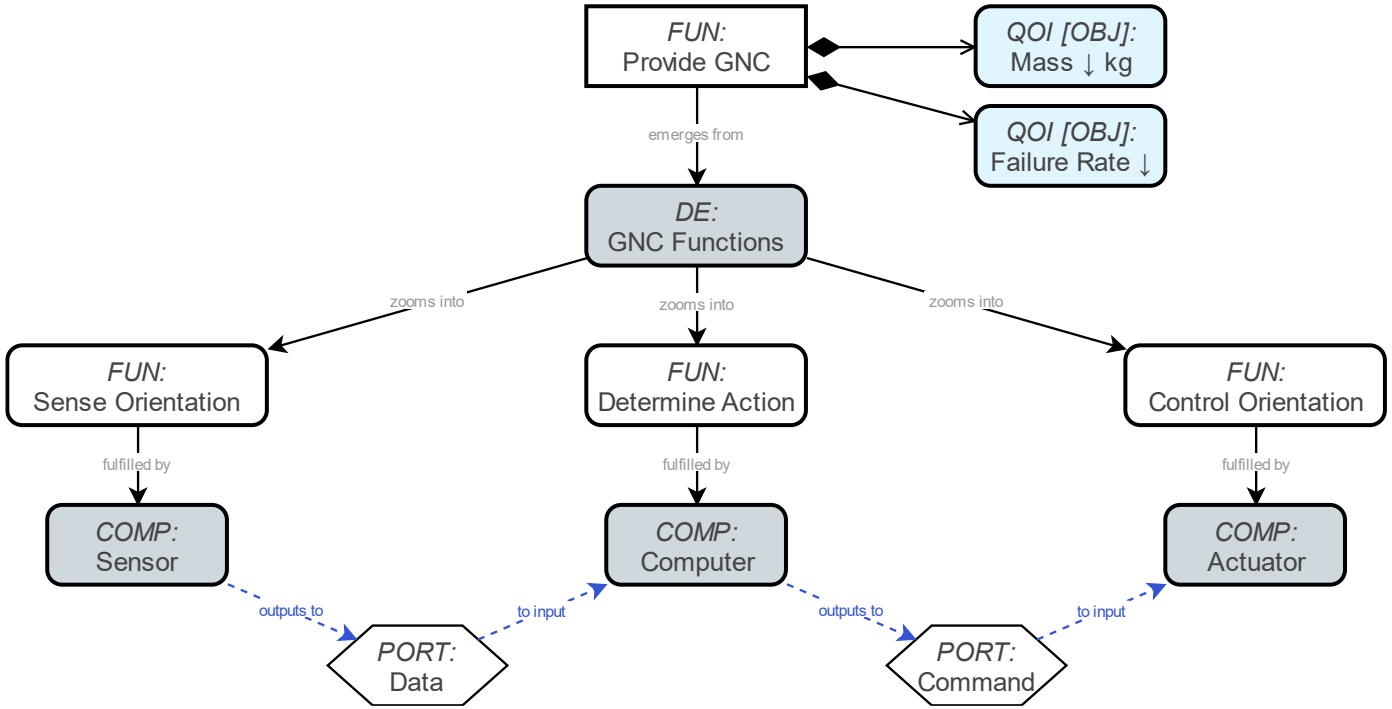
What if we can have up to 4 elements?
3 254 601 496 975

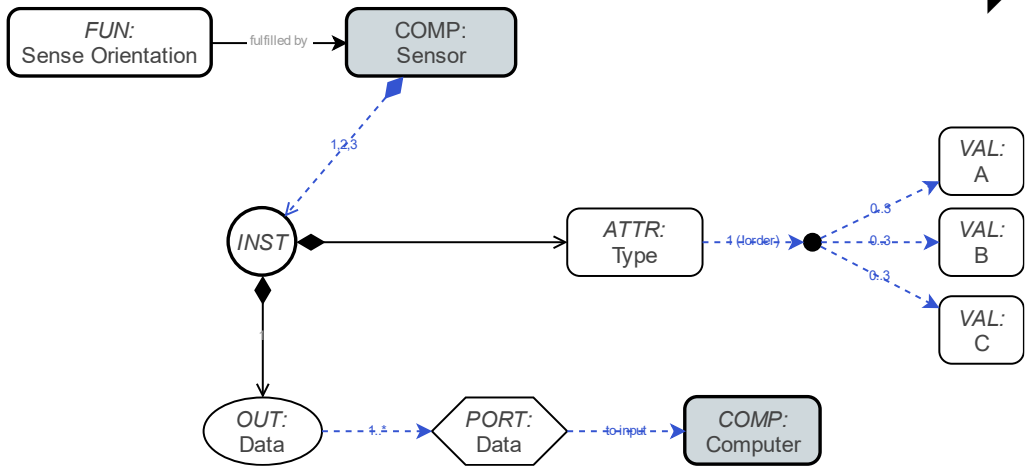
E. Crawley, B. Cameron, and D. Selva, "System Architecture", Harlow (UK), 2016

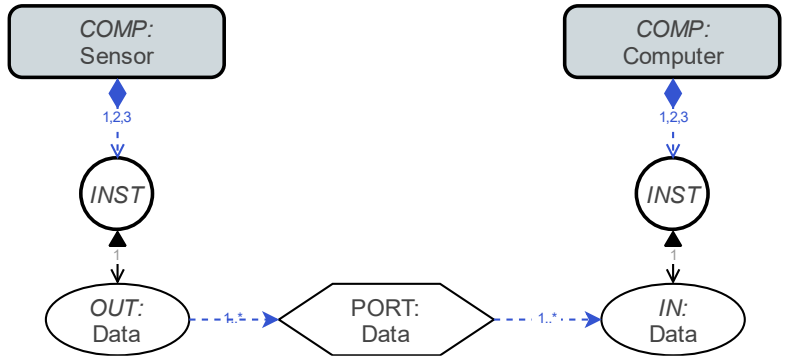
ADORE
https://adore.mbse-env.com/docs/

GNC ADORE Model

Open `gnc.adore` to familiarize with the model







Exercise: Setup ADORE

- Install ADORE locally
  - Follow installation instructions from the documentation

- Test the installation
  - Start python from the command line (in the activated environment)
  - `from adore.optimization.api import GraphApiEvaluator`
  - `evaluator = GraphApiEvaluator.from_file('gnc.adore')`
  - `problem = evaluator.get_arch_opt_problem()`
  - `problem.n_obj`

```python
import adore as ad

class MyPythonEvaluator(ad.GraphApiEvaluator):
    """Problem-specific Python-based evaluator."""

    def _evaluate(self, architecture: ad.Architecture, arch_qois: List[ad.ArchQOI], **kwargs) \
            -> Dict[ad.ArchQOI, Union[float, ad.Quantity]]:
        """
        The function that evaluates an architecture instance.
        Returns the value of each requested architecture QOI,
        either as a float (or NaN) or a Quantity (value + units).
        Use `self.quantity` to create a Quantity object.
        """

if __name__ == '__main__':
    # Instantiate your evaluator from your ADORE model
    evaluator = MyPythonEvaluator.from_file('model.adore')

    # Create random architectures and evaluate them
    for _ in range(10):
        architecture, design_vector, activeness_vector = evaluator.get_architecture(evaluator.get_random_design_vector())

        objectives, constraints = evaluator.evaluate(architecture)
        print(f'Objective values: {objectives}')
        print(f'Constraint values: {constraints}')

    # Save the ADORE file and a CSV export
    evaluator.to_file('updated_model.adore')
    evaluator.save_results_csv('results.csv')
```

DLR

ADORE

Python-based Evaluation API

Python-based Evaluation

Problem-specific evaluator
Implement the **`_evaluate`** function.

Instantiate
Instantiate the evaluator with your ADORE model.

Generate
Generate and evaluate architectures. Connect to optimization algorithms.

```python
import adore as ad

class MyPythonEvaluator(ad.GraphApiEvaluator):
    """Problem-specific Python-based evaluator."""

    def _evaluate(self, architecture: ad.Architecture,
                  arch_qois: List[ad.ArchQOI], **kwargs) \
            -> Dict[ad.ArchQOI, Union[float, ad.Quantity]]:
        """
        The function that evaluates an architecture instance.
        Returns the value of each requested architecture QOI,
        either as a float (or NaN) or a Quantity (value + units).
        Use `self.quantity` to create a Quantity object.
        """

if __name__ == '__main__':
    # Instantiate your evaluator from your ADORE model
    evaluator = MyPythonEvaluator.from_file('model.adore')

    # Create random architectures and evaluate them
    for _ in range(10):
        architecture, design_vector, activeness_vector = \
            evaluator.get_architecture(
                evaluator.get_random_design_vector())

        objectives, constraints = evaluator.evaluate(architecture)
        print(f'Objective values: {objectives}')
        print(f'Constraint values: {constraints}')

    # Save the ADORE file and a CSV export
    evaluator.to_file('updated_model.adore')
    evaluator.save_results_csv('results.csv')
```

Class Factory Evaluator (CFE)
Use the CFE to help extract data from the **`Architecture`**, by instantiating custom classes from architecture elements.

See next slide.

The _evaluate Function

```python
import math
from typing import *
import adore as ad

class MyPythonEvaluator(ad.GraphApiEvaluator):
    """Problem-specific Python-based evaluator."""

    def _evaluate(self, architecture: ad.Architecture,
                  arch_qois: List[ad.ArchQOI], **kwargs) \
            -> Dict[ad.ArchQOI, Union[float, ad.Quantity]]:
        """
        The function that evaluates an architecture instance.
        Returns the value of each requested architecture QOI,
        either as a float (or NaN) or a Quantity (value + units).
        Use `self.quantity` to create a Quantity object.
        """

        # Access the data in the architecture and run your analyses
        components = architecture.system.components

        # Do the calculations
        result_numerical = 42.
        result_quantity = self.quantity(42., 'kg')
        result_failed = math.nan

        # The output is a dict mapping the QOIs to their values
        results = {}
        for arch_qoi in arch_qois:
            if arch_qoi.name == 'My Numerical QOI':
                results[arch_qoi] = result_numerical
            elif arch_qoi.name == 'My Quantity QOI':
                results[arch_qoi] = result_quantity
            elif arch_qoi.name == 'My Failing QOI':
                results[arch_qoi] = result_failed
        return results
```

Inputs

The `architecture` instance to be evaluated and the list of architecture QOIs to be calculated.

Access architecture data

From the `architecture`.

Values

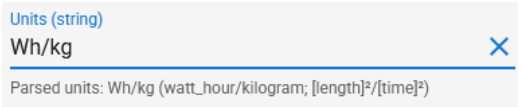
Output values can have different types:

- Numerical (`float`, `int`)
- Quantity: value + units
- NaN for a failed evaluation

Numpy values are also supported.

Units

ADORE uses Pint to parse its units. Enter your units in the "Units" field of a QOI in the GUI to confirm they are correct.



Output

The output is a dictionary mapping architecture QOIs to their values.

