CODEX

Semantic Knowledge-Based-Engineering

MODPROD

15th MODPROD Workshop on Model-Based Cyber-physical Product Development

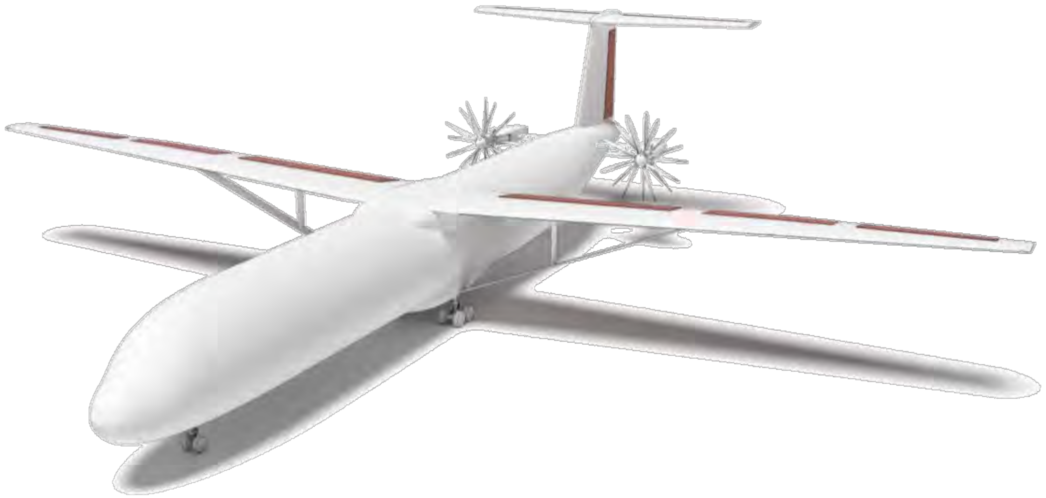
Linköping University, February 3-4, 2021

www.modprod.se

DLR

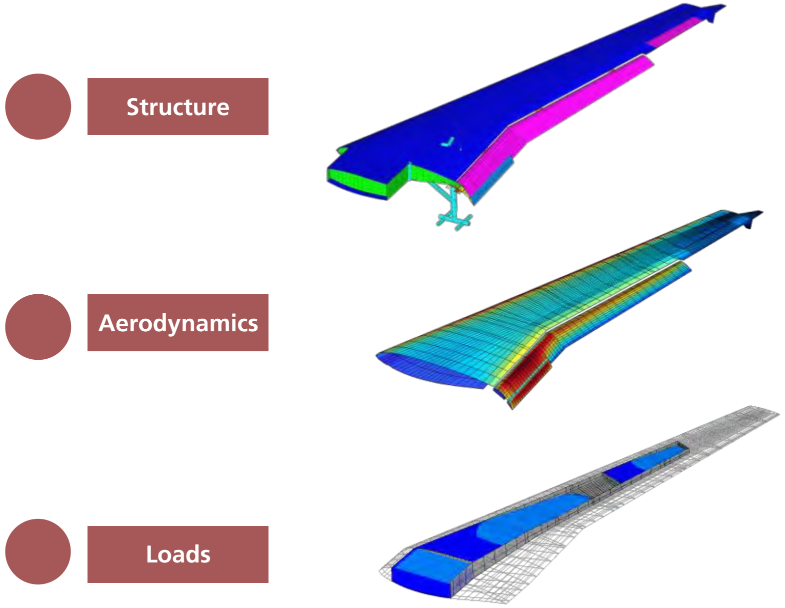
Presentation contents

- Challenges in aircraft design
- The Codex framework
- How can the new approach improve modelling and collaboration?
- Outlook

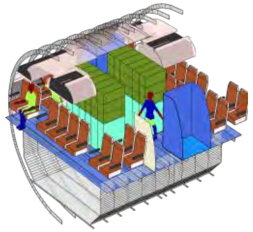


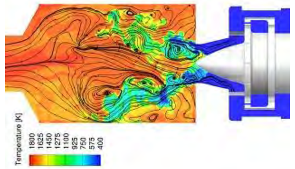
Aircraft design challenges

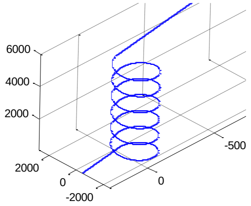
<u>A complex design problem</u>
- Multi-domain and multi-fidelity
- High amount of stakeholders involved

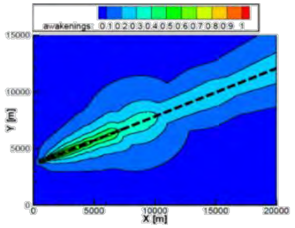


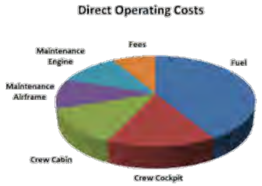
<u>High amount of legacy models available</u>
- Legacy: monolithic, difficult to maintain and adapt to everchanging requirements
- SotA: OOP yet difficult to integrate with other tools

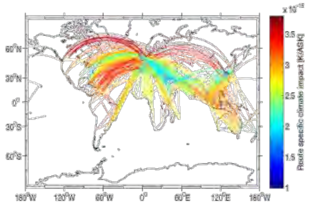












Collaboration in aircraft design – can it be improved?

Hypotheses

1. The application of semantic web technologies (SWT) provides a better infrastructure for multi-domain modelling than the current state-of-the-art Object-Oriented (OOP) methods.

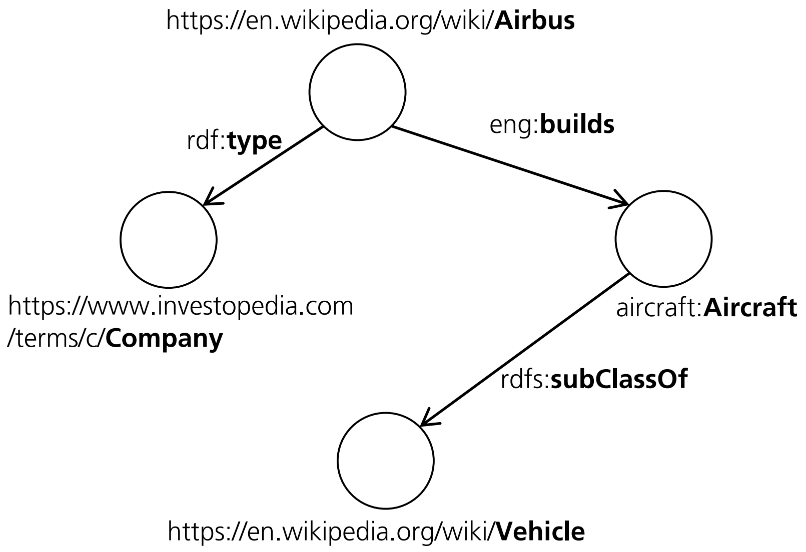
2. Schema-Less modelling decreases the communication overhead, improving the ease of collaboration.

