Deutscher Luft- und Raumfahrtkongress 2024
DocumentID: 630078

ADVANCED COLLABORATIVE AIRCRAFT DESIGN: COMPLEMENTING CPACS WITH KNOWLEDGE-BASED ENGINEERING AND CUSTOM GEOMETRY PARAMETRIZATION

J. Kleinert*, B. Boden†, A. Reiswich*

* German Aerospace Center, Institute of Software Technology, Linder Höhe, 51147 Cologne, Germany
† German Aerospace Center, Institute of System Architectures in Aeronautics, Hein-Saß-Weg 22, 21129 Hamburg, Germany

Abstract

This paper presents an advanced workflow for collaborative aircraft design, extending the Common Parametric Aircraft Configuration Schema (CPACS) with knowledge-based engineering and custom geometry parametrization. By integrating domain-specific tools through the parametric modeling framework grunk, we offer a flexible solution to enhance CPACS's capabilities. Two approaches are demonstrated: the integrated plugin approach, where all tools are available as grunk plugins, and the recipe export approach, where domain-specific tools generate parametric models, called grunk recipes. These methods enable the seamless combination of standardized data models with custom parametric extensions, improving collaboration and automation in multidisciplinary design processes. We exemplify the presented process by applying geometric modifications from both the CAD kernel OpenCascade Technologies and the knowledge–based engineering framework Codex to an existing CPACS model.

Keywords

Geometry; CPACS; Collaborative Design, Parametrization

1. INTRODUCTION

1.1. Motivation

The Common Parametric Aircraft Configuration Schema (CPACS) [1] as a central data model is an indispensable enabler for multidisciplinary aircraft design. As a hierarchical data model, it provides a finite range of standardized descriptions. Additionally, different experts maintain domain–specific knowledge that is not yet properly representable within CPACS. While the standard is actively maintained and extended, the time required for the definition of new data types as well as the implementation in CPACS-based tools such as TiGL [2] often exceeds the constraints of design projects.

In collaborative aircraft design projects, different departments often use custom tools to extract or modify geometry from CPACS files according to their specific needs. This leads to the proliferation of scripts and code snippets that are difficult for other project partners or automated MDAO processes to reproduce. These tools, scripts, and code snippets are typically tailored to run in local environments set up by individuals, making them inaccessible or incompatible with broader workflows. Furthermore, it is often unclear what the exact inputs and outputs are, or whether these tools and scripts are parametrically linked to the data model, which complicates their integration into a cohesive and automated design process.

When the focus of collaboration is a simulation or optimization workflow, tools like RCE [3] help address these issues by providing a structured framework for process integration and automation. However, when the collaborative effort centers around a parametric (geometry) model that needs to be tailored to the requirements of all project partners, the question of how to effectively manage and synchronize these custom CPACS extensions remains an open challenge.

1.2. Objective and Methodological Overview

In this study, we present a workflow that extends a CPACS model of an aircraft by parametrically incorporating domain–specific models. We do this systematically using the parametric modeling engine grunk [4].

We use CPACS and TiGL to describe the overall aircraft design, while employing the knowledge-based engineering (KBE) tool Codex [5, 6] to parametrically model the geometry of a fuel system, a component which is absent in the CPACS standard. We use OpenCascade Technology (OCCT) [7] for geometry modeling.

We demonstrate two different ways of using the parametric modeling framework grunk for this task,

CC BY-SA 4.0

doi: 10.25967/630078

namely the *integrated plugin approach* and the *recipe export approach*.

In the *integrated plugin approach* we use the functions and types of domain specific tools such as TiGL and OCCT within a single parametric model, a so–called grunk recipe. For this approach, all involved tools must be available as a grunk plugin, each providing building blocks for a certain domain or level of abstraction. The TiGL plugin provides methods and types from the domain of aircraft predesign, with types such as wings, fuselages, rips and spars. An OCCT plugin provides methodology to manipulate geometries based on a boundary representation with types such as points, curves and B-spline surfaces.

