# Technologies for Enabling System Architecture Optimization

J.H. Bussemaker, L. Boggero
German Aerospace Center (DLR)

ONERA-DLR Aerospace Symposium (ODAS) 2022
June 1-3, Hamburg, Germany

April 21, 2022

**Abstract**

Optimization of complex system architectures can support the non-biased search for novel architectures in the early design phase. Four aspects needed to enable architecture optimization and the author's views on how to solve them are discussed: formalization of the architecture design space, systematic exploration of the design space, conversion from architecture model to simulation model, and flexible simulation of architecture performance. Modeling the design space is done using the Architecture Design Space Graph (ADSG) implemented in ADORE. Systematic exploration can be done using evolutionary or surrogate-based optimization algorithms. Architecture to simulation model conversion can be done using an object-oriented approach using class factories, or using the MultiLinQ tool to synchronize a central data repository. Finally, simulation environments should expose a flexible and modular interface to be used in architecture optimization. A jet engine architecting problem is presented that demonstrates various aspects of system architecture optimization.

## Nomenclature

### Abbreviations

| | |
|---|---|
| ADORE | Architecture Design and Optimization Reasoning Environment |
| ADSG | Architecture Design Space Graph |
| CT | Component-Tool |
| DSO | Data Schema Operation |
| MBSE | Model-Based Systems Engineering |
| MDO | Multidisciplinary Design Optimization |
| NSGA2 | Non-dominated Sorting Genetic Algorithm 2 |
| QOI | Quantity of Interest |
| SBO | Surrogate-Based Optimization |
| TSFC | Thrust-Specific Fuel Consumption |

## 1 Introduction

The design of the system architecture greatly impacts how successful a system might be in its implementation. In finding the best system architecture to solve the problem at hand, however, often many different alternatives can be considered and have to be compared to each other. To mitigate this problem, often expert experience is used to narrow-down the architecture alternatives to be considered [19]. Thereby, however, the design process may be exposed to expert bias and conservatism [14]. The design process may benefit from keeping options open until later in the design process, thereby enabling a more complete and objective comparison of architecture alternatives. Such a goal can be achieved by applying numerical optimization methods to automatically search the design space to find promising architectures.

When generating possible design solutions, it is important to distinguish design goals, *what* is to be achieved, from design options, *how* are the goals achieved [14]. This is done by separating between function and form: function represents what the system does and ultimately represents the reason for the existence of the system; form represents the thing that will be implemented in the end to perform the functions, and can also be referred to as the components of a system. Therefore defining system architectures using functional decomposition is a useful method [16]: the breakdown is generic to any architecture alternative, it is free of solution bias by not suggesting architecture concepts, fits well with upstream systems engineering steps by explicitly specifying how requirements are fulfilled, and functions suggest types of solutions rather than specific technologies.

At the Institute of System Architectures in Aeronautics (SL) of the German Aerospace Center (DLR), efforts are being pursued to improve the design of complex systems by using Model-Based Systems Engineering (MBSE). Part of these efforts is the development of a novel architecture optimization methodology [9]. A previous publication has outlined which aspects need to be considered when implementing such an architecture optimization methodology [6], this paper builds on that and presents how the authors of this paper envision the specific implementation of these aspects. Specifically, the four main aspects that need to be considered when implementing architecture optimization are:

1. **Formalization** of the architecture design space in terms of functions and components;
2. Systematic **exploration** of the design space using optimization algorithms;
3. Generic and re-usable **interpretation** of an architecture model into a simulation model;
4. Flexible and modular quantitative **evaluation** of architecture performance.

The first two aspects enable the systematic generation of candidate architecture instances. These architecture instances fulfill functional requirements by assigning components to functions and further specifying component attributes; for example configuration options, connections, and sizing variables. Formalization of the architecture design space (aspect 1) and systematic exploration using optimization algorithm (aspect 2) fall in this category. To then select one or more system architectures to continue the detailed design process with, it is needed to evaluate the performances of the candidate architecture instances. Architecture model interpretation (aspect 3) and quantitative evaluation (aspect 4) fall in this category. These aspects can be combined to define a system architecture optimization process, visualized in Fig. 1.

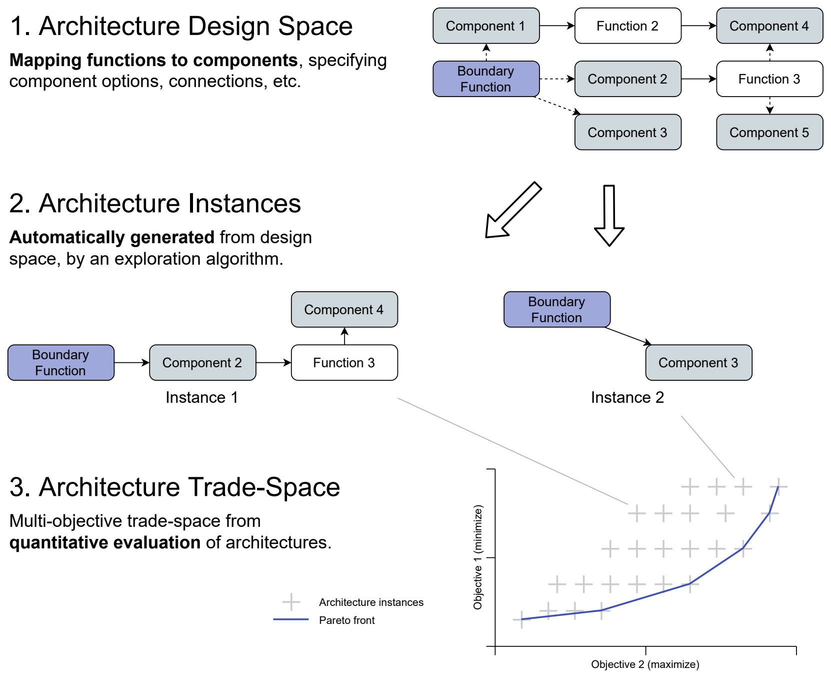

1. Architecture Design Space
**Mapping functions to components**, specifying component options, connections, etc.
Component 1
Function 2
Component 4
Boundary Function
Component 2
Function 3
Component 3
Component 5
2. Architecture Instances
**Automatically generated** from design space, by an exploration algorithm.
Component 4
Boundary Function
Component 2
Function 3
Instance 1
Boundary Function
Component 3
Instance 2
3. Architecture Trade-Space
Multi-objective trade-space from **quantitative evaluation** of architectures.
Architecture instances
Pareto front
Objective 1 (minimize)
Objective 2 (maximize)

Figure 1: Systematic architecture optimization concept. Figure reproduced from [6].