CODEX

The aim of the Codex framework is to provide a **knowledge-formalization** and **execution environment** that reflects the graph structure of an effective collaborative environment.

Codex capabilities

Module	Function
Codex-Semantic	SWT core functionalities and inference engine

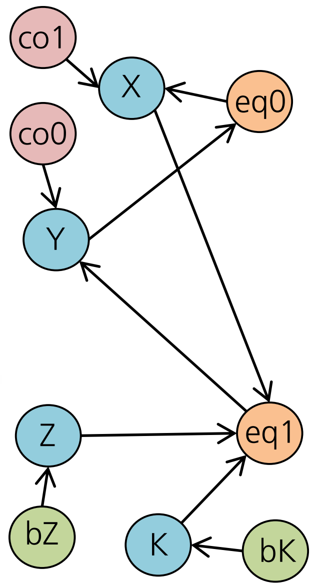


Codex capabilities

Module	Function
Codex-Semantic	SWT core functionalities and inference engine
Codex-Parametric	DSL for parametric rules and engines for constraint analysis and solution

CODEX

$$
\begin{cases}
eq0:\ y^2+x^2 = 5 \\
eq1:\ y = x^k - z \\
bZ: z = 3.0 \\
bK: k = 2.0 \\
co0: y < 0.0 \\
co1:\ x > 0.0
\end{cases}
$$

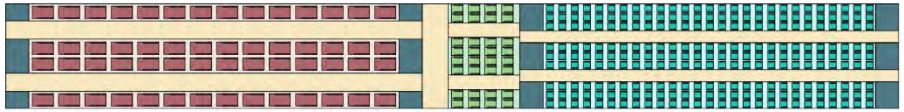


$$
\begin{aligned}
x &= \ \ 1.0 \\
y &= -2.0
\end{aligned}
$$

Codex capabilities

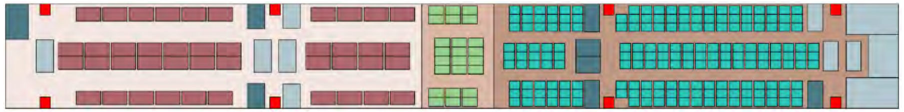
Module	Function
Codex-Semantic	SWT core functionalities and inference engine
Codex-Parametric	DSL for parametric rules and engines for constraint analysis and solution
Codex-Rules	DSL for production rules and engine for topological changes to the model

CODEX



+

Rule 1: If (…) then (add lavatories)
Rule 2: if (…) then (add emergency exits)

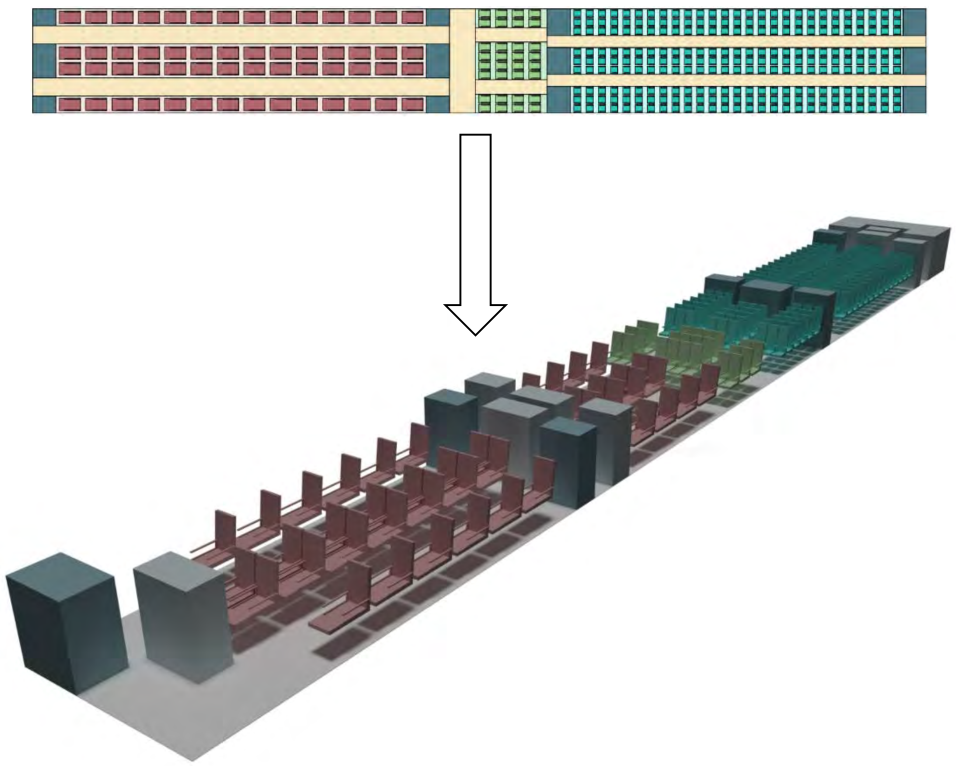


Codex capabilities

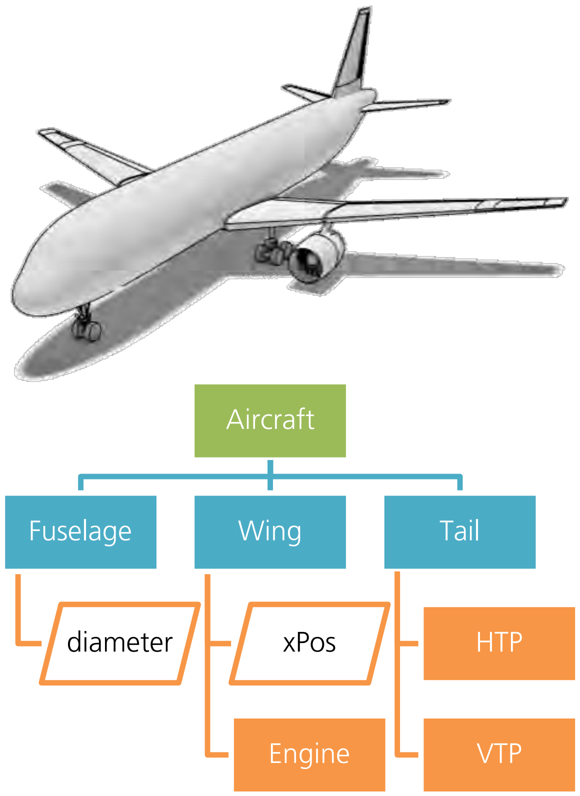
Module	Function
Codex-Semantic	SWT core functionalities and inference engine
Codex-Parametric	DSL for parametric rules and engines for constraint analysis and solution
Codex-Rules	DSL for production rules and engine for topological changes to the model
Codex-Geometry	Ontology for solid geometry modeling and 3D visualization of models
Codex-WebApp	Web-based UI for improved user experience and collaboration
Codex-Visualization	Knowledge graph and plots visualization

