A Generic Parametric Modeling Engine Targeted Towards Multidisciplinary Design: Goals and Concepts

Jan Kleinert[1] , Anton Reiswich[2] , Mladen Banović[3] , Martin Siggel[4]

[1]Institute for Software Technology, German Aerospace Center (DLR), Cologne, Germany, jan.kleinert@dlr.de
[2]Institute for Software Technology, German Aerospace Center (DLR), Cologne, Germany, anton.reiswich@dlr.de
[3]Institute of Software Methods for Product Virtualization, German Aerospace Center (DLR), Dresden, Germany, mladen.banovic@dlr.de
[4]Institute of Propulsion Technology, German Aerospace Center (DLR), Cologne, Germany, martin.siggel@dlr.de

Corresponding author: Jan Kleinert, jan.kleinert@dlr.de

Abstract. This paper presents the design concept of a generic parametric modeling engine that is completely decoupled from geometry generation. Driven by requirements extracted from preliminary multidisciplinary airplane design, the presented software architecture provides a platform that enables an interplay of different modeling and simulation tools on the one hand, and their efficient execution in a parametric tree on the other hand. An integrated plugin system allows users to define custom plugins exposing arbitrary types and functions. All geometric functionality is provided via plugins, decoupling it entirely from the parametric engine. First, we specify the goals that the software framework needs to fulfill, elaborating on the requirements encountered in early aircraft design. Then, we describe the software architecture and its modules, realized as a C++ library. As such, the software is a back-end that can be used by third party developers to create user-friendly and interoperable tools. The core of the framework is a parametric engine called `grunk` with its integrated plugin system and serialization functionality. The key feature of `grunk` is the possibility for users to define custom types in plugins and their use in the parametric tree. Geometric modeling functionalities are provided through the plugins `grocc` and `geo`: the first integrating OpenCascade Technology's functionalities and the latter extending it. A major feature on the geometry side is the provision of derivatives through algorithmic differentiation, making the framework particularly suitable for gradient-based optimization applications. Finally, we demonstrate the use of the software via examples and show the results.

Keywords: multidisciplinary design, parametric modeling, CAD engine, software architecture, geometric sensitivities
DOI: https://doi.org/10.14733/cadaps.2024.424-443

1 INTRODUCTION

Geometric modeling plays a central role in the early multidisciplinary design process of an airplane. This process involves an interplay of several engineering teams using various field-specific modeling and simulation tools. Naturally, such an environment introduces the following five software-related requirements:

(1) Since the involved parties working on the same aircraft need a common parametric geometry description for their optimization purposes, it is essential that this geometry is generated consistently and reproducibly within each department. To ensure this, each department should use the same software for generating the geometry.

(2) A common platform is needed for the interplay of tools used by different teams and their integrability in a modeling pipeline. This platform should enable the evaluation of chained computations in a parametric tree. Moreover, it should be straightforward to create interfaces for existing tools.

(3) Re-computation of the parametrically chained tools after a variation of initial parameters should be efficient, since the application of the involved tools is often computationally intensive and many model variations must be evaluated during the design process.

(4) Geometric sensitivities (e.g. derivatives of surface point coordinates with respect to design parameters) should be made accessible by using an algorithmically differentiated geometric library to support the design evaluation and its optimization using efficient gradient-based methods.

(5) Platform-independent free and open-source software libraries (FOSS) are advantageous in the context of research, as they are accessible to everyone, can be extended to meet specific needs and enhance reproducibility.

In this study, we present the design concept of a generic parametric modeling engine that is completely decoupled from the geometry generation, contrasting it to the existing FOSS geometric modeling tools and libraries available, see Section 1.2. This design allows the fulfillment of not just the *geometric requirements* (1) and (4), but also the *platform related requirements* (2) and (3) within a single framework.

Rather than providing a fully-featured CAD-tool, the goal is to provide an integrable modeling backbone that can be used by third party developers to easily create user-friendly and interoperable tools. The framework is currently under development and will be released in the near future under a permissive open-source license allowing commercial use. The feasibility and efficacy of the software design is demonstrated using an early implementation.

The two scientific contributions of this work are a novel software architecture for generic parametric design and the support for gradient-based shape optimization by means of algorithmic differentiation.

1.1 Targeted Usage Scenarios and Scope

In our projects, geometry is usually modeled in three different ways:

- It is performed by using a programming language. This includes C++, e.g. using geometry libraries such as OpenCascade Technology (OCCT) [18] as well as interpreted scripting languages, such as Python.
- In addition, domain-specific data models such as the Common Parametric Aircraft Configuration Schema (CPACS) [2] are used to describe geometries of technical systems in ASCII configuration files.
- Finally, the geometric modeling is done in various GUI applications.

The presented framework is designed in such a way, that it reflects these usage scenarios. The parametric engine is a C++ library with a small number of dependencies. We then provide Python bindings in order to access the full functionality via a scripting language. This low-level implementation in C++ and the provision of the Python bindings make the framework integrable into various environments. Users can write plugins to integrate data models and the framework design opens its use in various GUI-based modeling applications.

1.2 State-of-the-art of Free CAD Tools and Libraries

Prior to the development of this framework, a number of free parametric CAD tools and libraries were investigated for their potential to serve as a base instead of writing a new parametric modeling engine. Key features for evaluation include: availability as a (C++) library, licensing, a generic parametric engine that can use non-geometric objects, automatic differentiation to support geometric sensitivities and optimization, support for metadata, and availability of a plugin system to extend functionality.

In this section, five free CAD software tools with permissive licenses are evaluated: CadQuery [6], FreeCAD [9], ESP [12], OCCT [18], and Salome Shaper [23], based on the above-mentioned requirements.

Availability as a C++ library We want to exchange the backend of our existing CAD-like geometry generators, without changing their API which is used in several projects. This cannot be easily done by moving the whole ecosystem to a new platform. Furthermore, commercial CAD systems support the integration of C++ libraries via plugins. CAD interoperability with the framework is only possible if it is available as a standalone library, ideally written in C++ or similar compiled languages. In short, a parametric geometry kernel is required. Of the software listed above, only OCCT and ESP / EGADS can be used as a CAD kernel. FreeCAD does have a Python API, but this can only be used from within the FreeCAD application.

Generic parametric engine Some geometric features, such as the number of compressor stages in an aircraft engine, are not known at the outset of multidisciplinary design and optimization. They are only available after the first thermodynamic-based design in the beginning of the analysis. The geometric results (number of stages) depend on initial parameters such as thrust requirements. Therefore, it is desirable to integrate these non-geometric parameters into the parametric tree, which have a major impact on the resulting design. Therefore, a CAD software should also allow non-geometric/generic parameters and dependent variables in the parametric tree. From the above-mentioned software list, only FreeCAD seems to allow non-geometric parametric features.

Gradient calculation / algorithmic differentiation (AD) In a multidisciplinary design optimization, a chain of tools—including geometry generation, meshing, simulation, and post-processing—is repeatedly executed to optimize an objective function, such as minimizing the fuel consumption of an aircraft. This is typically computationally intensive, therefore gradient-based optimization methods are desired for such applications due to their efficiency. To employ these methods, derivative information is required from each tool included in the optimization chain. In Computational Fluid Dynamics (CFD), the adjoint approach is considered as state-of-the-art [10, 13, 19, 25] for computing derivatives of an objective function with respect to perturbation of mesh points. Since mesh points successively depend on a CAD parametrization, one requires geometric sensitivities to assemble the total gradient of the objective function with respect to CAD design parameters. Hence, a CAD software should be able to provide these sensitivities, preferably by applying AD. In ESP, the derivative computation is provided within the EGADS library by applying analytic differentiation to geometric primitives, while finite differences (FD) are used for more complex geometries. Nevertheless, an existing code for geometric modeling can be adapted to support the derivative computation using AD tools, which has been demonstrated on the OCCT library [3].