ADORE
https://adore.mbse-env.com/docs/

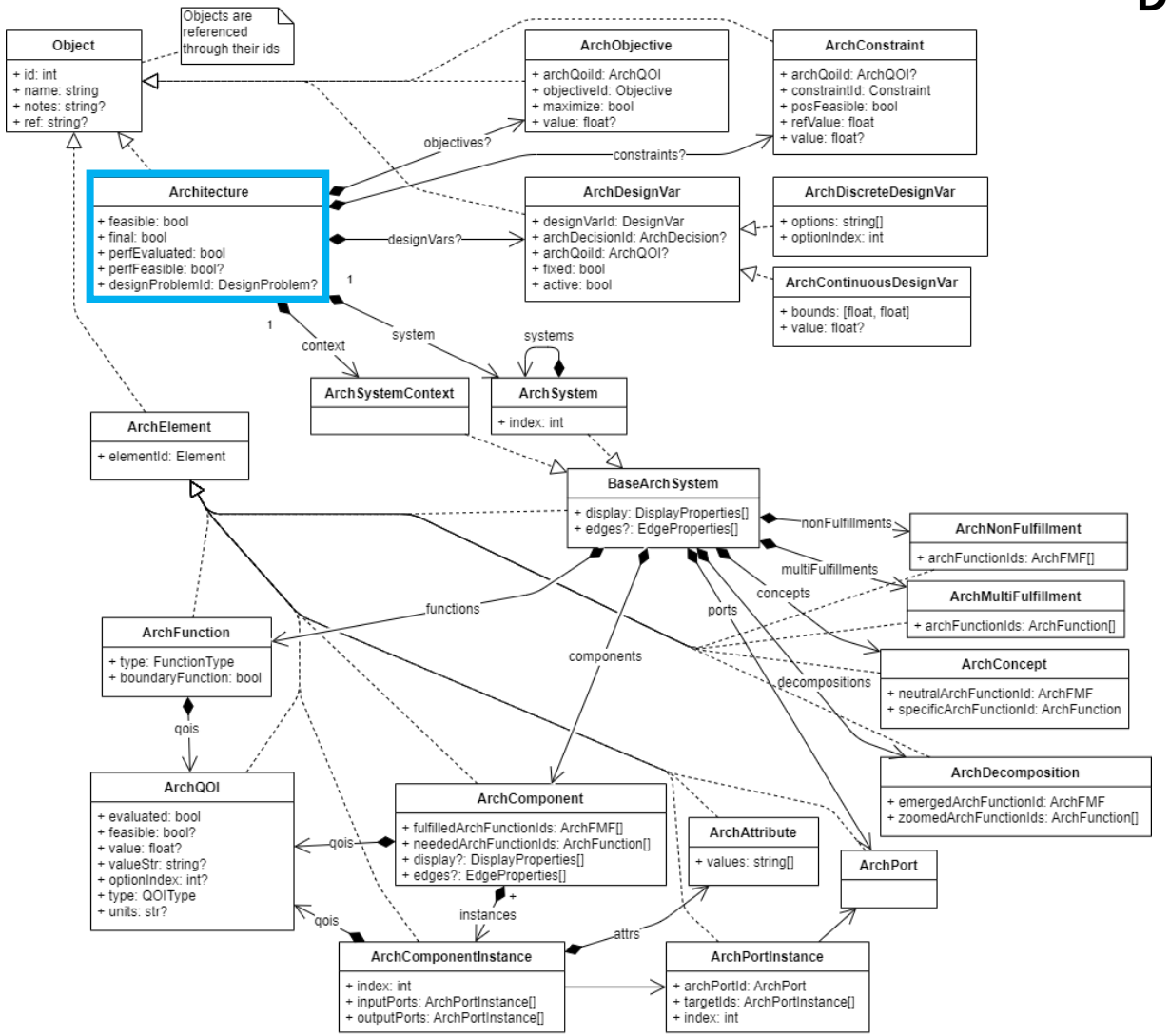
Data Model: The `Architecture`

- `Architecture` represents the architecture instance to be evaluated
  - Design variables, objectives, constraints
  - Top-level system and subsystems
  - System elements

- Type hinting is supported

```
def _evaluate(self, architecture: Architecture, arch_qois: List[ArchQOI], **_) -> Dict[ArchQOI, float]:
    architecture.system.co
```

```
components    ArchSystem
concepts      ArchSystem
decompositions ArchSystem
Ctrl+Down and Ctrl+Up will move caret down and up in the editor  Next Tip
```



Executing the Optimization Problem

```python
# Instantiate your evaluator from your ADORE model
evaluator = MyPythonEvaluator.from_file('model.adore‘)

# Connect to SBArchOpt and pymoo for optimization
from sb_arch_opt.algo.arch_sbo import get_arch_sbo_gp
from pymoo.optimize import minimize

problem = evaluator.get_arch_opt_problem()
algorithm = get_arch_sbo_gp(problem, init_size=50, results_folder='results‘)
result = minimize(problem, algorithm, termination=('n_eval', 100), verbose=True)

# Save the resulting ADORE file and a CSV export
evaluator.to_file(results.adore')
evaluator.save_results_csv('results.csv')
```

Setup and run

Create problem, select and configure the optimization algorithm, and execute the optimization.

SBArchOpt

https://sbarchopt.readthedocs.io/

SBArchOpt

Refer to the SBArchOpt documentation for more details about available optimization algorithms and executing optimization problems (including restart).

Exercise: Extract Data from the `Architecture`

In `gnc_evaluation_api_exercise.py` implement `_get_element_types`:

- Output should be a list of element types, for example:
  ['A', 'A']
  ['B', 'C', 'C']
- Workshop materials also includes the data model as a diagram
- Tip: use a debugger to live-inspect the data

```python
@staticmethod
def _get_element_types(architecture: Architecture, element_ref: str) -> List[str]:
    """Parses the Architecture to find the amount and types of the provided element, matching by name"""

    types = []

    # TODO implement code here:
    #  - Loop over system components
    #  - Match components by ref
    #  - For each component, loop over instances
    #  - For each instance, get the type attribute value (and add to the list)

    return types
```

```python
def _evaluate(self, architecture: Architecture, arch_qois: List[ArchQOI], **_) -> Dict[ArchQOI, float]:
    # Get element types
    sensor_types = self._get_element_types(architecture, 'sensor')
    computer_types = self._get_element_types(architecture, 'computer')
    actuator_types = self._get_element_types(architecture, 'actuator')
    if len(actuator_types) == 0:  # Check if this model includes actuators
        actuator_types = None
```

- When done, execute the script and inspect generated output
  - `gnc_evaluation_api.csv`: summary of all architectures
  - `gnc_evaluation_api.adore`: ADORE model containing the architectures, you can open it in the GUI to view them

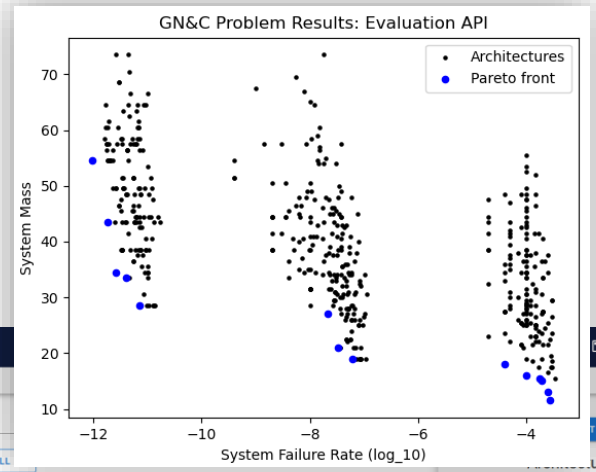
Exercise: Extract Data from the `Architecture`

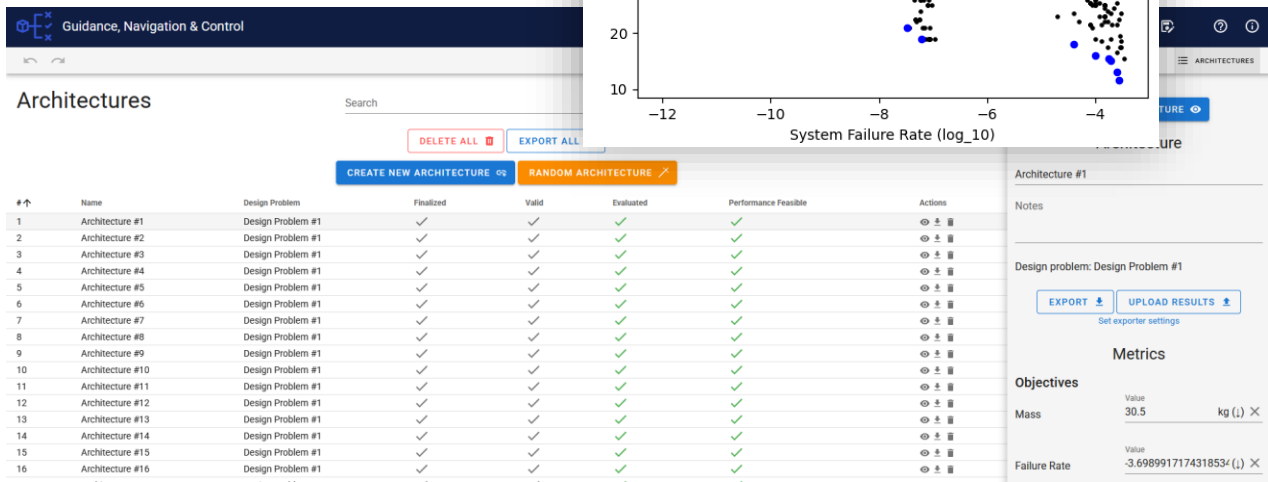
In `gnc_evaluation_api_exercise.py` implement `_get_element_types`:

- Output should be a list of element types, for example:
  ['A', 'A']
  ['B', 'C', 'C']
- Workshop materials also includes the data model as a diagram
- Tip: use a debugger to live-inspect the data

- When done, execute the script and inspect generated output
  - `gnc_evaluation_api.csv`: summary of all architectures
  - `gnc_evaluation_api.adore`: ADORE model containing the architectures, you can open it in the GUI to view them

```python
@staticmethod
def _get_element_types(architecture: Architecture, element_ref: str) -> List[str]:
    """Parses the Architecture to find the amount and types of the provided element, matching by name"""

    types = []
    for component in architecture.system.components:
        # Match by component name
        if component.ref == element_ref:

            # Loop over component instances
            for instance in component.instances:

                # Get type from the attribute value
                element_type = instance.attributes[0].values[0]
                assert element_type in ['A', 'B', 'C']

                types.append(element_type)
    return types
```





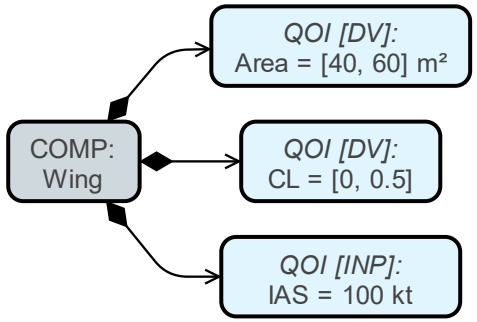
```python
from typing import *
from dataclasses import dataclass
import adore as ad

@dataclass
class Wing:
    area: float
    cl: float
    ias: float

class MyClassFactoryEvaluator(ad.ClassFactoryApiEvaluator):
    """Problem-specific Python-based Class Factory Evaluator."""

    def get_class_factories(self) -> List[ad.ClassFactory]:
        return [ad.ClassFactory(
            el=ad.ExternalComponentDef(name='Wing', auto=True),
            cls=Wing,
            props={
                'area': ad.ExternalQOIDef(name='Area', auto=True, units='m^2'),
                'cl': ad.ExternalQOIDef(name='Lift coefficient', auto_match_pattern='CL'),
                'ias': ad.ExternalQOIDef(name='IAS', auto=True, units='kts'),
            },
        )]

    def _evaluate(self, architecture, arch_qois, **_):
        # Get Wing objects from the architecture
        wings = self.instantiate(
            architecture, factories=self.get_factories_by_el_name('Wing'))
```

DLR

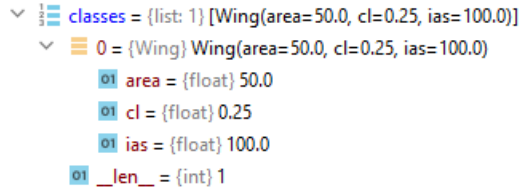
ADORE

Class Factory Evaluator (CFE)

The Class Factory Evaluator (CFE)




```python
from typing import *
from dataclasses import dataclass
import adore as ad

@dataclass
class Wing:
    area: float
    cl: float
    ias: float

class MyClassFactoryEvaluator(ad.ClassFactoryApiEvaluator):
    """Problem-specific Python-based Class Factory Evaluator."""

    def get_class_factories(self) -> List[ad.ClassFactory]:
        return [ad.ClassFactory(
            el=ad.ExternalComponentDef(name='Wing', auto=True),
            cls=Wing,
            props={
                'area': ad.ExternalQOIDef(
                    name='Area', auto=True, units='m^2'),
                'cl': ad.ExternalQOIDef(
                    name='Lift coefficient', auto_match_pattern='CL'),
                'ias': ad.ExternalQOIDef(
                    name='IAS', auto=True, units='kts'),
            },
        )]

    def _evaluate(self, architecture, arch_qois, **_):
        # Get Wing objects from the architecture
        wings = self.instantiate(
            architecture, factories=self.get_factories_by_el_name('Wing'))
        ...
```



Class factories

Rules for linking to ADORE elements and instantiating objects for each occurrence in an architecture instance.

Instantiation

Instantiate objects in the `_evaluate` function.