Vice versa, the *recipe export approach* is an integration of grunk via its API into a domain specific tool to be able to export grunk recipes for further manipulation. In this scenario, the domain specific tool does not need to be available as a grunk plugin. The domain specific tool is needed to generate the parametric model, but it is not needed to evaluate the model for a different parameter set or to extend the parametric model with further customizations.

The integrated plugin approach is ideal when all necessary tools can be made available as grunk plugins, which can be reused and shared across different workflows, but it requires the provision, installation and maintenance of multiple plugins. In contrast, the recipe export approach is more flexible, as it doesn't require the tools to be available as plugins, but it depends on external tools for model generation and may limit the ability to modify or extend the model parametrically afterwards. Users should opt for the plugin approach when tight integration and reusability are needed, and the export approach when tool availability or simpler workflows are prioritized.

We demonstrate the integrated plugin approach by creating a grunk recipe that interprets a CPACS file using a plugin as an interface to TiGL, extracts geometric information using TiGL and extents it using grocc, a grunk plugin for OCCT. We demonstrate the recipe export approach by exporting a fuel system, that has been created in Codex as a grunk recipe. We note, that it is not our objective to provide a realistic model of a fuel system, but rather to provide and demonstrate the methodology and design workflow in principle. Finally, both grunk recipes are merged into a single parametric model that leverages CPACS where possible and incorporates a custom parametrization where necessary.

2. TOOLS

In this section we describe the two main tools used in the workflow, the parametric modeling engine grunk and the knowledge–based engineering framework Codex.

2.1. grunk

grunk is a parametric modeling engine written in C++ that is developed at the German Aerospace Center (DLR). The framework is distinct from traditional CAD tools by decoupling geometry generation from the parametric engine, offering flexibility for integration with different modeling and simulation tools. The key feature of grunk is its plugin system, allowing custom types and functions to be added without altering the core engine. Plugins can provide building blocks for parametric models for any kind of domain or level of abstraction.

One of the primary goals of grunk is to facilitate consistent geometry generation across different departments working on a design, ensuring reproducibility and efficient parameter modifications. Built-in lazy evaluation and caching allows efficient re-evaluation of parametric models after parameter changes. Grunk recipes can be exported to a human-readable exchange format based on YAML.

Internally, grunk represents a parametric model as a directed acyclic graph of inputs, computations and their outputs. There are two kinds of nodes in the tree, namely so–called *features* and *steps*. A *feature* represents an independent parameter or an intermediate or final calculation result. A *step* represents a computation. Steps can have any number of input or output features. A feature can have at most one input step, but can be used in any number of steps as an input.

At the time of writing, there are five distinct kinds of steps that are supported in a grunk recipe, namely *expressions*, *actions*, *scripts*, *recipe evaluations* and *vectors*.

An *expression* corresponds to the evaluation of a term or formula.

An *action* simply represents the evaluation of a function, usually provided by a plugin. To ensure a consistent data flow from inputs to outputs, functions used in actions may not alter their inputs. This requirement poses a hard restriction when using legacy object oriented code, where many classes have member functions mutating the class instance, such as setters.

Scripts are a step kind specifically to work around this limitation: A script consists of a sequence of actions within a single step of a parametric tree. Variables created within a script may be altered by other calculations within the same script. When a script exits, its outputs are returned as immutable objects.

Grunk recipes can contain any number of grunk recipes and can therefore be nested indefinitely. On the one hand, this allows structuring complex models in hierarchies. On the other hand, this allows calling a subrecipe just like a function, mapping feature nodes from the outer recipe to inner nodes and then mapping the inner result nodes to new result features in the containing recipe. This kind of recipe step is called a *recipe evaluation*.

The fifth and final step kind is a *vector*. It represents the collection of any number of feature nodes in a con-

CC BY-SA 4.0

tainer. This container can be used as input to actions that expect vectors or lists of values.
The engine grunk itself makes no assumptions on the data types and functions that are used as actions in a recipe. All modeling functionality is implemented in plugins that can be loaded at runtime.
Geometric functionalities are provided through plugins like grocc, which integrates OCCT, and geo, which extends it further, making the platform highly modular and extendable. geo is the plugin for the geometry library geoml, also developed at DLR.