Metadata Due to its use in optimization, the geometry generation also needs to be automated. To facilitate subsequent mesh generation, metadata needs to be attached to topological entities such as an inflow plane to add boundary conditions, helper geometries for boundary layers, etc. This metadata should also be preserved during geometry modifications, as it is typically attached to geometric models during initial creation and not to the final geometric product. ESP allows flexible use of topological metadata. FreeCAD and Salome Shaper have some support for metadata, for instance names and colors. However, metadata is not preserved during model modification in FreeCAD. The limitations of FreeCAD come from the underlying OCCT geometry kernel. OCCT provides the basic functionality to track changes to models. This could be used to preserve the metadata after changes have been made to the models. However, this must be implemented for each application.

Plugin support The software should be modular and easy to extend. This allows third parties to add more geometric modeling algorithms or parametric model generators for special parts. A plugin system allows the software to be extended without recompilation and also allows the creation of commercial add-ons to free software. Furthermore, using native compiled plugins is advantageous from a commercial perspective, as only the compiled binaries can be distributed to customers. FreeCAD has good plugin support, with a wide range of model generators. These "add-ons" must be developed in Python. Another way to extend FreeCAD are workbenches that can be implemented in Python or C++. Salome and CADQuery also use a Python based plugin system. In fact, Salome Shaper is a plugin collection of Salome. ESP uses a more traditional plugin system with compiled plugins that allow parametric model generators to be added to ESP.

In summary, none of the above-mentioned software fully meets our requirements, although ESP comes closest. OCCT has proven to be a solid foundation for many CAD-like software, including ESP, so we decided to make it the foundation for the geometric modeling functionality of our software. The geometry generation functionality is provided by plugins. While our work focuses on the geometry generation using OCCT, it is still possible to integrate other CAD kernels into the framework via the plugin interface.

1.3 Outline

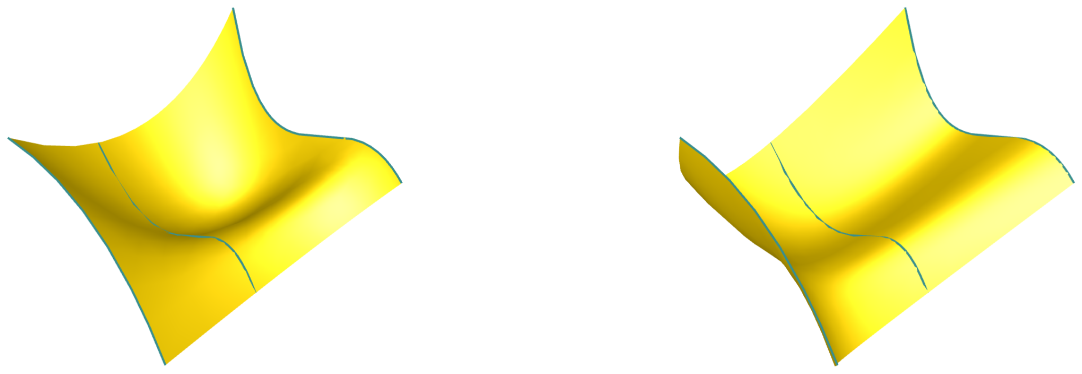
Section 2 elaborates in detail the requirements (1)–(4). Section 3 presents the current design of the framework targeting the requirements (2) and (3). Section 4 describes how the framework can be used for geometric modeling that satisfies the requirement (1). Section 5 elaborates how the geometric sensitivities are calculated using the algorithmic differentiation, thus satisfying the requirement (4). In Section 6 we demonstrate the feasibility of the framework based on the current implementation. Finally, Section 7 concludes our study and Section 8 provides an outlook.

2 REQUIREMENTS

This section provides more details and example scenarios for the requirements (1)–(4).

2.1 Geometric consistency

Requirement (1) addresses geometric consistency. In multidisciplinary preliminary aircraft design, the following problem might occur: The involved departments, although working on the same airplane, could use different software for geometric modeling. The involved parties usually need parametric, “non-dead“, descriptions of the aircraft's geometry in order to adjust certain design parameters through optimization processes. Now, since the implementation of geometric algorithms (e.g. for interpolating a set of curves by a surface, see Fig. 1) might vary among different CAD systems, sharing the same parametric description of the geometry



(a) A valid interpolating surface, where the second curve is an iso-curve of the resulting surface at parameter $v = 0.33$.

(b) A valid interpolating surface, where the second curve is an iso-curve of the resulting surface at parameter $v = 0.66$.

Figure 1: Three curves are lofted to a B-spline surface with parametric domain $[0, 1] \times [0, 1]$. The B-spline surface satisfying the interpolation requirement for the three curves is not unique.

with another department using a different CAD system may lead to a deviation in the resulting geometries. In addition, the different handling of geometric and floating point tolerances may add to deviations in the final geometries, too.

To avoid these deviations, and therefore achieve geometric consistency, all involved departments should use the same CAD system.

We note that the translation of parametric geometry definitions between different CAD systems [14, 22] is not in the scope of this study.

2.2 Common Platform

Requirement (2) addresses a common platform to enable the interplay between different tools for geometric modeling, analysis, simulation and optimization used by the involved engineering teams. The interplay should be realized by coupling the different tools in a chained way on the common platform, i.e. in a parametric tree. The computational nodes of the created tree can contain both functionalities from different activities (as for example geometric modeling, analysis, simulation, optimization), as well as functionalities from different CAD systems. In order to create an interface, allowing the integration of the tools in the platform, a plugin system is required.

It is common that the geometry generation is preceded by a conceptual design phase providing initial parameters or viable parameter ranges for the geometry. As an example, a mission description for an aircraft influences its total size and weight or the type and number of engines.

Furthermore, the geometry generation may depend on simulation results or other expensive computations. Taking the preliminary aircraft design for inspiration once more, the Vortex Lattice Method (VLM) is a technique that calculates force distribution on lifting surfaces at a given flight regime. The results of such simulations can be used as an input of the design and/or sizing of the load-bearing structure of a wing. The publications [11, 16, 17, 26] shed light on how the simulation and geometry generation are typically intertwined in multidisciplinary design analysis and optimization (MDAO) workflows and how they are influenced by a preceding conceptual design phase.

Overall, the parametric design of the geometry of an aircraft involves a large set of tools. However, currently most parametric modeling frameworks limit the set of modeling tools to geometry generation and manipulation functions. Both conceptual design and simulation tools play an important role in geometry generation and should not be excluded from parametric design.

2.3 Efficient Re-computation

Considering that the design process involves many tools (possibly also simulation runs), its re-computation can be very time and memory consuming. At the same time, many model variations are created in the design phase to either manually or automatically explore the design space.

Coupling tools by writing input and output files to a hard drive should be minimized. Already computed intermediate results, unaffected by the parameter change can be cached and re-used. Reversely, only the caches of those (intermediate) results that depend on the changed parameter shall be invalidated. Finally, a geometry should be generated lazily, i.e. only those entities explicitly queried should be generated on demand. As an example, consider once more a complex model of a full aircraft. An engineer working on the structural integrity of the horizontal tailplane should be able to query and manipulate only this part of the model, without having to generate the entire model, including e.g. the interior of the cockpit.

