POLITECNICO DI TORINO

Department of Mechanical and Aerospace Engineering
Master's Degree in Aerospace Engineering

Master Thesis

Enabling fuel system component placement on the basis of geometric and certification requirements within a semantic knowledge-based engineering framework

Academic supervisor:
Prof. Dr. Marco **FIORITI**

Candidate:
Matheus Henrique **PADILHA**

DLR supervisors:
Dr. Brigitte **BODEN**
Tim **BURSCHYK**
Carlos **CABALEIRO DE LA HOZ**

Academic Year 2022/2023

Success is not final, failure is not fatal:
it is the courage to continue that counts.
— Winston Churchill

Aos meus pais,
que me ensinaram a sempre seguir os meus sonhos,
cujos ensinamentos e orientação têm sido
o alicerce da minha jornada acadêmica,
e graças a eles sou a pessoa que sou.

À minha família,
pelo apoio contínuo e incentivo
ao longo dos meus estudos.

A Stefan,
che è stato sempre al mio fianco
e che ha creduto nel mio potenziale.
Grazie per avermi aiutato a raggiungere
i miei obiettivi e per avermi sostenuto
in ogni momento.

To my friends,
for the support and friendship
during my time in Germany.

Abstract

The aerospace industry is in a constant state of evolution, encountering fresh challenges amid technological advancements and shifting economic landscapes. To maintain a competitive edge, aerospace companies must continually foster innovation and adaptability. In response to a more competitive market and heightened customer expectations, the aerospace sector is directing its investments toward pioneering technologies capable of curbing project development costs. The early phases of aircraft development hold paramount significance, exerting great influence on overall project outcomes, and thus stand to gain immensely from novel processes and design methodologies. The accurate and comprehensive definition of aircraft geometry constitutes a critical requirement for a multitude of analyses, profoundly impacting the final aircraft design. Unfortunately, these detailed descriptions are often elusive during the initial project stages. Among the array of methodologies available, Knowledge-Based Engineering (KBE) emerges as an interesting approach for streamlining product development timelines and cost structures by capitalizing on pre-existing knowledge drawn from analogous projects. Harnessing DLR's innovative KBE framework, known as Codex, a set of geometric verification rules have been crafted to bolster early-stage fuel system design. Codex, facilitated by semantic-web technologies, empowers the development of domain-specific languages and tools and seamlessly integrates them into the framework. In conjunction with a suite of geometry creation rules aimed at enriching the geometric information accessible to designers during the initial phases, the GeoVerification tool equips engineers with valuable data concerning the correctness of the fuel system's geometry. In addition, a series of airworthiness requirements and guidelines have been implemented, demonstrating the potential for automating the usage of this knowledge. An important feature, seamlessly integrated into the tool, resides in its capacity to evaluate diverse scenarios of uncontained engine burst failures (UERFs) and provide remedial design actions. Efforts have culminated in the development and integration of 25 verification rules within the GeoVerification Tool. To illustrate the tool's capabilities, a fuel system architecture for a twin-engine small-medium range aircraft was designed. Leveraging existing geometric data pertaining to the aircraft's structure, detailed geometric characterizations of the fuel tanks were achieved. Moreover, spar-mounted and bottom-mounted fuel boost pumps, supply lines, fuel valves, and a simplified crossfeed subsystem were designed. Over 500 requirements' checks were executed in under four minutes using a standard personal computer for the fuel system architecture. By embracing a parametric design approach and leveraging the developed geometry creation rules, including additional rules aimed at supporting component placement automation, geometric incompatibilities were minimized. Furthermore, the tool's results, stored within the knowledge graph, facilitate the efficient management of design anomalies and the provision of corrective actions.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background and motivation

The aerospace industry is constantly evolving and facing new challenges as technology advances and new economic trends emerge. To stay ahead of the competition, companies in this industry must constantly innovate and adapt. Furthermore, globalization has created new opportunities for collaboration and expansion in the aerospace sector, necessitating the development of new product development methodologies capable of keeping up with this fast-paced market.
Early-stage aircraft design is an essential phase during the entire development process of an aircraft and has a major influence on the project's success. In the last few decades, computer-aided aircraft design has been increasingly applied, as it allows engineers to detect promising design patterns in the early stages of the design process. More recently, the concept of digital modeling for complex products has become not only a trend but a necessity in the industry as well, since the next generation of aircraft requires a system of systems approach with an ever-increasing amount of data exchanged between groups both within and across industrial partners. This need translates into the development of new methodologies to manage the complex data flow between domains. An example of this is the evolution of Knowledge-Based Engineering (KBE) methodologies, which allow knowledge from different domains to be captured and formalized for execution by computer systems.
The description of the aircraft geometry is a critical component of the challenges this phase of the design presents to the overall project. Precise and comprehensive descriptions of the aircraft geometry are essential for multiple analyses, including aerodynamic simulations and structural analysis, that impact the final aircraft design decisions. However, these detailed descriptions are frequently unavailable in the initial stages of the project. Furthermore, the handling of complex geometries, which is an inherent feature of aircraft fuel systems, is frequently not supported or poorly supported by conventional tools on the market.
Knowledge-based engineering (KBE) approaches can be used to share and integrate knowledge in a structured and efficient way when working with multidisciplinary designs. KBE makes it possible to establish a central knowledge base that all parties involved in the design process can access, facilitating cooperation and reducing mistakes brought on by misunderstandings. Furthermore, KBE tools can offer intelligent recommendations and automate monotonous tasks, increasing productivity and efficiency when designing intricate geometries for aircraft fuel systems. Today's market offers a variety of software that caters to certain knowledge-based engineering methodologies. These software solutions span a variety of categories, from more general solutions designed to ensure a high degree of interoperability to specialized tools designed to tackle particular issues. These instruments are all similar in that they take an object-oriented approach. Even though object-oriented programming (OOP) has many advantages, it requires strict model definitions. As a result, it frequently takes a while for all interested parties to agree on common terminology and schema definitions. Moreover, OOP makes it more difficult to create novel products because those products might not have the characteristics that the predefined object-oriented model requires. This might make it harder to be creative and flexible when coming up with new ideas. Furthermore, because OOP necessitates a thorough comprehension of the underlying model and all of its nuances, its complexity may result in longer development times and higher development costs.

1.2 Related work

At the German Aerospace Center's Institute for Systems Architectures in Aeronautics in Hamburg, a new framework (Codex: COllaborative DEsign and eXploration) is being developed, inspired by the potential of KBE in the aerospace sector. A first attempt at demonstrating the potential of Codex is provided in [20]. By looking at the codex-geometry module, the authors show how different geometric entities (points, curves, polylines, spheres, cuboids, etc.) can be used to define complex geometries based on the Constructive Solid Geometry principle. In order to enable users with little to no experience in software development to perform geometric operations like creating geometries and calculating volume and surface, the authors show how to integrate an open-source geometric kernel, such as the OpenCascade (OCC) CAD kernel, into a domain-specific language (DSL) service. Furthermore, the authors stress that geometric ontologies can be used to integrate geometric knowledge from different domains, even when the domain-specific models have no connection to one another. In the same paper, the authors demonstrate the possibility of performing geometric checks on the model using declarative rules enabled by the codex-rule module.
A second example of Codex possibilities can be found in [14], where the authors highlight Codex's potential for the rapid generation of new aircraft configurations by leveraging the codex-geometry module. In this article, the authors explain the advantages of a declarative rule-based approach, where rules can be exploited to generate complex geometries, expand the knowledge present in a model by creating new knowledge or connecting the knowledge present in different domains, and verify the knowledge itself. Furthermore, the authors demonstrate the advantages of a schema-less graph-based framework as well as its potential for improved knowledge reusability and interoperability. Aircraft configurations can be easily adapted and integrated with other systems or models using a schema-less framework, allowing for seamless collaboration and information exchange across domains. Codex's potential is ultimately demonstrated by the ability to generate multiple aircraft configurations using the same knowledge by simply altering a few user inputs that cause the framework's rules to operate differently.
The aforementioned works are cited in this master's thesis and provide a basis for demonstrating and augmenting Codex's capabilities. In fact, these examples show the value of the geometric definition of the model and how it can reduce development times without sacrificing the level of detail at the beginning of the design process.

1.3 Problem statement and objectives

Because it requires a lot of volume and is distinguished by complex geometries, the fuel system is one of the subsystems that could profit from additional consideration during the conceptual design in order to provide a well-rounded design solution. In particular, the fuel system design is heavily impacted by its geometric definition, which often poses difficulties to its design at early stages [20], requiring new methods to provide engineers with more detailed knowledge during these phases.
In order to better support the design of a fuel system architecture, it is important to expand the available information for this system at an early stage, which can be done by reusing existing knowledge based on previous projects and engineers' experience. The fuel system's geometric knowledge not only impacts the overall performance and efficiency of the system but also plays a crucial role in ensuring the safety and reliability of the system itself. Additionally, accurate geometric knowledge allows for effective integration with other subsystems and components, minimizing potential conflicts or design flaws. Therefore, it is essential to validate the developed model as soon as possible to ensure that the geometric knowledge is accurate and meets the desired objectives of the fuel system architecture design.
In this context, this work aims to define a methodology to support aircraft fuel system design while taking geometric, functional, and certification aspects into account. Applying knowledge-based engineering (KBE) methodologies to this task allows to make the most of already-existing knowledge and support design decisions. KBE can be used to compile past experiences, lessons learned, design guidelines, and requirements in order to give aircraft designers tools for evaluating their design decisions and making additional suggestions. Particularly for geometric definition, KBE can enable designers to concentrate on problem-solving tasks in a controlled setting, where they could be assisted by geometric rules checking for the accuracy of their designs with ease.
The use case guiding the work covers studying the placement of different system components and their validation using a declarative rule-based approach. Furthermore, the effect of a main rotor burst

and its potential to affect critical systems are also studied to demonstrate the potential to integrate multiple domains into the verification rules. On the basis of this, a methodology for effective fuel system placement is developed through the digitalization of complex geometry placement rules within a semantic web framework. Through its standardized input and output connections, the resulting KBE tool can then be used as part of overall aircraft conceptual design workflows.

Objectives

- Identify the critical components related to the fuel system and model them using complex parametric geometries with the appropriate level of detail for preliminary design within a Semantic Web framework.
- Develop methods to support the placement of components related to the fuel system in order to meet geometrical, functional, and certification constraints.
- Develop methods to assist the fuel system design, with a focus on certification requirements and the potential effects on critical fuel system components.
- Implement the methods within the Codex framework domain.
- Verify and validate the methodology by means of a use case focusing on conventional fuel system architectures.

Chapter 2

Aircraft fuel systems

Aircraft fuel systems are responsible for safely and efficiently delivering fuel to the engines, allowing airplanes to generate the necessary thrust to fly. These systems are designed to store and distribute fuel throughout the aircraft, ensuring a constant and reliable supply during flight. It is an essential and critical system in any airplane, requiring extensive knowledge and experience to optimally design it and ensure its proper functioning.
The fuel system must be carefully engineered to account for factors such as fuel weight, balance, and distribution, as well as the specific requirements of different aircraft models. Additionally, stringent safety regulations and protocols must be followed to minimize the risk of fuel leaks or fires, making the design and maintenance of aircraft fuel systems a top priority in aviation engineering. This chapter provides an overview of modern commercial aircraft's fuel system. An outlook of its main functions is provided in Section 2.1, while an overview of the main components necessary to perform these functions is provided in Section 2.2. Lastly, section 2.3 provides a brief outlook of the design process for the fuel system.

2.1 Fuel system functions

The main function of the fuel system is to provide a continuous fuel supply to the engines in every flight condition. This function can be further divided in order to have a better understanding of how each component contribute towards this goal. Based on ATA Chapter 28, the following subdivision is proposed:

- Fuel storage
- Fuel provision to the engines and APU
- Fuel transfer between tanks and fuel jettison
- Refueling and defueling
- Fuel measurement and management
- Venting and pressure management

These functions are present in all aircraft and, depending on the application, different architectures can meet the requirements. The fuel system operations for today's commercial aircraft, ranging from smaller business jets to larger transportation aircraft, are covered in this section.

2.1.1 Fuel storage

Due to the large amount of fuel in an aircraft, fuel storage may be seen as a design constraint one must overcome at the outset of the project rather than a function the system must perform. The fuel tanks, which can take many different forms, are responsible for storing fuel, and their positioning is of crucial matter to ensure proper aircraft stability.
Their installation can take place anywhere in the aircraft where integrity can be guaranteed after a crash, with typical locations being the wings, the center wing-box and the tail. Tail fuel tanks are particularly important in aircraft where the center of gravity is expected to shift a lot during the

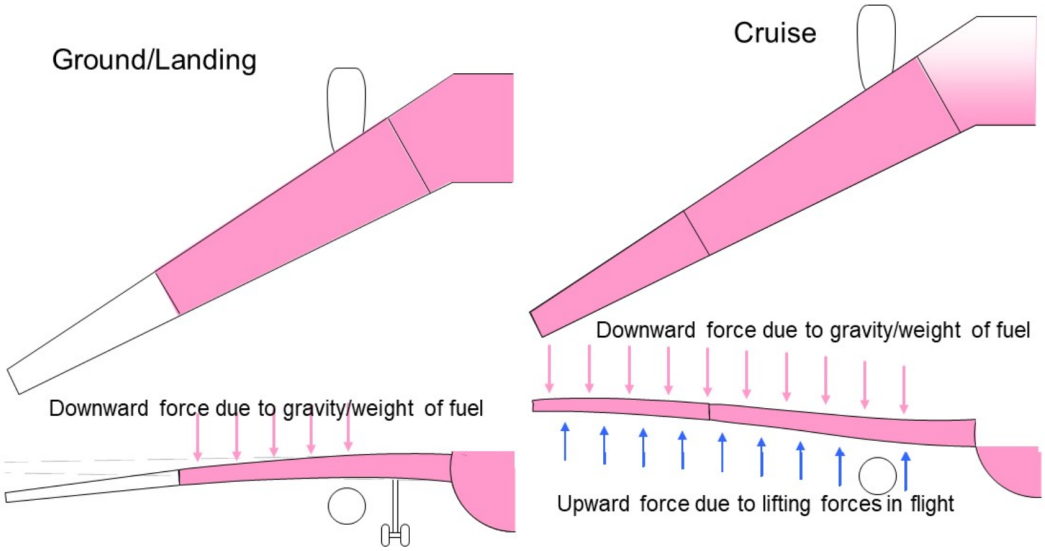

Figure 2.1: The effect of load alleviation due to the presence of wing tanks [1]

flight, and are used as means of reducing this shift to acceptable values by performing fuel transfer. In fact, when designing tail tanks, the aircraft designer must pose particular attention in studying different filling levels and transfer strategies, to make sure the aircraft's stability will not be compromised in any case, even when the system is faulty.
Fuel storage on the wings serve a secondary purpose in modern commercial aircraft, which have their engines below the wing, known as load alleviation. Large amounts of fuel can be stored in the wings to counterbalance the high loads of lift that cause the wings to bend upward during flight. This results in smaller fatigue cycles and a longer interval between maintenance activities.
In order to minimize wingtip bending and maximize this effect, the wing tanks should allow the fuel to remain as far outboard of the wings as possible until the end of the cruise phase. This strategy enables structure weight to be reduced. Figure 2.1 highlights the advantageous effect of maintaining the fuel at the most outward position of the wing and shows the effect of load alleviation due to the presence of wing tanks.
If additional tanks are needed, these are often placed at the lowest part of the fuselage, and are known as center tanks or auxiliary center tanks (ACTs). If these tanks are not used to directly feed the engine, transfer strategies must be put in place in order to guarantee appropriate engine feed through the feed tanks. The Airbus A321-XLR is a commercial aircraft that successfully integrates the auxiliary center tank strategy. In this configuration, Airbus gives its aircraft the ability to travel farther thanks to the addition of rear and forward center fuel tanks in the fuselage. This increases the aircraft's operational range by over 30% compared to the A321-neo configuration, giving it a maximum fuel capacity of 32940 liters and a non-stop operational range of 4700 nm [22].
Wing tanks are further divided into several compartments by using the already present structural elements in the wing, such as the wing ribs, in order to reduce fuel slosh in non-inertial flight conditions, which could potentially cause the aircraft to lose its stability. In order to allow for the movement of fuel inside the tank and the routing of pipes throughout the system, openings must be provided in these ribs. In ensuring a proper venting system, the ribs should also account for the passage of air for various aircraft attitudes.
When the design foresees communicating fuel storage compartments using gravity-assisted fuel transferring between tanks, it is necessary to ensure the fuel cannot return to the transferring tank for any aircraft attitude. This is usually performed guaranteeing the presence of check-valves between the tanks interface, whose capabilities are illustrated in Figure 2.2.
When a tank's geometry makes it difficult to use all of the fuel that is stored in it, it is important to plan for this in the design. This is typically accomplished by either ensuring that the feed boost pumps themselves are positioned so that they can pump all of the fuel from a tank, in any flight condition, or by ensuring that both the front and aft positions of the tanks contain components able to pump the fuel to the feeding zone, using the so-called scavenge pumps (most often jet pumps)

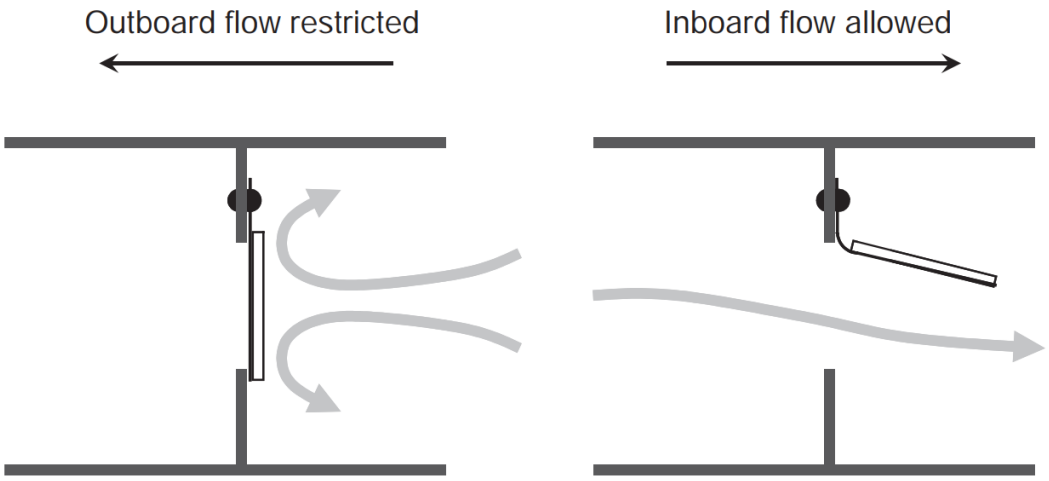

Figure 2.2: Check-valves and their one-directional flow capabilities [2]

[2]. These strategies rely on redundancy, necessitating the installation of at least two fuel pumps in various tank locations to meet the requirements.
Utilizing collector cells is one way to lessen the amount of unusable fuel or, at the very least, reduce the number of pumps. In this scenario, the feed tank is further divided, and a small area of it is converted into a closed area known as a collector cell. Fuel can enter this cell through check valves and scavenge pumps, sucking it up from the tank's bottom and adding it to the collector. This minimizes fuel waste and keeps the boost pumps inside the collector submerged at all times, even when performing negative-g maneuvers.

2.1.2 Fuel provision to the engines and APU

Engine feed is one of the main tasks to be executed by the fuel system and stringent requirements are put into place in order to guarantee the aircraft's airworthiness. Engine feed can be of two main types: gravity-fed and pressure-fed systems.
The easiest way to feed an engine is through gravity, and gravity-fed systems take advantage of the higher position of the tanks in relation to the engine to passively feed it through gravity. As a result, the reliability of the entire system is increased. The drawbacks of this type of system include the inability to feed at high pressures and the requirement that the head between the engine and the tank always be positive, which precludes its use in larger and more complex aircraft. The fact that gravity-fed systems are severely impacted by the vapour lock, a phenomenon that prevents fuel from flowing to the engines in high altitude and high temperature conditions [23], is another disadvantage of these systems. Vapour lock can be solved by adding fuel pumps to the system, able to provide positive pressure head to the line. The addition of pumps turn the system in a pressure-fed system. These systems are widespread in larger and more advanced aircraft and generally make use of motor-driven pumps in order to feed the engines.
According to regulations CS 25.955 [1] [2], aircraft engines must have dedicated feed pumps able to guarantee *at least* 100% of the fuel flow required in all operative conditions. In addition to that, CS 25.991 requires that the engine feed must be guaranteed by the main engine pump in any condition. Whenever this is not possible, a secondary stand-by pump must be available.
In contemporary aircraft, the engine feed is typically accomplished by two identical pumps placed side by side. Different strategies are used, with the most popular ones being either that both pumps always perform the engine feed at once or that one pump can be left in stand-by [2]. In the first scenario, it's important to dimension the pumps so that, in the event of a pump failure, a single pump can still supply the engine with the entire feed flow. In the second scenario, it's important to make sure the stand-by has a quick enough response time to prevent engine starvation in the event of a failure [1].
It is essential to address the interactions between the various functions in the fuel system. For instance, the location of the fuel storage tanks and the placement of the engines have a significant impact

on the design. An aircraft with wing tanks and rear-mounted engines will need longer, heavier fuel lines that must most likely pass through pressurized areas like the cabin. This requires double-walled fuel lines, which adds to the system's overall weight.
The main component responsible for feeding the engines and the Auxiliary Power Unit (APU), together with the supply lines, are the fuel pumps. These pumps were described to a certain extent in Section 2.2.2. Pumps must be able to supply fuel to the engines right away, so the aircraft designer must take special care when selecting a pump from the available options. For example, some architectures call for an auxiliary pump to start and restart engines, which cannot be done by more basic, passive pumps such as jet pumps. Furthermore, the designer must take into account the complete flight envelope and the overall aircraft requirements when deciding the architecture and the positioning of the pumps. Negative-g conditions are particularly interesting and present an extra level of difficulty when designing the fuel system, as the designer must ensure the continuous operation of the pumps for negative-g conditions up to 8 seconds in order to comply with airworthiness requirements [1] [24].
In addition to pipes and pumps, low-pressure valves are an important component of the feed subsystem because they are responsible for shutting off the fuel flow to an engine when necessary. This is the final interface between the fuel system and the engine system, and it serves as a final resource to stop feeding the engine when it is damaged or fails to operate.

Crossfeed

In the event that the engine's primary feed tank fails, multi-engine aircraft have the important feature of being able to fuel the engines from a different feed tank. Fuel crossfeed enables fuel to be moved from one side of the aircraft to the other in the event that a feeding tank, pump, or line fails or is damaged. In situations where there is an imbalance between the left and right wings, crossfeed can also serve as a fuel transfer system, assisting in controlling weight and stability.
It is typical practice to implement a crossfeed system with a dual crossfeed valve architecture in aircraft for which ETOPS (Extended range twin operations approval) certification is necessary [2].

APU feed

The APU is a small gas turbine that can power a portion of an aircraft's electrical and pneumatic loads. This makes the aircraft less reliant on ground systems and enables it to operate on the ground even in airports where ground systems are unavailable or only partially available. The primary function of the APU is to enable main engine starting, as well as to provide electrical and pneumatic ground capabilities for de-icing, air conditioning, and pressurizing. In addition to these functions, the APU can be used in flight to supply additional energy (electrical and pneumatic) to the aircraft in case of emergency, guaranteeing basic aircraft functions like the actuation of control surfaces [25].In these cases, the APU is considered an essential APU [1] and additional requirements are set to guarantee airworthiness. CS 25J951 requires the fuel system to provide fuel flow at a rate and pressure such to keep the APU operable at any flight condition, without incurring in flame-out.
If the APU is considered to be essential, CS 25J953 requires that *"fuel system must provide either a supply of fuel to the APU independent of the supply of fuel to the engine, or, if not independent, then a shut off means for the fuel flow to the APU must be provided. Similar to the engine fuel supply, CS 25J991 requires that there must be an emergency or second main pump to the supply the APU, when the primary main pump fails"* [24], requiring the APU line to be directly connected to the engine feed gallery, in order to be fed by the main feed pumps in case of APU pump failure.
In order to meet the requirements, a further shut-off valve is needed as an interface between the APU feed line and the main engine feed line.

2.1.3 Fuel transfer between tanks and fuel jettison

In order to move fuel between fuel tanks, two strategies are possible [1] [4]. The *Quantity Sequenced Transfer System* architecture is Airbus' choice of architecture and is mostly used in larger aircraft with multiple fuel tanks [2]. Typically, a main feed tank, which is solely in charge of feeding the engine, supplies fuel to it individually. Afterward, depending on the transfer philosophy defined by the aircraft constructor, the fuel from the other tanks is transferred to the main tanks using a variety of architectural designs. Transfer can be gravity assisted, using baffle check valves, or it can be done using fuel pumps. The advantages of this system are its simplicity when multiple tanks are in place

and the fact that it can rely on less fuel pumps, especially if gravity-transfer from the outer tanks is applied. The choice of transfer strategy has a significant impact on the size and criticality of the pump, which has an adverse effect on reliability. Transferring fuel gradually has the potential to be less risky than doing so only when the main tanks are at low or nearly empty levels [1]. In fact, in such circumstances, if a transfer system failure occurs, the pilots may have very little time to react, which could result in a catastrophic failure if mitigation strategies are not properly implemented.
The fuel levels oscillate between full and transfer levels during gradual transfers, the best approach when using this transfer technique. This causes the pumps to experience increased wear, which ultimately shortens the time between maintenance intervals. It is essential to consider ways to shorten the duration of such maintenance tasks in such circumstances. Implementing bottom-mounted boost pumps with a cartridge-in-canister configuration is one way to achieve this.
A second strategy, which is mostly used by Boeing, is to implement an *Override Transfer System*, which has the advantage of requiring no pump to the transfer process. In fact, because all tanks have the ability to perform engine feeding, in such architectures the transfer system and the feed system overlap. Despite its benefits, this transfer architecture is frequently used for architectures with no more than three fuel tanks because it becomes more complex as the number of tanks increases [2].
In its functioning, fuel tanks remains completely full until the previous tank has been emptied and its pump deactivated. This is possible because the pumps in the first tank are made to produce a pressure that is significantly higher than those of the other pumps. Due to the higher pressure of the outer flow, a check-valve is used to stop the flow of the pumps even though they are all running to prevent fuel starvation when a tank empties [2] [1].
A secondary function of the transfer system regards weight and balance and aircraft stability, where designers make use of the large quantities of fuel to maintain the aircraft's center of gravity within the desired range throughout the flight. The most interesting example of this usage can be found on Concorde, where fuel transfer was used to balance the aerodynamic center of the aircraft that shifted between subsonic and supersonic flight phases. Today, large aircraft leverage the use of tail trim tanks in order to properly balance the aircraft itself and to reduce trim drag in flight. In these circumstances, the transfer subsystem in charge of the trim transfer is a critical system, and efforts must be made to ensure its reliability since stability is a top priority that must always be ensured.
Figure 2.3 illustrates the fuel transfer procedure for the Concorde.

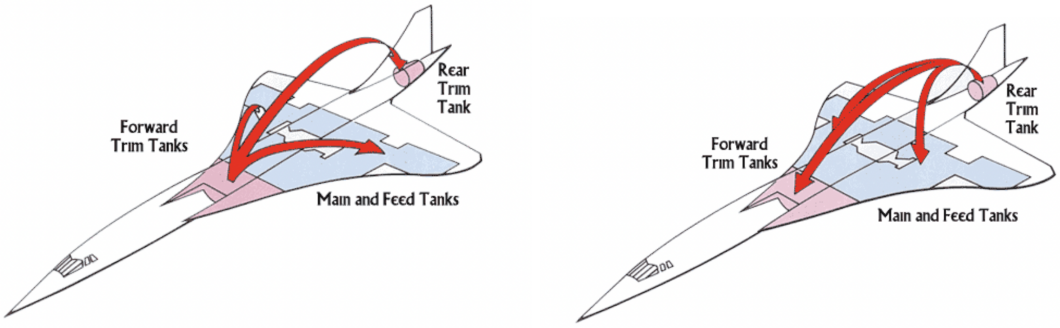

Figure 2.3: Fuel transfer from subsonic to supersonic flight (left) and from supersonic to subsonic flight (right) [3]

A320 transfer system

The A320 follows the transfer system approach as explained in this section. It uses two electrical valves to transfer the fuel from the most outboard wing section, the outer wing tank to the inner wing section - or inner feed tank, and there is no pumping involved in this procedure, since the system leverage the tanks geometry and gravity to perform the transferring. The center tank, together with the inner feeding tanks, have feed capabilities directly to the engine and the fuel transfer sequence is as follows [5]:

1. The center tank is emptied
2. The inner tank goes down to a predefined fuel level

3. The transfer from the outer tanks begin, while the inner tank keeps feeding the engine

The fuel transfer happens automatically with the transfer valves being latched open, and it is possible to manually override transferring routines if needed.

Fuel jettison

Fuel jettison is a technique for losing unused fuel, though it is not always a required part of the fuel system [16]. Given that larger aircraft typically have a maximum takeoff weight that is significantly higher than their maximum landing weight, it is especially crucial for larger aircraft and serves as a safety precaution in the event that a landing is necessary soon after a takeoff. If an emergency landing is needed shortly after takeoff, jettison has the capacity to quickly and efficiently expend a large amount of fuel, enabling the aircraft to land with a manageable weight. This not only reduces the structural landing load as a safety measure but also lengthens the time between maintenance inspections, which lowers the maintenance costs overall.
Jettison systems need to be free of any fire hazards that could lead a fire to spread to the fuel system. Additionally, it must be able to discharge fuel without striking with the aircraft and in a way that preserves aircraft controllability [1]. Moreover, jet fuels have a negative impact on the environment when released into the air, primarily because of their chemical makeup and evaporation patterns. Due to this, in order to comply with environmental regulations, this operation is only permitted in a very small portion of the airspace, above the mixing layer height and below the troposphere [26].

2.1.4 Refueling and defueling

For transport aircraft, the most common way to refuel or defuel an aircraft is by connecting it to a ground service station called "hydrant". For small, general aviation aircraft, the refueling is done by gravity, with the refuel ports on top of the wings, allowing the fuel to be dropped inside the tanks. As aircraft becomes larger, the need to provide a fast and reliable refueling system arises, requiring a pressurized refueling system. In such cases, the refueling time is the main driving requirement and is usually set by the client, most often the airline, whose main goal is to reduce the *turn-around time*. Refuel times for airliners last between 15 and 40 minutes [2] and are often performed by an automated system capable of optimizing the refuel procedure taking into consideration the mission requirements, the allowed center of gravity shift during the operation, protection against spillage, static discharges and over-pressurization.
Refueling ports are typically found under the wing, closer to the leading edge, with the right wing configuration having the most common single port placement. A second port can be added during the design process if a single port is unable to meet the turn-around specifications; this port is typically located on the opposite wing [1].

2.1.5 Fuel measurement and management

Fuel measurement components provide the feedback to the pilot or fuel management system that is required for the previously mentioned functions to be carried out correctly. Fuel quantity indication, also known as fuel gauging, determines fuel quantity based on direct sensor measurements and is subject to various sources of uncertainty. These challenges include, but are not limited to, changes in aircraft attitude, variations in fuel properties due to composition and environmental factors, complex tank geometry, and structural distortion of the tanks.
In order to quantify the fuel that is currently available, aircraft fuel management systems need both continuous and discrete methods, requiring a different type of sensor for each. Today's state-of-the-art fuel gauging technology uses capacitance sensors, providing an accuracy that can reach up to 1/2% of full scale $\pm$ 1/2% of point [2], and the number of necessary probes is determined by the accuracy requirements established for the system as well as tank geometry, as a tank with a complex geometry will need more probes to provide accurate measurement capabilities for varying fuel levels and aircraft attitudes.
The fuel management system is responsible for coordinating, controlling and monitoring all of the fuel system functions, assisting the pilots and reducing their workload. Depending on the size and mission requirements of an aircraft, this system can vary from easy, almost manual systems to very complex systems able to interface with other aircraft management systems to perform in-flight mission optimization, which is the case of modern large commercial aircraft.

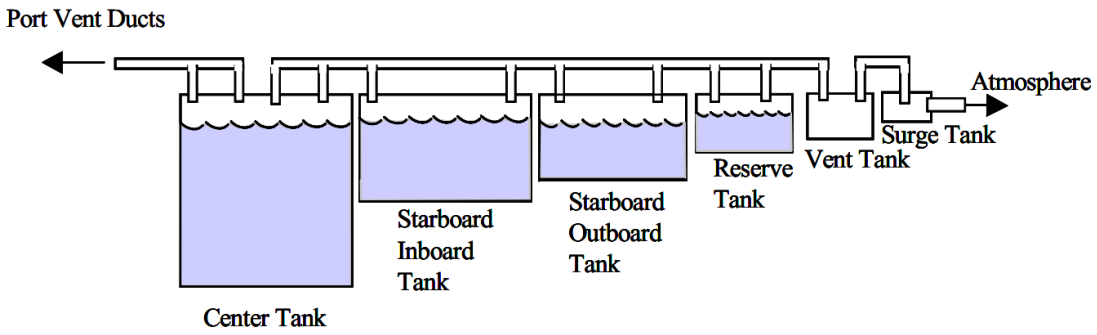

Figure 2.4: Open vent system for an airliner [4]

2.1.6 Venting and pressure management

Transport aircraft are distinguished by an open (non-pressurized) venting system that can keep the pressure differential between the interior of the fuel tanks and the outside environment within acceptable limits, particularly during maneuvering and refueling [4]. The ullage space at the top of the tank is utilized to connect the tanks to the vent system. The wing geometry is a key factor in the venting system's operation because it must guarantee that the moving airspace will always be able to communicate with the vent line and, consequently, with the surge (vent) tank at the wing tip, while also preventing fuel from entering that line [1]. Figure 2.4 illustrates an example of an open vent system for an airliner.
The vent system also needs ways to stop flames from spreading and it needs to stop explosions brought on by a potential flame spread after an incident for at least two and a half minutes in order to give the passengers enough time to evacuate the aircraft [1].

2.2 Fuel system components

The fuel system is a complex system that consists of various components working together to ensure enhanced capabilities that single components could not have by themselves. These components include fuel tanks, fuel pumps, filters, valves, and fuel lines, where each component plays a crucial role in maintaining the integrity and functionality of the fuel system.

2.2.1 Fuel tanks

Fuel tanks are responsible for storing the fuel inside an aircraft. A main challenge for these components is the large amount of fuel they must store, which often requires a deep study of the design space and the possible geometries for the tanks. In fact, the definition of the tank geometries and their positioning are among the first steps in the conceptual design of such a system [2] and often rely on engineers' expertise and previous experiences with similar problems.
In order to ensure a good design that is compliant with airworthiness requirements, several physical constraints have to be taken into account. For example, mass and volume are among the first constraints related to their design. In fact, tanks must be able to support the weight of the stored material and withstand any external forces or vibrations that may be encountered during operation and may cause deterioration and leakage [1]. Additionally, the volume of the tanks must be sufficient to accommodate the desired amount of fuel while also considering factors such as space limitations and transportation requirements.
In addition to that, since the boarded fuel is quite heavy, the tanks' positioning and geometry impact significantly on the aircraft's center of gravity and thus need to be properly studied from the very beginning of the project in order to guarantee the aircraft's overall weight distribution and stability through all aircraft operation phases. Moreover, the tanks' design should account for factors such as fuel flow rates, pressure and temperature variations, and safety measures to prevent leakage or potential hazards.
Aircraft fuel tanks can be divided into two main categories: bladder tanks and integral tanks [2].

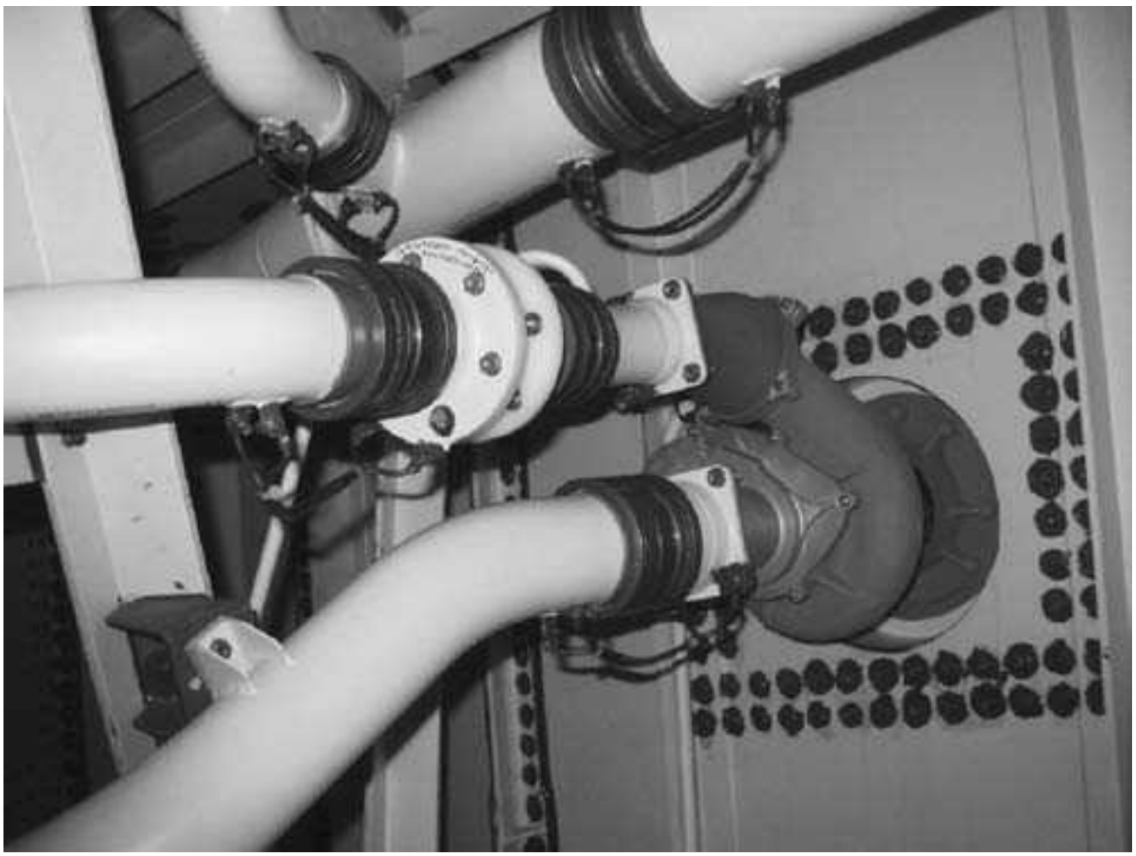

Figure 2.5: Internal details of an integral fuel tank [2]

Bladder tanks are flexible containers that can expand or contract based on the fuel volume, allowing for better weight distribution as the fuel is consumed. These tanks are usually made of thick rubber, which leads to a volume loss of about 10% of the available volume [27]. Nevertheless, this type of tank is still used nowadays, especially in general aviation aircraft and military aircraft for fuselage tanks, due to its self-sealing capabilities (guaranteeing a better survivability rate since these aircraft cannot provide the same level of reliability as larger aircraft due to their limited space), ease of installation and repair, and crash and vibration resistance [27].
Integral tanks, on the other hand, are built directly into the structure of the aircraft, usually between the front and rear wing spars, providing a more streamlined design with smaller center of gravity shifts, but with less weight distribution flexibility. Figure 2.5 depicts the interior of an integral wing tank, where various components such as fuel pipes and fuel pumps can be seen. Integral tanks became popular in the 1950s as aircraft structural stiffness increased in response to higher aircraft performance requirements. Nowadays, the use of carbon-fiber composites allows for the creation of complex shapes, allowing for a 10-15% increase in fuel capacity over standard aluminium structures [27].
The purpose of fuel tanks can also be used to distinguish them. Although storing fuel for the engines is the primary purpose of fuel tanks, they can also be used as transfer tanks, venting (or surge) tanks, and trim tanks:

- Feed tanks: these tanks are responsible for storing the fuel to be supplied to the engines through the pumps. They are usually located close to the engines, and their geometry is often subjected to multiple design constraints , such as space limitations, weight distribution, and certification requirements (e.g. the possibility of a main rotor burst). Feed tanks are designed to ensure a continuous and steady flow of fuel to the engines, preventing any interruptions or fluctuations in engine performance. Additionally, they are equipped with various systems and mechanisms to regulate fuel pressure and temperature for optimal engine operation. In modern commercial aircraft, these tanks are usually the (inner) wing tanks and/or central tanks.

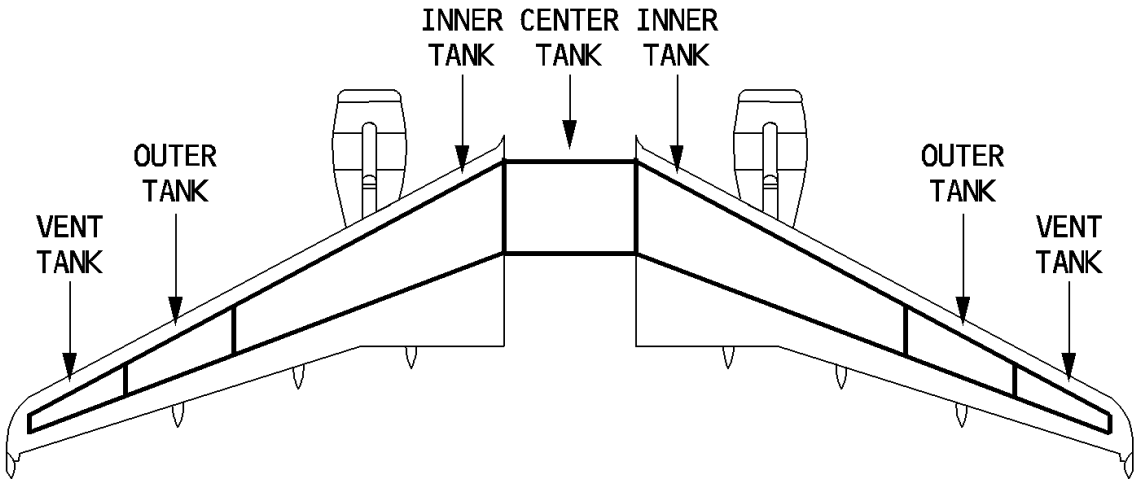

Figure 2.6: A320 Fuel Tanks [5]

- Transfer tanks: not always required, these tanks are used to transfer fuel from one tank to another, ensuring a balanced distribution of weight throughout the aircraft. They are typically located in the wings or fuselage and can be manually or automatically controlled. Transfer tanks play a crucial role in maintaining stability and fuel efficiency during flight. The fuel transfer can be assisted by pumps or rely on gravity, depending on the architecture and location of these tanks. Trim tanks can be considered a subcategory of transfer tanks , as they are specifically designed to adjust the aircraft's center of gravity during flight. Trim tanks are usually located in the tail section of the aircraft and are used to fine-tune the balance of weight, improving overall stability and control. They can be filled or emptied as needed to maintain optimal flight conditions.
- Venting or Surge tanks: these tanks are responsible for gathering fuel that spills from the main tanks, as well as collecting and condensing excess fuel vapor before it exits the aircraft through the overboard fuel vents. They are used to ensure adequate fuel vapor pressure inside the tanks and avoid explosions [2] [16].

Figure 2.6 illustrates the tank architecture for an Airbus A320. In this architecture, both the center tank and the inner tanks have feeding capabilities, while the outer tank is responsible for providing fuel for the inner tanks.

2.2.2 Fuel pumps

The fuel system is equipped with multiple pumps in order to overcome head losses in line and ensure adequate feed pressure and fuel flow to the engines. Additionally, when a straightforward gravity-fed system is unable to meet the needs of the aircraft, fuel pumps can be added to help with the transfer of fuel between tanks. These extra pumps aid in maintaining a steady flow of fuel and avert any potential problems with fuel starvation during flight. Pumps belonging to the fuel system can be of two main types:

- Motor driven pumps
- Ejector pumps or jet pumps

Driven by an electric motor, motor pumps are capable of moving large amounts of fuel. Large and high-performance aircraft are typically equipped with centrifugal pumps. These pumps are motor-driven and work by taking the fuel in the center of an impeller and expelling it to the outside as the impeller spins. These pumps, which are a type of boost pump, are efficient and reliable, ensuring a consistent fuel supply to the engines at a positive pressure head regardless of external and flight conditions, and are the only means of providing fuel to the engines at start for commercial aircraft. Among all possible motor driven pumps, centrifugal pumps are frequently used in aircraft fuel systems

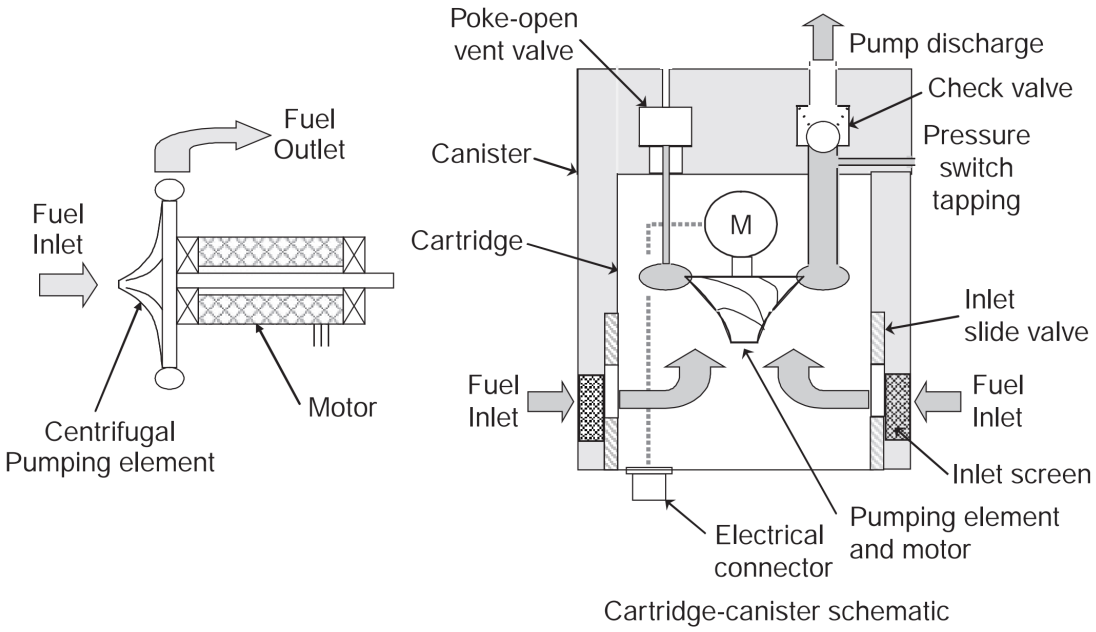

Figure 2.7: Schema of a motor-driven fuel pump [2]

since they enable high flow volumes, low pressure rise, and high reliability [2]. These pumps are usually mounted on the bottom of the tanks in a cartridge-in-canister configuration in order to easily be removed and replaced from the outside during maintenance activities. Nonetheless, different architectures are possible, depending on the needs and requirements set for the system. Figure 2.7 illustrates a cartridge-in-canister boost pump, where it is possible to observe the flow direction, where fuel follows a outwards centrifugal path at its discharge.

In circumstances where it is necessary to keep the pump outside the tank or not submerged at all times, spare-mounted boost pumps are typical (be it due to an unconventional architecture, space optimization, etc.). In this scenario, a "snorkel" inlet capable of being placed at the tank's most advantageous location inside the tank is used to capture the fuel from the bottom of the tank. Due to the additional line that is subject to friction, this solution has a higher head loss (or pressure drop), despite still being very effective. The system's overall performance and pump efficiency may suffer as a result of the increased head loss. Therefore, careful consideration should be given to selecting the appropriate diameter and material for the snorkel inlet to minimize friction and turbulence effects. Additionally, regular maintenance and cleaning of the snorkel inlet may be required to prevent clogging and ensure uninterrupted fuel flow.

Ejector pumps, or jet pumps, on the other hand, use the Venturi effect to create suction and draw fuel from tanks. Because they do not require an additional motor to operate, these pumps are commonly used in smaller aircraft where weight and space are limited. Although jet pumps are extremely reliable due to the absence of moving parts in their design, they do not have self-starting capabilities. In fact, ejector pumps require a high-pressure motive flow, which is typically drawn from the engine feed line, so they cannot be used to start the fuel flow and the engine. This type of pump is typically used in larger aircraft to help move fuel closer to the main boost pumps. It is used as the main feed pump to the engines in some architectures, but in these cases, it must always be paired with a motor-driven pump to ensure fuel flow at engine start and when the jet pump loses its prime. A schematic representation of a jet pump can be seen in Figure 2.8. It is possible to observe that, in order to provide suction capabilities, a motive, high pressure, fuel flow has to be present. Due to the low efficiency of jet pumps, which is around 30% [2], fuel lines must be larger than necessary to accommodate a volume flow that can satisfy engine feed requirements. This is a key reason why ejector pumps are not used in the engine feed of large commercial aircraft because they would require extremely large and heavy fuel lines. On the other hand, ejector pumps are frequently used as scavenge pumps to reduce the unused fuel present at remote corners of fuel tanks [2]. Both types of fuel pumps play a crucial role in maintaining proper fuel flow and preventing any issues that could

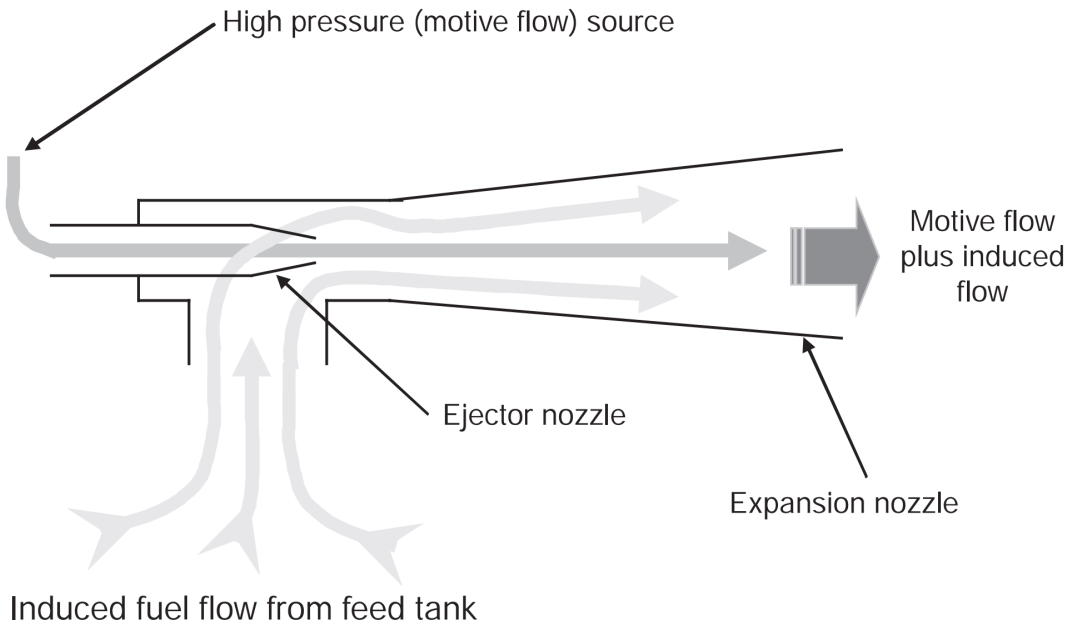

Figure 2.8: Schema of an ejector (jet) pump [2]

arise during flight.

2.2.3 Fuel lines

In an aircraft, fuel is physically moved from one location to another by fuel lines. Fuel lines must meet a number of certification and geometrical requirements, just like the other parts of the fuel system, and their design must be carefully considered because it has a significant impact on the mass of the system. Fuel lines need to be strong and resistant to corrosion and leaks in addition to their size and weight. Many of their design choices are influenced by the need for appropriate maintenance and routine inspections to guarantee the integrity of the fuel lines and avoid any potential fuel system failures during flight.
Recent developments on the field have made possible the usage of lighter material such as PTFE (Polytetrafluoroethylene), allowing to create flexible and lighter hoses, capable of better withstanding vibrations.
Additionally, because fuel lines are essential parts of fuel systems and are constructed in accordance with strict specifications, they are subject to numerous regulations. As an example, pipes must comply to specific line diameters and their positioning is heavily dependant on several geometric requirements depending on their surrounding environment.

2.2.4 Fuel valves

There are many fuel valves in the fuel system, and their main purpose is to control the fuel flow through a particular area of the system. They can be manually (physically) operated or electrically actuated, and they come in a variety of shapes and sizes.
Shut-off valves, which include various valves like crossfeed valves, transfer valves, and isolation valves, are the most frequently used valves in aircraft fuel systems. They provide a bidirectional flow that is guided by the pressure gradient in the line. Check valves are yet another widely used type of valve that can be found (or non-return valves). These valves, which only allow one direction of flow, are frequently used in gravity-assisted transfer tanks to prevent fuel reflux in the line by preventing the fuel from returning to the transfer tank in the event that the aircraft's attitude changes.
Draining valves are a third type of valve that are frequently found in airplanes. In addition to allowing for vapour venting, they also allow for the removal of accumulated water and fuel draining.

2.2.5 Other parts and subsystems

The fuel system is made up of a number of components that are outside the purview of this work and will only be briefly mentioned. The system's need for filters is essential, and their presence can significantly affect a line's pressure drop, which affects how the pumps are sized [2]. The fuel system also consists of a number of fittings and bearings, the purpose of which is to support the lines while allowing them to vibrate and oscillate within a specific range to prevent line rupture and leakage.
Numerous components in fuel systems are also in charge of ensuring that the system has an adequate level of vapour pressure, which prevents explosions. Flame arrester, which stop flames from spreading to an aircraft's interior, and OBIGGS (On-Board Inert Gas Generation) are a couple of the components in charge of this task. The newest inert systems, known as OBIGGS, are found on more recent aircraft. This system is in charge of producing inert gasses that are added to the fuel tanks to lower the fuel vapour pressure and help lower the risk of explosion.
It is worth noting that a variety of aircraft inert systems exist but are outside the scope of this work.
Another category of essential fuel system parts are connection ports, which ensure the connection to the outside world and permit refueling or defueling of the aircraft. Their design is governed by various specifications that must take into account a variety of factors, including the desired refueling time for an airline (if it is a commercial aircraft), pressure, static discharges, and flow specifications that must be in accordance with those set forth for the other subsystems (e.g. the size of the feed and transfer lines may impact the refueling pressure and line diameter).