2.2. Codex

The COllaborative DEsign and eXploration (Codex) platform [5, 6] is a platform for the development of knowledge-based engineering (KBE) applications based on Semantic Web Technologies (SWT), which is currently being developed at the German Aerospace Center. It allows the creation of domain-specific knowledge-bases and enables integrating these into a single model of the overall product. Thereby, it aims to support the digitalization of the aircraft design process and to allow designers to focus on the creative part of their design tasks.
The codex-geometry module [8], which is part of the Codex framework, allows the user to model complex geometries as well as geometric requirements, which can be automatically evaluated by Codex. It provides an ontology for the description of geometric shapes, including primitive shapes like spheres, cylinders, curves etc. as well as operations like union or intersection. In combination, these can be used to describe even complex geometries. codex-geometry also allows the definition of geometric requirements and can automatically check if these requirements are met by a given geometry. These automatic consistency checks can help to detect design errors early in the aircraft design process.

3. DESIGN WORKFLOW

In this section we demonstrate the use of both the integrated plugin approach and the recipe export approach for extending an existing parametric model with grunk. We shortly outline how two parametric models can be integrated into a single recipe.

3.1. Integrated Plugin Approach

As a prerequisite for the integrated plugin approach, we first needed to create a grunk Plugin for TiGL. The command line interface of grunk offers a semi-automated code generation process for the creation of grunk plugins from existing C++ libraries. This functionality has already been used to generate the grunk plugin for OCCT as well as for the geometry modeling library geoml. A plugin author must setup a project directory from a template with a YAML input file for the code generator. This input file specifies the name, version and some metadata of the plugin, as well as the header files of the C++ library that shall

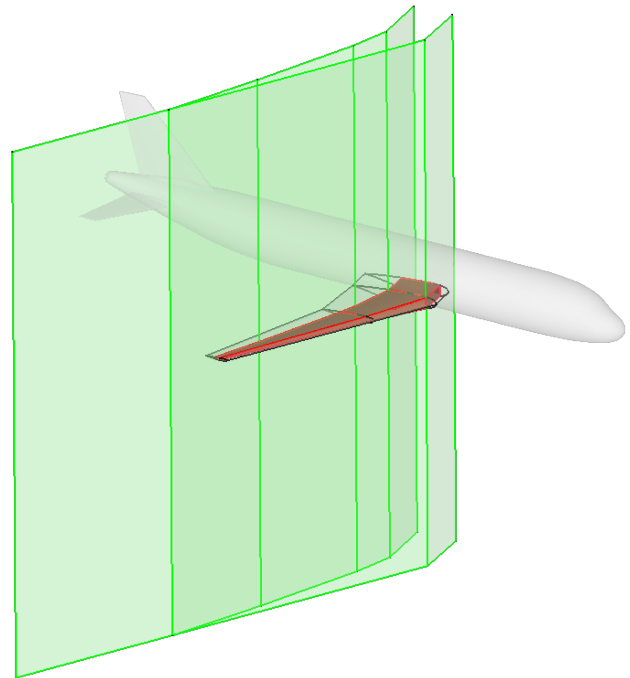


FIG 1. Generation of the wing box geometry for the D150, similar in size to the A320 [9] research configuration. The cut geometries of the leading edge and trailing edge spars are used to define the design space of the fuel system.

be parsed for type and function declarations. It is possible to specify blacklists and wildcards. From this information, the C++ code for the grunk plugin is generated, registering all class declarations together with their base class declarations, data members and member functions, constructors and conversions as well as free functions in grunk's dynamic type system, as specified in the YAML input file. A plugin author can customize type and function registrations, e.g. for the serialization and deserialization of class instances to the YAML representation of a grunk recipe or to add additional functionality that is unavailable in the underlying C++ library.
Plugins created in this way can be made availabe both in source code and in binary form to all project partners involved in an aircraft design project. grunk has a package manager to facilitate the sharing and installation of plugins from local or remote registries. This ensures that all participating parties work with the same tools and everybody can easily interpret and manipulate the same underlying parametric model.
With the TiGL plugin available, the next step is to extract certain geometric information from an existing CPACS file within a grunk recipe. TiGL already provides a wide range of functionality for geometry extraction. For some use cases this functionality might not suffice. For instance, TiGL currently has no functions to generate a solid wing box geometry, that is the space confined in between two spars, two ribs as well as the lower and upper skin of the wing. This wing box is the confining space for fuel lines and pumps in