2.4 Geometric Sensitivities

To include a parametric CAD model into a gradient-based shape optimization loop, one requires the calculation of geometric sensitivities. Typically, this information is not available in a CAD software.

One possibility to evaluate this information is by using inaccurate finite differences. In this case, one has to carefully choose a step size (perturbation) for each design parameter in order to limit truncation errors. Additionally, there is a risk of topology changes when constructing perturbed geometries—a possible solution to tackle this problem is elaborated in [1]. Nonetheless, the FD approach fits for applications where closed-source CAD systems are used.

On the contrary—if the CAD sources are available—one can apply AD to compute derivatives that are exact up to machine accuracy. By using this approach, one avoids the risk of topology changes because AD does not require any model perturbations. Furthermore, as observed on the differentiated OCCT kernel in [3], AD achieves a much better performance than FD when computing derivatives with respect to many design parameters. Therefore, this method is used in this study.

3 SOFTWARE FRAMEWORK ARCHITECTURE

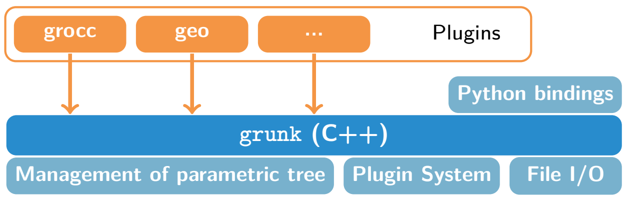


Figure 2: The `grunk` architecture.

The first building block of the framework is a software called `grunk`[1], a common platform addressing the requirements (2) and (3). `grunk` unites three important features, where each is implemented in a separate library that can be used independently of `grunk`, see Fig. 2.

The first main feature is the possibility to create generic parametric trees that chain algorithms. Any data types can be used as nodes, i.e. as initial parameters, intermediate or final calculation results. Furthermore, any function can be used as an algorithm. The evaluation of a parametric tree is lazy, i.e. nodes get computed only when

[1]At this early stage, software names are preliminary and are subject to change.

their value is queried. Moreover, it supports an automatic invalidation of the cache in case of a change of the initial parameters.

The second main feature of grunk is its plugin system, allowing each user to write their own plugins and import, as well as share their tools, thus providing the common platform. Plugins can be managed and shared via a dedicated plugin manager. In this way grunk provides a highly modularizable and extendable workbench. The platform grunk itself does not provide any built-in algorithms to be used as part of a parametric tree. The idea is to provide all functionality, including geometric modeling tools, through plugins.

The third main feature of grunk is its built-in serialization functionality, enabling the user to write the parametric tree to a human readable YAML-file called a grunk recipe.

3.1 Parametric Engine

Management utilities for the parametric modeling are implemented in a C++ library called parametric. It is a header-only library that allows the parametric lazy evaluation of functions.

To enable the parametric modeling, the model-generation history is stored as a directed acyclic graph (DAG), the so-called *parametric tree*. Since all initial parameters, intermediate results and final computation results are stored as nodes in the DAG, the data dependencies can be tracked. The DAG effectively is a recipe for the model-generation.

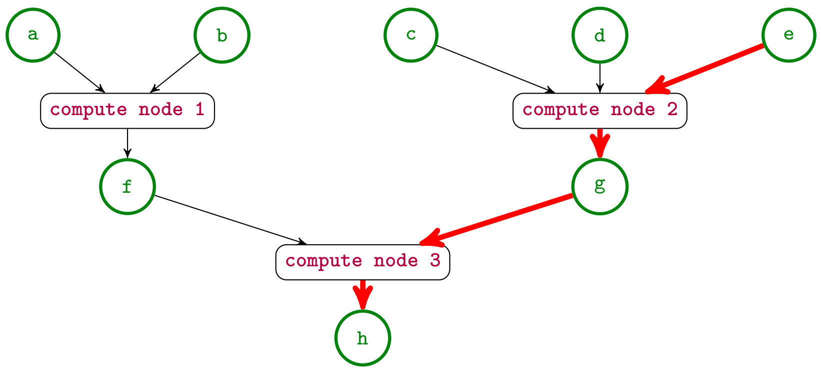


Figure 3: A parametric tree. Circular nodes are parameter nodes, rectangular nodes are the so-called compute nodes. The bold red arrows depict the invalidation path for parameter e. The caches of all parameter nodes along this path will be invalidated if a user changes the value of e.

This DAG consists of two kinds of nodes. The first kind is called a parameter node. It is a wrapper type that holds a cache to store parameter values. The second node kind is called a compute node. It is a wrapper type around any kind of a callable object such as a function pointer, a functor or a lambda expression. The only requirement is that the function is pure. For our purposes, we consider a function to be pure if it does not modify any of its input arguments. This is required to prevent circular dependencies.

Compute nodes own their parents, which are parameter nodes representing the input arguments for the computation. Functions with multiple output values are supported, so a compute node can have any number of child nodes, all of which are parameter nodes.

If a parameter node has no parents, it is an independent input parameter. If a parameter node has no children, it is a dependent output parameter. Otherwise, it is an intermediate calculation result. A parameter node can have at most one parent, namely a compute node. Just like a compute node owns its parents, a parameter node owns its parent compute node. A parameter node can have any number of children, all of which are the compute nodes using the parameter node as one of its arguments.

Assembling the parametric tree does not involve actually evaluating the compute nodes. Once the actual value of a parameter node is queried, the node can trigger it's own evaluation by calling it's parent compute node, which in turn recursively triggers the evaluation of its inputs. Once a node has been evaluated, it will store its value in a cache. The next time the parameter node's value is queried, the value will be returned from cache rather than re-evaluating the entire parametric tree. Thus, all nodes of the parametric tree are evaluated lazily, i.e. only once their value is queried. Intermediate nodes that are not involved in the calculation of the queried node's value will not be evaluated.

Only the values of independent input parameters can be changed manually. The change of the value of an independent input parameter will automatically trigger the recursive invalidation of the caches of its descendent nodes. The next time a user queries the value of one of its descendent nodes, part of the parametric tree must be recomputed because it contains invalidated caches.

Fig. 3 shows an example of this automatic invalidation. Assume the caches of all nodes are valid, because the value of the parameter node `h` has previously been queried by a user, triggering the evaluation of the parametric tree. Changing the value of node `e` will trigger the automatic invalidation of the caches of the parameter nodes `g` and `h`. The next time the value of `h` is queried, the `compute node 2` and the `compute node 3` must be recomputed, but not the `compute node 1`, because the cache of `f` has not been invalidated by the change of `h`.

3.2 Plugin System

`grunk` supports the use of functions and types exposed in a plugin that can be loaded at runtime. The plugin system is founded on a dynamic type system. Plugins can be created and installed via a command line tool.

3.2.1 Dynamic type system

`grunk` is a C++ library and C++ is statically typed, which means that all types must be known at compile time of the program and little useful type information is available at runtime. To overcome this problem, `grunk`'s plugin system is established on a dynamic type system. This dynamic type system is realized in a custom runtime type reflection library called `reflect`.

A plugin author must register all functions in `reflect`'s function registry in an initialization function call. The inputs and outputs of each function are wrapped in a type-erased wrapper class called `DynamicObject`. This class can hold any type and serves as a handle that `grunk` can pass around to the functions supplied by the plugins.

