Automation Opportunities in the Conceptual Design of Satellite Propulsion Systems

- SECESA 2016 -

5-7 October 2016

Universidad Politécnica de Madrid (UPM)
Spain

Jens Schmidt[(1)], Stephan Rudolph[(2)]

[(1)] IILS Ingenieurgesellschaft für Intelligente Lösungen und Systeme mbH
Leinfelderstrasse 60, D-70771 Echterdingen
schmidt@iils.de

[(2)]ISD Institut für Statik und Dynamik der Luft- und Raumfahrtkonstruktionen
Universität Stuttgart, Pfaffenwaldring 27, D-70569 Stuttgart
rudolph@isd.uni-stuttgart.de

ABSTRACT During the conceptual design phase, a multitude of different system topologies with different parametrical settings need to be imagined, synthesized, computed, simulated, analyzed and evaluated in order to come to an objective design decision. This usually results in a larger number of design iterations involving a multitude of different physical system models, each suited to cover a different design aspect. Design automation using an abstract, graph-based design language representation in UML (Unified Modeling Language) offers in this situation a huge potential for intelligent design automation. The advantages of such an automation support are three-fold: firstly, the *n* individual parameter sets in the *n* different physical models are consistent to each other, secondly, after either a topological or parametrical change all the models are automatically updated to restore and guarantee the overall model consistency, and thirdly, the design process it automated thus reducing the time needed for design cycles down to the execution time of the program. Using the example of the conceptual design of satellite propulsion systems, the automation of a requirements-driven engineering design process is shown via the automated generation of different satellite propulsion systems for different mission requirements.

INTRODUCTION

Finding a suitable system design for a given task with a multitude of requirements is the day to day of an engineer in the conceptual design phase. While this area of engineering with the consideration of all the (or at least many) possibilities is quite exciting, it is also somewhat repetitive. If for example a system topology has been fixed but not all parts have been selected yet, the very same calculations are done just with different numbers. Additionally, the conceptual design phase is characterized by a multitude of design changes resulting in many updates of the various design models, e.g. of the geometry, of the thermal or electric model etc. to stay consistent with each other and the current design. Design languages offer a method to automate the recurring tasks and also to generate consistent models fully automatically.
A design language consists of three parts: a *vocabulary* which describes the entities of a domain, *rules* which are created from the vocabulary and encode design knowledge in the domain and a *production system* which encodes the sequence of the rules and creates a product of the domain. These three parts are created manually by one or several human(s). The design language can then be processed by a so-called *design compiler* which creates the *design graph*, a holistic digital model of the product containing all parts, interconnections and parameters. From this abstract design graph different models can be created automatically. Fig. 1 shows the information flow in a design language's compilation process [1]. Graph-based design languages are encoded in UML [2].
This work shows how design knowledge of a product is encoded in a graph-based design language using the example of the conceptual design of propulsion systems for satellites. Then this graph-based design language is used to generate different propulsion system types and compares them with each other.

MODELING

The overall design of a propulsion system can be subdivided in multiple distinct steps. Some of these steps are modeled in the graph-based design language discussed hereafter. An overview of the first two steps (topology and parameter selection) was already given in a previous work [3].

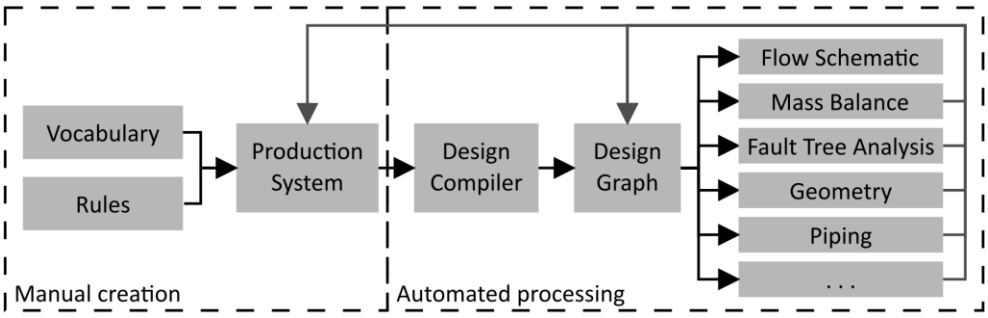

Fig. 1 Information flow and automatic model creation in graph-based design languages