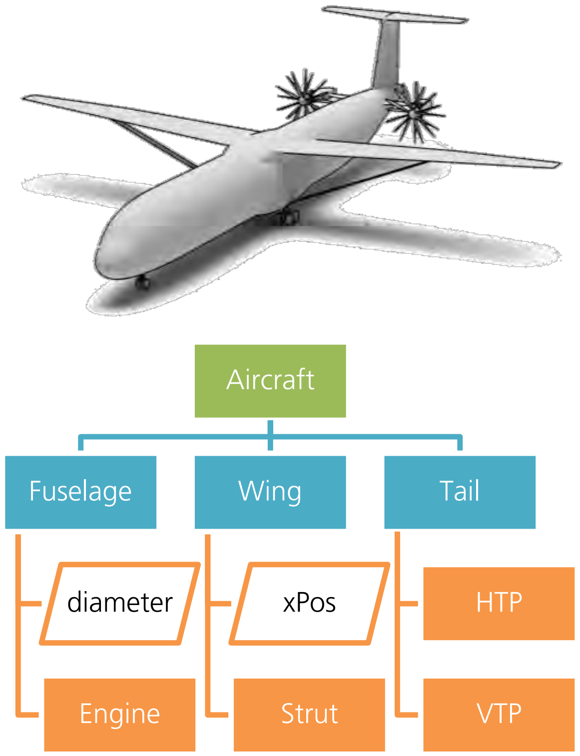
. . .

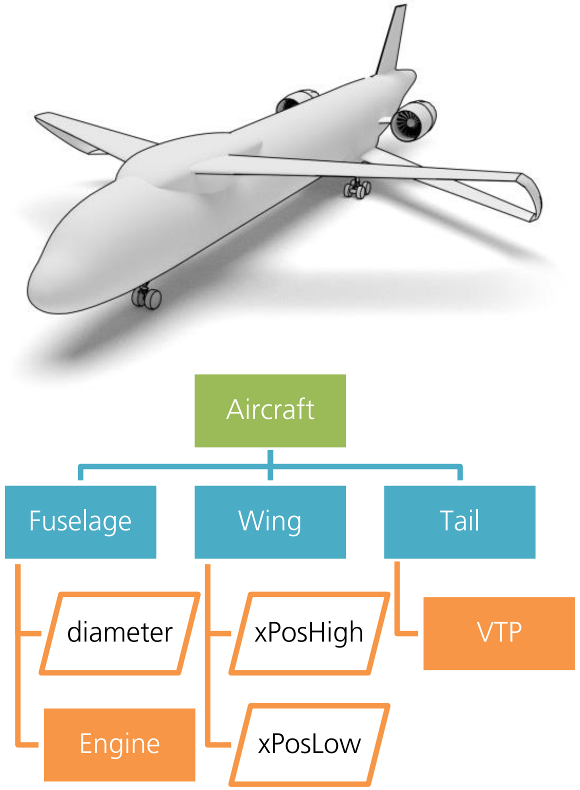
CODEX



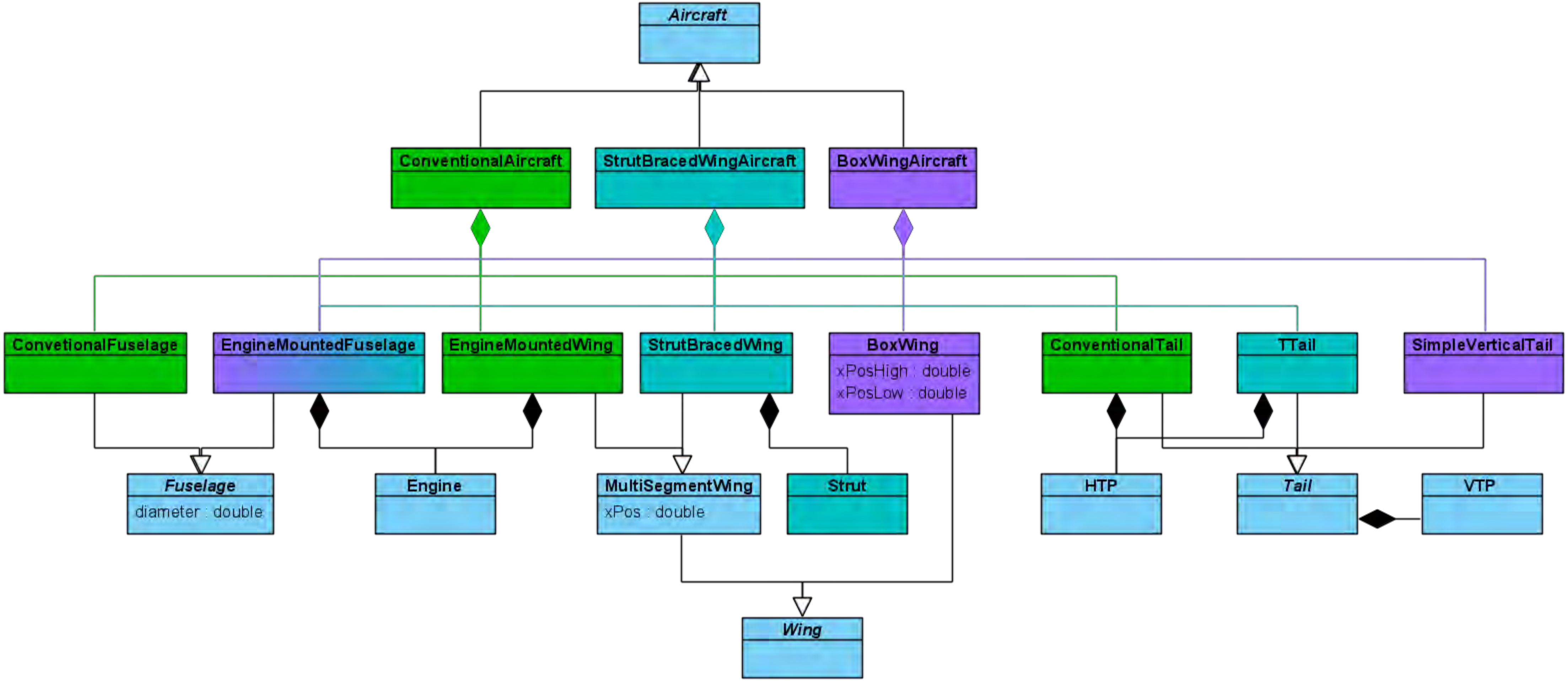
Challenge 1: Modelling configurations



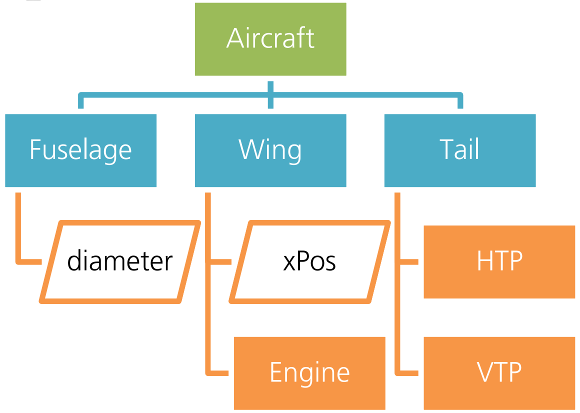




Creating Configurations – using the OOP paradigm
Step 1 – Create the Class hierarchy