Note that `grunk` is agnostic of the types passed from one plugin to another. For two plugins to work together, they must either be implemented to be compatible to each other or an additional mediator plugin must be used that converts the inputs and outputs of each function. `grunk` itself does not provide any built-in functionality to facilitate the code-coupling of two plugins.

In addition to the type-erased handle, a `DynamicObject` holds a pointer to a type descriptor instance stored in a static type registry. A type descriptor instance exists for every type used in the function signatures of the registered functions. By default, it does not hold much useful information. Plugin authors have the possibility to enrich this type descriptor instance by explicitly providing a string representation of the type name, a list of base classes, constructors, conversion functions, data members and member functions. All attributes will then be accessible at runtime. In addition, it is possible to register additional free functions as member functions to enhance the interface of existing types in the dynamic type system.

The benefit of this additional runtime type information is two-fold. First, `grunk` gains the opportunity to convert function arguments, perform polymorphic casts and perform a simplified overload resolution at runtime. Secondly, constructors, `const` member functions, and retrieval of data members can be used as compute nodes in the parametric tree.

```cpp
#pragma once

struct Base
{
    void foo();
};

struct Scalar : public Base
{
    Scalar(double);
    double value;
    operator double() const;
};

Scalar add(Scalar, Scalar);

void bar(bool);
void bar(int);
```

(a) Scalar.hpp: Example C++ types and functions that shall be made available at runtime via grunk's dynamic type system.

```cpp
#pragma once
#include "grunk/grunk.hpp"

class PluginA : public grunk::IPlugin
{
    std::string name() const override;
    std::string version() const override;
    void init() const override;
};
// macro to properly export PluginA
GRUNK_REGISTER_PLUGIN(PluginA)
```

(b) PluginA.hpp: Declaration of a grunk plugin called PluginA.

```cpp
#include "PluginA.hpp"
#include "Scalar.hpp"

std::string PluginA::name() const {
    return "PluginA";
}

std::string PluginA::version() const {
    return "1.0.0";
}

void PluginA::init() const {

    register_type<Base>("Base")
    .add_member_function(&Base::foo, "foo");

    register_type<Scalar>("Scalar")
    .add_base<Base>()
    .add_constructor<double>()
    .add_conversion<double>()
    .add_data_member(&Scalar::value, "value")
    .add_member_function(
        [](Scalar const& d){
            YAML::Node out(d.value);
            return out;
        },
        "serialize"
    )
    .add_member_function(
        [](YAML::Node const& y){
            return Scalar(y.as<double>());
        },
        "deserialize"
    );

    register_function(&add, "add");

    register_function<void(*)(bool)>(&bar, "bar");
    register_function<void(*)(int)>(&bar, "bar");
}
```

(c) Definition of PluginA from 4b. Types and functions declared in 4a are registered in the dynamic type system in the body of the PluginA::init function.

Figure 4: Example implementation of a grunk plugin. Types and functions are made available to the dynamic type system by registering them in the init function of a plugin.

Fig. 4c gives an example of how a grunk plugin would expose C++ types and functions via the dynamic type system in the plugin's init function. In lines 14–15 the type Base is registered. The type Scalar is registered in lines 17–34. Base is registered as a base class of Scalar enabling grunk to perform polymorphic casts at runtime. In addition, the member function foo of the base class can now be called for Scalar instances in the dynamic type system. A constructor is registered so that instances of this type can be created in the dynamic type system. A conversion operator is registered so that implicit argument conversion is possible. The data member value is registered, so that retrieval of this data member can be used as a compute node in a parametric tree. Finally, in lines 22–34, two new member functions are added to the type descriptor, that do not exist for the C++ types. These are special member functions used by grunk for reading and writing to a parametric file, see Section 3.3. In lines 36–39 free functions are registered. Note that it is possible to register overloaded functions with the same name—reflect will perform overload resolution at runtime.

Using this dynamic type system comes at the cost of a small runtime overhead due to type-erasure techniques involving virtual function calls. We expect the overhead to be small. In addition, using the dynamic type system is optional: It is only needed if the user wants to use either the plugin system or the reading and writing of grunk recipes.

3.2.2 Plugin Manager

grunk provides a plugin manager as a command line tool, which enables easy installation of plugins from remote sources and assists in the writing and sharing of plugins.

The plugin manager is built as a thin layer on top of the API of the Conan package manager [7] and thus all managed plugins must provide a valid Conan recipe. Conan already performs compatibility checks based on semantic versioning. With Conan's package id, we have a mechanism in place to decrease the chance of having ABI[2] incompatibility between grunk and any of the loaded plugins. There can be several parallel installations of a plugin having different versions or options. The plugin manager provides a possibility of downloading and uploading packages from remote sources. Custom remotes can be added, e.g. for hosting in-house, commercial or special purpose plugins. If a suitable binary for a plugin is not available in any of the remote sources, it will be automatically built from source on the host machine, if the source is available as part of the Conan package.

Currently, only C++ plugins are supported. The plugin manager has two modes for assisting in the creation of plugins. In the first mode, a new empty plugin can be created from a template. This new plugin must be adapted by the plugin author. The second mode is aimed at writing plugins that expose types and functions of already existing C++ libraries. In addition to creating a template plugin, it includes a source-to-source code generator. This code generator parses a list of header files provided in a configuration file for types and functions and automatically generates the code for registering these types and functions in reflect's dynamic type system. Using this functionality, bindings of existing (potentially large) C++ libraries to the grunk plugin system can be generated with a high degree of automation and customization.

3.3 File I/O

grunk can be used with and without the dynamic type system. If the dynamic type system is used, parametric trees can be written to a human-readable ASCII file, called a grunk recipe. grunk recipes are a description of a feature tree in the YAML format. The current implementation is based on the C++ library yaml-cpp [4]. Fig. 5 demonstrates how grunk recipes can be created and read.

When passing a node or a list of nodes from a parametric tree to the function grunk::write, grunk will recursively write all nodes required to reconstruct the input nodes. It creates a YAML document containing three blocks named uses, parameters and steps:

[2] Application Binary Interface

```cpp
// load pluginA.dll from disk
grunk::get_plugin_registry().prepend_path("C:\plugin_dir\");
grunk::get_plugin_registry().load_all();

// assemble parametric tree
auto x = grunk::Feature("x", "pluginA::Scalar", 4.3);
auto y = grunk::Feature("y", "pluginA::Scalar", 3.3);
auto a = grunk::action("a", "pluginA::add", x, y)->output();
// write grunk recipe to disk
grunk::write("model.grr", a);
```

(a) Example C++ code: Creating a parametric tree and writing to a grunk recipe `model.grr`. This file will contain instructions for reconstructing the variable `a` from the parameters `x` and `y`.

```yaml
uses:
  grunk: 0.1.3
  pluginA: 1.0.0

parameters:
  x: !<pluginA::Scalar> 4.3
  y: !<pluginA::Scalar> 3.3

steps:
 - !<pluginA::add> [[a], [x, y]]
```

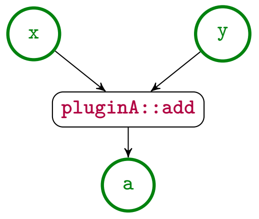
(b) The grunk recipe `model.grr` created in code 5a.

```python

# load pluginA.so and pluginB.so from disk
grunk.get_plugin_registry().prepend_path("~/plugin_dir/")
grunk.get_plugin_registry().load_all() #pluginA, pluginB

# Read grunk recipe and append it
a = grunk.read("model.grr")["a"]
z = grunk.Feature("z", "pluginA::Scalar", 2.0)
b = grunk.action("b", "pluginB::multiply", a, z).output()

# write grunk recipe to disk
grunk.write("model2.grr", b)