After the requirements have been determined, the system topology can be defined, i.e. the system type and all parts and their interconnections. Then the systems parametrics can be calculated, e.g. how much propellant and pressurant is needed or how big the tanks need to be. This step also further defines the parts. In the topology step, the knowledge of existence was sufficient, e.g. there are thrusters connected via a pressure regulator and pipes to a gas tank. In the parametrics step, the actual parts are selected and their values like mass, volume, reliability, etc. are set in the graph-based design language. With this step concluded, the design language can already be used to generate various different propulsion systems from different sets of requirements. For each system, a flow schematic and a mass balance is generated fully automatically. This allows for a comparison between the different system types and layouts.
Also, with a known topology and parameter set a plug-in for reliability can be used in the next step of the design language to fully automatically generate a fault-tree analysis (FTA) of the system in question. In a further developed design language for propulsion systems the results of this FTA analysis could be fed back into the design language to trigger a topology change or to promote parts with different reliability values for selection. How the automatic FTA works has been already shown in a previous paper [4] and will therefore not be discussed here further.
The next step in the design phase is to give the parts of the system their actual shape and to place them on panels for integration (PCA, PIA). Unfortunately, no company we inquired was able or willing to provide us the 3D geometry data of their components, so we had to resort to pictures of the components and create the 3D models of the geometry ourselves. To follow the "real" process somewhat, this modelling was partly done with a CAD program resulting in STEP files and partly done in the design compiler itself using its geometry generation capabilities. The placement of the parts on the panel was done manually. However, with more in depth knowledge of the requirements respective to the panel, an algorithm for fully automatic placement could be developed an included into the design language.
With the 3D geometry in place, a plug-in for automatic piping and routing is used to generate the pipes on the panels fully automatically, as the last execution step of this design language. The information about what is connected to what can be directly inferred from the system topology generated in the very first step. Right now, only the pipes on the panels are created, since the generation of the complete piping would require more information about the satellite the system is going to drive. Therefore the mass of the pipes is not yet taken into account in the mass balance.

Topology

Three variants of commonly used propulsion system can be modeled with this graph-based design language, i.e. the (regulated) *Coldgas*-, the (blowdown) *Monergol*- and the (regulated) *Diergol*system. Each system has one to many *Areas*, they describe areas of the system which hold one working fluid, e.g. the coldgas system has one area for its pressurant gas, a diergolsystem will have three areas: one for the fuel, the oxidizer and the pressurant gas. Each *Area* can have several *Tasks*. Usually that are *Store* to hold the working fluid, *Manage* to distribute, regulate and monitor and *Thrust* to use the working fluid in thrusters. A *Diergol*system will have the tasks *Store* and *Manage* in its area for the pressurant gas. This *Manage* then connects to *Store* of the two other areas for fuel and oxidizer.
Each *Task* has a 'one to many' relationship to *FunctionalElements*. They allow the description of a sequence of distinct requirements. Depending on the *Task* several functional requirements may arise. For *Store* this will be the requirement to include some facilities to store the working fluid the *StorageArea*. Or, to use a more complex example: *Manage* in the *Area* for pressurant gas in a *Diergol*system may have the functional requirements to isolate (*Isolation*) the *StorageArea*, e.g. for ground operations, then to limit the pressure (*LimitPressure*) for further use and finally verify that the pressure is indeed limited (*MeassurePressure*). Further functional requirements may include the testability (*FillDrain* before and after) during the testing phase or a redundancy by which this *FunctionalElement* should be fulfilled. The *FunctionalElements* define which type, how many and in what sequence (parallel or serial) the *FlowElements* shall be included in the system. The *FlowElements* are the actual parts of the propulsion system. To arrive here, some abstractions were necessary, Fig. 2 shows the discussed abstraction steps.

Actually this abstraction is quite straight forward: there is a *Coldgas*system with one *Area*, e.g. for Nitrogen, *Manage* has some *FunctionalElements*, one of them is *Isolation* after the *StorageArea* with a redundancy of one. This results in the inclusion of two *PyrovalvesNC* into the system in the shown way in Fig. 2