Defining Class Factories

1. Assumption: the evaluation framework is available and its input is defined using Python classes

2. Create an evaluator inheriting from `ClassFactoryApiEvaluator`

3. Define `ClassFactory`'s
   - Note: values from QOIs are normally of **float** type; however they will be **string**s for:
     - Input parameter (string)
     - Design variable (discrete)

4. Define a `MetricsFactory`

```python
from dataclasses import dataclass
from adore.optimization.api.factory_evaluator import *
from adore.api.schema import *
```

```python
@dataclass
class Wing:
    area: float  # m2
    cl: float
    ias: float  # kts

    def calc_lift(self):
        ias_ms = self.ias/1.94384
        rho0 = 1.225  # kg/m3
        return self.cl * .5*rho0*ias_ms**2 * self.area
```

"Evaluation framework"
Inputs
Calculations

```python
class WingClassFactoryEvaluator(ClassFactoryApiEvaluator):

    @staticmethod
    def get_class_factories() -> List[ClassFactory]:
        return [
            ClassFactory(
                el=ExternalComponentDef(name='Wing', n_inst=[1]),
                cls=Wing,
                props={
                    'area': ExternalQOIDef(name='Wing Area', qoi_type=QOIType.DESIGN_VAR, bounds=(40., 60.)),
                    'cl': ExternalQOIDef(name='Lift Coefficient', qoi_type=QOIType.DESIGN_VAR, bounds=(0., .5)),
                    'ias': ExternalQOIDef(name='Indicated Airspeed', qoi_type=QOIType.INPUT_PARAM, value=100.),
                },
            )
        ]

    @staticmethod
    def get_metrics_factory() -> MetricsFactory:
        return MetricsFactory(metrics={
            'lift': ExternalQOIDef(name='Lift', qoi_type=QOIType.CONSTRAINT, ref_value=15000*9.81, pos_better=True),
        })
```

`ClassFactory`
Defines a component
With properties (QOIs)

`MetricsFactory`
Defines outputs (QOIs)

Automatically Linking Elements

1. Factories can be automatically linked to design space elements using **auto match patterns**
   - Other links can still be created manually in the GUI

2. Patterns are applied on the *name* and *ref* fields of design space elements, and can be defined as:
   - Wildcard patterns (case-insensitive): `*` or `?` mean any or 1 character, respectively
   - Regular expressions: `/expr/` (or `/expr/i` for case-insensitive checking)
   - Same name as external element def: `auto=True`

3. Linking can be done from within Python (no need to switch between Python and the GUI)
   - After instantiating the evaluator, use the `update_external_databases` function to add the database and auto-match elements
   - Save to file to inspect the project in the GUI
   - Check console output to check matching was correct

```python
class WingClassAutoFactoryEvaluator(ClassFactoryApiEvaluator):

    @staticmethod
    def get_class_factories() -> List[ClassFactory]:
        return [
            ClassFactory(
                el=ExternalComponentDef(name='Wing', n_inst=[1], auto_match_pattern='/W.*/'),
                cls=Wing,
                props={
                    'area': ExternalQOIDef(
                        name='Wing Area', qoi_type=QOIType.DESIGN_VAR, bounds=(40., 60.), auto_match_pattern='area'),
                    'cl': ExternalQOIDef(
                        name='Lift Coefficient', qoi_type=QOIType.DESIGN_VAR, bounds=(0., .5),
                        auto_match_pattern=['cl', 'L* Coefficient']),
                    'ias': ExternalQOIDef(
                        name='Indicated Airspeed', qoi_type=QOIType.INPUT_PARAM, value=100., auto_match_pattern='i?s'),
                },
            )
        ]

    @staticmethod
    def get_metrics_factory() -> MetricsFactory:
        return MetricsFactory(metrics={
            'lift': ExternalQOIDef(
                name='Lift', qoi_type=QOIType.CONSTRAINT, ref_value=15000*9.81, pos_better=True, auto=True),
        })
```

```python
if __name__ == '__main__':
    # 1) Instantiate evaluator
    evaluator = WingClassAutoFactoryEvaluator.from_file('Class_Factory_Evaluator_Unlinked.adore')

    # 2) Auto match elements
    evaluator.update_external_database()
    evaluator.to_file('auto_linked.adore')
```

```
Linking report: design space elements --> external element definitions
System.Wing --> <AUTO> Wing
System.Wing.Area --> <AUTO> Wing Area (input)
System.Wing.CL --> <AUTO> Lift Coefficient (input)
System.Wing.IAS --> <AUTO> Indicated Airspeed (input)
System.Produce Lift.Lift --> <AUTO> Lift (output)
```

Exercise: Link Factories to Model Elements

- In `gnc_class_factory_evaluator_exercise.py` add a class factory for the actuator component
  - No need to track connection targets
- Run the script to verify connections
  - Check that “System.Actuator” is linked to the new factory
- Open `gnc_cfe_linked.adore` in the GUI to check created links (hint: look under the “External” tab)

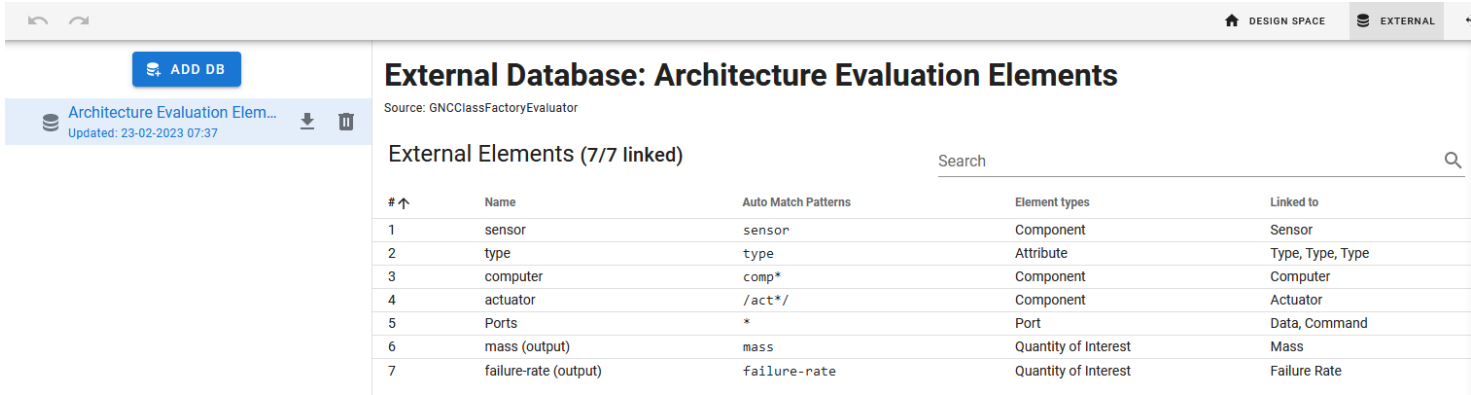
```python
@staticmethod
def get_class_factories() -> List[ClassFactory]:
    # Define an external element for the type attribute (auto matched by name)
    type_attr = ExternalAttributeDef(name='type', auto=True)

    # Define an external element for the ports
    port_def = ExternalPortDef(name='Ports', auto_match_pattern='*')

    return [
        ClassFactory(  # Class factory for the Sensor (auto matched by name)
            el=ExternalComponentDef(name='sensor', auto=True),
            cls=Sensor,
            props={
                'type': type_attr,  # Set the type property to the linked attribute value
                'element': SpecialValue.El,  # Architecture element to track the instance index
                'targets': ConnectionValue(conn_target_def=port_def, input_conn=False),  # Port connection targets
            },
        ),
        ClassFactory(  # Class factory for the Computer (auto matched by pattern)
            el=ExternalComponentDef(name='computer', auto_match_pattern='comp*'),
            cls=Computer,
            props={
                'type': type_attr,  # Set the type property to the linked attribute value
                'element': SpecialValue.El,  # Architecture element to track the instance index
                'targets': ConnectionValue(conn_target_def=port_def, input_conn=False),  # Port connection targets
            },
        ),

        # TODO add class factory for the actuator (note: no need for tracking connection targets)

        ClassFactory(el=port_def, cls=Port),  # Class factory for linking to all the ports
    ]
```

Exercise: Link Factories to Model Elements

- In `gnc_class_factory_evaluator_exercise.py` add a class factory for the actuator component
  - No need to track connection targets
- Run the script to verify connections
  - Check that “System.Actuator” is linked to the new factory
- Open `gnc_cfe_linked.adore` in the GUI to check created links (hint: look under the “External” tab)

```
    ClassFactory(  # Class factory for the Computer (auto matched by pattern)
        el=ExternalComponentDef(name='computer', auto_match_pattern='comp*'),
        cls=Computer,
        props={
            'type': type_attr,  # Set the type property to the linked attribute value
            'element': SpecialValue.El,  # Architecture element to track the instance index
            'targets': ConnectionValue(conn_target_def=port_def, input_conn=False),  # Port connection targets
        },
    ),
    ClassFactory(  # Class factory for the Actuator (auto matched by regex)
        el=ExternalComponentDef(name='actuator', auto_match_pattern='/act*/'),
        cls=Actuator,
        props={
            'type': type_attr,  # Set the type property to the linked attribute value
            'element': SpecialValue.El,  # Architecture element to track the instance index
        },
    ),
    ClassFactory(el=port_def, cls=Port),  # Class factory for linking to all the ports
]
```

```
Linking report: design space elements --> external element definitions
System.Sensor --> <AUTO> sensor
System.Sensor.Type --> <AUTO> type
System.Computer --> <AUTO> computer
System.Computer.Type --> <AUTO> type
System.Actuator --> <AUTO> actuator
System.Actuator.Type --> <AUTO> type
System.Provide GNC.Mass --> <AUTO> mass (output)
```



#	Name	Auto Match Patterns	Element types	Linked to
1	sensor	sensor	Component	Sensor
2	type	type	Attribute	Type, Type, Type
3	computer	comp*	Component	Computer
4	actuator	/act*/	Component	Actuator
5	Ports	*	Port	Data, Command
6	mass (output)	mass	Quantity of Interest	Mass
7	failure-rate (output)	failure-rate	Quantity of Interest	Failure Rate

Implementing the Evaluation Function

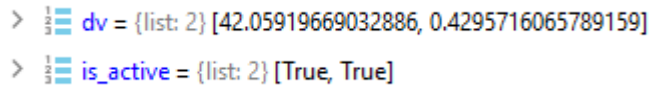
1. Extend the `_evaluate` function
   - Use `instantiate` and `process_results` as needed
   - You can select specific factories to be instantiated:
     `wings = self.instantiate(architecture, factories=[self._class_factories[0]])`
   - Or select the factories to be instantiated by name:
     `wings = self.instantiate(architecture, factories="Wing")`

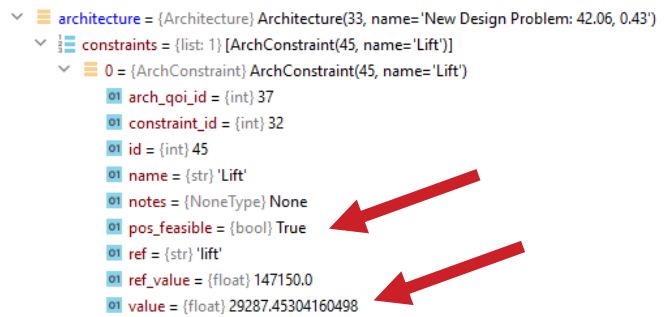
```python
def _evaluate(self, architecture: Architecture, arch_qois: List[ArchQOI]) -> Dict[ArchQOI, float]:
    # Instantiate classes: we expect one instance of Wing
    objects = self.instantiate(architecture)
    wing = objects[0]
    assert isinstance(wing, Wing)

    # Perform calculations
    results = {
        'lift': wing.calc_lift(),
    }

    # Interpret results so that ADORE can process them
    return self.process_results(architecture, arch_qois, results)
```

1. Run architecture evaluation as it would be in an optimization loop
   - Architecture can also be a manually generated architecture

```python
evaluator = WingClassFactoryEvaluator.from_file('Class_Factory_Evaluator_Example.adore')
architecture, dv, is_active = evaluator.get_architecture(evaluator.get_random_design_vector())
evaluator.evaluate(architecture)
```

3. Check results





ADORE
https://adore.mbse-env.com/docs/

Exercise: Class Factory Evaluation

- In `gnc_class_factory_evaluator_exercise.py` implement instantiation and results processing in the evaluation function

- Remove the `exit()` near the end of the file
- When done, execute the script and inspect generated output
  - A plot is created showing the design space
  - `gnc_class_factory_evaluator.csv`: summary of all architectures
  - `gnc_class_factory_evaluator.adore`: ADORE model containing the architectures, you can open it in the GUI to view them

```python
def _evaluate(self, architecture: Architecture, arch_qois: List[ArchQOI], **kwargs) -> Dict[ArchQOI, float]:

    sensors = []
    computers = []
    actuators = []
    # TODO add code above to instantiate sensor, computer, and actuator classes:
    #  - Use the self.instantiate function
    #  - Use the factories argument to only instantiate specific classes

    if len(actuators) == 0:
        actuators = None


    # Calculate and match metrics
    metrics = {
        'mass': GNCCalculator.calc_mass(sensor_types, computer_types, actuator_types),
        'failure_rate': GNCCalculator.calc_failure_rate(
            sensor_types, computer_types, sensor_computer_conns, actuator_types, computer_actuator_conns),
    }

    # TODO add a return statement using the self.process_results function to match metrics to QOIs
```