2.3 Fuel system design

This section gives a brief introduction to contemporary design drivers for fuel systems and serves as a foundation for understanding its key features and how to use knowledge to enable more in-depth analysis of the fuel system during conceptual design.
To come up with the best design solution, system requirements are analyzed and tradeoff studies are put into practice during the conceptual design phase. At this stage few requirements are frozen and aircraft designers have a lot of autonomy in their choices. Although this freedom allows to explore different system architectures, the lack of knowledge of the final product could eventually lead to design inconsistencies that require further changes. Designers must therefore fully understand how the choices they make at this point may have an impact on the entire product lifecycle. Figure 2.9 illustrates the concept definition phase and its iterative characteristic.
As with any other subsystem, the top-level (or aircraft level) requirements serve as the foundation. These include, among other things, the intended mission the vehicle is to perform, reliability and operational availability goals, which specify the degrees of redundancy that a system must meet, system management capabilities, the design of the electrical plant, and the overall system architecture. The intended aircraft mission specifies the mission range and any additional certification requirements, such as ETOPS availability, that the aircraft must meet. In order to size and position the fuel tanks in an aircraft, this is one of the most important requirements that must be followed.
In addition to tank sizing, ETOPS requirements specify the amount of redundancy that crossfeed lines must have and any measures that must be taken to ensure fuel rerouting when necessary. Costs are a different class of influencing factors that direct design. As engineers define the fuel system architecture, the committed costs, which define the overall cost for a product's lifecycle [13], are frozen. It is essential to offer both technically and economically viable solutions from the very beginning of the design process in order to maximize the use of the available resources and to ensure the competitiveness of the final product.
Due to their large volume, the task of placing the fuel tanks require a great effort throughout the process. For modern commercial aircraft, with fuel tanks in its wings and engines below the wings, the possibility of a Main Rotor Burst (MRB) must be taken into account during the design. In fact, the release of high energy rotating parts from the engine could lead to ruptures close to its placement, thus affecting the fuel tanks and other components. It is necessary to provide precautionary means to reduce the possible damages and to mitigate any accident possibility caused by them. According to AMC-20128A [17], where this failure mode could affect the tanks, dry-bays can be defined as an acceptable design precaution to avoid high energy fragments of damaging the tank. These bays are empty spaces left in the airframe and their geometry can span from very simple to complex geometries, depending on the fragment's possible trajectories, which influences the available space and geometry of the fuel tanks.

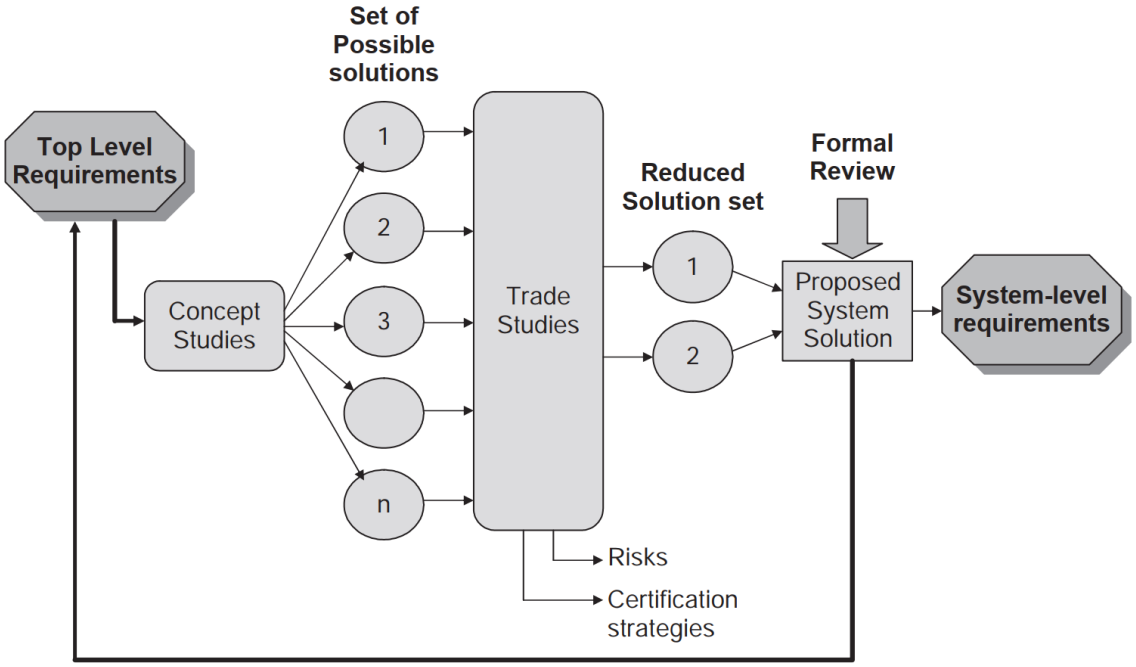

Figure 2.9: System concept definition [2]

Fuel system is subject to many different requirements and require large amount of information in order to correctly design and size its components such as pumps, valves and lines. Due to its high complexity and inter-dependency with several other subsystems, this system can greatly benefit from early design studies and definition. A design strategy capable of leveraging previous knowledge to enhance design capabilities is described in Chapter 3.

Chapter 3

Knowledge Based Engineering

3.1 Introduction

Aerospace is a highly technological and complex industry where the client plays a decisive role not only in the customization process but also in defining the design requirements at an early stage. Competences, relationships, and capabilities are crucial sources of competitive advantage and profits in a globalizing industry where trade is fiercely contested. In fact, the capacity to fulfill customer needs is strictly associated with the integration of technologies, resources, organizational roles, and fundamental procedures. This aspect cannot be separated from the emerging new patterns of industrial relations, which have changed in response to shifting market demands. In recent years, traditional relationships among subcontractors have been replaced by mergers, acquisitions, joint ventures, and alliances in order to share and combine all types of resources and to develop the necessary competencies [28]. These new patterns have also led to a greater emphasis on flexibility and adaptability in the workforce as companies seek to respond quickly to changes in the market.
Additionally, technological advancements have played a significant role in shaping these new industrial relations as companies increasingly rely on automation and digitalization to improve efficiency and reduce costs. In fact, the primary drivers of the aerospace industry have shifted in recent decades in favor of lowering total cost and lead time. Over 30% of the total cost of an aircraft is thought to be spent on production, but 80% of this cost is decided upon during the conceptual design phase [29], when engineers have more freedom in their design choices. As a result, engineers are beginning to think about how their decisions will affect production early in the design process.
In the current aerospace economy, complex projects are often conducted 24/7 around the globe. This creates another challenge regarding the limited communication between different teams working in different locations, leading to confusion and misunderstanding of design choices.
Moreover, it is estimated that in 20 years, the aeronautical systems will be as different from today's systems as the actual systems differ from those from 1930 [30]. To cope with these changes in such a short time, a collaborative effort between different departments within a company is required, as well as with external partners, to ensure that the design is feasible and can be produced efficiently.
The use of advanced simulation tools and digital twins has also become increasingly important in the aerospace industry, allowing engineers to test and optimize designs before they are manufactured. This not only reduces costs but also improves the overall quality of the final product. As the industry continues to evolve, companies that are able to adapt quickly and effectively to changing market demands while also leveraging technological advancements will be the ones that succeed in the long run.
In addition to the difficulties already mentioned, the aerospace industry is experiencing rapid growth and a shortage of qualified workers. At the same time, it is also dealing with an aging workforce, with an expected retirement rate of 80% in its experienced workforce in the next few years [31]. This issue calls for fresh approaches in order to more effectively store and repurpose the knowledge of seasoned workers and to train and assist new engineers through knowledge transfer.
Considering this technological competitiveness, various methodologies have been developed and are currently being built to attempt to solve most of the problems. Knowledge-based engineering (KBE) is one of the most promising of these. There are numerous definitions of KBE in the literature, all of which are accurate, but almost none of them fully explain what KBE is. One of the foremost authorities in the area, Gianfranco La Rocca, appears to offer the definition that best satisfies researchers

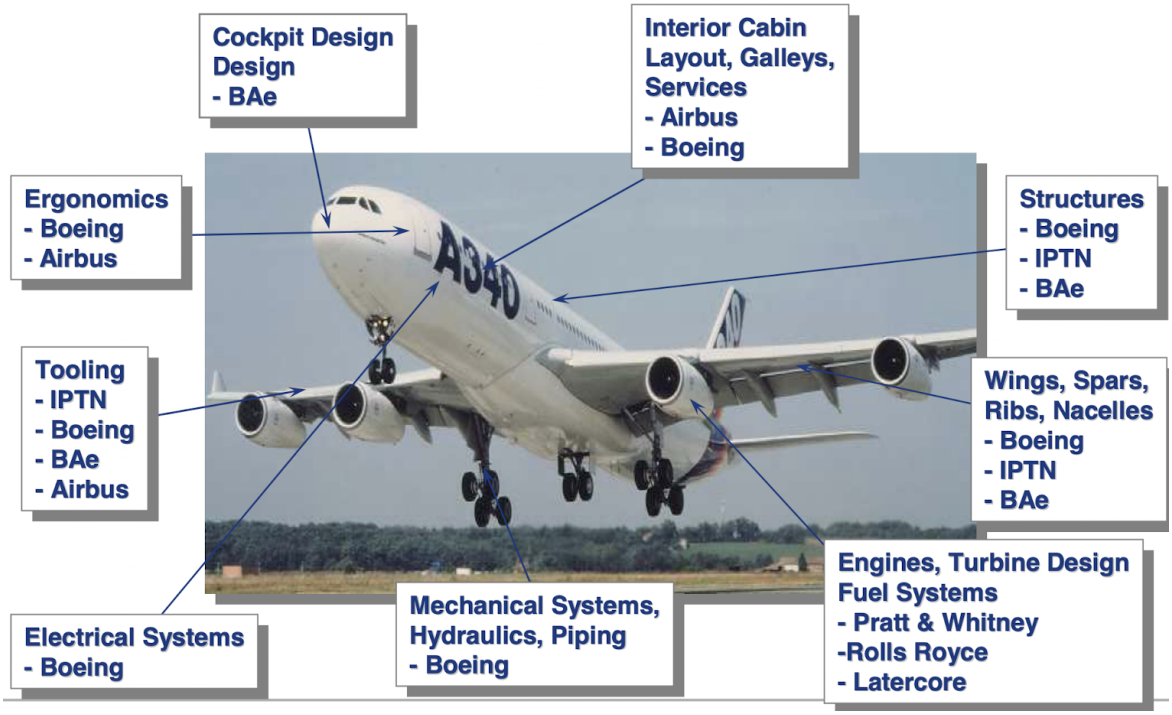

Figure 3.1: A380 - system providers

from various fields and tries to cover KBE as a whole: "Knowledge Based Engineering (KBE) is a technology based on the use of dedicated software tools (i.e., KBE systems) that can capture and reuse product and process engineering knowledge. The main objective of KBE is to reduce the time and cost of product development by means of automation of repetitive, non-creative design tasks and by supporting multidisciplinary integration starting from the conceptual phase of the design process" [12]. KBE systems can also improve product quality and consistency by ensuring that design rules and best practices are followed consistently across different projects. Additionally, KBE can facilitate knowledge sharing and collaboration among designers, engineers, and other stakeholders involved in the product development process.
This chapter aims to give a thorough overview of knowledge-based engineering methodologies and their application in the aerospace industry, outlining their main advantages as well as their current issues and challenges.

3.2 Basic concepts of KBE

Knowledge-based systems (KBS) are a type of artificial intelligence that uses knowledge representation and reasoning techniques to solve complex problems. When applied to engineering problems, these methodologies are often called knowledge-based engineering (KBE).
KBE is still a recent innovation in industrial engineering, despite not being brand new. Its applications are primarily found in the automotive and aerospace industries, industries known for requiring a high level of customization of the final product, which can be seen as a variant of a family of products [21]. In these industries, KBE has been used to speed up time to market and enhance design processes, and one of the main advantages of KBS is their ability to capture and reuse knowledge, which can lead to significant improvements in productivity by means of automating decision-making procedures.
Figure 3.1 depicts how the aerospace industry is currently developing its products, with a well-known segmentation and globalization of the entire supply chain. Knowledge-based systems can aid in the integration and streamlining of this process by facilitating communication and collaboration among various supply chain segments. This can eventually lead to shorter development times, lower costs, and higher overall product quality.

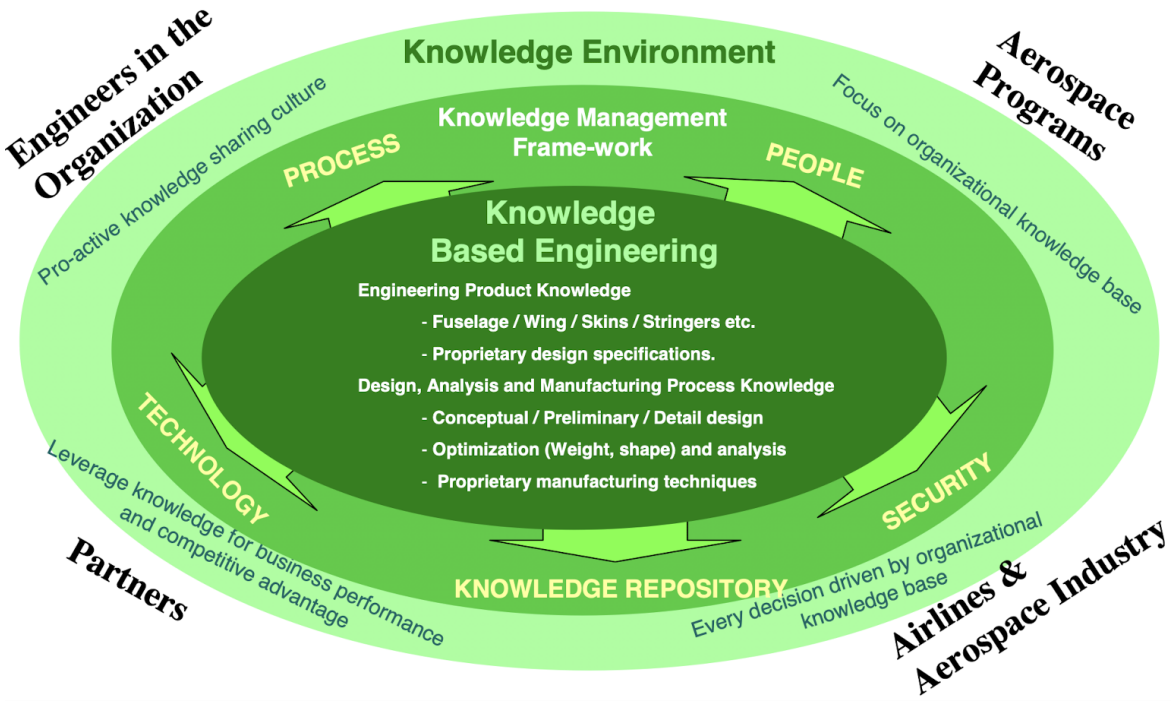

Figure 3.2: Knowledge framework

3.2.1 Traditional design method vs. knowledge-based design

Traditionally, the design and production planning phases are carried out independently of one another, with little to no communication between the involved parties. In this situation, designers oversee creating a design that will later be given to production designers, who will assess the model's viability in light of manufacturing and assembly procedures and give feedback to the design engineer. As a result of a lack of integration between design and production, the process iteratively continues until it is satisfactory to the designer and the production engineers, which results in a lengthy development period. Additionally, because of the difficulty in communicating between the stakeholders, it is common to overlook factors like maintenance that are usually considered later in the design process. As a result, the later a flaw is found in the design process, the more expensive product development becomes.

To address this issue, many businesses are now taking a more comprehensive approach to product development that considers the entire product lifecycle. This approach entails collaboration among various departments and stakeholders, such as engineers, designers, marketers, and customers, to ensure that all aspects of the product are considered from the start. Companies can reduce development costs while improving product quality and customer satisfaction by doing so. Designers can use KBE to re-use knowledge generated in previous projects, increasing creative design time, reducing the number and duration of design iterations, and improving collaborative design to ultimately reduce development time [29]. In fact, 80% of engineering tasks are repetitive and routine, while the remaining 20% call for the designer's creative abilities [7].

KBE combines CAD technologies and artificial intelligence (AI) techniques, often using object-oriented programming (OOP), to represent an evolutionary step in computer-aided engineering (CAE). While older definitions were narrower and technology-oriented [13], when KBE was previously described as a CAD system extension that mainly provided geometric functions, experts are now aware of KBE's power outside the geometric domain. These systems aim to capture not only product knowledge but also process knowledge regarding the entire lifecycle to enable businesses to model engineering design processes and then use this model to partially automate the process itself by means of rule declaration. The emphasis is on providing the most complete product representation possible, based on previous knowledge of similar products.

Figure 3.2 depicts the entire knowledge framework that must be implemented in order to successfully apply KBE with results. It's interesting to see how this approach considers more than just engineering tools and is entirely dependent on the company's business strategy and vision.

3.2.2 Knowledge management

Knowledge management focuses on the use of techniques and tools to enhance the usage of knowledge assets in an organization and concerns the identification of important knowledge for a business, deciding what knowledge should be captured to provide adequate solutions to problems, and capturing, representing, and storing knowledge in order to reuse it [6]. When using KBE, the lifecycle starts by eliciting raw, unstructured knowledge. In a study conducted by Sanya et al. [7], five main sources of knowledge were identified:

- Geometry constraints
- Engineering processes
- Data types
- Algorithms and design rules
- Written word

These sources are mainly stored in three different repositories: human, document, and digital repositories.
The elicitation process varies depending on the type of knowledge. Knowledge can be seen as a system that drives the process of extracting data and information to generate actions and decisions. Figure 3.3 provides a few knowledge definitions that are used in academia when working with KBE. In [6] the following definition of knowledge is given: Knowledge can be classified in different ways, but two

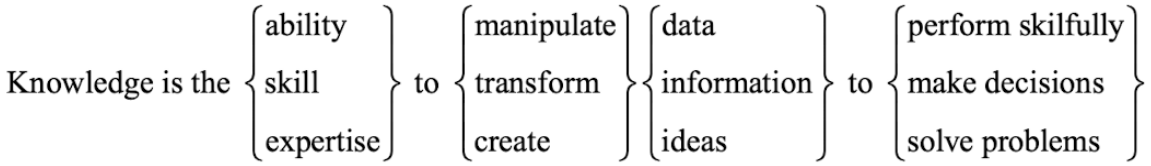

Figure 3.3: Knowledge definition [6]

important factors when describing it are:

1. Procedural knowledge versus conceptual knowledge
2. Explicit knowledge versus tacit knowledge

Procedural knowledge, which can be thought of as processes, tasks, and activities, is used to describe the circumstances in which duties are carried out, the order in which they are conducted, and the resources that are required to carry out tasks. Instead, conceptual knowledge can be thought of as a description of concepts and their relationships, addressing how different objects and properties relate to one another. This type of knowledge is often more abstract and theoretical and can be applied in a variety of contexts. For example, understanding the concept of supply and demand can be useful not only in analyzing economic trends but also in understanding how pricing decisions are made in a retail setting.
One fundamental type of knowledge that occupies the foreground of a specialist's mind is explicit knowledge. It has a straightforward "train of thought" and is simple enough for a third agent to externalize and understand. This kind of knowledge focuses on fundamental concepts, fundamental properties, and fundamental duties performed by a domain expert. The engineer's experience is inextricably linked to tacit knowledge, which is a deep form of knowledge. It is difficult to impart this knowledge through traditional educational methods because it relates to automatic processes that use subconscious expertise as an input. It can be referred to by a variety of common terms, including "hunches," "gut feelings," "intuition," and "instinct."

The steps of knowledge management

Knowledge management is a complex field of study and has multiple steps, from its raw acquisition to its formalization and maintenance. The process begins with the application of knowledge acquisition techniques, whose goal is to capture and structure knowledge. Several researchers have discovered, through various case studies over the years, that knowledge from domain experts is difficult to obtain. While explicit knowledge is relatively easy to extract (via books, equations, etc.), expert knowledge is typically difficult for knowledge engineers to isolate, as engineers frequently do not recall or know how to explain everything they know. Furthermore, different experts may hold alternative opinions on the same issue, making it difficult to obtain a single coherent picture. This problem, combined with the fact that domain experts develop specific conceptualizations and mental shortcuts for their reasoning, makes it difficult. As a result, it is important to approach expert knowledge with a critical and open mind, seeking to understand the underlying assumptions and biases that may influence their perspectives. Additionally, interdisciplinary collaboration and communication can help to bridge gaps in understanding and facilitate a more comprehensive understanding of complex issues.
To manage these difficulties, a number of tools have been developed in order to better capture knowledge. Figure 3.4 illustrates a knowledge acquisition matrix, which provides multiple tools depending on the knowledge to be extracted. Different methodologies and software tools are available to perform knowledge acquisition, some of which will be discussed in the next sessions.

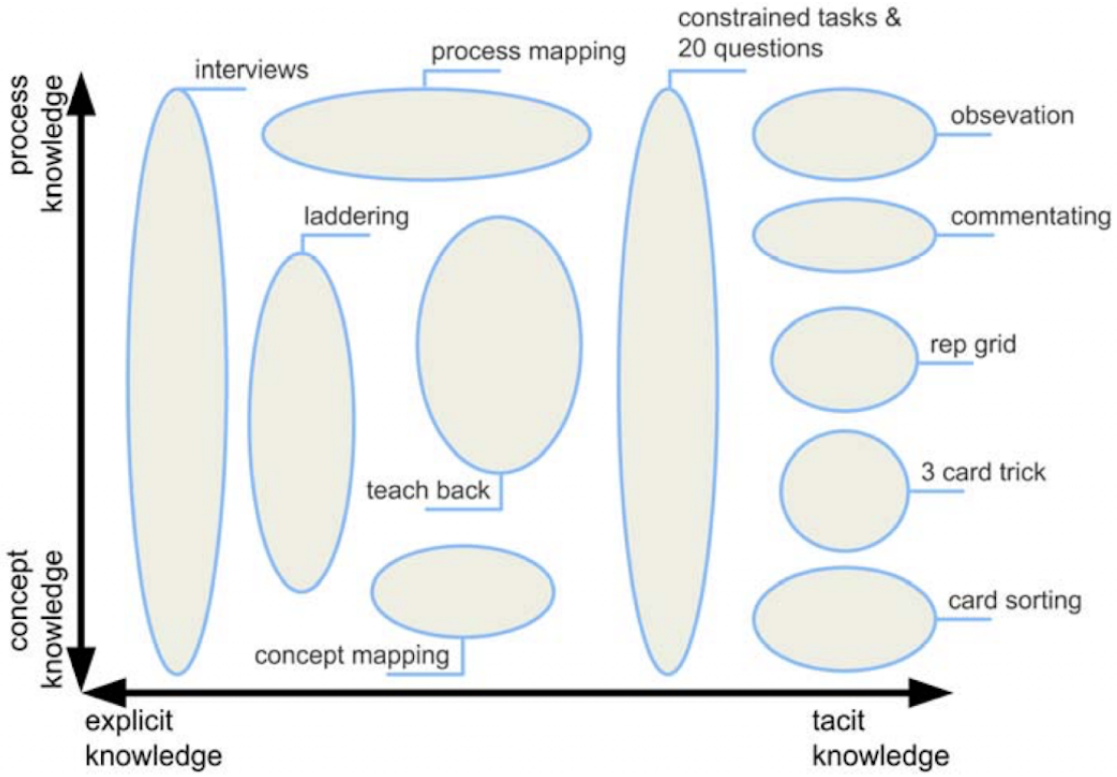

Figure 3.4: Knowledge acquisition matrix [6]

In Figure 3.5, the whole knowledge management (or knowledge engineering) process is represented. The process starts with the process analysis, a phase in which the company's needs and the most promising areas in which KBE could be applied are identified. This phase is intrinsically related to the high management of a company and is decisive for the KBE implementation. In fact, due to the high costs of the creation and maintenance of KBE tools, this strategy can only be executed once significant gains can be identified for the business. The deliverable for this phase is a value map that contains promising opportunities to apply KBE.
The second step is knowledge acquisition, which has been explained in the previous paragraphs. This step is followed by the knowledge structuring phase, which is responsible for formalizing the captured knowledge and incorporating it into machine-readable information useful for task automation. After

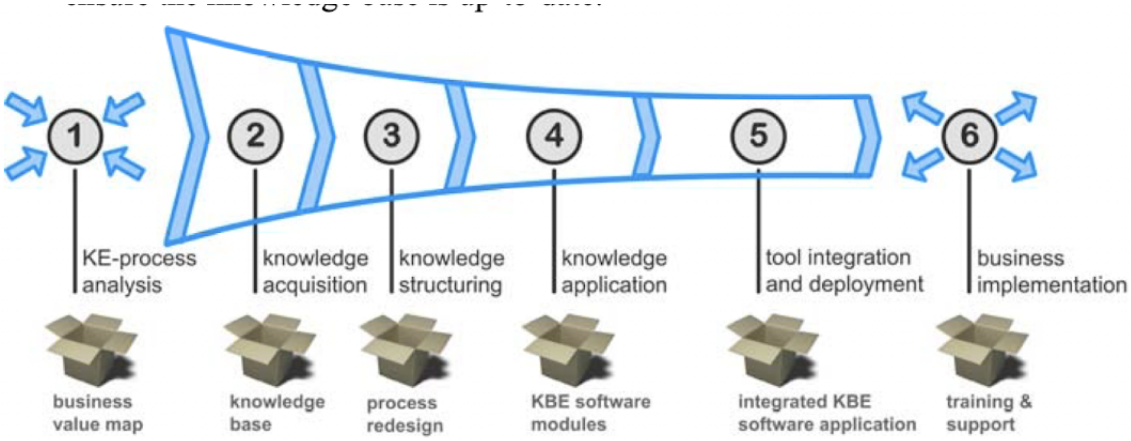

Figure 3.5: Knowledge management process [6]

having identified the needs and captured and structured the knowledge, the knowledge application phase comes into action. This phase concerns software development and is based on the formal model previously defined using KBE methodologies.
The fifth phase concerns the integration of KBE modules into the framework. In fact, one of the most important aspects of defining a KBE environment is allowing it to be modularized. This has two important outcomes. First, the system must have a clear separation between knowledge and its execution. In modern aircraft design, different companies must share product information in order to achieve the desired goals. These companies may sometimes be competing companies in different fields and may be reluctant to share sensitive information with the project partners. This clear separation between knowledge and execution allows companies to use the same knowledge framework without sharing company knowledge. In addition, it allows companies to expand their knowledge to better represent their needs. To satisfy these requirements, KBE systems should be as generic as possible [29].
Second, this modularization enables each discipline specialist to work with their own trusted tools [30] and to change and/or expand them as needed without rendering the entire framework inoperable for the various stakeholders. For example, in the design of a new aircraft, the aerodynamics team may use their own specialized software to generate wing designs and simulations, while the structural engineering team may use their own tools for analyzing stress and deformation. These tools can be integrated into a KBE system, allowing for seamless communication and sharing of data using a single model between different teams without compromising proprietary information. The modularity also allows for future updates and improvements to be made to each tool individually without disrupting the entire system. This modularity enables the tools to be reused in future projects and makes maintenance easier [32].
Figure 3.6 shows how most KBE applications are currently configured and how they should operate, according to researchers. It is possible to detect a discernible trend toward software that is language-independent (or platform-independent) and able to store various types of information from various sources using a common language. When complexity is involved, ontologies or markup languages can be used to accomplish this. Each tool is then connected to the framework through an API and transforms this model as necessary. The sixth and last step in the process is support, maintenance, and training. In fact, regular updates and revisions are necessary to ensure that the knowledge remains relevant and accurate. This can be achieved through continuous learning and improvement processes as well as effective communication and collaboration among team members. Moreover, providing ongoing support and training to employees can help them stay up to date with the framework and sponsor its usage.

3.2.3 KBE and the design process

When considering KBE systems, academia and industry tend to have different opinions on their usage. Academia tends to see KBE systems as a theoretical foundation based on multiple domains and product knowledge that must remain as generic as possible, even before considering them as a

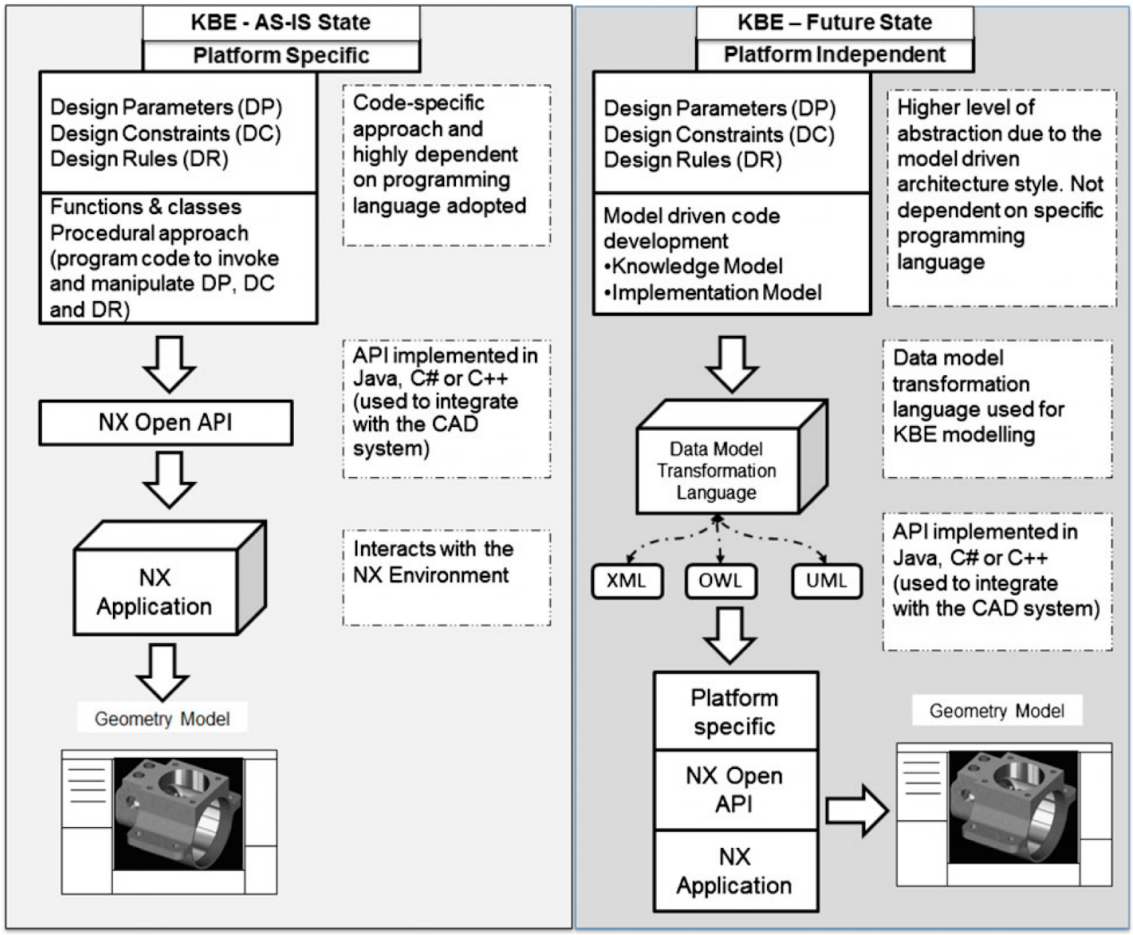

Figure 3.6: Platform specific vs platform independent KBE [7]

system or a tool, ensuring maximum modularity and flexibility [33]. That is, ideally, a system that can store any and every piece of knowledge available in a company and, sometimes, automate tasks with it. Industry, on the other hand, has a much more pragmatic view of KBE, where it is seen as solving a specific business need in order to achieve tangible needs that must justify the investment. These needs are usually expressed in terms of reduced lead times and product development costs. In such cases, companies tend to neglect the development of a complete KBE framework and instead adopt design automation (DA) strategies. Design automation (DA) strategies are focused on automating specific design tasks, while a KBE framework involves a more comprehensive approach to knowledge management and automation. However, it is important to note that a KBE framework can provide long-term benefits, such as increased flexibility and adaptability in product design.
Tasks that lend themselves to automation usually have one or more of the following characteristics [33]:

- Simplification of more complex design tasks
- Integration of tools and datasets
- Documenting and report generation
- Low level, repetitive, and/or highly manual procedural tasks
- Generation of manufacturing data and tooling design

When it comes to design strategies, the traditional process is heavily reliant on the quality of information exchanged between stakeholders and necessitates consistent and effective communication. Furthermore, this process is characterized by an excessive amount of manual labor to provide adequate information exchange between designers, simulation, and production engineers, each of which

requires specific models, in specific formats, and containing specific information, which, when combined with an increased number of design iterations, significantly increases project development time and costs.
On the other hand, the deployment of a KBE framework allows experts from different disciplines to work on a unified model with their desired analysis tools. This process, paired with greater product knowledge at earlier stages of the product development cycle and automation techniques, allows for easier model generation and requirements verification for a more multidisciplinary project development. With lower iteration cycles and fewer misunderstandings between stakeholders, designers are able to use more of their time to creative and critical thinking activities that eventually lead to enhanced product quality by executing a more detailed design evaluation and analyzing more candidate designs in the same amount of time as before[32][8]. By doing so, it is possible to reduce the level of abstraction and increase the knowledge of a product at the early design stages, as depicted in Figure 3.7. However, a counterexample to the benefits of a KBE framework can be seen in small-scale

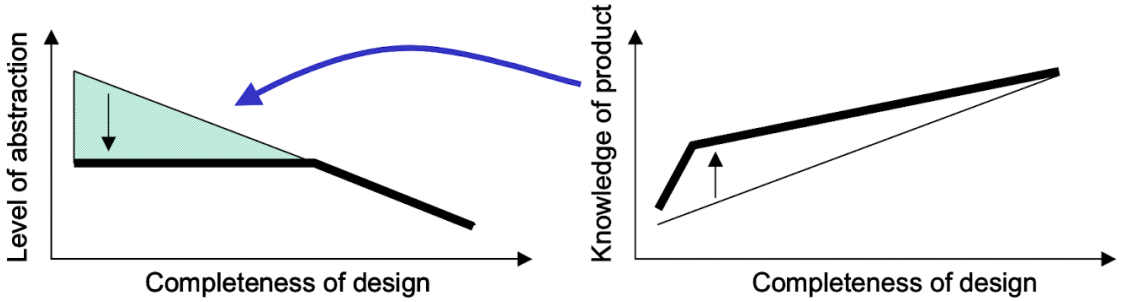

Figure 3.7: Design completeness with respect to the level of abstraction and the knowledge available [8]

projects where the cost and time investment required for implementing such a framework outweigh the benefits. In such cases, design automation (DA) strategies may provide a more cost-effective solution for automating specific design tasks.

3.2.4 KBE methodologies

When dealing with knowledge-based systems, it is important to clearly define the goals before starting the application development phase. In [34] four categories of deliverables by KB systems were identified:

- Automation of narrow tasks: this includes automation of rudimentary tasks. An example would be the automation of primitive geometries for CAD tools (e.g., lines, circles, etc.)
- Model automation and data abstraction: these include high level operations capable of adding new knowledge to the previous model, such as adding primitives together to create a complex shape, and programming tools such as application programming interfaces (APIs).
- Design process automation consists of automating different engineering tasks made up of lower-level tasks (narrow tasks, model automation, and abstraction) into a single process, automated by means of specific parameters given by the user. An example of design process automation is the use of parametric design software, which allows designers to create complex shapes and structures by inputting specific parameters and rules.
- Exploration of the design space and discovery of new solutions add a higher level of complexity to the tool by means of reasoning, or semantics. By making use of a library of multidisciplinary knowledge, it is possible to solve new problem situations using explorative means. It enables the tool to adapt to changing problems and requirements, making it more versatile and adaptable.

To enable the development and deployment of KBE tools, different methodologies were developed in an attempt to formalize and improve the KBE application design process. A few examples of these methodologies are [9]:

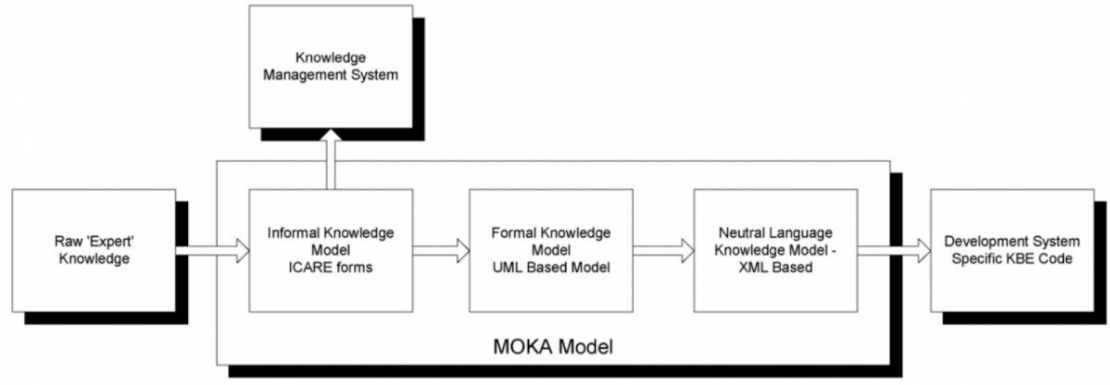

Figure 3.8: MOKA knowledge models [9]

- MOKA (Methodology and tools Oriented to Knowledge-based engineering Applications): supports the knowledge acquisition process, focusing on knowledge capture and formalization.
- CommonKADS focuses on the organizational aspects of KBS and how the different stakeholders communicate and relate.
- KNOMAD extends the MOKA methodology to account for life-cycle management issues, focusing on multidisciplinary aspects of KBE applications. It uses ontologies to account for knowledge change, improving the application's transparency [35].

Most knowledge engineering projects include four general phases [10]:

1. Scoping phase, used to define in detail the objectives
2. Knowledge elicitation phase
3. Knowledge modeling phase
4. Knowledge implementation phase

The MOKA methodology

After recognizing that Europe lagged behind the USA and East Asia, a European research project called MOKA was launched in 1998 with the goal of promoting the use of KBE techniques in Europe. The absence of clearly defined standardized methods was found to be one of the primary factors preventing KBE from being widely adopted in Europe.
The phases of knowledge acquisition and structuring, which the MOKA methodology emphasizes, occurred when developing a KBE application, which incurred most costs. The main objectives of MOKA were [9]:

1. Reduce the lead times and costs of developing KBE applications by 20–25%.
2. Provide a consistent way of developing and maintaining KBE applications.
3. Develop a methodology that will form the basis of an international standard.

The MOKA methodology consists of different modules that aim to help the knowledge engineer better extract knowledge from domain experts and structure this knowledge by a common, standardized language to be used by software developers [36]. Figure 3.8 illustrates MOKA's knowledge models, while Figure 3.9 shows the KBE lifecycle according to the methodology. The methodology consists of eight steps, the first of which uses the informal knowledge module, which is structured using ICARE forms (Illustration, Constraints, Activities, Rules, and Entities) to formalize the knowledge capture phase. These forms give the knowledge engineer a method for classifying and connecting the knowledge collected, offering a simple method for categorizing unstructured knowledge, and can be described as [9] [11]:

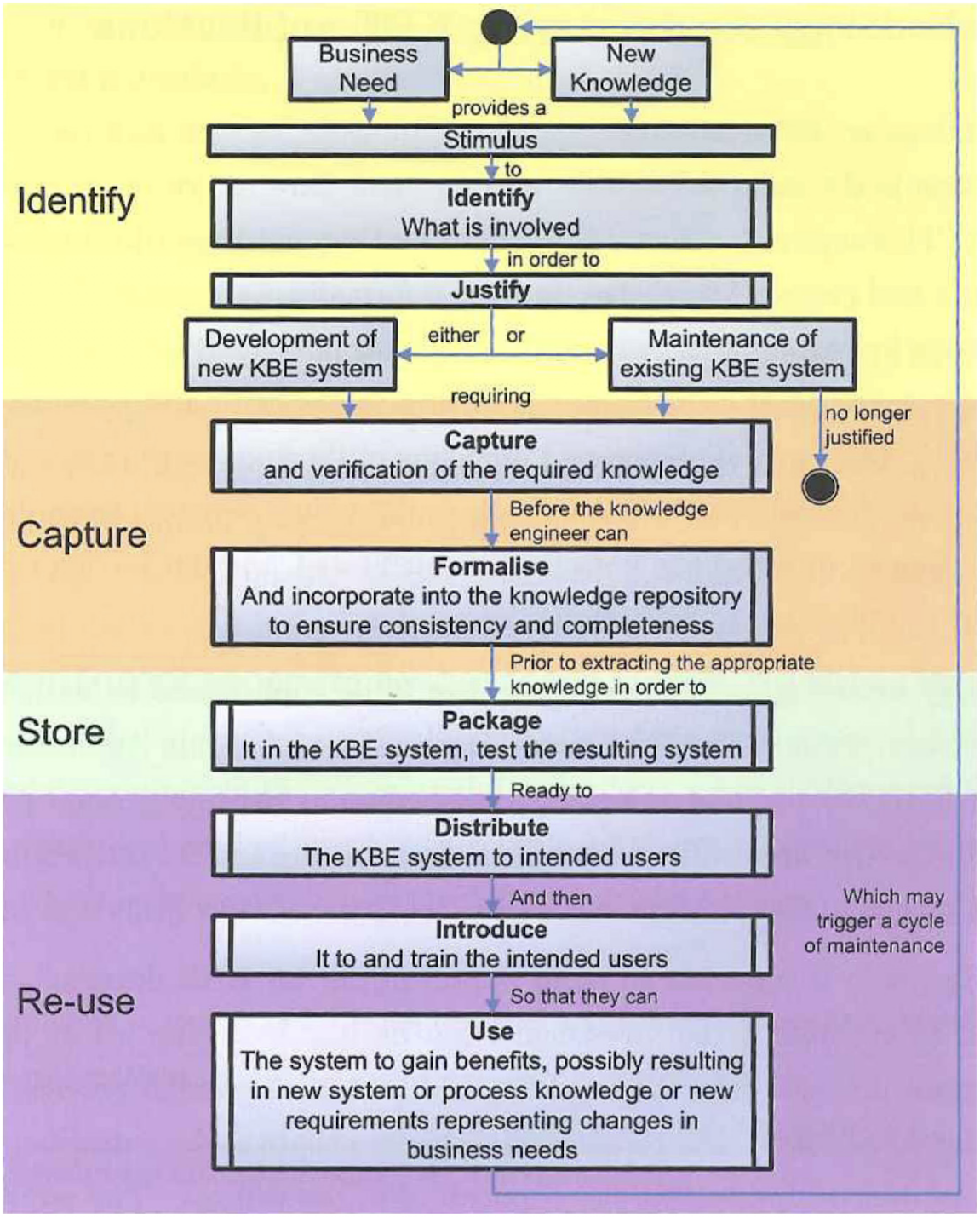

Figure 3.9: KBE lifecycle [10]

- Illustration forms: represent all kinds of generic knowledge using descriptions and comments.
- Constraint forms: model the interdependencies between different entities in the model.
- Activity forms: describe the crisis resolving steps of the design process.
- Rule forms: model the rules necessary to apply the knowledge.
- Entities: describe the product by means of object classes, according to the object-oriented view, defining product structure, function, and behavior.

Figure 3.10 graphically represents the informal model. At this level, the knowledge must be understandable by all the stakeholders: the knowledge engineer, the domain expert, and the software developer, as it is crucial for this acquired knowledge to be validated by all these experts in order to avoid misunderstandings and knowledge misuse. This allows for a shared view of the problem among the three parties. Furthermore, this collaborative effort ensures that the knowledge base is accurate and up to date, as it can be continuously refined and improved based on feedback from all stakeholders. Ultimately, this leads to a more effective and efficient system that better serves its intended purpose. Using a well-defined modeling language, the formal knowledge model converts

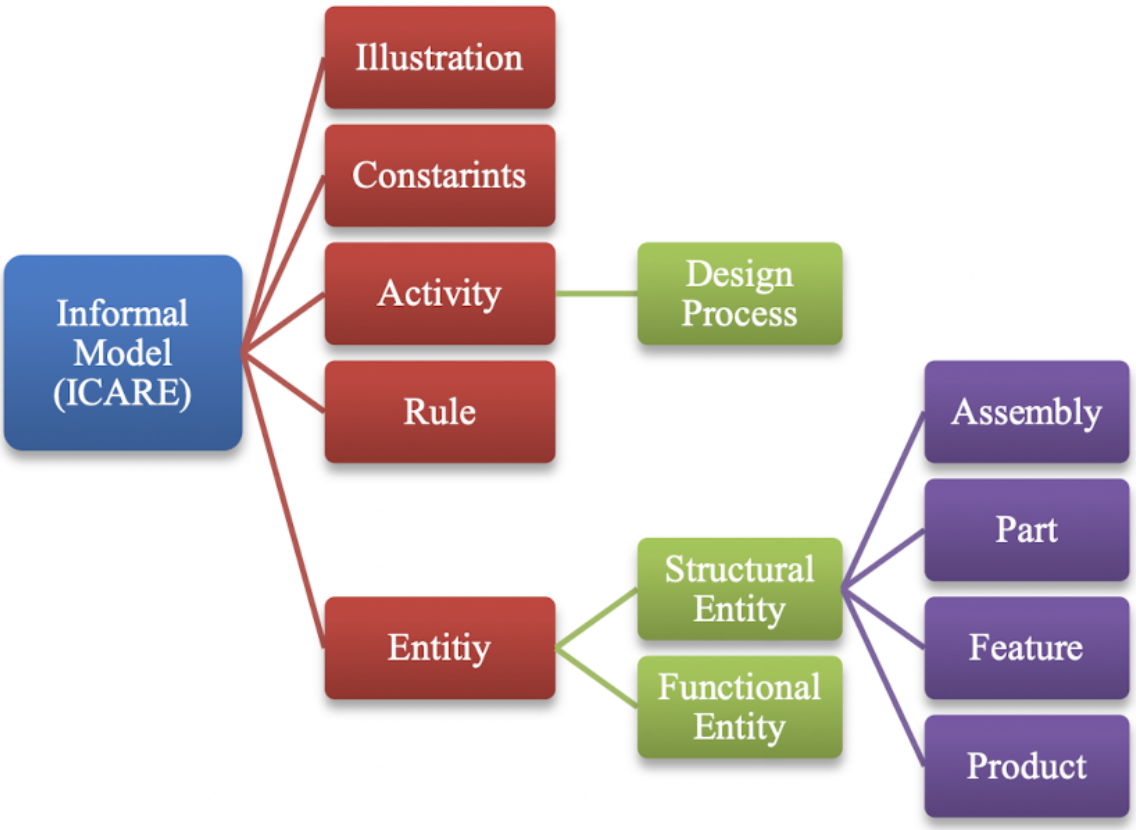

Figure 3.10: MOKA informal model [11]

the informal knowledge model into structured knowledge. To achieve this, the knowledge engineer performs a manual conversion of the ICARE forms into the MOKA Modeling Language (MML), a variation of the Unified Modeling Language (UML). The use of a modeling language, in particular UML, is crucial because it enables the standardization of knowledge that the software developer can interpret in a single way. The usage of a formal knowledge module, while very helpful to ensure the correct application development, is frequently unknown to the expert engineer, which has the disadvantage of keeping the domain expert out of the software development. This calls for informal knowledge to be well understood and for a common vision among all parties. Figure 3.11 represents the formal model included in MOKA's methodology, which can be seen as two separate models that communicate between themselves: the product model and the process model. As a means of es-

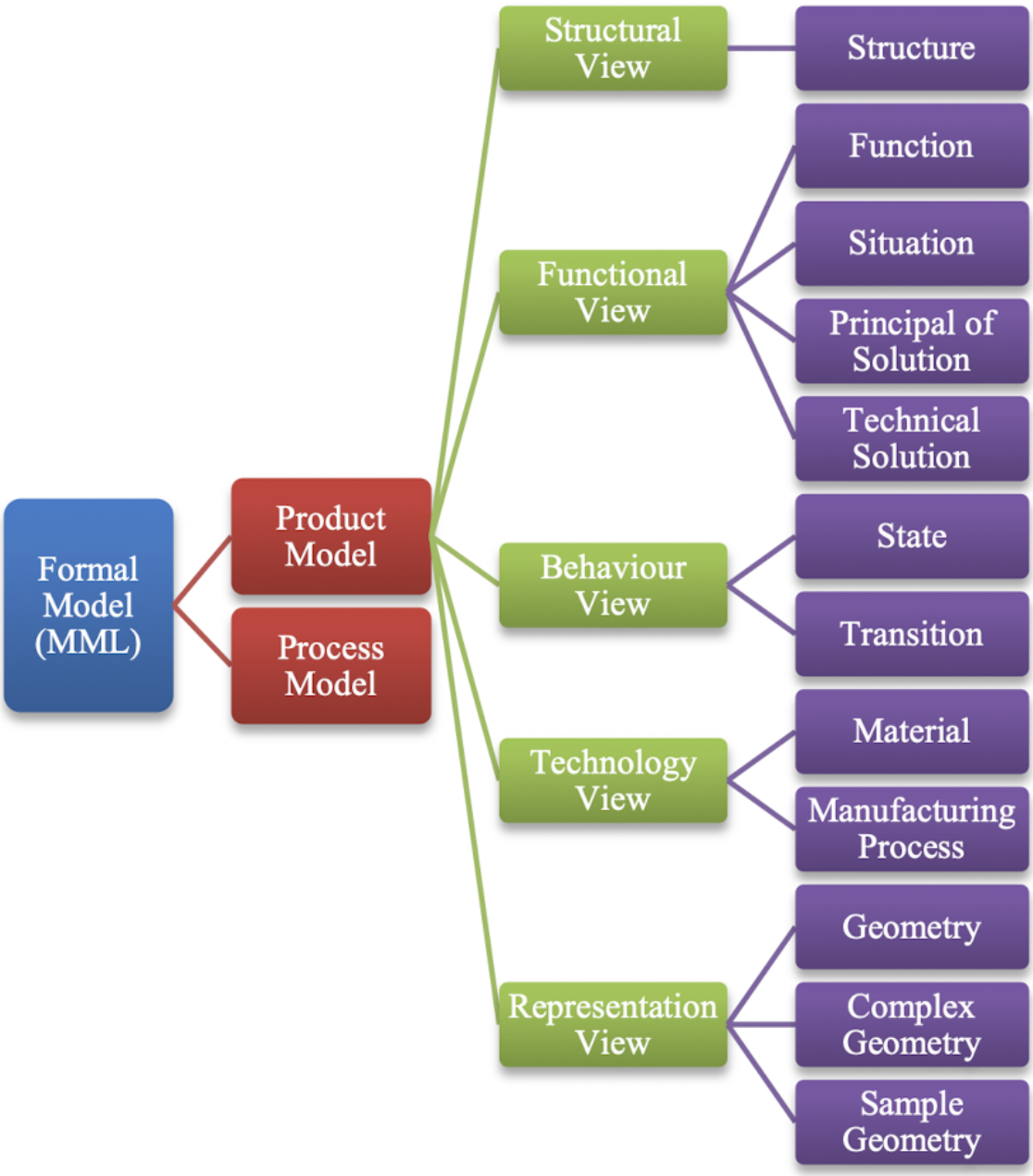

Figure 3.11: MOKA formal model [11]

tablishing a link between the software code and the knowledge models, a neutral language model is used. To do so, MOKA recommends the use of XML, or eXtensible Markup Language, to allow the knowledge model to be easily translated.
Despite being a very thorough and reliable methodology, MOKA also has some fallacies. The methodology, which focuses more on capturing and structuring knowledge through informal and formal models than on the end-user, is initially centered on the knowledge engineer. A lot of work is then needed to ensure that the end user is a part of the development process by iterating through the informal model as often as necessary and that the end user can have a great benefit from using the tool [13], which typically translates into reduced lead-time and the automation of tedious tasks. Additionally, because MOKA is unable to consider the origin of knowledge and the consequences of its transformation, it is unable to deal with knowledge change[35].
Various methodologies have been developed in recent years to address MOKA's weaknesses. With most of its principles derived from MOKA, KOMPRESSA (Knowledge-Oriented Methodology for the Planning and Rapid Engineering of Small-Scale Applications) was developed to support small and medium-sized enterprises (SMEs). It places a greater emphasis on risk analysis and management. The KNOMAD method (Knowledge Nurture for Optimal Multidisciplinary Analysis and Design) is another approach derived from MOKA, KOMPRESSA was developed to support small and medium-sized enterprises (SMEs). It places a greater emphasis on risk analysis and management.
By implementing six steps—knowledge formalization, normalization, organization, modeling, analysis, and delivery— KNOMAD attempts to address MOKA's flaws. It focuses on the design process [13] and enables the extension of multidisciplinary engineering knowledge to design and production, in contrast to MOKA, which is a product-oriented methodology.

3.2.5 KBE approaches

Following the literature study on KBE, it is possible to mainly identify five different approaches to KBE, each enhancing a few of KBE capabilities. Depending on the goal, these approaches can be used individually or together, highlighting the necessary features the tool in development shall have.

Parametric Modeling

One of the first applications of KBE was to support CAD systems by using parametrically modeled geometries. This method employs mathematical equations as the foundation for the rule-based decision-making process and provides an easy way to reuse existing design processes. As an example, consider the development of High-Level Primitives (HLP), in which CAD systems account for geometric libraries containing reusable geometries such as wing sections that can be changed and integrated based on the needs of the user. Nowadays, several CAD software programs implement parametric modeling to assist the end-user's design.
This approach has the advantage of providing a good representation of the product at the early stages of the design and allowing for design changes as well.

Function Based Modeling

This approach is used when the design process tends to gain complexity. Its goal is to leverage the potential of the digital model to connect design functions to the physical embodiments used to recognize the function [11]. This approach sees rules, objects, and constraints as the main forms of knowledge.

Case-Based KBE

With this method, new problems are attempted to be solved using the information from earlier, comparable problems. A database that houses use cases that can be referred to as "previous experiences" is necessary. It is constructed in four steps:

- Locating related cases
- Reusing prior knowledge to develop a solution for the new situation
- Updating the information with the new approach
- Keeping this revised response as a new case

Since there are numerous solutions to a problem, there is a significant amount of uncertainty to be dealt with, despite being very effective at finding new solutions.

Web-Based Approach

The web-based approach allows different stakeholders to work together in a collaborative environment, regardless of their physical location. Additionally, this method enables real-time communication and feedback, making it simpler to make changes and adjustments throughout the design process while considering the potential to include the client in the design process loop and improve the design process itself.
Moreover, web-based applications allow for real-time product visualization, which leads to better product understanding for all stakeholders. HTML can be used to add detail to the product through text, audio, video, and animations.
Although this is a great approach to use in collaborative scenarios, there needs to be an effort made to standardize the exchange formats because the knowledge that is available online is not standardized. Stakeholders can resolve this by deciding on the ideal format for exchanging each type of information.

Ontology Based KBE

With the aim of providing a machine-readable model that is not overly cryptic for humans, ontologies enable the formal representation of the knowledge of a specific domain in a structured and organized model made up of classes, properties, relationships, constraints, and rules. This kind of knowledge representation is particularly helpful in industries like engineering, finance, and medicine where there are intricate dependencies between various entities. Organizations can enhance their decision-making procedures, increase data integration and interoperability, and facilitate more effective knowledge sharing among various stakeholders by utilizing ontology based KBs.
By utilizing reasoning and inference processes for complex systems, this strategy enables the incorporation of additional knowledge. The use of a common language allows for KBE that is ontology-based to represent a specific domain.

3.3 KBE in the aerospace industry

KBE application attempts in the aerospace industry are not new. The need to work with a globalized supply chain, which constantly includes competitor suppliers, and the need to cope with mass personalization of its products are the main drivers for KBE development in OEMs. In such cases, the high costs are justified by the ability to create new product designs in a shorter amount of time, reducing reliance on a specific employee's knowledge.
This section aims to give an overview of the current trends in the aerospace industry regarding KBE applications, allowing the reader to have a broad outlook on the possibilities. In order to identify where KBE can be applicable throughout the design process, [6] proposes five simple criteria to evaluate the feasibility and potential of a KBE development:

- Is the process step a routine process step?
- Is each step of the process based on formalized rules?
- Is each step of the process complex, requiring iterations?
- Qualitatively, what is the lead time for this process step?
- What is the potential return on investment (ROI) for this KBE application?

With these five simple rules, it is already possible to identify potential candidates for KBE.