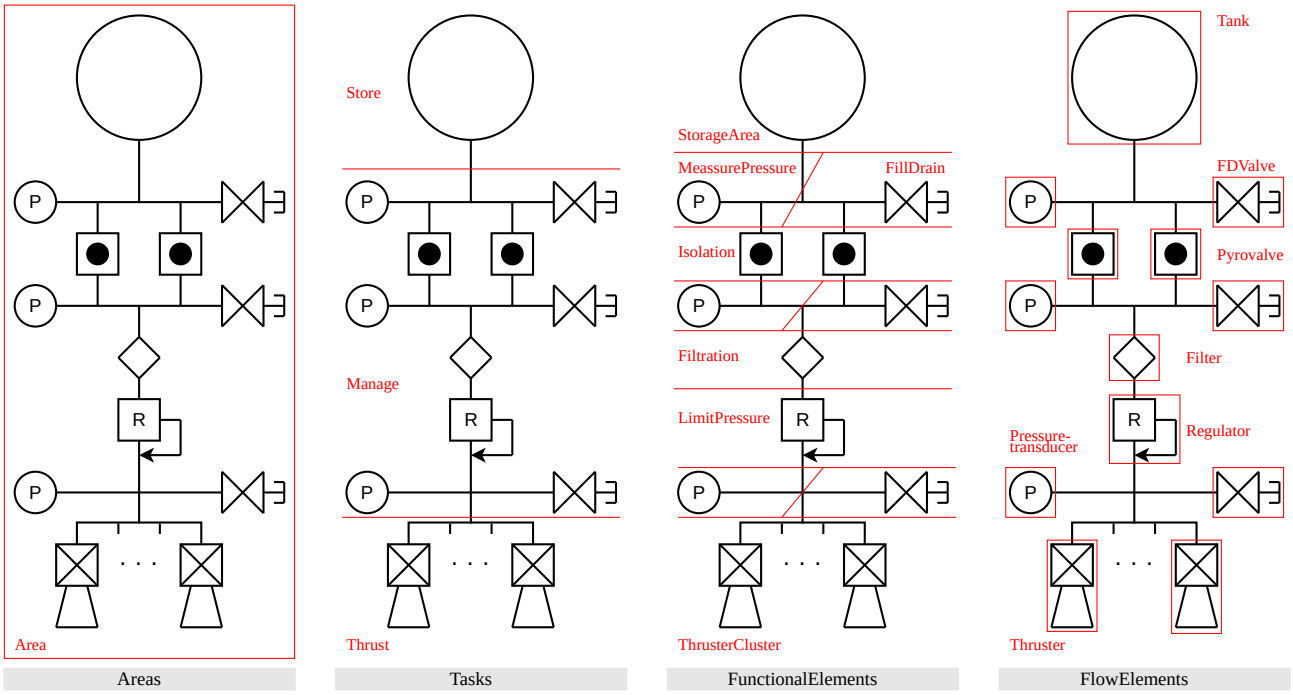

Fig. 2 Abstractions layers for an exemplary cold gas propulsion system, from left to right: *Areas* include a working fluid, e.g. a pressurant gas. *Tasks* define the responsibilities of an *Area*, i.e. *Store*, *Manage* and *Thrust*. *FunctionalElements* are used to describe a network of requirements, e.g. *Filtration* before *LimitPressure* or *Isolation* after a *StorageArea* with a defined redundancy. This network of requirements then leads to *FlowElements* the actual parts in the propulsion system.

Parametrics

With parts in the system and their connections set up, it is time to model parameters, constraints and design equations of the propulsion system. Variables of these equations are stored as attributes in the classes. The equations themselves are also stored in the classes (not shown).

For the conceptual design phase some assumptions were made: gasses are ideal, liquids are incompressible, maneuvers are isotherm, the temperature stays constant, pressurant does not dissolve in fuel, fuel does not create a gaseous phase in the tank and the specific impulse stays constant over the duration of the mission. For simplicity, for almost all parts there is only one kind, i.e. regardless of system choice the same part (e.g. filter) is used. An exception to this are the tanks, which are selected from a table and the engines which are given. If more data regarding all the various possible parts in a propulsion system would be provided, the part selection via lookup table could be extended to include all parts.

The goal is to lay out a propulsion system for a satellite from given ΔV-requirements and engines starting with the Ziolkowsky equation $\Delta V = I_s g_0 \ln\left(m_{wet} / m_{dry}\right)$. *„This equation is expanded with several other equations: The wet mass is the sum of the dry mass and the mass of the propellant. The dry mass is the sum of the masses of the satellite (m_{sat}) and the propulsion system (m_{sys}). The mass of the propulsion system is the sum of the dead mass, the pressurant mass and all masses of the system components. To select a suitable tank from the table, the required storage volume needs to be known. It is calculated with the propellant mass, the dead mass and the propellant density. The tank is part of the system, so its selection has an impact on the mass of the propulsion system. Thus the calculation of the propellant mass and the tank selection form a combined iterative process, which is solved in an iteration loop”*, see [3].