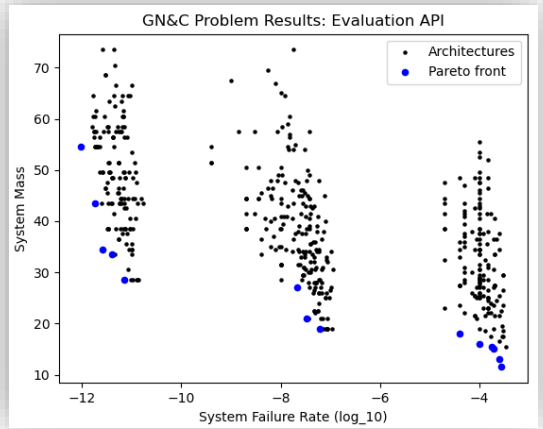
Exercise: Class Factory Evaluation

- In `gnc_class_factory_evaluator_exercise.py` implement instantiation and results processing in the evaluation function

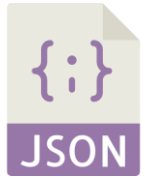
```python
def _evaluate(self, architecture: Architecture, arch_qois: List[ArchQOI], **kwargs) -> Dict[ArchQOI, float]:
    # Instantiate classes
    sensors = self.instantiate(architecture, factories='sensor')
    computers = self.instantiate(architecture, factories='computer')
    actuators = self.instantiate(architecture, factories='actuator')
    if len(actuators) == 0:
        actuators = None
```

- Remove the `exit()` near the end of the file
- When done, execute the script and inspect generated output
  - A plot is created showing the design space
  - `gnc_class_factory_evaluator.csv`: summary of all architectures
  - `gnc_class_factory_evaluator.adore`: ADORE model containing the architectures, you can open it in the GUI to view them

```python
    # Calculate and match metrics
    metrics = {
        'mass': GNCCalculator.calc_mass(sensor_types, computer_types, actuator_types),
        'failure_rate': GNCCalculator.calc_failure_rate(
            sensor_types, computer_types, sensor_computer_conns, actuator_types, computer_actuator_conns),
    }

    return self.process_results(architecture, arch_qois, metrics)
```



```
{
    "projectName": "Guidance, Navigation & Control",
    "architecture": {"id": 652...},
    "outputs": [
        {
            "archQoiId": 658,
            "name": "Mass",
            "ref": "mass",
            "type": "OBJECTIVE",
            "value": null
        },
        {
            "archQoiId": 659,
            "name": "Failure Rate",
            "ref": "failure-rate",
            "type": "OBJECTIVE",
            "value": null
        }
    ],
    "designSpace": {...},
    "designProblem": {"id": 65...}
}
```

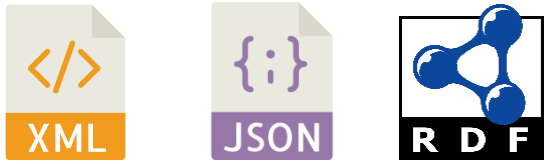


ADORE

File-based Evaluation

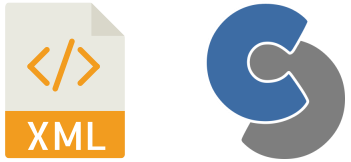
File-based Evaluation

- Direct serialization of the architecture instance

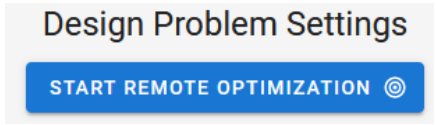


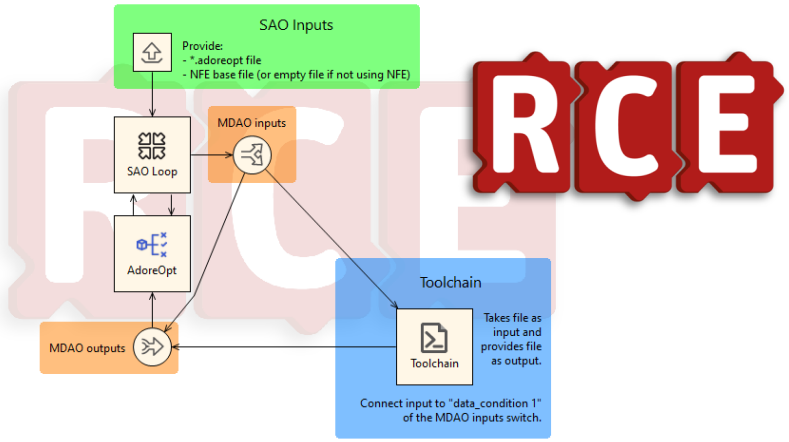

```
{
    "projectName": "Guidance, Navigation & Control",
    "architecture": {"id": 103...},
    "outputs": [
        {
            "archQoiId": 109,
            "name": "Mass",
            "ref": "mass",
            "type": "OBJECTIVE",
            "units": "kg",
            "value": null
        },
        {
            "archQoiId": 110,
            "name": "Failure Rate",
            "ref": "failure-rate",
            "type": "OBJECTIVE",
            "units": null,
            "value": null
        }
    ],
    "designSpace": [ 6 elements… ],
    "designProblem": {"id": 67...}
}
```


- Rule-based construction of the evaluation input file
  - XML Node Factory Evaluator
  - CPACS



- Supports remote optimization





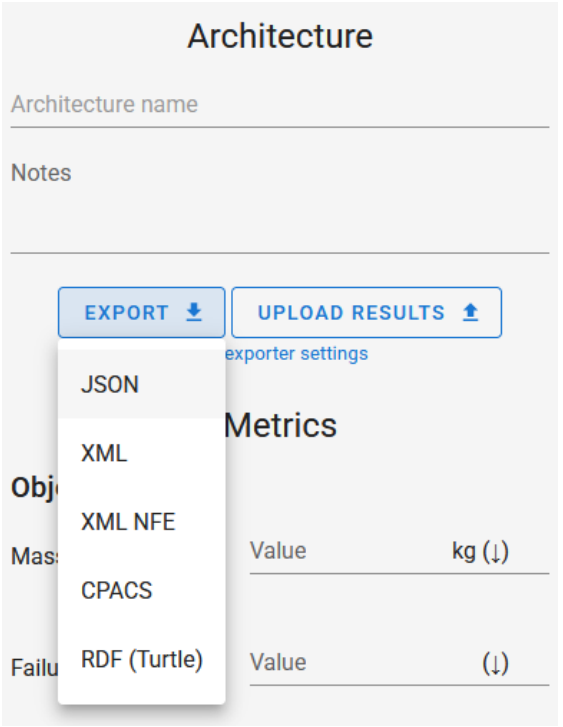
Direct Serialization Formats

- Each file contains:
  - A serialization of the design space, design problem and architecture instance
  - Placeholders to evaluation outputs

- Formats

- Testing in the GUI
  - Select an architecture and export it
  - Add output values to the file
  - Upload the resulting file and observe the metrics being updated


```
{
    "projectName": "Guidance, Navigation & Control",
    "architecture": {"id": 103...},
    "outputs": [
        {
            "archQoiId": 109,
            "name": "Mass",
            "ref": "mass",
            "type": "OBJECTIVE",
            "units": "kg",
            "value": null
        },
        {
            "archQoiId": 110,
            "name": "Failure Rate",
            "ref": "failure-rate",
            "type": "OBJECTIVE",
            "units": null,
            "value": null
        }
    ],
    "designSpace": [ 6 elements… ],
    "designProblem": {"id": 67...}
}
```

ADORE
https://adore.mbse-env.com/docs/

Exercise: File-Based Evaluation

- Manually create an architecture and export it to JSON
- In `gnc_file_evaluation_exercise.py` implement the logic in `_get_element_types`
  - Output should be a list of element types, for example:

    [‘A’, ‘A’]

    [‘B’, ‘C’, ‘C’]
  - Use the previously exported file as a data model reference
  - Workshop materials also includes the data model as a diagram
- When done, execute the script and inspect generated output
  - A plot is created showing the design space

```python
@staticmethod
def _get_element_types(data, element_ref) -> List[str]:
    types = []

    # TODO implement code here:
    #  - Loop over system components
    #  - Match components by ref
    #  - For each component, loop over instances
    #  - For each instance, get the type attribute value (and add to the list)

    return types
```

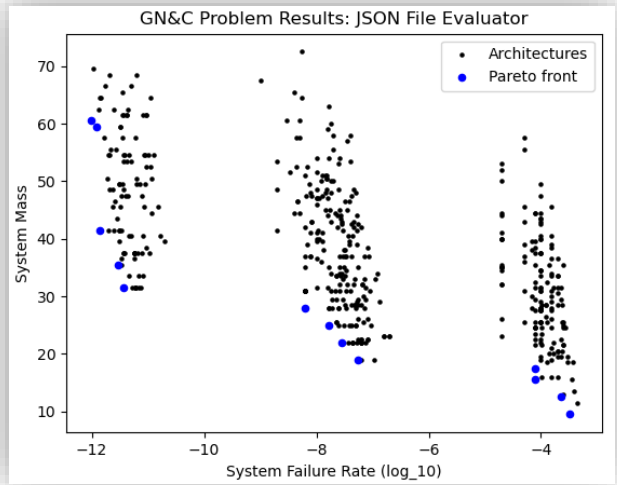
Exercise: File-Based Evaluation

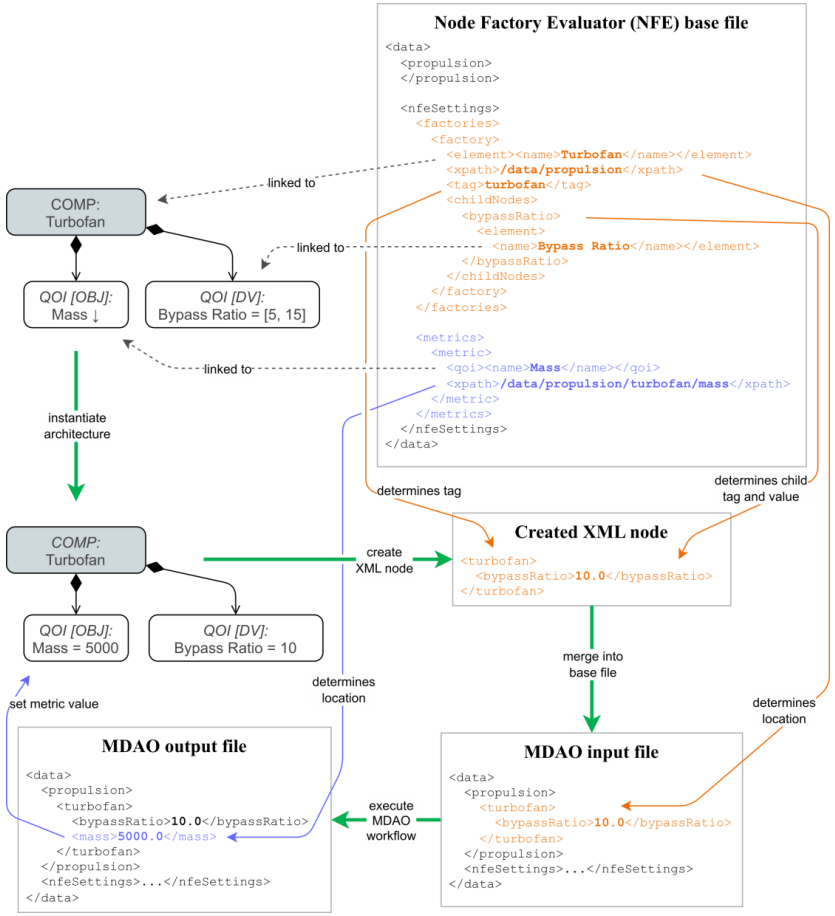
- Manually create an architecture and export it to JSON
- In `gnc_file_evaluation_exercise.py` implement the logic in `_get_element_types`
  - Output should be a list of element types, for example:

    [‘A’, ‘A’]

    [‘B’, ‘C’, ‘C’]
  - Use the previously exported file as a data model reference
  - Workshop materials also includes the data model as a diagram
- When done, execute the script and inspect generated output
  - A plot is created showing the design space

```python
@staticmethod
def _get_element_types(data, element_ref) -> List[str]:
    types = []
    for component_data in data['architecture']['system']['components']:
        # Match by component name
        if component_data['ref'] == element_ref:

            # Loop over instances
            for instance_data in component_data['instances']:

                # Get type from attribute value
                element_type = instance_data['attributes'][0]['values'][0]
                assert element_type in ['A', 'B', 'C']

                types.append(element_type)
    return types
```