First, in Section 2 the aspects needed for systematically generating architecture instances are discussed. Aspects related to architecture performance evaluation are discussed in Section 3. Section 4 presents an example application of the discussed architecting process, and the paper concludes with Section 5.

## 2 Generating System Architectures

One of the aspects to consider in architecture optimization is the mechanism by which candidate architecture instances are generated. An architecture instance represents one design solution in the architecture design space, the space spanning all possible architectures as made up by the combination of all architectural choices. Before evaluation, they are denoted as candidate architecture instances, because whereas they satisfy function fulfillment and structural constraints, it is not yet known whether or not these architecture are feasible from a performance point of view.

The first step to enable architecture generation is to model the architectural design space by identifying all the different solution options available using a function-based decomposition. This design space model is then used to identify architectural decisions to be taken in order to define an architecture instance. Not taking structural constraints and incompatibilities between components into account, this design space model can also be used to get a rough estimate on the number of possible architectures; it is not uncommon to see extremely large numbers of possible architectures (e.g. $1.16 \cdot 10^{19}$ for an aircraft [13]). Modeling the design space and formalization of architectural choices is further discussed in Sec. 2.1.

The formalized architectural choices, together with the definition of evaluation metrics, can then be used to formulate an optimization problem. The types and number of design variables, objectives, and constraints, along with properties of the architecture evaluation determine which optimization algorithms are most appropriate for the design space exploration. Design space exploration using optimization is discussed in Sec. 2.2.

### 2.1 Modeling the Architecture Design Space

The formalization of architectural choices is an important enabler for architecture optimization. Not only because design automation requires a formalized mathematical model to work with, but also because it helps map out and document the design space and thereby enables discussion and offloads the mind to focus on one part of the design space at a time [14]. The architectural choice model should be built from upstream systems engineering results, such as requirements, scenarios, and use cases. This way, traceability of decisions is maintained, and it is made sure that all stakeholder needs are taken care of. The resulting architectural design space model then specifies how candidate architecture instances can be generated, and explicitly identifies all architectural choices and performance evaluation metrics.

In the framework of the authors, architecture design space modeling is done using the Architecture Design Space Graph (ADSG) [5]: a graph-based formulation mapping functions to components and representing component characterization and connection decisions. Fig. 2 shows an example of an ADSG. Decision nodes are automatically inserted for some patterns, for example when a function points to multiple components: this would indicate an architectural choice related to which component should fulfill the function in an architecture alternative.

An architecture optimization problem can be formulated from the ADSG by mapping decision nodes to design variables. Objectives and constraints are defined using Quantities of Interest (QOIs): values associated to functions or components that can serve different roles during an optimization process. QOIs can also be used as design variables (in addition to design variables defined from decision nodes), input parameters, and output metrics.

The ADSG is edited using a tool called ADORE (Architecture Design and Optimization Reasoning Environment). It provides a web-based graphical user interface for editing the ADSG, manually creating architecture alternatives, and defining design optimization problems. The tool also provides Python interfaces for connecting to several optimization frameworks, including OpenMDAO[1] and pymoo[2]. Architecture evaluation is achieved by implementing a Python interface that for a given architecture instance should return numerical values of relevant output QOIs (e.g. objectives, constraints). Fig. 3 shows the user interface of ADORE. For more information, the reader is referred to [3].