3.3.1 Examples of KBE applications

In the following paragraphs, a number of cases from the literature will be provided to help the reader better understand the potential applications of KBE to the aerospace industry. These will go over the main issue, the conventional solution, and how KBE can help engineers. These applications can be generally divided into 4 classes [32] (or a mix of them):

- Model generators: generate different product representations, or models.
- Structural analysis tools: focus on making the analysis process quicker by applying either analytical or numerical methods. These tools are usually used together with model generator tools.
- Manufacturability analysis tools: cost estimation and technical feasibility tools that evaluate the project's potential both from a business and technical point of view.
- Multi-disciplinary tools: combine several tools. If developed in different layers and modules, it can be seen as a framework to support KBE tools.

Production modeling of an aircraft fuselage

In [29] the researchers used a KBE application to model the production process of an aircraft fuselage both during manufacturing and in assembly. For the manufacturing study, researchers focused on the production process of stringers, while for the assembly study, the focus was on the fuselage assembly of the cabin. A semantic-web KBE approach was used with a methodology-guided approach to the project. The writers created the MAMOT (Manufacturing and Assembly Modeling Tool) application, consisting of a knowledge module, an assessment engine, and an inference engine.
The knowledge module accounts for all the ontologies and rules describing the process and are expressed using the Web Ontology Language (OWL) and Semantic Web Rule Language (SWRL). In this case, the ontologies are responsible for modeling the knowledge regarding the manufacturing and assembly processes, while the rules are used for the decision-making process.
The rules are executed using an inference engine, which interprets the semantic knowledge and infers new knowledge. This knowledge is what expert engineers have traditionally held, and their experience, in reports and in literature, on KBE is formalized by the knowledge engineer.
Using this tool, the researchers determined that repetitive tasks like creating new scenarios and refactoring the model to accommodate them could be automated, giving designers more creative time to explore the design space.

CAD-CAE model generators for turbofan engine vanes

During the iterative design process, data in various formats must be exchanged. CAD software is preferable for design engineers, but analysis engineers typically require pre-processing of the CAD model to generate useful features for CAE tools. This process is complex and time consuming, but with KBE, this task can be automated using a model generator. In [21], the researchers created the "CTX Vane Creator" to ease the handoff between CAD and CAE users in collaboration with AVIO Aereo in Italy. In this application, there is a master model containing a great deal of data, both geometric and non-geometric, which is used to create simpler models based on the user's needs. This is an interesting approach and goes towards a complete digitalization of the new product development (NPD) process, as it allows all the users to work on the same model, which works as a repository.
Before developing the KBE tool, engineers had to go through a laborious process of simplifying the CAD geometry, removing extraneous features, splitting complicated vane surfaces into smaller dimensions, which had to be done for each airfoil, subdividing the entire vane into smaller pieces, aligning the simpler forms, and converting it into a neutral format, such as Parasolid. This process was heavily reliant on the engineer's abilities and was prone to numerous errors due to inattention to the task. Furthermore, this activity had to be repeated for every iteration of the process.
With the introduction of the KBE application, the number of non-productive interactions between CAD and CAE engineers has decreased. This activity allowed a 93% reduction in the time required to prepare a model, with a mean time reduction of 95% on the process, allowing for streamlined knowledge sharing throughout the company. At the end of the research, a direct ROI of 35% was generated.
The authors also identified the main costs of KBE development to be associated with the development and testing of the tool and training engineers to use it.
As indirect benefits, the reduction of time spent on repetitive tasks allowed engineers to dedicate their time to optimizing the design, leading to a better product in the end.

Time metric	Before	After	Change [%]
ODT	30 minutes	2 minutes	-93
ATR	8.1 days	0.5 days	-93

Table 3.1: Complexity in time before and after the KBE tool introduction [21]

Ontology-based KBE framework

In [7], the authors tackle the problem of defining a general framework based on ontology KBE methods to support the creation of highly complex geometries using different primitives. In order to achieve a generic framework, the authors describe the necessity of developing a platform-independent system capable of separating the system itself from the application.
The three most important pillars of object-oriented programs are implemented in this framework:

- **Inheritance** allows child classes to be derived from a superclass and to inherit all their properties.
- **Encapsulation** prevents unauthorized access to information.
- **Polymorphism** consists of giving more than one form to a function, object, or attribute.

Furthermore, the use of declarative rules enhances the reusability and maintainability of the framework. This application was tested to work with a parametric tool to ease the creation of complex geometries using high-level primitives, and the framework was validated by nine different stakeholders, each with a different role in the project. As a result of this project, the interviewees perceived a high level of genericness (76.4%) and good cost reductions (36.3%) following the KBE tool development [7].
Finally, the researchers emphasize that developing a platform-independent framework necessitates more effort, both personally and financially, than platform-specific frameworks currently on the market. This is motivated by the need to train experts to work with APIs, which is required to ensure the framework's long-term usability.

Fiber Metal Laminates design process

In [10], the authors take advantage of rule-based design to study the layering process of laminates. In this study, three different business opportunities were identified at Stork Aerospace in the Netherlands. To demonstrate KBE's potential to aid the design process, the subdivision of prepreg plies was chosen as a use case, where a highly structured but repetitive process was identified, with production preparation time accounting for around 60% of the total design time, identifying this process as a potential business opportunity. Although the process of developing the tool itself is long, taking around 16 man-weeks, the authors concluded that this high price is paid after only six design iteration cycles. In fact, in the time taken to execute 10 design cycles traditionally, 100 can be done using KBE, with a 75% lead-time reduction. Furthermore, the recurring costs see a reduction after eight design cycles. Figure 3.12 illustrates the results obtained for both the traditional and the KBE approaches in terms of design cycles and time.
The researchers also identified that the rule-based approach allowed for a reduction in waste on 90° prepreg plies, leading to an overall reduction in cost.

Automatic wire routing

With about 530 km of wiring with more than 100,000 wires and 40,000 connectors to drive 1150 electrical components in an A380, wire routing is a very demanding task for aircraft constructors. The placement of these elements, as with any type of line in a complex machine, is subject to many regulations and design rules. In the traditional way, engineers must manually study the routing for each of the thousands of wires present. This process is highly repetitive and mutable as the design advances, leading to a high amount of time spent on this task. In [34], the authors studied the usage of KBE applications to aid engineers in planning wire routing through the aircraft. By using path optimization algorithms together with a discrete FE model of the aircraft's fuselage, it is possible to define a KBE application fed by a set of complex design rules that look for the best optimal placement

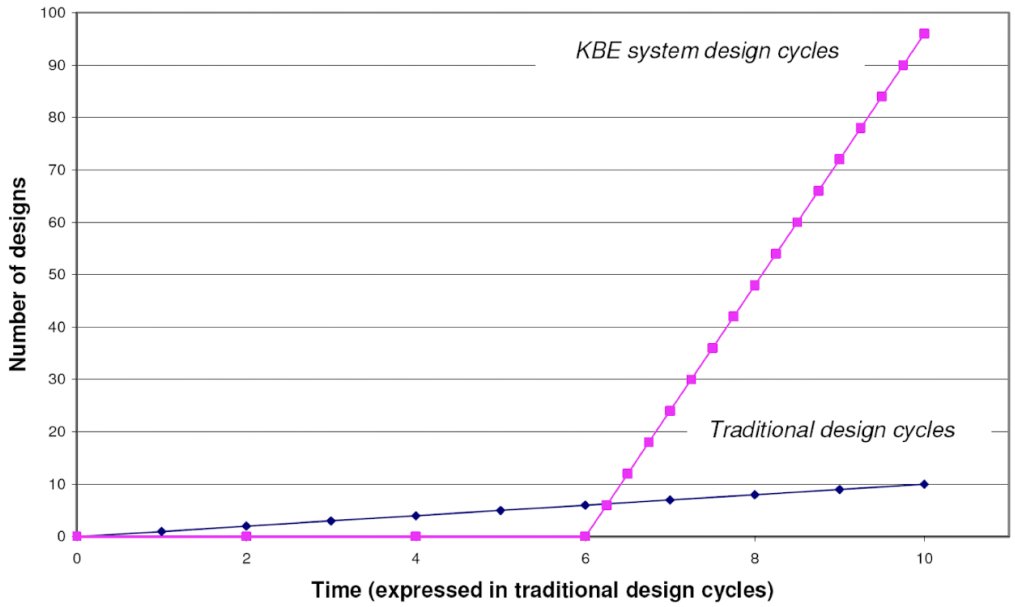

Figure 3.12: Design cycle for traditional and KBE approaches [10]

of cables in order to minimize cable lengths and thus mass.
The authors discuss the necessity of avoiding "black box" applications, where engineers do not understand what the code is doing. In fact, by using declarative rules that can be accessed by engineers, it is possible to understand the choices the rule-based algorithm takes when defining the optimum path, gaining the trust of designers. As a result, a significant time reduction in the process was found, taking only 10 to 15 minutes to analyze more than 3 million nodes to route the cables. Furthermore, the system is capable of coping with design changes, such as the addition of prohibited zones or extra components.

KBE to support innovative design

Despite the fact that the majority of KBE research focuses on automation for mass production, KBE also sees its potential applied to support innovative design. In [12] and [30], the authors developed a KBE tool in order to aid designers in dealing with alternative aircraft designs. The tools could apply parametric rules to support the CAD phase by using a series of high-level primitives common to any vehicle. As an example, the authors mention the possibility of declaring a primitive responsible for lift. The authors argued in favor of using high level primitives because engineers frequently view products as a set of functional requirements that must be met rather than as points, curves, etc. This kind of primitive is common to any flying vehicle, both fixed wing and non-fixed wing. The tool produced good results when modeling a blended wing body aircraft.
In addition, the researchers discuss a generative model approach to the problem, resulting in a multidisciplinary framework that makes it possible to execute several tasks based on a single master model. This guarantees that every stakeholder works on an updated version of the project by using separate relational models derived from a unique definition for the product. Figure 3.13 illustrates the generative model method, which was validated throughout the project with different research and industry partners. Another finding from this study is that modeling the HLP in scripted form, as opposed to the conventional CAD method, increases the robustness of the geometry generation process, enabling the system to correctly manipulate widely known errors by using the declarative rules method. The developed application was able to execute a MDO study with more than 50 aircraft variants over a weekend, completely autonomously, using high fidelity tools, reducing the length of the design process from months to days.

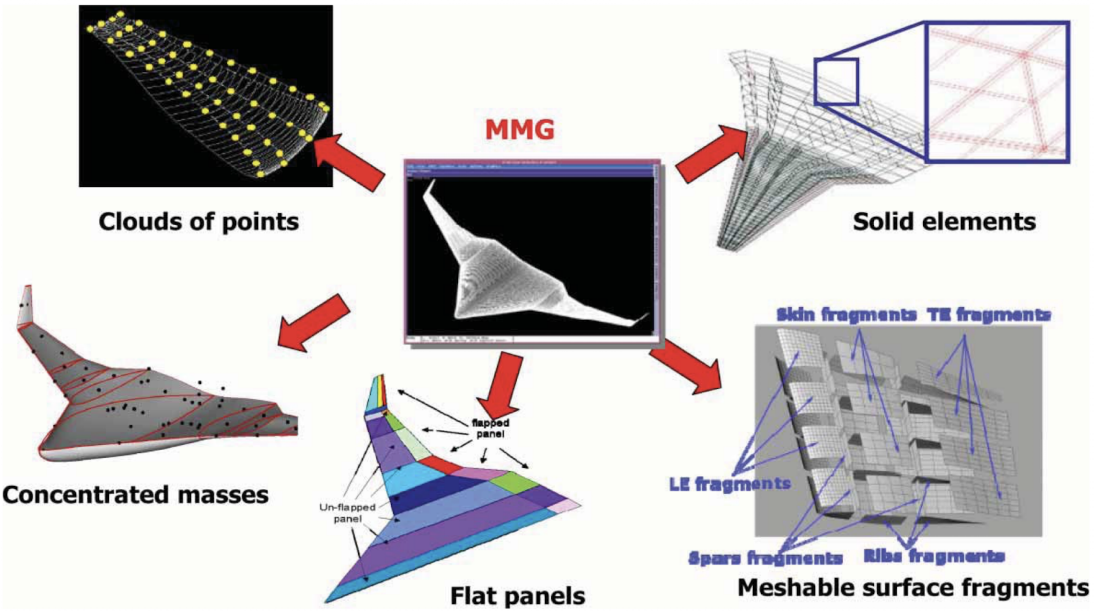

Figure 3.13: Generative model method [12]

3.4 The KBE rationale

One of the biggest justifications for adopting KBE methodologies is due to the high costs of the entire product lifecycle that are committed at the end of the early design phase (around 70–80%). It would be best if it were possible to better explore the design space at an early stage before making decisions that highly impact costs. Currently, the knowledge available to a designer at an early stage is very limited, which leads to design choices that later must be rethought. This is a very time-consuming activity that is mostly linked to routine tasks, as already mentioned. Therefore, there is a need for tools and methods that can support designers in making informed decisions at an early stage. These tools should be able to provide accurate cost estimations and help designers explore different design options by adding detailed knowledge to the model based on previous experience.
According to Baxter et al. [37], almost 20% of the designer's time is spent looking for a solution and absorbing information. Furthermore, it is estimated that over 40% of the information required to manage design requirements is currently extracted from personal knowledge, implying that knowledge is not represented in an easy and sharable way. This highlights the need for a more efficient and organized system for managing design requirements. Implementing a knowledge management system could significantly reduce the time spent searching for solutions and increase collaboration among designers.
In Figure 3.14, the product life-cycle costs are represented. It is interesting to observe the inverse proportionality between design freedom and available knowledge. This is one of the things KBE tries to solve by allowing designers to have more knowledge of the product at an early stage, allowing for design decisions that reflect on the whole product life cycle, which eventually yields to a reduction in the committed costs at an early stage.

3.4.1 Benefits of using KBE

Improved maintainability

Due to the high costs and lengthy development times of good KBE frameworks, businesses often only apply a few of the KBE principles to their tools. However, by compartmentalizing the framework into different modules, the advantages of KBE in terms of long-term maintenance of the tool have been demonstrated. Numerous academic and industrial researchers have identified this advantage, which has sparked a shift toward a comprehensive KBE process, particularly from OEMs that have more resources.

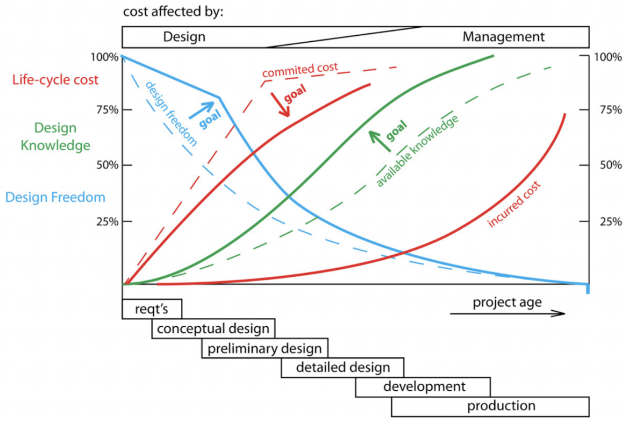

Figure 3.14: Project time vs product costs, available knowledge and freedom of design [13]

Shortening development time

One of the primary benefits of using KBE is that it allows businesses to save valuable resources while also accelerating time-to-market. Companies that can adapt faster than their competitors and provide new products and services to their customers will thrive in today's globalized and dynamic world. Additionally, design engineers can use the time they save to increase the caliber of their products and respond more quickly to design modifications imposed by other departments [38]. It is estimated that by implementing KBE strategies, costs and time can be reduced by 92% in this phase [11].

Extending the available knowledge in the early design phases

KBE enables design engineers to have a more in-depth view of the problem during the early design phases by considering previous experiences on similar projects (knowledge reusability). Furthermore, it enables the incorporation of various disciplines into the project design, some of which would be impossible to incorporate at an early stage without prior knowledge, such as maintenance and detailed cost predictions.

Better exploration of the design space

Designers can concentrate on the creative aspects of project development by using KBE. Furthermore, by enabling the evaluation of more what-if scenarios, KBE promotes NPD innovation by lowering the cost of taking risks while exploring the design space. This allows for the creation of more innovative and unique products that can better meet the needs of consumers while being compliant with multiple design requirements. Additionally, KBE can also improve the efficiency of the NPD process by automating certain design tasks and reducing errors, ultimately leading to faster time-to-market and increased profitability.

Knowledge for all

Knowledge sharing and dissemination within a company are crucial for its success. Junior engineers' performance in the workplace is impacted because they are unable to reach their full potential because of their lack of experience when it comes to putting forth new ideas [39]. KBE allows you to store different types of knowledge to execute tasks. This allows not only for better knowledge management of explicit knowledge but, most importantly, for the storage of tacit knowledge. This tacit knowledge is what most junior engineers lack. With KBE, junior engineers can contribute significantly to the design process by better exploring the design space and the company's resources.

Keeping the knowledge inside the company

Companies rely on the collective talent and expertise of their employees and must frequently deal with turnover, which leads to knowledge leaving the company in traditional project development. By storing expert knowledge in KBE, companies gain an extra layer of resilience when there are personnel changes or when employees retire. This not only ensures that the company's knowledge

base is preserved, but it also allows for faster and more efficient onboarding of new employees [40], as they can quickly access the information they need to perform their job functions.

Separation of business and functionalities

By separating the functional module, where the tools to run the application are built, and the business model, which contains the necessary knowledge to run it, competitors can collaborate on product development without having to share confidential information. This strategy also allows for ad hoc knowledge expansion based on one's needs. This is especially important in the aerospace industry, where a few OEMs compete for the large commercial aviation business (Boeing and Airbus). Because their customers are located all over the world, these businesses must award contracts to subcontractors in various countries [21].

Consistent, coherent, and synchronized product knowledge

By adopting web technologies for KBE applications, it is possible for teams across the world to collaborate in real time on the same product without the need for constant file exchanges that affect traditional design strategies [12]. This not only saves time and resources but also ensures that all team members have access to the most up-to-date product information. Additionally, it allows for a more efficient and streamlined communication process between businesses and subcontractors in different countries. Furthermore, changes made to the product can spread through the various models by incorporating dependency backtracking, enabling up-to-date analysis, and reducing the risk of errors or delays in production. This ultimately leads to a more cost-effective and timely delivery of products to customers, enhancing overall customer satisfaction and loyalty.

3.4.2 Challenges and limitations of KBE

Although KBE has several intriguing, empirically supported business benefits, there are still barriers to its widespread adoption. One of the major challenges is the massive amount of investment required to establish a business-wide KBE framework, which keeps it as a technology used by large companies as OEMs and in a limited number of industries, primarily aerospace, automotive, and medical. Since there is no industry-accepted neutral-language standard model and no mechanisms to translate formal knowledge models into neutral language model schemas [9], one of the many causes of this significant investment (both in terms of time and money) is that the application development is complicated and entirely manual.
Another barrier is the lack of skilled personnel who can effectively manage and utilize KBE systems, which makes it difficult for small and medium-sized enterprises to adopt this technology. However, as the benefits of KBE become more widely recognized, efforts are being made to address these challenges and make it more accessible to a wider range of industries and businesses.
A second important consideration is professional resistance to using the tool. In [40] the authors discuss how it is a common practice in the aerospace industry to reward professionals based on their expertise. This approach results in engineers leveraging their knowledge to ensure job security, access to new and exciting projects, and the establishment of value-added services for the company. In such cases, the process of knowledge capture may be perceived negatively by employees as a threat to job security. To address this, corporate values must be shifted to encourage knowledge sharing across the organization and reward knowledge reuse in order to maximize the use of the (expensive) KBE tools developed. Additionally, it is important for management to communicate the benefits of knowledge sharing and reuse to employees, such as increased efficiency and innovation. Training programs can also be implemented to ensure that employees are equipped with the necessary skills to effectively use the KBE tools and contribute to the knowledge sharing culture.
Furthermore, in order to gain the trust of the end-user, "black box" behavior must be avoided. This is one of the most difficult aspects of developing an application because it occurs when designers are unable to understand what the system is doing and how it is reasoning due to a lack of transparency in the design rules. In fact, it is critical to include the end-user in the KBE tool's design loop and provide him or her with a clear understanding of the rules that are being implemented. This can be done in many ways. For example, when developing a KBE framework, it is necessary to give the user the possibility to add new knowledge and modify it if necessary. Code annotation can be a useful tool for managing knowledge and should be encouraged. In addition to that, exploiting knowledge graphs

through the use of semantics can be a powerful tool when dealing with heterogeneous and complex systems. This can help to better understand the relationships between different pieces of information and improve decision-making processes. By leveraging these technologies, organizations can gain a competitive advantage by effectively managing and utilizing their knowledge resources.
As one of the technical factors that present a challenge to KBE, the lack of skilled staff with great domain technical experience is a limiting factor to its implementation in many companies. Furthermore, the software development itself is a major limitation. Different studies have identified how the current KBE applications have limited scope and, more often than not, are hard-coded. Due to its quicker development and lower implementation costs, this method is frequently employed in business [33], but its limited long-term reusability can result in knowledge loss, abuse, and underutilization [13]. A declarative rule approach, kept separate from the system's core, is preferred because it allows knowledge to be added and amended without having to rebuild the entire tool [34].
Technical factors such as the salary of a software developer may be a constraint in countries where these figures are known to earn significantly more than design engineers. In such cases, the time saved by an engineer by using the application may not be worth the financial resources spent on a software developer to create the tool. The expensive software licenses and programming languages designed primarily for seasoned developers and not the average user act as an additional barrier to KBE's widespread adoption [6].
A constant challenge when developing KBE systems is the difficulty of using them to foster innovation. This problem was addressed in [12] and [30], and it is demonstrated how this problem could be tackled by using HLP for computer-aided design activities. Although researchers concur that KBE may not be the best methodology if the design process is largely dependent on the creative process and is highly variable [13], making it difficult to standardize processes, they also acknowledge that this situation is rare in the aerospace industry.
The fear of being locked into proprietary software was also noted as a deterrent to using KBE applications. A few solutions for how to handle this fear include the development of a generalized modular framework that can integrate various tools through APIs to build a dependable system that can withstand the test of time and the challenges posed by innovation.
Finally, the business evaluation of a KBE tool can be difficult. The high risk associated with the large amount of investment required to develop a KBE application is frequently used to justify the cancellation of a project that could produce meaningful results for the company. Both industry and academia recognize the critical importance of developing concrete and robust methodologies to evaluate the feasibility of a KBE tool in order to encourage companies to develop them. This is also regarded as one of the major bottlenecks currently preventing its widespread use. Furthermore, the lack of standardized evaluation methods also hinders the comparison and benchmarking of different KBE tools, making it difficult for companies to make informed decisions on which tool to adopt. Therefore, it is crucial for industry and academia to collaborate in establishing a set of reliable evaluation criteria that can be universally applied to KBE tools.

3.5 The Codex framework

Codex is a knowledge-based engineering framework developed at the German Aerospace Center's Institute of System Architectures in Aeronautics in Hamburg, Germany. Its primary goal is to harness the power of semantic web technologies in order to build a general, robust, and multidisciplinary framework capable of knowledge capture, formalization and execution.
Differently from most KBE solutions available, Codex provide a non-frame based, or schema-less, environment capable of knowledge capturing, formalization and execution [14]. By leveraging Semantic Web Technologies (SWT), the knowledge is modelled through statements that are added to the model. These statements are the most basic unit of the knowledge and are made by 3 elements, consisting in a triple; `subject`, `predicate` and `object` [41].
Subjects are always made by resources (or individuals) and, as in standard grammar, they represent what (or who) the statement is about [42]. As for subjects, resources can also take the place of an object in a statement. In such cases, statements are said to link two resources or individuals (e.g. "Mary - is the mother of - Peter). In addition to that, objects can also assume the form of literals, such as strings and values, leading to statements that link subjects to different attributes or characteristics. An example of a statement describing a characteristic could be "Mary - was born in - 1980", where the year 1980 is a singular information, meaning no more knowledge is linked to it.
Due to its genericness, it is important to point out that Semantic Web Technologies allow multiple

ways to model the same problem. For example, if the year of 1980 had additional information attached to it, such as a list of people who were born in the same year, one could assume 1980 to be a resource to which different individuals (the people born in 1980) would be linked. By using a different property, such as `isYear` linking the resource `1980` to the value "1980", the same properties that were previously described become again available. It is easy to observe how the complexity of modelling increases significantly as new information is added to the problem, thus increasing its own complexity.
Although SWT allow the integration of multi-domain knowledge effortlessly, its abstraction and lack of a fixed hierarchy capabilities are not well suited for the modelling of complex engineering problems [14]. Due to the high specificity of engineering concepts, their modelling may be subject to the definition of several statements, thus leading to a very long modelling time [14]. To cope with that, Codex allows the creation of Domain Specific Languages (DSL), which grants the creation of simplified syntax that are specific to a certain domain and may be better understood by the domain experts, providing modelling simplification. An example of DSL developed for Codex is the **Geometry Syntax**, which provides tools and methods to deal with geometric modelling. By using a DSL, developers reduces the workload left to the designer, while enabling complex problem modelling with specific domain terminologies. In Listing 3.1 an example of a geometry problem modelled using the Geometry Syntax is illustrated, while in Figure 3.15 the resulting knowledge is shown. It is easy to observe the high amount of knowledge created by the much simpler functions `createPoint` and `createEllipse` available to the users.

Listing 3.1: Usage example of the Geometry Syntax [14]

```
// Create a point and ellipse using geometry syntax
using(GeometrySyntax) {
point = createPoint(0.0[m], 0.0[m], 0.0[m])
createEllipse(point, 4.0[ft], 1.0[m], AxisZ  AxisY)
}
```

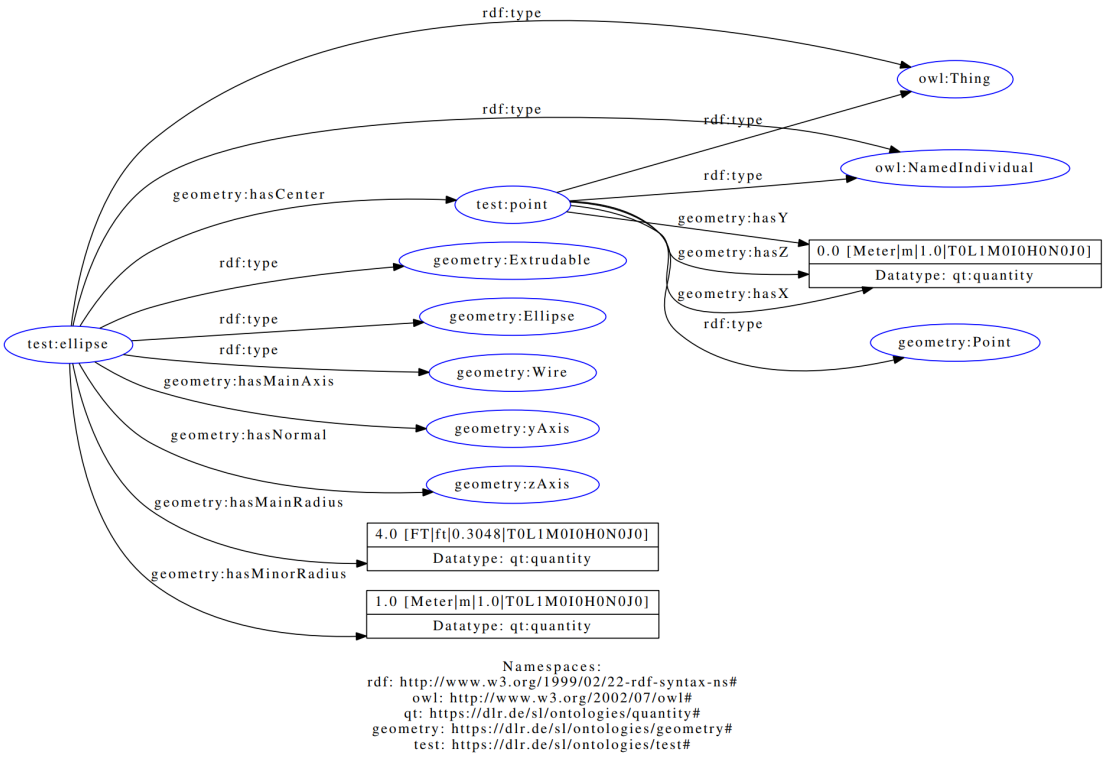

Figure 3.15: Semantic graph obtained from the geometry syntax in Listing 3.1 [14]

The framework, which makes use of plugins and APIs, enables experts from various domains to contribute their own ontologies to the problem while maintaining well-established standards for knowledge representation [15]. Domain experts can connect various tools, languages, and technologies to the

plugin architecture. The freedom Codex provides users in naming and modeling their ontologies is an important feature. Equal knowledge modeled with different syntax by different domains can be linked together using inference engines and semantic rules, yielding a univocal representation of the product and allowing knowledge multi-domain integration, as illustrated in Figure 3.16.

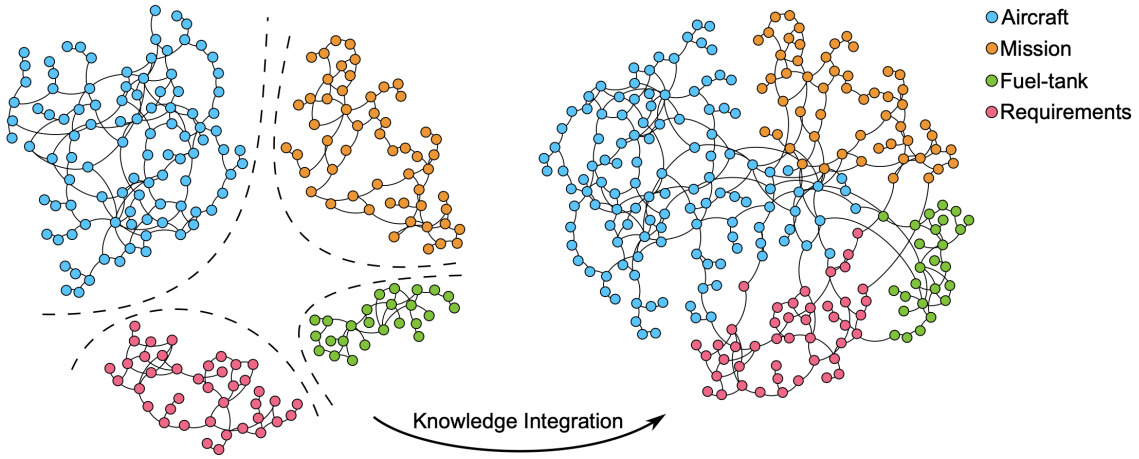

Figure 3.16: Multi-domain integration using ontologies [15]

Codex excels in another area as well: knowledge representation. The limitations of Object-Oriented Programming (OOP) regarding class hierarchies are not applicable to Codex because of its schema-free architecture. This makes it possible to more effectively reuse knowledge as everyone is entitled to access it, not just its "children" elements. In the Codex framework, there are no levels of hierarchy and all knowledge is treated at the same level, making class concepts and hierarchy ineffective. Codex uses its graph-based modeling strategy to simplify the model and enable a single, easily accessible representation of an entity, whereas traditional, OOP modeling would necessitate the repetition of information whenever it is outside the scope of a predefined hierarchy.
Besides DSL and inference capabilities, Codex makes use of different rules to extend the knowledge available in a model. The declarative characteristic of these rules allows the users to focus their attention on the implementation of the problem itself and not on its execution order. In fact, developers are responsible for defining the best way to implement the rules and to increase the framework's performance. Still, experience shows that there are best ways to declare a rule itself, in order to reduce the reasoning time behind this process.
Codex provides two categories of rules: parametric rules and production rules. Parametric rules focus on equations and describe relationships between values characterizing the different knowledge nodes. Production rules are used to add new knowledge or to connect existing entities in a model [14]. These rules are often referred to as topological rules as they change the knowledge graph on a topological level and are the main type of rule of interest for this project.
To implement the rules, Codex uses different engines to support knowledge inference:

- semantic engine: used to infer knowledge linked through statements made in different ontologies
- parametric engine: used to solve parametrically defined entities, by adding numerical entities to the knowledge
- production engine: execute the production rules and change the graph's topology with both creation and removal capabilities of entities.

Among the different modules currently under development for codex, codex-geometry allows to exploit semantics to tackle geometry modelling. By using Open CASCADE Technologies (OCCT) kernel [43], this module provides the Geometry Syntax DSL that allows complex geometry creation effortlessly. In addition to that, it leverages the production engines to develop production rules able to translate the domain-specific objects into 3D geometric shapes [20]. These objects can then be exported into different geometry representation extensions (e.g. BREP, IGS, STEP, etc.) and can be visualized in

TiGL Viewer [44].
Geometry management is an area of great interest for engineering applications and has been identified as a promising feature of Codex in need of further development. The ability to easily create complex geometry and to leverage existing knowledge and rules to better explore the design space at an early design stage could be better exploited if verification strategies were put into place with the goal of providing a safer environment where designers can focus on optimizing a product by studying the complete design space, while still being supported and advised about unfortunate design choices that do not respect geometric rules (e.g. an overlapping between geometries) or certification rules and guidelines. The creation of verification rules to aid the geometric modelling of a system is the main goal of this thesis. In particular, the development of geometric and certification checks capable of supporting the fuel system design for conventional aircraft is discussed in the next chapters.

Chapter 4

Methodology

This chapter presents to the reader the methodology developed to create the geometry validation tool, hereafter referred to as GeoVerification tool or geometry verification tool, for the fuel system design of a modern commercial aircraft. Although this methodology was developed with the aim of enabling component placement for this particular application, it can be expanded and adapted to any kind of geometry verification tool.
In this project, two main categories of analysis were identified:

- Geometric analysis: analysis where the verification algorithms focus purely on geometric aspects of the product, highlighting solutions that are physically impossible to exist in order to fulfill their function. Examples of this category would be overlapping components, missing connections in a single geometry, etc.
- Certification and guidelines analysis: takes into account how the geometry of a component may be affected by certification requirements or design guidelines. Examples of such analysis could be the effect of a main rotor burst to the fuel system or the relative position of components inside a tank

In addition to the analysis, in order to better support component placement, a series of production rules aimed at the geometry creation of fuel system components such as pumps and valves were developed as a part of this project.
One of the key benefits of KBE systems is their capacity to store both explicit and tacit knowledge, as covered in Chapter 3. When it comes to certification requirements for the aerospace industry, certification bodies like the FAA and EASA hardly ever give original equipment manufacturers (OEMs) specific metrics or ways to meet a certain airworthiness requirement, leaving it up to the companies to show the certification can be met. As a result, engineers rarely make decisions based on explicit knowledge that guides the design process in order to verify the requirements. Instead, businesses must develop unique methodologies based on experience in order fulfill the requirements, which becomes tacit knowledge.

4.1 Knowledge capturing

As for any KBE methodology, the first step on the development of the geometry verification tool is the knowledge capturing (see Section 3.2.2). This process allows to gather as much information as possible about the user's needs, in order to choose which features to implement. In addition, it gathers technical information concerning the process behind formal requirements definition and validation.
An integrated approach that combined interviews and literature research was used to better comprehend the need for geometry verification tools for the fuel system. For the literature studies, SAE Recommended Practices [1] providing guidelines for the fuel system design and the certification specifications provided by EASA's CS-25 normative [24] were used as a support to understand the most typical issues when developing such systems. From those, a list of possible verification features, both geometric and certification related, was issued and then confronted with user's needs gathered within DLR's Institute for System Architectures in Aeronautics. After multiple discussion sessions held as weekly meetings, a list of possible features to implement filtered by priority level was produced. The

priority levels each feature received were based on the implementation feasibility, considering the time and knowledge available at the moment, and subjective feedback from an aircraft design point of view given by designers from the institute. The features that have been implemented are collected in Tables 4.1 and 4.2. Each of these features are covered in detail in Chapter 5 and are here divided by their primary objective: geometry or certification verification.
In addition to issuing a list of potential features to implement - or "checks" - based on certification and guidelines, the knowledge capturing phase was used to collect knowledge from an aircraft designer's experience that could be added to the tool. The information gathered in the form of informal knowledge through written text, schetches, and commented worksheets needed to be formalized in order to be implemented through code.

Function	**Description**
`checkConnectedToForSupplyLines`	Checks the correctness of the connection between two supply lines
`checkTankVolume`	Checks if the create tank volume has the specified volume
`checkContainedInForConnectionElements`	Checks if the connection element* is contained in the specified design space
`checkConnectedToForPortsAndLines`	Checks the connection's geometry between lines and the connection elements through the connection ports
`checkAttachedToBottomForBoostPumps`	Checks if pumps are correctly attached to the bottom of a tank
`checkAlignmentForConnectionPortsAndLines`	Checks for disalignment errors between connection ports and lines
`checkLinePortMatchingSize`	Checks if ports and lines have the same nominal diameter
`checkIntersectionPumpsRibs`	Checks if pumps overlap with the wing's structure (ribs)
`checkTankCutGeometry`	Checks whether a tank opening (cut) is properly represented on a tank's surface
`checkPipeTankIntersection`	Checks for errors in the design where there are overlaps between pipes and fuel tanks
`checkLongitudinalCoGTank`	Checks if the geometric barycenter of a tank is within its specified range
`checkLongitudinalCoGFuelSubsystem`	Checks if the geometric barycenter of the fuel system (fuel storage components only) is within its specified range
`checkMinimumDistancePipesTanks`	Checks if a pipe respects the minimum distance with all external surfaces of a tank
`checkMinimumDistancePipesTanksSurface`	Checks if a pipe respects the minimum distance with a specific surface (side) of a tank

Table 4.1: Geometry related checks implemented as part of the geometric verification tool
* Connection elements is a category of components comprehending fuel pumps and fuel valves

Function	Description
`checkFuelMassSystem`	Checks the maximum useful mass possible considering the complete fuel storage space
`checkFuelMassTank`	Checks the maximum useful mass for a specific fuel tank
`checkPumpIndividuality`	Checks if the pumps contained in a tank can be considered distinct individuals based on the Open World Assumption \cite{}
`checkPumpRedundancy`	Checks if the pumps meet the minimum quantity required
`checkValveIndividuality`	Checks if the valves contained in a tank can be considered distinct individuals based on the Open World Assumption
`checkValveRedundancy`	Checks if the valves meet the minimum quantity required
`checkSOVBeforeEngineAdapter`	Checks if a shutoff valve is present before the engine adapter
`checkRelativePositionSOVLines`	Checks the valves positioning to avoid water accumulation
`checkHighEneergyRotorBurst`	Checks for possible overlaps between a high energy (small fragment) rotor burst with predefined critical components in the system and computes the probability of direct hit
`checkMediumEnergyRotorBurst`	Checks for possible overlaps between a intermediate energy (medium fragment) rotor burst with predefined critical components in the system and computes the probability of direct hit
`checkFanRotorBurst`	Checks for possible overlaps between a fan burst with predefined critical components in the system and computes the probability of direct hit

Table 4.2: Certification and guideline related checks implemented as part of the geometric verification tool

Selection of production rules for component creation

Another need identified by the knowledge capturing phase was the provision of more detailed geometric information on the fuel system. Different fuel system architectures for commercial aircraft were analyzed and 4 main categories of components were identified to be required in order to have a good level of detail at an early design stage:

- Fuel tanks: in order to geometrically represent the fuel system, it was identified the need to distinguish the different fuel tanks and to represent them individually
- Fuel pumps: pumps are the most important category of components in large aircraft fuel systems as they allow the fuel to flow and to feed the engine.
- Valves: these components allow to isolate parts of the fuel system and may have stringent requirements regarding their positioning and quantity.
- Pipes: represent the basic entity responsible for transporting the fuel and form the basis of the different subsystems (feed, transfer, crossfeed, etc.)

These four categories led the development of production rules capable of geometrically representing them. For the fuel tanks, it was identified the need to be able to extract their design spaces from already existent aircraft configuration developed by DLR. These configuration are often stored in a

schema-based description file, commonly known as CPACS (Common Parametric Aircraft Configuration Schema) [45], and a novel method capable of extracting structural definition by means of geometry modelling was required.
Fuel pumps were identified of being of 3 main types: bottom mounted boost pumps, spar-mounted snorkel pumps and jet pumps. For the scope of this work, only motor-driven pumps of type boost and snorkel were considered. Moreover, in order to simplify the development rules and the verification functions to be implemented, valves were assumed to be shutoff valves and to have the same geometry, which is dependent on the connecting pipes' diameter. The development of such production rules is discussed in Section 5.2. Production rules for fuel line creation were already developed as part of previous work related to this project and thus won't be discussed. A detailed description of such production rule can be found in [20].

4.2 Knowledge formalization

Knowledge formalization is the second step in developing KBE applications. This activity is required to convert written or spoken language into code-ready language. Based on the formalized knowledge, multiple geometry management algorithms were developed for the implementation of the geometry verification tool. This process took place directly in the code development phase to reduce development time.
Formalized knowledge is necessary in order to have a single view of the requirements, leading to a good software development that leaves no space for biases in its interpretation. Geometric requirements, which are often expressed using verbal language, need to be quantified in order to be implemented in code. As an example, a requirement stating *"Req: Fuel pumps shall be contained in a single design space"* needs multiple information to be quantified properly: what is the design space? What geometric property will be used to check its containment? What tolerance shall be given to the result? These are a few of the questions the knowledge engineer must pose in order to build the verification function and avoid errors.
As to not overlap the information in this thesis, the implementation process is described in a separate chapter. Nonetheless, the reader should bear in mind it is a major step on the methodology itself and its planning shall be done ahead of its execution.

4.3 Validation

In order to ensure the correct functioning of a tool, software requires extensive testing before its deployment. Unit testing methodologies were applied in order to promptly validate the implemented rules. This methodology is based on three basic attributes [46]:

1. Unit tests verify small portions of a code called units
2. Unit tests are executed quickly
3. Unit tests are done in an isolated manner

This approach involved testing each geometry production rule and geometry validation tool individually to ensure that they behaved in accordance with the necessary specifications. Although the applied strategy is based on unit testing principles, it is important to note that some of the meaning of unit testing is lost because of the features of the Codex framework and the logic of the various reasoning engines, which are outside of the scope of this work. In actuality, a reasoning must be run in order to test the verification rules, putting the test under the correct operation of these engines since they cannot be run independently of the reasoning. Nevertheless, whenever it was felt necessary, standalone functions were created to test the logic behind the rules before incorporating them into the Codex framework.
Figure 4.1 shows the geometric output for the implementation of the rule `checkConnectedToForConnectionPortsAndLines`, which provides indication to the users on whether the fuel pump's connection port is aligned with the connecting fuel line. In this example, it is possible to see on the right picture the disalignment between both elements. Additional information on the disalignment angle is calculated and provided to the user by means of this production rule.
The geometries for the pipe and the pump must be constructed according to production rules using the Geometry Syntax offered by the codex-geometry module in order to carry out the test. The rule

uses the recently acquired information to examine whether the verification rule that has been put in place is operating as intended. The test's execution on the verification rule is straightforward, but in order to create the geometries and acquire the necessary information, it is necessary to run the reasoning. As a result, the test is not solely reliant on the implemented feature it is verifying. Real standalone unit tests cannot currently be deployed using a strategy that is entirely independent of reasoning engines because doing so would require a lot of time-consuming manual knowledge entry into each test. Still, these private dependencies [46] are generated in each individual test and do not affect different tests, thus guaranteeing the tests' mutual isolation. Listing 4.1 shows the statements that must be added to the test in order to trigger the rule in description. Although the `pump` individual was directly created through the user to the model, until now no `connectionPort1` individual was mentioned in the description. In fact, this resource only begins to exist when the production rule for the geometric creation of the pump is executed, showing the true potential of these topological rules, capable of adding useful information to the model. As explained in Section 3.5, assertions are given in the form of triples `<subject, predicate, object>`. In the example given in Listing 4.1, both subject and object are individuals, while the properties are of type `ObjectProperty`, linking the two individuals in the knowledge graph. The modules that derive these properties are expressed as prefixes in the code (i.e. `FS` for the Fuel System and `Ver` for the Verification module).

Listing 4.1: Example of adding verification requirements to a model through statements

```
// add the requirements to the model through statements
assertedModel.addAssertion(pump, Ver.connectedToRequired, feedLine)
assertedModel.addAssertion(feedLine, FS.isConnectedToPort, connectionPort1)
```

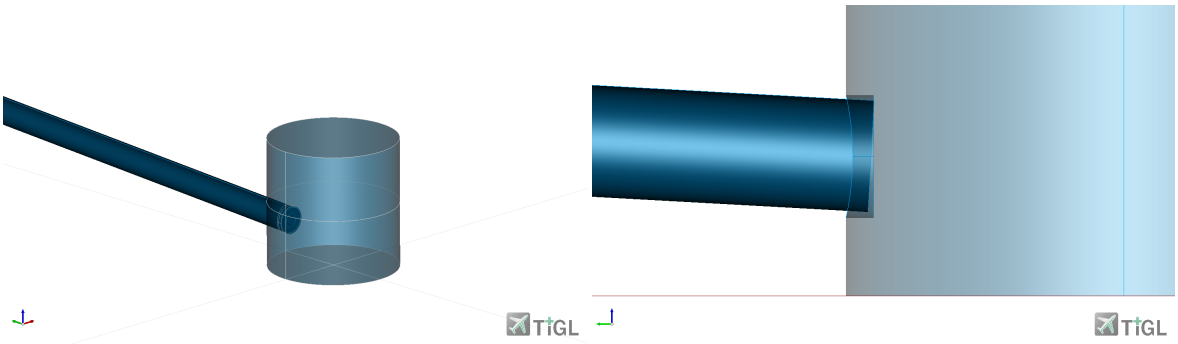

Figure 4.1: Unit test output for the connection between pumps and lines and a detailed view (right) showing the geometry errors in the connection

Chapter 5

Implementation

This chapter aims to provide a thorough explanation of how the different rules were implemented, providing the logical steps taken in order to formalize the knowledge as seen in Chapter 4.

5.1 Codex's production rules

In order to understand how the geometry creation and the geometry verification rules are implemented, it is first necessary to understand how production rules work with Codex. As described in Section 3.5, production rules are able to change the topological information of a knowledge graph. To do so, they are composed by two parts: a `When` side and a `Do` side. On the "when" side it is possible to describe queries matching a specific pattern in the knowledge graph. When these patterns are found, the production rule is capable of triggering the "do" side, responsible for executing the rule itself [14]. These rules, due to their topological nature, are used to create new knowledge and add it to the model, enhancing its capabilities.
Listing 5.1 illustrates a production rule responsible for verifying if two elements of class `SupplyLine` are connected to each other and it is made by five separate queries. In such case, whenever the knowledge graph returns true to a combination of all the queries for the same two individuals, the rule is triggered. The dependence of these queries on three different knowledge modules — `FS` (Fuel System), `Ver` (Verification), and `GC` (Geo Completeness) — shows how simple it is to incorporate different knowledge modules into the project. The verification and geo completeness modules are used in numerous projects based on Codex, despite the fact that the fuel system knowledge model has only been fully developed for this particular application. Additionally, the geo completeness module itself offers a set of rules that verify the completeness of the parameters necessary to correctly represent the geometries.

Listing 5.1: Example of rule querying for production rules

```
When<SIndividual, SIndividual> { line1, line2 ->
    +(line1..Ver.connectedToRequired..line2)
    +(line1..a..FS.SupplyLine)
    +(line2..a..FS.SupplyLine)
    +(line1..a..GC.Complete)
    +(line2..a..GC.Complete)
} Do { line1, line2 ->
    // code execution
}
```

The usage of production rules with the reasoning provided by the production engines allow to detach the rule development from the application, allowing it to be easily maintained, reused and integrated in different projects. As a drawback, the need to match queries in order to execute the rule means that there is a constant need to observe the graph for possible changes that may match the required queries throughout the reasoning process. This leads to an increased run time which could become a problem when large knowledge databases are involved. Nonetheless, it has been identified that ordering the "when" side queries [14] leaving the most stringent queries on top and the most general ones on the bottom allows to reduce the reasoning time required. This strategy has become a common

practice among Codex developers in recent applications.
The next sections tackle each production rule individually and are divided in three different categories:

- Geometry creation
- Geometry verification
- Certification and Guidelines verification

5.2 Geometry creation

As stated in Section 4.1, in order to properly implement geometric verification tools to the fuel system, it is necessary to first provide a larger level of detail to the geometry of the components.Priority in their design was given to fuel pumps and valves because it was recognized that the architecture would greatly benefit from the greater level of detail provided by these components. Because they are essential to the fuel system, their placement must be carefully considered.
The implementation of the production rules, as previously mentioned, is intended to establish a proof of concept for the usability of the codex-geometry module and its various applications. The production rules are not extensive. Due to the lack of detailed geometric information regarding fuel system components available, the geometry here presented is to be considered cautiously. Reference geometry was taken from Lufthansa's training manual for the A320 series [47], whenever this information was available.

5.2.1 Boost pumps

Cartridge-in-Canister architecture for the boost pumps were considered, resulting in a cylindrical shape. In order to provide additional detail about its geometry, connection ports, responsible for connecting the pump to the fuel lines, were designed and integrated into the primitive pump geometry. To properly create the geometry, users need to provide information on the positioning of such ports, other than information on the boost pump itself.
Two main information are required to define the pump's placement: a root point. given by the `hasRootPosition`, from where the geometry will develop and a direction, identifying the pump's longitudinal axis. The direction is added to the model through the property `hasDirection`
In addition to that, users need to provide information on the connection port, in order to correctly place it on the pump. To do so, users must provide the port's radius, length, height, and angular position on the pump's surface. This angular position is given with respect to the X axis provided in the model. Listing 5.2 illustrates how it is possible to add this knowledge to the model in order to activate the rule. The programming language Kotlin is used in this project to leverage Codex's capabilities.
An interesting thing to observe is how the used property affects the required object. Whenever an `ObjectProperty` is required, the assertion expects the third element of the triple to be a resource, or an individual. On the other hand, if the information is a quantity, or a value, a `DataProperty` can be used, thus requiring the creation of a typed literal, which takes the place of the resource. Another type of property existent is the `ParameterProperty`, which is a subclass of data properties focusing on quantities, and thus requiring a specific dimension attached to it. These properties are very useful when the developer wants to restrict what kind of information should be given, in this case, the measurement unit it expects.
Through the use of cylindrical coordinates, the production rule is capable of correctly positioning the port on the pump. By using the root position, the pump's radius (directly integrated in the production rule), the angle and the port's length, the correct placement point is found as follows

$$
\begin{aligned}
x &= rootPointX + portHeight \\
y &= rootPointY + (pumpRadius - portLength) * \cos(portAngle) \\
z &= rootPointZ + (pumpRadius - portLength) * \sin(portAngle)
\end{aligned}
$$

Listing 5.2: Example of knowledge declaration to create a boost pump by the production rules

```
val pump = assertedModel.createAnonymousIndividual(FS.BoostPump)
assertedModel.addAssertion(pump, FS.hasRootPosition, point.individual)
assertedModel.addAssertion(pump, FS.hasDirection, direction.individual)

val connectionPort1 =
assertedModel.createAnonymousIndividual(FS.ConnectionPort)
assertedModel.addAssertion(connectionPort1, FS.hasRadius,
    assertedModel.createTypedLiteral(0.02[u.m]))
assertedModel.addAssertion(connectionPort1, FS.hasLength,
    assertedModel.createTypedLiteral(0.01[u.m]))
assertedModel.addAssertion(connectionPort1, FS.hasHeight,
    assertedModel.createTypedLiteral(0.05[u.m]))
assertedModel.addAssertion(connectionPort1, FS.hasAngularPosition,
    assertedModel.createTypedLiteral(90.0[u.deg]))

assertedModel.addAssertion(pump, FS.hasConnectionPort, connectionPort1)
```

Where rootPointX, rootPointY and rootPointZ are the three cartesian coordinates for the root point, pumpRadius is the radius predefined in the production rule, the portLength and the portAngle are given by the user. An important feature of Codex is given by its codex-quantity module. This module offers over 250 different dimensions (i.e. length, area, volume, time, etc.) and 270 units, with units conversion and matching capabilities [14]. This allows for an easier usage of the geometric module, as users do not need to focus on agreeing on a common unit and can leverage Codex's reasoning capabilities to provide unit conversion seamlessly.
The direction is computed using the aforementioned cylindrical coordinates as follows:

$$direction = (0, \cos(portAngle), \sin(portAngle))$$

Since ports are modelled as cylinders as well, these two information (port placement and direction) are added to the model and used to create its geometry. Afterwards, geometry verification rules are put into place in order to check the correctness of the recently created geometry. During this process, the positioning of the port together with its dimension are analyzed to make sure the geometry can be created. This allows for a better control on the geometry creation, leaving less margin for errors due to bad calculations or lack of attention to the design. Up until this stage, the pump is created in a standardized way, with the Z axis being the reference axis for its creation and thus ignoring the user provided direction. This is done to allow an easier integration of the connection port to the geometry. In fact, in order to use the cylindrical coordinates as previously mentioned, it is important to know exactly the pump's position with respect to the main cartesian axis. If the implementation considered from the start the pump on its desired position, the computation of the cartesian coordinates would become much harder to implement. To solve that, it was chosen to implement a pump rotation after the completion of its geometry. Using basic principles of linear algebra, it is then possible to rotate the recently created complex geometry and relative geometric knowledge such as the port's position and direction. To do so, a series of rotations are executed based on the normalized pump direction. The rotation with respect to the first axis (X) is taken as follows:

$$\theta_X = \arctan \frac{u_Y}{u_Z}$$

Where u is the normalized direction vector and u_Y and u_Z are two of its coordinates, Y and Z respectively. With θ_X a first rotation vector v_1 is computed:

$$v_1 = (u_X, u_Y * \cos(-\theta_X) - u_Z * \sin(-\theta_X), u_Y * \sin(-\theta_X) + u_Z * \cos(-\theta_X))$$

The second rotation angle is then calculated as:

$$\theta_Y = \arctan \frac{v_{1_z}}{v_{1_x}}$$

And the second rotation vector, v_2:

$$v_2 = (v_{1_x} * \cos(-\theta_Y) + v_{1_z} * \sin(-\theta_Y), v_{1_y}, -v_{1_x} * \sin(-\theta_Y) + v_{1_z} * \cos(-\theta_Y))$$

Finally, the third rotation angle, θ_Z is computed:

$$\theta_Z = \arctan \frac{v_{2_y}}{v_{2_x}}$$

These three angles are then used to rotate the 3D geometry in space using a 3-2-1 sequence, that is, the first rotation is done with θ_Z, the second with θ_Y and the final rotation with θ_X. After rotating all the necessary geometric entities, the newly computed information is added to the knowledge graph. Listing 5.3 provides an example of how the production rule is capable of adding new information to the model through assertions. This information is linked to the individual `port`, that was previously defined by the user in order to trigger the production rule, and can be further used, together with the created geometry, to provide different kinds of geometric checks to the model.

Listing 5.3: Example of added knowledge to a model through a production rule

```
model.addAssertion(port, GKM.hasPoints, connectionPoint)
model.addAssertion(port, FS.hasFace, portFace)
model.addAssertion(port, FS.hasDirection, portDirection)
```

In Figure 5.1, an example of boost pump is given. It is possible to note the connection port, which is an integral part of the pump design.