The above process yields the mass of the propulsion system for one ΔV-requirement, i.e. the sum of maneuvers executed with the same engine type. Depending on the mission one type of engines may not suffice, a common case is the usage of one type of engines as main engine, e.g. for orbit insertion, and another type of engines for attitude control, e.g. detumbling and station keeping. In this paper it is assumed that maneuvers with different engines do not take place

at the same time. This allows to model the sequence of maneuvers as tandem staging with a constant dry mass. With this considerations the Ziolkowsky equation can be expanded for each maneuver *i* as (1).

$$\Delta V_i = I_{si}\, g_0 \ln\left(\frac{m_{dry} + \sum_{j=i}^{n} m_{tj}}{m_{dry} + \sum_{j=i+1}^{n} m_{tj}}\right) \tag{1}$$

Eqn. (1) can be rewritten with $\chi_i = e^{\Delta V_i / \left(I_{si} g_0\right)}$ and some re indexing into (2).

$$\frac{-m_{ti}}{1-\chi_i} - \sum_{j=i+1}^{n} m_{tj} = m_{dry} \tag{2}$$

All the equations of each maneuver *i* can be rewritten into an implicit matrix equation of the form (3) which then can be solved with an iteration method for all m_{ti}, e.g. the Newton-Raphson method.

$$A \cdot M_T = M_{DRY} \tag{3}$$

A is a upper triangular matrix with entries after (2), M_T is a vector with tuples of the propellant masses for each maneuver m_{ti} and M_{DRY} is a vector containing the dry mass in each tuple. Please mind that the mass m_{dry} contains also the tank mass which changes depending on the amount of propellant needed, i.e. this forms a second iteration loop. For an example with two maneuvers with different engines (3) expands to (4).

$$\begin{bmatrix} \frac{-1}{1-\chi_1} & -1 \\ 0 & \frac{-1}{1-\chi_2} \end{bmatrix} \cdot \begin{bmatrix} m_{t1} \\ m_{t2} \end{bmatrix} = \begin{bmatrix} m_{dry}\left(m_{t1} + m_{t2}\right) \\ m_{dry}\left(m_{t1} + m_{t2}\right) \end{bmatrix} \tag{4}$$

A more detailed description of all the other equations used to determine the parametrics of a propulsion system in the design language unfortunately goes beyond the scope of this paper. More general information on this sort of calculations can be found in the book [5] and the paper [6].

Abstract Geometry

With the parametrics of the system defined, the next step is to materialize the propulsion system and give the parts their shape. For this step, another graph-based design language in UML which facilitates several means to express the product structure and its geometry in an abstract way is interfaced. The product structure can express parts and assemblies. The geometry for these parts and assemblies can be either A) direct geometrical entities, e.g. *Box*, *Cylinder*, *Spline*, etc. in conjunction with the Boolean operations, *Union*, *Difference* and *Intersection*, or B) previously created existing geometry outside of the design compiler, e.g. STEP-Files.
This abstract specification enables the usage of the same geometry description across domain boundaries, i.e. the same input can be used for a thermal simulation as well as for packaging, routing and piping, etc. Additionally, if approach A) is used, the geometry is not bound to a specific CAD program. More information regarding the abstract geometry and a sample usage can be found in [7] and [8]. As stated earlier, in the design language for propulsion systems both approaches (A and B) are used to build the geometry of the parts.

Piping/Routing

After giving the parts their shape, the interconnections (pipes) need to be modeled as well. This is achieved again by interfacing another design language, in this case a design language for automatic routing (work done by Marc Eheim and Roland Weil at IILS, see [9]). This design language builds on top of the abstract geometry and allows the definition of connections between entities. Usually these connections represent pipes or cables and the entities are parts of a product structure. The connections are created fully automatically via a modified A*-search algorithm inside arbitrary complex 3D geometry [9].