[1]OpenMDAO: https://openmdao.org/
[2]pymoo: https://pymoo.org/

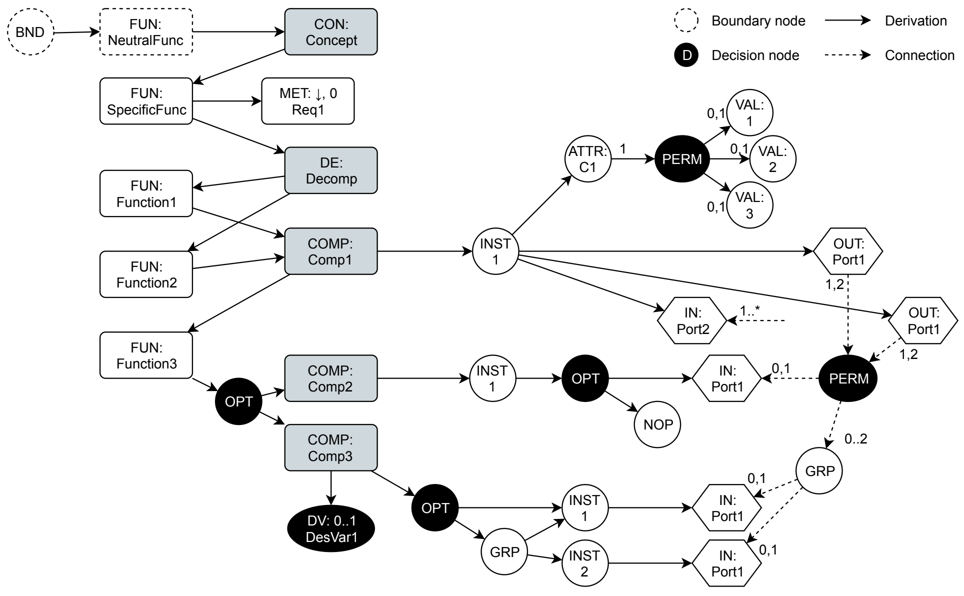
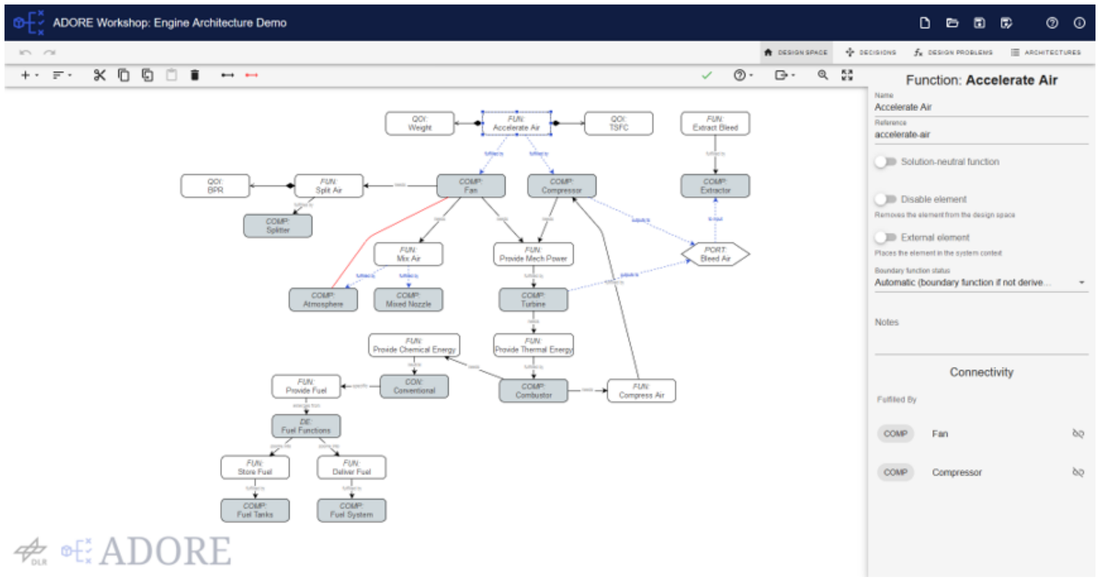


Figure 2: An example Architecture Design Space Graph (ADSG). Figure from [5].



Function: **Accelerate Air**

Name
Accelerate Air

Reference
accelerate-air

Solution-neutral function

Disable element
Removes the element from the design space

External element
Places the element in the system context

Boundary function status
Automatic (boundary function if not derive...

Notes

Connectivity

Fulfilled By

COMP Fan

COMP Compressor

Figure 3: ADORE web-based graphical user interface showing the design space editing canvas. Figure from [3].

## 2.2 Searching the Design Space

The other aspect needed for generating architectures is a way to suggest new design points to evaluate. Architectural decisions are known from the design space model, and can readily be mapped to mixed-discrete design variables. Architecture optimization problems often involve many different architecture alternatives, sometimes in the millions, due to combinatorial explosion of alternatives. It then becomes necessary to use some optimization algorithm to systematically search the design space if evaluating one architecture takes a significant amount of time, something which usually applies if numerical simulation tools are involved in the evaluation. In the most general sense, architecture optimization problems have the following properties [5]:

- Design variables can be **hierarchical**, due to decision options depending on other decisions taken before;
- Design variables are **mixed-discrete**, as architectural decisions often are categorical or integer design variables, and (continuous) sizing parameters can also be included in an architecture optimization;

- The optimization problem can be **multi-objective**, due to the presence of conflicting design goals coming from stakeholder needs.

Evolutionary algorithms such as NSGA2 are robustly able to deal with these challenges. However, they require a high number of function evaluations to find an optimum or Pareto front. An alternative class of optimization algorithms are Surrogate-Based Optimization (SBO) algorithms: a surrogate model is trained that is used to query the design space for promising new points. These new points are then evaluated, the surrogate model is updated, and the infill-evaluation cycle starts again.

Currently, hierarchical surrogate models exist to be used with SBO algorithms [12]. Also, mixed-discrete design variables [18] and multiple objectives [20] can be dealt with. Recently, research has been performed on combining all these features into SBO algorithms that can deal with all three architecture optimization challenges [2]. It was shown that SBO algorithms can be effective in finding a Pareto front in architecture optimization problems in several hundreds of function evaluations, compared to thousands needed for evolutionary algorithms. However, when exposed to hidden constraints [17] (i.e. evaluations that may sometimes fail, as often the case when evaluation involves physics simulation) the SBO algorithms are less effective than evolutionary algorithms.

## 3 Quantitative Evaluation of System Architectures

Implementing architecture optimization in practice depends on the capability to quantitatively evaluate the performance of candidate architecture instances. This is needed both for comparing different architecture instances for selection of the best architecture(s), and as feedback to the exploration (i.e. optimization) algorithm as needed for exploring new regions in the design space. For complex mechanical systems, architecture evaluation often involves the simulation of the performance of the system in various operating scenarios and from the point-of-view of different engineering disciplines (e.g. structural design, aerodynamic/hydrodynamic design, electronics, thermodynamics). The architecture evaluation capabilities are realized by focusing on two aspects: how to convert the abstract architecture instance model into a simulation model (Sec. 3.1), and how to prepare the simulation capabilities for use in architecture optimization (Sec. 3.2).

### 3.1 Architecture Model to Simulation Model Translation

Generated architecture instances, as coming from the design space exploration algorithm and design space model, are abstract descriptions of a system architecture: they describe the architecture in terms of architectural decisions, options, functions, etc. To evaluate the performance of an architecture, it is therefore needed to translate the model to something that can be interpreted by a simulation environment first. This conversion should be an automated procedure, as it needs to happen within the optimization loop [5]. Several approaches to convert architecture models to simulation environments exist, for a comprehensive overview the reader is referred to [4]. This section focuses on how to convert ADORE architecture models, as developed by the authors.