Figure 5.1: Boost pump

Adding multiple ports to a fuel pump

In order to cope with different geometry requirements, a second production rule for creating fuel pumps was developed. By declaring multiple connection ports linked to a same pump, it is possible to create different pump types with a single production rule. An example of usage of this production rule is the creation of snorkel pumps, which require a port to connect the pump outlet, as for the traditional pumps, and a secondary port to link the snorkel line to the pump. Listing 5.4 illustrates the "when" side that queries the production rule for the creation of spar-mounted snorkel pumps, where the ports are taken as a list of resources (`SRDFList`). The construction process is similar to the one for simple pumps, with the addition that ports are created individually and then added to the primitive geometry iteratively. Figure 5.2 illustrates a spar-mounted APU pump created using this rule and its connecting

Listing 5.4: When side for the creation of snorkel pumps using the production rules

```
When<SIndividual, SIndividual, SIndividual, SRDFList>
{pump, point, direction, ports ->
    +(pump..FS.hasRootPosition..point)
    +(pump..a..FS.SnorkelPump)
    +(pump..GKM.hasDirection..direction)
    +(pump..FS.hasConnectionPort..ports)
    +not(pump..a..GC.Complete)
    +(point..a..GKM.Point)
    +(direction..a..GKM.Direction)
} Do {...}
```

lines.

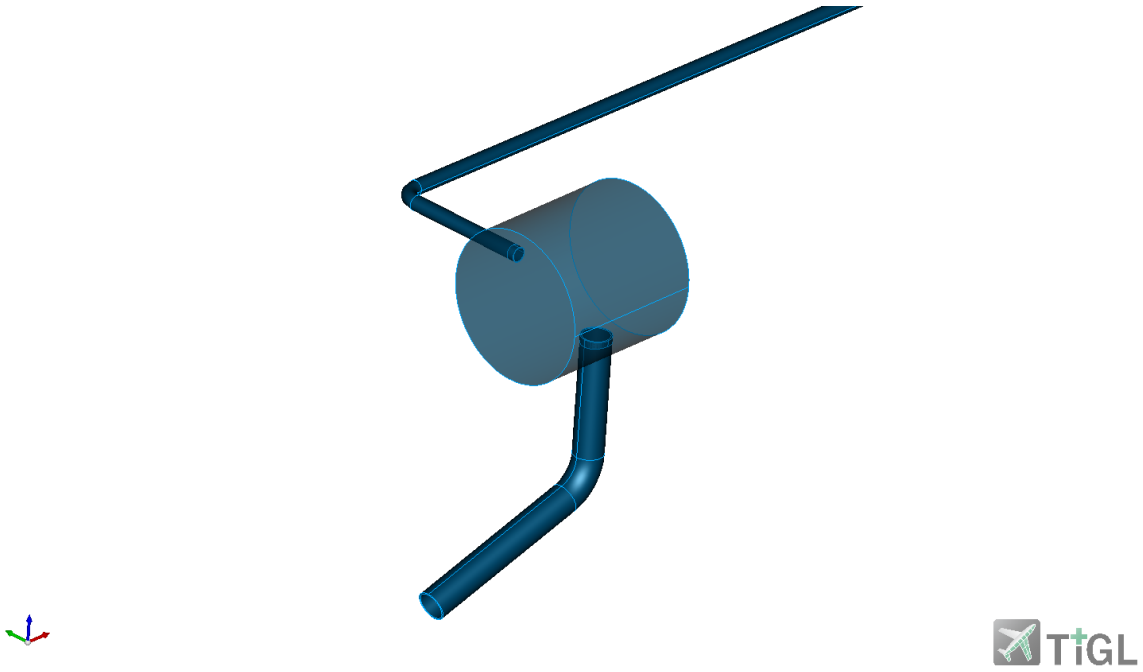

Figure 5.2: Snorkel pump with an inlet and an outlet port

Production rule to automate the root point definition

Because this type of pump is frequently installed at the bottom of a fuel tank, and because fuel tanks can have complex geometry, making manual placement of a pump difficult, a production rule aimed at determining the correct root point for a pump based on a desired approximate position was added. In fact, by using the property `hasDesiredPosition` linked to a specific fuel tank instead of `hasRootPosition`, the reasoning triggers a secondary production rule responsible for finding an optimal root position on the tank's internal surface.
To do so, an algorithm based on Opencascade's open source kernel was developed and is described in Figure 5.3.
In order to extract the bottom surfaces of a tank, an approach based on analytic geometry was used. First, the different surfaces of the tank are saved to a list. The algorithm iterates through the list of surfaces and computes the normal vector to each surface at their center. To do that, it is assumed each surface to be plane, that is a face, in order to have the same normal vector at each point of its surfaces. Curved surfaces, whenever existent, are converted into smaller surfaces using a "face" representation, provided by Opencascade. A dot product (scalar product) is then computed for each surface's normal vector with a reference vector representing the direction pointing to the negative Z axis, or the bottom of a geometry according to the conventions defined for this development. Whenever the dot product is over a certain tolerance, thus meaning the vectors are more or less aligned, both the surface and its respective direction are saved and used to find the optimal candidate to become the root point for

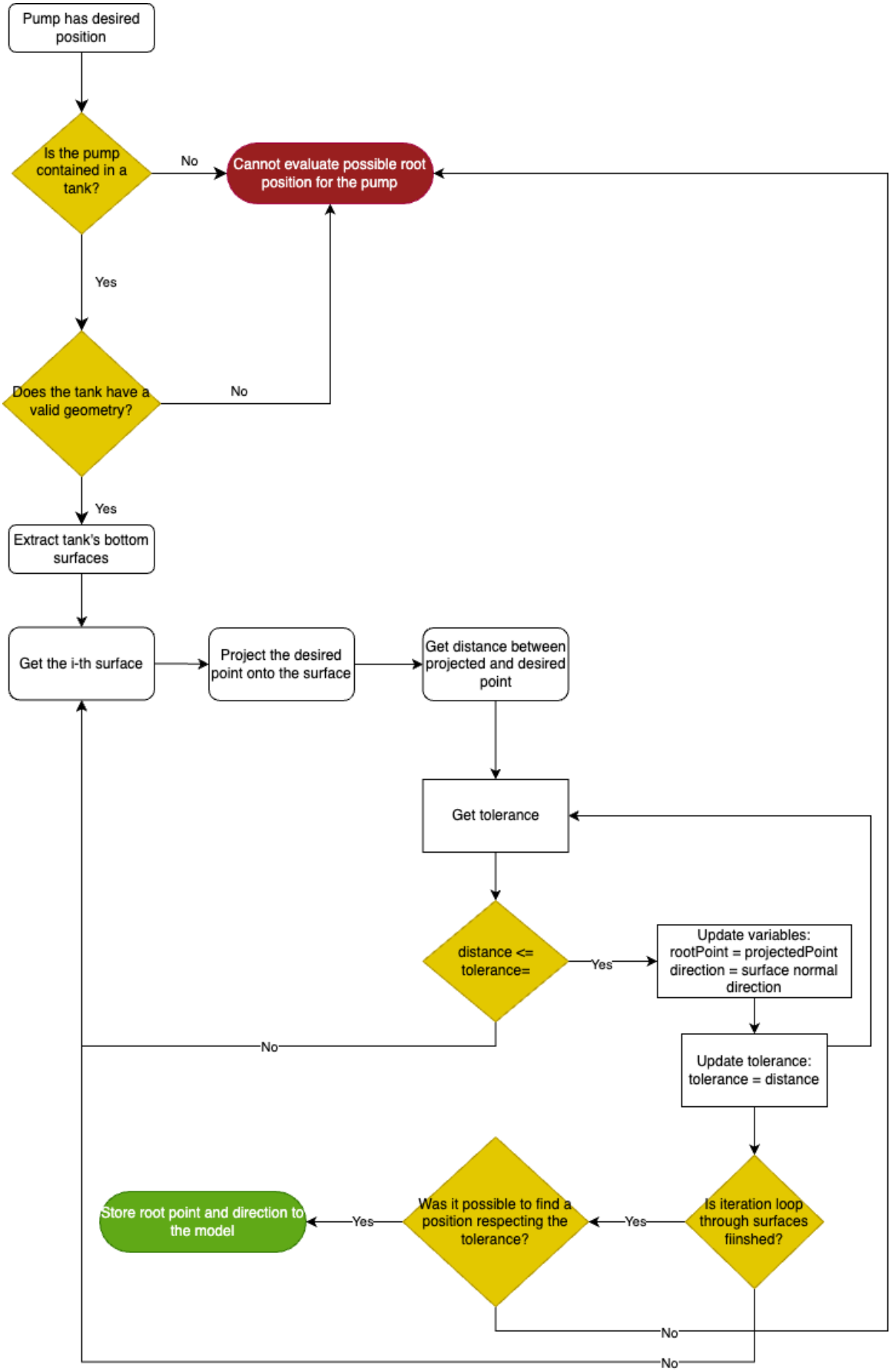

Figure 5.3: Logic representation of the algorithm for finding the optimal root point on the bottom surface of a tank for a fuel pump

the pump.
A new loop is then started, this time with the extracted bottom surfaces. In this new cycle, for each surface the algorithm projects the desired point onto the surface and then three values are calculated: the distances in the X and Y axis and the complete distance. While the X and Y distances are straightforward, the complete distance is taken using the euclidean distance between the two points as follows:

$$\text{Total distance} = \sqrt{distX^2 + distY^2}$$

The tolerances for the X and Y distances are fixed to 0.5 m, while the tolerance for the total distance has a starting value of 0.5 m and then is iteratively decreased whenever the following condition is reached: *Req: The three main distances must be smaller or equal to their respective tolerances.*
In such cases, a temporary variable stores the value of the projected position and the surface's direction and updates the total distance's tolerance with the current distance. If a second candidate surface gives a projected point that respects this now more stringent requirement, the values are swapped, guaranteeing in the end that the final candidate to be the root point is the one closest to the user's desired point. In case the desired point does not lie exactly above or below any of the surfaces, or in cases the defined tolerances do not provide any good candidate, the algorithm provides an error stating that it could not find any viable point close to the desired one that matches the criteria, finishing the reasoning process. Although the recommended approach is for the user to provide a different desired point, paying attention it is at least close to the tank's surface, it is always possible to change the tolerances, if needed.

5.2.2 Valves

A simplified valve representation was determined to be adequate for this master's thesis. This led to the development of a production rule that focused on building linear shut-off valves connecting two distinct fuel lines. As for the boost pumps, a specific query is necessary in order to trigger this rule. Adding the geometric characteristics of the valve to the query ("when" side) allows to correctly trigger this rule only when all the geometric information is available. The geometric representation of this two-way shutoff valve takes inspiration on linear valves illustrated in [47] and is composed by three main parts:

1. Left port connection
2. Right port connection
3. Central part

The central, cubic part, which is a geometric representation of the opening mechanism of these valves, is attached to the left and right ports, both of which are modeled as hollowed cylinders. Figure 5.4 illustrates a shutoff valve created with the production rule.
Listing 5.5 shows the "when" side for this production rule. It is possible to see how all the required geometric information are requested on the "when" side and are of type literal (`RDFLiteral`).

Listing 5.5: When side for the creation of shutoff valves using the production rules

```
When<SIndividual, SIndividual, SIndividual, SRDFLiteral>
{valve, point, direction, outerRadius ->
    +(valve..FS.hasRootPosition..point)
    +(valve..GKM.hasDirection..direction)
    +(valve..a..FS.SOValve)
    +(point..a..GKM.Point)
    +(valve..FS.hasRadius..outerRadius)
    +not(valve..a..GC.Complete)
} Do {...}
```

For the creation of these valves, in addition to the root position, which represents the valve's center point, and the direction, defining the longitudinal axis - the same direction of the pipes -, a third geometric information is required: the radius. This radius should be equivalent to the outer radius of the connecting pipes, in order to provide an adequate pipe-valve match. The algorithm infers the remaining geometric knowledge by making use of parametric rules. As an example, the outer radius for the shutoff valves is created considering a 20% increase on the port's inner radius. The same

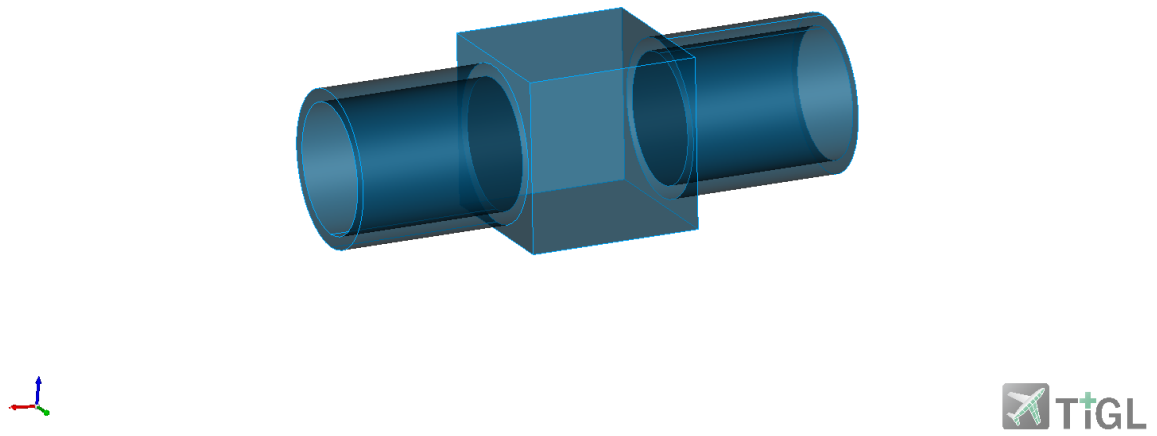

Figure 5.4: Two-way shutoff valve

rotation algorithm implemented for the boost pumps can be found for the valves.
After the geometry is created, the production rule is in charge of adding the pertinent knowledge to the knowledge graph, making it accessible by any additional production and parametric rules, if they are registered to the reasoning. Due to their more static geometric nature, these linear valves do not require the ports geometric characteristics to be predefined and added to the model beforehand, which is an interesting difference between the creation processes for the valve and the pump. In fact, the amount of information that must be provided by the user is significantly decreased by utilizing a streamlined geometric representation of these components. The knowledge graph will nonetheless continue to be enhanced after the production rule is triggered, and the connection ports and any information related to them that is produced by this rule will also be automatically added to the model.

5.2.3 Fuel tanks

An important characteristic of Codex is its great flexibility when modeling the knowledge. Although very helpful to the user, the geometry creation based on production rules is not mandatory. Still, it is highly recommended to use them in order to be sure that the necessary knowledge will be added to the model, in order to allow the geometry verification tool to work as expected. Fuel tanks enjoy the same freedom as any other component for what concerns their creation.
As an attempt to show how codex-geometry can easily integrate imported information from different sources, an additional function capable of extracting geometric information from aircraft structural components was created. At DLR and in multiple aerospace related institutions, CPACS is the state-of-the-art configuration schema used to share information between stakeholders. A function capable of extracting information related to the wing's structural geometry was implemented in Python. Leveraging TiGL Geometry Library [44], it is possible to extract the geometry related to wing spars and ribs. An algorithm that uses Opencascade technologies then takes each rib's geometry and transforms it into an object of class `TopoDS_Face` [43]. Assuming that each face is rectangular, each face is further divided into four connecting wires. The last step extracts the start and end points for each wire and arranges them in a clockwise fashion. Every rib in the model goes through this process once, and at the end of it, a list of the four ordered extreme points for every rib is generated and saved into a JSON file that can be exported to Kotlin and then integrated into Codex.
It is important to note that only the right-sided elements are explicitly described into CPACS by convention, whereas the left side is typically a mirrored entity based on the right side, and as a result,

only the geometric data related to the right wing is stored to this JSON file.
No methods were developed to support the design space creation for tanks based on different parts of the aircraft. Nonetheless, Codex allows for their creation using the Geometry Syntax provided by codex-geometry.

Definition of tank volumes based on the design space

After the design space has been defined and the JSON file containing the ribs positioning is imported to the Codex environment, a second function can be used to separate the design space into multiple fuel tanks. To do so, the user can declare an estimate volume for each tank, that will be taken into account by an algorithm responsible for this process. The algorithm searches for a combination of ribs to create a lofted space whose total volume corresponds, within a certain tolerance, to the volume entered by the user using the lofting methods provided by codex-geometry. The most inner position, which is typically occupied by the center tank or the innermost wing tank, is where the process begins. The algorithm then starts looking for the volume of the following tank, starting from the most recent saved position or the end-position for the previous tank.
In the end, the knowledge graph is enriched with the newly created lofted tanks, adding extra value to the model and allowing further computation.

Tank's external surfaces

Due to limitations to the codex-geometry module, calculations involving lofted surfaces, such as determining their area, volume, and intersection with other components, call for the lofted geometry to be entirely filled, giving it a solid appearance. It is anticipated that this limitation will be resolved in the future as a result of Codex's advancements.
Production rules were put into action to provide the external surface of a fuel tank in order to deal with such issues and to provide a more realistic geometry for the model representation. These production rules make use of the exploration functions of Opencascade, which can be used to extract the external surfaces connected to a geometry. In order to visualize the geometry, these surfaces are then stitched back together and saved as a BREP file.

Tank openings

Through the use of production rules, the GeoVerification module has yet another category of geometric attributes added: tank openings. Fuel pipes, which move fuel between tanks, are the main reason they are present. Since the fuel tanks frequently take up a significant amount of space on the wing's structure, pipes are frequently routed internally to these tanks, resulting in the presence of openings in order for the lines to properly pass through them.
These openings are modeled as circular openings and require three basic information to define their geometry:

- The tank on which it shall be created
- Its root position
- Its radius

Since fuel tanks are often modeled without gaps in between at early design stages, it is often possible to use the same root position to create tank openings on two different tanks. This of course requires for two sequences of statements to be added to the model, each linked to its specific tank. The production rule is split into two different rules: the first rule is responsible for simply creating the opening geometry at the specified point with a specified direction. The second is responsible for changing the tank's external surface geometry by means of subtracting the opening's geometry from the tank's geometry itself.
As for the other components, it may be difficult to correctly place a tank opening manually, as it requires both a specific point that must lay on the tank's surface and a direction that shall coincide with the normal vector to this same surface. To aid the user, an additional production rule, as for the fuel pumps, is used in order to find the most optimal position. Given a point linked to an opening by the `hasDesiredPosition` property and an additional information stating to each of the tank's surfaces this cut should be applied, an algorithm finds the most suitable position for the opening's root position and the direction of creation and adds it to the model. As soon as this information

is available, the previously mentioned rule responsible for creating its geometry is activated. Listing 5.6 illustrates a user-declared tank cut provided the desired position. Six different properties are possible to be used by the user in order to define the projection face: `isProjectedOntoBottom`, `isProjectedOntoTop`, `isProjectedOntoFront`, `isProjectedOntoRear`, `isProjectedOntoLeft`, and `isProjectedOntoRight`. Based on these properties, the algorithm provides a different reference vector to be used by the surface extraction function implemented (see Figure 5.3).

Listing 5.6: Example of user-declared tank opening (or cut) to the knowledge model

```
val cut = assertedModel.createAnonymousIndividual(FS.TankCut)
assertedModel.addAssertion(cut, FS.hasDesiredPosition, point.individual)
assertedModel.addAssertion(cut, FS.isProjectedOntoRight, tank)
assertedModel.addAssertion(cut, FS.hasRadius,
    assertedModel.createTypedLiteral(0.05[u.m]))
```

5.2.4 Additional production rules

The GeoVerification module provides additional production rules aimed at aiding the user when constructing the geometry. Since these rules [20] are the result of earlier research, this thesis will not go into great detail about them. In order to correctly produce the geometry, the construction of supply lines necessitates a number of production rules that must be activated sequentially. The activation of this sequential rule is reliant on the addition of new knowledge to the graph by the starting production rule, which in turn triggers the activation of the second rule, thereby triggering the addition of new knowledge to the model, and so on. The creation of supply lines relies on calling a specific function and passing the required arguments (reference model, curve radius, list of points guiding the line, and inner and outer radius) rather than requiring users to add statements to the model. The function is then responsible for computing any missing parameter and adding the information to the knowledge graph through statements.

5.3 Geometry verification

As one of the major goals of this project, the possibility to verify the geometric correctness of a component or the connection between components is necessary to ensure proper system design. This section covers the detailed description of the geometry related features implemented into the GeoVerification tool, as listed in Table 4.1. When possible, features with similar capabilities will be explained in one place while making clear how they differ. For each production rule is provided the respective created knowledge that is added to the graph, in order to give the reader a comprehensive view of how these verification functions affect the design and how their features can be leveraged by designers.

Disclaimer: when describing properties and classes using the semantic syntax, prefixes are represented by semicolons before the property/class' name (e.g. `FS:thisProperty` *, where FS is the prefix representing the Fuel System knowledge module.*).

During the course of the project's development, it was decided to primarily rely on the fuel system domain (`FS`) to define these properties. Codex's inference abilities were utilized to enable the incorporation of these properties within the verification domain itself. By categorizing the confirmed and disproven properties from `FS` as subcategories of `Ver:verified` and `Ver:falsified`, respectively, a unified domain serves as a repository for all evaluated properties, even when originating from diverse domains. This implies that, even if an additional domain were to be introduced—such as the structural domain `Struc`—it would still be feasible to associate all properties with the verification domain, thereby preserving their origin information. Notably, the model can display all verified or falsified statements within the `Ver` domain, while also allowing for statement filtering to discern whether they pertain to the fuel system or the structural domain.

5.3.1 Tank volumes

A very basic and preliminary check is related to the tank volumes. As explained in Subsection 5.2.3, fuel tanks can be created using the methods integrated into the GeoVerification tool. Still, users have

complete freedom to design their fuel tanks as they better prefer. In such cases, to aid their design choices, it is possible to set a required tank volume that can be checked as soon as its geometry is created. This function leverages the codex-geometry module and uses the GeometryAdapter [20] provided. If the volume matches the required value, within the tolerance, this check is verified, otherwise it is falsified.

Resulting statements

- Verified:

  `<tank, FS:hasVolumeVerified, volume>`

- Falsified:

  `<tank, FS:hasVolumeFalsified, volume>`

5.3.2 Components inside tanks

The `checkContainedInForConnectionElements` rule is used whenever it is necessary to check whether a component is fully contained inside a fuel tank. In order to be activated, this production rules looks for patterns on the knowledge graph where the following triple can be identified:

`<component, Ver:containedInRequired, tank>`

Provided the tank has a design space and both the component and the tank passed the geocompleteness test (provided by codex-geometry), this function is capable of extracting both the tank's and component's geometry with the `getShape()` method provided by the GeometryAdapter. The intersection between both shapes is evaluated and its volume then computed. If this volume exceeds the tolerance of $1 \times 10^{-5}\,\mathrm{m}^3$ the check is falsified.
An important feature of the GeoVerification functions is that they are capable of providing feedback to the user while the simulation runs through messages being printed to the logger. A common characteristic all the functions have is to start with a message stating which check is currently being executed and, after the computation is over, another message is printed out providing the result of such analysis. Listing 5.7 illustrates both the start and end messages provided to the user.

Listing 5.7: Messages provided to the user when checking for a components containment inside a tank

```
// starting message
println("Checked␣if␣${contained.readableName}␣is␣contained
in␣${containing.readableName}.")
// feedback message providing the results
println("␣␣␣>>␣${contained.readableName}␣is␣contained␣in
${containing.readableName}:␣$check,␣overlap:␣$vol
(tolerance:␣$tolerance)")
```

Resulting statements

- Verified:

  `<component, Ver:containedInVerified, tank>`

- Falsified:

  `<component, Ver:containedInFalsified, tank>`

Differently from the conclusions drawn in Subsection 5.3.1, the terminology used to describe the mentioned containment properties belongs to the Verification package. Codex has the capability to integrate information from different domains, allowing the utilization of properties from various disciplines without encountering compatibility issues.

5.3.3 Connection between components and lines

Fuel system components such as fuel pumps and valves are herein identified as connection elements. In order to verify the geometric effectiveness of their connection to the supply lines, the information on their related connection ports is exploited. Due to the geometric complexness and heterogenicity of requirements related to their connection, three separate functions are defined in order to properly evaluate them:

1. `checkConnectedToForPortsAndLines`
2. `checkAlignmentForConnectionPortsAndLines`
3. `checkLinePortMatchingSize`

Although they complement each other, each check is executed independently. This is done in order to provide precise information for the user regarding the encountered problems in a model, while also providing feedback on parts of the design that are compliant with requirements, even if the overall requirement is not respected.

Connection

The first rule focuses on disclosing details about potential intersections between the lines and the ports and/or potential gaps in their distance. The supply line's intersection with the component is the first property to be assessed. You can determine the volume of a potential overlap by using the intersection method offered by GeometrySyntax. When the geometry is accurate, the overlap should typically be zero. However, a minor overlap could occasionally be discovered as a result of representational problems. A tolerance of $1 \times 10^{-6}\,\mathrm{m}^3$ is provided to account for these exceptions. This tolerance can be modified by the user according to his/her needs.
Afterwards, knowledge regarding the connection port is used to find gaps in the connection. The GeometryAdapter offers means of computing the minimum distance between two shapes using the `getMinimumDistance` function between two shapes. In order for it to properly work, the shape from both the connection port and the supply line are used and a tolerance of $1 \times 10^{-5}\,\mathrm{m}$ is applied.
The analysis is successful only if both the overlapping and the distance measurements are within the required tolerances. Figure 5.5 shows a non-compliant case due to an overlap, while Figure 5.6 illustrates a gap between the line and the port end-point. In the first case, where the pipe overlaps with the tank, the following output is displayed to the user:

```
>> Front inner pump is connected to Feed line through
Connection port 1: false, overlap 4.712388980384944E-6[m3]
(tolerance: 1.0E-6[m3])
```

The second error, where components present a gap on their connection, give a similar error, this time providing the distance:

```
>> Front inner pump is connected to Feed line through
Connection port 1: false
overlap: 0.0[m3] (tolerance: 1.0E-6[m3])
distance: 0.0019999999999999463[m] (tolerance: 1.0E-5[m])
```

Alignment

Additional information provided by the connection port can be used to verify the correct alignment with the supply line. By leveraging the geometric knowledge linked to the supply line, it is possible to individuate the points that guide the same. The two points composing the line's end points are taken and their distance with the port itself is evaluated in order to find which end of the line should be connected to the port. Then, the list of points is reordered such that the first point is the one closest to the port, the p_1. The remaining points are added following the geometric sequence, with the last point, p_n, representing the opposite extremity. Since two consecutive points make a segment on the line, only the first segment, the one closest to the port, is studied. The direction of this segment is computed by:

$$\text{segment direction} = ((p_{2_x} - p_{1_x}), (p_{2_y} - p_{1_y}), (p_{2_z} - p_{1_z}))$$

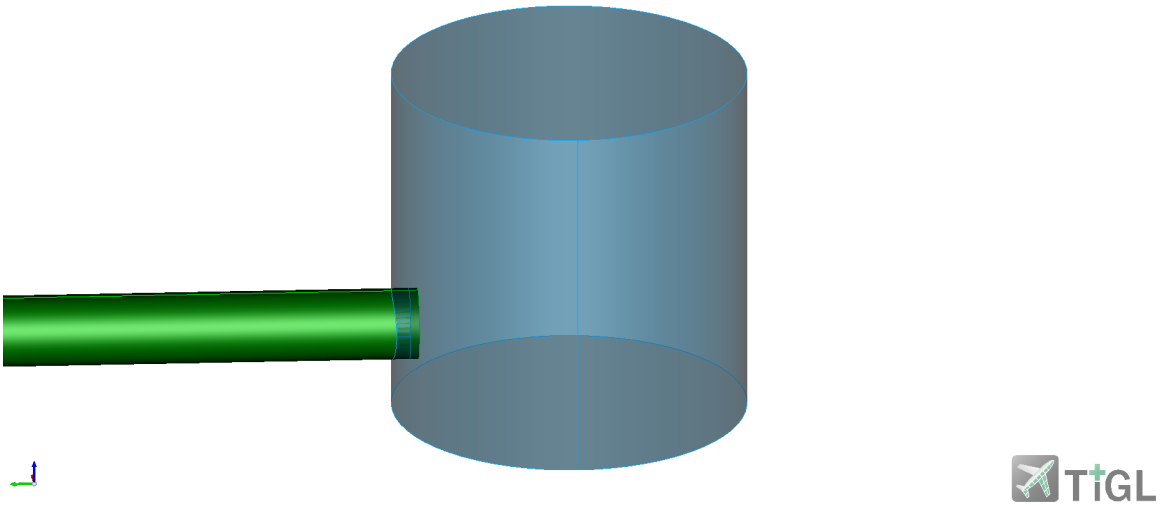

Figure 5.5: Geometry mismatch due to an overlapping supply line

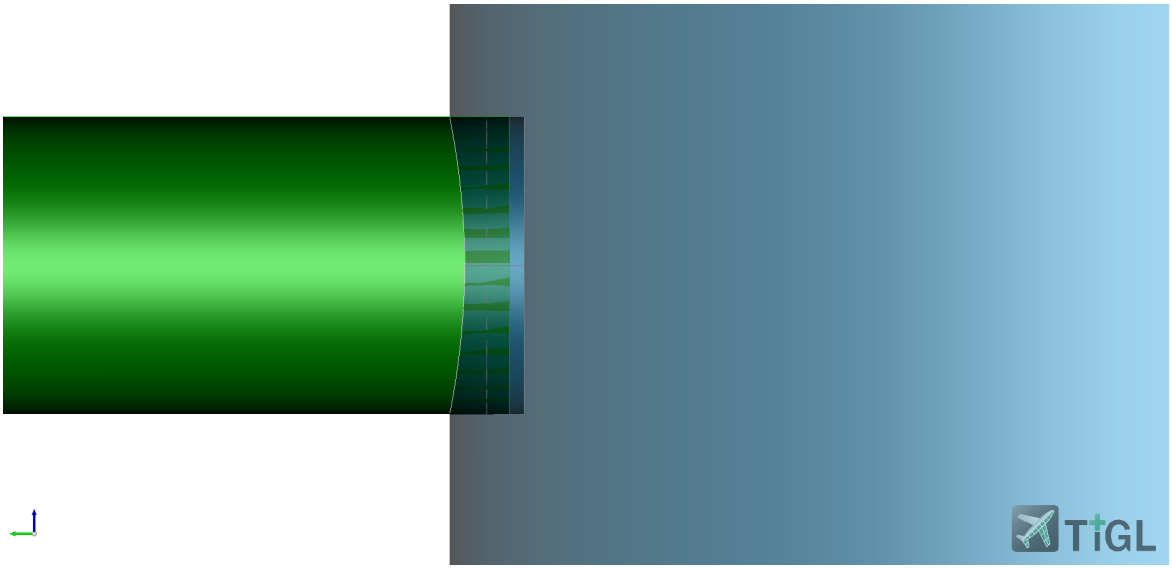

Figure 5.6: Geometry mismatch due to a gap between the supply line and the connection port

The port's direction comes from the query related to the production rule's "when" side. In fact, whenever a quantity is necessary to make computations used in the rule, it is common practice to query it. Doing so allows the rule to be triggered as soon as this property becomes available (as long as the remaining queried information is already available). This strategy allows to avoid unintentional usage of the rule, that could lead to wrong calculations or crashes.
In order to compute the effective alignment between the line and the port's normal direction, the cross product between the vectors, $\vec{c}$ is computed and the result normalized. The resulting angle can be computed as:

$$\theta = \arcsin\|\vec{c}\| \tag{5.1}$$

With:

$$\vec{c} = \vec{a} \times \vec{b}$$
$$\|\vec{c}\| = \sqrt{c_x^2 + c_y^2 + c_z^2}$$

Where $\vec{c}$, $\vec{a}$, $\vec{b}$ are the resulting cross product, the segment's direction and the port's normal vector, respectively, $\|\vec{c}\|$ the euclidean norm for $\vec{c}$, and θ the measured angle in radian. c_x, c_y, and c_z represent the x, y and z coordinates for $\hat{c}$. The tolerance set to this function is 1×10^{-6} rad.
Figure 5.7 illustrates a case of misalignment between the port and the line, leading to the following output:

```
>> Feed line is aligned to Connection port 1
from Front inner pump: false, angle: 3.1484103072924685[deg]
```

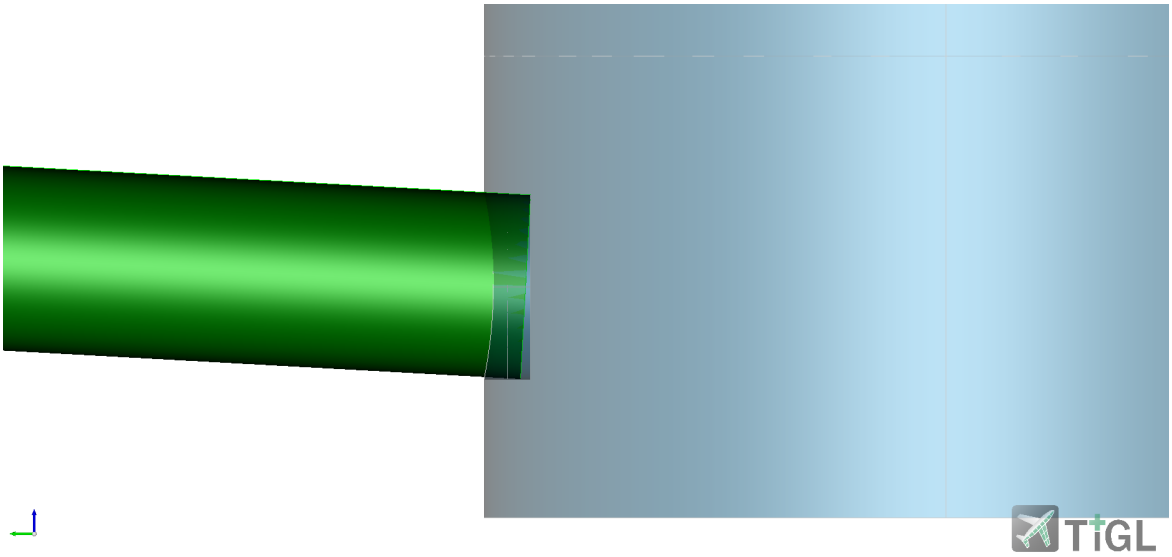

Figure 5.7: Geometry mismatch due to misalignment

Port and line dimension

The last check related to the connection between a pump or valve and a supply line is provided by `checkLinePortMatchingSize`. This rule checks for the absolute difference between the line's outer radius and the port's radius and turns out to be successful if their difference is under 1×10^{-5} m. Listing 5.8 illustrates this rule in its completeness. By exploiting the codex-quantity module, it is possible to make comparisons between two values without worrying about their units, as long as they have the same dimension (i.e. a length).

Listing 5.8: Geometry rule to check if a supply line and a connection port have matching sizes

```
val checkLinePortMatchingSize by Rule(FS, Ver, GKM, GC) {
When<SIndividual, SIndividual, SRDFLiteral, SRDFLiteral>
{line, port, radius1, radius2 ->
    +(line..FS.isConnectedToPort..port)
    +(line..FS.hasOuterRadius..radius1)
    +(port..FS.hasRadius..radius2)
    +(line..a..GC.Complete)
} Do {line, port, radius1, radius2 ->
    val tol = 1e-5 [u.m]
    val diff = abs(radius1.quantity - radius2.quantity)
    val check = diff <= tol
    println("Checked␣if␣${line.readableName}␣and␣${port.readableName}
␣␣␣␣have␣matching␣sizes:␣$check")
    if (diff > 0.0 [u.m]) {println("␣␣␣␣Difference␣in␣radius␣size:␣$diff")}
    model.addAssertion(line, if(check) FS.hasMatchingSizesVerified
    else FS.hasMatchingSizesFalsified, port)
}
}
```

The resulting statements for the three functions here discussed are provided below:

Resulting statements

- Verified:
  - Connection:

    `<connectionElement, Ver:connectedToVerified, line>`

    `<line, Ver:connectedToVerified, connectionElement>`
  - Alignment:

    `<line, FS:isAlignedToVerified, port>`
  - Size:

    `<line, FS:hasMatchingSizesVerified, port>`
- Falsified:
  - Connection:

    `<connectionElement, Ver:connectedToFalsified, line>`

    `<line, Ver:connectedToFalsified, connectionElement>`
  - Alignment:

    `<line, FS:isAlignedToFalsified, port>`
  - Size:

    `<line, FS:hasMatchingSizesFalsified, port>`

5.3.4 Boost pump attachment to tank's bottom surface

An aircraft's wing-mounted fuel tanks are very difficult to access internally, necessitating extra time and specialized personnel to perform necessary maintenance. The components that need maintenance the most frequently include fuel pumps because of their essential functions. To address this, many aircraft manufacturers use the cartridge-in-canister pump configuration, which enables simple pump removal without direct access to the internal volume of the tank and eliminates the need for it to be emptied beforehand. This pump is typically placed at the bottom of the tank to ensure easy access from the outside and to maximize the fuel suction assisted by gravity.
These characteristics make this requirement a good candidate for a geometric verification of the pump's design when integrated with the fuel tank. The `checkAttachedToBottomForBoostPumps` function allows to verify whether a fuel pump is physically attached to the bottom surface of a specific tank. To do so, users may activate the rule using the property `Ver:attachedToRequired` linking the subject, the fuel pump, to the object tank.
Due to codex-geometry's module not being yet complete, some functionalities are currently missing and are taken directly from Opencascade. One of these functionalities is the surface extraction, needed to identify the bottom surfaces of a fuel tank. Figure 5.8 illustrates the logic behind the algorithm. The first step is to identify which surfaces can be categorized as bottom surfaces. To do so, the `extractSurfacesFromDesignSpace` function, created for the scope of this project, is used. By comparing the reference vector $(0, 0, -1)$ with each surface's normal vector, this function returns a list of surfaces that match the reference vector, meaning they are parallel up to a certain tolerance. The algorithm proceeds checking two main aspects that constitute the attachment:

1. the distance between the pump and the closest point to it on the surface.
2. the misalignment offered by the pump's longitudinal axis to the surface's normal vector.

The first step on the process is to identify which of the bottom surfaces is the closest one to the pump. To do so, the information on the pump's root point is extracted from the knowledge graph and used to find the surface that provides the closest distance from the root point, by applying point projection algorithms. Listing 5.9 illustrates how to extract information from the model using statements. The `null` object allows to get any object matching the pattern. In this instance, the `hasRootPosition` property was not queried on the "when" side because the pump is expected to have a valid geometry. Instead, the production rule that produced the geometry itself is what is necessary for its existence. Users must keep in mind that the GeoVerification tool will not function properly if they choose not to add the necessary knowledge to the model whenever the available geometry creation rules are not used.

Listing 5.9: Information extraction from the knowledge graph through statements

```
val pumpRootIndiv = model.statements(pump, FS.hasRootPosition,
null).first().obj.getAs<SIndividual>()
val pumpRootPoint = getPoint(pumpRootIndiv).asDoubleArray()
```

The minimum distance between the pump's geometry and the target surface is then computed and a tolerance of 0.05 m is provided to account for the existence of supports below the fuel pump. Afterwards, the cross product between the directions is computed using equation 5.1 and a tolerance of 15 degrees is provided to account for misalignments between the two geometries. Overlaps in the geometries are not computed here as they are already dealt by the `checkContainedInForConnectionElements` function. At the end of the evaluation, users receive a message output stating the distance between elements and their misalignment angle. If both measures respect the tolerance, the check is successful.

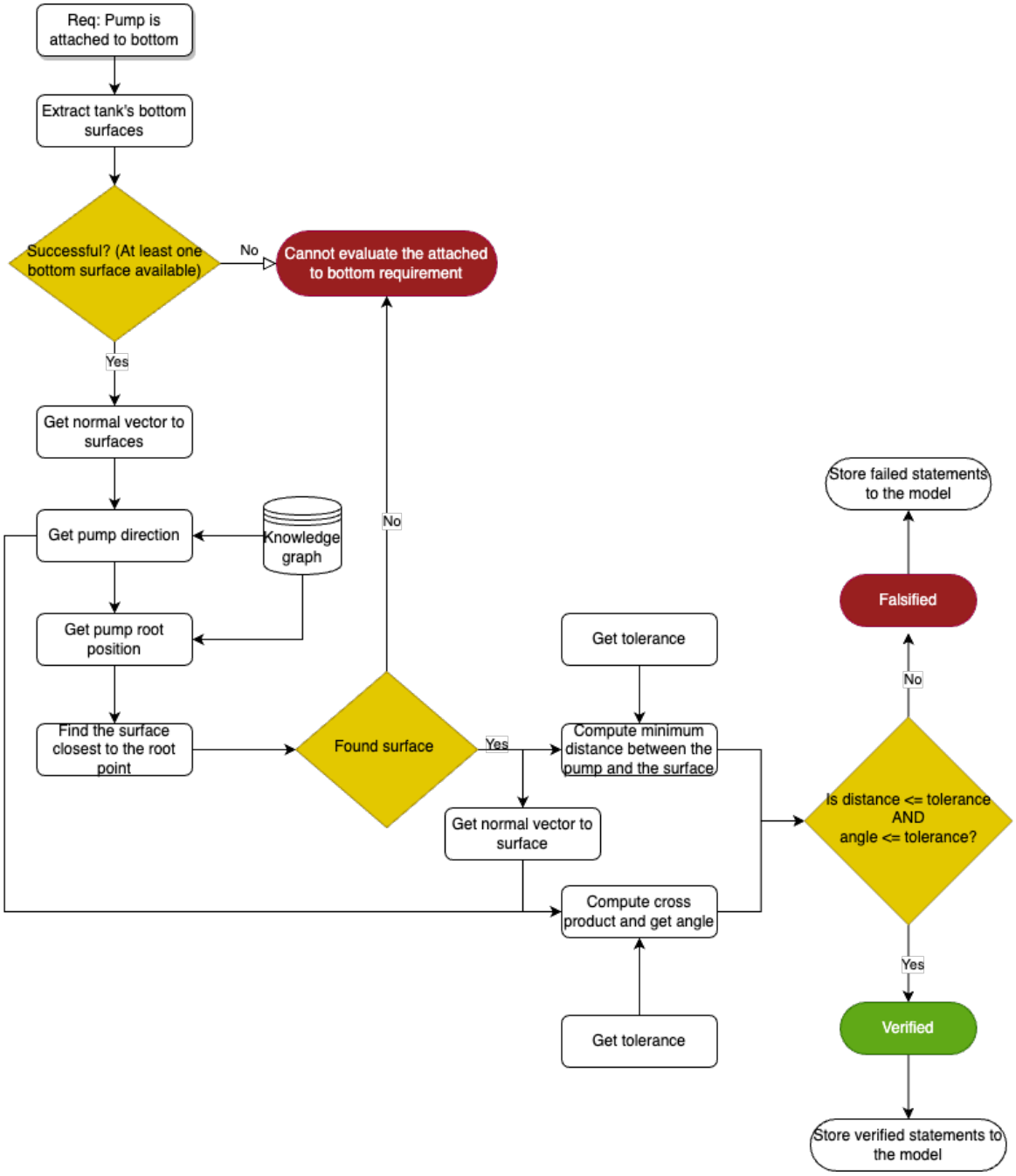

Figure 5.8: Logic workflow to check if a pump is attached to the bottom of a tank

Resulting statements

- Verified:

```
<pump, Ver:attachedToVerified, tank>
```

- Falsified:

```
<pump, Ver:attachedToFalsified, tank>
```

5.3.5 Intersection between pumps and ribs

In order to check for errors in the geometry caused by an overlap in positioning between structural and fuel system components, it is possible to use the geometric representation of the wing ribs by utilizing the knowledge of the aircraft's structure. Due to their relatively large size, which could cause geometry issues if they are positioned in areas where the ribs are close together, fuel pumps

are an excellent candidate to test this functionality. By understanding how their design decisions affect and are affected by other systems, fuel system designers can optimize their choices early in the development of an aircraft. This leads to a better understanding of the final product.
The knowledge of the ribs' positioning must be known before this function can be activated. The overlap with all the ribs is then calculated for each pump that is present in the model. An error message identifying the overlapping region and the respective rib is made available, if any of the ribs do in fact overlap with the fuel pump. When a problematic condition is identified, the `all` iterator returns the information and the iteration ends. Therefore, if more than one rib was overlapping with the component, it is possible that even after the user takes a corrective action, this function still returns a falsified value in a next reasoning.
To make the development process easier, it was decided to go through an iteration with all of the ribs for each fuel pump. This strategy has been determined to be feasible, with minimal time penalties in simulation time. If a model becomes too complex, it may be preferable to modify the function to only consider the ribs related to the fuel tank in which the pump is contained. Of course, more geometric information about the architecture is required beforehand.
Figure 5.9 illustrates a case where a fuel pump intersects one of the ribs, where the intersecting rib is colored in red, while Listing 5.10 illustrates the iteration loop through all the ribs for each pump, made possible by the codex-geometry module.

Listing 5.10: Iterative loop to find intersecting ribs with fuel system components

```
val check = ribFaces.all { rib ->
    val intersection = pumpGeo.intersect(rib)
    val intersectionArea = model.getGeometryAdapter().
getSurface(intersection)
    val checkIntersection = intersectionArea == 0.0[u.m2]
    if (!checkIntersection) {
        println("The␣element␣${pump.readableName}␣intersects␣the
␣␣␣␣␣␣␣␣rib␣#$ribNumber")
        println("␣␣␣Overlap:␣$intersectionArea")
    }
    ribNumber ++
    checkIntersection
}
```

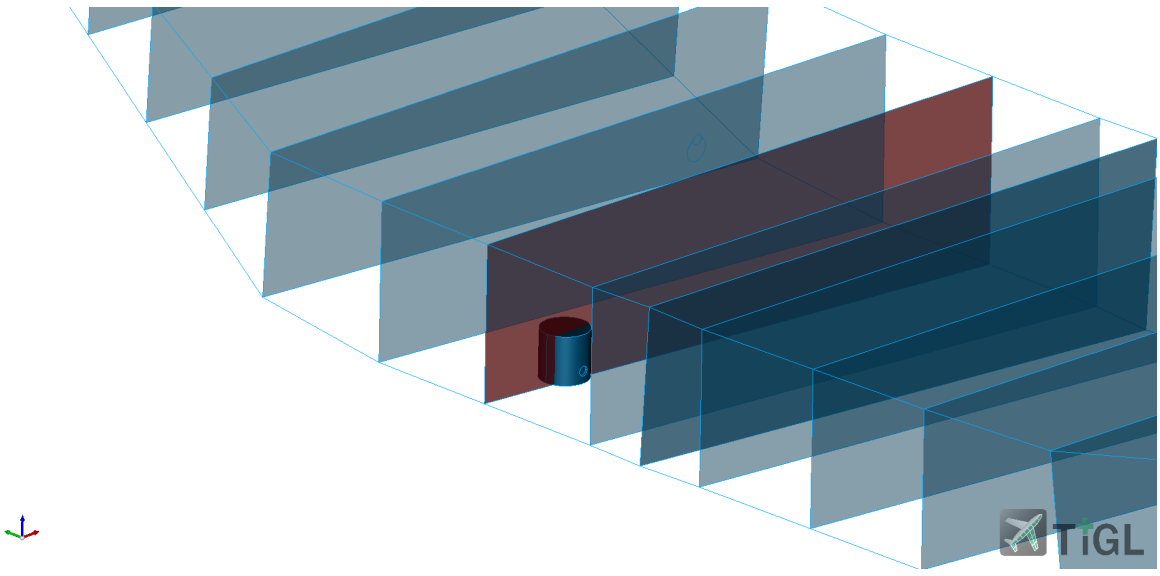

Figure 5.9: Wing rib overlapping with a fuel pump

Resulting statements

- Verified:
  `<pump, FS:isContainedBetweenRibsVerified, fuelSystem>`
- Falsified:
  `<pump, FS:isContainedBetweenRibsFalsified, fuelSystem>`

5.3.6 Tank opening geometry

Tank openings are created using two different production rule: while the first rule is responsible for creating the opening's geometry itself, the second rule is responsible for integrating the opening to the respective fuel tank. Since the opening is represented as a 2D geometric element, slight misalignments between the tank surface's normal vector and the direction vector used to create the opening could result in a mismatch between the desired opening geometry and that created on the tank's surface. In order to verify if the geometries match, a production rule was put into place.
The tank's surface is geometrically subtracted from the opening geometry using the geometry adapter provided by codex-geometry. Because "negative" geometries are not possible, if the opening is completely on one of the tank's surfaces, the remaining surface as a result of this operation should be zero. If, on the other hand, this operation yields a non-null shape, it is possible to conclude that the opening was not completely integrated on the tank's surface, and thus the desired and actual opening geometries differ. To account for slight differences due to geometric representation, a small tolerance of $1 \times 10^{-8}\,\mathrm{m}^2$ is considered.

Resulting statements

- Verified:
  `<opening, Ver:containedInVerified, tank>`
- Falsified:
  `<opening, Ver:containedInFalsified, tank>`

5.3.7 Intersection between supply lines and tanks

In fuel system design, supply lines are one of the biggest timing consuming components to design due to the need to properly route them through the design space. These lines can often assume complex shapes, with multiple turns, and can be very long, passing through multiple tanks. In order to aid the designer, a production rule capable of identifying overlaps between lines and fuel tanks was developed. In order to trigger it, it is necessary that the user sets the property `setCheckPipeTank-Intersection` between the individual belonging to the class `FS:FuelSystem` and the class of supply lines, `FS.SupplyLine`. Because the main goal of this function is to find design inconsistencies between pipes and tanks that the designer could not have identified, the rule is executed for any combination of pipe and tank. Due to the amount of iterations required, it was identifying that calling this function for every combination of supply line and tank was very time consuming and not optimized. In fact, if n is the number of declared fuel tanks and k is the number of supply lines present in the model, the amount of times this rule would be executed is given by combinatorial analysis as:

$$\#\text{function calls} = C_k^{n+k-1}$$

As an example, if there are three tanks and two pipes in the model, this function would be called six times: $C_2^{3+2-1} = C_2^4 = \frac{4!3!}{2!} = 6$. This is not ideal since nested functions called by this rule, such as the surface extraction function, would be called a large amount of times, requiring extra time and increasing significantly the problem's complexity. In an attempt to reduce the algorithm's complexity, the function was set to be called n times, one for each fuel tank. The algorithm then has to call the extraction function only once for each fuel tank, thus reducing the time required for the simulation. By exploiting the graph's properties, it is possible to extract all the supply lines belonging to the model at once in order to iterate through each of them, looking for possible intersections.
In case an intersection is found, the algorithm is capable of extracting geometrical information from it. If a pipe intersect with a tank in multiple, separate zones, the intersection will be composed by

disjoint parts, which are identified by the algorithm and then split into individual shapes, each containing their own geometries. By exploiting Opencascade's capabilities, it is the possible to compute the centroid for each intersection, which in turn can be saved to the model to give additional information on the encountered problems. **This strategy allows to add an extra layer of detail to the GeoVerification tool, where not only it provides a binary result, as passed or failed, but also additional information to aid the designer in providing corrective measures to the model.**
Whenever an overlapping condition is found, an individual representing the centroid of each intersection is created and the information is added to the model as illustrated in Listing 5.11.

Listing 5.11: Intersection centroid's definition and its addition to the knowledge graph

```
val barycenter = intersecProp.centreOfMass()
val pointInd = createPoint(barycenter[0][u.m],
    barycenter[1][u.m], barycenter[2][u.m]).individual
// add the statement to the model
model.addAssertion(pipe, FS.hasOverlapWithTankPosition, pointInd)
```

Resulting statements

- Verified:

  `<supplyLine, FS:isNotOverlappingWithTankVerified, tank>`

- Falsified:

  `<supplyLine, FS:isNotOverlappingWithTankFalsified, tank>`

  `<supplyLine, FS:hasOverlapWithTankPosition, pointIndividual>`

5.3.8 Center of gravity shift

The fuel system design has a significant impact on the overall aircraft's center of gravity (CoG) due to the large amount of fuel contained in the tanks and the complex geometries these tanks can assume. The development of a verification function capable of computing the center of gravity for a specific tank or the entire fuel system and verifying if it is within the user-specified boundaries aims to assist designers in focusing on the design and being advised by the GeoVerification tool if their choices provide a CoG shift that exceeds the requirements. Codex geometry is capable of managing the complex shapes that fuel tanks may assume and can easily provide the geometric centroid.
As a proof of concept, two separate functions were developed:

- `checkLongitudinalCoGTank`: provides verification capabilities focusing on a specific tank.
- `checkLongitudinalCoGFuelSubsystem`: provides verification capabilities that focus on all declared tanks at the same time and may be useful for configurations where tail tanks play a major role on CoG shift

Both functions are solely concerned with tank geometry and do not take into account different filling levels or aircraft attitude. To be activated, these rules require that both the upper and the lower bounds are set for the longitudinal axis by using the `hasLongitudinalCOGLowerBoundRequired` and `hasLongitudinalCOGUpperBoundRequired` properties, which link a resource (a tank or the fuel system) to a literal, representing the boundary value in meters - or any other unit representing a length. The `centerOfMass()` method is provided by Opencascade as a volume property and provides the CoG in meters. It is required that the tank - or group of tanks - have an appropriate complete shape in order to use it. Upper and lower bounds are independently verified, and a statement for each boundary is added to the graph, indicating whether or not the boundary is respected. If both the lower and the upper boundaries are respected, a single statement is added to the model stating the information.

Resulting statements

- Verified:

  `<tank, FS:hasLongitudinalCOGBoundariesVerified, centroid>`

OR

`<tank, FS:hasLongitudinalCOGLowerBoundVerified, centroid>`

`<tank, FS:hasLongitudinalCOGUpperBoundVerified, centroid>`

- Falsified:

`<tank, FS:hasLongitudinalCOGBoundariesFalsified, centroid>`

OR

`<tank, FS:hasLongitudinalCOGLowerBoundFalsified, centroid>`

`<tank, FS:hasLongitudinalCOGUpperBoundFalsified, centroid>`

Where the **"OR"** represents the possible verified and falsified statements but not the combination assumed by the model (i.e. if both boundaries are verified only the first verified statement is added, otherwise one of the two statements below it **could** be added to the model, depending on the case.)

5.4 Certification and guidelines verification

This category of verification functions includes all geometry-related checks that are also subject to certification specifications and guidelines provided by recognized organizations (e.g., EASA, SAE). Furthermore, knowledge based on engineering experience was implemented in the form of verification rules, which also fall under this category.

