



Title:

A generic parametric modeling engine targeted towards multidisciplinary design: goals and concepts

Authors:

Jan Kleinert, jan.kleinert@dlr.de, Institute for Software Technology, German Aerospace Center (DLR)
 Anton Reiswich, anton.reiswich@dlr.de, Institute for Software Technology, German Aerospace Center (DLR)
 Martin Siggel, martin.siggel@dlr.de, Institute of Propulsion Technology, German Aerospace Center (DLR)
 Mladen Banović, mladen.banovic@dlr.de, Institute of Software Methods for Product Virtualization, German Aerospace Center (DLR)

Keywords:

multidisciplinary design, parametric modeling, CAD engine, open source, software architecture, geometric sensitivities

DOI: 10.14733/cadconfP.2023.101-105

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 the 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.

Requirements (1) and (4) are concerned with the geometry generation and analysis, while requirements (2) and (3) address a common platform used by all engineers.

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 [1, 2, 3, 4, 5, 6, 7, 8] for a non-exhaustive list. 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.

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

- On the lowest level, it is performed by using various programming languages. This includes C++, e.g. using geometry libraries such as OpenCascade Technology (OCCT) [4] as well as interpreted scripting languages, often using bindings of the geometry functionalities provided in C++ libraries.
- In addition, domain-specific data models such as CPACS [9] 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 minimal number of dependencies. We then provide Python bindings in order to access the full functionality via a scripting language. Users can write plugins to integrate data models and—while a GUI is not planned at the moment—the framework design opens its use in various GUI based modeling applications: As a C++ library, the parametric engine can be integrated into commercial (CAD) systems via their respective APIs. Vice versa, the APIs of commercial tools can be used to write plugins for the parametric engine.

The focus of this study is to identify a potential software architecture well suited to meet the requirements (1)–(5), rather than demonstrating a ready-to-use tool. The goal is to provide an integrable modeling backbone that can be used to easily create user-friendly and interoperable tools. The feasibility and efficacy of the software design is demonstrated using an early implementation.

Software framework architecture

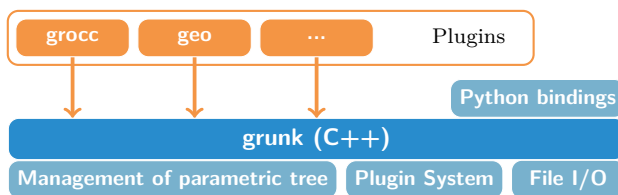


Fig. 1: The **grunk** architecture.

can be used as nodes, i.e. as initial parameters, intermediate or final calculation results. Furthermore, any function that does not modify its input parameters can be used as an algorithm. The evaluation of a parametric tree is lazy, i.e. nodes get computed only when their value is queried. Moreover, it supports an automatic invalidation of the cache in case of a change of the initial parameters. This invalidation happens in such a way that only the cache of those nodes gets cleared, that are directly affected by the changed parameters. This allows efficient re-computation of the parametric tree, which is particularly important in environments with cost intensive computations and the need for a large number of re-computations after parameter variations.

The first building block of the framework is a software called **grunk**, a 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. 1.

The first main feature is the possibility to create generic parametric trees that chain algorithms. Any data types

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 a common platform. Plugins can be managed and shared via a dedicated package manager. In this way **grunk** provides a highly modularizable and extendable workbench. See Fig. 2 for an example usage of **grunk**. 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. A geometry can be modified by changing the input parameters of the parametric tree or by extending the parametric tree with compute nodes.

Note that **grunk** is agnostic of the types passed from one plugin to another. For two plugins to work together, they must be implemented to be compatible to each other. **grunk** itself does not provide any built-in functionality to facilitate the code-coupling of two 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*, see Fig. 3.

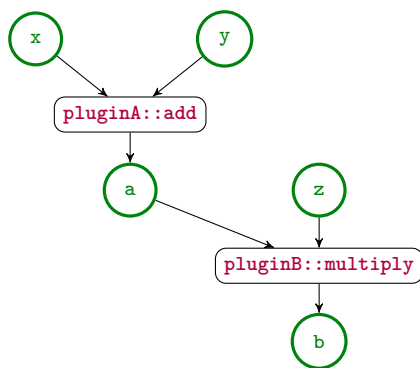
```
1 grunk::get_plugin_registry().prepend_path("C:\\plugin_dir\\");
2 grunk::get_plugin_registry().load_all(); // load pluginA
3
4 auto x = grunk::Feature("x", "pluginA::Scalar", 4.3);
5 auto y = grunk::Feature("y", "pluginA::Scalar", 3.3);
6 auto a = grunk::action("a", "pluginA::add", x, y)->output();
7 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`.

```
1 grunk.get_plugin_registry().prepend_path("~/plugin_dir/")
2 grunk.get_plugin_registry().load_all() # load pluginA, pluginB
3
4 z = grunk.Feature("z", "pluginA::Scalar", 2.0)
5 a = grunk.read("model.grr")["a"]
6 b = grunk.action("b", "pluginB::multiply", a, z).output()
7 grunk.write("model2.grr", b)
```

(b) Example Python code: Reading `model.grr` from code 2a, modifying it and writing to another **grunk** recipe `model2.grr`. See Fig. 3 for the resulting parametric tree.

Fig. 2: Example usage in C++ and in Python. **grunk** enables parametric modeling with arbitrary algorithms and data types. The algorithms and data types are implemented in plugins that are loaded at run time.



(a) Parametric tree created in code 2b.