In general, the following type of information needs to be passed to the simulation model:

- The **allocation** of components to functions;
- The number of **component instances** of a component;
- Per component instance, **values** of associated Quantities of Interest (QOIs);
- Information regarding **connections** between components (using ports).

This information is available in the architecture alternative objects generated by ADORE. However, for the engineer implementing an architecture optimization problem directly parsing this data format is not convenient, and would involve a lot of repeated code between projects. To ease this process, the authors are currently developing two main approaches to build this conversion bridge: an object-oriented approach and an approach based on a central data repository. A description of both approaches follows.

#### 3.1.1 Object-Oriented Translation

The object-oriented approach for translating architecture models to simulation models is implemented in Python and directly integrates with the ADORE API. The idea behind it is that so-called **class factories** can be defined that instantiate custom classes based on elements appearing in an architecture instance. For example, a class factory might define the rule: instantiate the PROPELLER class for every "Propeller" component instance, and set the DIAMETER property to the value of the QOI called "Diameter". The advantage of this is that all information relevant for starting the simulation can be extracted from the architecture instance model without having to parse it directly. Additionally, these class factories represent explicit definitions of what is possible to be simulated, and therefore assist in creating the design space model by constraining the types of elements that can be modeled: it naturally facilitates the interaction between the design space model and the evaluation capabilities as illustrated in Fig. 4.

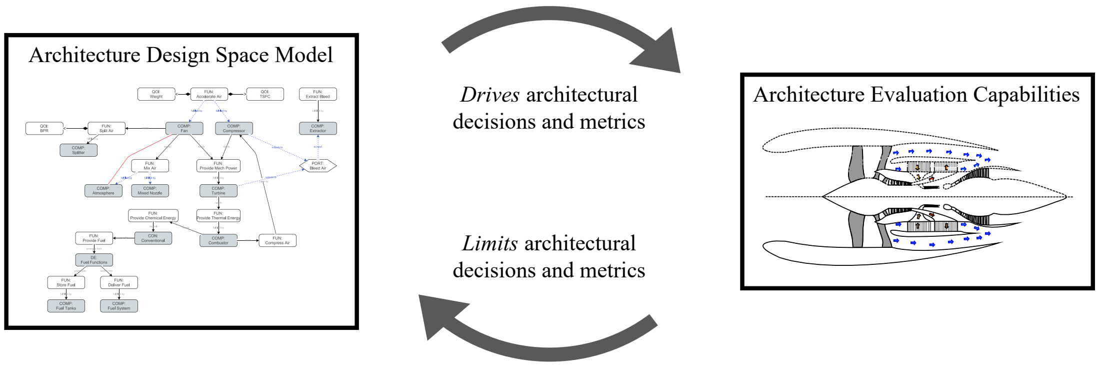

Architecture Design Space Model
*Drives* architectural decisions and metrics
*Limits* architectural decisions and metrics
Architecture Evaluation Capabilities

Figure 4: Interaction between the architecture design space model and evaluation capabilities. Figure from [6].

Setting up class factories is an iterative process that is done concurrently with implementing the simulation framework, if not already available. This is because the interface of the simulation framework should be in terms of these classes-to-be-instantiated. The process then roughly consists of the following (iterative) steps:

1. Define class factories for building input for the simulation framework;
2. Export class factory definitions to ADORE, and link architecture elements to class factories;
3. Implement the evaluation function that uses the class factories to instantiate classes and triggers the simulation run;
4. Test the implementation: class instantiation, metrics extraction, and finally evaluation of generated architectures.

The evaluation process for one specific design vector during an optimization run is visualized in Fig. 5. The green boxes denote problem-specific programming effort associated to item 3 in the list above. In summary, the object-oriented class factory evaluation is a convenient Python-based approach for extracting information from architecture instances. However, it does require programming effort on the engineer's part.

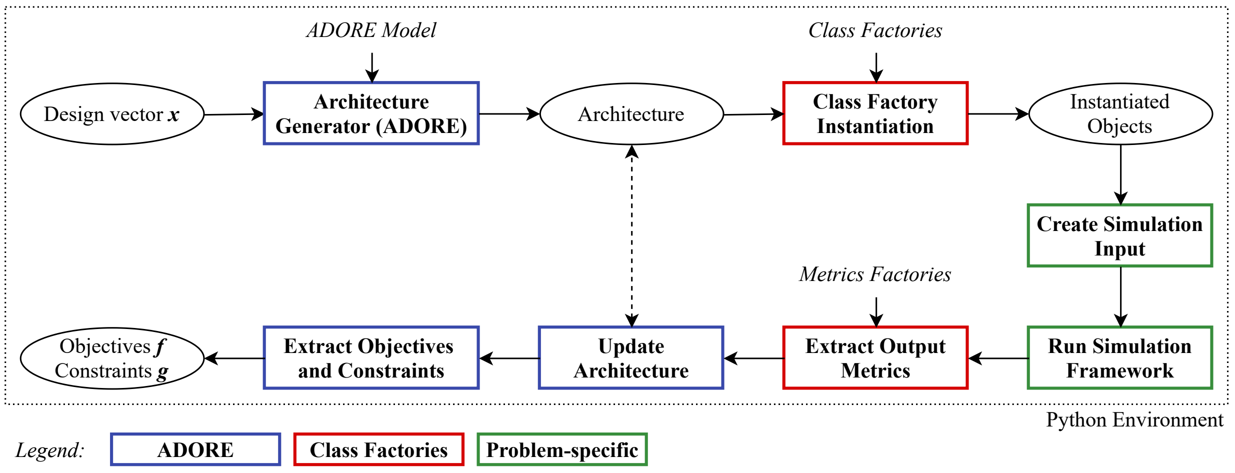


Figure 5: Architecture translation and evaluation procedure when using class factories for object-oriented translation.