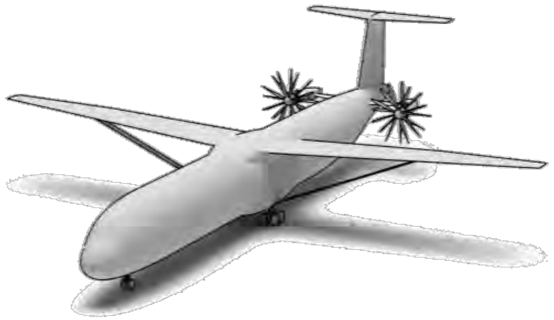


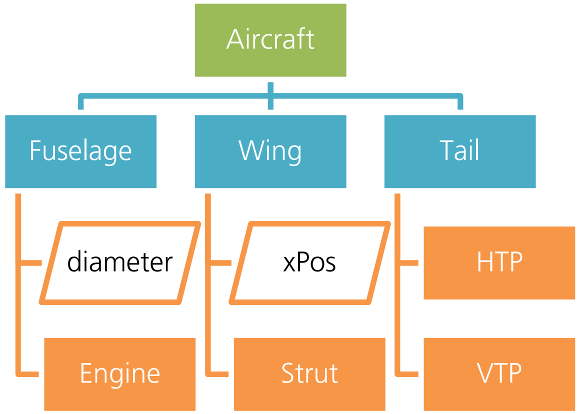
Creating Configurations – using the OOP paradigm
Step 2 – Create an instance of the model




```
val aircraft = ConventionalAircraft()
aircraft.fuselage = ConventionalFuselage()
aircraft.fuselage.diameter = 4.0
aircraft.wing = EngineMountedWing()
aircraft.wing.xPos = 15.0
aircraft.wing.engine = Engine()
aircraft.tail = ConventionalTail()
aircraft.tail.htp = HTP()
aircraft.tail.vtp = VTP()
```

Creating Configurations – using the OOP paradigm
Step 2 – Create an instance of the model





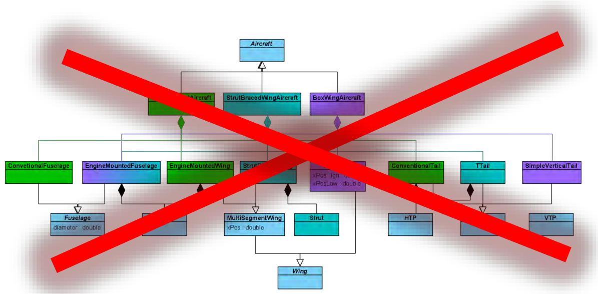

```
val aircraft = StrutBracedWingAircraft()
aircraft.fuselage = EngineMountedFuselage()
aircraft.fuselage.diameter = 4.0
aircraft.fuselage.engine = Engine()
aircraft.wing = StrutBracedWing()
aircraft.wing.xPos = 15.0
aircraft.wing.strut = Strut()
aircraft.tail = TTail()
aircraft.tail.htp = HTP()
aircraft.tail.vtp = VTP()
```

Creating Configurations – using the OOP paradigm
Step 3 – Apply design rules

```
if (aircraft is ConventionalAircraft) {
   ...
} else if (aircraft is StrutBracedWingAircraft) {
   ...
} else if (aircraft is BoxWingAircraft) {
   ...
}
if (aircraft.wing is MultiSegmentWing) {
    if (aircraft.wing is EngineMountedWing) {
       ...
    } else if (aircraft.wing is StrutBracedWing) {
       ...
    }
} else if (aircraft.wing is BoxWing) {
   ...
}
```

The set of design rules need to cover ALL possible cases explicitly

Creating Configurations – using SWT approach
Step 1 – Declare the properties and individuals you need