DLR

ADORE

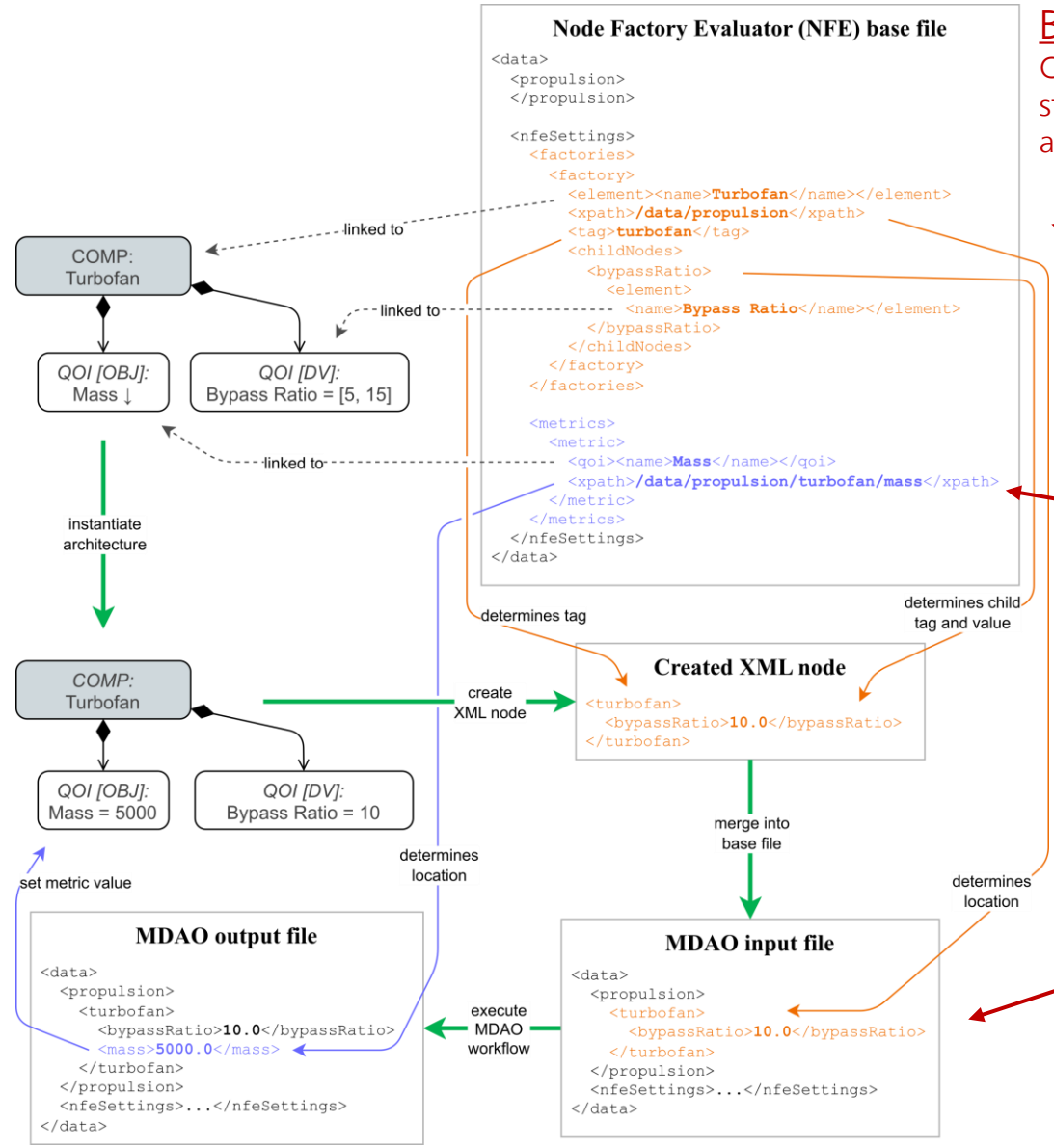
Node Factory Evaluator (NFE)

XML Node Factory Evaluator

Rule-based mechanism for creating nodes from ADORE architecture elements, and reading metrics from the output file.

Analogous to the CFE for Python.

ADORE needs to know where the factories are defined by providing the settings xpath: `/data/nfeSettings`



Base file

Contains static XML structure, as well as node and metric factories.

Node factories

Rules for linking to ADORE elements and creating *XML nodes* for each occurrence in an architecture instance.

Metric factories

Tells ADORE where to load the output metrics from the XML file.

Output file

Metric QOIs read from output file.

Input file

Input file containing nodes from ADORE elements.

Element Linking Configuration

Name
Element name in the external database.

Auto-linking
Match by name (`auto`) or by pattern.

Types
Optionally restrict linkable element types.

```
<factories>
    <factory>
        <!-- The architecture element this factory is associated to -->
        <element>
            <!-- Element name, may be used for matching -->
            <name>Element Name</name>

            <!--
            The factory should be associated to one or more ADORE architecture elements, done by
            matching to name/ref of elements.
            Matching rules are specified using <match> tags, which can contain one of:
            - "auto": match by name provided in <name>
            - a pattern including wildcards
              (* = any characters, ? = 0 or 1 character, [..] = optional characters)
            - a regular expression, e.g. /c.*/i
            Multiple rules can be specified.
            -->
            <match>auto</match><!-- Match "Element Name" -->
            <match>El*</match><!-- Match anything starting with "El" (case insensitive) -->
            <match>/[Ee]l[^N]*N.+/</match><!-- Match by regex -->

            <!-- Optionally restrict to element type, one of (case insensitive):
                 COMPONENT, FUNCTION, QOI, ATTRIBUTE, PORT -->
            <type>component</type>
        </element>
```

Node Creation Settings

```
<factories>
    <factory>
        <element>…</element>

        <!--
        The xpath at which nodes should be created.
        Following template parameters are available:

          {INDEX}        : 0-index of the linked architecture element (empty string if not applicable)
          {COMP_INDEX}   : 0-index of the component instance containing the element (if applicable)
          {SYS_INDEX}    : 0-index of the system containing the architecture element
          {SYS_INDEX_n}  : 0-index of the containing system n levels up (SYS_INDEX_0 is equivalent to
                           SYS_INDEX, SYS_INDEX_1 is the system above)

          {INDEX1}       : 1-index of the linked architecture element (empty string if not applicable)
          {COMP_INDEX1}  : 1-index of the component instance containing the element (if applicable)
          {SYS_INDEX1}   : 1-index of the system containing the architecture element
          {SYS_INDEX1_n} : 1-index of the containing system n levels up (SYS_INDEX_0 is equivalent to
                           SYS_INDEX, SYS_INDEX_1 is the system above)

        Note: "0-index" means counting starts at zero, "1-index" counting starts at 1
        -->
        <xpath>/data/elements</xpath>

        <!-- The tag of the created nodes.
             Supports the same template parameters as <xpath>. -->
        <tag>createdNode</tag>
```

Location

xpath where the new node should be created at. Optionally including template parameters.

Tag

Tag of the created node.

Node Contents

Contents

Attributes, child nodes (recursive) and the node value can be specified.

Value types include: literal, value from another architecture element, special value, or a connection value.

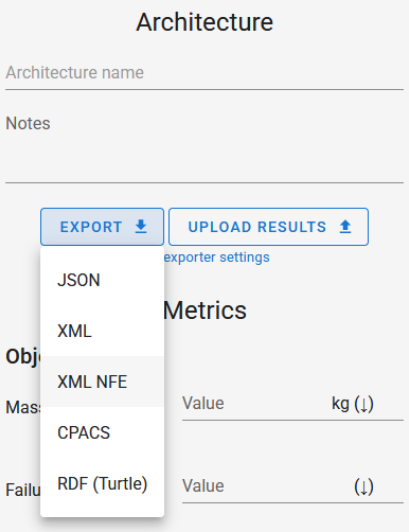
```
<!--
The following valueDef types are possible:
- Literal: just place the text there and it will be used as value for the attribute
- Value from architecture element: use <element> (see below)
- Special value: use <special> (see below)
- Connection value: use <connection> (see below)
-->
<attributes>
    <key>literal value (string)</key><!-- Literal value -->

    <key><element><!-- Get a value associated to an architecture element, e.g. a QOI -->
        <name>Quantity</name>
        <match>auto</match>
        <type>QOI</type>
    </element></key>

    <!-- Get a value derived from the top-level factory element, one of:
         ID, NAME, VALUE, EL, INDEX -->
    <key><special>NAME</special></key>

    <!-- To track port connections -->
    <key><connection>
        <element>...</element><!-- Element matching the associated port -->
        <dir>input</dir><!-- Optionally restricting to input or output connections -->
    </connection></key>
</attributes>

<!-- Child nodes to add to the created nodes -->
<childNodes>
    <!-- Key-value pairs specified in the same way as attributes -->
    <key>valueDef</key>
</childNodes>

<!-- Directly specify the node value, specified in the same way as attributes -->
<value>valueDef</value>
</factory>
```

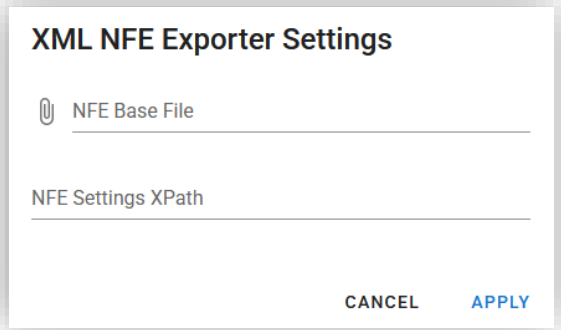

https://adore.mbse-env.com/docs/

Developing NFE Factories

In the ADORE GUI

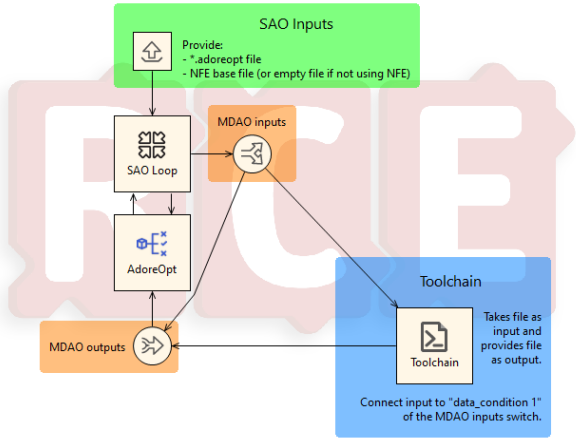
1. Export manually created architectures: "XML NFE"
   - Upload base file
   - Configure settings xpath
2. Add output values to exported file
3. "Upload results" to check if metrics are read correctly





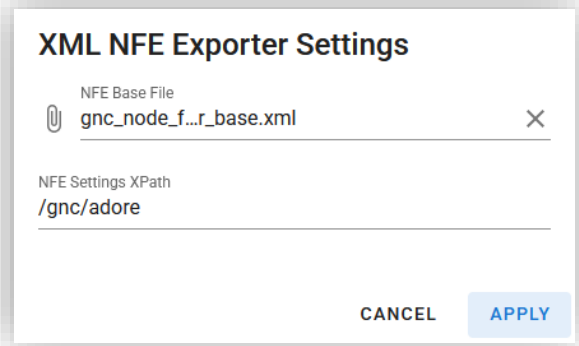
In the execution environment (e.g. RCE)

1. Manually create architectures in ADORE
2. Create the AdoreOpt file (see RFSAO slides)
3. In the base file, force the evaluation of a specific architecture using `archForceEval`
4. Execute the workflow and observe results



Exercise: Test Node Factory Evaluator

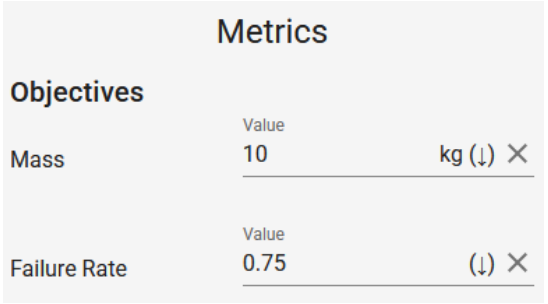
- Open `gnc.adore` and select an architecture instance
- Export to XML NFE:
  - `gnc_node_factory_evaluator_base.xml` as base file
  - `/gnc/adore` as settings path
- Inspect console output to see if the NFE behaves correctly
- Upload results: `Architecture_1_NFE.xml`
  - Observe metrics being loaded correctly



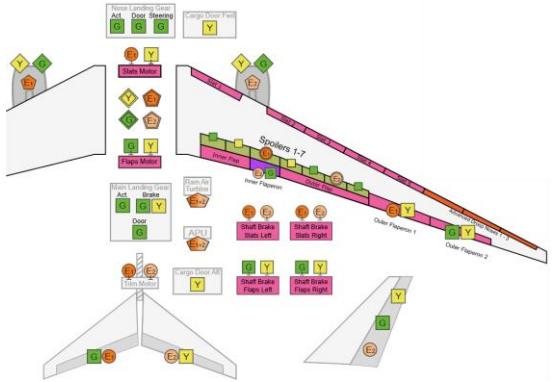

```
<gnc>
    <!-- ADORE node factory settings -->
    <adore>
        <factories>
            <factory><!-- Sensor factory -->
                <element>
                    <name>sensor</name>
                    <match>auto</match>
                </element>
                <xpath>/gnc/sensors</xpath>
                <tag>sensor</tag>

                <contents>
                    <type><element><!-- Type attribute -->
                        <name>type</name>
                        <match>auto</match>
                        <type>ATTRIBUTE</type>
                    </element></type>

                    <!-- Serialized architecture element -->
                    <!--<element><special>EL</special></element>-->

                    <targets><connection>
                        <element>
                            <name>port</name>
                            <match>*</match>
                        </element>
                        <dir>output</dir>
                    </connection></targets>
                </contents>
            </factory>
```