### 3.1.2 Central Data Repository Synchronization

A central data repository (e.g. CPACS in the field of aircraft design [1]) can be helpful when implementing Multidisciplinary Design Optimization (MDO) toolchains for reducing the number of data interfaces that must be implemented [1]. Additionally, it forces everyone involved to "speak the same language" and thereby reduces problems arising from different data interpretation or jargon mismatch. The central data repository synchronization approach then enables information an an ADORE architecture instance to be correctly represented in the central data repository, so that the MDO toolchain may analyze it.

The link from architecture to central data repository is implemented by the MultiLinQ tool [3], developed by the authors. MultiLinQ focuses on providing the two-way data communication needed to close the optimization loop, by providing a web-based graphical user interface to specify so-called **Data Schema Operations (DSOs)**. A DSO is some operation that is applied to the data repository based on the number of times the associated architecture element is instantiated in the architecture, and on other information such as QOI values and component connections. The evaluation loop during an optimization run then looks like Fig. 6: after an architecture is generated, DSOs are applied on an initial central data repository instance. The mapped product data is then passed to the existing MDO workflow, and finally metrics are extracted from the output data and fed back to the optimizer. Several methods for communicating the input and output files between the Python environment and MDO integration environment are provided. For more details on this process, the reader is referred to [3].

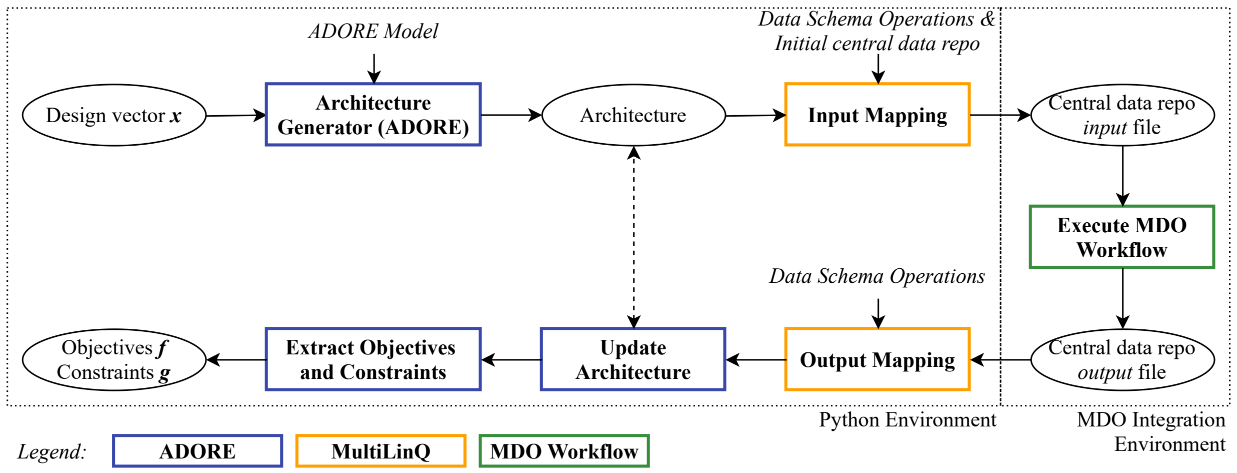


Figure 6: Architecture translation and evaluation procedure when using MultiLinQ to synchronize with a central data repository.

MultiLinQ also relates the central data repository to the disciplinary analysis tools that read from (input) or write to (output) nodes in the central data repository. Combining this knowledge with the defined DSOs allows associating architecture elements, like components and QOIs, to disciplinary analysis tools in a so-called Component-Tool (CT) matrix. This matrix can assist the engineer by identifying interchangeable (i.e. redundant), missing, or unnecessary disciplinary tools, and thereby assist also in the formulation of the MDO toolchain.

Finally, after DSOs are defined, their logic can be verified directly from the user interface of MultiLinQ. To do this, the user would select an ADORE architecture and an initial central data repository file, and then verify that the architecture is interpreted correctly (e.g. in terms of number of component instances and QOI values) and verify that DSOs are applied correctly to the central data repository. The complete process in MultiLinQ then consists of the following (iterative) steps:

1. Import tool input/output definitions;
2. Import the ADORE architecture design space model;
3. Define DSOs for architecture elements (components, QOIs, etc.);
4. Use the CT-matrix to improve the MDO toolchain if necessary;
5. Verify DSO behavior for generated ADORE architecture.

In summary, the central data repository approach implemented in MultiLinQ offers a programming-free and user-friendly way to link ADORE architectures to MDO toolchains using a central data repository. However, it depends on the availability of a central data repository and associated MDO toolchain.

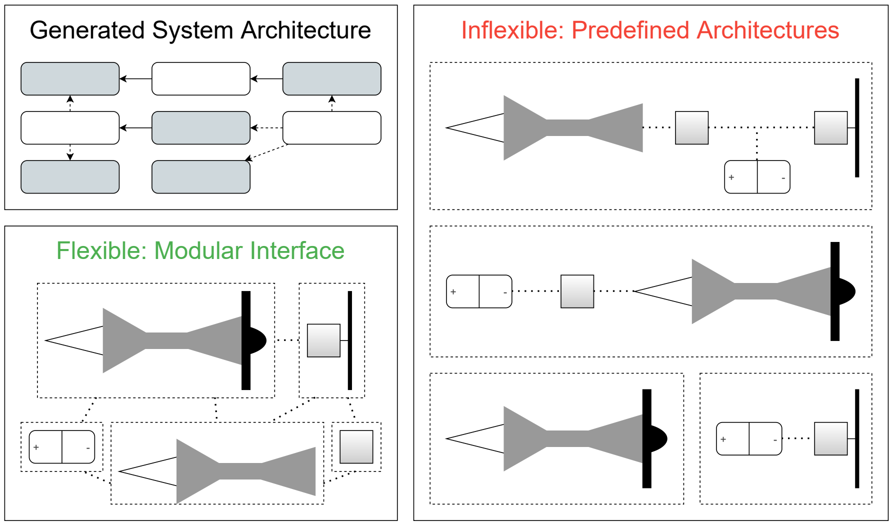

Generated System Architecture
Inflexible: Predefined Architectures
Flexible: Modular Interface

Figure 7: Comparison between flexible and inflexible interfaces to the simulation environment. Figure reproduced from [6].