```

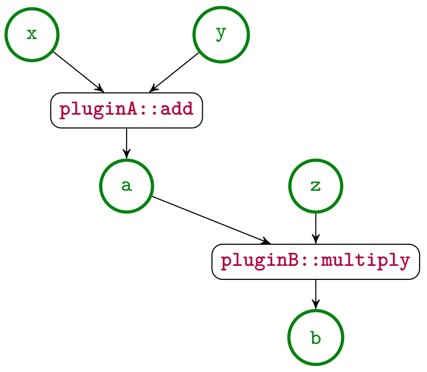
(c) Example Python code: Reading grunk recipe 5b from code 5a, modifying it and writing to another grunk recipe `model2.grr`.

```yaml
uses:
  grunk: 0.1.3
  pluginA: 1.0.0
  pluginB: 2.0.1

parameters:
  x: !<pluginA::Scalar> 4.3
  y: !<pluginA::Scalar> 3.3
  z: !<pluginA::Scalar> 2.0

steps:
 - !<pluginA::add> [[a], [x, y]]
 - !<pluginB::multiply> [[b], [a, z]]
```

(d) The grunk recipe `model2.grr` created in code 5c.



(e) Parametric tree created in code 5a respective grunk recipe 5b.



(f) Parametric tree created in code 5c respective grunk recipe 5d.

Figure 5: Parametric trees can be written to or read from a human-readable file format, the so-called grunk recipe.

uses: A list of all currently loaded plugins with their respective versions, as well as the `grunk` version.

parameters: A list of all independent parameters. This is a YAML dictionary, where the keys are the names of the parameter nodes. The values consist of two parts. The first is a YAML tag *(enclosed in angle brackets following an exclamation mark, !<tag>)*. This tag specifies the type of the parameter. The second part is a YAML node specific to that type. The type descriptor of the parameter's type, as specified in the YAML tag, must provide a static `deserialize` method to parse this YAML node and construct an instance of the type. Conversely for writing the file, the type descriptor must provide a non-static `serialize` method that serializes an instance of the type to a YAML node with enough information, to reproducibly reconstruct the instance, see Section 3.2.1. Providing `serialize` and `deserialize` methods is only required for types, which are used as independent parameters in the `parameters` block.

steps: A list of all compute nodes. The steps are ordered topologically, so that they can be performed from top to bottom without any undefined intermediate input nodes. A step consists of a YAML tag denoting the name of the function to be called. This name can correspond to a free function, a member function, a data member or a type name. In the latter case, it is interpreted as a constructor call. The YAML tag is followed by a list with two elements. The first is a list of names for the outputs and the second is a list of names for the inputs.

4 GEOMETRIC MODELING PLUGINGS

`grunk` itself is a parametric modeling engine, agnostic of the types and functions used for the parameter and compute nodes. Therefore, any CAD kernel with a C/C++ API can be used. Since `grunk` plugins must already use semantic versioning, the requirement (1) concerning geometric consistency is satisfied if all relevant plugins are available to all team members. For our projects, we provide two specific software packages for geometric modeling.

First, we create a `grunk` plugin called `grocc` that links against OCCT and exposes its types and algorithms. This allows the use of OCCT within the new parametric framework, without having to link and re-compile. This plugin is automatically generated using the source-to-source code generator introduced in Section 3.2.

Secondly, we collect and share geometric functionalities and algorithms developed across departments in a centralized geometry library geo, which is implemented in C++. This library is built on top of OCCT and it extends it to meet our specific geometric modeling requirements. This library can be used entirely independently of `grunk`. In order to use it within `grunk`, a `grunk` plugin of geo is created, which enables the use of the geometric functionalities as nodes in the parametric tree. geo is designed to be compatible with OCCT/`grocc`, see Fig. 7. Both `grocc` and geo can be used to model geometry within one and the same parametric tree.

The geometry computed in the parametric tree can now be parametrically modified either through variation of the high level input parameters of the tree, or through adding further nodes to the parametric tree. These nodes could take a geometric instance as input and create a modified version of it, by altering attributes of the geometric instance. This procedure enables the user to undertake low level parametric variations of the geometry, e.g. by changing the position of the control points of a B-spline curve, and returning the modified B-spline curve as an output feature. However one has to be careful when doing such operations, as the number and position of control points may depend on other parameters.

5 CALCULATION OF GEOMETRIC SENSITIVITIES

To meet the requirement (4), we provide the user a capability to calculate and retrieve derivatives in the plugins geo and `grocc`, that can be later applied in a gradient-based optimization loop. This feature is achieved by means of AD.

Since the plugins are built on top of the OCCT kernel, first its differentiation has to be tackled, after which one can proceed with the differentiation of plugins. For this purpose, the OCCT kernel version 7.6.2 is differentiated by integrating the AD software tool ADOL-C (*Automatic Differentiation by OverLoading in C++*) into its source code.

To differentiate a code with ADOL-C, the declaration types of all relevant *real* variables (e.g. `float`, `double`) have to be replaced with the ADOL-C data-type `adouble`. There are two implementations of the `adouble` class: *trace-based* and *traceless*, offering different ways of derivative computation. The latter is used in this study. It computes derivatives directly during the function (*primal*) evaluation by applying the *forward/tangent* mode of AD.

The `adouble` data-type is integrated into OCCT such that the alias `Standard_Real` (defined in the OCCT header file `Standard_TypeDef.hxx`) is changed from `double` to become the alias for `adouble`. Due to the complexity of the OCCT sources, this modification triggered a large number of compile and run-time issues that had to be resolved to complete the differentiation. More details about the certain difficulties faced during this process and their corresponding solutions can be found in [3]. Once completed, the AD-enabled OCCT provides geometric sensitivities for arbitrary parametrizations, that are accurate up to machine precision.

Next, the differentiation process of the `geo` and `grocc` plugins is triggered by linking them against the differentiated OCCT. Furthermore, it is important to continue using the `Standard_Real` alias as the declaration type of *real* variables defined in the plugin code, otherwise one cannot compile the sources. Additional compile-time errors had to be resolved due to the `adouble` integration. Nevertheless, the number of these errors is negligible comparing to the OCCT differentiation issues.

Finally, the integration of ADOL-C into the parametric engine `grunk` is straightforward (in terms of code changes) since `grunk` allows to transparently expose any data-type in a generic way. All computations of derivatives actually occur on the plugin level. Here, one has to register the `adouble` data-type and use it to define inputs (e.g. design parameters) and outputs (e.g. coordinates of surface points). After that, one can proceed to the derivative computation.

6 RESULTS

This section demonstrates the use of the new software framework based on the current implementation. We will show how `grunk` can be used for the parametric modeling in Section 6.1 and demonstrate the AD-enabled version of the geo library in Section 6.2.

6.1 Parametric Modeling

We will provide two examples of using the parametric modeling engine `grunk`. Section 6.1.1 depicts the usage of `grunk` solely for the creation of parametric trees without using the dynamic type system. Section 6.1.2 shows an example `grunk` recipe using the geometric modeling plugins `grocc` and `geo` introcuded in Section 4.