Exercise: Test Node Factory Evaluator

- Open `gnc.adore` and select an architecture instance
- Export to XML NFE:
  - `gnc_node_factory_evaluator_base.xml` as base file
  - `/gnc/adore` as settings path
- Inspect console output to see if the NFE behaves correctly
- Upload results: `Architecture_1_NFE.xml`
  - Observe metrics being loaded correctly




```xml
<gnc>
  <!-- Data model for GNC problem -->
  <sensors>
    <sensor>
      <type>A</type>
      <targets>Computer_0_1b4e62</targets>
    </sensor>
  </sensors>
  <computers>
    <computer uID="Computer_0_1b4e62">
      <type>A</type>
      <targets>Actuator_0_2ff4d6</targets>
    </computer>
  </computers>
  <actuators>
    <actuator uID="Actuator_0_2ff4d6">
      <type>A</type>
    </actuator>
  </actuators>
  <performance>
    <mass units="kg">10</mass>
    <failureRate>.75</failureRate>
  </performance>
```



ADORE

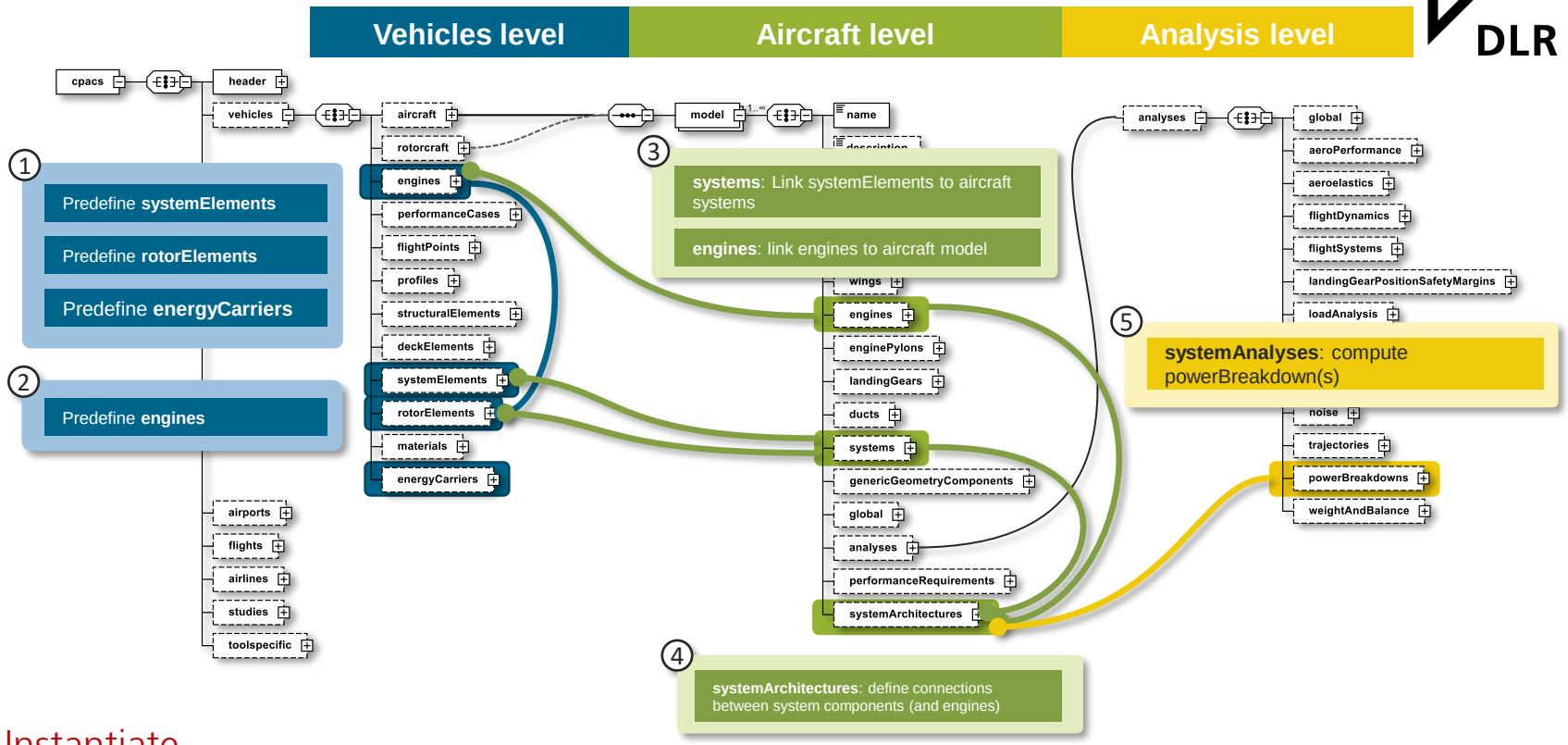
CPACS Interface

ADORE to CPACS Interface

CPACS

Common Parametric Aircraft Configuration Schema; https://cpacs.de/

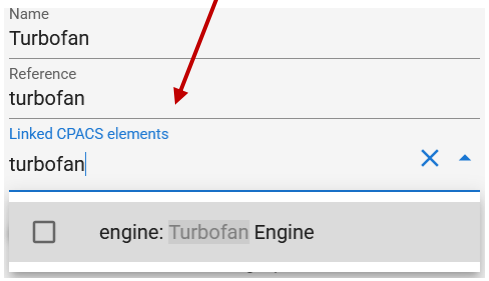
XML format for modeling aircraft. Includes onboard system architecture definitions.



Burschyk, T. et al. (2025) Introduction of a System Definition in the Common Parametric Aircraft Configuration Schema (CPACS). MDPI Aerospace, 12 (5). doi: 10.3390/aerospace12050373

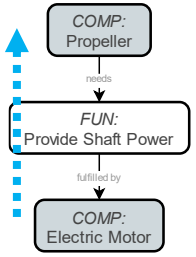
Link elements

Link ADORE elements to CPACS elements.



Instantiate

Automatically create CPACS architecture elements.



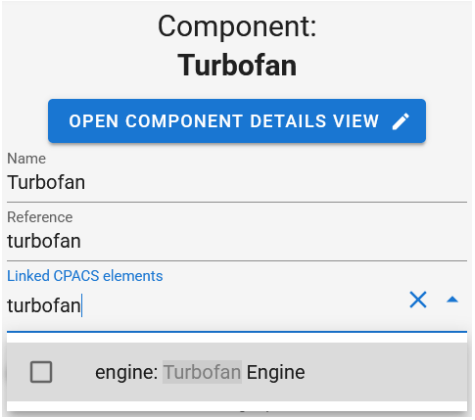

```
<systemArchitectures>
  <systemArchitecture>
    <connections>
      <connection uID="Electric_Motor_0_d6f0c9_to_Propeller_0_50d6d9">
        <source>
          <componentUID>Electric_Motor_0_d6f0c9</componentUID>
        </source>
        <target>
          <componentUID>Propeller_0_50d6d9</componentUID>
        </target>
      </connection>
```

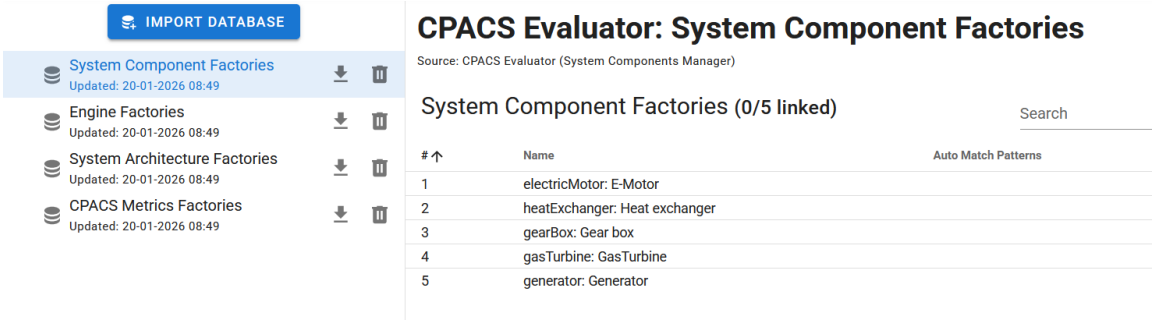
ADORE
https://adore.mbse-env.com/docs/

ADORE – CPACS Modeling Principles

- Define reusable elements in `/cpacs/vehicles/systemElements` (and `engines`)
- Optionally define explicit factories in ADORE toolspecific
- Import CPACS base file in ADORE: external → import database → CPACS
  - Can be repeated as many times as needed
- Link ADORE elements to CPACS factories/metrics

```
<cpacs>
    <vehicles>
        <aircraft>…</aircraft>
        <engines>
            <engine uID="engine">
                <name>Engine</name>
                <rotors></rotors>
            </engine>
        </engines>
        <systemElements>
            <electricMotors>
                <electricMotor uID="eMotor">
                    <name>E-Motor</name>
                    <geometry>
                        <frustum>
                            <lowerRadiusX>1</lowerRadiusX>
                            <height>2</height>
                            <upperRadiusX>1</upperRadiusX>
                        </frustum>
                    </geometry>
                </electricMotor>
            </electricMotors>
            <heatExchangers>
                <heatExchanger uID="hx">
                    <name>Heat exchanger</name>
                    <geometry>
                        <parallelepiped>
                            <length>1</length>
                            <width>1</width>
                            <height>1</height>
                        </parallelepiped>
                    </geometry>
                </heatExchanger>
            </heatExchangers>
```



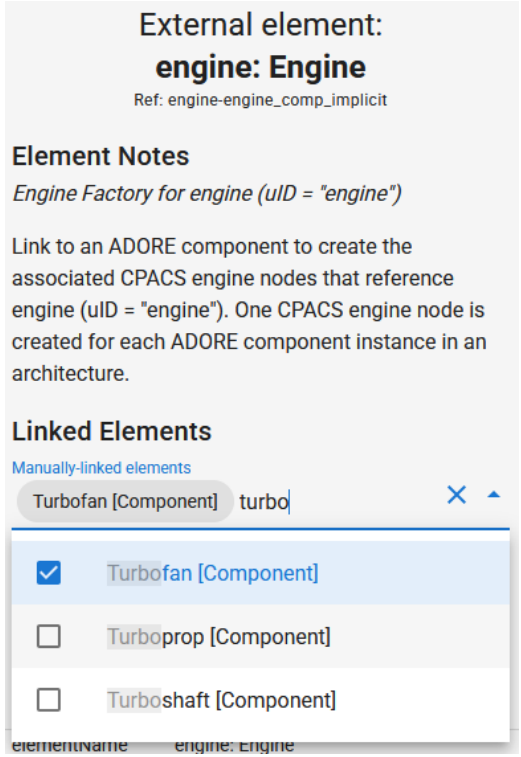


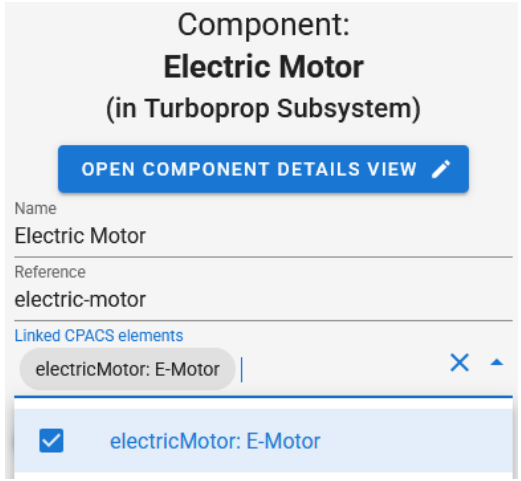
Exercise: Import and Manually Link Elements