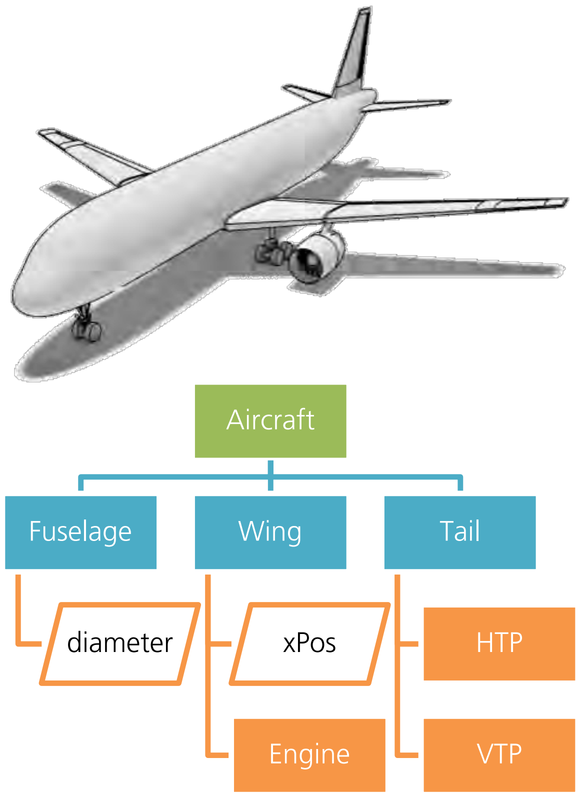


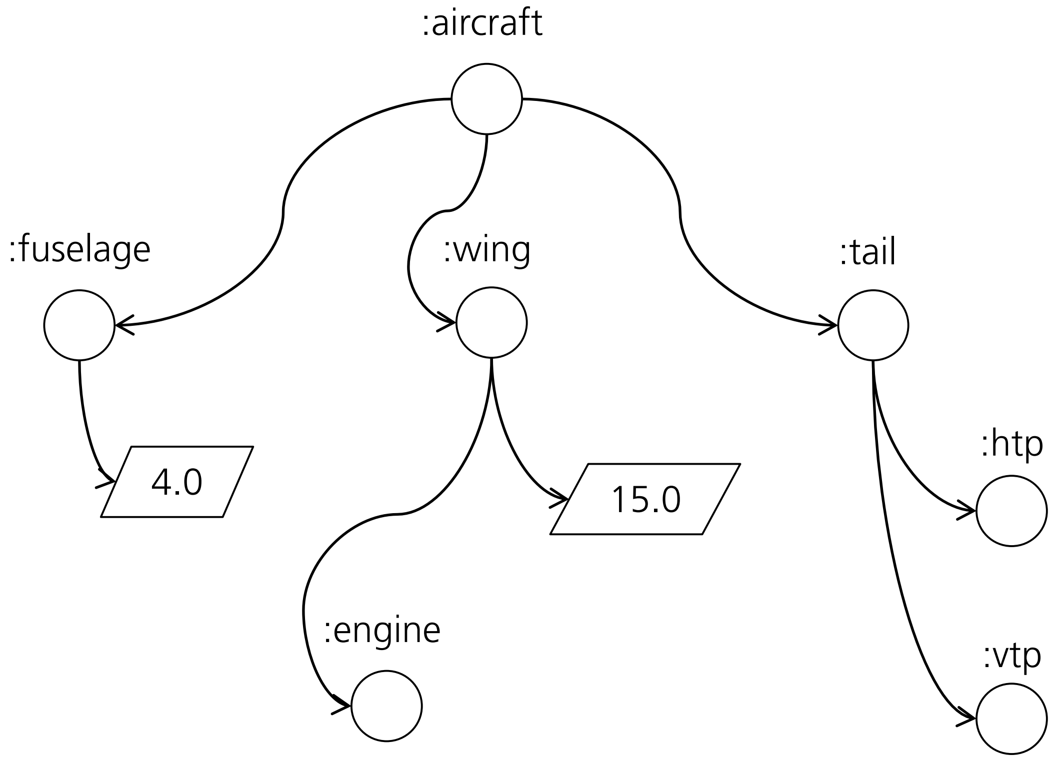

```
val hasPart   = ObjectProperty("hasPart")

val diameter  = DataProperty("diameter")
val xPos      = DataProperty("xPos")
val xPosHigh  = DataProperty("xPosHigh")
val xPosLow   = DataProperty("xPosLow")

val aircraft  = Individual("aircraft")
val fuselage  = Individual("fuselage")
val wing      = Individual("wing")
val htp       = Individual("htp")
val vtp       = Individual("vtp")
val tail      = Individual("tail")
val engine    = Individual("engine")
```

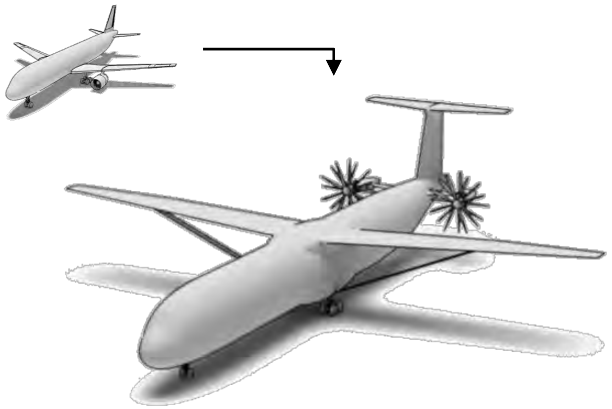
Creating Configurations – using SWT approach
Step 2 – Connect the individuals and properties

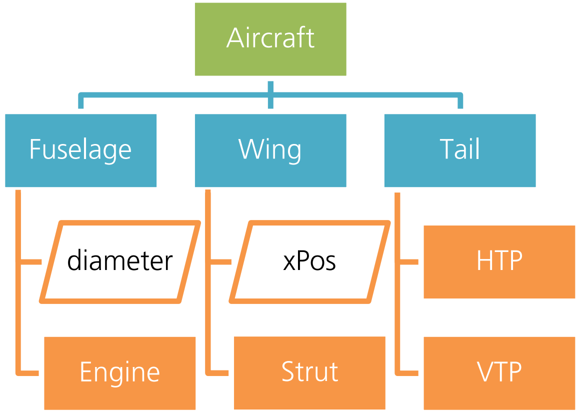


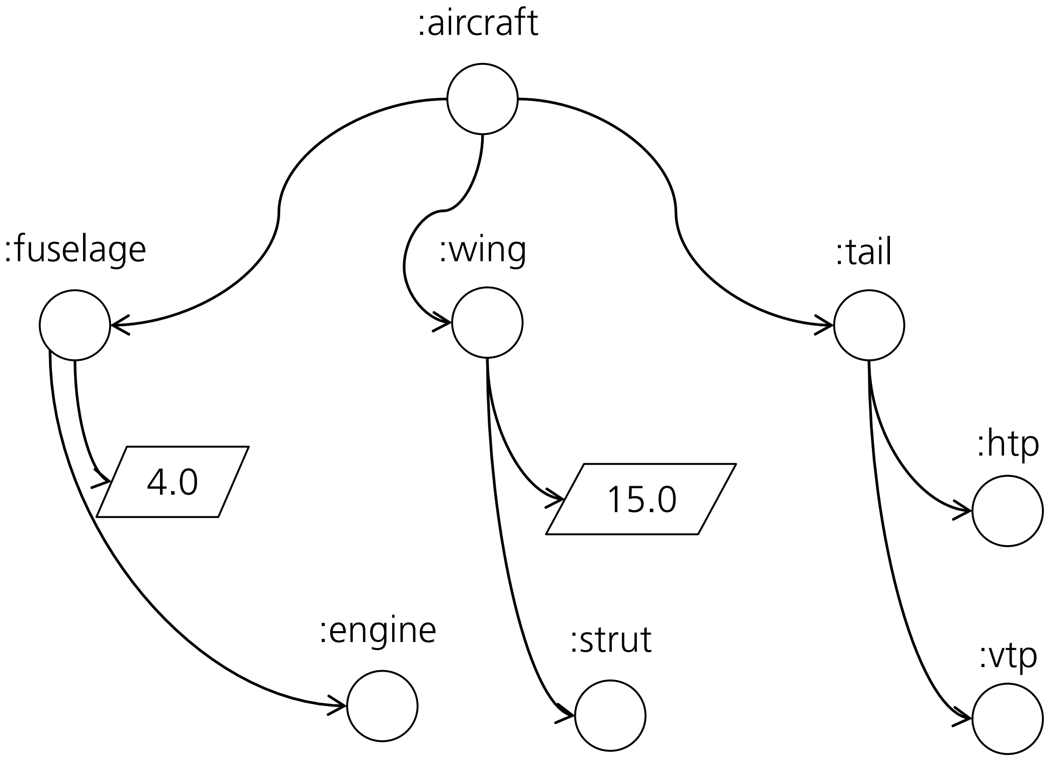



```
aircraft {
    Facts {
        hasPart(fuselage)
        hasPart(wing)
        hasPart(tail)
    }
}
fuselage {
    Facts {
        diameter(4.0)
    }
}
wing {
    Facts {
        xPos(15.0)
        hasPart(engine)
    }
}
tail {
    Facts {
        hasPart(htp)
        hasPart(vtp)
    }
}
```