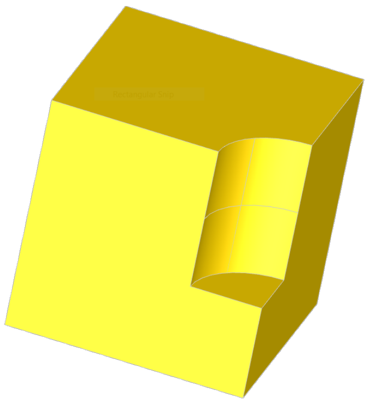
6.1.1 Lazy evaluation, caching and automatic invalidation

Figure 6 demonstrates how `grunk` can be used directly with an external geometry kernel like OCCT to leverage parametric modeling, lazy evaluation, caching and automatic invalidation. Note that neither the dynamic type system, nor the plugins `geo` and `grocc` are needed in this scenario.

In lines 4–19 three functions are defined, one for creating a box, one for creating a cylinder and one for creating a Boolean difference of two shapes. All functions print an output to console and forward to appropriate functions from the geometric kernel OCCT.

In lines 23–32 the parametric tree is defined. In lines 23–27 the independent input features are defined, namely the radius and height of a cylinder as well as the width, height and depth of a box. In lines 30–32 three

```cpp
// #include statements ommitted for brevity

// create cylinder and print to console
TopoDS_Shape make_cylinder(double radius, double height) {
        std::cout << "\tCreating Cylinder\n";
        return BRepPrimAPI_MakeCylinder(radius, height);
}

// create box and print to console
TopoDS_Shape make_box(double w, double h, double d) {
    std::cout << "\tCreating Box\n";
    return BRepPrimAPI_MakeBox(w, h, d);
}

// cut two shapes and print to console
TopoDS_Shape cut(TopoDS_Shape const& shape, TopoDS_Shape const& tool) {
    std::cout << "\tPerforming Cut\n";
    return BRepAlgoAPI_Cut(shape, tool);
}

int main() {
    // 1. define independent variables
    grunk::Feature radius("radius", 1.5);
    grunk::Feature height("height", 3.);
    grunk::Feature box_w("box_w", 5.);
    grunk::Feature box_h("box_h", 5.);
    grunk::Feature box_d("box_d", 5.);

    // 2. define compute nodes of feature tree
    grunk::Feature cyl = grunk::action("cyl", &make_cylinder, radius, height)->output();
    grunk::Feature box = grunk::action("box", &make_box     , box_w, box_h, box_d)->output();
    grunk::Feature res = grunk::action("res", &cut          , box, cyl)->output();

    // 3. demonstrate lazy evaluation and caching
    std::cout << "Querying result triggers lazy evaluation...\n";
    TopoDS_Shape design = res.value();

    std::cout << "Querying result again reads from cache...\n";
    BRepTools::Write(res.value(), "initial_design.brep");

    // 4. demonstrate automatic invalidation
    std::cout << "Changing radius of cylinder automatically invalidated dependent features...\n";
    radius.access_value() = 4.;

    std::cout << "Querying result again triggers recalculation of cylinder and cut, bot not box...\n";
    BRepTools::Write(res.value(), "changed_design.brep");
}
```



(a) grunk is used together with OCCT to create a feature tree and leverage lazy evaluation, caching and automatic invalidation.

```
Querying result triggers lazy evaluation...
        Creating Cylinder
        Creating Box
        Performing Cut
Querying result again reads from cache...
Changing radius of cylinder automatically invalidated dependent features...
Querying result again triggers recalculation of cylinder and cut, bot not box...
        Creating Cylinder
        Performing Cut
```

(b) The console output produced by the program from Fig. 6a.

Figure 6: C++ code demonstrating the use of grunk together with OCCT directly.

compute nodes are added to the tree via calls to the function `grunk::action`. One compute node represents the creation of a box, another the creation of a cylinder and the last the generation of the Boolean difference of the box and the cylinder. Note that until here, no compute node is evaluated. The parametric tree has only been assembled.

Lines 35–39 demonstrate lazy evaluation and caching. Nodes of the parametric tree are only calculated once their value is queried. This query will trigger the evaluation of the compute nodes, which recursively trigger the evaluation of their inputs. Once a node is evaluated, the result is stored in a cache. A subsequent evaluation of a node will not trigger a re-evaluation, but just return the value stored in the cache. This can be seen by the console output in Fig. 6b.

Lines 42–46 demonstrate automatic invalidation. The radius of the cylinder is changed, which triggers the invalidation of the caches of all nodes that depend on this parameter. Re-evaluation of the `res` node will require a re-evaluation of the compute nodes for the creation of the cylinder and the Boolean difference. The cache of the box has not been invalidated, so the associated compute node must not be re-evaluated as can be seen in the console output in Fig. 6b.

6.1.2 Using geometric modeling plugins at runtime

```
uses:
  grunk: 0.1.3
  geo: 0.1.2
  grocc: 0.1.0

parameters:
  naca2412: !<geo::PntList> [[1.00084, 0.001257, 0.], [0.975825, 0.006231, 0], ...]
  transform1: !<geo::Transformation> [[100.,   0., 0., 0.], ...]
  transform2: !<geo::Transformation> [[ 80.,   0., 0., 0.], ...]
  transform3: !<geo::Transformation> [[ 50.,   0., 0., 0.], ...]
  tol: !<double> 1e-8
  filename: !<String> "wing.brep"

steps:
 - !<geo::interpolate_points> [[base_curve], [naca2412]]
 - !<geo::transform> [[curve1], [base_curve, transform1]]
 - !<geo::transform> [[curve2], [base_curve, transform2]]
 - !<geo::transform> [[curve3], [base_curve, transform3]]
 - !<geo::interpolate_curves> [[wing_surf], [curve1, curve2, curve3]]
# geo is compatible with OCCT/grocc:
 - !<grocc::BRepBuilderAPI_MakeFace> [[wing_face], [wing_surf, tol]]
 - !<grocc::BRepTools::Write> [[sink], [wing_face, filename]]
```

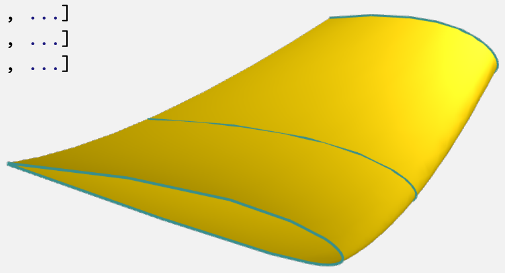


Figure 7: A `grunk` recipe making use of the geo plugin. A simple wing is created using geo and the result is exported using OCCT/`grocc` *(initial parameters are truncated for better readability)*.

Figure 7 shows a `grunk` recipe that uses the geometric modeling plugins `grocc` and `geo`. This recipe entails a feature tree for the generation of a very simple wing surface.

The important independent input parameters are a point list, representing a NACA airfoil and three 4×4 transformation matrices. The point list is interpolated to a curve, which is then transformed to three individual curves using the given transformation matrices. These curves are interpolated to the resulting surface. Then `grocc` is used to generate a `TopoDS_Face` using OCCT and write this face to disk in OCCT's native BREP format.

The grunk recipe from Fig. 7 uses geo types and functions for geometry generation and grocc for writing the generated geometry to disk. It can be parsed by grunk to reconstruct the parametric tree and evaluate any of its nodes. This does not require linking against OCCT, as all of its functionality is loaded at runtime via the plugin grocc. The geometry can be modified with the help of a large set of modeling tools without the need to re-compile anything.