```
uses:
  grunk: 0.1.0
  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]]
```

(b) The **grunk** recipe `model2.grr` created in code 2b.

Fig. 3: Parametric trees can be written to or read from a human-readable file format, the so-called *grunk recipe*.

Geometric modeling with `grunk`

In order to address the geometric requirement (1), we provide two additional software packages.

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.

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. 4. Both `grocc` and `geo` can be used to model geometry within the same parametric tree.

```

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]]

```

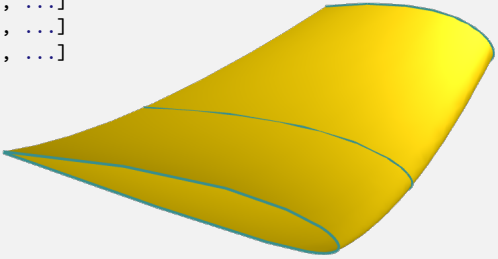


Fig. 4: A `grunk` recipe making use of the `geo` plugin. A simple wing is created using `geo` and the result is exported using OCCT/`grocc` (*initial parameters are truncated for better readability*).

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 algorithmic differentiation (AD).

In particular, the plugins are based on the differentiated version of the OCCT kernel, which was developed by integrating the AD tool ADOL-C into its source code. Due to the complexity of the OCCT sources, this process involved a significant amount of code modifications—more details can be found in [10]. Once completed, the AD-enabled OCCT provides geometric sensitivities for arbitrary

parametrizations, that are accurate up to machine precision.

Next, the differentiated OCCT code is used by the `geo` and `grocc` plugins. Every additional code that is written as a part of these plugins is also differentiated with ADOL-C such that all derivatives are correctly propagated.

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 a datatype specific to ADOL-C called `adouble` and use this datatype to define inputs (e.g. design parameters) and outputs (e.g. coordinates of surface points). After that, one can proceed to the derivative computation.

Conclusion and outlook

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 geometry modeling context. Its high modularity, generic architecture and the strict decoupling of the geometric functionalities lift it from the context of geometric modeling end enables a flexible use in a large variety of other fields, that are characterized by the requirements (2) and (3). The software is still in an early development stage. Several additional features are work in progress and e.g. include metadata, which can be pinned to topological entities, which are propagated during geometric modeling as well as unique topological entity identifiers that resemble the shape modeling history using a persistent naming scheme [11]. It is planned to release the software under an open source license, once the software evolves to a mature state. The license will allow the provision and use of both FOSS and commercial plugins.

References:

- [1] M. Siggel, J. Kleinert, T. Stollenwerk and R. Maierl: TiGL: An Open Source Computational Geometry Library for Parametric Aircraft Design, *Math.Comput.Sci.* [10.1007/s11786-019-00401-y](https://doi.org/10.1007/s11786-019-00401-y) , (2019).
- [2] OpenVSP. OpenVSP - A parametric aircraft geometry tool <https://openvsp.org/>. Accessed: 2023-01-10.
- [3] R. Haimes and J. Dannenhoffer: The Engineering Sketch Pad: A Solid-Modeling, Feature-Based, Web-Enabled System for Building Parametric Geometry. [10.2514/6.2013-3073](https://doi.org/10.2514/6.2013-3073) , (2013).
- [4] OpenCascade. Open CASCADE Technology, 3d modeling and numerical simulation <https://www.opencascade.com>. Accessed: 2023-01-06.
- [5] FreeCAD. FreeCAD <https://www.freecadweb.org/>. Accessed: 2023-01-06.
- [6] Salome Platform. Salome Shaper https://www.salome-platform.org/?page_id=327. Accessed: 2023-01-06.
- [7] OpenSCAD. OpenSCAD <https://openscad.org/>. Accessed: 2023-01-06.
- [8] CADQuery. CADQuery <https://github.com/CadQuery/cadquery>. Accessed: 2023-01-06.
- [9] M. Alder, E. Moerland, J. Jepsen and B. Nagel. Recent Advances in Establishing a Common Language for Aircraft Design with CPACS. Aerospace Europe Conference 2020, Bordeaux, France (2020).
- [10] M. Banović, O. Mykhaskiv, S. Auriemma, A. Walther, H. Legrand and J.-D. Müller. Algorithmic differentiation of the Open CASCADE Technology CAD kernel and its coupling with an adjoint CFD solver. *Optimization Methods and Software*, [10.1080/10556788.2018.1431235](https://doi.org/10.1080/10556788.2018.1431235) , (2018).
- [11] S. H. Farjana and S. Han. Mechanisms of Persistent Identification of Topological Entities in CAD Systems: A Review. *Alexandria Engineering Journal*, [10.1016/j.aej.2018.01.007](https://doi.org/10.1016/j.aej.2018.01.007) , (2018)