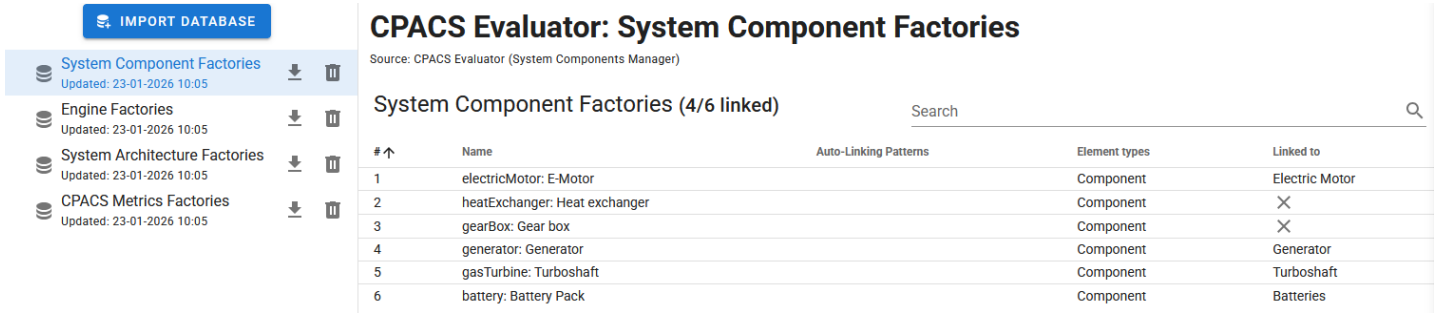
- Open `cpacs_evaluation_workshop.adore`
- Import `cpacs_example_reusable_elements.xml` as an external database (CPACS)
- Link components to CPACS elements
  - Engines
    - Turbofan
    - Turboprop
  - Components
    - Electric motor
    - Turboshaft
    - Batteries
    - Generator
- Link QOIs to CPACS metrics
  - Weight (mTOM)
  - Flight time
- View established connections in the external databases tab

Exercise: Import and Manually Link Elements

- Open `cpacs_evaluation_workshop.adore`
- Import `cpacs_example_reusable_elements.xml` as an external database (CPACS)
- Link components to CPACS elements
  - Engines
    - Turbofan
    - Turboprop
  - Components
    - Electric motor
    - Turboshaft
    - Batteries
    - Generator
- Link QOIs to CPACS metrics
  - Weight (mTOM)
  - Flight time
- View established connections in the external databases tab







ADORE
https://adore.mbse-env.com/docs/

Explicit Factories for Auto-Linking

- Defining factories in the ADORE toolspecific section allows more control over behavior
  - Auto-linking to ADORE elements
  - Modifying specific settings

- Generic factories
- CPACS-specific factories: components, engines, connectors

```
<componentFactory>
    <element><!-- ADORE element linking settings (COMPONENT) --></element>

    <!-- Optional genericSystem UID to create the connections in.
         If not provided, connections will be added to the first genericSystem -->
    <systemUID>uid</systemUID>

    <!-- Optional UID of a reusable system element -->
    <systemElementUID>systemElementUID</systemElementUID>
</componentFactory>

<engineFactory>
    <element><!-- ADORE element linking settings (COMPONENT) --></element>

    <!-- Optional referenced engine and parent UIDs -->
    <engineUID>systemElementUID</engineUID>
    <parentUID>systemElementUID</parentUID>
</engineFactory>
```

```
<cpacs>
    <toolspecific>
        <tool>
            <name>ADORE</name>
            <factories>...</factories>
            <metrics>...</metrics>
        </tool>
    </toolspecific>
</cpacs>

<factory>
    <!-- The architecture element this factory is associated to -->
    <element>
        <!-- Element name, may be used for matching -->
        <name>Element Name</name>

        <!-- The factory should be associated to one or more ADORE architecture elements,
             done by matching to name/ref of elements. -->
        <match>auto</match><!-- Match "Element Name" -->
        <match>El*</match><!-- Match anything starting with "El" (case insensitive) -->
        <match>/[Ee]l[^N]*N.+/</match><!-- Match by regex -->
    </element>
    ...
</factory>
```

Refer to `cpacs_nfe_format.xml` for more details.

Exercise: Explicit Factory

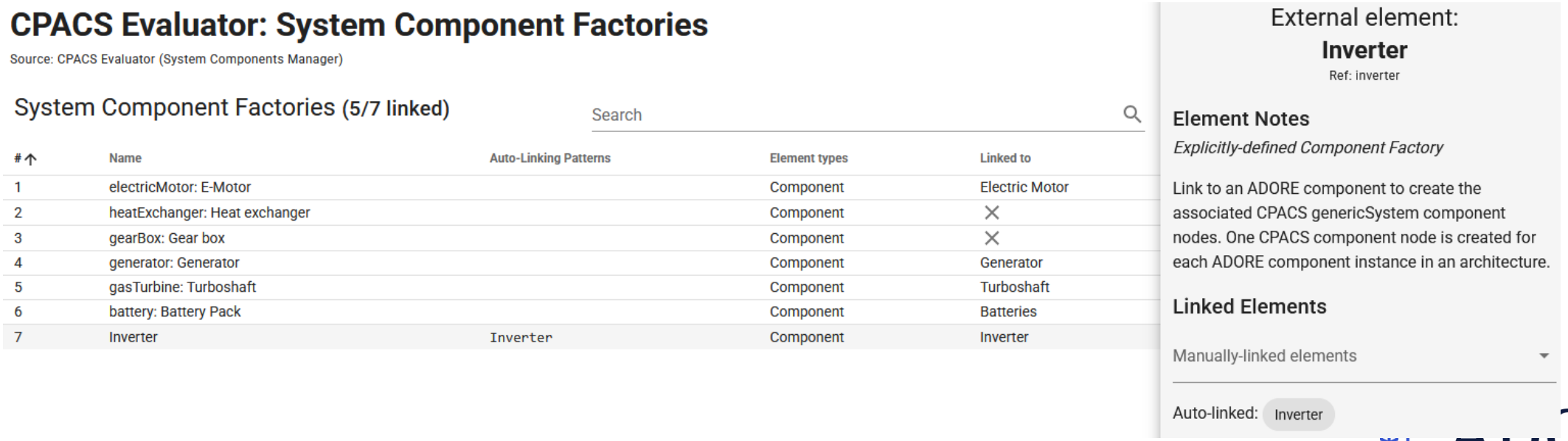
- Start from `cpacs_example_factory.xml`
- Define an explicit component factory to link to the “Inverter” component
- Once ready, import the CPACS file again as an external database
  - Observe the auto-linkined element

```
<cpacs>
    <toolspecific>
        <tool>
            <name>ADORE</name>
            <factories>...</factories>
            <metrics>...</metrics>
        </tool>
    </toolspecific>
</cpacs>

<factory>
    <!-- The architecture element this factory is associated to -->
    <element>
        <!-- Element name, may be used for matching -->
        <name>Element Name</name>

        <!-- The factory should be associated to one or more ADORE architecture elements,
             done by matching to name/ref of elements. -->
        <match>auto</match><!-- Match "Element Name" -->
        <match>El*</match><!-- Match anything starting with "El" (case insensitive) -->
        <match>/[Ee]l[^N]*N.+/</match><!-- Match by regex -->
    </element>
    ...
</factory>

<componentFactory>
    <element><!-- ADORE element linking settings (COMPONENT) --></element>

    <!-- Optional genericSystem UID to create the connections in.
         If not provided, connections will be added to the first genericSystem -->
    <systemUID>uid</systemUID>

    <!-- Optional UID of a reusable system element -->
    <systemElementUID>systemElementUID</systemElementUID>
</componentFactory>
```

Exercise: Explicit Factory

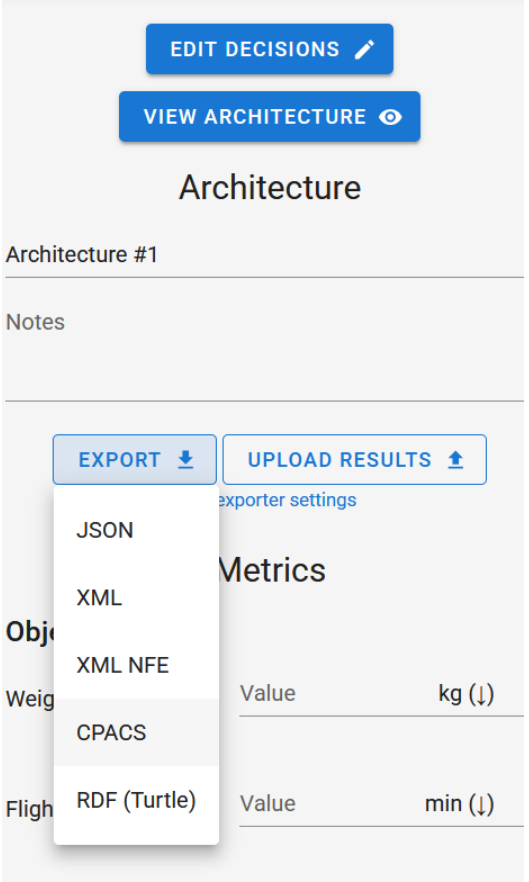
- Start from `cpacs_example_factory.xml`
- Define an explicit component factory to link to the “Inverter” component
- Once ready, import the CPACS file again as an external database
  - Observe the auto-linkined element

```
<cpacs>
    <toolspecific>
        <tool>
            <name>ADORE</name>
            <factories>
                <componentFactory>
                    <element>
                        <!-- Automatically link to the "Inverter" component -->
                        <name>Inverter</name>
                        <match>auto</match>
                    </element>
                </componentFactory>
            </factories>
        </tool>
    </toolspecific>
</cpacs>
```



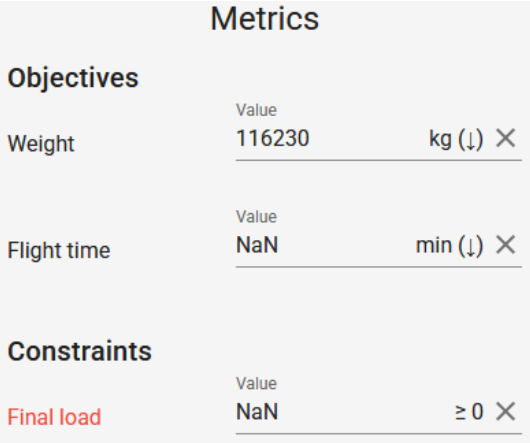
Exercise: Test the CPACS Interface

- Manually create an architecture
- Export the architecture to CPACS
  - Select the previously-prepared base file
- Inspect the created CPACS file
- Upload the created file as results
  - Observe how metrics are read



Exercise: Test the CPACS Interface

- Manually create an architecture
- Export the architecture to CPACS
  - Select the previously-prepared base file
- Inspect the created CPACS file
- Upload the created file as results
  - Observe how metrics are read



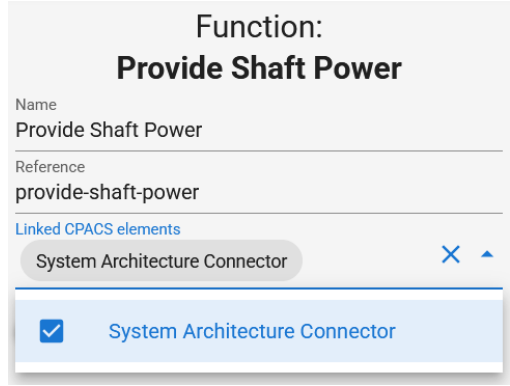

```
<cpacs>
  <vehicles>
    <aircraft>
      <model uID="exampleAircraft">
        <engines>
          <engine uID="Turboprop_0_1ba319">
            <engineUID>engine</engineUID>
            <name>Turboprop 0</name>
          </engine>
          <engine uID="Turboprop_0_956327">
            <engineUID>engine</engineUID>
            <name>Turboprop 0</name>
          </engine>
        </engines>
        <systems>
          <genericSystems>
            <genericSystem uID="mySystem">
              <name>Example system</name>
              <components>
                <component uID="Inverter_0_ffc962">
                  <name>Inverter 0</name>
                </component>
                <component uID="Electric_Motor_0_470397">
                  <systemElementUID>eMotor</systemElementUID>
                  <name>Electric Motor 0</name>
                </component>
                <component uID="Electric_Motor_0_c1f8c0">
                  <systemElementUID>eMotor</systemElementUID>
                  <name>Electric Motor 0</name>
                </component>
                <component uID="Batteries_0_6fbf49">
                  <systemElementUID>batteryPack</systemElementUID>
                  <name>Batteries 0</name>
                </component>
              </components>
            </genericSystem>
          </genericSystems>
        </systems>
      </model>
    </aircraft>
    <engines>...</engines>
    <systemElements>...</systemElements>
  </vehicles>
  <toolspecific>...</toolspecific>
</cpacs>
```

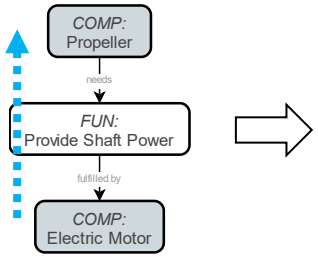
System Architecture Definition in CPACS

- In CPACS, the `systemArchitecture` is simply a collection of connections between components

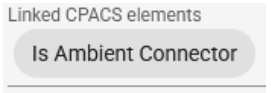
- Connections can be created by linking the “System Architecture Connector” to a function or to a port:
  - If linked to a **function:** the connection is established from needed to fulfilled
  - If linked to a **port**: from source to target