Ontology Mapping and Merging

Ontology mapping and merging is the technique used to connect different design languages and use them together. Ontology here means the vocabulary of a graph-based design language. Mapping and merging means to incorporate or reference one entity from one ontology in another. The beauty of this technique lays in the minimal effort needed to bring different concepts, i.e. different concepts in different design languages, flawlessly together.
Usually one ontology covers one domain, e.g. the geometry domain, the routing or propulsion system domain. These domains will overlap in some areas, e.g. the actual parts of the propulsion system will have some kind of shape. A shape is also needed for the routing process to define source and target of the connections and, after the algorithm ran, also for the connections themselves. Exactly those overlapping concepts in the domains are the candidates for the mapping and merging. They are expressed in the consuming ontology, e.g. the routing ontology references some parts of the abstract geometry ontology. Actually there is only one mapping needed: from *RtComponent* (routing) to *Component* (geometry). *Component* is the class which is used to build the product structure in the abstract geometry. It can reference an arbitrary shape created from its own entities or an existing one, see [7] for details.
RtComponent is the class which is used to describe entities within the routing plug-in which have geometry, it inherits from *Component*. This means all the geometry handling is delegated to the abstract geometry and the routing is only concerned with its own algorithms, i.e. how the actual geometry is built is of no concern for the routing. The routing only needs to know that there are entities which have a geometry, and if queried, will return their geometry representation by means of the abstract geometry.
A similar mapping is used in the ontology for the propulsion system. *FlowElements* are the actual parts and therefore need some kind of shape. Thankfully geometric entities are already handled in the abstract geometry, so a simple import and reference to *Component* suffices.
This means there are two kinds of mapping and merging. One is explicit, e.g. the routing is referring to the abstract geometry, or the propulsion system is referring to the abstract geometry. The other one is implicit, e.g. there is no explicit mapping from routing to propulsion system or vice versa. The “glue” in this case is the geometry which both ontologies refer to. Or, to put it another way, the routing does not know, and does not need to know, that it routes a propulsion system. The propulsion system does not know that some of its parts (the pipes) where generated with an algorithm.
However if in later design stages requirements for the pipes arise, e.g. minimal bending radius, distances between pipes, the link from propulsion system to routing can easily be included to facilitate more control over the routing process. As of this moment the placement of the parts (*FlowElements*) on the panel is done manually, though a future design language for packaging can be included with the same ontology mapping and merging principles discussed here.

Creating the Design Language

Putting all this elements together into the vocabulary of the design language for propulsion systems yields a UML Class Diagram (partly) shown in Fig. 3. The vocabulary is used in rules to encode all requirements and design knowledge, e.g. *Filtration* after any *StorageArea* or the initial m_{sat} is 800kg. Rules modify the design graph. They are either graphical rules describing graph transformations in a two quadrant scheme or procedural rules (e.g. JAVA code) if no easy graphical formulation is known or exists. Rules can be grouped into sub programs. The rules and sub programs are then aggregated in the production system encoded in an UML activity diagram. The production system of the design language for propulsion systems is shown in Fig. 4 in addition to the corresponding sample output of each subprogam.
SubTopology generates the system topology from a set of starting conditions and a flow schematic for visualization. *SubParametrics* sets up the iteration loops and selects appropriate tanks. Here the design equations discussed earlier are instantiated together with their classes thus forming an equation system in the background. This system is reordered by a so called solution path generator (SPG) resulting in a solution sequence of the system. With this sequence and the known boundary conditions a computer algebra system is triggered to obtain the results. The results are stored in the design graph for further usage, see [3]. *SubReliability* generates a FTA. *SubGeometry* materializes the parts, sets up a product structure and generates the equipment panels utilizing the abstract geometry. *SubPiping* then connects all parts via pipes. Thus enabling, with the exception of the part placement on the panels, a fully automatic design of a propulsion system in the conceptual design phase.
This graph-based design language can then be used with varying input parameters to subsequently gain knowledge of the design space, i.e. find the best system for a given set of requirements.