6.2 Verification of Geometric Sensitivities

To verify the correctness of the AD-enabled geo plugin and OCCT kernel, we define a test-case as follows. There are three 3D-point sets and their (x, y, z) coordinates are considered as design parameters (inputs). Each point set is interpolated as a B-spline curve. Next, these B-spline curves are interpolated as a B-spline surface. Finally, points are computed on the B-spline surface and their (x, y, z) coordinates are considered as outputs.

The aim is to compute surface sensitivities, i.e. the derivatives of surface point coordinates with respect to the design parameters using AD and FD. A qualitative comparison of AD and FD surface sensitivities is presented in Fig. 8, where the y-coordinate of the marked point is defined as the design parameter (independent variable). As observable, they match to a very high extent. Moreover, the quantitative comparison of several surface sensitivities is presented in Table 1, showing mutual consent.

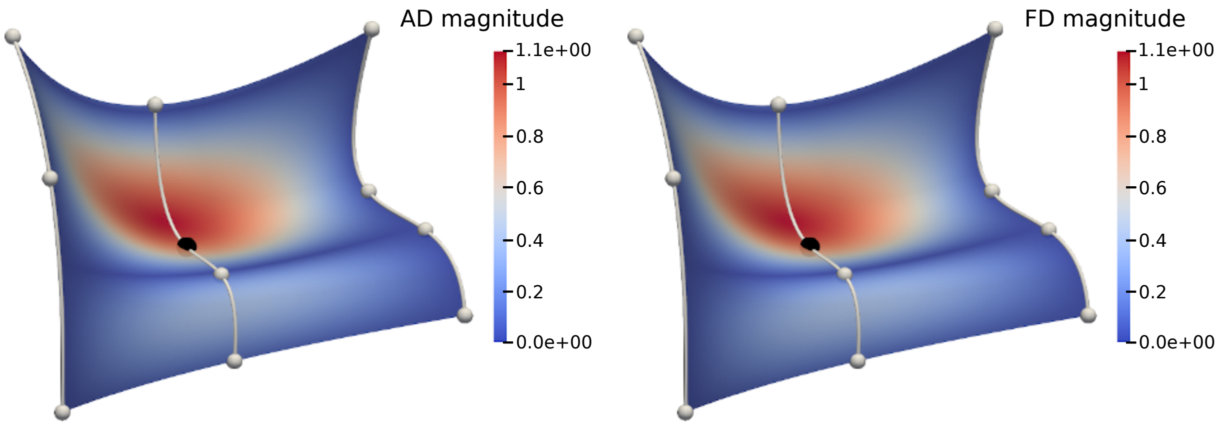


Figure 8: Surface sensitivities calculated with AD (left) and FD (right) with respect to the y-component of the marked interpolation point.

Table 1: AD and FD value comparison for several surface point y-coordinates (*maximal difference).

AD value	FD value	Absolute difference
4.3033**838653**e-04	4.3033**776542**e-04	6.21e-10
-5.00268**42183**e-02	-5.00268**44889**e-02	2.71e-09
1.92161**49433**e-02	1.92161**39791**e-02	9.64e-09
⋮	⋮	⋮
-2.08246**59267**e-01	-2.08246**68923**e-01	9.66e-08*

7 CONCLUSION

This study presents a framework for a generic parametric modeling engine with entirely decoupled geometry functionalities, which are provided as plugins. The motivation for this software architecture emerged out of a geometric modeling context. Its high modularity, generic architecture, and the strict decoupling of the geometric functionalities lift it from the context of geometric modeling and enable a flexible use in a large variety of other fields, that are characterized by the requirements of a common platform (2) and efficient re-computation (3).

We demonstrated the feasibility and usefulness of the software architecture based on its first implementation. The software is still in the early development stage. On the path to a mature and usable parametric modeling engine, several additional features are yet to be addressed.

8 OUTLOOK

This section provides an outlook for the next steps in the development of the software framework. We discuss planned features that are not yet implemented, usage scenarios and expected hurdles.

Persistent naming and metadata Persistent naming refers to the problem of associating topological entities such as vertices, edges or faces with a unique identifier that persists even after a model modification [5, 8, 15]. A persistent name would provide the ability to identify the relevant geometric entity even after a topology-modifying parameter change. With the persistent naming, a compute node can reference a particular topological entity as an input argument through its persistent name, even after a potentially topology-changing model update. However, there are situations though where the persistent name is not unique after a change of the shape topology, which is a typical problem in most CAD software and reduces the robustness of the parametric geometry generation. The resolution of this ambiguity depends on the user's modeling intent. While CAD systems can make intelligent assumptions about the user's intent to increase robustness, we argue that the CAD system should not make implicit assumptions when designing complex engineering systems. To address this, we plan to explicitly encode this intent into a set of user-defined criteria that can be automatically checked for all parameter variations. Another way to identify topological entities in assemblies is to use metadata, which can be attached to them. Initial support for metadata is already implemented. The metadata can be of any type, as long as it can be serialized into strings. A topological entity can have any number of metadata objects attached to it. Metadata are written to a separate metadata store and referenced by the persistent name. This architecture is similar to the OCCT architecture, but has no restrictions on the type of metadata. Metadata will be preserved by `grunk` recipes and during file exchange using typical exchange formats such as STEP or IGES, as far as the exchange file format allows.

Parametric assemblies `grunk` will also support parametric assemblies in the future. This requires the ability to identify not only whole geometric shapes, but also their subshapes, e.g. when formulating a constraint between subshapes of two parts. To be conceptually robust, this requires persistent naming and metadata.

Reverse-Mode AD As described in Sec. 5, the calculation of geometric sensitivities is enabled by using the *traceless* forward mode of ADOL-C. Nevertheless, the *trace-based* option has a greater potential because it offers the reverse mode of AD which can dramatically reduce the temporal complexity of the derivative computation compared to the forward mode of AD. Thus, the next step is to integrate the *trace-based* `adouble` class into the geometric modeling framework.

Domain-specific data models In addition to incorporating geometric modeling plugins, our software will also support domain-specific data models through plugins. These data models provide an abstract representation of engineering products, including their geometry, and allow for the parameterization of a wide range of geometries using a fixed set of parameters. In the field of aircraft pre-design, an important domain-specific data model is CPACS [2], which is a data definition for the air transport system and also includes e.g. an abstract geometry definition of aircraft. The TiGL Geometry Library [24] is an OCCT-based geometry generator for generating aircraft geometries from this abstract CPACS data model. We intend to create a grunk plugin based on TiGL, as it allows to combination of the fixed parameter set of the aircraft with custom parameters for further geometry modifications into a common feature tree. Similarly, the software GTlab [21] defines a data model for aircraft engines [20], including their geometry. It is planned to port GTlab's geometric modeling engine from OCCT to grunk to take advantage of new features such as the algorithmic differentiation of geometry generation and the ability to extend fixed engine parameters with user-defined ones.

Graphical user interface A GUI is not included in the presented software framework. However, the realization as a C++ library opens its use within a variety of other preexisting graphical interfaces. Furthermore, it is possible to extend the software framework by developing a dedicated graphical user interface. Irrespective of the use of a GUI, grunk will continue to work as a stand-alone library through the programming or scripting interface.

Integration in commercial CAD systems It is considered to use grunk in combination with both commercial and open-source CAD systems. On the one hand, it can be employed as a plugin of a CAD system. In this case, grunk's geometry might be visualized in the CAD system's GUI, enabling to explore and visualize parameter variations at runtime and at the same time benefiting from the geometric modeling functionalities of the CAD system. On the other hand, one can use other CAD systems inside of grunk. This enables the use of their geometric functionalities in a parametric way and in interaction with other grunk plugins. The feasibility of both approaches depends on the particular CAD system's API accessibility.