Creating Configurations – using SWT approach
Step 2 – Connect the individuals and properties








```
aircraft {
    Facts {
        hasPart(fuselage)
        hasPart(wing)
        hasPart(tail)
    }
}
fuselage {
    Facts {
        diameter(4.0)
        hasPart(engine)
    }
}
wing {
    Facts {
        xPos(15.0)
        hasPart(strut)
    }
}
tail {
    Facts {
        hasPart(htp)
        hasPart(vtp)
    }
}
```

Creating Configurations – using SWT approach
Step 3 – Creating classes … only if needed

- Classes give specific meaning to individuals

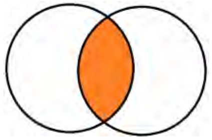
```
val Wing = Class("Wing")
val myWing = Individual("myWing") {
    Types(Wing)
}
```

- Classes don't need to have a name
  → Anonymous Classes

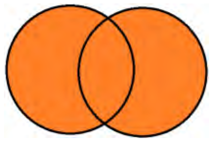
```
val anonClass = hasPart some Engine
```

- One can easily create classes using set-theory

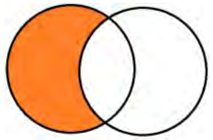
```
Wing and Fuselage
```



```
Wing or Fuselage
```



```
Wing and not(Fuselage)
```



Creating Configurations – using SWT approach
Step 4 – Apply design rules

```
if (aircraft is ConventionalAircraft) {
    ...
} else if (aircraft is StrutBracedWingAircraft) {
    ...
} else if (aircraft is BoxWingAircraft) {
    ...
}
if (aircraft.wing is MultiSegmentWing) {
    if (aircraft.wing is EngineMountedWing) {
        ...
    } else if (aircraft.wing is StrutBracedWing) {
        ...
    }
} else if (aircraft.wing is BoxWing) {
    ...
}
```

- No need for covering the complete permutation of design rule sets (IF…THEN…ELSE…)
- Design rules are applied to an individual corresponding to a specific Class or sub-graph
- This makes rules easily exchangeable
  → They are declarative, not procedural

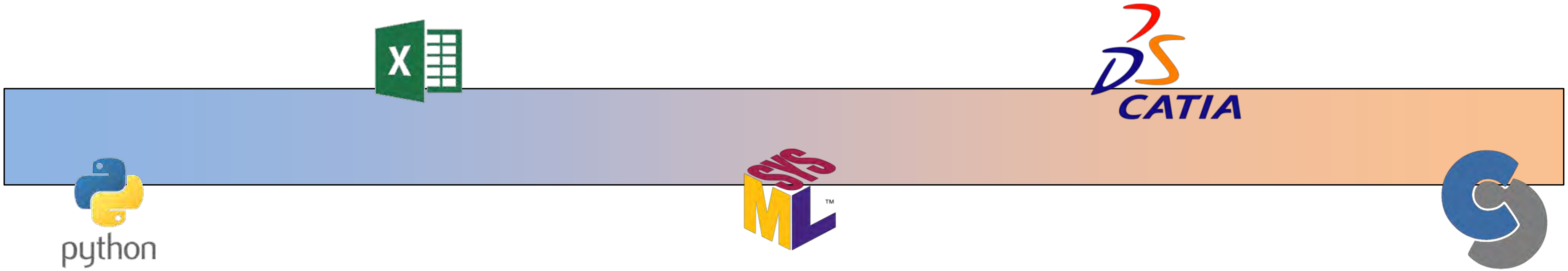
```
hasPart some Engine {
   // apply some rule to everything that
   // has an Engine
}

Wing that (hasPart some Strut) {
   // apply some rule to
   // all StrutBracedWings
}
```

Challenge 2: Balancing collaboration and modelling needs

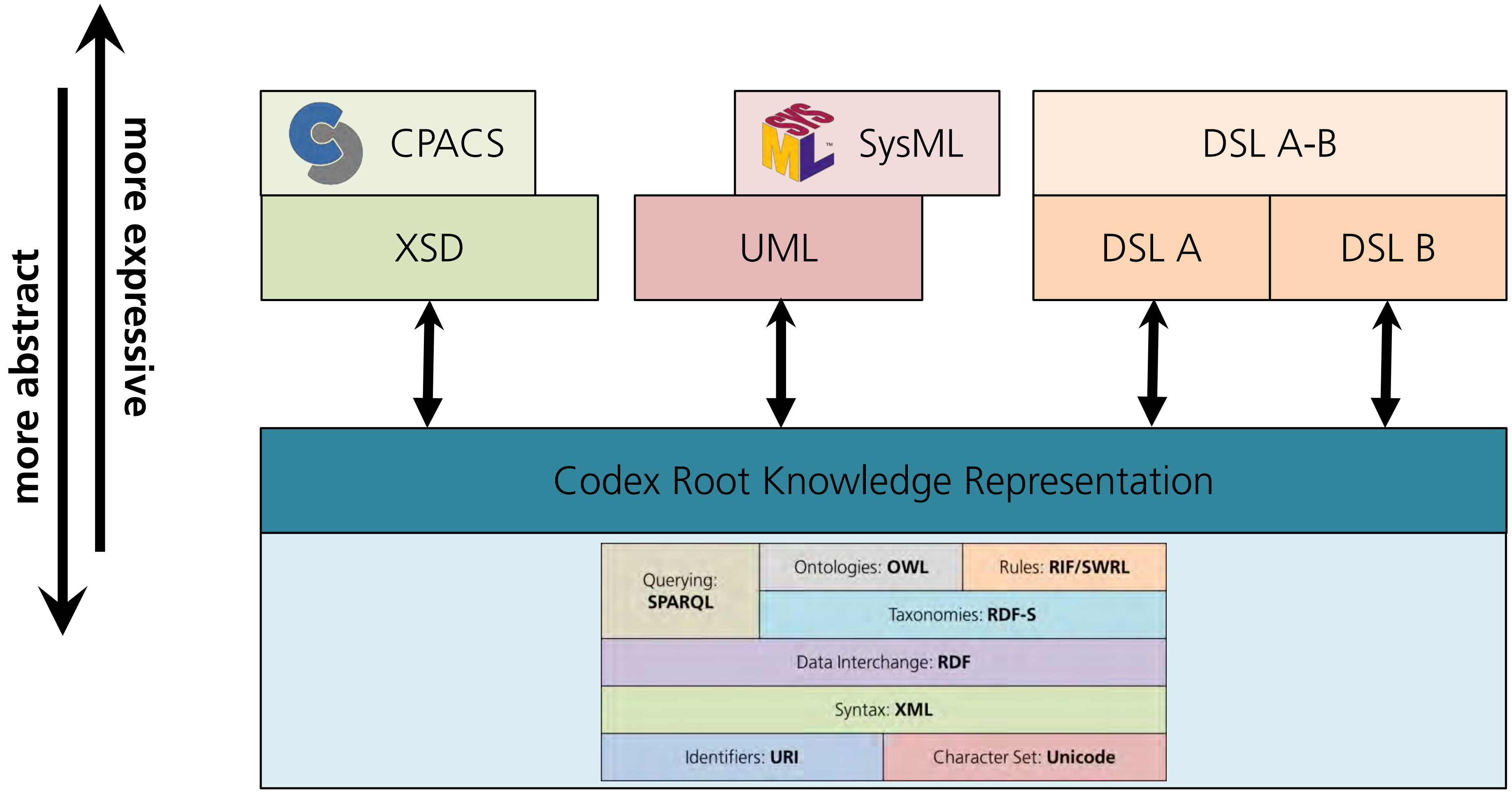
Multi-Domain Modelling demands a high degree of **abstraction** for easy cross-domain integration.