## 3.2 Flexible Multidisciplinary Simulation

The simulation capabilities are the main enabler of the architectural options that can be explored in an architecture optimization problem. The possibilities for the architecture optimization problem are mainly limited by which architecture elements and architectures can be modeled, the availability and accuracy of performance metrics, and the execution time of one simulation. The first relates to the design space model itself; if the simulation environment is not capable of representing some of the components in the architecture design space model, these cannot be taken into account in the architecture optimization problem. The second point relates to the performance metrics used to compare different architecture instances: if it is not possible to compute certain metrics currently, this can signal the need for selection or development of additional engineering tools. The third point drives the selection of the optimization algorithm: for longer execution times algorithms specifically designed to limit the number of function evaluations, such as Surrogate-Based Optimization (SBO) algorithms, might be needed.

### 3.2.1 Object-Oriented Translation

When using object-oriented architecture translation, the evaluation environment should support a similar level of modularity and flexibility as the architecture model: if the simulation environment is less flexible than the architecture design space model, it reduces the types of architecture instances that can be explored (the first point in the list above). The simulation environment should expose a **modular interface** that enables an almost 1-to-1 translation from the architecture elements and connections to the evaluation model. The only extra information that then needs to be specified are boundary and operating conditions. The interface modularity concept is visualized in Fig. 7: an inflexible interface means that only predefined architectures can be simulated, whereas a flexible interface enables the evaluation of any generated architecture.

Implementing a simulation framework that exposes such a modular interface can be supported by the use of the concepts of **abstraction and inheritance**. These concepts allow elements and processes to be represented as hierarchies: all objects share a common interfaces, and progressively refine their internal implementations. This allows both the development of specialized solvers (e.g. for differential equations), and the co-location of element and its behavior. Thereby the development of the core solver and implementation of new elements is decoupled, and the definition of new architecture by (re)combination of elements is supported.

An example of a simulation framework that is based on these principles is pyCycle [11], a thermodynamic cycle solver used for the design of aircraft engines, with a library of engine components (e.g. compressor, combustor, turbine). At their basic level, each of these components define a transformation of airflow in terms of thermodynamic quantities such as pressure, entropy, and temperature.

### 3.2.2 Central Data Repository Synchronization

It can help to use a central data repository to support the data exchange between multiple engineering disciplines in an MDO workflow, especially in a collaborative context with engineering tools distributed across team and/or

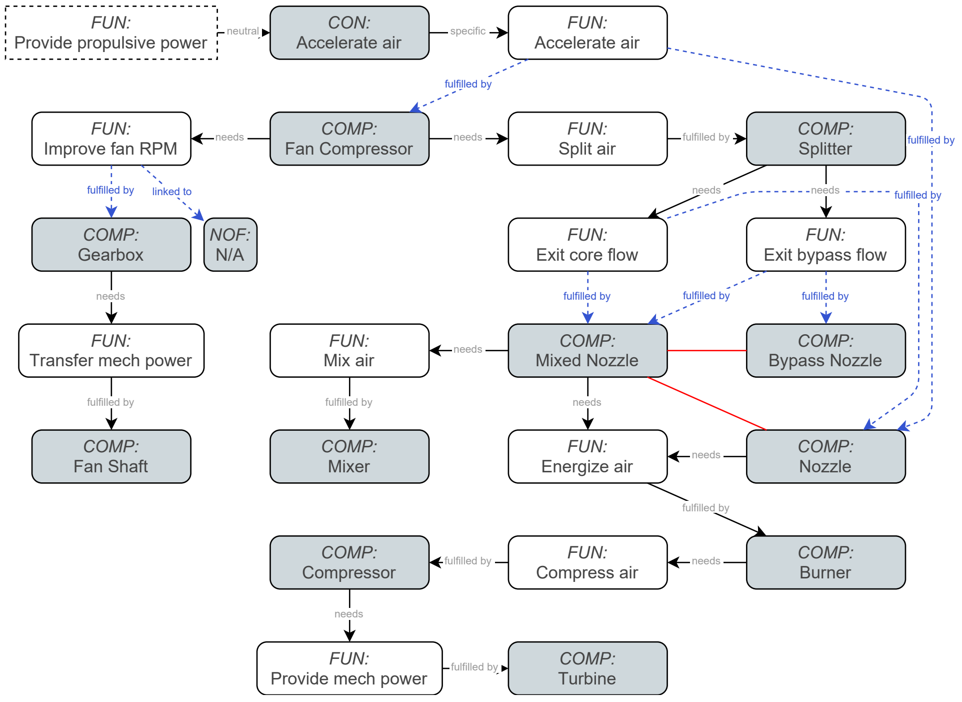


Figure 8: Part of the ADORE architecture design space model of the jet engine architecture design problem. Figure reproduced from [3].

organizational boundaries [8]. To use such an approach for architecture optimization, it should be ensured that the central data repository format supports the flexible definition of architectures. Once that is achieved, the associated MDO workflow also need to be flexible enough, which can be achieved either by ensuring that the workflow itself can support all architectures, or by creating different workflows to deal with different architectures.