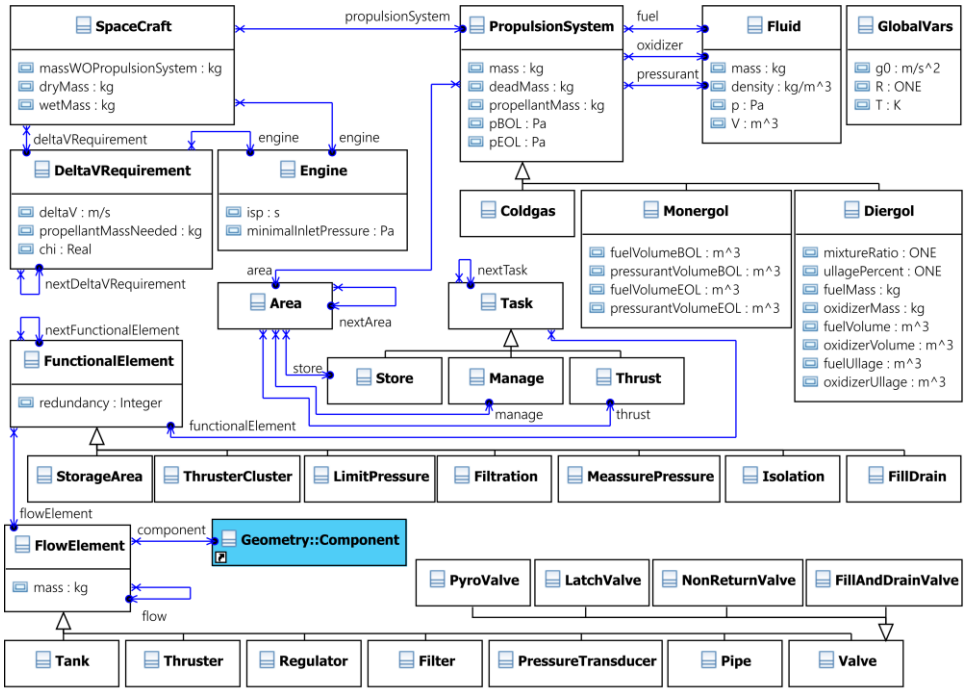

Fig. 3 Simplified class diagram. The boxes are classes (reads as *"there is"*), a closed arrowhead denotes an inheritance (reads as *"is a"*), an open arrowhead denotes an association (reads as *"has a"*), and subclasses inherit all attributes of their parent class. Turquoise color denotes ontology mapping and merging.

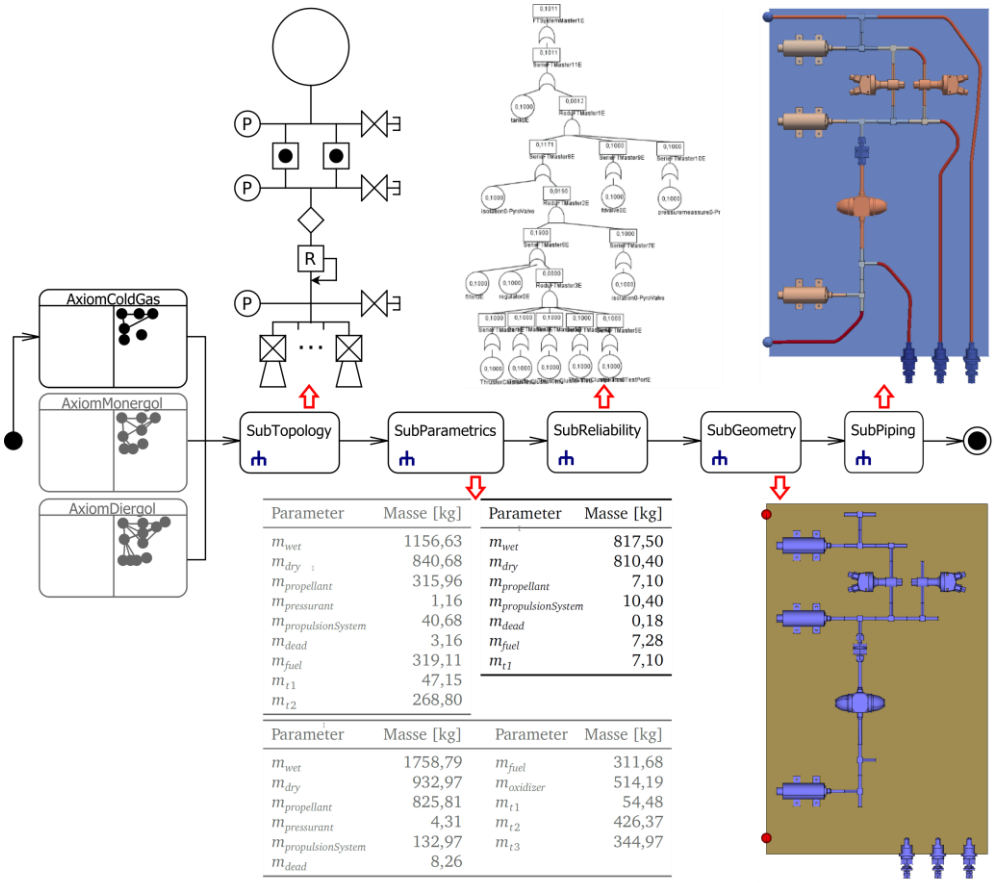

Fig. 4 Production system of the design language for satellite propulsion systems (horizontally) with overlay of exemplary results (vertically).

SAMPLE APPLICATION

With the use of graph-based design languages, a design space can be systematically expanded, analyzed and searched for optimal solution candidates. Parts of the design language for propulsion system (topology and parametrics) are used to refine our first analysis from previous works, see [3]. Starting with the same satellite mass of 800 kg, different propulsion systems are generated for each of the three system types with an increasing ΔV-requirement for one maneuver resulting in the design space shown in Fig. 5. With a regular personal computer the generation of 453 propulsion systems took about 5 hours (Quadcore processor at 2,8GHz).