CC BY-SA 4.0

the wing and is needed to check the validity of the fuel system geometry.
To still obtain a valid wing box from a CPACS file, we can use TiGL to get the so-called *cut geometries* of ribs and spars, which are the rib and spar geometries before trimming with the outer wing shape. These cut geometries can be used to cut out a solid section from the wing to create the wing box using Boolean operations, see Figure 1.
The grunk recipe responsible for opening a CPACS file, extracting geometric information and creating the wing box is generated using the python bindings of grunk. The recipe is exported to YAML, see Figure 2.

3.2. Recipe Export Approach

In [8] and [10], Codex and codex-geometry were used to create a geometric model of a conventional aircraft fuel system. Certain parameters of the fuel system, like the wall thickness of the fuel pipes can be automatically determined by Codex based on requirements, such as the required fuel mass or pressure. Furthermore, a variety of geometric requirements were created for these fuel system models. The aircraft fuel system has a large volume, a complex geometry and is highly integrated with other components of the aircraft, which makes it an important use case for this work.
As the fuel system can currently not be described using the CPACS standard, in this work we aim at extending the CPACS model using a fuel system model created in Codex. We create the fuel system for an aircraft based on the domain-specific model introduced in [8, 10]. The resulting geometry is exported as a grunk recipe and thus can be easily combined with the wing box extracted directly from CPACS.

3.3. Unification of recipes

grunk recipes can be nested indefinitely. Therefore the grunk API can be used to easily create a signle recipe that contains the recipe for the wing box and aircraft shape as one subrecipe and the fuel system recipe as a second. These subrecipes can be invoked in the outer recipe like functions. This way, the inputs and outputs of subrecipes could be connected to obtain a consistent fully parametric model, see Fig. 3.

4. RESULTS

As design space to test our approach, we use the outer aircraft geometry from the Avacon project [11], similar to a B767. The resulting geometry is shown in Figure 4. For the integrated plugin approach a grunk recipe was prepared using the python bindings of grunk, see Figure 2. The fuel system was generated on the basis of the same aircraft geometry in Codex and a grunk recipe was created. Using the grunk API from python, the input parameters for the wing box recipe, such as the indices of the rips and spars as well as the CPACS file name are modified and the two recipes are merged into a single recipe. Currently, parameteric changes within CPACS will not affect the fuel system. The fuel system recipe currently has 165 Carthesian points as independent input parameters. To fully couple the CPACS configuration to the fuel system model, these parameters must be set based on the CPACS geometry. This coupling will be performed in a future task.

5. CONCLUSION AND OUTLOOK

By incorporating domain-specific models from Codex and leveraging the parametric modeling engine grunk, we offer a practical approach to extend CPACS functionality while maintaining its benefits.
In the future, we plan to connect the inputs and outputs of the two submodels for a fully automatic design, respecting the requirements that were used in the geometry definition in Codex.
One of TiGL's primary goals is to eliminate ambiguity in geometry interpretation left by CPACS. When all project partners use TiGL for geometry generation, it ensures that everyone operates with the same geometric model. However, this advantage is lost when CPACS lacks certain geometric features or when TiGL has yet to implement them. Extending CPACS or TiGL can be time-consuming and may not fit within the tight deadlines of a design project. By enabling the extension of CPACS parametrizations with custom geometries and domain-specific models in a consistent manner, design projects are no longer constrained by the absence of standard functionality. Custom parametric models can be exchanged seamlessly across the consortium, and if they remain consistently parametric, they could serve as the foundation for future CPACS definitions and TiGL implementations, such as for fuel systems, thereby accelerating future aircraft design efforts.
Additionally, we want to enable grunk recipes in which low level CPACS parameters such as transformations and positionings are calculated from arbitrary custom high level parameters, such as wing aspect ratio or wing span. A graphical user interface for grunk is under development to facilitate parametric modeling for the end user.
By addressing these challenges, we aim to create a fully integrated, parametric design workflow that not only extends CPACS but also ensures consistency across diverse domain-specific requirements, setting the foundation for future advancements in collaborative and automated aircraft design.

