- The methodology shall be independent from any specific application domain.
- The methodology shall enable a product-line oriented product development, i.e., the metamodel must allow modeling of different variants of a product and ensure a consistent configuration and parametrization.
- The methodology shall enable inclusion of already existing domain models, i.e., models in a domain-specific modeling language.
- The methodology shall enable automatic verification of models, i.e., it shall be possible to check if the built models adhere to the modeling paradigm and to user-defined constraints.
- The methodology shall enable consistent modeling not only of the product itself but also of the context, such as the industrial system used to build the product and allow the creation of relationships between the modeled artifacts.

The requirements for the application framework supporting this new modeling paradigm are as follows:

- The application framework shall be deployable in the current corporate IT infrastructure
- The application framework shall allow a heterogeneous technology stack to deliver the best solution for a designated purpose.
- The application framework shall be scalable with increasing number of models and users.
- The application framework shall be scalable in terms of model calculation performance.
- The application framework shall support continuous deployment strategies and agile frameworks to enable fast delivery and high flexibility.
- The application framework shall be efficient with regards to computing resources and reduce the company's ecological footprint.

A. OOC Process Description

The system lifecycle process as applied by most modern transportation system manufacturers including Airbus includes Design, Development, Production, Operation, Support, and Disposal (Figure 1).

Figure 1. System lifecycle according to [9]

As described in [9], according to the systems engineering approach, the system design process can be further divided into four major phases: conceptual design phase, preliminary design phase, detailed design phase, and test and evaluation phase (Figure 2).

The starting point for the current design process are requirements. Based on these requirements, initial design concepts are elaborated, assessed according to their feasibility, and evaluated against key performance indicators such as operating cost, weight, and range. A few concepts are then selected and refined in a preliminary design phase, and after a more profound analysis one of the design alternatives is chosen

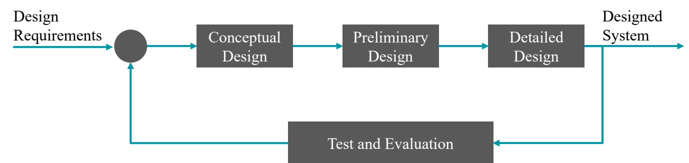

Figure 2. Major design Activities according to [9]

and further refined during detailed design analysis. To support the assessment of design concepts, various models describing different aspects of the aircraft system are developed. However, developing models to describe design alternatives is usually expensive and time consuming. In practice, when a new aircraft program is launched time pressure tends to lead to a situation where only very few alternative design concepts can be defined and assessed.

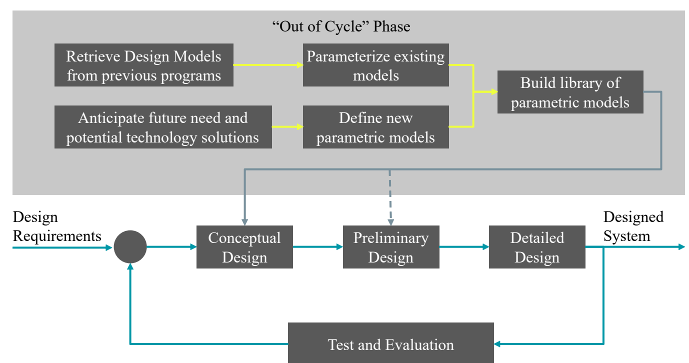

Figure 3. To-Be process with OOC phase

To address this challenge, it is suggested to introduce a new OOC phase as depicted in Figure 3. This phase is independent from aircraft programs and theoretically never ends. The aim is to produce reference architectures. Reference architectures will be decomposed into SCMs which simultaneously embed the knowledge of product design, its manufacturing system, operability, maintainability, cost and lead time. A SCM describes the technical solution for an architecture item in a reference architecture decomposition and its interfaces to other parts of the architecture, i.e., to other SCMs. Within the OOC phase, a library of SCMs of aircraft systems and components shall be defined that will grow over time. When a new aircraft program is launched, these models can be used to set-up different design concept alternatives. This shall save time and allow definition and analysis of a greater number of design alternatives, including more radical design concepts.