5.4.1 Available fuel mass

As for any fluid, aircraft fuel's behavior is subject to environmental conditions, requiring additional attention when sizing its storage compartment. EASA's CS §25.969 requires that when sizing fuel tanks, an expansion space of at least 2% of the tank's total volume shall be considered for an eventual fuel expansion. This is based on standard Jet A and Jet A-1 fuels, considering an expansion due to a temperature change from 25 °C until the first engine fuel burn, at an expansion rate of 0.8%/10 °C [1]. In addition to that, it is important to consider the integration between tanks and fuel pumps, where a portion of fuel may remain unreachable by the pump, thus accumulating in the so called sump areas. As aircraft fuel tends to have a small portion of water associated to it, this accumulation leads to water separation and sedimentation, as its density is higher than that of the fuel. The sizing of this sump area is ruled by CS §25.971, which requires that the tank's volume shall account for a sump volume of 0.25 L or 0.1% the tank's capacity, whichever is greater. Figure 5.10 illustrates the different volume definitions provided by SAE's Aircraft Fuel System Design Guidelines (SAE AIR7975) [1]. Two verification rules were implemented to determine whether the available mass meets the requirements, one focusing on the entire fuel system (all fuel storage components) and the other on individual tanks.
In order to be activated, information regarding the aircraft's fuel must be present in the model to be properly queried. As the fuel is the same for all fuel system components, its presence is linked to the fuel system itself and not to the components. Listing 5.12 illustrates the "when" side for the production rule responsible for computing the available mass in the fuel system. It only differs from the second production rule by the first query, where instead of having an individual of class `FS:FuelSystem` with a design space (or shape), we have an individual of class `FS:FuelStorage`, a fuel tank. As soon as the relevant information related to the fuel is declared (i.e. its density) and the mass requirements are found in the knowledge graph respecting the patterns requested, the production rule takes place. Using the geometry adapter it is possible to compute the volume of each tank or the complete tank arrangement, depending on which rule is triggered. Then, the volume required for the fuel expansion and that needed by the surge are computed as follows:

$$V_e = 0.2 * V$$
$$V_s = max(0.25\,\mathrm{L}, 0.1 * V)$$

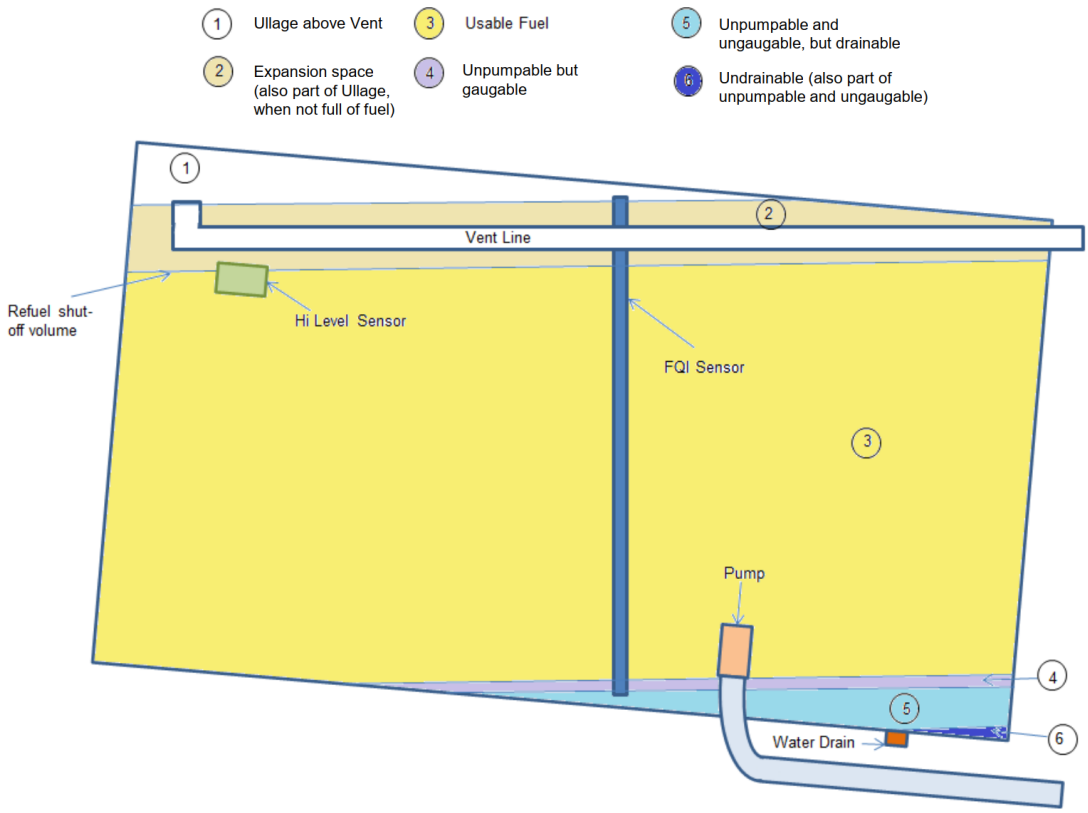

Figure 5.10: Tank volumes as defined by SAE [1]

Where V_e and V_s are the expansion volume and the surge volume, respectively, and V is the total tank volume computed with the `getVolume()` method. The available volume thus become

$$V_{\text{available}} = V - V_e - V_s$$

and the available fuel mass, m_f, is

$$m_f = \rho_f * V_{\text{available}}$$

With ρ_f the fuel's density. The calculated available mass is then compared with the mass requirement set by the user, using the `FS:hasMassRequired` parameter property, and the check results successful whenever the computed mass is equal or higher than the one set by the user.

Listing 5.12: When side for the verification function responsible for checking the available fuel volume and mass for the fuel system

```
When<SIndividual, SIndividual, SRDFResource, SRDFLiteral, SRDFLiteral>
{fuelSystem, designSpace, fuel, mass, density ->
    +(fuelSystem..FS.hasDesignSpace..designSpace)
    +(fuelSystem..FS.hasFuelType..fuel)
    +(fuel..FS.hasDensity..density)
    +(fuelSystem..FS.hasMassRequired..mass)
    +(fuelSystem..a..FS.FuelSystem)
} Do {...}
```

On the logger screen, a message informing the user of the available and required mass is displayed, and the resulting statements are added to the graph. The resulting statements differ only by subject, depending on whether the subject was the fuel system or the tank that had its mass evaluated.

Resulting statements

- Verified:

  - Fuel system:
    `<fuelSystem, FS:hasMassVerified, mass>`
  - Tank:
    `<tank, FS:hasMassVerified, mass>`
- Falsified:
  - Fuel system:
    `<fuelSystem, FS:hasMassFalsified, mass>`
  - Tank:
    `<tank, FS:hasMassFalsified, mass>`

5.4.2 Minimum distance between pipes and fuel tanks

Integral fuel tanks use the already existent aircraft structure as boundaries for their design space. When designing the fuel system, in particular the pipe routing, it is advisable to maintain a minimum distance between it and the surrounding structure and other components and subsystems [1]. A minimum distance of 0.25 inch is considered to be acceptable according to SAE AS18802 [48] without needing to demonstrate possible separation via clamping, which could be verified only at advanced design stages.
Two production rules were developed to give the user more options when defining this requirement. A first production rule allows the minimum distance between the pipe and the tank as a whole to be verified. However, if pipes crossing multiple tanks' boundaries are expected, their minimum distance with the tank's surface will most likely be less than 0.25 inches, unless an opening at least 0.25 inch larger than the tank's radius is also provided. To cope with that, a second production rule, able of dealing with the tank's sides individually, was implemented. Assuming wing tanks to have a hexahedral shape, six properties are available, where the directions are taken with respect to the aircraft's fuselage considering the front to be pointing towards the aircraft's nose (towards negative X axis). Figure 5.11 shows the reference frame provided by TiGL viewer and how it relates to the aircraft representation. The fuel tank sides are defined using TiGL's frame as follows:

- `hasMinimumDistanceToFrontRequired`: front side points towards the negative X axis.
- `hasMinimumDistanceToRearRequired`: rear side points towards the positive X axis.
- `hasMinimumDistanceToTopRequired`: top side points towards the positive Z axis.
- `hasMinimumDistanceToBottomRequired`: bottom side points towards the negative Z axis.
- `hasMinimumDistanceToLeftRequired`: left side points towards the negative Y axis.
- `hasMinimumDistanceToRightRequired`: right side points towards the positive Y axis.

If the complete tank is to be considered for the analysis, the following property can be set

`FS:hasMinimumDistanceToRequired`

In order to activate this rule, it is necessary to define the combination of tanks and supply lines to be verified by means of the above mentioned properties. These properties take as the subject the supply line and as object the tank individuals.
The standard minimum distance of 0.25 inches can be changed according to the user's needs by using the property `FS:hasMinimumDistancePipesTanks`. If this property linking the individual of class `FS:FuelSystem` and the distance is not found in the graph, the standard value is used. Codex-quantity supports the algorithm on providing the correct conversion between units, if conflicts arise. After setting the correct tolerances, the reference surfaces are extracted from the tank's geometry. The "hasMinimumDistanceTo...Required" properties, each of which corresponds to a specific reference vector, define which surfaces are supposed to be extracted. Once these surfaces are extracted, they are sewed again into a single shape using the `BRepBuilderAPI_Sewing()` provided by Opencascade. This strategy allows to unite all the single pieces of surface that were extracted in order to evaluate only once the minimum distance. The geometry for the pipes is extracted and then the minimum

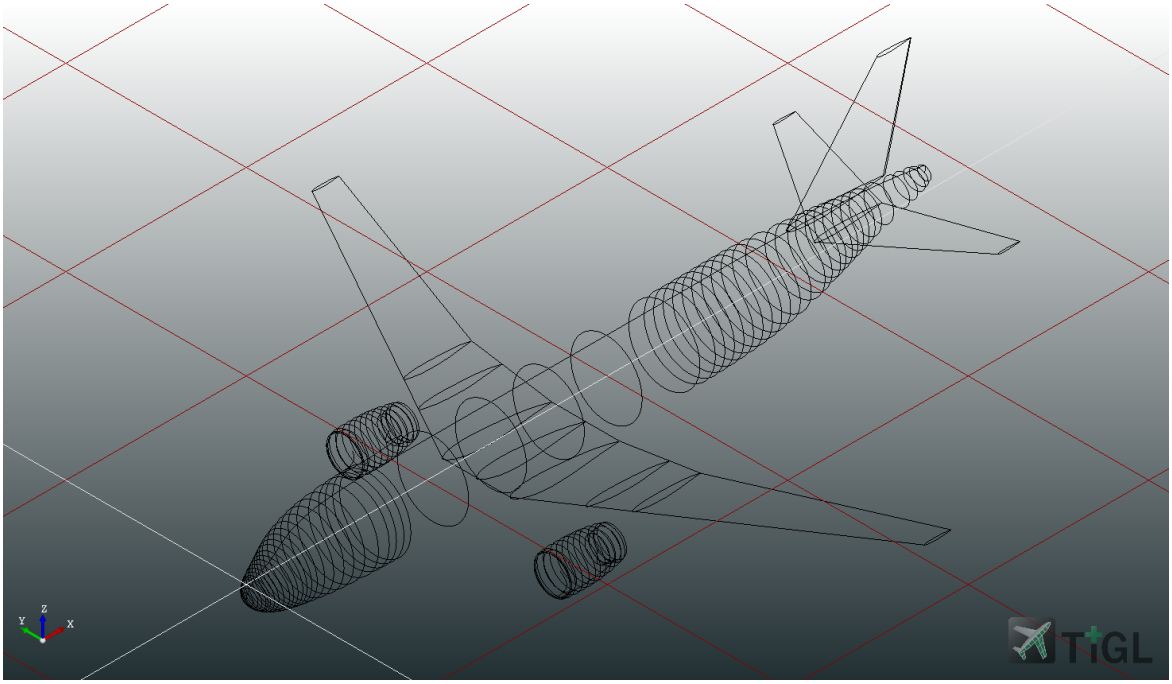

Figure 5.11: Aircraft representation on TiGL's main frame of reference

distance is computed, as illustrated in Listing 5.13. Because codex-geometry does not yet provide integrated methods for performing surface loft computations, it was decided to proceed with the implementation using the methods provided directly by Opencascade's geometry kernel.

Listing 5.13: Algorithm to compute the minimum distance between a tank's external surfaces and a supply line

```
val tankGeo = OpencascadeConnector(model).
    getOrCreateShape(designSpace.getShape())
val pipeGeo = OpencascadeConnector(model).
    getOrCreateShape(pipe.getShape())
val distanceFunction = BRepExtrema_DistShapeShape(tankGeo,
    pipeGeo)
distanceFunction.Perform() // compute the distance
// get the minimum distance between tankGeo and pipeGeo
val distance = distanceFunction.value()[u.m]
val check = distance >= minimumDistance
```

If the calculated distance is less than the reference distance, the check returns the verified statements. A visual output is also provided to the user on the logger's screen, stating the minimum distance taken as reference and the algorithm's computed minimum distance. The two production rules differ only by the surfaces to be considered during the process and the resulting statements. Whereas only one type of statement is required for the first production rule, the second production rule requires a clear definition of which side of the tank respects or does not respect the minimum distance requirement. To accomplish this, six distinct properties were defined for both verified and falsified statements. The following is a list of possible resulting statements. It is important to remember that in order to verify multiple tank surfaces (top, rear, etc...), multiple statements must be set, activating the production rule multiple times and thus providing separate statements to each analysis run.

Resulting statements

- Verified:
  - Tank:

    `<supplyLine, FS:hasMinimumDistanceToVerified, tank>`

  - Tank's surface:

```
<tank, FS:hasMinimumDistanceToFrontVerified, tank>
<tank, FS:hasMinimumDistanceToRearVerified, tank>
<tank, FS:hasMinimumDistanceToTopVerified, tank>
<tank, FS:hasMinimumDistanceToBottomVerified, tank>
<tank, FS:hasMinimumDistanceToLeftVerified, tank>
<tank, FS:hasMinimumDistanceToRightVerified, tank>
```

- Falsified:
  - Tank:

```
<supplyLine, FS:hasMinimumDistanceToFalsified, tank>
```

  - Tank's surface:

```
<tank, FS:hasMinimumDistanceToFrontFalsified, tank>
<tank, FS:hasMinimumDistanceToRearFalsified, tank>
<tank, FS:hasMinimumDistanceToTopFalsified, tank>
<tank, FS:hasMinimumDistanceToBottomFalsified, tank>
<tank, FS:hasMinimumDistanceToLeftFalsified, tank>
<tank, FS:hasMinimumDistanceToRightFalsified, tank>
```

5.4.3 Redundant components

An important aspect when designing the fuel system is providing means to accommodate failures that may arise throughout the aircraft's life cycle. Among different failure management strategies, the redundancy of a component is widely applied by the industry when it comes to the fuel system. A main requirement leading the development of this type of analysis is provided by CS §25.903 [24] and AMC 128-A [17] regarding an engine rotor burst.
If a rotor burst may affect critical components such as engine feed pumps, AMC 128-A requires that the component be duplicated and adequately spaced as an acceptable means of compliance. ETOPS certification requirements, in addition to rotor burst requirements, call for crossfeed and valve redundancy when dealing with twin-engine aircraft to perform ETOPS-certified operations, as stated in the AMC 20-6 [17].These are just a few examples of how using the GeoVerification tool to verify a component's redundancy can be beneficial.
Although easily verified by humans, the verification of redundant components becomes more difficult when modeled in semantic-web optics, since counting operations using OWL (Web Ontology Language) [41] are impacted by the "Open World Assumption". According to this assumption, due to the dynamic and distributed characteristics of the knowledge graph (or semantic webs), it is not possible to assume that, at any time, all information is available or that the graph contains all the necessary and existent knowledge on a certain resource [42]. Semantic webs rarely permit making absolute assumptions, such as the case of counting elements, unless it is possible to precisely state it, because new information may be determined by inference or manually added to the graph. Although the model is said to have multiple components, this logic forbids making assumptions about their uniqueness because it is impossible to predict whether or not new information will ultimately show that the components are the same. To cope with it, OWL provides means of explicitly declaring that two elements are not the same through the `differentFrom` property.
Redundancy verification rules were developed for fuel pumps and valves. The assertion of a component's individuality requires two steps, implemented in two separate functions:

1. component's individuality verification
2. component's redundancy verification

This strategy has been implemented separately for both fuel pumps and fuel valves, for a total of four distinct production rules.

Component's individuality verification

In order to respect the Open World Assumption, the first step in counting elements is to provide a way to check whether components can be explicitly defined as different. Listing 5.14 illustrates the "when" side for the function `checkPumpIndividuality`. The distinctive statements responsible for activating this function, and thus required to be added to the model, depend on the existence of the property `FS:hasDistinctIndividualsRequired` linking a fuel tank to the class of components in question (i.e. the fuel pump class) and on the **non-**existence of the property `FS:hasDistinctIndividualsVerified`. While the first is required to enable the check, the latter allows for time optimization during the reasoning. In fact, in the absence of the latter, this rule would be activated once for each pump present in this tank. As opposed to this, because the algorithm's main objective is to determine whether or not the available pumps differ from one another, once this rule is applied it is applicable to all pumps.

Listing 5.14: When side for the checkPumpIndividuality function

```
When<SIndividual, SIndividual> {pump, tank ->
    +(pump..a..FS.FuelPump)
    +(tank..a..FS.FuelStorage)
    +(tank..FS.hasDistinctIndividualsRequired..FS.FuelPump)
    +(pump..Ver.containedInRequired..tank)
    +not(tank..FS.hasDistinctIndividualsVerified..FS.FuelPump)
} Do {...}
```

When the query identifies resources on the graph matching the desired patterns, the algorithm proceeds extracting all the individuals belonging to the requested class (fuel pumps or fuel valves) and filtering them by the `Ver:containedInRe-quired` property linking the components to the tank in analysis. It then proceeds looking for the `owl:differentFrom` property between every pair of components belonging to the filtered individuals. If the property is shared by every pair of elements, it can be assumed that each element (component) in the sought-after class and present in the required tank is a unique individual. Instead, if this claim does not hold for at least one pair of components, the Open World Assumption makes it impossible to state that these two components are distinct elements, which compromises the counting abilities. Lastly, the algorithm alters the graph by adding the statements: one for each pump, stating its individuality or indetermination, and one for the tank, stating if all components of a same class belonging to this fuel tank are distinct individuals. This last statement does not have a falsified correspondent, meaning that, if the check turns out to fail, no statement saying the tank does not have distinct individuals is provided. This follows the Open World Assumption, as new knowledge saying otherwise may appear at any moment.
Listing 5.15 illustrates how it is possible to extract the individuals belonging to the `FS:FuelPump` class who also belong to the designated fuel tank using Kotlin embedded iteration functions and filtering capabilities. The method `individualsOf` allow to extract all individuals present in the model belonging the same class while `hasProperty()` allows to correctly filter them according to the desired property, where the individual in question is the subject of a statement.

Listing 5.15: Knowledge extraction and filtering in Kotlin

```
val pumpIndividuals = model.individualsOf(FS.FuelPump).
filter { element ->
    element.hasProperty(Ver.containedInRequired, tank)
}.toMutableSet() // get all the declared pumps within a
//fuel tank
pumpIndividuals.remove(pump)
```

Component's redundancy verification

Only after the individuality has been asserted it is possible to proceed with counting the elements and verifying whether they meet the redundancy requirements. Listing 5.16 shows the "when" side for the `checkPumpRedundancy` function. It is clear how this rule is triggered if, and only if, the fuel tank gained the `FS:hasDistinctIndividualsVerified`, which is provided by the previous function.

Listing 5.16: When side for the checkPumpRedundancy function

```
When<SIndividual, SRDFLiteral> {tank, minPumpNo ->
    +(tank..a..FS.FuelStorage)
    +(tank..FS.hasMinimumPumpNumberRequired..minPumpNo)
    +(tank..FS.hasDistinctIndividualsVerified..FS.FuelPump)
} Do {...}
```

The `FS:hasMinimumPumpNumberRequired` linking a fuel tank to a literal is required to be set by the user in the model, whenever needed. Although not used in this project, a different approach might be to define the requirements that call for component redundancies and then add the statements to the model through inferences (for example, if a user specifies that ETOPS requirements must be met, it is simple to infer through a production rule that a redundant crossfeed line, with redundancies on the crossfeed valve, are necessary). Using a similar approach as seen in Listing 5.15, a new list of components containing the property `FS:isDistinctIndividualVerified` with the fuel tank is extracted. Having already identified the different components, the counting is straightforward in Kotlin. The resulting value is then compared with the one set by the user through the hasMinimumPumpNumberRequired, or hasMinimumValveNumber, and a boolean check is provided stating the result of the operation. Lastly, the consequent statements are added to the graph.

Resulting statements

- Verified:
  - Component's individuality:

    `<component, FS:isDistinctIndividualVerified, tank>`

    `<tank, FS:hasDistinctIndividualsVerified, FS.FuelPump>`

    `<tank, FS:hasDistinctIndividualsVerified, FS.FuelValve>`
  - Component's redundancy:

    `<tank, FS:hasRedundantPumpsVerified, individuals>`

    `<tank, FS:hasRedundantValvesVerified, individuals>`
- Falsified:
  - Component's individuality:

    `<component, FS:isDistinctIndividualFalsified, tank>`
  - Component's redundancy:

    `<tank, FS:hasRedundantPumpsFalsified, individuals>`

    `<tank, FS:hasRedundantValvesFalsified, individuals>`

5.4.4 Shutoff valves before engine adapter

According to CS §25.903 [24], segregation means are necessary to provide engine isolation if failures arise. In the event of an engine malfunction, pilots may be able to isolate the engine by turning it off and preventing fuel from being fed. This requirement can be met by incorporating shutoff valves as a last-resort resource at the interface between the fuel system and the aircraft's powerplant. In order to verify the presence of a shutoff valve before the engine adapter, a production rule was developed. This rule can be activated whenever an engine adapter has been declared to the model and is linked to a feed line. The algorithm looks for patterns on the knowledge graph where a connection between the queried supply line, which is connected to the engine adapter, is also connected to a fuel valve. Provided the following pattern of statements exist, the rule successfully returns the connection as a visual output to the logger.

`<supplyLine, Ver:connectedToVerified, connectionPort>`

5.4.5 Relative position between valves and lines

Using the GeoVerification tool it is possible to leverage geometric knowledge to provide additional information about possible design mistakes that may greatly affect the system's performance. Although aircraft fuel has a high level of purity, the presence of water cannot be completely excluded [17][1][1]. Led by ETOPS requirements concerning a crossfeed valve failure due to icing, a verification rule was developed to study possible water accumulation points in the design. To be triggered, the rule searches the graph for patterns where the connection between valves and lines has been declared, as well as information on which connection port belonging to the valve the line is connected to, as shown in Listing 5.17. Furthermore, the geometric knowledge regarding the components' positioning is required.

Listing 5.17: When side for the checkRelativePositionSOVLines function

```
When<SIndividual, SIndividual, SIndividual, SIndividual>
{valve, line, port, point ->
    +(valve..Ver.connectedToRequired..line)
    +(line..FS.isConnectedToPort..port)
    +(port..FS.hasPoints..point)
    +(valve..FS.hasConnectionPort..port)
    +(valve..a..FS.SOValve)
    +(line..a..FS.SupplyLine)
    +(valve..a..GC.Complete)
    +(line..a..GC.Complete)
    +(point..a..GKM.Point)
    +(point..a..GC.Complete)
} Do {...}
```

The logic steps behind this rule are depicted in Figure 5.13. The algorithm begins by obtaining geometric information about the supply line, as well as the points that form it and the location of the connection port. Afterwards, a simple search between both extremities of the supply line, in order to find the one closest to the connection port, is performed, using basic euclidean norm to calculate the distance between the line's point and the connection port itself. Since a connection is expected between one of the line's endpoints and the valve, a small tolerance of 1×10^{-8} m is given to the distance. If both extremities do not meet this tolerance, an error is provided and the simulation ended.
After finding the closest point to the connection port, the points that constitute the supply line are ordered with the first being that closest to the port itself. For each point, the relative difference on height is computed with respect to the start point, P_0, as follows:

$$\text{relative height} = \frac{P_{i_z} - P_{0_z}}{\|P_{0_z}\|}$$

With P_i, $i = 1..n$ the points in the list of points defining the pipe's geometry of dimension $n + 1$.
Three main outcomes are expected from this iteration cycle:

1. Relative height >= 2%: the starting point can be considered in a low position with respect to the supply line, as at least one segment in the line blocks the fuel in a stationary condition (when no pumps are actively providing the fuel flow).

2. Relative height <= −2%: the valve is assumed to be in a high position. In fact, considering an aircraft in leveled flight, if the respective fuel pump stops, it is possible to rely on gravity to passively distance the water formation away from the valve.

3. Relative height between −2% and 2%: no low or high position can be inferred, requiring the evaluation of more segments on the line.

The 2% tolerance can be manually changed by the user if necessary. In cases 1 and 2, the algorithm stops searching through the line once a condition is identified since only one favorable (high) or unfavorable (low) position is necessary to evaluate the requirement. The user is given more details about the location where the condition was discovered via the logger terminal. Case 3 on the other hand makes the assumption that the line is flat up until the i-th point, needing a closer examination of its geometry. The supply line is assumed to be completely flat and not to be in an icing-prone state

if the iteration process fails to identify a low or high position for the valve (cases 1 and 2). The small amount of water that might be present in the line will be distributed throughout this line without a significant concentration close to the valve that could cause ice formation.
Figure 5.12 depicts the concept of low and high valve position. It is possible to see that the supply line on the left (in red) develops in such a way that it may lead to water accumulation close to the valve, whereas the supply line on the right, in green, is designed in such a way that water can be brought to a lower position, away from the valve, to avoid its accumulation.

Resulting statements

- Verified:

  `<valve, FS:isNotProneToIcingVerified, supplyLine>`

- Falsified:

  `<valve, FS:isNotProneToIcingVerified, supplyLine>`

  `<supplyLine, FS:hasOverlapWithTankPosition, pointIndividual>`

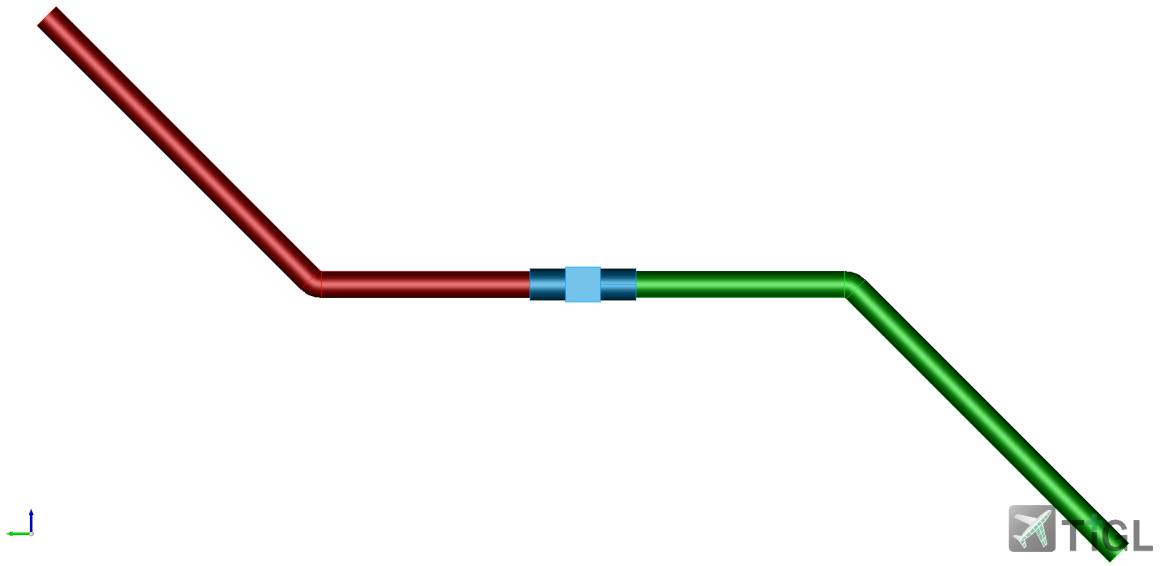

Figure 5.12: Fuel valve connected to two supply lines

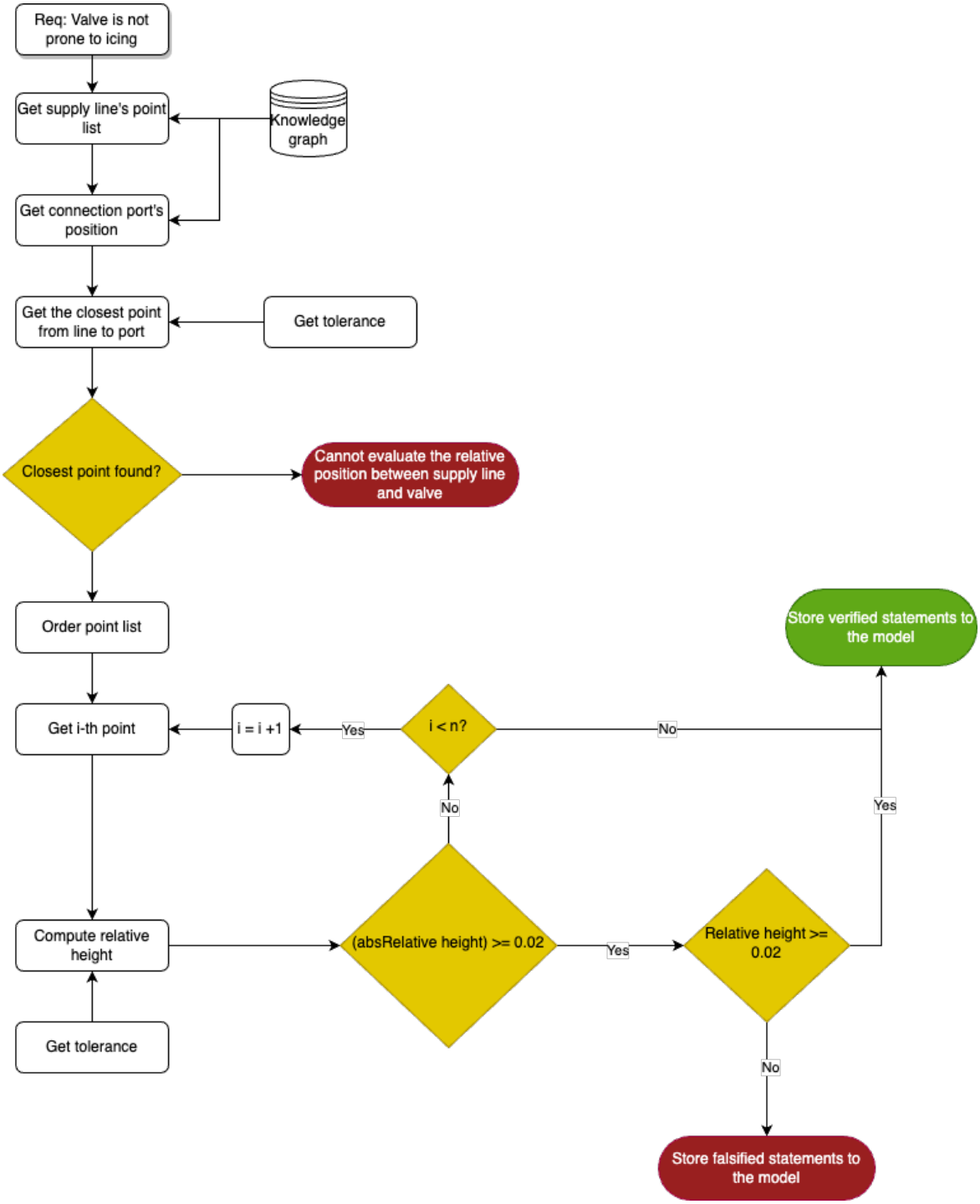

Figure 5.13: Logic workflow to verify the relative position between a fuel valve and its connecting supply lines

5.4.6 Rotor burst

Because aircraft requires large amounts of fuel to fly, the fuel storage position and geometry, together with the components placed within, pose additional constraints to the design. Although engine manufactures have made significant improvements to reduce uncontained rotor burst, this failure mode, however unlikely, continues to occur. According to CS §25.903(d) [24], it is necessary to take precautions during the design phase to mitigate, or at least minimize, the effects of an impact due to an uncontained burst associated to the engines (UERF) or, less likely, to the Auxiliary Power Unit. Due to the high speed rotating components of these devices, the enormous amount of kinetic energy

that can be released by their fragments has the potential to seriously damage the aircraft's structure by cutting into the fuselage, the fuel tanks, and causing damage to the fuel pumps, valves, and lines. The fuel tank's design is severely constrained by the potential harm caused by a rotor burst, so two main tactics are used to lessen its effects. Following the identification of areas in the fuel tank that may be harmed by a rotor's fragment, dry bays can be considered. These areas of "dead space" can be placed on the aircraft's structure in such a way that, if this failure mode occurs, fragments won't be able to hit the fuel tanks because the dry bay is set up to cover potential hit zones, as illustrated in Figure 5.14. The aircraft remains damaged but its critical components remain intact and an emergency landing is not compromised by the fuel system [2].

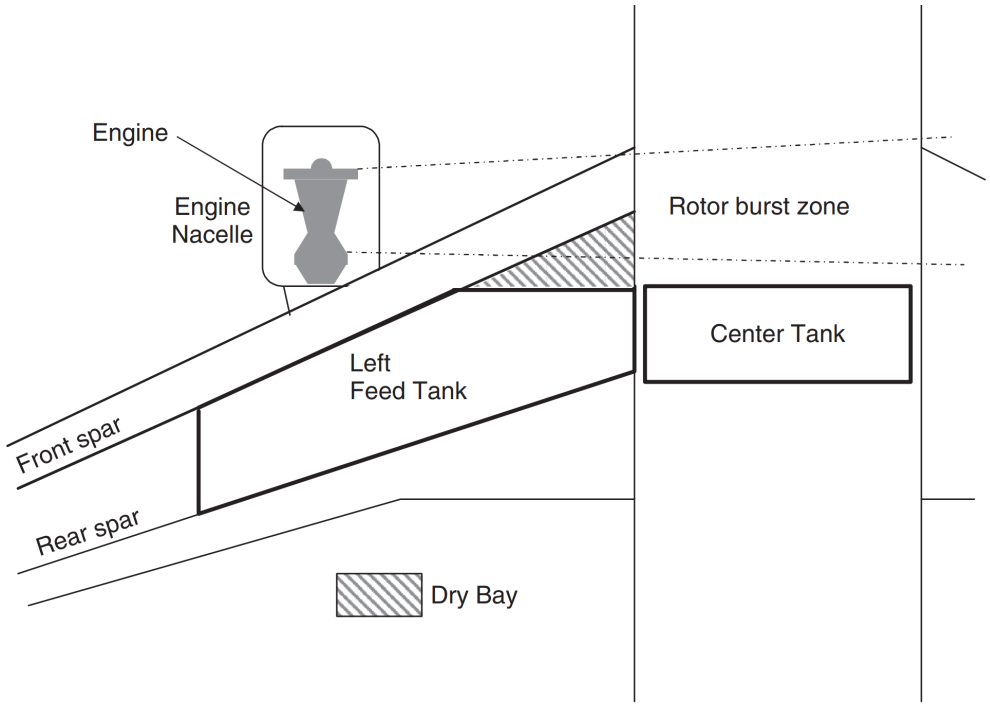

Figure 5.14: Uncontained rotor burst example for a wing-mounted engine [2]

Whenever the creation of dry bays is not possible due to the large amount of fuel to be stored, segmenting the tank on the hit zone can be done. Figure 5.15 shows the fuel tanks geometry for an Airbus A330. It is possible to see the division of the inner wing tank into two. The smaller section, responsible for the feed, is placed in an area where an uncontained rotor failure could potentially damage the tank. Following this segmentation strategy, if this failure mode happens and affects the aircraft's structure, compromising the fuel tank, it is possible to isolate the affected portion of the tank without losing the system functionalities by means of a split valve. If the rotor burst affecting the tank was due to the opposite engine, it is still possible to feed the engine that was fed by the compromised tank through the crossfeed valve, as can be seen in Figure 5.16. By enabling the crossfeed valve, engine feed is still possible by the remaining feed system. In addition to providing means to adequately feed the engines after a rotor burst, it is necessary to foresee during design strategies to transfer fuel to account for fuel imbalance that could reduce in-flight stability [2].
According to AMC-128A, there are four main failure modes foe engines: three for the main rotor and one the fan. Table 5.1 depicts these failure modes with the respective spread angles. The fragment spread angle is defined as the angle measured fore and aft from the center where the plane of rotation lies, for an individual rotor stage, initiating at the engine (or APU) shaft centerline [17].
AMC-128A states that knowledge of the hit zones and subsequent changes to the design to account for the failure are necessary for acceptable means of compliance that can ensure airworthiness following an uncontained burst failure mode. Changes to the tank's design space, the addition of dry bays or tank segmentation, adequate pipe rerouting, extensive study on the impact of placement of a critical component on failure mitigation, component redundancy, and adequate spacing between redundant components are a few of the strategies designers may employ.

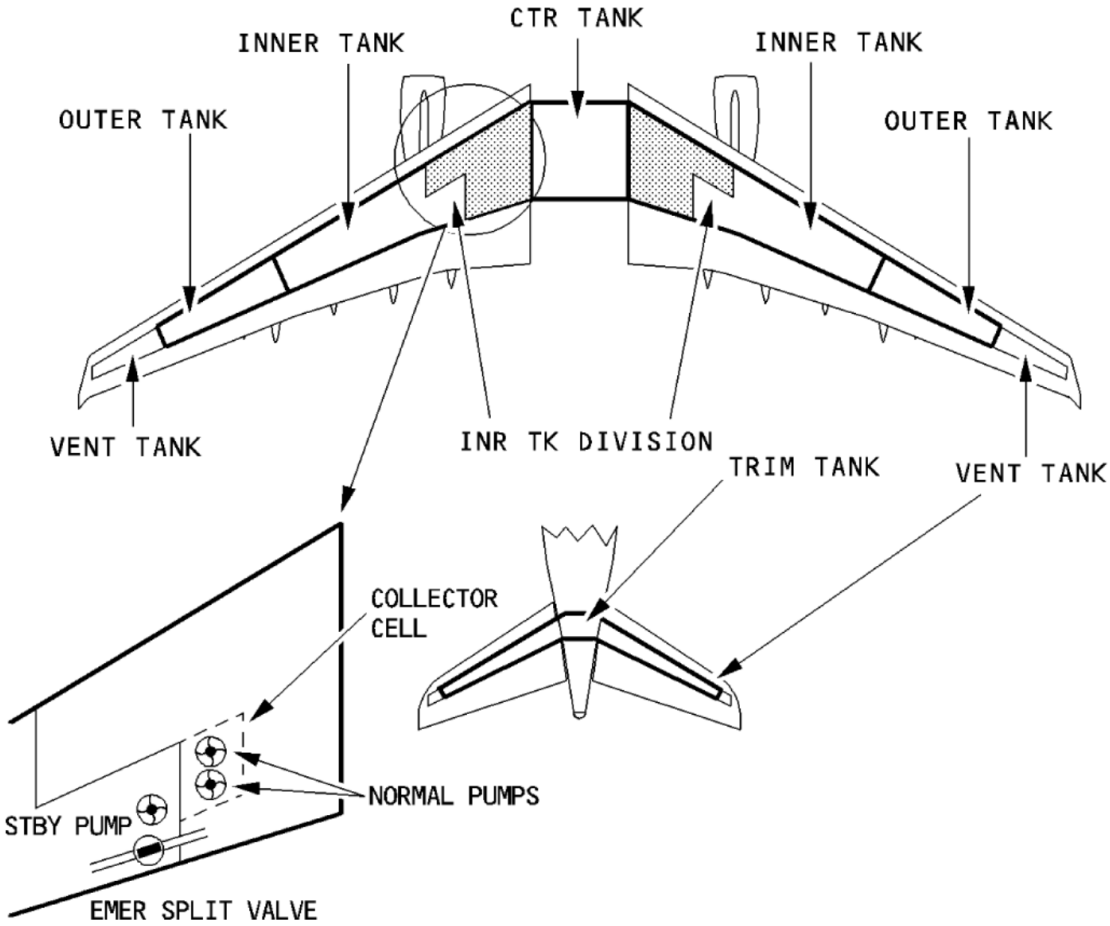

Figure 5.15: A330 - Fuel tanks [16]

Three of the four primary failure modes have been developed and integrated into the GeoVerification tool to help engineers better understand the impact of a rotor burst on their design. In order to provide the user with better control over the tool, three different properties were created, each responsible for triggering a specific failure mode analysis. The correct use of these properties is provided below:

```
<fuelSystem, FS:setCheckSmallFragmentRotorBurst, rotor>

<fuelSystem, FS:setCheckMediumFragmentRotorBurst, rotor>

<fuelSystem, FS:setCheckFanRotorBurst, rotor>
```

Since the implemented logic for each function employs a similar methodology and the geometry of the fragment being considered and the spread angle are the only variables that differ, the logic of the algorithm is only described once for the first function. The remaining functions provide additional information about the geometry of the fragment under consideration, but their implementation is unchanged.

Single one-third disk fragment

The rules pertaining to a rotor burst are implemented through a different rule module called FuelSystemUncontainedRotorBurstCheckRules, which is different from where the other verification rules are implemented. The installation of various rule packages for a project is shown in Listing 5.18, demonstrating the excellent modularity provided by Codex. It is possible to see how the rotor burst rules and the geometry verification rules come from different packages. Although it is not necessary for the rules to be divided into modules for proper operation, doing so enables focused analysis and

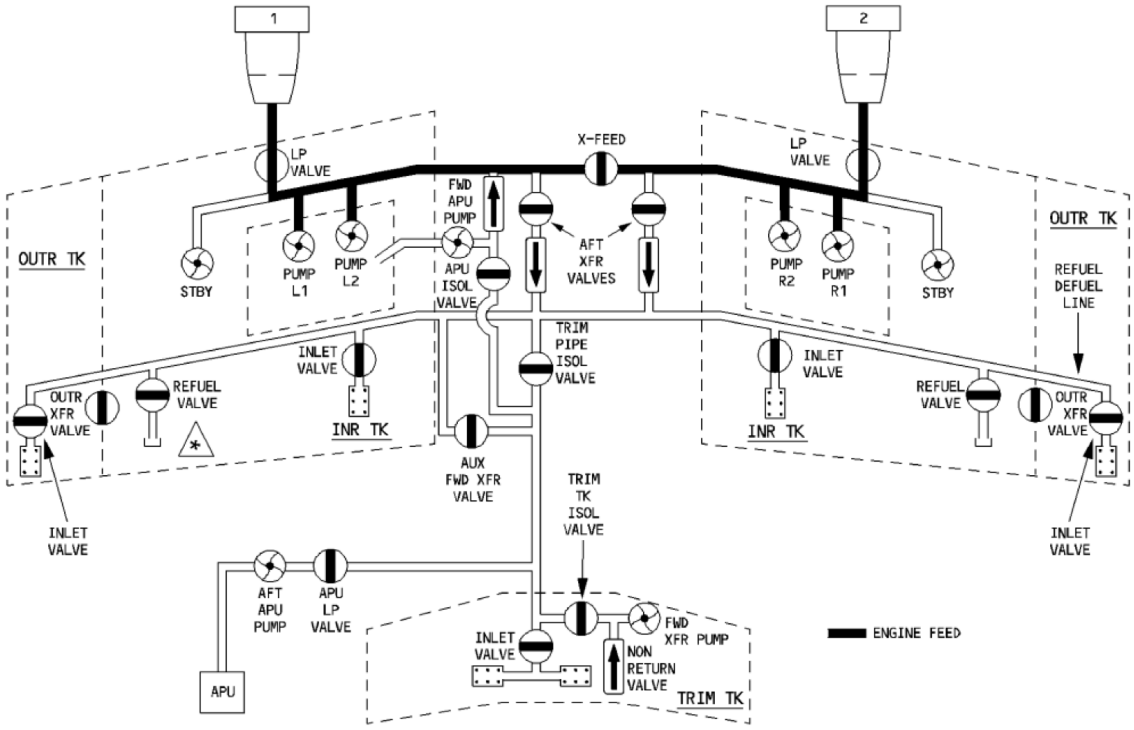

Figure 5.16: A330 - Fuel system architecture [16]

Maximum fragment size	Rotating element	Spread angle	Energy
1/3 disk fragment	Rotor	±3°	Infinite
Intermediate fragment - 1/3 bladed disk radius	Rotor	±15°	Finite
Small fragment - tip half of the bladed foil	Rotor	±15°	Protection by the primary structure
1/3 blade airfoil height	Fan	±15°	

Table 5.1: Uncontained rotor failure - impact zones

cuts down on simulation time. Additionally, this modularity enables the sequential verification of various design elements. One can, for instance, divide the design process into different steps, with the first step containing only the rules required for the creation of the geometry, and the rule set needed to carry out the geometry verification being installed separately after the creation.

Listing 5.18: Rule registering using Codex

```
install(RuleEngine) {
    registerRules(FuelSystemProductionRules)
    registerRules(GeometryCompletenessOntology.CompletenessRules)
    registerRules(GeometryCheckRules)
    registerRules(FuelSystemGeometryCheckRules)
    registerRules(FuelSystemRotorBurstCheckRules)
}
```

In order to be activated, the rule requires geometric information on the rotor, such as the rotor disk diameter, the original diameter and its positioning. It is considered that rotors have their rotation plane in the YZ axis, perpendicular to the aircraft's longitudinal axis. In addition, information regarding the class of the rotation object is needed. In the fuel system knowledge module, two different classes are provided:

1. `FS:EngineRotor`: any of the engine's rotors besides the fan.

2. `FS:EngineFan`: due to the large size of its blades compared to the disk, engine fans are considered as a separate class of rotor elements.

This rule requires a rotor element of type `EngineRotor` to be activated. Additionally, information regarding the fuel system critical components must be given. In order to assert a component is critical, a class assertion can be added to each individual containing the `FS:isCriticalComponent` class. In Listing 5.19 an example of adding such statement to components of class `FS:FuelPump` is given.

Listing 5.19: Example of adding a class assertion to multiple components in Codex

```
// add the assertion of critical component
assertedModel.individualsOf(FS.FuelPump).forEach {
    assertedModel.addClassAssertion(it, FS.CriticalComponent)
}
```

Users have the possibility to change some of the function's parameters such as the spread angle by adding additional statements to the model. Three parameters accept manual override through the statement pattern

`<fuelSystem, property, value>`

1. Spread angle ψ: defines the fore and aft angle made with the rotation plane. Set with `FS:hasSpreadAngle`.

2. ψ resolution: sets the sampling interval for the spread angle ψ. Set with `FS:hasPsiResolution`.

3. ϕ resolution: this angle sets the sampling interval for the translational angle $\phi = [0, 2\pi]$. Set with `FS:hasPhiResolution`.

The algorithm looks for information regarding the aforementioned values and using the codex-quantity capabilities, converts them to degrees, in case the information was given in radians. If no information can be found on the graph, standard values are provided (see Table 5.2).

Parameter	**Value**
ψ	According to the fragment type and size. See Table 5.1.
ψ resolution	1.0°
ϕ resolution	5.0°

Table 5.2: UERF - Standard values for the reference angles

For every critical component defined to the model, the function continues creating a discrete map of angles for both the translational plane, or rotation plane, and the spread plane, normal to the rotation plane, according to the defined resolution. Figure 5.17 illustrates both the translational angle ϕ and the spread angle ψ and how they interact with the target (critical component). For every combination of discrete ϕ and *psi* angles, a trajectory path is created using the fragment's geometry. As a first step for defining the trajectory path, the fragment's geometry is defined. For the single one-third disk, the following is defined:

$$b = \frac{D - d_R}{2} H_{reduced} = (D + \frac{2b}{3})$$
$$D_{centriod} = H_{reduced} * \cos(30°)$$
$$Rl_{centroid} = \frac{H_{reduced}}{2} * \cos(60°)$$

Where b is the blade length, d_R the rotor disk diameter, D the original diameter, $H_{reduced}$ the reduced height diameter, $D_{centroid}$ the fragment's diameter from its centroid and $Rl_{centroid}$ the fragment's radius from its centroid. For each value of translational angle ψ, the fragment's locus point is created:

$$locus = (PR_x, PR_y + Rl_{centroid} * \cos\phi, PR_z + Rl_{centroid} * \sin\phi) \tag{5.2}$$

Where PR_x, PR_y, PR_z are the x, y and z coordinates for the rotor's position. In order to create the trajectory, first a trajectory shape defining the reach the burst may have is created on the rotation

plane YZ. This trajectory is considered to be quadrilateral, starting at the locus point. The definition of the trajectory's reach is done once for the fragment and reused to map the complete probable hit zones. The four points defining its shape are defined as follows:

$$\begin{aligned} P_1 &= (PR_x, locus_y, locus_z + D_{centroid}/2) \\ P_2 &= (PR_x, 100, locus_z + D_{centroid}/2) \\ P_3 &= (PR_x, locus_y, locus_z - D_{centroid}/2) \\ P_4 &= (PR_x, 100, locus_z - D_{centroid}/2) \end{aligned} \tag{5.3}$$

Since the analysis focus only on the aircraft, a radius of 100 meters was defined more than enough to compute the burst's reach. For each translational angle ϕ mapped, the trajectory a rotation pf ϕ is performed taking as center of rotation the locus point and the axis of rotation the X axis. In addition, due to its rotating characteristics, a tangential exit is expected for the fragment, leading to an additional rotation of $\pm 90.0°$. This additional rotation is defined by an extra parameter called `rotationDirection`. If not declared to the model, a standard rotation of $+90.0°$, corresponding to a clockwise rotation, is performed.

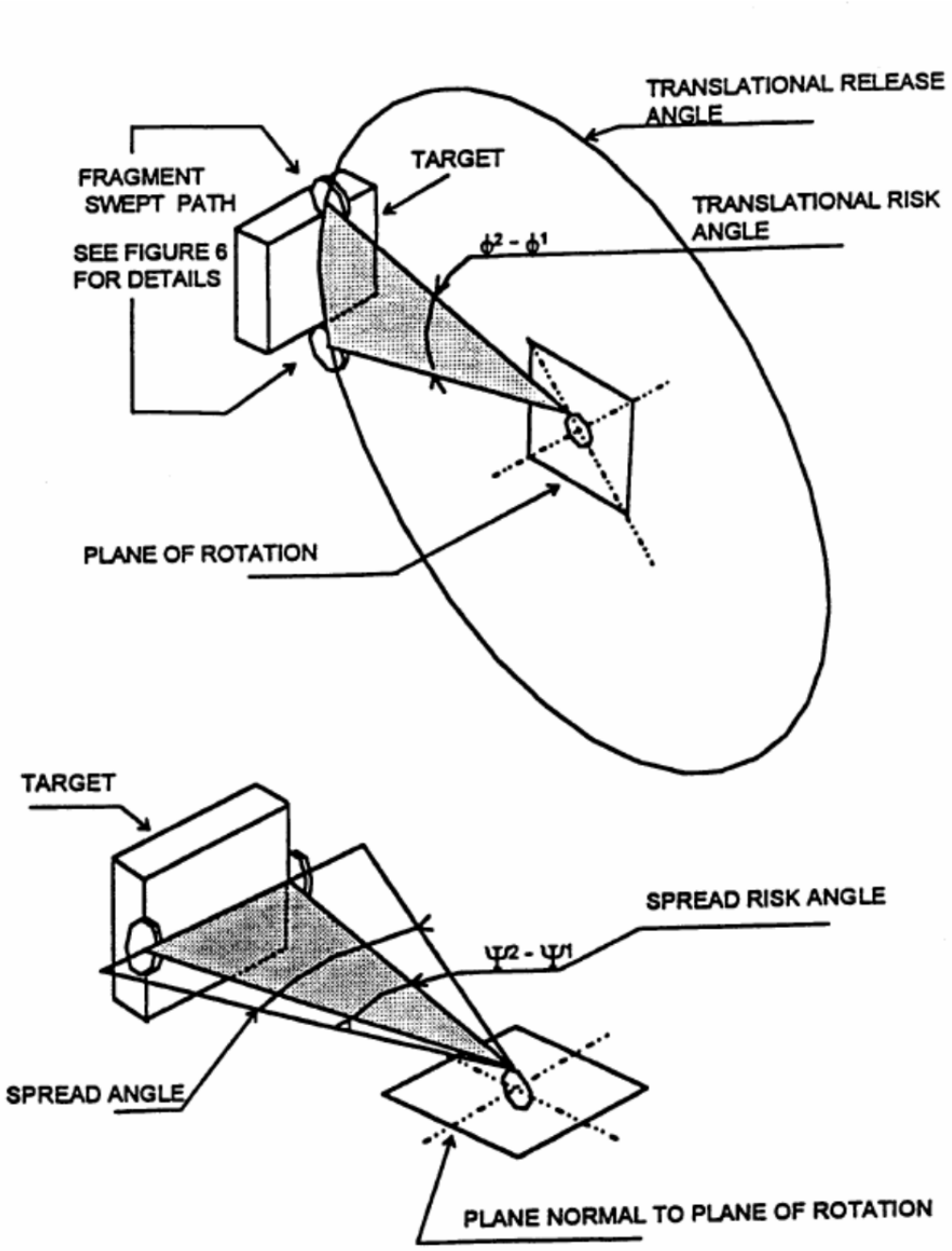

Figure 5.17: Threat window for a rotor burst according to AMC-128A [17]

Listing 5.20: Trajectory path rotation due to the translational angle ϕ

```
val trajectoryFace = createFace(trajectoryWire).
    rotate(locusPoint, AxisX, phiQuantity).
    rotate(locusPoint, AxisX, rotationAngle)
```

To produce the translational-spread trajectory map, an inner loop iterates through the discretized spread angles ψ and adds a second rotation to the trajectory path. The critical component's geometry and the final trajectory geometry, which represents the path after the three rotations, are intersected for each potential trajectory. If an intersection is identified, the respective (ϕ, ψ) pair is printed to the screen, together with the rotor and the component's identifiers. An example of these outputs is pro-

Listing 5.21: Output example for the rotor burst production rule

vided in Listing 5.21.

```
>> Detected hit for Rotor 1 - Right Engine and Left
    boost pump in center tank: Phi 80.0, psi -1.0
>> Detected hit for Rotor 1 - Right Engine and Left
    boost pump in center tank: Phi 85.0, psi -1.0
```

The user is also given access to BREP files that contain the overlapping geometries, which improves the problem's ability to be visualized. Users can provide solutions by pinpointing the exact location of the issue using these geometric files. Finally, the probability of direct hit is computed and the resulting statements are added to the model. The probability of direct hit, (P), is calculated as a fraction considering the possible direct hits individuated my the algorithm and all the possible combinations of ϕ and ψ:

$$P = \frac{\#\text{direct hits}}{n * m} * 100\%$$

Where m and n are the number of discrete translational and spread angles.

Intermediate fragment

Intermediate fragments are assumed to have finite energy and a maximum dimension equal to one-third of the bladed disk radius. Figure 5.18 depicts the fragment's dimension with respect to the rotor. According to AMC 128-A, a spread angle of 5° can be considered for calculations (see Table 5.2). Listing 5.22 illustrates the "when" side for this production rule, which is similar to all three

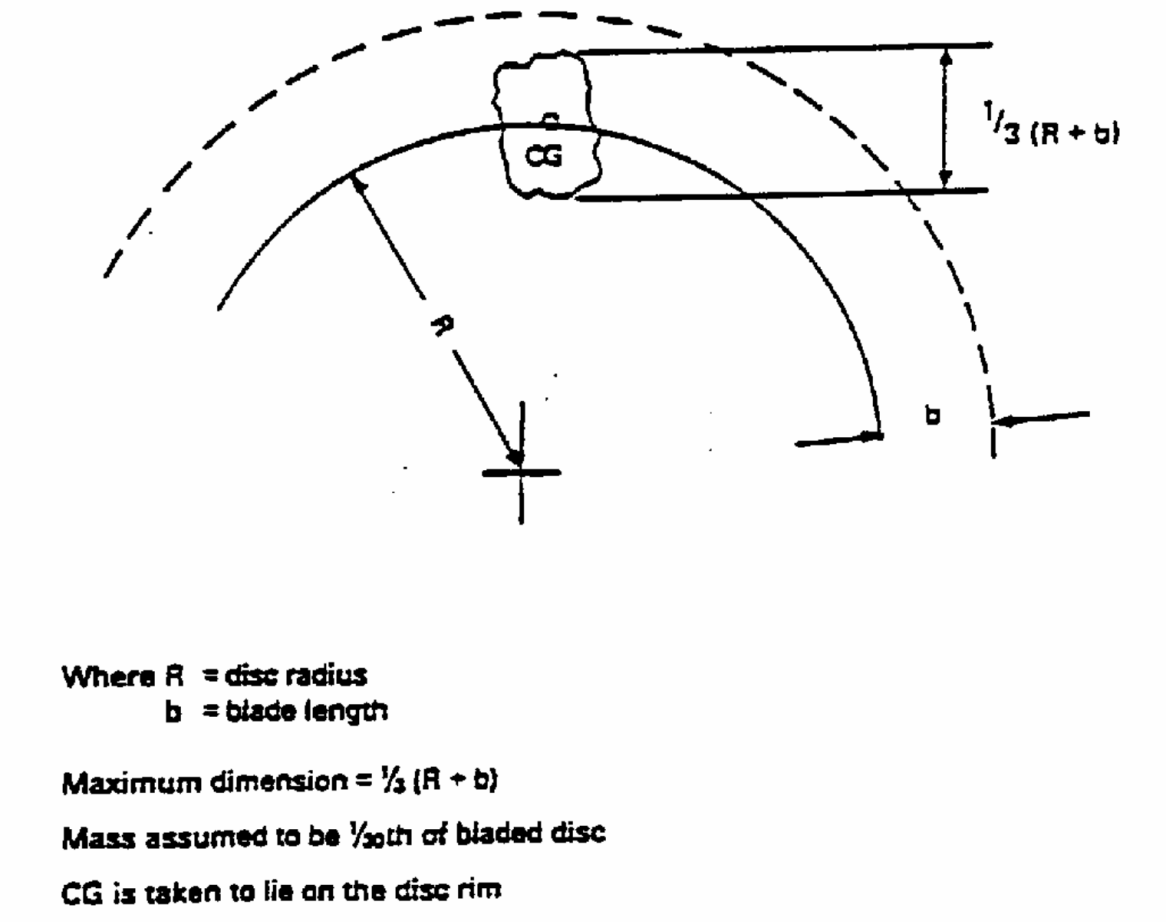

Figure 5.18: Dimensions for the intermediate fragment [17]

rules.