A **single workflow** supporting the evaluation of all architectures is the easiest to in its use, as it would be directly compatible with MultiLinQ, involves less data management, and is easier for maintaining an overview. To achieve this, tools can be designed to work for any number of component instances: for example, a propeller thrust calculation tool might be able to calculate the thrust of one propeller as well as several propellers or even many propellers in the case of a distributed-propulsion architecture. Another way to achieve flexibility within the workflow is to add simple switches that would bypass or include certain tools based on some data in the central data repository: for example, hydrogen fuel cell calculations might be skipped if there are no fuel cells in the architecture.

It is expected, however, that for some architecture design problems this approach becomes too complicated too manage, and **multiple workflows** will need to be created to support the evaluation of different architectures. The number of different workflows probably is less than the number of possible architectures, due to shared disciplines between architectures. Automatically creating workflows for different architectures or subsets of architectures was done by Frank et al. [10] based on matching patterns in morphological matrices.

## 4 Example Application: Aircraft Jet Engine Architecting

The previously discussed aspects are demonstrated using the multidisciplinary aircraft jet engine architecting problem from [2]: a benchmark problem that demonstrates the specific behaviors of architecture optimization problems. The engine evaluator part of that work is able to model many different engine architectures, including a jet engine and turbofan engine, and including several performance-improving technologies like an intercooler, and a fan gearbox. The engine evaluation has been wrapped using class factories to enable object-oriented architecture translation, and an accompanying architecture design space model has been constructed to enable architecture optimization.

The first aspect of system architecture optimization, formalization of the architecture design space, is done by creating an ADORE design space model, partially shown in Fig. 8. Derivation of architecture alternatives starts at the boundary function "provide propulsive power". Blue-dashed lines represent architecture decisions, for example whether to add a fan or not, and whether to use a mixed nozzle or not. Red lines represent incompatibility constraints, meaning that these elements cannot exist together in an architecture alternative.

The engine evaluation code is based on pyCycle [11], a modular framework based on OpenMDAO that implements several engine components. All turbomachinery components are implemented using the principles of inheritance and abstraction: for example, most of these components are flow-transformation (e.g. pressure, enthalpy, temperature change) elements at their base, however each of them implements specific equations related to their function. Engine architectures are defined by instantiating classes representing turbomachinery and mechanical elements to establish airflow and mechanical connections (i.e. shafts). The evaluator outputs the following performance metrics: thrust-specific fuel consumption (TSFC), weight, length, diameter, NOx emissions, and noise level. One engine evaluation takes about two minutes.

Translation from ADORE architecture alternatives to a turbofan architecture that can be interpreted by the evaluation code is done using the object-oriented translation approach. Class factories are defined for each of the engine components, including related sizing QOIs such as bypass ratio, pressure ratios, Mach numbers, bleed airflows, shaft power offtake, and operating conditions. Airflow, bleed, and shaft connections are defined using ports.

The multi-objective evolutionary algorithm NSGA2 is used to solve the architecture optimization problem as defined from the design space model. It was feasible to apply this algorithm with a budget of several thousand function evaluations due to the relatively short evaluation time of two minutes. The optimization problem features 41 design variables of which 6 categorical, 5 integer, and 30 continuous variables. Three objectives were optimized for: weight, TSFC, and NOx emissions (all to be minimized). One challenging factor was the presence of hidden constraints [17]: about 66% of the initial design of experiments did not converge and therefore yielded invalid design points. By the end of the optimization, this was reduced to only 8% of the population, showing that NSGA2 indeed is robust in the presence of hidden constraints.

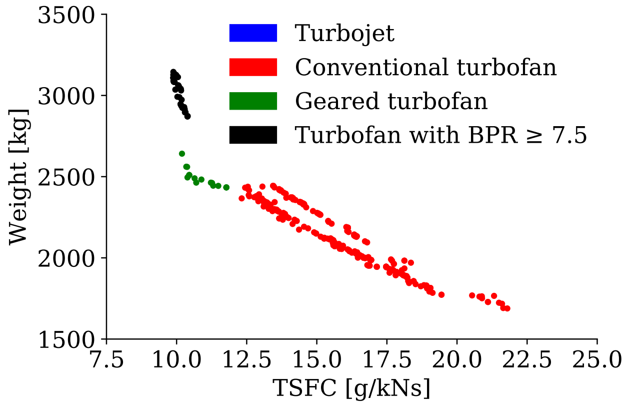


Figure 9: Pareto front of TSFC, weight and NOx emissions for the jet engine architecting problem. Figure reproduced from [7].

Fig. 9 shows results of the optimization: conventional turbofan architectures are better at reducing weight, high bypass ratio turbofans achieve the lowest TSFCs and levels of NOx emission, and geared turbofans provide a good compromise between the three objectives. This jet engine architecting problem demonstrates that it is possible to perform multi-objective architecture optimization. The method demonstrated here is flexible, however comes at the price of spending more time before results can be generated compared to manually formulating several architectures directly using the evaluation code, a trend observed MBSE in general [15]. When it comes to reducing expert bias in system architecting, this problem shows that bias is moved from architecture selection to the selection of architectural choices and performance metrics. Architecture selection is an objective step, because comparison between architectures is quantitative. However, choice and metric selection still involves prior decision making, mostly coming from time and project constraints related to the implementation of the choices and metrics.

## 5 Conclusions and Outlook

Systematically exploring and optimization system architectures can help non-biased search for novel architectures in large design spaces subject to the combinatorial explosion of alternatives. Four aspects that enable the optimization of system architectures are discussed: formalization of the architecture design space, systematic exploration using optimization algorithms, interpretation of the architecture model into a simulation model, and quantitative evaluation of architecture performance.

The first step to generate system architectures is to formally model the architecture design space. The architecture design space can be modeled using the Architecture Design Space Graph (ADSG), practically implemented using the web-based ADORE tool.