Domain-Specific Modelling demands high **expressivity** to make the modelling task easy for that domain.



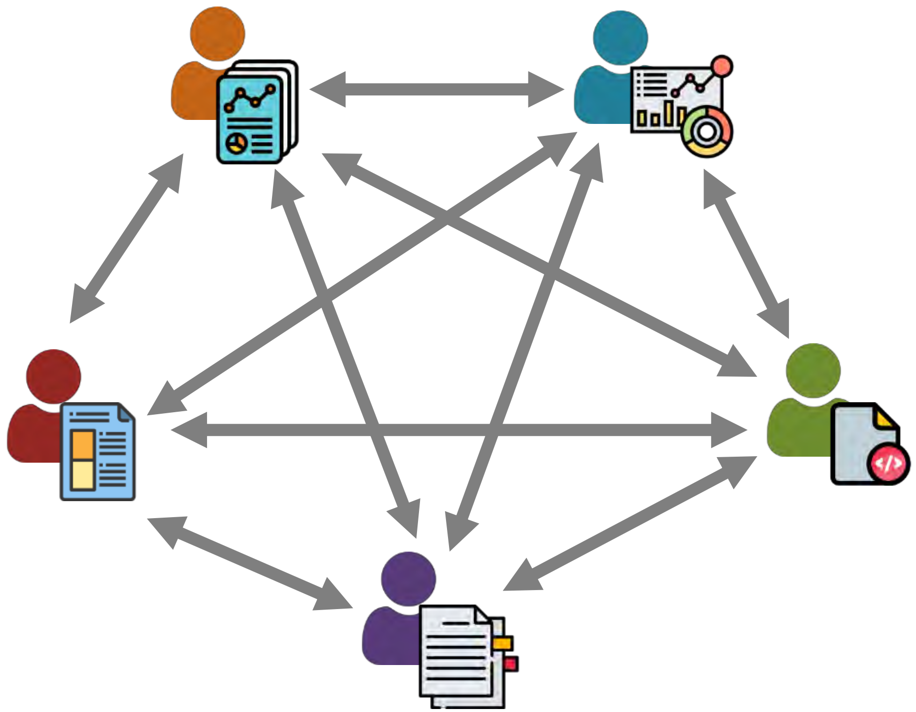
For effective collaboration the modelling approach should cover the entire spectrum

Collaboration and Modelling
Language layers

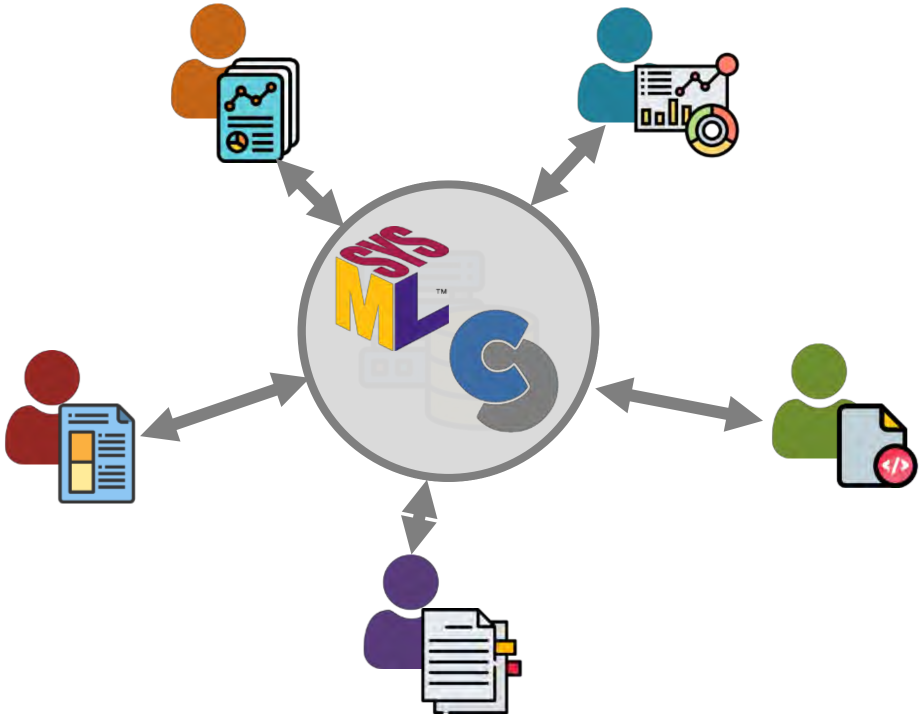


Challenge 3: Multi-Domain Collaboration

Legacy: document based approach



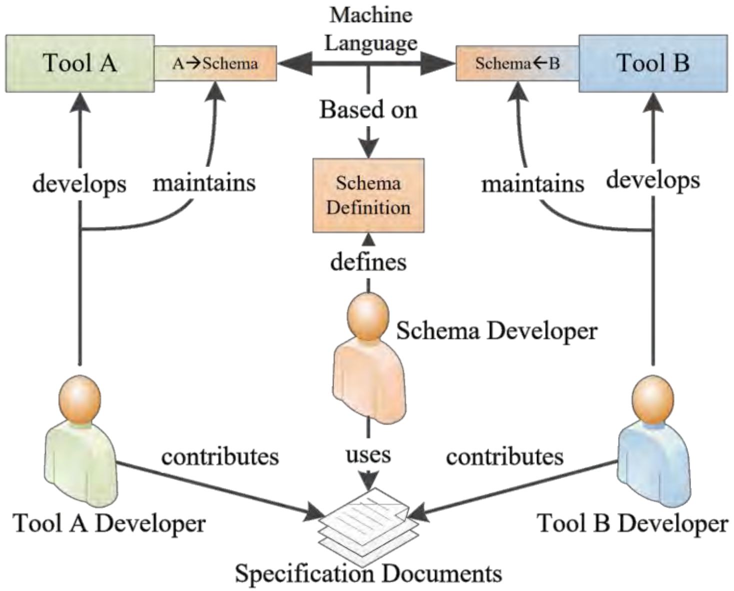
Current: model-based using common schema