Listing 5.22: When side for the medium fragment rotor burst verification rule

```
When <SIndividual, SIndividual, SIndividual, SRDFLiteral,
SRDFLiteral, SIndividual> {system, component, rotor,
rotorDiskDiameter, originalDiameter, rotorPoint ->
    +(system..a..FS.FuelSystem)
    +(component..a..FS.CriticalComponent)
    +(system..FS.setCheckMediumFragmentRotorBurst..rotor)
    +(rotor..a..FS.EngineRotor)
    +(rotor..FS.hasRotorDiskDiameter..rotorDiskDiameter)
    +(rotor..FS.hasDiameter..originalDiameter)
    +(rotor..FS.hasPoints..rotorPoint)
    +(rotorPoint..a..GKM.Point)
    +(component..a..GKM.Shape)
    +(component..a..GC.Complete)
} Do {...}
```

In order to create the locus point for the trajectory path, the following parameters are used:

$$\begin{aligned} R &= \frac{d_R}{2} \\ R_{bladed} &= \frac{D}{2} \\ b &= R_{bladed} - R \\ d_{max} &= \frac{R+b}{3} \end{aligned}$$

With R the disk radius, R_{bladed} the bladed radius, b the blade length and d_{max} the fragment's maximum dimension. Since now the fragment is assumed to lie on the disk radius, the locus point can be easily found as:

$$locus = (PR_x, PR_y + R * \cos\phi, PR_z + R * \sin\phi) \tag{5.4}$$

And the points defining the trajectory:

$$\begin{aligned} P_1 &= (PR_x, locus_y, locus_z + d_{max}/2) \\ P_2 &= (PR_x, 100, locus_z + d_{max}/2) \\ P_3 &= (PR_x, locus_y, locus_z - d_{max}/2) \\ P_4 &= (PR_x, 100, locus_z - d_{max}/2) \end{aligned} \tag{5.5}$$

The remaining variables remain the same as the previous failure mode. Figure 5.19 illustrates the top view of the fragment trajectories for a spread angle of 5°. The tangential characteristic due to the rotation path is illustrated in Figure 5.20, where the trajectory paths are represented in blue. It is also possible to see the fragment's dimension. Moreover, it is clear that the locus point for the intermediate fragment lies at the disk's rim, at the blades' root positions.

Fan fragment

Because engine fans have large blades, it is assumed that any fragments will have a maximum size equal to one-third of the blade airfoil, with the fragment's centroid is assumed to lie at its centerline. As seen in Table 5.2, a spread angle of 15° is considered. The fragment's dimensions are defined as follows:

$$\begin{aligned} R &= \frac{d_R}{2} \\ R_{bladed} &= \frac{D}{2} \\ b_f &= R_{bladed} - R \\ d_{max} &= \frac{b_f}{3} \end{aligned}$$

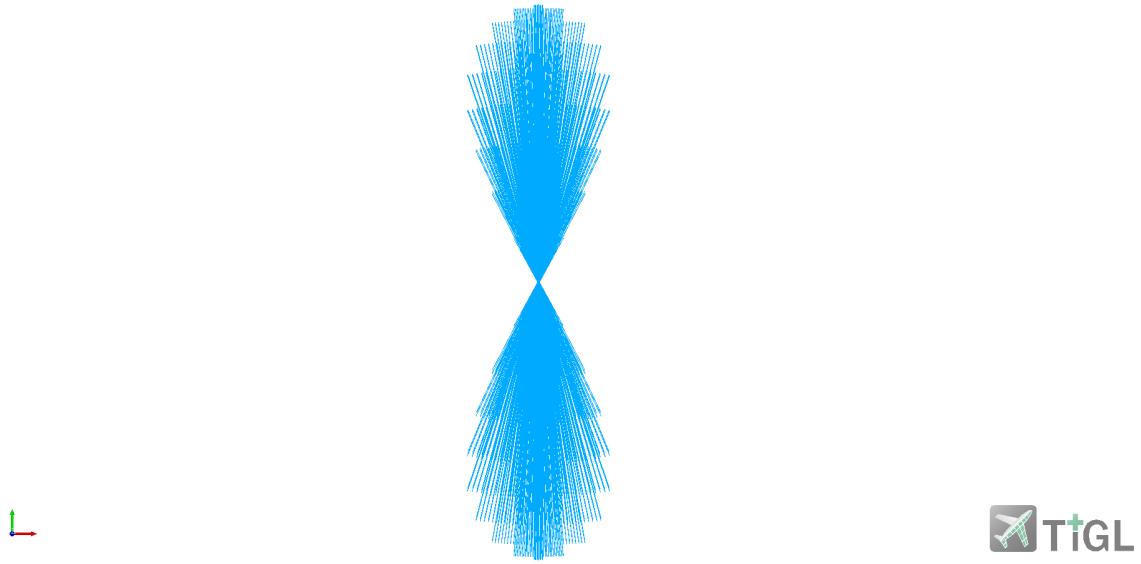

Figure 5.19: Trajectory paths for the intermediate fragment - top view

Where b_f is the fan's blade length. All the other quantities have been defined above and their meaning remain the same. The locus point is given by:

$$locus = (PR_x, PR_y + (\frac{5}{6} * b_f + R) * \cos\phi, PR_z + (\frac{5}{6} * b_f + R) * \sin\phi) \tag{5.6}$$

The points defining the trajectory path are defined using Equations 5.5. Figure 5.21 depicts the front view of a fan fragment's trajectory path. The fragment corresponds to one-third the blade's length. Figure 5.22 depicts the BREP output for a generic system that demonstrates the function's visual capabilities. This system is composed of three boxes in space, an engine fan, and an engine rotor. The failure modes of an engine rotor intermediate fragment and an engine fan fragment were investigated, and the trajectory paths colliding with at least one of the boxes are depicted. The fan fragment is represented by the front disk with orange trajectories, while the engine rotor is represented by the back disk with purple trajectories. This analysis is depicted in detail in Figure 5.23. Because of the different radii and failure modes of the disks, the fragments take on different dimensions. This is yet another demonstration of Codex's incredible ability to deal with geometry verification in a straightforward manner.

Resulting statements

The following are the possible resulting statements for the three verification rules concerning an uncontained rotor burst. If the verification rule identifies a possible hit with the component, an additional statement is added to the model that links the component to an individual and gathers all properties related to the probability of direct hit in order to provide additional information about it other than its value. This is necessary to inform the user of the failure mode that occurs in order to provide the stated probability of hit. Listing 5.23 depicts the creation of a hit probability individual and the statements associated with it in order to better characterize it.

- Verified:
  - Single one-third disk fragment:

    `<component, FS:hasSmallFragmentCollisionPathWithVerified, rotor>`

  - Intermediate fragment:

    `<component, FS:hasMediumFragmentCollisionPathWithVerified, rotor>`

  - Fan fragment:

    `<component, FS:hasFanFragmentCollisionPathWithVerified, rotor>`

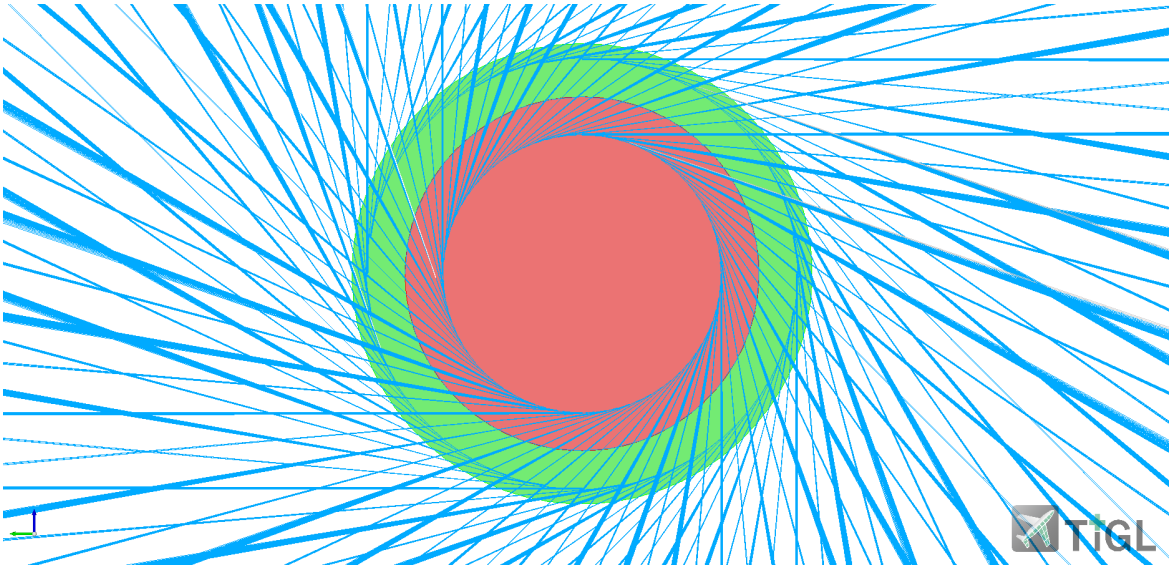

Figure 5.20: Trajectory paths for the intermediate fragment - front view. Rotor disk in red and blade path in green

- Falsified:
  - Single one-third disk fragment:

    <component, FS:hasSmallFragmentCollisionPathWithFalsified, rotor>

    <component, FS:hasProbabilityDirectHit, probHit>
  - Intermediate fragment:

    <component, FS:hasMediumFragmentCollisionPathWithFalsified, rotor>

    <component, FS:hasProbabilityDirectHit, probHit>
  - Fan fragment:

    <component, FS:hasFanFragmentCollisionPathWithFalsified, rotor>

    <component, FS:hasProbabilityDirectHit, probHit>

Listing 5.23: Probability of hit as an individual and related knowledge modeling

```
val probabilityDirectHit = (resultList.size.toDouble() /
    (phiSteps.size * psiSteps.size)) [u.ul]
val probHitIndv = model.createAnonymousIndividual(FS.ProbabilityRotorHit)
model.addAssertion(probHitIndv, RDFS.comment,
    model.createTypedLiteral("Probability␣of␣a␣direct␣hit␣from␣a␣small
␣␣␣␣fragment␣(high␣energy)␣-␣rotor:␣${rotor.readableName}.␣Value␣in
␣␣␣␣decimals␣of␣unit␣(e.g.␣0.1␣=␣10%)"))
model.addAssertion(probHitIndv, FS.hasValue,
    model.createTypedLiteral(probabilityDirectHit))
model.addAssertion(component, FS.hasProbabilityDirectHit, probHitIndv)
```

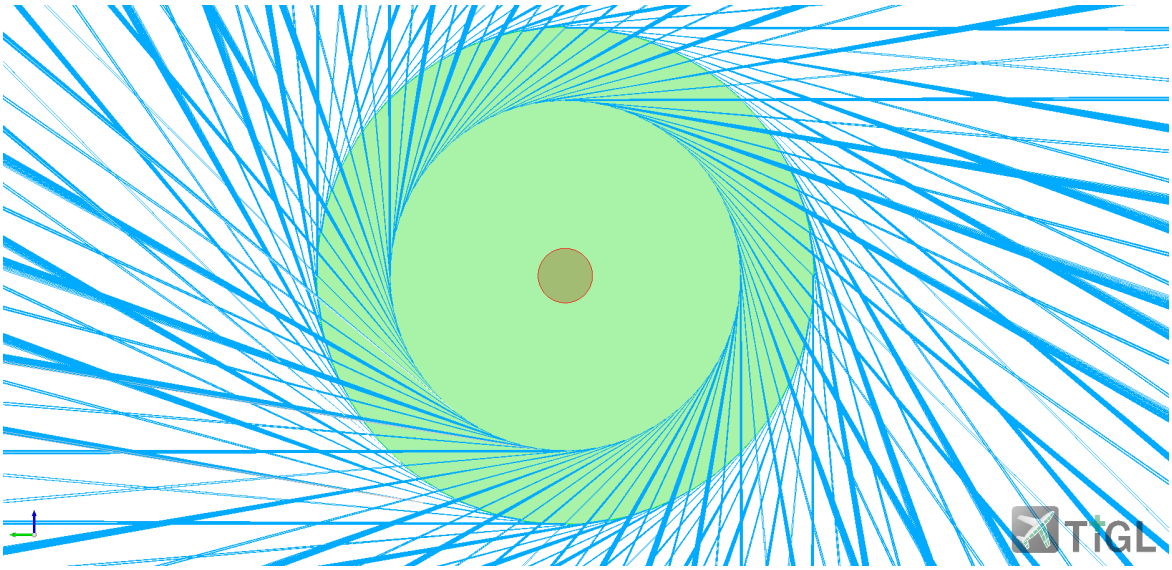

Figure 5.21: Trajectory paths for the fan fragment - front view. Rotor disk in red and fan blade path in green

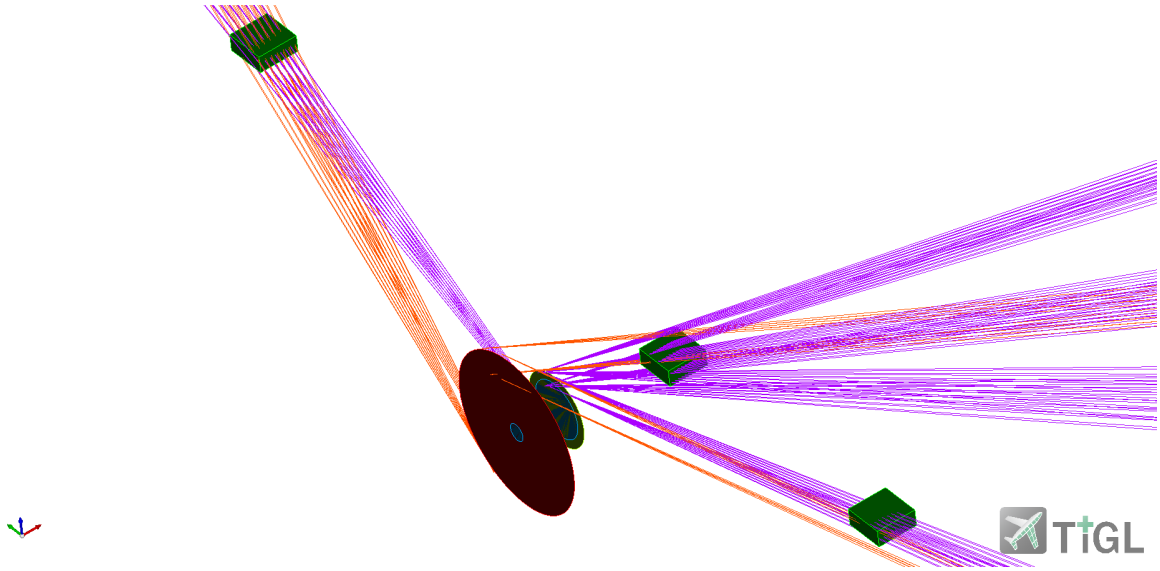

Figure 5.22: Trajectory paths due to a fan and an intermediate fragment burst for a generic system

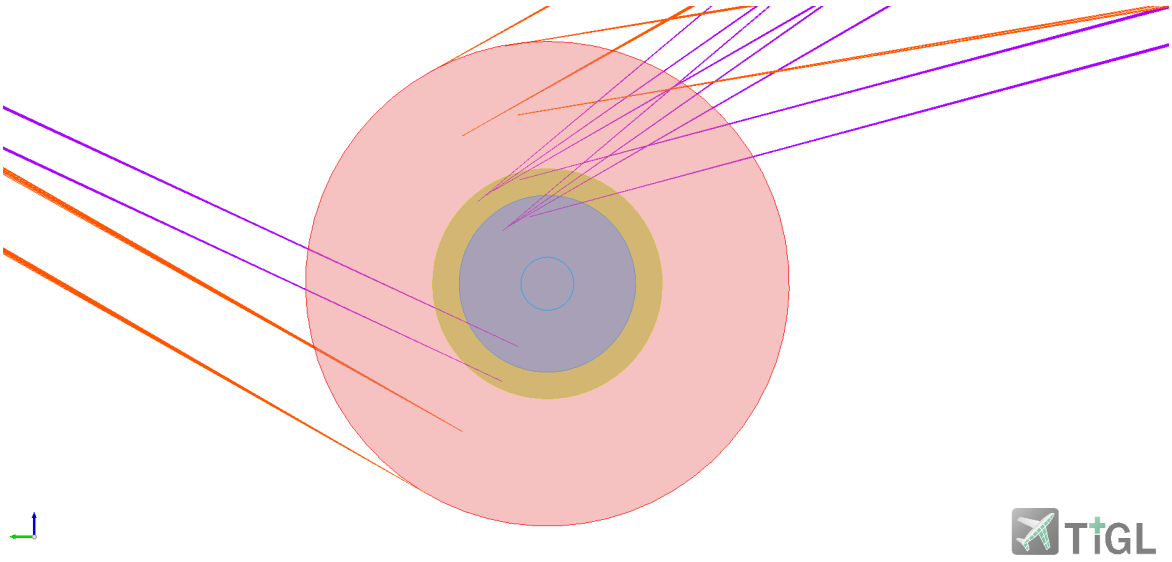

Figure 5.23: Detailed front view of a fan and an intermediate fragment trajectory

Chapter 6

DLR-D150: Fuel system modeling, verification and results

This chapter offers a use case based on a twin-engine modern commercial aircraft that was developed to show the full range of the GeoVerification tool's capabilities as well as the potential of codex-geometry for modeling complex geometries. To demonstrate the validity of the tool, the modeling process will be presented, along with the use of the GeoVerification tool to validate the created geometry, and the reader will be given the results of this simulation process.
Different fuel system architectures [2] were examined in order to gather data regarding the transfer architecture, quantity and placement of fuel pumps, crossfeed strategy, number and placement of fuel tanks, separated by fuel feed and fuel transfer tanks, APU feed architecture, and refueling architecture before making a decision on the architecture of the system. Since the Airbus A320 has a relatively complex fuel system architecture that could benefit from codex-geometry to its modeling, it was determined that this architecture could serve as a starting point to create a proof of concept for the GeoVerification tool. Figure 6.1 depicts the fuel system architecture for an A320, where the feed, transfer, refuel and crossfeed lines can be seen, together with their respective fuel pumps and main valves. In this architecture the main boost pumps in the center and inner wing tanks can be seen, feeding to both the left and right engine independently. In addition, a single crossfeed valve architecture and the low-pressure valves separating the fuel system from the engine can be found.
Although the A320 serves as inspiration to define the fuel system, the aircraft of reference is a design created at the German Aerospace Center [19] and is referred to as D150. A representation of this aircraft using TiGL viewer can be seen in Figure 6.2.
Because the D150 is designed as a small-medium range (SMR) aircraft, a wing-only fuel tank architecture is capable of providing enough fuel storage to meet the mission requirements. A five fuel tanks architecture was chosen; the center tank, the left and right innermost wing tanks, and the two remaining outer wing tanks on the left and right of the wing are in charge of providing engine feed capabilities and transferring the remaining fuel to the inner wing tanks, respectively. In order to accommodate the eventual surge/venting system, an additional requirement defining two additional spaces at the most outer section of both wings was set, though they were not explicitly modeled for the purposes of this use case. Section 6.1 contains a detailed explanation of the geometric modeling of the fuel system architecture.
Since the goal of this proof of concept is to demonstrate how the Codex framework, and in particular the codex-geometry module, can be used to leverage geometry verification, a simulation workflow was set up and is depicted in Figure 6.3. The GeoVerification tool is divided in a set of three separate files containing each a set of verification rules, which is aimed at better segmenting the design process, allowing users to choose specific packages of rules to be used at a certain time. These rules were divided into modules as follows:

- FuelSystemGeometryCheckRules: contains all the rules in Section 5.3.
- FuelSystemRotorBurstCheckRules: contains the three production rules describing the three considered failure modes for an uncontained rotor burst.
- FuelSystemCertificationRelatedCheckRules: contains all the certification related production rules, minus those responsible for evaluating the uncontained rotor bursts.

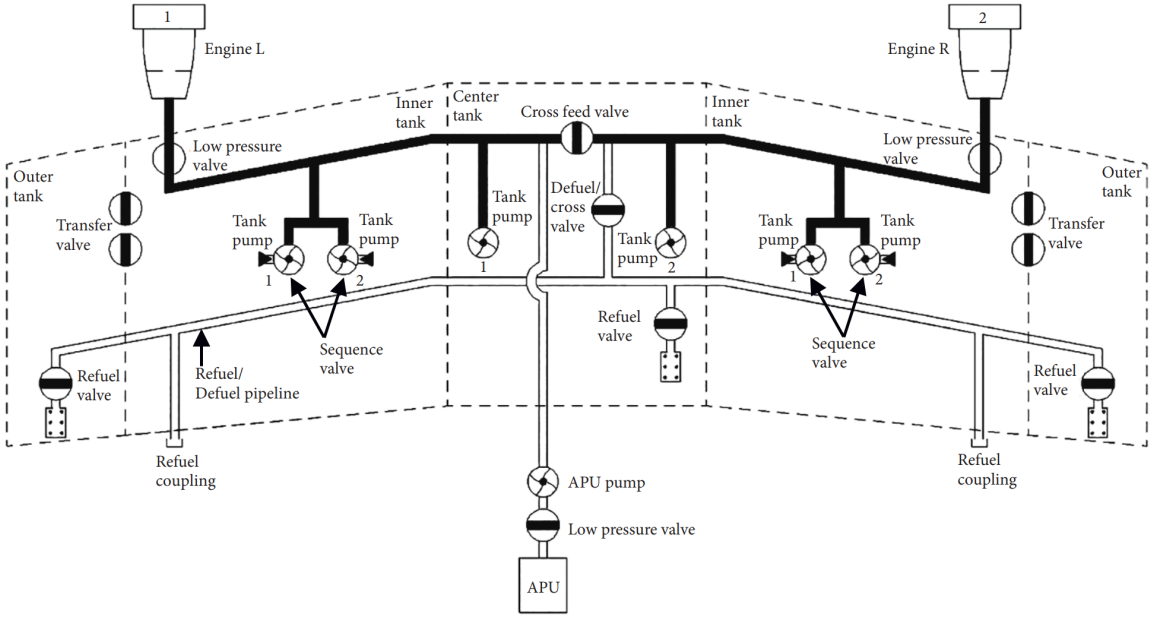

Figure 6.1: A320's fuel system architecture [18]

The process starts by the fuel system modeling using the geometry syntax provided by codex-geometry. In this step, the design space for the fuel system is created, fuel tanks are then defined based on their desired volume and components are placed. Fuel pumps are set first and their positioning lead the pipe routing design, together with the shutoff valves' placement. Supply lines are manually defined through the definition of its root points, which linked together define the lines spine. The geometric knowledge regarding the shutoff valves, such as their positioning and port diameter, is also added during this step. The first reasoning step is led by registering the parametric rules, provided by codex-parametric, which helps to define any missing geometry parameter for the fuel lines [20], and the geometry creation production rules, responsible for creating the complete fuel system architecture. After the activation of the production rules, if the reasoning converges, the architecture is created and saved to a BREP file. A second knowledge base is then created by copying the complete model created and the production rules regarding the geometry verification and geometry completeness are registered. This activity of creating different knowledge bases for each step on the process allows to work with only the necessary rules for the specific step, reducing significantly the simulation time while allowing a better control on code errors' identification and management. To this second knowledge base, all the geometric requirements are set to the model through statements. A second reasoning takes place and evaluates all the geometric requirements using the production rules described in Section 5.3, providing timely information on identified geometric problems on the logger screen.
After the reasoning, a list of verified and failed statements are provided to the user, providing information on the test executed as a property and the related individuals. At this stage, if there are requirements that could not be met, changes on the geometry may be done, thus requiring the rerunning of the first knowledge base and the geometry creation rules. On the other hand, if the design results to be adequate, it is possible to proceed with the second layer of verification, focusing on the geometry-related certification and functional requirements. By adding a third knowledge base and importing to it the previous knowledge, the certification rules defined in Section 5.4 can be registered and executed. As before, the simulation output provides a list of all the successful and failed requirements, providing the user indications of design fallacies. In addition, by extracting information added to the graph by the verification rules, engineers can gather additional information on corrective measures to their design. As an example, suppose that, after running the verification rules, a problem is identified due to an intersection between a supply line and the center feed tank. In order to make geometric corrections to the model, such as adding a tank opening, designers can first look at the model for patterns linking to the property that states this problem, and then look for geometric information on the precise point in space the intersection occurs. When all the conditions

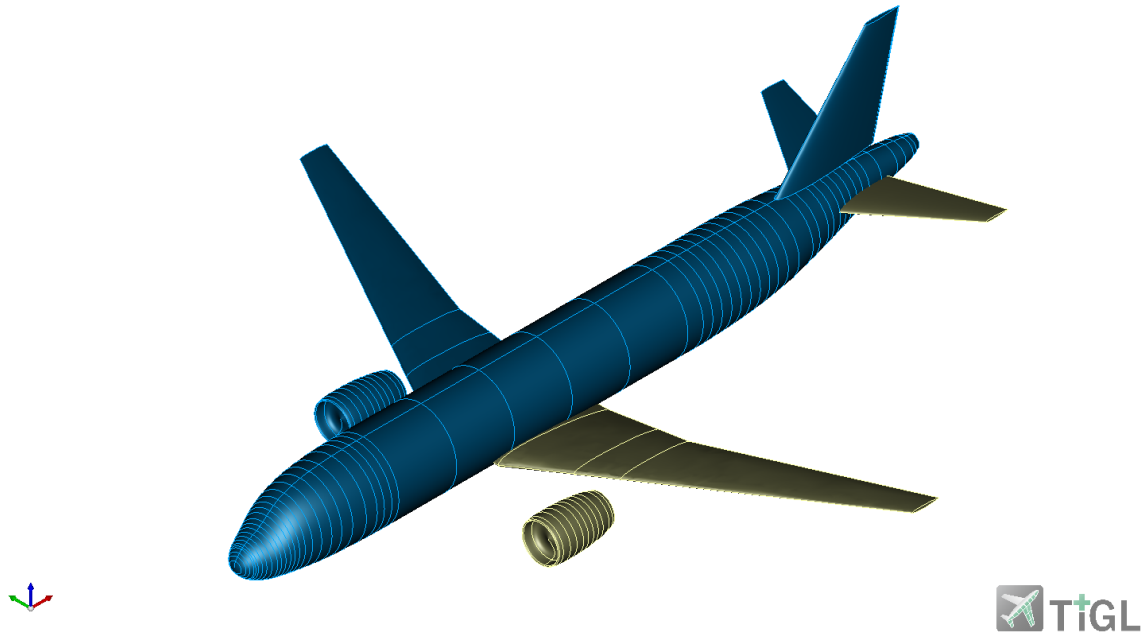

Figure 6.2: DLR's D150 concept aircraft [19]

are satisfied after providing design corrections, the model is said to be geometrically coherent, and as a result, the geometry is said to be verified.
The sections that follow offer a more thorough explanation of each of these steps as well as interesting examples of how to use the geometry syntax to build the fuel system.

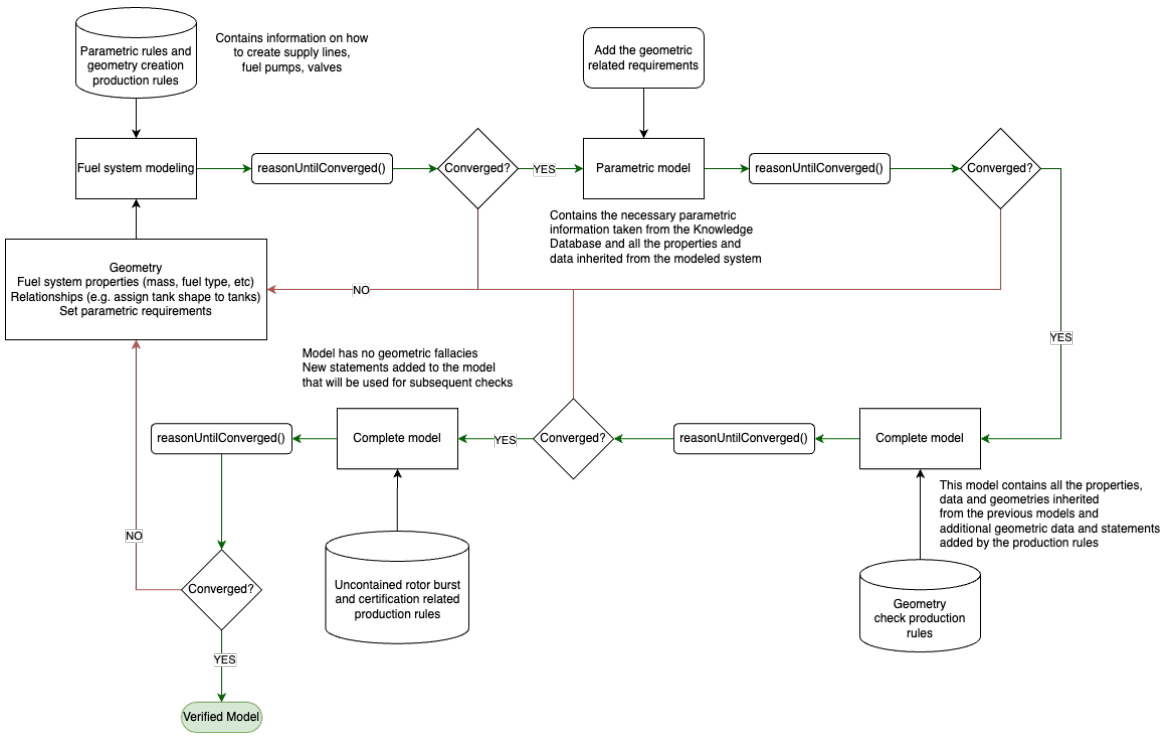

Figure 6.3: Simulation process from system modeling to system validation

6.1 Geometry modeling

The process starts with the design space definition for the fuel system. The D150's data is provided in a XML file following the CPACS standard [45]. Using the python script described in Subsection 5.2.3, the geometric definition of wing spars and wing ribs was extracted and the design space was created. Using the A320's fuel tank volumes as reference, the volume requirements for the D150's fuel tanks are set in Table 6.1.

Tank	Volume
Center tank	4 m^3
Inner wing tanks	6 m^3 / each
Outer wing tanks	0.8 m^3 / each

Table 6.1: Fuel tank volume requirements - D150

Figure 6.4 illustrates the design space created from the wing structural definition found in CPACS, using TiGL. The red elements are the wing's front and rear spars and the wing ribs, while the overlapping blue loft represents the design space that can be extracted from the ribs positioning. A production rule allows to divide the design space volume according to the requirements. Table 6.2 depicts the volumes each tank assume after the design space segmentation. A large amount of the design space has been dedicated to the venting system and a portion of it could be further used for fuel storage.

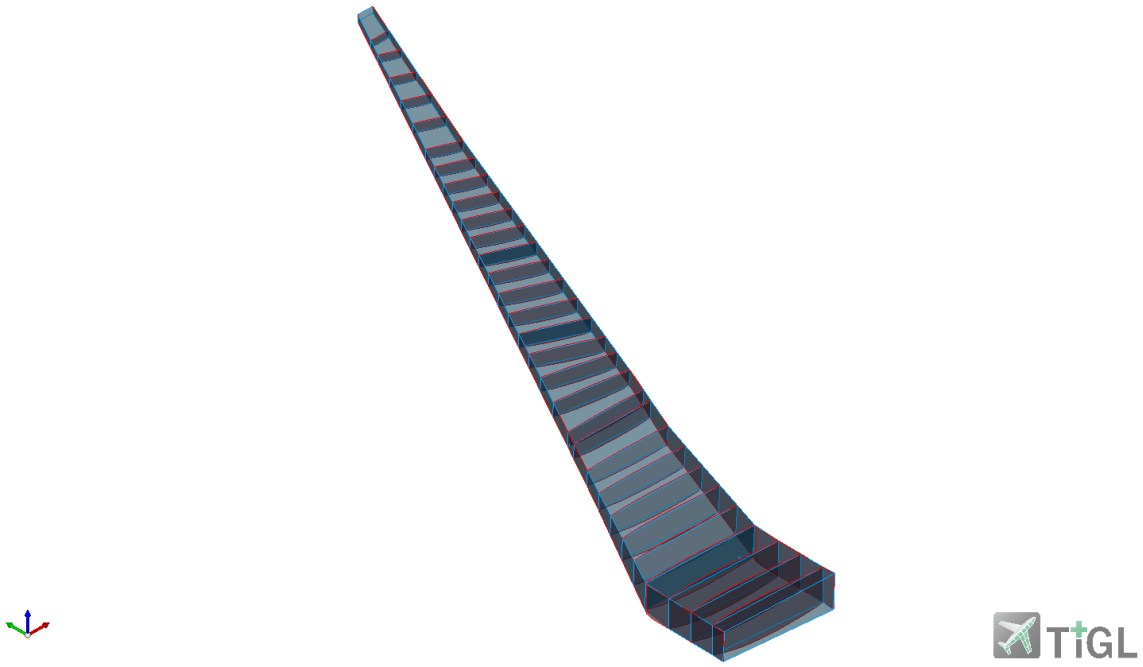

Figure 6.4: Aircraft wing structure (in red) and extracted fuel system design space (in blue)

Tank	Volume
Center tank	4.053 m^3
Inner wing tanks	6.005 m^3 / each
Outer wing tanks	0.883 m^3 / each
Surge/vent	1.177 m^3 / each
Total volume	18.231 m^3

Table 6.2: Fuel tank volume after design space segmentation - D150

As seen in Subsection 5.2.3, the geometry extraction for the aircraft wing structure provides information only on the right wing. After the design space has been divided into sections and the required lofts have been built, it is possible to make mirrored entities that represent the left side wing tanks. It's crucial to point out that only the fuel tank geometry has been created thus far; the actual fuel tank has not yet been connected to it. In Figure 6.5, the created fuel tanks are shown, with the outermost volume serving as the surge/venting tank's available volume before the outer wing tank and inner wing tank. The center tank, which is constructed using the wingbox structure, is housed in the interior's most central and inner volume.

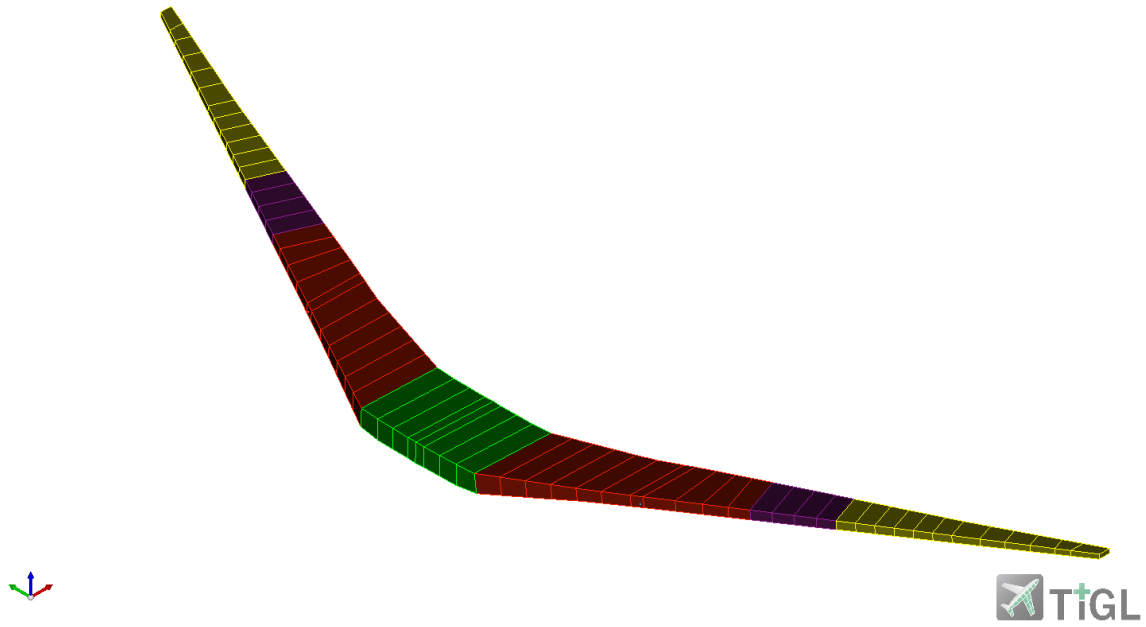

Figure 6.5: D150 - From most outer to most inner, the fuel tank design spaces are as follows: surge tanks (yellow), outer wing tanks (purple), inner wing tanks (red), and center tank (green).

After creating the design spaces, the tank individuals are created and their design space linked to them through the following statement pattern:

```
<tank, FS:hasDesignSpace, designSpace>
```

The entity representing the fuel system is created as can be seen in Listing 6.1. A label property is added to the element in order to name it. Although not necessary for the machine, adding labels provides a more human-friendly model.

Listing 6.1: Fuel system individual creation and structural inputs

```
val fuelSystem = assertedModel.
    createAnonymousIndividual(FS.FuelSystem)
assertedModel.addAssertion(fuelSystem, RDFS.label,
    assertedModel.createTypedLiteral("Fuel System"))
```

The chosen architecture is composed by six boost pumps placed as follows:

- Center tank: two boost pumps are placed close to the front most outer portions of the center tank, one to the left and one to the right side, each providing feed capabilities to the respective engine individually.
- Inner tanks: each wing's inner tank contains two boost pumps, responsible for feeding the respective engine. Both pumps are sized equally in order to provide equal capabilities of engine feed, allowing to maintain the correct fuel pressure to the engine in case of a single failure. The boost pumps are placed separated by wing ribs to allow adequate distancing in case of an uncontained rotor burst.

Fuel is transferred from the outer wing tanks to the inner wing tanks using gravity, taking advantage of the wing's geometry. The placement of the pump makes use of the production rule's ability to determine the ideal location. By applying the production rule, it is possible to give an approximate position based on the system's top view, without the need to correctly place it on the Z axis.
The final root point following the application of the production rule is collected in Table 6.3 along with the desired pump position that is input given for each fuel pump. It is easy to observe how the X and Y root position respect closely the desired position. the Z position is such that the pumps root points lie on the tanks' bottom surface. A tolerance of 0.5 meter was used and judged to provide good results on the pumps' placement. The convention to enumerate the pumps is provided in Figure 6.6. In order to create the pumps, a first reasoning is conducted to activate the necessary production

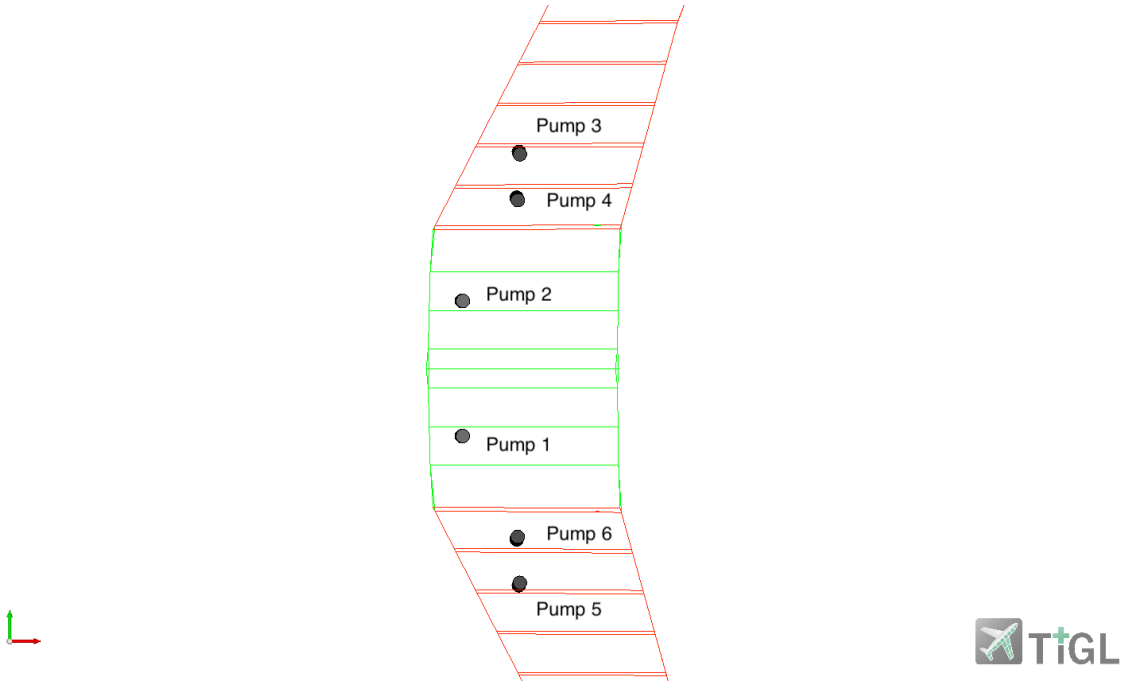

Figure 6.6: Fuel pumps - naming convention

Pump	Desired position [m]	Computed root position [m]
1	(14.0, -1.0, 0.0)	(13.860, -0.992, -1.655)
2	(14.0, 1.0, 0.0)	(13.860, 0.992, -1.655)
3	(14.70, 3.18, 0.0)	(14.698, 3.184, -1.469)
4	(14.8, 2.2, 0.0)	(14.667, 2.510, -1.596)
5	(14.70, -3.18, 0.0)	(14.699, -3.184, -1.468)
6	(14.8, -2.2, 0.0)	(14.666, -2.510, -1.596)

Table 6.3: Input position and calculated root position for the fuel pumps

rules and build the geometries. To the pumps creation follow the supply lines definition. Two main feed lines were created in order to feed both engines separately. Boost pumps 1, 5 and 6 connect to the left engine feed line, while pumps 2, 3 and 4 connect to the right main feed line. In order to build the connection between pumps and the main feed lines, it is possible to use the recently created knowledge regarding the connection port belonging to each of these components. By exploiting their positioning and direction contained in the knowledge graph, it is possible to define the supply line's shape and direction. In fact, by exploiting the port's direction to create the first segment on the supply line connected to the pump, it is possible to avoid possible misalignment errors. Listing 6.2 illustrates the creation of the supply line connecting the pump 1 to the main feed line, where the connection port's positioning and direction is used to create the first segment of the line by connecting the `polePump1Port` and the `pole1`. The line is created using the `createSupplyLineWithRounded-Edges()`, which is responsible for verifying the provided parameters and add the necessary knowledge

to the model through statements, in order to trigger the necessary geometry creation rules.

Listing 6.2: Supply line creation from boost pump related knowledge

```
// first line connecting the pump1 to the system
// get the port direction from the model
val directionPort1 = getDirection(completeModel.statements(port1,
    GKM.hasDirection, null).first().obj.getAs<SIndividual>())
// pole1 will be in the same direction as polePumpPort1
val pole1 = createPoint(polePump1Port.x + 0.2[u.ul] * directionPort1.x,
    polePump1Port.y + 0.2[u.ul] * directionPort1.y,
    polePump1Port.z + 0.2[u.ul] * directionPort1.z)
val pole2 = createPoint(pole1.x + 0.2[u.m], pole1.y , pole1.z - 0.05[u.m])
val pole3 = createPoint(pole2.x + 1.5[u.m], pole2.y , pole2.z - 0.1[u.m])
val line1 = createSupplyLineWithRoundedEdges(assertedModel,
    listOf(polePump1Port, pole1, pole2, pole3), curveRadius,
    innerPipeRadiusEngFeed, outerPipeRadiusEngFeed)
```

The top image in Figure 6.7 shows the main feed supply lines and their connections to each of the supply lines leading to the six boost pumps. Two detail views of pumps 1 and 2 are displayed at the bottom. In these views, a misalignment error in the connection with boost pump 2 can be seen because the related connection port's information was not used when creating the line. On the other hand, since the aforementioned characteristics were taken into consideration during the design, boost pump 1 has a proper connection. Listing 6.3 displays the output message delivered to the user in relation to the boost pump 2's described geometric error.

Listing 6.3: Output message describing a misalignment error for pump 2

```
Checked if line 2 from Right boost pump in center tank is aligned
to Port 2 in Right boost pump in center tank.
    >> line 2 from Right boost pump in center tank is aligned to
    Port 2 in Right boost pump in center tank from Right boost
    pump in center tank: FALSE,   angle: 4.856391638587414 [deg]
```

Designers are required by certification standards to include methods for isolating specific fuel system components in the event that a failure or damage is confirmed. The system can be given isolation capabilities by simply adding shutoff valves. The main feed lines between the center tank and the corresponding inner wing tank for each engine have valves added because they can both be used as feed tanks. The main goal of these measures is to isolate the center tank in the event of a failure and to prevent fuel flow to this tank when it is empty and its pump fails. A second pair of shutoff valves are added between the last portion of the engine feed line and the engine fuel adapter, providing means to isolate the engines in case of failure, as required by CS §25.903.
Through the crossfeed line, which connects the left and right feed lines with a fifth shutoff valve, fuel can also be transferred from the left to the right side. In order to comply with ETOPS specifications, a second crossfeed line is added in parallel to the first. To improve the engine feed and fuel transfer management capabilities and to improve the geometric data that engineers have at their disposal, a total of six shutoff valves are added at this stage, and their positioning can be found in Table 6.4. Points relative to the supply lines are used as a reference to position them. The direction of the valve is assumed to be the same as the flow of the line to which it must be added. In order to create the geometry, a reasoning takes place. At this time, there may be geometric overlaps between the recently created valves and the connecting supply lines. It is possible then to leverage the knowledge created by the production rules that generated the valves to find the exact points in which the connection must happen, through the connection ports, and adjust the lines' dimension. An additional reasoning takes place to change the lines' geometry. The geometry should have converged at this point while still adhering to the specifications.
Listing 6.4 illustrates the parametric approach to execute knowledge modeling for Valve 1, demonstrating how information on the supply line is used to correctly position the component.

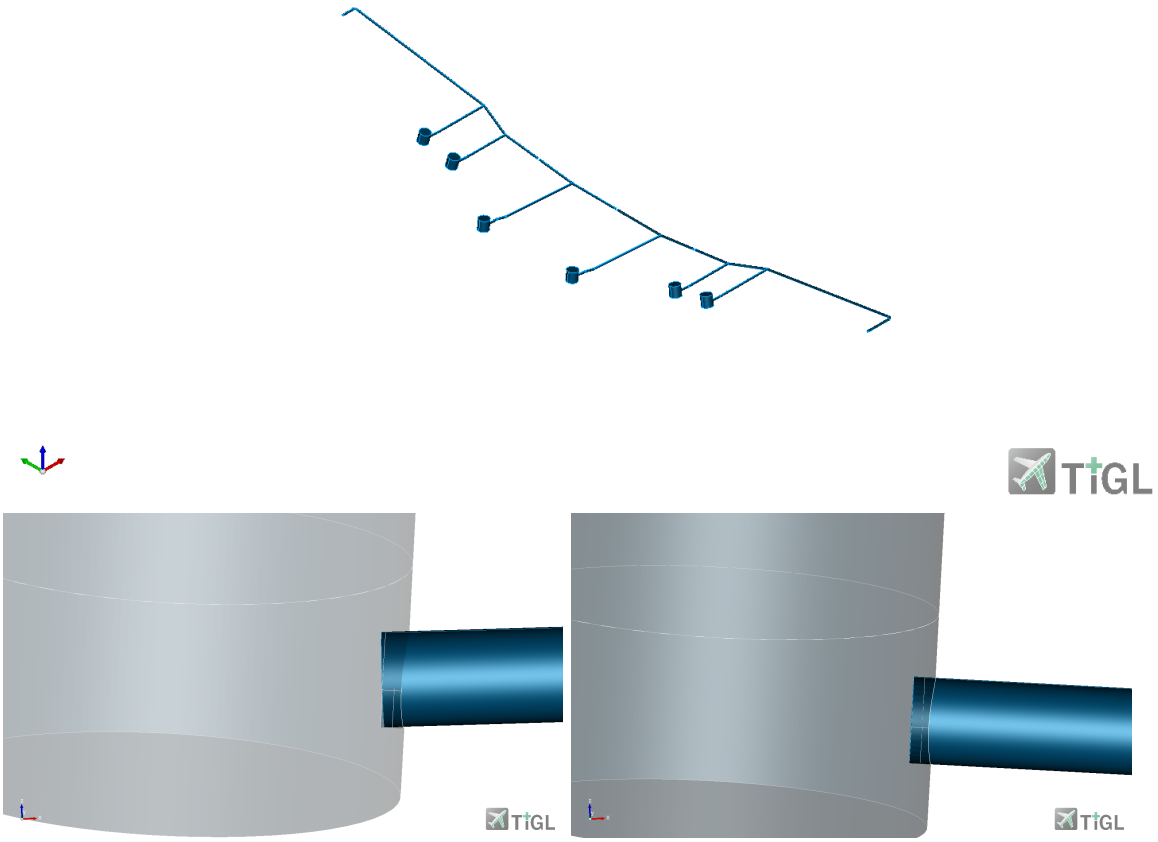

Figure 6.7: Engine feed line and boost pumps (top), detailed view of boost pump 2 (bottom left), and detailed view of boost pump 1 (bottom right)

Listing 6.4: Knowledge modeling to create a shutoff valve geometry

```
val valve1 = assertedModel.createAnonymousIndividual(FS.SOValve)
assertedModel.addAssertion(valve1, RDFS.label,
    assertedModel.createTypedLiteral("Valve␣1"))
// get valve's positioning based on line 1's poles
val poleValve1 = between(pole3, pole6, 0.5).individual
// define valve's geometric parameters
assertedModel.addAssertion(valve1, FS.hasRootPosition, poleValve1)
assertedModel.addAssertion(valve1, FS.hasDirection, directionLine1)
assertedModel.addAssertion(valve1, FS.hasOuterRadius,
    assertedModel.createTypedLiteral(outerPipeRadiusEngFeed))
```

An additional fuel pump was added to provide additional pressure to the APU. This pump assumes a spar-mounted configuration with two connection ports: the first is connected directly to the left main feed line while the second, the outlet port, provides the fuel to the APU.
Additionally, a manifold refuel line was modeled and included in the fuel system. This line is attached to a refueling port inside the outer left wing tank. The construction of this refueling port, which is positioned on the tank's bottom surface, makes use of the production rules created in this project to determine the ideal root position and direction. In Figure 6.8, the refueling port is shown in greater detail. It can be seen that two connection ports were modeled to it: the bottom connection port serves as an inlet port for pressurized ground refueling, while the second port connects the refueling

Valve	Root position [m]
1	(15.853, 1.909, -1.771)
2	(15.853, 5.789, -1.076)
3	(15.853, -1.743, -1.677)
4	(15.7, 6.0, -0.945)
5	(15.5, 6.0, -1.065)
6	(15.353, 1.909, -1.671)

Table 6.4: Feed and transfer valves' positioning

main line. Six ramifications are modeled as a result of the main line's manifold configuration: one for each outer and inner wing tank and two for the central wing tank. The boost pumps on each tank are situated close to these secondary refueling lines. To control which tanks must be filled, additional shutoff valves are added to the ends of each of these lines.

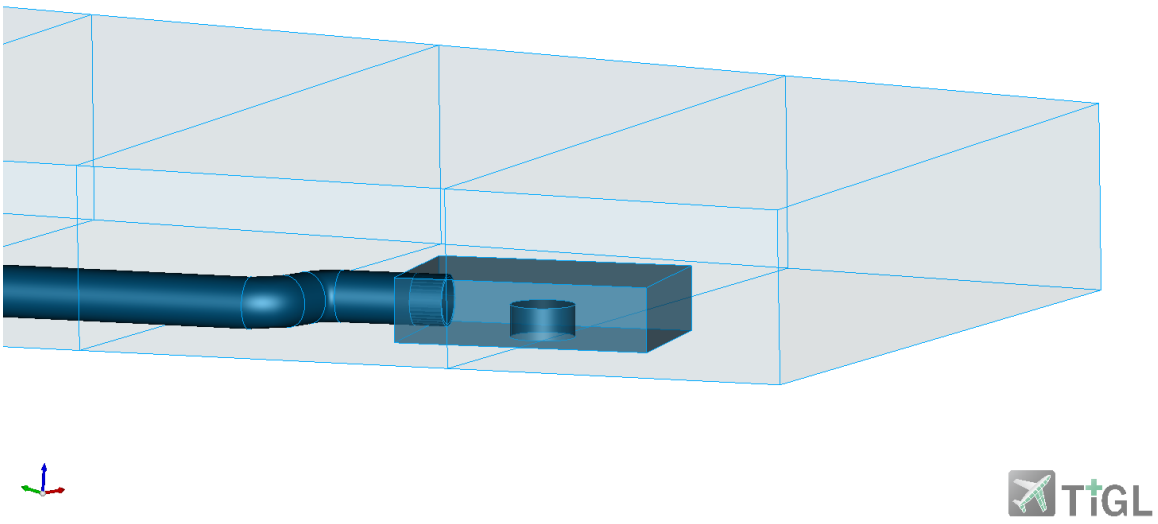

Figure 6.8: Refueling port

Listing 6.5 shows the knowledge modeling process for the refueling port. An additional rotation was added to the refueling adapter in order to better align it to the wing's leading edge.

Listing 6.5: Refueling system creation and refueling adapter's knowledge modeling

```
val refuelSystem = assertedModel.createAnonymousIndividual(FS.RefuelSystem)
assertedModel.addAssertion(refuelSystem, RDFS.label,
    assertedModel.createTypedLiteral("Refuel subsystem"))
val refuelAdapter1 = assertedModel.createAnonymousIndividual(FS.RefuelAdapter)
assertedModel.addAssertion(refuelAdapter1, RDFS.label,
    assertedModel.createTypedLiteral("Refuel adapter 1"))
assertedModel.addAssertion(refuelAdapter1, FS.hasDesiredPosition,
    createPoint(18.0[u.m], -10.0[u.m], 0.0[u.m]).individual)
assertedModel.addAssertion(refuelAdapter1, FS.isProjectedOntoBottom,
    leftTransferTank)
assertedModel.addAssertion(refuelAdapter1, FS.hasRefuelingAdapterRadius,
    assertedModel.createTypedLiteral(outerPipeRadiusRefuelingAdapter))
assertedModel.addAssertion(refuelAdapter1, FS.hasRefuelingLineRadius,
    assertedModel.createTypedLiteral(outerPipeRadiusRefueling))
assertedModel.addAssertion(refuelAdapter1, FS.hasRotationOnPlane,
    assertedModel.createTypedLiteral(120.0[u.deg]))
reasonUntilConverged()
```

An overview of the geometric characteristics of the supply lines is provided in Table 6.5. No study was done on the correct line diameters because the goal of this thesis is to demonstrate the potential of the methodology that has been developed; the values were taken directly from a previous geometry modeling study that is referenced in [20]. Figure 6.9 shows the created geometry for the fuel system.