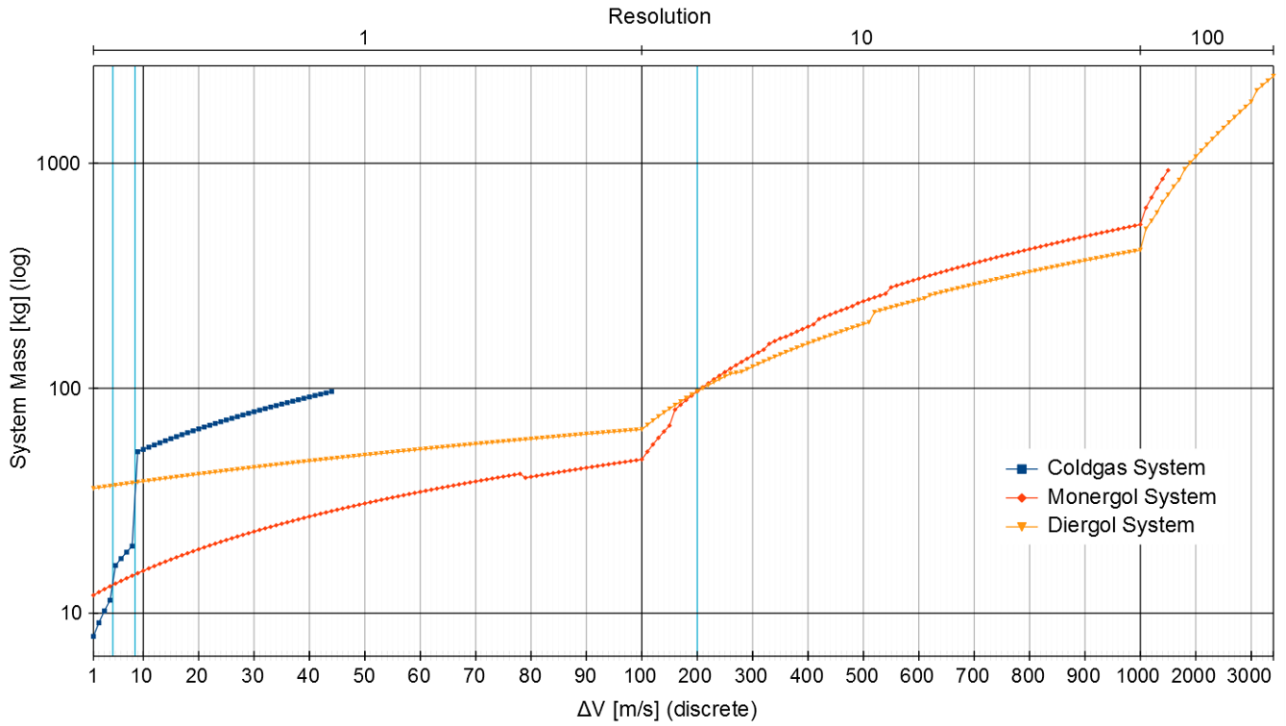

Fig. 5 Comparison of system performances generated with a design language, data points are connected for better visualization of trends, sharp bends indicate tank changes. The three turquoise lines show candidates for system topology changes (so-called “topology change points”, see [3]).

As expected, the candidate for "best" (lightest) system for a given ΔV–requirement changes from low performance but light systems to high performance but heavy systems - from the coldgas- over the monergol- to the diergolsystem. Steep bends in Fig. 5 are caused by the selection of a bigger tank from the lookup table. The storage volume needed for fuel raises proportionally with the ΔV required. At some point the selected tank must be switched to the next bigger one - which is usually much larger than needed, thus resulting in a sudden jump of the mass of the propulsion system. This newly provided volume is “slowly consumed with increasing ΔV resulting in a much gentler slope [3].” If no suitable tank can be found in the lookup table the system generation is aborted resulting in the truncated curves. However with the addition of more tanks in the table this limitation can be easily remedied. Usually, what defines an optimal system is not determined by one parameter alone. Therefore additional investigations and considerations would be necessary to find a decent solution for a given set of requirements, e.g. if a low contamination environment is needed, the coldgas system would be preferred even if this resulted in a higher system mass. The shown sample application of a graph-based design language is a viable tool to identify optimal technology candidates for a set of system requirements [3].
This process can be expanded to do analysis for the whole propulsion system including the piping if the installation conditions of the system are known and an algorithm for automatic placement of the parts on the panels is developed.
To span a design space is a fruitful application for design languages since the overall process is fully automatic after the initial setup of the design language. Another example can be found in [10] where the design space exploration with design languages is shown for some subsystems of a satellite.