Jan Kleinert, http://orcid.org/0000-0002-2709-214X
Anton Reiswich, http://orcid.org/0009-0006-2118-3530
Mladen Banović, http://orcid.org/0000-0002-9056-3784
Martin Siggel, http://orcid.org/0000-0002-3952-4659

REFERENCES

[1] Agarwal, D.; Robinson, T.T.; Armstrong, C.G.; Marques, S.; Vasilopoulos, I.; Meyer, M.: Parametric design velocity computation for CAD-based design optimization using adjoint methods. Engineering with Computers, 34, 225–239, 2018. http://doi.org/10.1007/s00366-017-0534-x.

[2] Alder, M.; Moerland, E.; Jepsen, J.; Nagel, B.: Recent Advances in Establishing a Common Language for Aircraft Design with CPACS. In Aerospace Europe Conference 2020, 2020. https://elib.dlr.de/134341/.

[3] Banović, M.; Mykhaskiv, O.; Auriemma, S.; Walther, A.; Legrand, H.; Müller, J.D.: Algorithmic differentiation of the Open CASCADE Technology CAD kernel and its coupling with an adjoint CFD solver. Optimization Methods and Software, 33(4-6), 813–828, 2018. http://doi.org/10.1080/10556788.2018.1431235.

[4] Beder, J.: yaml-cpp, 2023. https://github.com/jbeder/yaml-cpp. Accessed: 2023-03-29.

[5] Bidarra, R.; Bronsvoort, W.: Semantic feature modelling. Computer-Aided Design, 32(3), 201–225, 2000. ISSN 0010-4485. http://doi.org/10.1016/S0010-4485(99)00090-1.

[6] CADQuery: CADQuery, 2023. https://github.com/CadQuery/cadquery. Accessed: 2023-01-06.

[7] Conan: Conan, 2023. https://conan.io/. Accessed: 2023-03-29.

[8] Farjana, S.H.; Han, S.: Mechanisms of Persistent Identification of Topological Entities in CAD Systems: A Review. Alexandria Engineering Journal, 57(4), 2837–2849, 2018. ISSN 1110-0168. http://doi.org/10.1016/j.aej.2018.01.007.

[9] FreeCAD: Freecad, 2023. https://www.freecadweb.org/. Accessed: 2023-01-06.

[10] Giles, M.B.; Duta, M.C.; Müller, J.D.; Pierce, N.A.: Algorithm developments for discrete adjoint methods. AIAA journal, 41(2), 198–205, 2003. http://doi.org/10.2514/2.1961.

[11] Görtz, S.; Ilic, C.; Abu-Zurayk, M.; Wunderlich, T.; Schulze, M.; Klimmek, T.; Süelözgen, Ö.; Kier, T.; Schuster, A.; Petsch, M.; Häßy, J.; Becker, R.G.; Siggel, M.; Kleinert, J.; Mischke, R.; Gottfried, S.: Collaborative high fidelity and high performance computing-based mdo strategies for transport aircraft design. In 32nd Congress of the International Council of the Aeronautical Sciences (ICAS2021), 2021. https://elib.dlr.de/144430/.

[12] Haimes, R.; Dannenhoffer, J.: The engineering sketch pad: A solid-modeling, feature-based, web-enabled system for building parametric geometry. In 21st AIAA Computational Fluid Dynamics Conference, 2023. http://doi.org/10.2514/6.2013-3073.

[13] Jameson, A.: Aerodynamic design via control theory. Journal of Scientific Computing, 3, 233–260, 1988. http://doi.org/10.1007/BF01061285.

[14] Kim, J.; Pratt, M.; Iyer, R.; Sriram, R.: Data exchange of parametric CAD models using ISO 10303-108, 2007. http://doi.org/10.6028/NIST.IR.7433.

[15] Kripac, J.: A mechanism for persistently naming topological entities in history-based parametric solid models. Computer-Aided Design, 29(2), 113–122, 1997. ISSN 0010-4485. http://doi.org/10.1016/S0010-4485(96)00040-1. Solid Modelling.

[16] Liersch, C.M.; Schütte, A.; Siggel, M.; Dornwald, J.: Design studies and multi-disciplinary assessment of agile and highly swept flying wing configurations. CEAS Aeronautical Journal, 11(3), 781–802, 2020. http://doi.org/10.1007/s13272-020-00453-y.

[17] Nagy, P.; Jones, B.; Minisci, E.; Fossati, M.; Cea, A.; Palacios, R.; Roussouly, N.; Gazaix, A.: Multi-fidelity nonlinear aeroelastic analysis of a strut-braced ultra-high aspect ratio wing configuration. In AIAA Aviation 2022 Forum, 2022. http://doi.org/10.2514/6.2022-3668.

[18] OpenCascade: OpenCASCADE Technology, 3d modeling and numerical simulation, 2023. https://www.opencascade.com. Accessed: 2023-01-06.

[19] Pironneau, O.: On optimum design in fluid mechanics. Journal of Fluid Mechanics, 64(1), 97–110, 1974. http://doi.org/10.1017/S0022112074002023.

[20] Reitenbach, S.; Hollmann, C.; Schmeink, J.; Vieweg, M.; Otten, T.; Haessy, J.; Siggel, M.: Parametric datamodel for collaborative preliminary aircraft engine design. In AIAA Scitech 2021 Forum, 1419, 2021. http://doi.org/10.2514/6.2021-1419.

[21] Reitenbach, S.; Vieweg, M.; Becker, R.; Hollmann, C.; Wolters, F.; Schmeink, J.; Otten, T.; Siggel, M.: Collaborative aircraft engine preliminary design using a virtual engine platform, part a: Architecture and methodology. In AIAA Scitech 2020 Forum, 0867, 2020. http://doi.org/10.2514/6.2020-0867.

[22] Safdar, M.; Jauhar, T.A.; Kim, Y.; Lee, H.; Noh, C.; Kim, H.; Lee, I.; Kim, I.; Kwon, S.; Han, S.: Feature-based translation of CAD models with macro-parametric approach: issues of feature mapping, persistent naming, and constraint translation. Journal of Computational Design and Engineering, 7(5), 603–614, 2020. ISSN 2288-5048. http://doi.org/10.1093/jcde/qwaa043.

[23] Salome: Salome Shaper, 2023. https://www.salome-platform.org/?page_id=327. Accessed: 2023-01-06.

[24] Siggel, M.; Kleinert, J.; Stollenwerk, T.; Maierl, R.: Tigl: An open source computational geometry library

for parametric aircraft design. Mathematics in Computer Science, 2019. http://doi.org/10.1007/s11786-019-00401-y.

[25] Yu, G.; Müller, J.D.; Jones, D.; Christakopoulos, F.: CAD-based shape optimisation using adjoint sensitivities. Computers & Fluids, 46(1), 512–516, 2011. ISSN 0045-7930. http://doi.org/10.1016/j.compfluid.2011.01.043. 10th ICFD Conference Series on Numerical Methods for Fluid Dynamics (ICFD 2010).

[26] Zakrzewski, A.S.; Lange, F.; Hollmann, R.W.: Multi-fidelity aerodynamic design process for moveables at dlr virtual product house. In AIAA, ed., AIAA Aviation and Aeronautics Forum and Exposition, AIAA AVIATION Forum 2022. American Institute of Aeronautics and Astronautics, Inc., 2022. https://elib.dlr.de/190308/.