Collaboration in aircraft design
Data Federation

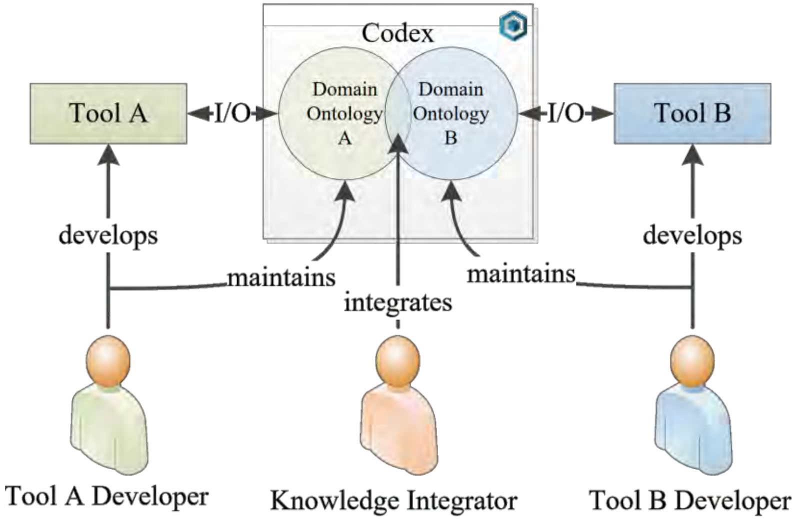
Schema-Based integration

- + Straightforward tool integration
- − Too rigid for highly dynamic knowledge formalization phase
- − All stakeholders must agree on the common schema



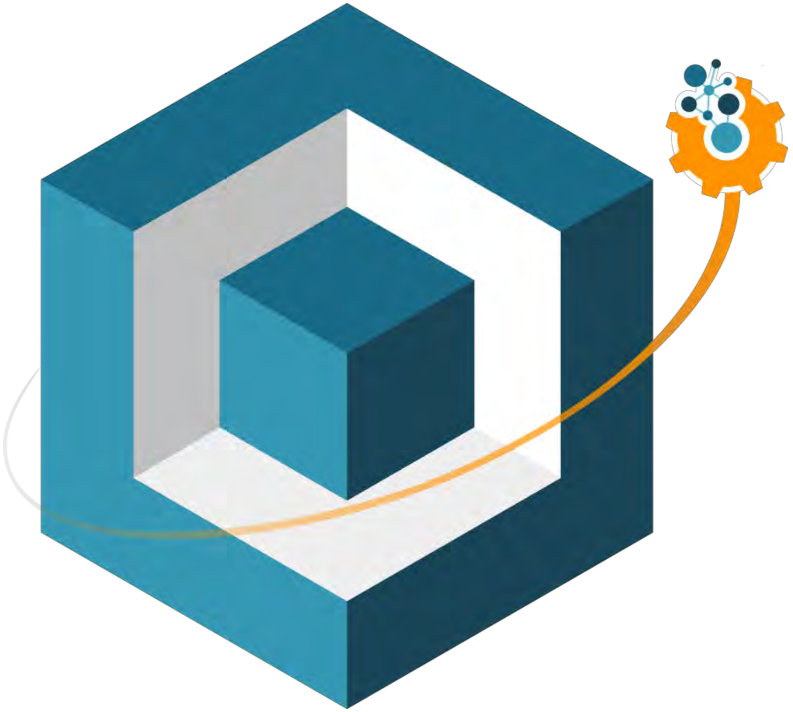
Schema-Less integration

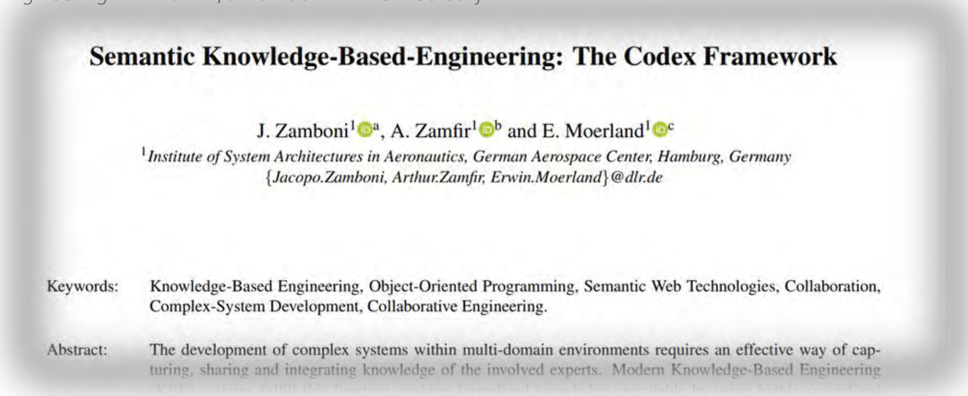
- + Domain experts model their knowledge independently
- + Model integrators link domain-specific models using the expressive power of SWT
- − Requires change of mindset, not straightforward



Outlook

- Integrating with other languages and software eco-systems
- Create new KBE tools using the Codex framework
- Managing scalability and complexity
- Developing a collaborative web application for knowledge engineering and integration




Semantic Knowledge-Based-Engineering: The Codex Framework

J. Zamboni[1][a], A. Zamfir[1][b] and E. Moerland[1][c]
[1] *Institute of System Architectures in Aeronautics, German Aerospace Center, Hamburg, Germany*
{Jacopo.Zamboni, Arthur.Zamfir, Erwin.Moerland}@dlr.de

Keywords: Knowledge-Based Engineering, Object-Oriented Programming, Semantic Web Technologies, Collaboration, Complex-System Development, Collaborative Engineering.

Abstract: The development of complex systems within multi-domain environments requires an effective way of capturing, sharing and integrating knowledge of the involved experts. Modern Knowledge-Based Engineering

Thank you for your attention

12th International Conference on Knowledge Engineering and Ontology Development – KEOD 2020 [link]

DOI: 10.5220/0010143202420249

Questions?

Contact

Institute of System Architectures in Aeronautics
@ German Aerospace Center (DLR) in Hamburg

Arthur Zamfir
✉ arthur.zamfir@dlr.de

Jacopo Zamboni
✉ jacopo.zamboni@dlr.de

Erwin Moerland
✉ erwin.moerland@dlr.de

DLR