Contact address:

jan.kleinert@dlr.de

References

[1] Marko Alder, Erwin Moerland, Jonas Jepsen, and Björn Nagel. Recent advances in establishing a common language for aircraft design with cpacs. In *Aerospace Europe Conference 2020*, 2020.

CC BY-SA 4.0

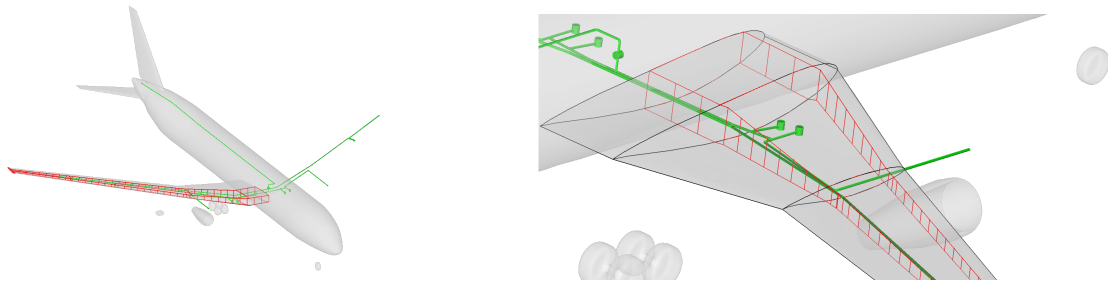
```yaml
uses:
  grunk: 0.3.2
  grocc: 0.1.4
  tigl: 0.1.0
parameters:
  cpacs_file: !<String> CPACS_30_D150.xml
  cpacs_config: !<String> ""
  wing_idx: !<int> 1
  cs_idx: !<int> 1
  spar_le_idx: !<int> 1
  spar_te_idx: !<int> 2
  ribs_def_root_idx: !<int> 1
  rib_root_idx: !<int> 1
  ribs_def_tip_idx: !<int> 7
  rib_tip_idx: !<int> 1
steps:
  - !<tigl::ConfigurationWrapper> [[tigl], [cpacs_file, cpacs_config]]
  - !<script>
    steps:
      - !<tigl::ConfigurationWrapper::get_configuration> [[config], [tigl]]
      - !<tigl::CCPACSConfiguration::AircraftFusingAlgo> [[fuser], [config]]
      - !<tigl::CTiglFusePlane::FusedPlane> [[fused_plane], [fuser]]
      - !<tigl::CNamedShape::Shape> [[fused_shape], [fused_plane]]
    returns:
      - fused_shape
  - !<script>
    steps:
      - !<tigl::ConfigurationWrapper::get_configuration> [[config], [tigl]]
      - !<tigl::CCPACSConfiguration::GetWing> [[wing], [config, wing_idx]]
    returns:
      - wing
  - !<script>
    steps:
      - !<tigl::CCPACSWing::GetComponentSegment> [[cs], [wing, cs_idx]]
      - !<tigl::generated::CPACSComponentSegment::GetStructure> [[structure_opt], [cs]]
      - !<tigl::CCPACSWingCSStructure_optional::value> [[structure], [structure_opt]]
    returns:
      - structure
  - !<tigl::CCPACSWing::GetWingCleanShape> [[wing_loft], [wing]]
  - !<tigl::CNamedShape::Shape> [[wing_shape], [wing_loft]]
  - !<tigl::CCPACSWingCSStructure::GetSparSegment> [[spar_le], [structure, spar_le_idx]]
  - !<tigl::CCPACSWingSparSegment::GetSparCutGeometry> [[spar_le_shape], [spar_le]]
  - !<grocc::gp_Vec> [[spar_le_extrusion_vec], [!<double> -100., !<double> 0., !<double> 0.]]
  - !<grocc::BRepPrimAPI_MakePrism> [[wing_le_tool], [spar_le_shape, spar_le_extrusion_vec]]
  - !<grocc::BRepAlgoAPI_Cut> [[split1], [wing_shape, wing_le_tool]]
  # truncated here for better readibility
```