- The source and target elements need to be linked to CPACS elements, or be assigned a connector marker (e.g. ambient connection)



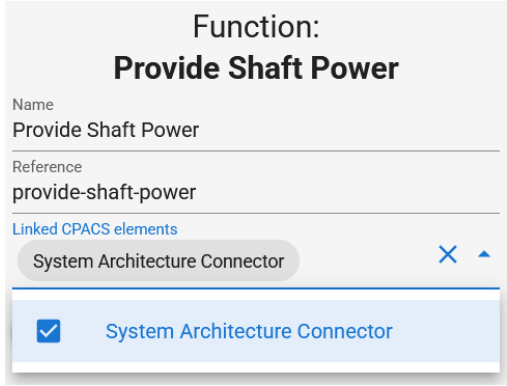



```
<systemArchitectures>
  <systemArchitecture>
    <connections>
      <connection uID="Electric_Motor_0_d6f0c9_to_Propeller_0_50d6d9">
        <source>
          <componentUID>Electric_Motor_0_d6f0c9</componentUID>
        </source>
        <target>
          <componentUID>Propeller_0_50d6d9</componentUID>
        </target>
      </connection>
```



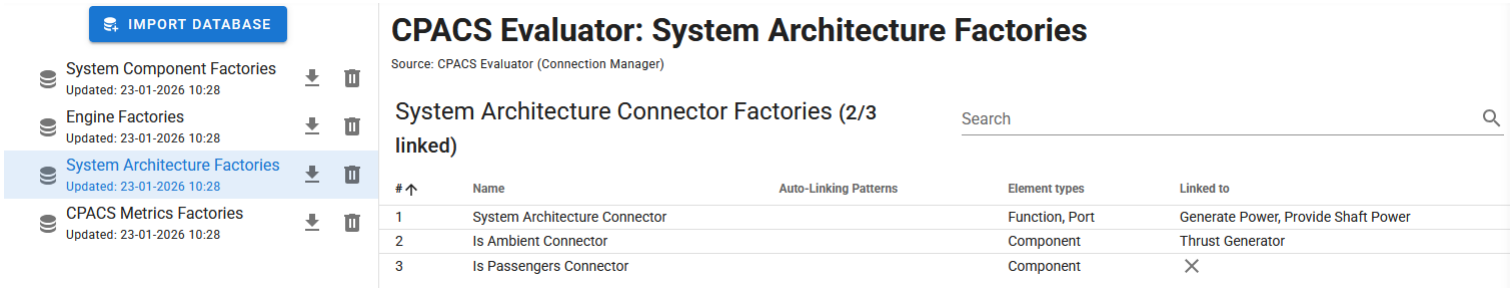
Exercise: CPACS System Architecture

- Establish a connection between the turboprop and the electric motor or turboshaft
  - Define “Provide Shaft Power” to be a connector
- Establish a connection between the turbofan/turboprop and the ambient
  - Define “Generate Power” as a connector
  - Mark “Thrust Generator” as being the ambient
- Observe the `systemArchitecture` being created in the exported CPACS file

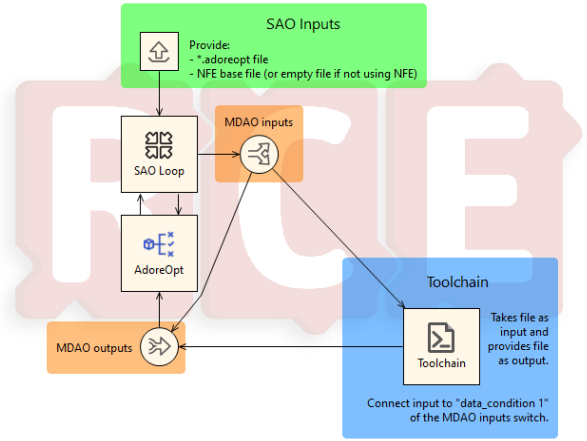


Exercise: CPACS System Architecture

- Establish a connection between the turboprop and the electric motor or turboshaft
  - Define “Provide Shaft Power” to be a connector
- Establish a connection between the turbofan/turboprop and the ambient
  - Define “Generate Power” as a connector
  - Mark “Thrust Generator” as being the ambient
- Observe the `systemArchitecture` being created in the exported CPACS file




```
<cpacs>
  <vehicles>
    <aircraft>
      <model uID="exampleAircraft">
        <name>Example aircraft</name>
        <engines>...</engines>
        <systems>...</systems>
        <systemArchitectures>
          <systemArchitecture>
            <name>Example system</name>
            <systemType>generic</systemType>
            <connections>
              <connection uID="Turboprop_0_6367de_to_ambient">
                <name>Turboprop_0_6367de_to_ambient</name>
                <source>
                  <componentUID>Turboprop_0_6367de</componentUID>
                </source>
                <target>
                  <externalElement>ambient</externalElement>
                </target>
              </connection>
              <connection uID="Turboprop_0_bc8264_to_ambient">
                <name>Turboprop_0_bc8264_to_ambient</name>
                <source>
                  <componentUID>Turboprop_0_bc8264</componentUID>
                </source>
                <target>
                  <externalElement>ambient</externalElement>
                </target>
              </connection>
              <connection uID="Electric_Motor_0_7fc8ae_to_Turboprop_0_6367de">
                <name>Electric_Motor_0_7fc8ae_to_Turboprop_0_6367de</name>
                <source>
                  <componentUID>Electric_Motor_0_7fc8ae</componentUID>
                </source>
                <target>
                  <componentUID>Turboprop_0_6367de</componentUID>
                </target>
              </connection>
              <connection uID="Electric_Motor_0_ec2d77_to_Turboprop_0_bc8264">
                <name>Electric_Motor_0_ec2d77_to_Turboprop_0_bc8264</name>
                <source>
                  <componentUID>Electric_Motor_0_ec2d77</componentUID>
                </source>
                <target>
                  <componentUID>Turboprop_0_bc8264</componentUID>
                </target>
              </connection>
            </connections>
          </systemArchitecture>
        </systemArchitectures>
      </model>
    </aircraft>
  </vehicles>
</cpacs>
```



ADORE

Remote File-based SAO (RFSAO)

Remote File-Based SAO (RFSAO)

Remote Execution

Configure the algorithm and evaluation interface in the GUI. Execute the optimization in an external environment.

Create a design problem in the GUI and click "start remote optimization".

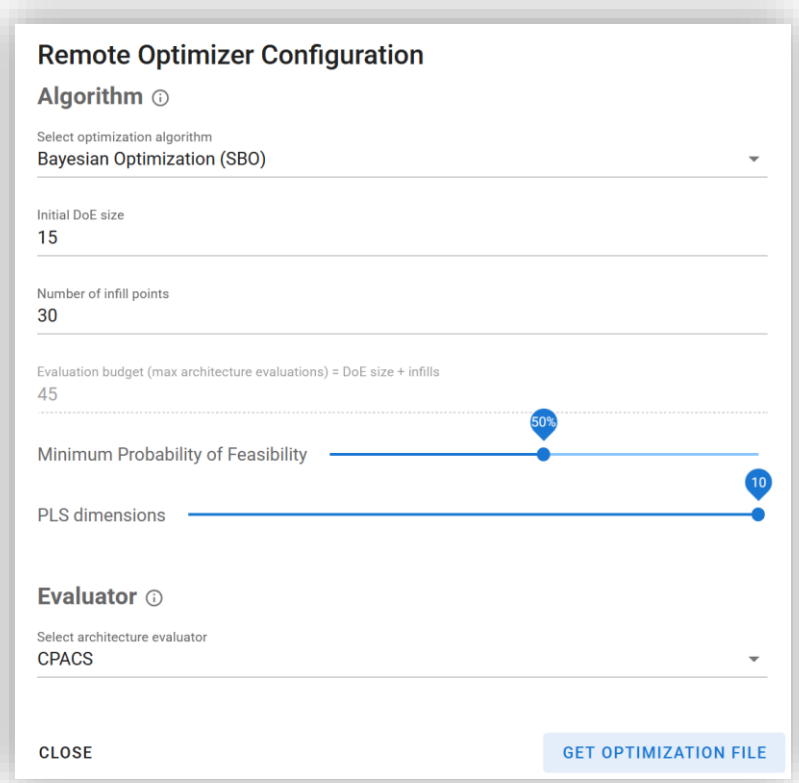
START REMOTE OPTIMIZATION

Algorithm

Choose from DoE, NSGA-II, SBO or SEGOMOE.
Refer to SBArchOpt for more details about the algorithms.

Evaluator

Choose from XML, JSON, XML NFE or CPACS.

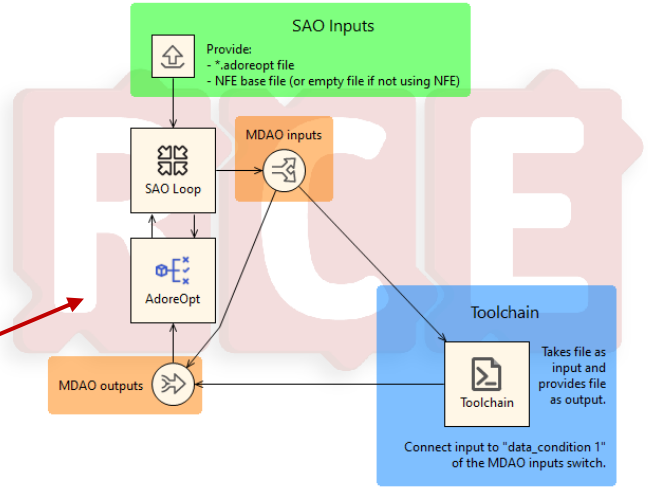


AdoreOpt file

Configuration and state is stored in an `.adoreopt` file.

Resulting AdoreOpt file

Open the resulting `.adoreopt` file in the GUI to view results and export to CSV.



ADORE
https://adore.mbse-env.com/docs/

RCE Workflow Template

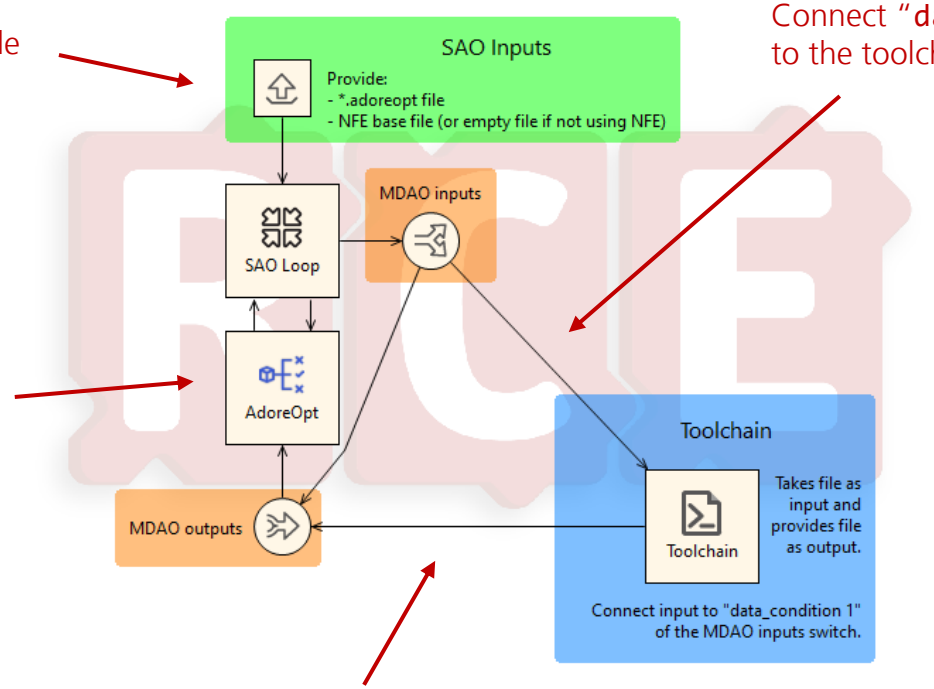
Remote Component Environment

RCE is an open-source process integration platform, developed by the DLR.
https://rcenvironment.de/

Workflow template
Start from `adore_rfsao_template.wf`.
Replace "toolchain" by the actual evaluation tools.

Workflow inputs
Provide the AdoreOpt file and (optionally) an XML or CPACS base file.

Evaluation input file
Connect "`data_condition 1`" to the toolchain.



AdoreOpt block
Remotely available or locally integrated. Refer to `remote_file_based/rce_integration` for more details.

Restart / Viewing Results
At each iteration, the AdoreOpt block outputs a new `.adoreopt` file:

- Provide it as workflow input to **restart** the workflow from that iteration.
- Open it in the GUI to **view** (intermediate) optimization results and export to CSV.

Evaluation output file
Connect to the joiner.