Line	**Inner radius [m]**	**Outer radius [m]**	**Curve radius [m]**
Feed	0.018	0.020	0.054
Transfer + crossfeed	0.018	0.020	0.054
Main refueling	0.044	0.048	0.132
Secondary refueling	0.022	0.024	0.065
APU feed	0.007	0.010	0.021
APU pump skorkel	0.018	0.020	0.054

Table 6.5: Inner and outer radius for the fuel system supply lines

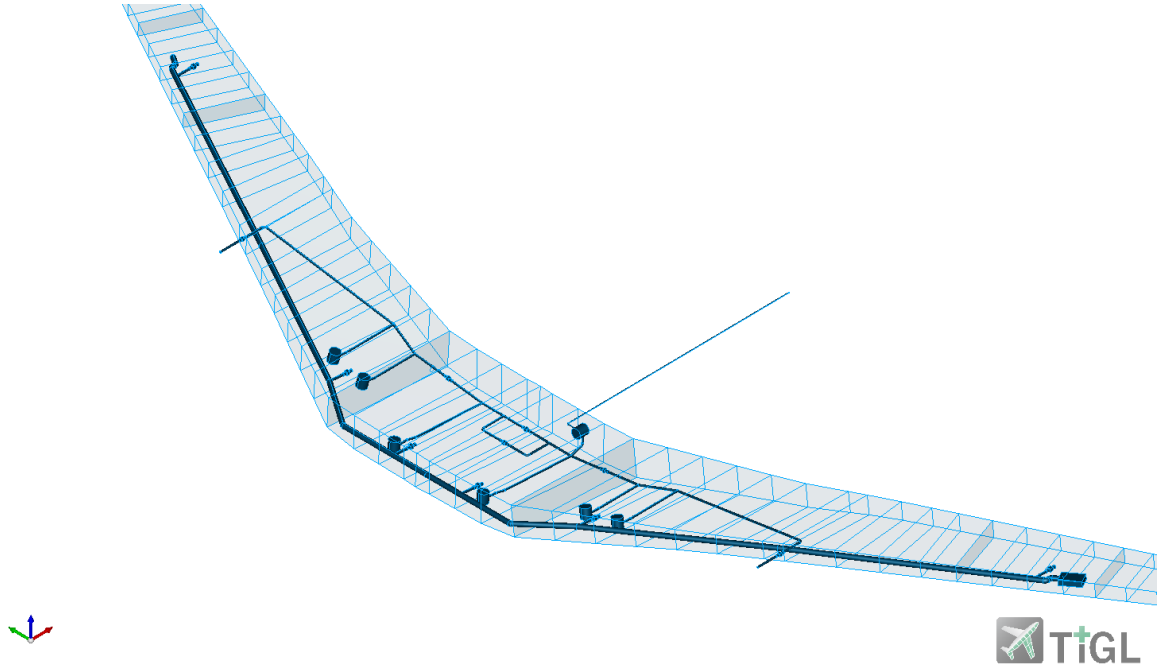

Figure 6.9: Generated fuel system architecture - first iteration

6.2 Geometry verification

By adding the correct properties between elements, a first geometry verification takes place using the `FuelSystemGeometryCheckRules` rules. This initial verification provides some insight into potential design flaws that the designer might have overlooked. This initial verification provides more information on the overlaps between tanks and supply lines since the designed supply lines often cross multiple tanks, which is helpful information for positioning tank openings in the model. In Table 6.6, an overview of the properties added to the model in order to activate the geometry verification rules is represented, where the subject and object are represented by a common name given to the individuals of a same class, while the predicate is the linking property. After running the geometric verification, design flaws can be tracked using the verified/falsified statements that are printed to the logger screen. An additional Turtle (`.ttl`) document containing a textual representation of the knowledge graph is generated. This file can be imported into Codex's web application (under development), which provides a more user-friendly interface to manage the knowledge. Figure 6.10 gathers a few geometry flaws identified by the GeoVerification rule; on top intersections between fuel pipes and the center tank are shown. In addition to that, the crossfeed shutoff valve results not contained in the center tank, thus not meeting the set requirement. On the bottom picture, it is possible to see another valve that does not meet the containment requirements, while Figure 6.11 a failed connection between the supply lines and a fuel valve is depicted, where the lines were wrongly connected to the ports. Listing 6.6 shows a portion of the output logger containing the falsified statements geometrically represented in Figure 6.10.

Listing 6.6: Examples of failed checks printed to the logger

```
<line 15, fuelsystem:isNotOverlappingWithTankFalsified, Center feed tank>
<line 16, fuelsystem:isNotOverlappingWithTankFalsified, Center feed tank>
<Valve 4, geoverification:containedInFalsified, Right feed tank>
<Valve 5, geoverification:containedInFalsified, Left feed tank>
<Valve 6, geoverification:containedInFalsified, Center feed tank>
```

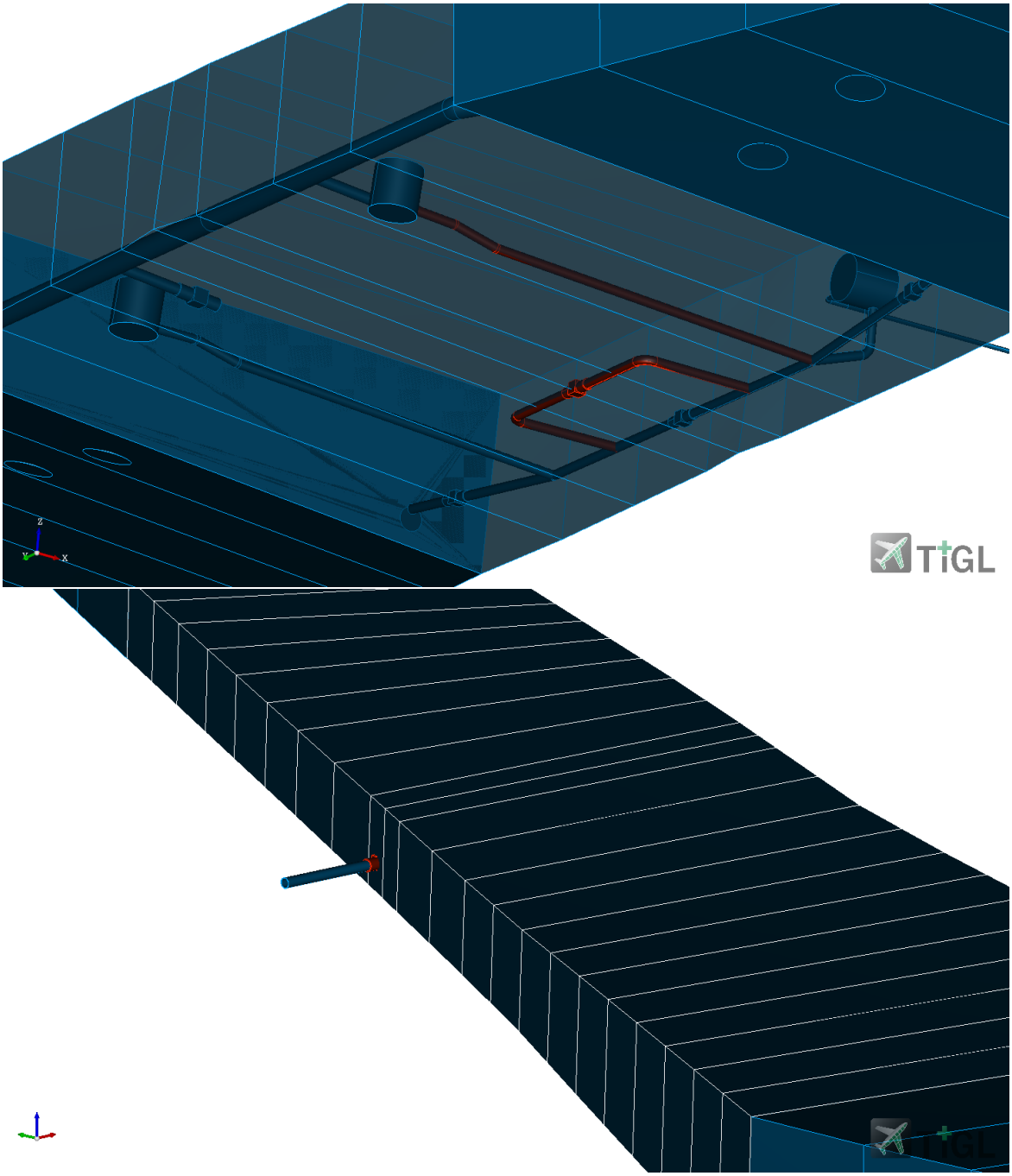

Figure 6.10: Examples of geometric flaws identified by the GeoVerification tool

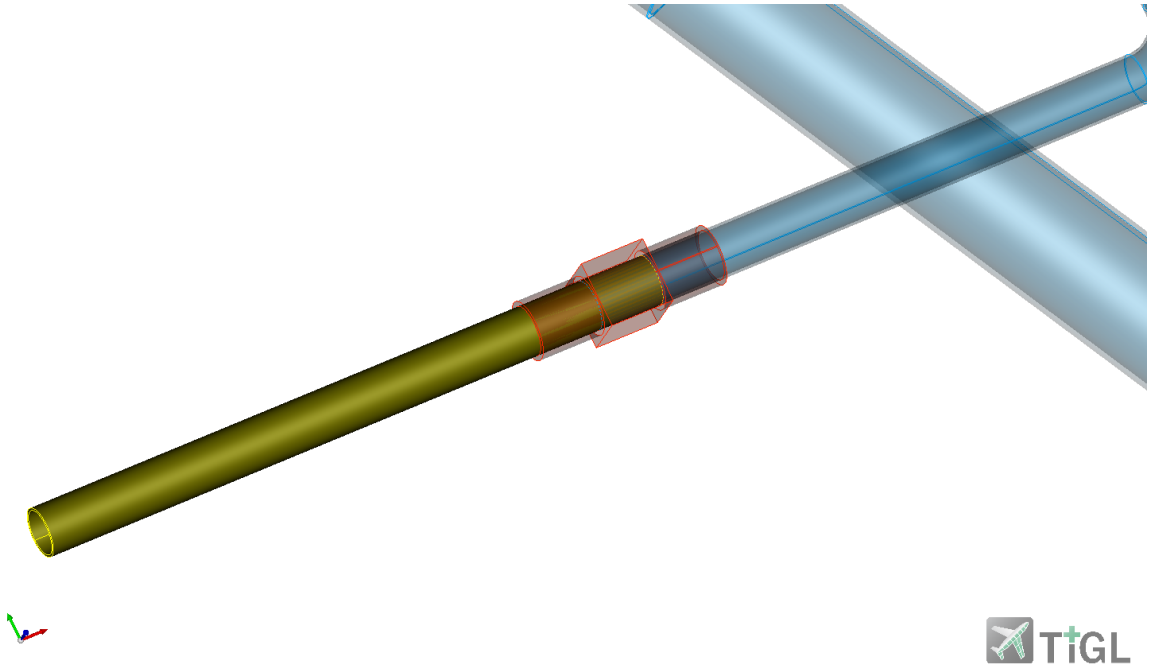

Figure 6.11: Connection error between a supply line and a fuel valve

At this stage, 24 failed checks are present on the model, thus requiring further geometric modeling. By using Kotlin's capabilities to find the necessary statements, collect the necessary geometric data, and then create new knowledge and activate the necessary production rules to modify the design, it is possible to automate this process rather than manually searching for design flaws in the model.
A total of 4 iterations to the geometry modelling were done. After each of them, the GeoVerification tool performed a geometric evaluation of the model with indication of corrective measures. After the last iteration, as anticipated, incorrect intersections between fuel lines and tanks have been eliminated by adding more precise geometric data to the fuel system by including tank openings. The complete list of verified and falsified checks after the last design iteration can be found in Appendix A. After the last design changes, a total of 430 successful geometric checks were performed, validating the geometry for the next step. Figure 6.12 shows the final geometry proposed for the fuel system integrated into the aircraft model, where all geometric flaws have been solved. A detailed view of the center tank and its components, together with a view of two of the tank openings present between the center and the right feed tanks is shown in Figure 6.13, where it is possible to observe two tank openings, from which the refueling line and the right feed line can pass through the tanks.

Subject	Predicate	Object
FS:FuelPump	a	FS:CriticalComponent
	owl:differentFrom	FS:FuelPump
	FS:connectedToRequired	FS:SupplyLine
	Ver:attachedToRequired	FS:FuelStorage
	Ver:containedInRequired	FS:FuelStorage
FS:FuelStorage	a	FS:CriticalComponent
	FS:hasMassRequired	Quantity
	FS:hasMinimumPumpNumberRequired	Quantity
	FS:hasDistinctIndividualsRequired	FS:FuelPump
	FS:hasDistinctIndividualsRequired	FS:FuelValve
FS:FuelSubsystem	FS:hasMinimumValveNumberRequired	Quantity
	FS:hasDistinctIndividualsRequired	FS:FuelValve
FS:FuelSystem	FS:hasFuelType	FS:FuelType
	FS:hasRibsPositions	GKM:Shape
	FS:hasMinimumDistancePipesTanks	Quantity
	FS:setCheckPipeTankIntersection	FS:SupplyLine
	FS:setCheckFanFragmentRotorBurst	FS:EngineFan
	FS:setCheckSmallFragmentRotorBurst	FS:EngineRotor
	FS:setCheckMediumFragmentRotorBurst	FS:EngineRotor
	FS:hasLongitudinalCOGLowerBoundRequired	Quantity
	FS:hasLongitudinalCOGUpperBoundRequired	Quantity
FS:FuelValve	a	FS:CriticalComponent
	FS:belongsTo	FS:FuelSubsystem
	FS:connectedToRequired	FS:SupplyLine
FS:SupplyLine	a	FS:CriticalComponent
	FS:isConnectedToPort	FS:ConnectionPort
	FS:connectedToRequired	FS:SupplyLine
	FS:connectedToRequired	FS:EngineAdapter
	FS:hasMinimumDistanceToRequired	FS:FuelStorage

Table 6.6: Geometric requirements' overview

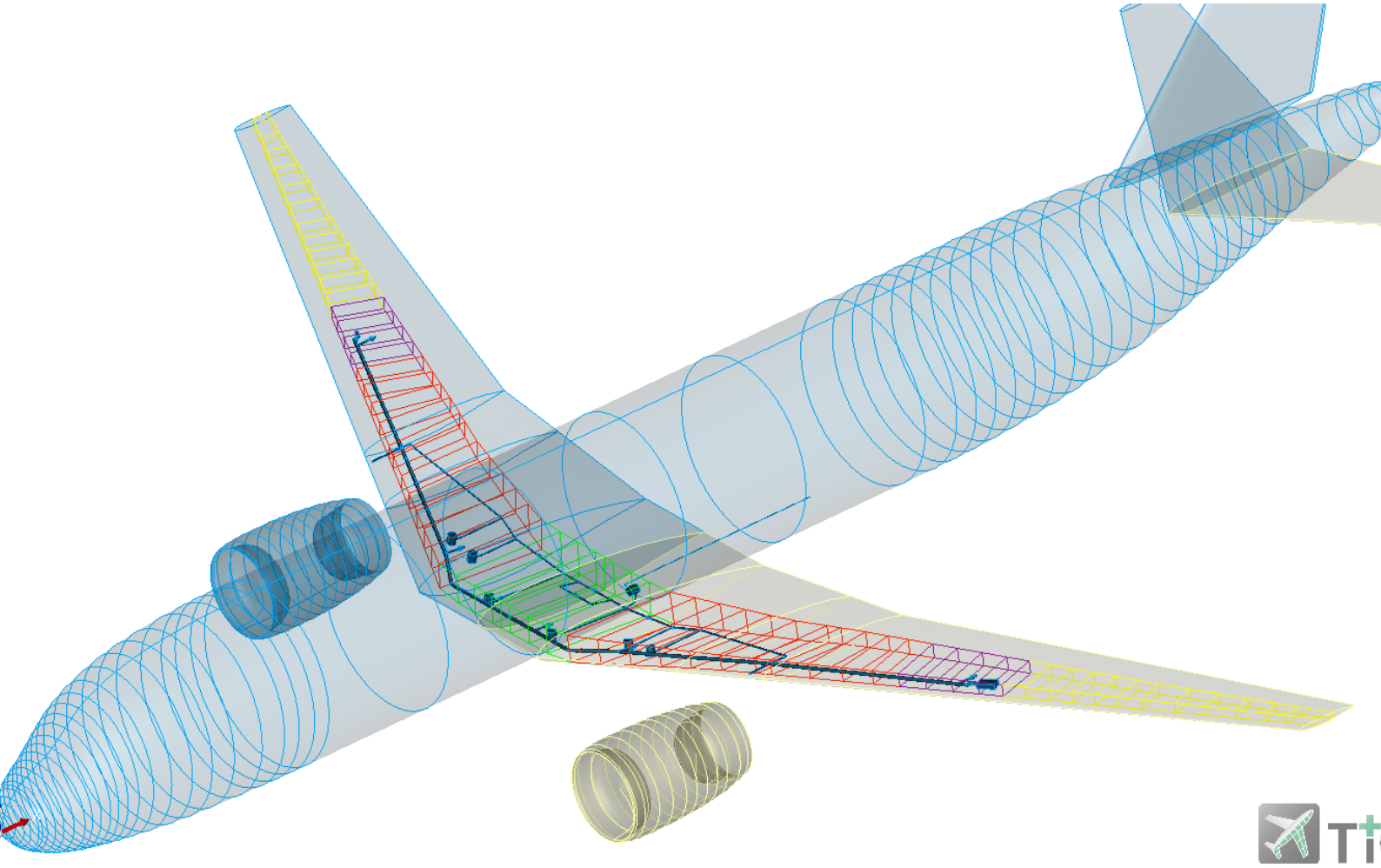

Figure 6.12: Final fuel system design integrated into the aircraft model

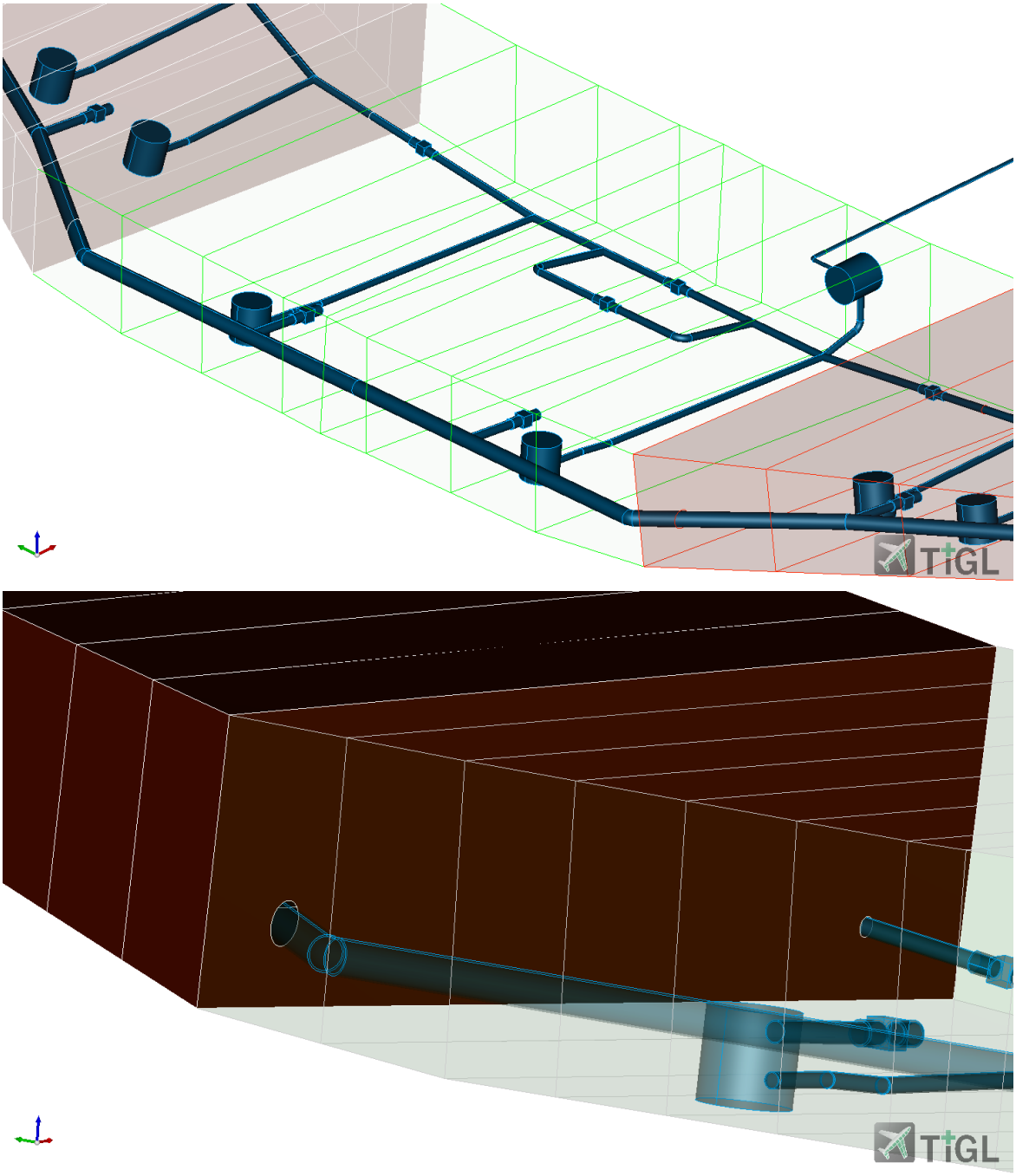

Figure 6.13: Detailed view of the center tank and its components (top picture) and of the tank openings between the center and the right feed tanks (bottom picture)

It was determined that there are two different types of design flaws that affect how supply lines and fuel tanks overlap: overlaps caused by improper line positioning in the design and overlaps caused by closed tank openings. While the former can be resolved by altering the geometry-defining points of the line, the latter necessitates altering the geometry of the tanks by adding openings to their surface in order to allow supply lines to pass through them. Additional geometric problems regarding the connection between supply lines were identified for the supply lines connecting pumps 1 and 2 to the main feed line and was solved by moving the end position for the supply lines slightly upwards.
After gathering data regarding the required tank openings, a second geometry modeling cycle was performed. Not only were the initial design flaws fixed during this process, but new information was also added to the model to allow for changes in the tank geometries that would take opening into account and allow for proper pipe routing. Listing 6.7 provides an example of tank opening, also

defined as a tank cut, knowledge modeling. This knowledge provides geometric information on the opening's size and location. By adding the illustrated knowledge to the model, it is possible to exploit the necessary production rules to correctly place and create the geometry, as seen in Subsection 5.2.3.

Listing 6.7: Examples of geometric knowledge modeling to create a tank opening

```
// get opening's position from the graph
val openingPos = getPoint(completeModel.statements(line8,
    FS.hasOverlapWithTankPosition, null).first().obj.
    getAs<SIndividual>()).individual
val tankCut = assertedModel.createAnonymousIndividual(FS.TankCut)
assertedModel.addAssertion(tankCut, RDFS.label,
    assertedModel.createTypedLiteral("Tank opening"))
// add the geometric knowledge to the tank opening
// hasDesiredPosition triggers the automatic component placement rule
assertedModel.addAssertion(tankCut, FS.hasDesiredPosition, openingPos)
assertedModel.addAssertion(tankCut, FS.isProjectedOntoLeft, centerFeedTank)
assertedModel.addAssertion(tankCut, FS.hasRadius,
    assertedModel.createTypedLiteral(1.1[u.ul] * outerPipeRadiusEngFeed))
```

Table 6.7 provides an overview of the tank openings location added to the model. By setting the following statement pattern to the model

```
<fuelTank, FS:createTankWithOpeningRequired, fuelTank
```

where `fuelTank` is the fue tank individual in question, a production rule responsible for changing the external geometry of the fuel tank is activated. Figure 6.14 illustrates the final geometry for the center feed tank. It is possible to observe 5 openings on this tank, required to pass the main refueling line, in the front, the crossfeed lines, in the back lateral sides, and the APU feed line, exiting this tank through its rear surface. Each of these openings have different dimensions, which are dependent on the respective supply lines, where the opening was radius is taken as 10% larger than the supply line's radius.

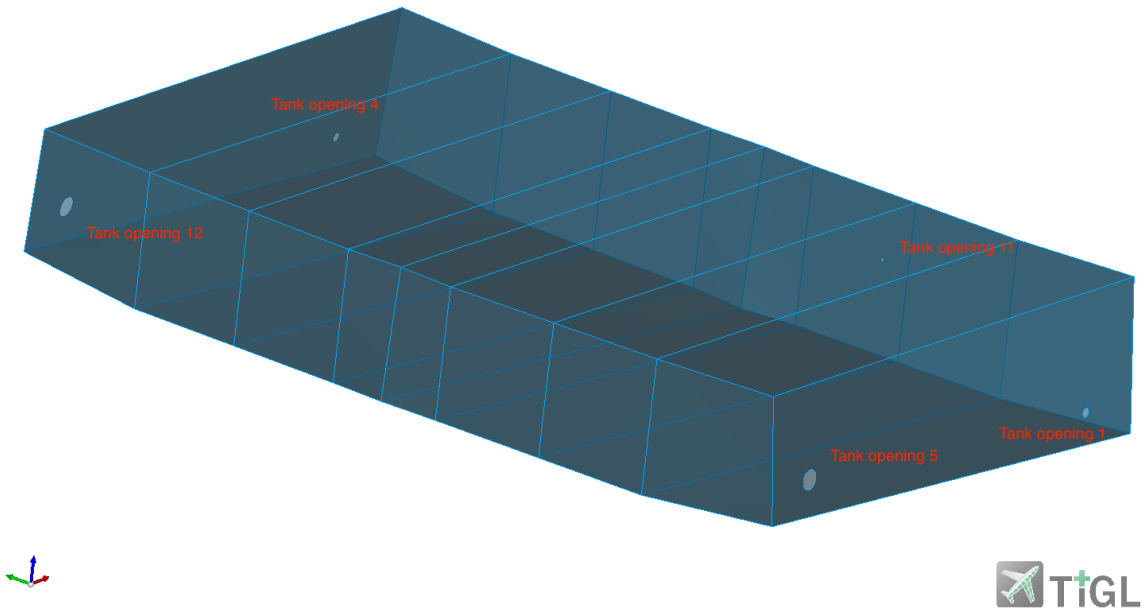

Figure 6.14: Center tank's openings

Tank opening	Position [m]	Connecting tanks
1, 2	(15.85, -2.10, -1.62)	Center Feed Tank Left Feed Tank
3, 4	(15.85, 2.10, -1.62)	Center Feed Tank Right Feed Tank
5, 6	(13.75, -2.08, -1.42)	Center Feed Tank Left Feed Tank
7, 8	(17.09, -8.54, -0.76)	Left Feed Tank Left Transfer Tank
9	(15.47, -6.0, -0.84)	Left Feed Tank
10	(15.47, 6.0, -0.84)	Right Feed Tank
11	(16.15, -0.71, -1.40)	Center Feed Tank
12, 13	(13.75, 2.08, -1.42)	Center Feed Tank Right Feed Tank
14, 15	(17.09, 8.54, -0.76)	Right Feed Tank Right Transfer Tank

Table 6.7: Tank openings' positioning

6.3 Certification related geometry verification

Production rules for the certification-related geometric checks (see Section 5.4) are registered to the project as the final step in the fuel system geometry validation process. Based on certification requirements and guidelines, these rules search for geometry-related design flaws. In order to demonstrate the capabilities of the GeoVerification tool concerning an uncontained rotor burst, geometric information on one rotor stage and on the engine's fan was added to the model for both engines, which can be found in Table 6.8. The rotor and fan dimensions are considered to be similar to high bypass turbofan engines used in the A320, such as the PW6000 [49]. In addition to the knowledge regarding the engine rotors, the fuel pumps were set as target components by means of the `FS:CriticalComponent` class. The computed fragment trajectories for the left engine's fan are shown in Figure 6.15. Due to the fan's forward positioning in relation to the fuel system's components, it was determined that it does not present design challenges, as can be seen in Figure 6.16, where it is clear no trajectory crosses the critical components in the fuel system. The analysis was conducted using the standard values for the translational and spread angles, which can be found in Tables 5.1 and 5.2. Listing 6.8 illustrates the knowledge modeling for the right engine's fan, where geometric information regarding the fan's positioning and the fan's geometry is added to the model.

Listing 6.8: Geometry modeling for the right engine's fan

```
val fan1 = assertedModel.createAnonymousIndividual(FS.EngineFan)
assertedModel.addAssertion(fan1, RDFS.label,
    assertedModel.createTypedLiteral("Engine␣Fan␣-␣Right␣Engine"))
// add geometric information to the engine fan individual
assertedModel.addAssertion(fan1, FS.hasRotorDiskDiameter,
    assertedModel.createTypedLiteral(1.0[u.m]))
// hasDiameter accounts for the complete, bladed, diameter
assertedModel.addAssertion(fan1, FS.hasDiameter,
    assertedModel.createTypedLiteral(1.3[u.m]))
// position the fan in space
assertedModel.addAssertion(fan1, FS.hasPoints,
    createPoint(11.8[u.m], 5.75[u.m], -2.3[u.m]).individual)
assertedModel.addAssertion(fan1, FS.hasRotationDirection,
    assertedModel.createTypedLiteral("clockwise"))
```

After running the GeoVerification tool with the FuelSystemRotorBurstCheckRules module, no fuel pumps result to be subject to a rotor burst due to a small fragment. A second attempt was performed, this time adding the supply lines as critical components as well. From this simulation a total of 9 possible hits for each engine rotor was detected, for a total of 18 possible hits with 9 supply line

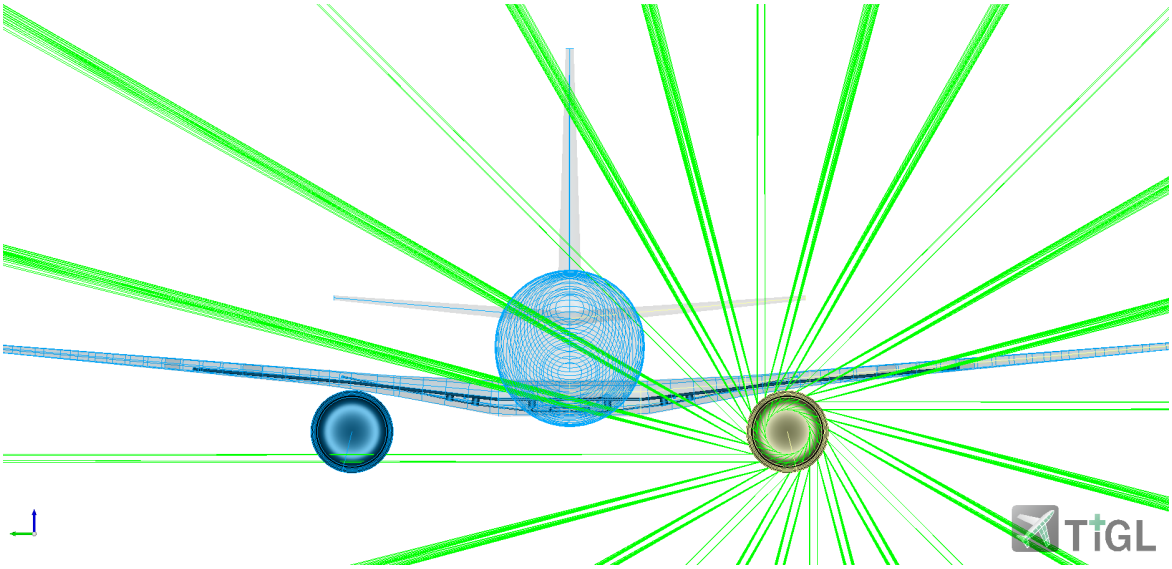

Figure 6.15: Trajectories for the left engine's fan fragment

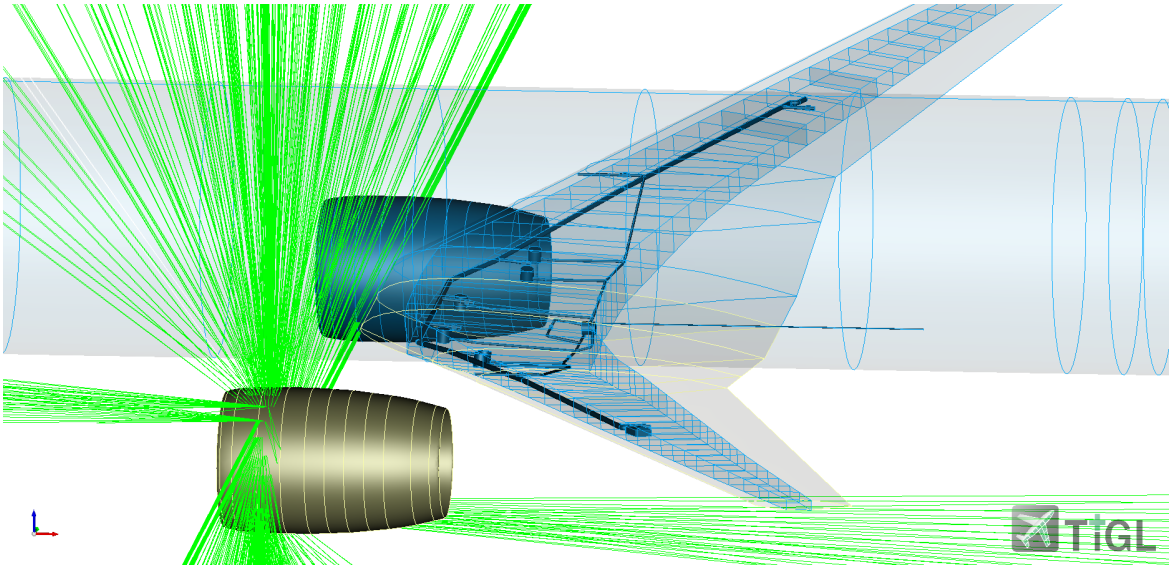

Figure 6.16: Trajectories for the left engine's fan fragment - detailed view focusing on the fuel system

segments. Table 6.9 shows the possible hits due to a small fragment originated from the right engine's rotor 2 and the pair of translational (ϕ) and spread angles (ψ) for which they occur. The intersecting trajectories for this rotor's fragment can be seen in Figure 6.17, where the intersecting supply lines are depicted in red. In this example, the refueling lines are affected by a potential rotor burst from the right engine. With this information available, aircraft designers can proceed to making risk evaluation on the system's behavior and potentially provide design changes to the architecture.
Although the GeoVerification tool also identified other non-compliant geometries, such as valves in potentially icing-prone positions, it was decided not to pursue further geometry changes at this point because the purpose of this proof of concept was to show the tool's verification capabilities rather than to provide a fuel system without any geometric flaws. Figure 6.18 shows an example of a valve in an icing-prone situation. The left connecting supply line, shown in red, is higher than the valve, while the right connecting line, shown in green, is positioned so that fuel and water move away from the valve with the help of gravity, preventing the formation of water.
Apart from the uncontained rotor burst verification, 65 certification-related checks totaling 12 failed checks were carried out. Table 6.10 displays the list of failed checks. Three of the five fuel tanks do not adhere to the imposed mass requirements, as can be seen from this list. By exploring the knowledge

Rotation object	Position [m]	Bladed radius [m]	Disk radius [m]	Sense of rotation
Rotor 1 - Left Engine	(13.5, -5.75, -2.3)	0.65	0.5	Clockwise
Rotor 2 - Right Engine	(13.5, 5.75, -2.3)	0.65	0.5	Clockwise
Fan 1 - Left Engine	(11.8 -5.75, -2.3)	0.8	0.15	Clockwise
Fan 2 - Right Engine	(11.8 -5.75, -2.3)	0.8	0.15	Clockwise

Table 6.8: Geometric information for the engine rotors and fans

model and examining the information at hand, it is possible to see that the mass requirements are not satisfied by a few tens of kilograms. Since there is a sizable amount of space available for the venting system, it would be feasible to use this space to increase the amount of room for the fuel tanks. This would enable the mass requirements to be met without the need to add additional tanks to the aircraft's fuselage. The complete list of performed checks can be found in Appendix A.

Component	Possible hit for (ϕ, ψ)
Line 23	$(80.0°, 3.0°)$
	$(75.0°, 2.0°)$
	$(75.0°, 3.0°)$
Line 18	$(80.0°, 2.0°)$
	$(80.0°, 3.0°)$
	$(85.0°, 3.0°)$
	$(80.0°, 2.0°)$
Line 17	$(80.0°, 3.0°)$
	$(85.0°, 3.0°)$

Table 6.9: Supply lines potentially damaged by an uncontained rotor burst (small fragment) for the right engine's example rotor

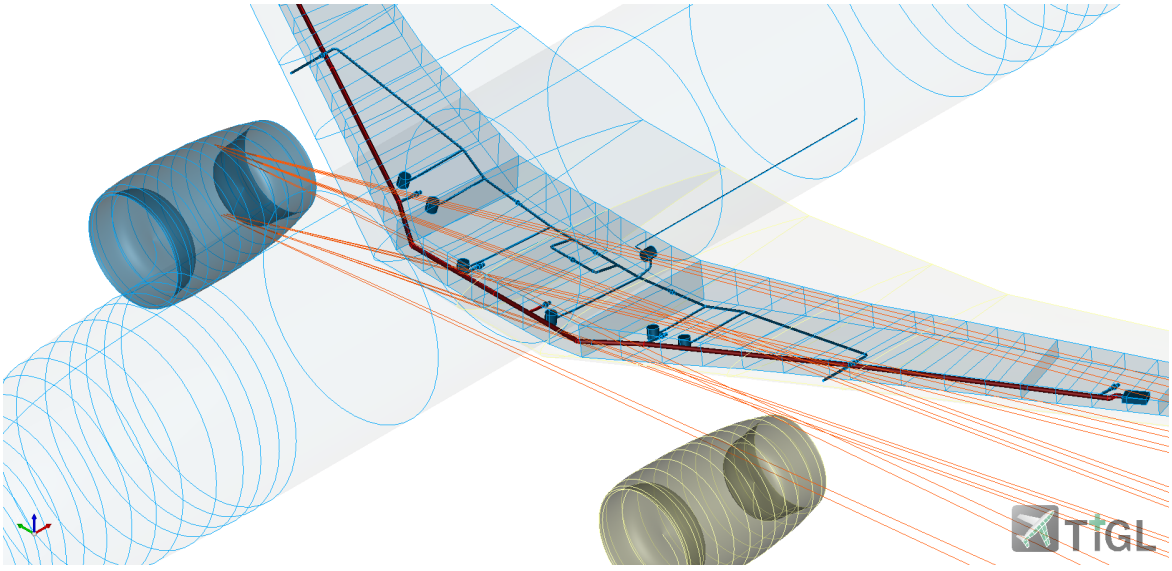

Figure 6.17: Hit trajectories for the right engine's rotor fragment - components in the hit zone

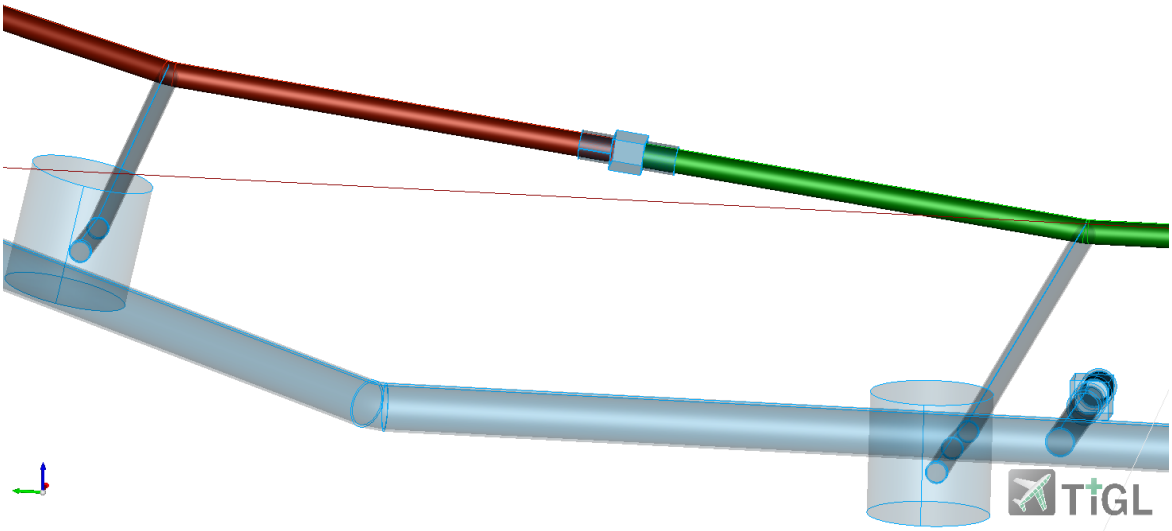

Figure 6.18: Example of fuel valve subject to icing condition on the right engine feed line

Subject	Predicate	Object
Center feed tank	FS:hasMassFalsified	6476 [kg]
Fuel system	FS:hasMassFalsified	18728 [kg]
Left feed tank	FS:hasMassFalsified	5435 [kg]
Right feed tank	FS:hasMassFalsified	5435 [kg]
Valve 1	FS:isNotProneToIcingFalsified	Line 7
Valve 1	FS:isNotProneToIcingFalsified	Line 8
Valve 11	FS:isNotProneToIcingFalsified	Line 27
Valve 12	FS:isNotProneToIcingFalsified	Line 29
Valve 2	FS:isNotProneToIcingFalsified	Line 10
Valve 3	FS:isNotProneToIcingFalsified	Line 9
Valve 8	FS:isNotProneToIcingFalsified	Line 21
Valve 9	FS:isNotProneToIcingFalsified	Line 23

Table 6.10: Failed checks - certification related geometry checks (excluding UERF rules)

6.4 Enhancing geometric detail in early fuel system design through production rules

Providing extended geometric information at an early design stage is a main goal defined for this project. By utilizing the geometry creation production rules within the GeoVerification tool, it is possible to enhance the level of detail for the fuel system in its early design stage. While the components might not possess the same level of geometric complexity required for later stages, their accurate placement and existence enable the early detection of design flaws, as demonstrated in this section. Previous efforts to model a fuel system using Codex are discussed in Section 1.2 and detailed in [20]. In Figure 6.19, a comparison between the fuel system proposed in [20] with the geometrically improved fuel system developed in this project is provided. The benefits of introducing production rules for geometry creation are evident, as they facilitate a more intricate fuel system design capable of more advanced geometric assessments than its predecessor.
While the groundwork laid by the work in [20] led to the creation of the GeoVerification tool, the authors recognized the potential advantages of incorporating additional geometric details into the

model. This recognition served as the initial inspiration for this master's thesis. Figure 6.20 provides a detailed comparison of a fuel pump in both scenarios. The upper depiction showcases the earlier design, where fuel pumps were simplified as cylinders and supply lines were overlapped onto them. In contrast, the lower image exhibits the newly enhanced geometric representation of the fuel pumps, complete with connection ports. The value of incorporating production rules becomes evident, as it now accurately enables the connection of a supply line to a fuel pump. Additionally, geometry assessments regarding the correctness of such connections can be conducted using the implemented production rules in the tool.

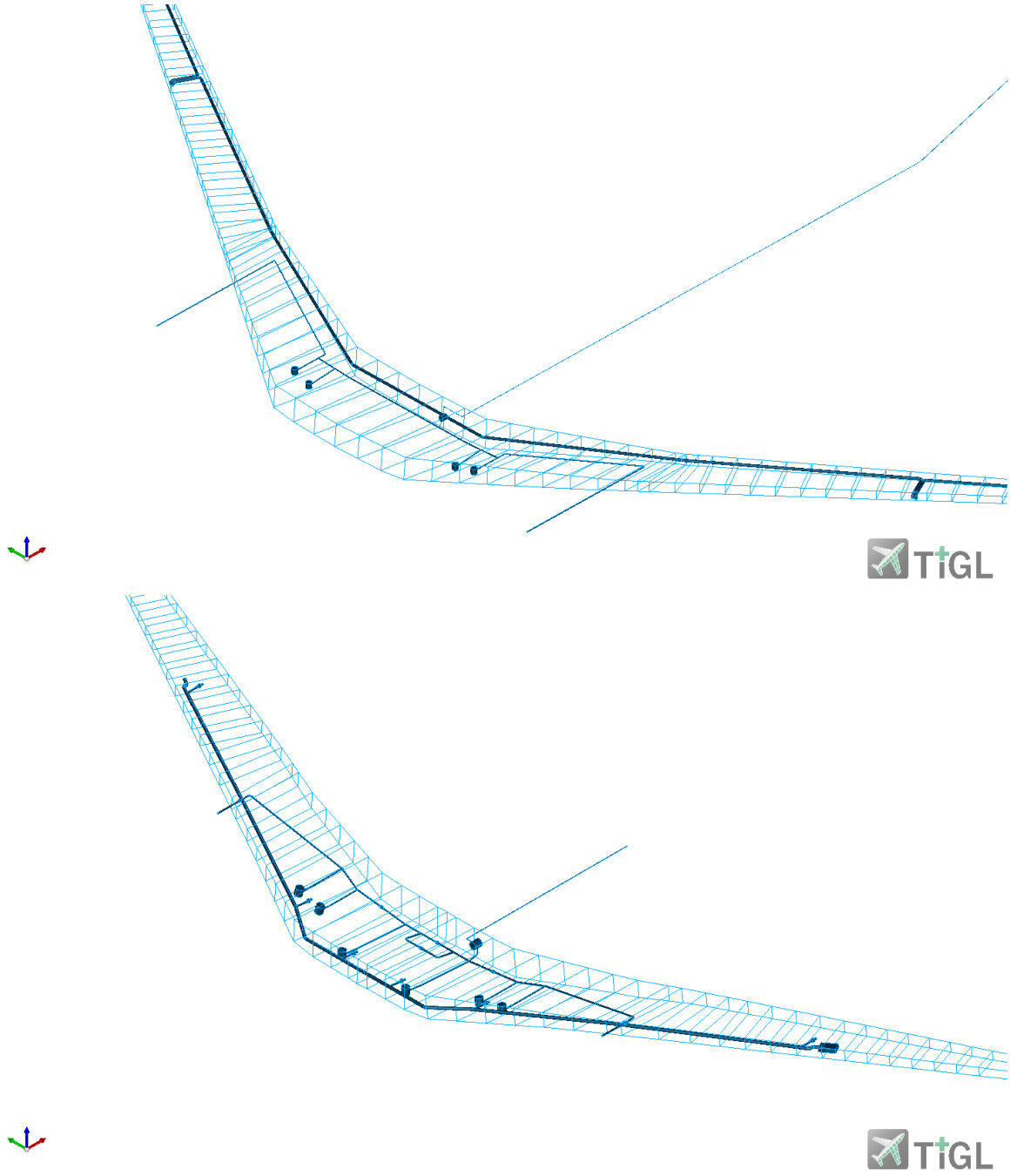

Figure 6.19: Geometric detail comparison between the fuel system found in [20] (top) and the proposed fuel system design (bottom)

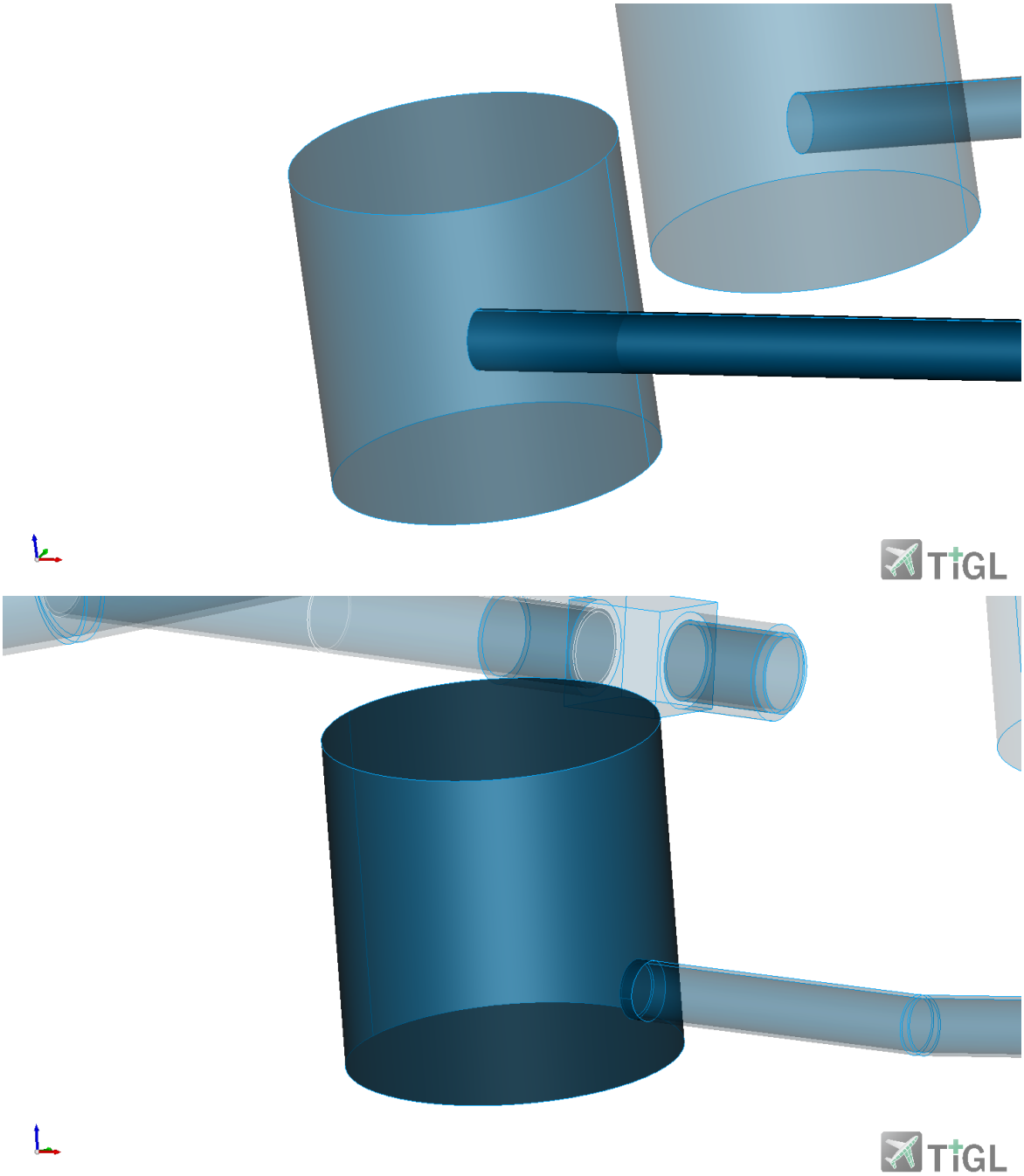

Figure 6.20: Geometric detail comparison for the fuel pumps and their connection with a supply line

Chapter 7

Conclusions

This master thesis provided an overview of how it is possible to exploit semantic web technologies (SWT) to perform geometry verification activities on the fuel system design at an early design stage, exploiting the Codex framework. By using Knowledge Based Engineering methodologies, it was demonstrated the capabilities of automating the creation of fuel system components. Additionally, the benefits of enriching the geometric knowledge available at an earlier stage were demonstrated. In fact, as more geometric information is available, it is possible to provide additional capabilities and means of identifying geometric flaws in the design at a conceptual level. Furthermore, it was shown how it is possible to gather both explicit and tacit knowledge from heterogeneous sources to develop rules that can be easily integrated and reused. The potential of Codex's reasoning engines together with a declarative-rule architecture show the high modularity this framework offers. The advantage of this modularity was shown in Chapter 6, where different rules were registered and used in the project at different stages. This modularity allows for faster reasoning times as only the necessary module of rules are used at each step of the project. Beyond this, the adoption of a graph-based approach, as exemplified by Codex, emancipates design from the conventional hierarchical confines of model-based engineering. This characteristic empowers unrestricted access to information, facilitating a more direct pathway to information retrieval compared to traditional frame-based designs. The developed use case has provided a comprehensive insight into potential basic geometric anomalies inherent in fuel system design, effectively showcasing the capabilities of the developed GeoVerification tool in identifying design irregularities. Furthermore, the study has showcased the formulation of intricate geometric rules, translating multifaceted certification prerequisites into geometric verification rules to assist designers, which is only possible thanks to the more detailed geometric information available to the user, which also translates in a faster tool development.
A step towards assisted design was taken and explained in Chapter 5, where the development of the production rules to find a feasible position for the components' placement were explained. The combined influence of these verification and placement rules might pave the way for automated component positioning aligned with the system's requisites in the foreseeable future. Encompassing the comprehensive GeoVerification framework, the endeavor yielded a collection of 25 verification rules and 10 geometry creation rules.
While the tool's efficiency has been demonstrated, real-time capabilities remain elusive due to the requisite reasoning time for rule activation and knowledge graph evaluation. The design construction reasoning, as illustrated in the presented use case, necessitates approximately 5 minutes to complete all essential elements. Encouragingly, the geometric verification conducted through the GeoVerification tool has showcased promising evaluation times, with sub-4-minute reasoning times achievable for over 500 geometric evaluations using a standard personal computer. The specification of the aforementioned computer is displayed in Table 7.1.

Component	**Technical Specification**
Processor	Intel Core i5 11th Gen
RAM	32 GB DDR4
OS	Windows 10

Table 7.1: Simulation computer technical specifications

In tandem with the goal of achieving real-time capabilities, a discernment has emerged that focusing reasoning solely on altered portions of the knowledge graph is imperative. This strategic approach holds promise for substantially reducing reasoning times, as only modified knowledge necessitates supplementary reasoning efforts.
The GeoVerification tool resulted to be a great asset to support aircraft fuel system design. It not only furnishes users with insights into detected geometric incongruities but also endeavors, when feasible, to present corrective actions to the identified design shortcomings. Lastly, the Codex framework, alongside its constituent modules, has proven its utility in engineering KBE tools aimed at supporting aircraft design. The integration of the codex-geometry module streamlines geometry management, alleviating associated workload burdens. Additionally, the codex-quantity module simplifies unit conversions, endowing users with greater flexibility in knowledge modeling, ultimately contributing to a design process less susceptible to errors.
In light of the accomplishments and insights derived from this study, several avenues for future research and development are discernible. One notable avenue is the pursuit of enhancing real-time capabilities within the GeoVerification tool. The reduction of reasoning times through selective activation of rules in response to altered knowledge graph sections presents a compelling prospect to achieve seamless real-time performance. Furthermore, the extension of the tool's capabilities to encompass a broader spectrum of geometric complexities and verification scenarios could greatly amplify its utility. Exploring advanced machine learning techniques for the identification and formulation of geometric verification rules could potentially enhance the tool's automated detection accuracy and efficiency. Additionally, the integration of collaborative features could facilitate concurrent design reviews and collaborative error resolution among design teams. Expanding the framework's scope to include compatibility with evolving semantic web standards and technologies would ensure its relevance and longevity within the dynamic landscape of engineering practices. Ultimately, these potential avenues for future work hold the promise of fostering even greater automation, accuracy, and efficiency in the aircraft fuel system design process, thereby contributing to the advancement of both the aviation industry and semantic web technologies.