CONCLUSION & OUTLOOK

Parts of the domain of propulsion systems for satellites in the conceptual design phase could be encoded in a design language. The language follows a (natural) design process were in sequence topology, parametrics, reliability, geometry and piping are generated. This enables the automated design of three propulsion system types: the coldgas-, monergol- and diergolsystem. Design results are the system topology visualized in a flow schematic, the mass balance including propellant masses for maneuvers with different engines, a fault tree analysis, and a 3D-assembly of the panels including piping. Since all this different models are generated from a central model, the so-called design graph, model consistency is automatically ensured. With the exception of the placements of the parts on the panels (which is still ongoing work), this design process is fully automatic. The resulting design language then was used to analyse the design space and the performance of each of the tree system types with increasing ΔV-requirements.
Future versions of the language could include design rules to automate the placement of the parts and even to integrate the system into a sample satellite. This then would allow a fully automatic design process of a propulsion system, thus enabling even more in depth analysis of the design space (as shown in Fig. 5), consequentially leading to improved designs and understanding of propulsion systems.
With the addition of further classes and design equations electrical propulsion systems, e.g. arcjets, ion engines, etc. could be modeled. The part selection via a lookup table could be extended to include all parts of a propulsion system. As an alternative to that, individual parts could also be designed with their own design language.
The parametrics step could also be improved to include real gas effects. With a finer-grained resolution of the different mission phases (time of maneuvers, cruise phases) additional effects could be factored in, e.g. heat transfer with the satellite, mixing of gaseous phases in the tank, solution of pressurant gas in fuel, etc. the conceivable applications are quite wide and will give much space for further engineering creativity.
In short, graph-based design languages offer a way to capture design knowledge in a re-executable format, thus enabling the fully automatic generation of consistent domain models and the analysis of various product design variants.

ACKNOWLEDGEMENTS

SPONSORED BY THE
Federal Ministry of Education and Research

Parts of the research (i.e. abstract geometry, ontology mapping and merging) leading to these results were performed within the European ITEA2 project IDEaliSM (#13040) as part of the EUREKA cluster program. The authors would like to express their gratitude to the consortium members for their support and contributions in the European research project IDEaliSM (see https://itea3.org/project/idealism.html for details).
Parts of this project (same as above) are sponsored by the Federal Ministry of Education and Research in Germany.

REFERENCES

[1] Rudolph, Stephan: *Übertragung von Ähnlichkeitsbegriffen [Mapping of Similarity Concepts]*, Habilitationsschrift, Univ. of Stuttgart, 2002.
[2] Reichwein, Axel: *Application-Specific UML Profiles for Multidisciplinary Product Data Integration*, PhD thesis, Univ. of Stuttgart, 2011.
[3] Schmidt Jens; Rudolph Stephan: *Gaining System Design Knowledge by Systematic Design Space Exploration with Graph Based Design Languages*, ICCMSE, 2014.
[4] Riestenpatt genannt Richter, Marius; Schmidt, Jens; Rudolph, Stephan: *Automated fault tree analysis for satellite propulsion systems*. SECESA 2014, Stuttgart, Germany.
[5] Wertz, James; Everett, David; Puschel, Jeffery: *Space Mission Engineering: The New SMAD*, Space Technology Library, 2011.
[6] Thunnissen, Daniel; Engelbrecht, Carl; Weiss, Jeffrey*: Assessing Model Uncertainity in the Conceptual Design of a Monopropellant Propulsion System*, in 39th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit, 2003.
[7] Schmidt, Jens; Rudolph, Stephan: *Graph-Based Design Languages: A Lingua Franca for Product Design Including Abstract Geometry*, IEEE Computer Graphics and Applications, July/August 2016, in press.
[8] Gross, Johannes; Rudolph, Stephan: *Geometry and Simulation Modeling in Design Languages*, J. Aerospace Science and Technology, vol. 54, July 2016, pp. 183–191.
[9] IILS Routing, http://www.iils.de/#routing, last visited 07/07/2016.
[10] Gross, Johannes; Rudolph, Stephan: *Modeling Graph-Based Satellite Design Languages*, J. Aerospace Science and Technology, vol. 49, Feb. 2016, pp. 63–72.