Architecture optimization problems can have hierarchical and mixed-discrete design variables, and can feature multiple objectives due to conflicting stakeholder needs. These kind of problems can be solving using evolutionary algorithms (e.g. Genetic Algorithm, NSGA2). Surrogate-Based Optimization (SBO) algorithms can also be applied, and generally need less function evaluations that evolutionary algorithms. Dealing with hidden constraints is an ongoing topic of research for SBO algorithms.

Quantitative evaluation of architecture instances, as needed for optimization, is enabled by converting architecture instances to simulation models, and by having sufficiently flexible simulation tools for representing the great variety of system architectures typically seen in architecture optimization problems. Architecture translation can be achieved using object-oriented translation or central data repository synchronization. The former is a Python-based approach using class factories that instantiate custom classes based on elements in the generated architectures. The latter is compatible with MDO workflows that use a central data repository for communicating data between engineering disciplines and uses Data Schema Operations (DSOs) to modify the central data repository based on elements in the generated architectures. Several requirements to the evaluation environment are discussed as well.

The four aspects are demonstrated using a jet engine architecting problem. It is shown how architecture optimization is enabled in this architecting problem, and how flexibility and modularity of the evaluation code is maintained. A Pareto front of optimal engine architectures is found.

Next steps for research include verifying that ADORE can model all relevant architectural choices by applying it to more realistic engineering problems. On the optimization side, SBO algorithms need to be extended to correctly handle hidden constraints. On the evaluation side, support most be strengthened for translation to central data repositories and for multiple MDO workflows.

## 6 Acknowledgments

The research presented in this paper has been performed in the framework of the AGILE 4.0 project (Towards Cyber-physical Collaborative Aircraft Development) and has received funding from the European Union Horizon 2020 Programme under grant agreement n∘ 815122.

## References

[1] M. Alder, E. Moerland, J. Jepsen, and B. Nagel. Recent advances in establishing a common language for aircraft design with CPACS. In *Aerospace Europe Conference*, 2020.

[2] J.H. Bussemaker, N. Bartoli, T. Lefebvre, P.D. Ciampa, and B. Nagel. Effectiveness of surrogate-based optimization algorithms for system architecture optimization. In *AIAA AVIATION 2021 FORUM*, Virtual Event, August 2021.

[3] J.H. Bussemaker, L. Boggero, and P. D. Ciampa. From system architecting to system design and optimization: A link between mbse and mdao. *INCOSE International Symposium*, June 2022.

[4] J.H. Bussemaker and P.D. Ciampa. *Handbook of Model-Based Systems Engineering*, chapter MBSE in Architecture Design Space Exploration (in review). Springer, 2021.

[5] J.H. Bussemaker, P.D. Ciampa, and B. Nagel. System architecture design space exploration: An approach to modeling and optimization. In *AIAA AVIATION 2020 FORUM*, Virtual Event, June 2020.

[6] J.H. Bussemaker, P.D. Ciampa, and B. Nagel. System architecture design space modeling and optimization elements. In *32nd Congress of the International Council of the Aeronautical Sciences, ICAS 2020*, Shanghai, China, September 2021.

[7] J.H. Bussemaker, T. De Smedt, G. La Rocca, P.D. Ciampa, and B. Nagel. System architecture optimization: An open source multidisciplinary aircraft jet engine architecting problem. In *AIAA AVIATION 2021 FORUM*, Virtual Event, August 2021.

[8] P. D. Ciampa and B. Nagel. AGILE paradigm: The next generation of collaborative mdo for the development of aeronautical systems. *Progress in Aerospace Sciences*, 119, November 2020.

[9] P.D. Ciampa and B. Nagel. Accelerating the development of complex systems in aeronautics via mbse and mdao: a roadmap to agility. In *AIAA AVIATION 2021 FORUM*, Virtual Event, June 2021.

[10] C.P. Frank, R. Marlier, O.J. Pinon-Fischer, and D.N. Mavris. An Evolutionary Multi-Architecture Multi-Objective Optimization Algorithm for Design Space Exploration. In *57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, number January, pages 1–19, Reston, Virginia, January 2016.

[11] E.S. Hendricks and J.S. Gray. pyCycle: A tool for efficient optimization of gas turbine engine cycles. *Aerospace*, 6(8):87, aug 2019.

[12] D. Horn, J. Stork, N. Schüßler, and M. Zaefferer. Surrogates for hierarchical search spaces. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, July 2019.

[13] K. Kernstine, B. Boling, L. Bortner, E. Hendricks, and D. Mavris. Designing for a green future: A unified aircraft design methodology. *Journal of Aircraft*, 47(5):1789–1797, September 2010.

[14] A.M. Madni. Novel options generation. In *Transdisciplinary Systems Engineering*, pages 89–102. Springer International Publishing, October 2017.

[15] A.M. Madni and S. Purohit. Economic Analysis of Model-Based Systems Engineering. *Systems*, 7(1):12, February 2019.

[16] D. Mavris, C. de Tenorio, and M. Armstrong. Methodology for Aircraft System Architecture Definition. In *46th AIAA Aerospace Sciences Meeting and Exhibit*, number January, pages 1–14, Reston, Virigina, January 2008. American Institute of Aeronautics and Astronautics.

[17] J. Müller and M. Day. Surrogate optimization of computationally expensive black-box problems with hidden constraints. *INFORMS Journal on Computing*, 31(4):689–702, oct 2019.

[18] R. Priem, N. Bartoli, Y. Diouane, and A. Sgueglia. Upper trust bound feasibility criterion for mixed constrained bayesian optimization with application to aircraft design. *Aerospace Science and Technology*, 105:105980, October 2020.

[19] M.N. Roelofs and R. Vos. Correction: Uncertainty-Based Design Optimization and Technology Evaluation: A Review. In *2018 AIAA Aerospace Sciences Meeting*, Reston, Virginia, January 2018.

[20] A. Tran, M. Eldred, S. McCann, and Y. Wang. srmo-bo-3gp: A sequential regularized multi-objective constrained bayesian optimization for design applications. July 2020.