Bibliography

[1] SAE International, "AIR7975: Aircraft Fuel System Design Guidelines," 12 2019.

[2] R. Langton and E. Al, *Aircraft Fuel Systems*. American Institute Of Aeronautics And Astronautics ; Chichester, 2009.

[3] Heritage Concorde, "Concorde fuel transfer."

[4] H. Gavel, *On Aircraft Fuel Systems*. Linköpings Universitet, 2007.

[5] AIRBUS GmbH, "A320 flight crew operating manual - fuel."

[6] S. van der Elst, M. van Tooren, B. Vermeulen, C. Emberey, and N. Milton, "Application of a knowledge based design methodology to support fuselage panel design," *Cambridge University Press*, vol. 114, pp. 589–597, 09 2010.

[7] I. Sanya and E. Shehab, "An ontology framework for developing platform-independent knowledge-based engineering systems in the aerospace industry," *International Journal of Production Research*, vol. 52, pp. 6192–6215, 05 2014.

[8] O. Isaksson, "A Generative Modeling Approach to Engineering Design," *International Conference on Engineering Design*, 08 2003.

[9] A. R. Kulkarni, "Requirement-product-process Ontology," 05 2022.

[10] C. Emberey, N. Milton, J. Berends, M. Van Tooren, S. Van Der Elst, and B. Vermeulen, "Application of Knowledge Engineering Methodologies to Support Engineering Design Application Development in Aerospace," *7th AIAA ATIO Conf, 2nd CEIAT ; followed by 2nd TEOS Forum*, 09 2007.

[11] E. Jayakiran Reddy, C. Sridhar, and V. Pandu Rangadu, "Knowledge Based Engineering: Notion, Approaches and Future Trends," *American Journal of Intelligent Systems*, vol. 5, pp. 1–17, 01 2015.

[12] G. L. Rocca and v. Tooren, "Knowledge-based engineering to support aircraft multidisciplinary design and optimization," *Proceedings Of The Institution Of Mechanical Engineers, Part G: Journal Of Aerospace Engineering*, vol. 224, pp. 1041–1055, 09 2010.

[13] W. J. C. Verhagen, P. Bermell-Garcia, R. E. C. van Dijk, and R. Curran, "A critical review of Knowledge-Based Engineering: An identification of research challenges," *Advanced Engineering Informatics*, vol. 26, pp. 5–15, 01 2012.

[14] J. Zamboni, A. Zamfir, E. Moerland, and B. Nagel, "A semantic knowledge based engineering framework for the rapid generation of novel air vehicle configurations," 2022.

[15] J. Zamboni, A. Zamfir, and E. Moerland, "Semantic Knowledge-Based-Engineering: The Codex Framework," *12th International Conference on Knowledge Engineering and Ontology Development*, 01 2020.

[16] F. Olives, *Weight estimation of parametrically design of fuel and hydraulic systems of a commercial airplane*. PhD thesis, 09 2019.

[17] European Union Aviation Safety Agency EASA, "Easy Access Rules for Acceptable Means of Compliance for Airworthiness of Products, Parts and Appliances (AMC-20)," 03 2021.

[18] Y. Tu, X. Chen, L. Yin, Q. Zheng, and Y. Zeng, "Numerical Study on Pipeline Pressure Surge of the Large Aircraft Fuel System," *Shock and Vibration*, vol. 2022, p. e7529857, 05 2022.

[19] W. Krüger, B. Gerlinger, O. Brodersen, T. Klimmek, and Y. Günther, "Das DLR-Projekt Kontekst: Konzepte und Technologien für emissionsarme Kurzstreckenflugzeuge," 2020.

[20] B. Boden, Y. Cabac, T. Burschyk, and B. Nagel, "Rule-based Verification of a Geometric Design using the Codex Framework," *33rd Congress of the International Council of the Aeronautical Sciences, ICAS 2022*, 11 2022.

[21] A. Corallo, R. Laubacher, A. Margherita, and G. Turrisi, "Enhancing product development through knowledge-based engineering (KBE)," *Journal of Manufacturing Technology Management*, vol. 20, pp. 1070–1083, 10 2009.

[22] Airbus, "A321XLR — Xtra Long Range performance — Airbus Aircraft," 05 2022.

[23] IJARIIE, *Review of Aircraft Fuel System* , vol. 1, 2015.

[24] European Union Aviation Safety Agency (EASA), "Certification Specifications and Acceptable Means of Compliance for Large Aeroplanes CS-25," 12 2020.

[25] M. D. Fernandes, S. T. de P. Andrade, V. N. Bistritzki, R. M. Fonseca, L. G. Zacarias, H. N. C. Gonçalves, A. F. de Castro, R. Z. Domingues, and T. Matencio, "SOFC-APU systems for aircraft: A review," *International Journal of Hydrogen Energy*, vol. 43, p. 1631116333, 08 2018.

[26] T. R. Quackenbush, M. E. Teske, and C. E. Polymeropoulos, "A model for assessing fuel jettisoning effects," *Atmospheric Environment*, vol. 28, no. 16, pp. 2751–2759, 1994.

[27] Z. Goraj and P. Zakrzewski, "Aircraft fuel systems and their influence on stability margin," pp. 29–40, 01 2005.

[28] A. La Bella, D. Canzano, and M. Grimaldi, "Critical capabilities and performance in the aerospace industry: a knowledge management approach," 03 2004.

[29] J. P. Risueno and B. Nagel, "Development of a Knowledge-Based Engineering Framework for Modeling Aircraft Production," *AIAA Aviation Forum*, 06 2019.

[30] G. L. Rocca and M. V. Tooren, "Enabling distributed multi-disciplinary design of complex products: a knowledge based engineering approach," *J. of Design Research*, vol. 5, p. 333, 2007.

[31] R. Suryaprakash, "Knowledge Based Engineering (KBE) in Aerospace Product Development," *AIAA*, 11 2007.

[32] A. van der Lann, *Knowledge based engineering support for aircraft component design*. PhD thesis, 01 2008.

[33] C. van der Velden, C. Bil, and X. Xu, "Adaptable methodology for automation application development," *Advanced Engineering Informatics*, vol. 26, pp. 231–250, 04 2012.

[34] C. Van der Velden, C. Bil, X. Yu, and A. Smith, "An intelligent system for automatic layout routing in aerospace design," *Innovations in Systems and Software Engineering*, vol. 3, pp. 117–128, 05 2007.

[35] S. Quintana-Amate et al., "A new knowledge sourcing framework for knowledge-based engineering: An aerospace industry case study," *Computers & Industrial Engineering*, vol. 104, pp. 35–50, 02 2017.

[36] R. Klein, "Knowledge Modeling In Design — The MOKA Framework," *Artificial Intelligence in Design*, pp. 77–102, 01 2000.

[37] D. Baxter et al., "An engineering design knowledge reuse methodology using process modelling," *Research in Engineering Design*, vol. 18, pp. 37–48, 04 2007.

[38] C. B. Chapman and M. Pinfold, "The application of a knowledge based engineering approach to the rapid design and analysis of an automotive structure," *Advances in Engineering Software*, vol. 32, pp. 903–912, 12 2001.

[39] S. Ahmed and K. Wallace, "Understanding the knowledge needs of novice designers in the aerospace industry," *Design Studies*, vol. 25, pp. 155–173, 03 2004.

[40] D. C. Ramanigopal, "Knowledge Management Strategies for Successful Implemention in Aerospace Industry," *INTERNATIONAL JOURNAL OF MANAGEMENT RESEARCH AND REVIEW*, vol. 2, 10 2012.

[41] W3C, "Standards - W3C," 2019.

[42] D. Allemang and J. Hendler, *Semantic Web for the Working Ontologist*. Elsevier, 07 2011.

[43] Opencascade, "Introduction - Open CASCADE Technology Documentation."

[44] M. Siggel, J. Kleinert, T. Stollenwerk, and R. Maierl, "TiGL: An Open Source Computational Geometry Library for Parametric Aircraft Design," *Mathematics in Computer Science*, vol. 13, pp. 367–389, July 2019.

[45] M. Alder, E. Moerland, J. Jepsen, and B. Nagel, "Recent Advances in Establishing a Common Language for Aircraft Design with CPACS," in *Aerospace Europe Conference 2020*, 2020.

[46] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[47] Lufthansa Technical Training LTT, "Training Manual Airbus A318/A319/A320/A321 - ATA 28 FUEL," 06 2020.

[48] SAE International, "AS18802 - Fuel and Oil Lines, Aircraft, Installation of," 08 2013.

[49] MTU Aero Engines, "PW6000 - MTU Aero Engines."

Appendix A

DLR-D150's resulting statements from the GeoVerification tool

This appendix provides the lists of falsified statements for each of the iterations performed to correctly model the geometry. For the last iteration, the resulting list of complete statements is provided. In addition, a list containing the certification related statements is given.

A.1 Geometry verification - falsified statements

A.1.1 First iteration

Table A.1: Falsified geometric statements - first iteration

line 11 - right engine feed line after Valve 4 fuelsystem:isNotOverlappingWithTankFalsified Right feed tank
line 12 - Left engine feed line after Valve 5 fuelsystem:isNotOverlappingWithTankFalsified Left feed tank
line 12 - Left engine feed line after Valve 5 geoverification:connectedToFalsified ShutOff valve right connection port
line 12 - Left engine feed line after Valve 5 geoverification:connectedToFalsified Valve 5
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Left feed tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Left transfer tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Right feed tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Right transfer tank
line 2 from Right boost pump in center tank geoverification:connectedToFalsified line 7 - crosfeed from Valve 1 to Right boost pump in center tank
line 7 - crosfeed from Valve 1 to Right boost pump in center tank geoverification:connectedToFalsified line 2 from Right boost pump in center tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankFalsified Left feed tank
line 9 - feed line from Valve 3 to Valve 4 geoverification:connectedToFalsified ShutOff valve left connection port
line 9 - feed line from Valve 3 to Valve 4 geoverification:connectedToFalsified Valve 5
ShutOff valve left connection port geoverification:connectedToFalsified line 9 - feed line from Valve 3 to Valve 4
ShutOff valve right connection port geoverification:connectedToFalsified line 12 - Left engine feed line after Valve 5
Valve 5 geoverification:connectedToFalsified line 12 - Left engine feed line after Valve 5
Valve 5 geoverification:connectedToFalsified line 9 - feed line from Valve 3 to Valve 4
Valve 5 geoverification:containedInFalsified Left feed tank

A.1.2 Second iteration

Table A.2: Falsified geometric statements - second iteration

line 10 - feed line from Valve 2 to Valve 5 fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 10 - feed line from Valve 2 to Valve 5 fuelsystem:isNotOverlappingWithTankFalsified Right feed tank
line 11 - right engine feed line after Valve 4 fuelsystem:isNotOverlappingWithTankFalsified Right feed tank
line 12 - Left engine feed line after Valve 5 fuelsystem:isNotOverlappingWithTankFalsified Left feed tank
line 12 - Left engine feed line after Valve 5 geoverification:connectedToFalsified ShutOff valve right connection port
line 12 - Left engine feed line after Valve 5 geoverification:connectedToFalsified Valve 5
line 14 - from spar-mounted APU pump in the center tank - left side to APU unit fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Left feed tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Left transfer tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Right feed tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Right transfer tank
line 2 from Right boost pump in center tank geoverification:connectedToFalsified line 7 - crosfeed from Valve 1 to Right boost pump in center tank
line 7 - crosfeed from Valve 1 to Right boost pump in center tank geoverification:connectedToFalsified line 2 from Right boost pump in center tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankFalsified Left feed tank
line 9 - feed line from Valve 3 to Valve 4 geoverification:connectedToFalsified ShutOff valve left connection port
line 9 - feed line from Valve 3 to Valve 4 geoverification:connectedToFalsified Valve 5
ShutOff valve left connection port geoverification:connectedToFalsified line 9 - feed line from Valve 3 to Valve 4
ShutOff valve right connection port geoverification:connectedToFalsified line 12 - Left engine feed line after Valve 5
Valve 5 geoverification:connectedToFalsified line 12 - Left engine feed line after Valve 5
Valve 5 geoverification:connectedToFalsified line 9 - feed line from Valve 3 to Valve 4
Valve 5 geoverification:containedInFalsified Left feed tank

A.1.3 Third iteration

Table A.3: Falsified geometric statements - third iteration

line 10 - feed line from Valve 2 to Valve 5 fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 10 - feed line from Valve 2 to Valve 5 fuelsystem:isNotOverlappingWithTankFalsified Right feed tank
line 11 - right engine feed line after Valve 4 fuelsystem:isNotOverlappingWithTankFalsified Right feed tank
line 12 - Left engine feed line after Valve 5 fuelsystem:isNotOverlappingWithTankFalsified Left feed tank
line 14 - from spar-mounted APU pump in the center tank - left side to APU unit fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Left feed tank

line 17 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Left transfer tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Right feed tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankFalsified Right transfer tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankFalsified Center feed tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankFalsified Left feed tank

A.2 Geometry verification - Resulting statements

Table A.4: Verified geometric statements - last iteration

Boost pump 1 in left inner tank fuelsystem:isContainedBetweenRibsVerified Fuel System
Boost pump 1 in left inner tank geoverification:attachedToVerified Left feed tank
Boost pump 1 in left inner tank geoverification:connectedToVerified line 5 from Boost pump 1 in left inner tank
Boost pump 1 in left inner tank geoverification:containedInVerified Left feed tank
Boost pump 1 in right inner tank fuelsystem:isContainedBetweenRibsVerified Fuel System
Boost pump 1 in right inner tank geoverification:attachedToVerified Right feed tank
Boost pump 1 in right inner tank geoverification:connectedToVerified line 3 from Boost pump 1 in right inner tank
Boost pump 1 in right inner tank geoverification:containedInVerified Right feed tank
Boost pump 2 in left inner tank fuelsystem:isContainedBetweenRibsVerified Fuel System
Boost pump 2 in left inner tank geoverification:attachedToVerified Left feed tank
Boost pump 2 in left inner tank geoverification:connectedToVerified line 6 from Boost pump 2 in left inner tank
Boost pump 2 in left inner tank geoverification:containedInVerified Left feed tank
Boost pump 2 in right inner tank fuelsystem:isContainedBetweenRibsVerified Fuel System
Boost pump 2 in right inner tank geoverification:attachedToVerified Right feed tank
Boost pump 2 in right inner tank geoverification:connectedToVerified line 4 from Boost pump 2 in right inner tank
Boost pump 2 in right inner tank geoverification:containedInVerified Right feed tank
Center feed tank geoverification:attachedToVerified Left boost pump in center tank
Center feed tank geoverification:attachedToVerified Right boost pump in center tank
Left boost pump in center tank fuelsystem:isContainedBetweenRibsVerified Fuel System
Left boost pump in center tank geoverification:attachedToVerified Center feed tank
Left boost pump in center tank geoverification:connectedToVerified line 1 from Left boost pump in center tank
Left boost pump in center tank geoverification:containedInVerified Center feed tank
Left feed tank geoverification:attachedToVerified Boost pump 1 in left inner tank
Left feed tank geoverification:attachedToVerified Boost pump 2 in left inner tank
line 1 from Left boost pump in center tank fuelsystem:hasMatchingSizesVerified Port 1 in Left boost pump in center tank
line 1 from Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 1 from Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 1 from Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 1 from Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 1 from Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 1 from Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 1 from Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 1 from Left boost pump in center tank geoverification:connectedToVerified Left boost pump in center tank
line 1 from Left boost pump in center tank geoverification:connectedToVerified line 8 - crosfeed from Valve 1 to Left boost pump in center tank
line 1 from Left boost pump in center tank geoverification:connectedToVerified Port 1 in Left boost pump in center tank
line 10 - feed line from Valve 2 to Valve 5 fuelsystem:isNotOverlappingWithTankVerified Center feed tank

line 10 - feed line from Valve 2 to Valve 5 fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 10 - feed line from Valve 2 to Valve 5 fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 10 - feed line from Valve 2 to Valve 5 fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 10 - feed line from Valve 2 to Valve 5 fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 10 - feed line from Valve 2 to Valve 5 fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 10 - feed line from Valve 2 to Valve 5 fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 10 - feed line from Valve 2 to Valve 5 geoverification:connectedToVerified line 3 from Boost pump 1 in right inner tank
line 10 - feed line from Valve 2 to Valve 5 geoverification:connectedToVerified line 4 from Boost pump 2 in right inner tank
line 10 - feed line from Valve 2 to Valve 5 geoverification:connectedToVerified ShutOff valve left connection port
line 10 - feed line from Valve 2 to Valve 5 geoverification:connectedToVerified ShutOff valve right connection port
line 10 - feed line from Valve 2 to Valve 5 geoverification:connectedToVerified Valve 2
line 10 - feed line from Valve 2 to Valve 5 geoverification:connectedToVerified Valve 4
line 11 - right engine feed line after Valve 4 fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 11 - right engine feed line after Valve 4 fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 11 - right engine feed line after Valve 4 fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 11 - right engine feed line after Valve 4 fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 11 - right engine feed line after Valve 4 fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 11 - right engine feed line after Valve 4 fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 11 - right engine feed line after Valve 4 geoverification:connectedToVerified ShutOff valve left connection port
line 11 - right engine feed line after Valve 4 geoverification:connectedToVerified Valve 4
line 12 - Left engine feed line after Valve 5 fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 12 - Left engine feed line after Valve 5 fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 12 - Left engine feed line after Valve 5 fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 12 - Left engine feed line after Valve 5 fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 12 - Left engine feed line after Valve 5 fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 12 - Left engine feed line after Valve 5 fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 12 - Left engine feed line after Valve 5 geoverification:connectedToVerified ShutOff valve left connection port
line 12 - Left engine feed line after Valve 5 geoverification:connectedToVerified Valve 5
line 13 - from feed line to spar-mounted APU pump in the center tank - left side fuelsystem:hasMatchingSizesVerified Port 7 in the spar-mounted APU pump in the center tank - left side
line 13 - from feed line to spar-mounted APU pump in the center tank - left side fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 13 - from feed line to spar-mounted APU pump in the center tank - left side fuelsystem:isNotOverlappingWithTankVerified Left feed tank

line 13 - from feed line to spar-mounted APU pump in the center tank - left side fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 13 - from feed line to spar-mounted APU pump in the center tank - left side fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 13 - from feed line to spar-mounted APU pump in the center tank - left side fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 13 - from feed line to spar-mounted APU pump in the center tank - left side fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 13 - from feed line to spar-mounted APU pump in the center tank - left side fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 13 - from feed line to spar-mounted APU pump in the center tank - left side geoverification:connectedToVerified line 8 - crosfeed from Valve 1 to Left boost pump in center tank
line 14 - from spar-mounted APU pump in the center tank - left side to APU unit fuelsystem:hasMatchingSizesVerified Port 8 in the spar-mounted APU pump in the center tank - left side
line 14 - from spar-mounted APU pump in the center tank - left side to APU unit fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 14 - from spar-mounted APU pump in the center tank - left side to APU unit fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 14 - from spar-mounted APU pump in the center tank - left side to APU unit fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 14 - from spar-mounted APU pump in the center tank - left side to APU unit fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 14 - from spar-mounted APU pump in the center tank - left side to APU unit fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 14 - from spar-mounted APU pump in the center tank - left side to APU unit fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 14 - from spar-mounted APU pump in the center tank - left side to APU unit fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 15 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 15 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 15 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 15 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 15 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 15 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 15 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 15 - crossfeed parallel line geoverification:connectedToVerified line 8 - crosfeed from Valve 1 to Left boost pump in center tank
line 15 - crossfeed parallel line geoverification:connectedToVerified ShutOff valve right connection port
line 15 - crossfeed parallel line geoverification:connectedToVerified Valve 6
line 16 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 16 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 16 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 16 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 16 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 16 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 16 - crossfeed parallel line fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 16 - crossfeed parallel line geoverification:connectedToVerified line 7 - crosfeed from Valve 1 to Right boost pump in center tank
line 16 - crossfeed parallel line geoverification:connectedToVerified ShutOff valve left connection port
line 16 - crossfeed parallel line geoverification:connectedToVerified Valve 6
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 17 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Right transfer tank

line 17 - refueling main line geoverification:connectedToVerified line 19 - secondary refueling line from main refuel line Valve 7
line 17 - refueling main line geoverification:connectedToVerified line 21 - secondary refueling line from main refuel line Valve 8
line 17 - refueling main line geoverification:connectedToVerified line 23 - secondary refueling line from main refuel line Valve 9
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 18 - refueling main line fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 18 - refueling main line geoverification:connectedToVerified line 25 - secondary refueling line from main refuel line Valve 10
line 18 - refueling main line geoverification:connectedToVerified line 27 - secondary refueling line from main refuel line Valve 11
line 18 - refueling main line geoverification:connectedToVerified line 29 - secondary refueling line from main refuel line Valve 12
line 19 - secondary refueling line from main refuel line Valve 7 fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 19 - secondary refueling line from main refuel line Valve 7 fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 19 - secondary refueling line from main refuel line Valve 7 fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 19 - secondary refueling line from main refuel line Valve 7 fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 19 - secondary refueling line from main refuel line Valve 7 fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 19 - secondary refueling line from main refuel line Valve 7 fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 19 - secondary refueling line from main refuel line Valve 7 fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 19 - secondary refueling line from main refuel line Valve 7 geoverification:connectedToVerified line 17 - refueling main line
line 19 - secondary refueling line from main refuel line Valve 7 geoverification:connectedToVerified ShutOff valve left connection port
line 19 - secondary refueling line from main refuel line Valve 7 geoverification:connectedToVerified Valve 7
line 2 from Right boost pump in center tank fuelsystem:hasMatchingSizesVerified Port 2 in Right boost pump in center tank
line 2 from Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 2 from Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 2 from Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 2 from Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 2 from Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 2 from Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 2 from Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 2 from Right boost pump in center tank geoverification:connectedToVerified line 7 - crosfeed from Valve 1 to Right boost pump in center tank

line 2 from Right boost pump in center tank geoverification:connectedToVerified Port 2 in Right boost pump in center tank

line 2 from Right boost pump in center tank geoverification:connectedToVerified Right boost pump in center tank

line 20 - secondary refueling line from Valve 7 to Left transfer tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank

line 20 - secondary refueling line from Valve 7 to Left transfer tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank

line 20 - secondary refueling line from Valve 7 to Left transfer tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank

line 20 - secondary refueling line from Valve 7 to Left transfer tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank

line 20 - secondary refueling line from Valve 7 to Left transfer tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank

line 20 - secondary refueling line from Valve 7 to Left transfer tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank

line 20 - secondary refueling line from Valve 7 to Left transfer tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank

line 20 - secondary refueling line from Valve 7 to Left transfer tank geoverification:connectedToVerified ShutOff valve right connection port

line 20 - secondary refueling line from Valve 7 to Left transfer tank geoverification:connectedToVerified Valve 7

line 21 - secondary refueling line from main refuel line Valve 8 fuelsystem:isNotOverlappingWithTankVerified Center feed tank

line 21 - secondary refueling line from main refuel line Valve 8 fuelsystem:isNotOverlappingWithTankVerified Left feed tank

line 21 - secondary refueling line from main refuel line Valve 8 fuelsystem:isNotOverlappingWithTankVerified Left surge tank

line 21 - secondary refueling line from main refuel line Valve 8 fuelsystem:isNotOverlappingWithTankVerified Left transfer tank

line 21 - secondary refueling line from main refuel line Valve 8 fuelsystem:isNotOverlappingWithTankVerified Right feed tank

line 21 - secondary refueling line from main refuel line Valve 8 fuelsystem:isNotOverlappingWithTankVerified Right surge tank

line 21 - secondary refueling line from main refuel line Valve 8 fuelsystem:isNotOverlappingWithTankVerified Right transfer tank

line 21 - secondary refueling line from main refuel line Valve 8 geoverification:connectedToVerified line 17 - refueling main line

line 21 - secondary refueling line from main refuel line Valve 8 geoverification:connectedToVerified ShutOff valve left connection port

line 21 - secondary refueling line from main refuel line Valve 8 geoverification:connectedToVerified Valve 8

line 22 - secondary refueling line from Valve 8 to Left feed tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank

line 22 - secondary refueling line from Valve 8 to Left feed tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank

line 22 - secondary refueling line from Valve 8 to Left feed tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank

line 22 - secondary refueling line from Valve 8 to Left feed tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank

line 22 - secondary refueling line from Valve 8 to Left feed tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank

line 22 - secondary refueling line from Valve 8 to Left feed tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank

line 22 - secondary refueling line from Valve 8 to Left feed tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank

line 22 - secondary refueling line from Valve 8 to Left feed tank geoverification:connectedToVerified ShutOff valve right connection port

line 22 - secondary refueling line from Valve 8 to Left feed tank geoverification:connectedToVerified Valve 8
line 23 - secondary refueling line from main refuel line Valve 9 fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 23 - secondary refueling line from main refuel line Valve 9 fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 23 - secondary refueling line from main refuel line Valve 9 fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 23 - secondary refueling line from main refuel line Valve 9 fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 23 - secondary refueling line from main refuel line Valve 9 fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 23 - secondary refueling line from main refuel line Valve 9 fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 23 - secondary refueling line from main refuel line Valve 9 fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 23 - secondary refueling line from main refuel line Valve 9 geoverification:connectedToVerified line 17 - refueling main line
line 23 - secondary refueling line from main refuel line Valve 9 geoverification:connectedToVerified ShutOff valve left connection port
line 23 - secondary refueling line from main refuel line Valve 9 geoverification:connectedToVerified Valve 9
line 24 - secondary refueling line from Valve 9 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 24 - secondary refueling line from Valve 9 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 24 - secondary refueling line from Valve 9 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 24 - secondary refueling line from Valve 9 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 24 - secondary refueling line from Valve 9 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 24 - secondary refueling line from Valve 9 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 24 - secondary refueling line from Valve 9 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 24 - secondary refueling line from Valve 9 to Center feed tank geoverification:connectedToVerified ShutOff valve right connection port
line 24 - secondary refueling line from Valve 9 to Center feed tank geoverification:connectedToVerified Valve 9
line 25 - secondary refueling line from main refuel line Valve 10 fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 25 - secondary refueling line from main refuel line Valve 10 fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 25 - secondary refueling line from main refuel line Valve 10 fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 25 - secondary refueling line from main refuel line Valve 10 fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 25 - secondary refueling line from main refuel line Valve 10 fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 25 - secondary refueling line from main refuel line Valve 10 fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 25 - secondary refueling line from main refuel line Valve 10 fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 25 - secondary refueling line from main refuel line Valve 10 geoverification:connectedToVerified line 18 - refueling main line
line 25 - secondary refueling line from main refuel line Valve 10 geoverification:connectedToVerified ShutOff valve left connection port

line 25 - secondary refueling line from main refuel line Valve 10 geoverification:connectedToVerified Valve 10
line 26 - secondary refueling line from Valve 10 to Right transfer tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 26 - secondary refueling line from Valve 10 to Right transfer tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 26 - secondary refueling line from Valve 10 to Right transfer tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 26 - secondary refueling line from Valve 10 to Right transfer tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 26 - secondary refueling line from Valve 10 to Right transfer tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 26 - secondary refueling line from Valve 10 to Right transfer tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 26 - secondary refueling line from Valve 10 to Right transfer tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 26 - secondary refueling line from Valve 10 to Right transfer tank geoverification:connectedToVerified ShutOff valve right connection port
line 26 - secondary refueling line from Valve 10 to Right transfer tank geoverification:connectedToVerified Valve 10
line 27 - secondary refueling line from main refuel line Valve 11 fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 27 - secondary refueling line from main refuel line Valve 11 fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 27 - secondary refueling line from main refuel line Valve 11 fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 27 - secondary refueling line from main refuel line Valve 11 fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 27 - secondary refueling line from main refuel line Valve 11 fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 27 - secondary refueling line from main refuel line Valve 11 fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 27 - secondary refueling line from main refuel line Valve 11 fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 27 - secondary refueling line from main refuel line Valve 11 geoverification:connectedToVerified line 18 - refueling main line
line 27 - secondary refueling line from main refuel line Valve 11 geoverification:connectedToVerified ShutOff valve left connection port
line 27 - secondary refueling line from main refuel line Valve 11 geoverification:connectedToVerified Valve 11
line 28 - secondary refueling line from Valve 11 to Right feed tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 28 - secondary refueling line from Valve 11 to Right feed tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 28 - secondary refueling line from Valve 11 to Right feed tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 28 - secondary refueling line from Valve 11 to Right feed tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 28 - secondary refueling line from Valve 11 to Right feed tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 28 - secondary refueling line from Valve 11 to Right feed tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 28 - secondary refueling line from Valve 11 to Right feed tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 28 - secondary refueling line from Valve 11 to Right feed tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 28 - secondary refueling line from Valve 11 to Right feed tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank

line 28 - secondary refueling line from Valve 11 to Right feed tank geoverification:connectedToVerified ShutOff valve right connection port
line 28 - secondary refueling line from Valve 11 to Right feed tank geoverification:connectedToVerified Valve 11
line 29 - secondary refueling line from main refuel line Valve 12 fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 29 - secondary refueling line from main refuel line Valve 12 fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 29 - secondary refueling line from main refuel line Valve 12 fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 29 - secondary refueling line from main refuel line Valve 12 fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 29 - secondary refueling line from main refuel line Valve 12 fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 29 - secondary refueling line from main refuel line Valve 12 fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 29 - secondary refueling line from main refuel line Valve 12 fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 29 - secondary refueling line from main refuel line Valve 12 geoverification:connectedToVerified line 18 - refueling main line
line 29 - secondary refueling line from main refuel line Valve 12 geoverification:connectedToVerified ShutOff valve left connection port
line 29 - secondary refueling line from main refuel line Valve 12 geoverification:connectedToVerified Valve 12
line 3 from Boost pump 1 in right inner tank fuelsystem:hasMatchingSizesVerified Port 3 in Boost pump 1 in right inner tank
line 3 from Boost pump 1 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 3 from Boost pump 1 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 3 from Boost pump 1 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 3 from Boost pump 1 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 3 from Boost pump 1 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 3 from Boost pump 1 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 3 from Boost pump 1 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 3 from Boost pump 1 in right inner tank geoverification:connectedToVerified Boost pump 1 in right inner tank
line 3 from Boost pump 1 in right inner tank geoverification:connectedToVerified line 10 - feed line from Valve 2 to Valve 5
line 3 from Boost pump 1 in right inner tank geoverification:connectedToVerified Port 3 in Boost pump 1 in right inner tank
line 30 - secondary refueling line from Valve 12 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 30 - secondary refueling line from Valve 12 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 30 - secondary refueling line from Valve 12 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 30 - secondary refueling line from Valve 12 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 30 - secondary refueling line from Valve 12 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 30 - secondary refueling line from Valve 12 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank

line 30 - secondary refueling line from Valve 12 to Center feed tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 30 - secondary refueling line from Valve 12 to Center feed tank geoverification:connectedToVerified ShutOff valve right connection port
line 30 - secondary refueling line from Valve 12 to Center feed tank geoverification:connectedToVerified Valve 12
line 4 from Boost pump 2 in right inner tank fuelsystem:hasMatchingSizesVerified Port 4 in Boost pump 2 in right inner tank
line 4 from Boost pump 2 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 4 from Boost pump 2 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 4 from Boost pump 2 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 4 from Boost pump 2 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 4 from Boost pump 2 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 4 from Boost pump 2 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 4 from Boost pump 2 in right inner tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 4 from Boost pump 2 in right inner tank geoverification:connectedToVerified Boost pump 2 in right inner tank
line 4 from Boost pump 2 in right inner tank geoverification:connectedToVerified line 10 - feed line from Valve 2 to Valve 5
line 4 from Boost pump 2 in right inner tank geoverification:connectedToVerified Port 4 in Boost pump 2 in right inner tank
line 5 from Boost pump 1 in left inner tank fuelsystem:hasMatchingSizesVerified Port 5 in Boost pump 1 in left inner tank
line 5 from Boost pump 1 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 5 from Boost pump 1 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 5 from Boost pump 1 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 5 from Boost pump 1 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 5 from Boost pump 1 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 5 from Boost pump 1 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 5 from Boost pump 1 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 5 from Boost pump 1 in left inner tank geoverification:connectedToVerified Boost pump 1 in left inner tank
line 5 from Boost pump 1 in left inner tank geoverification:connectedToVerified line 9 - feed line from Valve 3 to Valve 4
line 5 from Boost pump 1 in left inner tank geoverification:connectedToVerified Port 5 in Boost pump 1 in left inner tank
line 6 from Boost pump 2 in left inner tank fuelsystem:hasMatchingSizesVerified Port 6 in Boost pump 2 in left inner tank
line 6 from Boost pump 2 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 6 from Boost pump 2 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 6 from Boost pump 2 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank

line 6 from Boost pump 2 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank

line 6 from Boost pump 2 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank

line 6 from Boost pump 2 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank

line 6 from Boost pump 2 in left inner tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank

line 6 from Boost pump 2 in left inner tank geoverification:connectedToVerified Boost pump 2 in left inner tank

line 6 from Boost pump 2 in left inner tank geoverification:connectedToVerified line 9 - feed line from Valve 3 to Valve 4

line 6 from Boost pump 2 in left inner tank geoverification:connectedToVerified Port 6 in Boost pump 2 in left inner tank

line 7 - crosfeed from Valve 1 to Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank

line 7 - crosfeed from Valve 1 to Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank

line 7 - crosfeed from Valve 1 to Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank

line 7 - crosfeed from Valve 1 to Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank

line 7 - crosfeed from Valve 1 to Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank

line 7 - crosfeed from Valve 1 to Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank

line 7 - crosfeed from Valve 1 to Right boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank

line 7 - crosfeed from Valve 1 to Right boost pump in center tank geoverification:connectedToVerified line 16 - crossfeed parallel line

line 7 - crosfeed from Valve 1 to Right boost pump in center tank geoverification:connectedToVerified line 2 from Right boost pump in center tank

line 7 - crosfeed from Valve 1 to Right boost pump in center tank geoverification:connectedToVerified ShutOff valve left connection port

line 7 - crosfeed from Valve 1 to Right boost pump in center tank geoverification:connectedToVerified ShutOff valve right connection port

line 7 - crosfeed from Valve 1 to Right boost pump in center tank geoverification:connectedToVerified Valve 1

line 7 - crosfeed from Valve 1 to Right boost pump in center tank geoverification:connectedToVerified Valve 2

line 8 - crosfeed from Valve 1 to Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Center feed tank

line 8 - crosfeed from Valve 1 to Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left feed tank

line 8 - crosfeed from Valve 1 to Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left surge tank

line 8 - crosfeed from Valve 1 to Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Left transfer tank

line 8 - crosfeed from Valve 1 to Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right feed tank

line 8 - crosfeed from Valve 1 to Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right surge tank

line 8 - crosfeed from Valve 1 to Left boost pump in center tank fuelsystem:isNotOverlappingWithTankVerified Right transfer tank

line 8 - crosfeed from Valve 1 to Left boost pump in center tank geoverification:connectedToVerified line 1 from Left boost pump in center tank

line 8 - crosfeed from Valve 1 to Left boost pump in center tank geoverification:connectedToVerified line 13 - from feed line to spar-mounted APU pump in the center tank - left side

line 8 - crosfeed from Valve 1 to Left boost pump in center tank geoverification:connectedToVerified line 15 - crossfeed parallel line
line 8 - crosfeed from Valve 1 to Left boost pump in center tank geoverification:connectedToVerified ShutOff valve right connection port
line 8 - crosfeed from Valve 1 to Left boost pump in center tank geoverification:connectedToVerified ShutOff valve right connection port
line 8 - crosfeed from Valve 1 to Left boost pump in center tank geoverification:connectedToVerified Valve 1
line 8 - crosfeed from Valve 1 to Left boost pump in center tank geoverification:connectedToVerified Valve 3
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankVerified Center feed tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankVerified Left feed tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankVerified Left surge tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankVerified Left transfer tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankVerified Right feed tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankVerified Right surge tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:isNotOverlappingWithTankVerified Right transfer tank
line 9 - feed line from Valve 3 to Valve 4 geoverification:connectedToVerified line 5 from Boost pump 1 in left inner tank
line 9 - feed line from Valve 3 to Valve 4 geoverification:connectedToVerified line 6 from Boost pump 2 in left inner tank
line 9 - feed line from Valve 3 to Valve 4 geoverification:connectedToVerified ShutOff valve left connection port
line 9 - feed line from Valve 3 to Valve 4 geoverification:connectedToVerified ShutOff valve right connection port
line 9 - feed line from Valve 3 to Valve 4 geoverification:connectedToVerified Valve 3
line 9 - feed line from Valve 3 to Valve 4 geoverification:connectedToVerified Valve 5
Port 1 in Left boost pump in center tank fuelsystem:hasMatchingSizesVerified line 1 from Left boost pump in center tank
Port 1 in Left boost pump in center tank geoverification:connectedToVerified line 1 from Left boost pump in center tank
Port 2 in Right boost pump in center tank fuelsystem:hasMatchingSizesVerified line 2 from Right boost pump in center tank
Port 2 in Right boost pump in center tank geoverification:connectedToVerified line 2 from Right boost pump in center tank
Port 3 in Boost pump 1 in right inner tank fuelsystem:hasMatchingSizesVerified line 3 from Boost pump 1 in right inner tank
Port 3 in Boost pump 1 in right inner tank geoverification:connectedToVerified line 3 from Boost pump 1 in right inner tank
Port 4 in Boost pump 2 in right inner tank fuelsystem:hasMatchingSizesVerified line 4 from Boost pump 2 in right inner tank
Port 4 in Boost pump 2 in right inner tank geoverification:connectedToVerified line 4 from Boost pump 2 in right inner tank
Port 5 in Boost pump 1 in left inner tank fuelsystem:hasMatchingSizesVerified line 5 from Boost pump 1 in left inner tank
Port 5 in Boost pump 1 in left inner tank geoverification:connectedToVerified line 5 from Boost pump 1 in left inner tank
Port 6 in Boost pump 2 in left inner tank fuelsystem:hasMatchingSizesVerified line 6 from Boost pump 2 in left inner tank
Port 6 in Boost pump 2 in left inner tank geoverification:connectedToVerified line 6 from Boost pump 2 in left inner tank

Port 7 in the spar-mounted APU pump in the center tank - left side fuelsystem:hasMatchingSizesVerified line 13 - from feed line to spar-mounted APU pump in the center tank - left side

Port 8 in the spar-mounted APU pump in the center tank - left side fuelsystem:hasMatchingSizesVerified line 14 - from spar-mounted APU pump in the center tank - left side to APU unit

Right boost pump in center tank fuelsystem:isContainedBetweenRibsVerified Fuel System

Right boost pump in center tank geoverification:attachedToVerified Center feed tank

Right boost pump in center tank geoverification:connectedToVerified line 2 from Right boost pump in center tank

Right boost pump in center tank geoverification:containedInVerified Center feed tank

Right feed tank geoverification:attachedToVerified Boost pump 1 in right inner tank

Right feed tank geoverification:attachedToVerified Boost pump 2 in right inner tank

ShutOff valve left connection port geoverification:connectedToVerified line 10 - feed line from Valve 2 to Valve 5

ShutOff valve left connection port geoverification:connectedToVerified line 11 - right engine feed line after Valve 4

ShutOff valve left connection port geoverification:connectedToVerified line 12 - Left engine feed line after Valve 5

ShutOff valve left connection port geoverification:connectedToVerified line 16 - crossfeed parallel line

ShutOff valve left connection port geoverification:connectedToVerified line 19 - secondary refueling line from main refuel line Valve 7

ShutOff valve left connection port geoverification:connectedToVerified line 21 - secondary refueling line from main refuel line Valve 8

ShutOff valve left connection port geoverification:connectedToVerified line 23 - secondary refueling line from main refuel line Valve 9

ShutOff valve left connection port geoverification:connectedToVerified line 25 - secondary refueling line from main refuel line Valve 10

ShutOff valve left connection port geoverification:connectedToVerified line 27 - secondary refueling line from main refuel line Valve 11

ShutOff valve left connection port geoverification:connectedToVerified line 29 - secondary refueling line from main refuel line Valve 12

ShutOff valve left connection port geoverification:connectedToVerified line 7 - crosfeed from Valve 1 to Right boost pump in center tank

ShutOff valve left connection port geoverification:connectedToVerified line 9 - feed line from Valve 3 to Valve 4

ShutOff valve right connection port geoverification:connectedToVerified line 10 - feed line from Valve 2 to Valve 5

ShutOff valve right connection port geoverification:connectedToVerified line 15 - crossfeed parallel line

ShutOff valve right connection port geoverification:connectedToVerified line 20 - secondary refueling line from Valve 7 to Left transfer tank

ShutOff valve right connection port geoverification:connectedToVerified line 22 - secondary refueling line from Valve 8 to Left feed tank

ShutOff valve right connection port geoverification:connectedToVerified line 24 - secondary refueling line from Valve 9 to Center feed tank

ShutOff valve right connection port geoverification:connectedToVerified line 26 - secondary refueling line from Valve 10 to Right transfer tank

ShutOff valve right connection port geoverification:connectedToVerified line 28 - secondary refueling line from Valve 11 to Right feed tank

ShutOff valve right connection port geoverification:connectedToVerified line 30 - secondary refueling line from Valve 12 to Center feed tank

ShutOff valve right connection port geoverification:connectedToVerified line 7 - crosfeed from Valve 1 to Right boost pump in center tank

ShutOff valve right connection port geoverification:connectedToVerified line 8 - crosfeed from Valve 1 to Left boost pump in center tank

ShutOff valve right connection port geoverification:connectedToVerified line 8 - crosfeed from Valve 1 to Left boost pump in center tank

ShutOff valve right connection port geoverification:connectedToVerified line 9 - feed line from Valve 3 to Valve 4
spar-mounted APU pump in the center tank - left side fuelsystem:isContainedBetweenRibsVerified Fuel System
spar-mounted APU pump in the center tank - left side geoverification:containedInVerified Center feed tank
Tank opening 1 geoverification:containedInVerified Center feed tank
Tank opening 10 geoverification:containedInVerified Right feed tank
Tank opening 11 geoverification:containedInVerified Center feed tank
Tank opening 12 geoverification:containedInVerified Center feed tank
Tank opening 13 geoverification:containedInVerified Right feed tank
Tank opening 14 geoverification:containedInVerified Right feed tank
Tank opening 15 geoverification:containedInVerified Right transfer tank
Tank opening 2 geoverification:containedInVerified Left feed tank
Tank opening 3 geoverification:containedInVerified Center feed tank
Tank opening 4 geoverification:containedInVerified Right feed tank
Tank opening 5 geoverification:containedInVerified Center feed tank
Tank opening 6 geoverification:containedInVerified Left feed tank
Tank opening 7 geoverification:containedInVerified Left feed tank
Tank opening 8 geoverification:containedInVerified Left transfer tank
Tank opening 9 geoverification:containedInVerified Left feed tank
Valve 1 geoverification:connectedToVerified line 7 - crosfeed from Valve 1 to Right boost pump in center tank
Valve 1 geoverification:connectedToVerified line 8 - crosfeed from Valve 1 to Left boost pump in center tank
Valve 1 geoverification:containedInVerified Center feed tank
Valve 10 geoverification:connectedToVerified line 25 - secondary refueling line from main refuel line Valve 10
Valve 10 geoverification:connectedToVerified line 26 - secondary refueling line from Valve 10 to Right transfer tank
Valve 10 geoverification:containedInVerified Right transfer tank
Valve 11 geoverification:connectedToVerified line 27 - secondary refueling line from main refuel line Valve 11
Valve 11 geoverification:connectedToVerified line 28 - secondary refueling line from Valve 11 to Right feed tank
Valve 11 geoverification:containedInVerified Right feed tank
Valve 12 geoverification:connectedToVerified line 29 - secondary refueling line from main refuel line Valve 12
Valve 12 geoverification:connectedToVerified line 30 - secondary refueling line from Valve 12 to Center feed tank
Valve 12 geoverification:containedInVerified Center feed tank
Valve 2 geoverification:connectedToVerified line 10 - feed line from Valve 2 to Valve 5
Valve 2 geoverification:connectedToVerified line 7 - crosfeed from Valve 1 to Right boost pump in center tank
Valve 2 geoverification:containedInVerified Center feed tank
Valve 3 geoverification:connectedToVerified line 8 - crosfeed from Valve 1 to Left boost pump in center tank
Valve 3 geoverification:connectedToVerified line 9 - feed line from Valve 3 to Valve 4
Valve 3 geoverification:containedInVerified Center feed tank
Valve 4 geoverification:connectedToVerified line 10 - feed line from Valve 2 to Valve 5
Valve 4 geoverification:connectedToVerified line 11 - right engine feed line after Valve 4
Valve 4 geoverification:containedInVerified Right feed tank
Valve 5 geoverification:connectedToVerified line 12 - Left engine feed line after Valve 5
Valve 5 geoverification:connectedToVerified line 9 - feed line from Valve 3 to Valve 4
Valve 5 geoverification:containedInVerified Left feed tank
Valve 6 geoverification:connectedToVerified line 15 - crossfeed parallel line
Valve 6 geoverification:connectedToVerified line 16 - crossfeed parallel line
Valve 6 geoverification:containedInVerified Center feed tank

Valve 7 geoverification:connectedToVerified line 19 - secondary refueling line from main refuel line Valve 7
Valve 7 geoverification:connectedToVerified line 20 - secondary refueling line from Valve 7 to Left transfer tank
Valve 7 geoverification:containedInVerified Left transfer tank
Valve 8 geoverification:connectedToVerified line 21 - secondary refueling line from main refuel line Valve 8
Valve 8 geoverification:connectedToVerified line 22 - secondary refueling line from Valve 8 to Left feed tank
Valve 8 geoverification:containedInVerified Left feed tank
Valve 9 geoverification:connectedToVerified line 23 - secondary refueling line from main refuel line Valve 9
Valve 9 geoverification:connectedToVerified line 24 - secondary refueling line from Valve 9 to Center feed tank
Valve 9 geoverification:containedInVerified Center feed tank

A.3 Certification and guidelines verification - Resulting statements

Table A.5: Verified certification-related statements

Boost pump 1 in left inner tank fuelsystem:isDistinctIndividualVerified Left feed tank
Boost pump 1 in right inner tank fuelsystem:isDistinctIndividualVerified Right feed tank
Boost pump 2 in left inner tank fuelsystem:isDistinctIndividualVerified Left feed tank
Boost pump 2 in right inner tank fuelsystem:isDistinctIndividualVerified Right feed tank
Center feed tank fuelsystem:hasDistinctIndividualsVerified fuelsystem:FuelPump
Center feed tank fuelsystem:hasRedundantPumpsVerified 3395e5f2-b797-42dc-b8e1-4a69f6fbdc90
Crossfeed System fuelsystem:hasDistinctIndividualsVerified fuelsystem:FuelValve
Crossfeed System fuelsystem:hasRedundantValvesVerified 1b408336-8c33-4d1b-a8ac-044ea2e85e17
Left boost pump in center tank fuelsystem:isDistinctIndividualVerified Center feed tank
Left feed tank fuelsystem:hasDistinctIndividualsVerified fuelsystem:FuelPump
Left feed tank fuelsystem:hasRedundantPumpsVerified de7ee484-dffa-4d4d-b8d6-0a4c19052348
Left transfer tank fuelsystem:hasMassVerified 691.0[kg]
line 1 from Left boost pump in center tank fuelsystem:hasMinimumDistanceToVerified Center feed tank
line 1 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 1 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 1 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 1 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 10 - feed line from Valve 2 to Valve 5 fuelsystem:hasMinimumDistanceToBottomVerified Right feed tank
line 10 - feed line from Valve 2 to Valve 5 fuelsystem:hasMinimumDistanceToRearVerified Right feed tank
line 10 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 10 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 10 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 10 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 11 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 11 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 11 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 11 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 12 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 12 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 12 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 12 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 13 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 13 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 13 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 13 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 14 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 14 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 14 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 14 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 15 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 15 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 15 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 15 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 16 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 16 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 16 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 16 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 17 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 17 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine

line 18 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 18 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 19 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 19 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 19 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 19 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 2 from Right boost pump in center tank fuelsystem:hasMinimumDistanceToVerified Center feed tank
line 2 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 2 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 2 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 2 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 20 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 20 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 20 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 20 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 21 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 21 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 21 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 21 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 22 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 22 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 22 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 22 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 23 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 23 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 23 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 24 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 24 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 24 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 25 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 25 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 25 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 25 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 26 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 26 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 26 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 26 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 27 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 27 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 27 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 27 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 28 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 28 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 28 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 28 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 29 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 29 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 29 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 29 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 3 from Boost pump 1 in right inner tank fuelsystem:hasMinimumDistanceToVerified Right feed tank
line 3 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 3 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 3 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 3 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 30 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 30 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine

line 30 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 30 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 4 from Boost pump 2 in right inner tank fuelsystem:hasMinimumDistanceToVerified Right feed tank
line 4 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 4 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 4 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 4 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 5 from Boost pump 1 in left inner tank fuelsystem:hasMinimumDistanceToVerified Left feed tank
line 5 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 5 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 5 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 5 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 6 from Boost pump 2 in left inner tank fuelsystem:hasMinimumDistanceToVerified Left feed tank
line 6 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 6 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 6 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 6 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 7 - crosfeed from Valve 1 to Right boost pump in center tank fuelsystem:hasMinimumDistanceToBottomVerified Center feed tank
line 7 - crosfeed from Valve 1 to Right boost pump in center tank fuelsystem:hasMinimumDistanceToBottomVerified Right feed tank
line 7 - crosfeed from Valve 1 to Right boost pump in center tank fuelsystem:hasMinimumDistanceToRearVerified Center feed tank
line 7 - crosfeed from Valve 1 to Right boost pump in center tank fuelsystem:hasMinimumDistanceToRearVerified Right feed tank
line 7 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 7 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 7 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 7 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 8 - crosfeed from Valve 1 to Left boost pump in center tank fuelsystem:hasMinimumDistanceToBottomVerified Center feed tank
line 8 - crosfeed from Valve 1 to Left boost pump in center tank fuelsystem:hasMinimumDistanceToBottomVerified Left feed tank
line 8 - crosfeed from Valve 1 to Left boost pump in center tank fuelsystem:hasMinimumDistanceToRearVerified Center feed tank
line 8 - crosfeed from Valve 1 to Left boost pump in center tank fuelsystem:hasMinimumDistanceToRearVerified Left feed tank
line 8 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 8 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 8 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 8 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:hasMinimumDistanceToBottomVerified Left feed tank
line 9 - feed line from Valve 3 to Valve 4 fuelsystem:hasMinimumDistanceToRearVerified Left feed tank
line 9 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 1 - Left Engine
line 9 fuelsystem:hasFanFragmentCollisionPathWithVerified Fan 2 - Right Engine
line 9 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 1 - Left Engine
line 9 fuelsystem:hasSmallFragmentCollisionPathWithVerified Rotor 2 - Right Engine
Right boost pump in center tank fuelsystem:isDistinctIndividualVerified Center feed tank
Right feed tank fuelsystem:hasDistinctIndividualsVerified fuelsystem:FuelPump
Right feed tank fuelsystem:hasRedundantPumpsVerified 38d6d9dd-c14c-4841-8548-38cebbeba229
Right transfer tank fuelsystem:hasMassVerified 691.0[kg]
spar-mounted APU pump in the center tank - left side fuelsystem:isDistinctIndividualVerified Center feed tank
Valve 1 fuelsystem:isDistinctIndividualVerified Crossfeed System

Valve 10 fuelsystem:isNotProneToIcingVerified line 25 - secondary refueling line from main refuel line Valve 10

Valve 10 fuelsystem:isNotProneToIcingVerified line 26 - secondary refueling line from Valve 10 to Right transfer tank

Valve 11 fuelsystem:isNotProneToIcingVerified line 28 - secondary refueling line from Valve 11 to Right feed tank

Valve 12 fuelsystem:isNotProneToIcingVerified line 30 - secondary refueling line from Valve 12 to Center feed tank

Valve 2 fuelsystem:isNotProneToIcingVerified line 7 - crosfeed from Valve 1 to Right boost pump in center tank

Valve 3 fuelsystem:isNotProneToIcingVerified line 8 - crosfeed from Valve 1 to Left boost pump in center tank

Valve 4 fuelsystem:isNotProneToIcingVerified line 10 - feed line from Valve 2 to Valve 5

Valve 4 fuelsystem:isNotProneToIcingVerified line 11 - right engine feed line after Valve 4

Valve 5 fuelsystem:isNotProneToIcingVerified line 12 - Left engine feed line after Valve 5

Valve 5 fuelsystem:isNotProneToIcingVerified line 9 - feed line from Valve 3 to Valve 4

Valve 6 fuelsystem:isDistinctIndividualVerified Crossfeed System

Valve 6 fuelsystem:isNotProneToIcingVerified line 15 - crossfeed parallel line

Valve 6 fuelsystem:isNotProneToIcingVerified line 16 - crossfeed parallel line

Valve 7 fuelsystem:isNotProneToIcingVerified line 19 - secondary refueling line from main refuel line Valve 7

Valve 7 fuelsystem:isNotProneToIcingVerified line 20 - secondary refueling line from Valve 7 to Left transfer tank

Valve 8 fuelsystem:isNotProneToIcingVerified line 22 - secondary refueling line from Valve 8 to Left feed tank

Valve 9 fuelsystem:isNotProneToIcingVerified line 24 - secondary refueling line from Valve 9 to Center feed tank

Table A.6: Falsified certification-related statements

Valve 8 fuelsystem:isNotProneToIcingFalsified line 21 - secondary refueling line from main refuel line Valve 8
Valve 3 fuelsystem:isNotProneToIcingFalsified line 9 - feed line from Valve 3 to Valve 4
Valve 1 fuelsystem:isNotProneToIcingFalsified line 7 - crosfeed from Valve 1 to Right boost pump in center tank
Valve 1 fuelsystem:isNotProneToIcingFalsified line 8 - crosfeed from Valve 1 to Left boost pump in center tank
Right feed tank fuelsystem:hasMassFalsified 5435.0[kg]
Center feed tank fuelsystem:hasMassFalsified 6476.0[kg]
Fuel system fuelsystem:hasMassFalsified 18728.0[kg]
Valve 12 fuelsystem:isNotProneToIcingFalsified line 29 - secondary refueling line from main refuel line Valve 12
Valve 11 fuelsystem:isNotProneToIcingFalsified line 27 - secondary refueling line from main refuel line Valve 11
Left feed tank fuelsystem:hasMassFalsified 5435.0[kg]
Valve 9 fuelsystem:isNotProneToIcingFalsified line 23 - secondary refueling line from main refuel line Valve 9
Valve 2 fuelsystem:isNotProneToIcingFalsified line 10 - feed line from Valve 2 to Valve 5
line 17 fuelsystem:hasSmallFragmentCollisionPathWithFalsified Rotor 1 - Left Engine
line 17 fuelsystem:hasSmallFragmentCollisionPathWithFalsified Rotor 2 - Right Engine
line 18 fuelsystem:hasSmallFragmentCollisionPathWithFalsified Rotor 1 - Left Engine
line 18 fuelsystem:hasSmallFragmentCollisionPathWithFalsified Rotor 2 - Right Engine
line 23 fuelsystem:hasSmallFragmentCollisionPathWithFalsified Rotor 2 - Right Engine
line 24 fuelsystem:hasSmallFragmentCollisionPathWithFalsified Rotor 1 - Left Engine