FIG 2. An excerpt of the grunk recipe for the wing box. The recipe has been truncated after the first Boolean operation for better readibility.

```yaml
uses:
  grunk: 0.3.2
  grocc: 0.1.4
  tigl: 0.1.0
parameters:
  point1: !<grocc::gp_Pnt> [20, -4, -1.6]
  ribs_def_tip_idx: !<int> 35
steps:
  - !<recipes::wingbox> [{wingbox: split4}, {ribs_def_tip_idx: ribs_def_tip_idx}]
  - !<recipes::fuelsystem> [{fuelsystem: compound}, {n91dbdcd0fc5c4d68bdfaf9fdaeb17aecb180: point1}]
recipes:
  fuelsystem:
    # truncated
  wingbox:
    # truncated
```

FIG 3. An examplatory excerpt of the unified grunk recipe. It contains the fuel system and the wing box as subrecipes, which can be invoked like functions to overwrite parameters and extract results. Note that the subrecipes `fuelsystem` and `wingbox` have the same structure as any other grunk recipe. They have been truncated here for better readibility.

CC BY-SA 4.0



(a) The wing box and fuel system for the aircraft geometry from the Avacon project **(b) A closeup of the wing box and fuel system near the engine pylon**

FIG 4. The extended geometry model including the wing box and fuel system.

[2] Martin Siggel, Jan Kleinert, Tobias Stollenwerk, and Reinhold Maierl. Tigl: An open source computational geometry library for parametric aircraft design. *Mathematics in Computer Science*, 2019.

[3] Brigitte Boden, Jan Flink, Niklas Först, Robert Mischke, Kathrin Schaffert, Alexander Weinert, Annika Wohlan, and Andreas Schreiber. Rce: An integration environment for engineering and science. *SoftwareX*, 15:100759, July 2021.

[4] Jan Kleinert, Anton Reiswich, Mladen Banovic, and Martin Siggel. A generic parametric modeling engine targeted towards multidisciplinary design: Goals and concepts. *Computer-Aided Design and Applications*, 21(3):424–443, September 2023.

[5] J. Zamboni, A. Zamfir, and E. Moerland. Semantic knowledge-based-engineering: The codex framework. In *Proceedings of the 12th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, pages 242–249. SCITEPRESS - Science and Technology Publications, 11/2/2020 - 11/4/2020. ISBN: 978-989-758-474-9. DOI: 10.5220/0010143202420249.

[6] Jonas Jepsen, Arthur Zamfir, Brigitte Boden, Jacopo Zamboni, and Erwin Moerland. On the development of a collaborative knowledge platform for engineering sciences. In *15th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, pages 208–215, 2023.

[7] OpenCascade. OpenCASCADE Technology, 3d modeling and numerical simulation, 2024. https://www.opencascade.com. Accessed: 2024-03-19.

[8] Brigitte Boden, Yannic Cabac, Tim Burschyk, and Björn Nagel. Rule-based verification of a geometric design using the codex framework. In *33rd Congress of the International Council of the Aeronautical Sciences, ICAS 2022*, ICAS Proceedings, November 2022.

[9] Wolf R. Krüger, Berit Gerlinger, Olaf Brodersen, Thomas Klimmek, and Yves Günther. Das dlr-projekt kontekst: Konzepte und technologien für emissionsarme kurzstreckenflugzeuge. In *DLRK 2020 - Deutscher Luft- und Raumfahrtkongress*, 2020.

[10] Brigitte Boden, Matheus Padilha, Tim Burschyk, Carlos Cabaleiro de la Hoz, Erwin Moerland, and Marco Fioriti. Automating the verification of geometric requirements for aircraft fuel systems using knowledge-based engineering. In *34rd Congress of the International Council of the Aeronautical Sciences, ICAS 2024*, ICAS Proceedings, 2024.

[11] Sebastian Woehler and Johannes Hartmann. Preliminary aircraft design for a midrange reference aircraft featuring advanced technologies as part of the avacon project for an entry into service in 2028. In *ONERA-DLR Aerospace Symposium*, Mai 2018.

CC BY-SA 4.0