As shown in Figure 4, the OOC process can have different phases, during which the SCMs evolve and are being refined. The general idea is that a component matures in the OOC process until it is potentially ready to be used in an actual aircraft program. Once this stage is reached, the SCM is uploaded to a central library. At any time during the development of a new aircraft, the program can decide to pull the generic and parametrized component out of the library and specialize it to its needs. This process increases the reuse of the SCMs across multiple different programs resulting in an overall cost reduction and a decrease in the time to market for new products. This allows Airbus to react more quickly to the

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

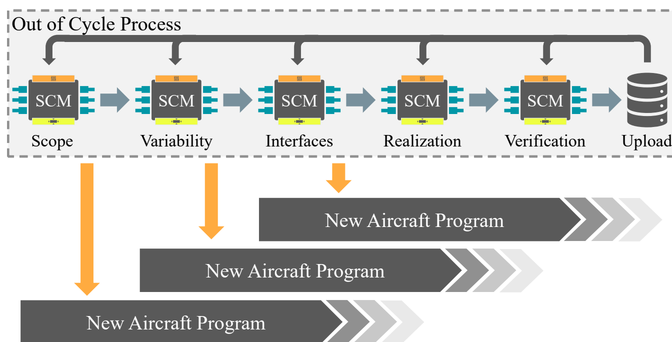

Figure 4. Evolution of a SCM over time

ever changing demands of the aerospace market.

Current design artifacts, however, do not have the features required to deal with not-yet specified product configurations and to support product evolution. They are not able to anticipate all features at design time.

Since the SCMs are being defined outside of any particular aircraft program, when requirements are not yet fixed, they have to be parametric in order to anticipate scalability and variability features and enriched with their associated limits. In case that the models are originating from previous programs, technologies have to be applied to parametrize them. Alternatively, the models may also be defined from scratch in a parametric way based on anticipation of future needs and technology trends.

SCMs provide an opportunity to capitalize interconnected multi-functional knowledge present in the organization, and also the capability to quickly generate new product designs by efficiently generating parametrized versions of components while maintaining its consistency in the overall architecture and considering its impact on integration and manufacturing processes. Their use will also naturally encourage reuse practices, which promise to make the development cycle faster and more cost-efficient.

B. Relation to other modeling languages

The most popular general purpose modeling language for systems engineering is SysML [8], which is itself an adaption of Unified Modeling Language (UML) [10] for systems engineering, yet it has still not become widely accepted [11], [12]. Karban et al. state challenges in using SysML, which have been figured out in the Active Phasing Experiment (APE) project of the SE2 challenge team of the Gesellschaft für Systems Engineering, German INCOSE chapter (GfSE) [13]. They propose several tasks for the advancement of SysML, which underlines that the language is still under development and will be further advanced in the future. Common points of criticism are: that SysML is too complex, which in turn causes complexity of SysML modeling tools; that it lacks a precise semantic; and that it does not come with a ready to use methodology, which is rooted in the fact that SysML was designed as a general purpose modeling language that should not impose a certain modeling approach. Currently, a completely reworked version 2 of SysML is being developed with the goal to increase adoption and effectiveness of MBSE by enhancing precision and expressiveness of the language, consistency and integration among language concepts and usability by model developers and consumers.

On the other hand, there are Domain-specific language (DSL), languages that are tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose modeling languages in their domain of application but generally require both domain knowledge and language development expertise to develop [14].

It seems to be an interesting question whether it is better to develop a new DSL from scratch by adding modeling elements iteratively, or start with a general purpose modeling language and restricting it until it fits the specific use.

[15] is talking about a ”yo–yo effect here: in the 1990s, many methods and modeling languages were popularized. [15] is talking about a ”yo–yo effect here: in the 1990s, many methods and modeling languages were popularized. Then, for a while, unification based on UML was very helpful. T hen, DSLs that were developed from scratch began to emerge. The next trend may be a repository of UML/SysML-based DSLs that actually unify DSLs and UML/SysML thinking.”

Our approach can be considered such a unified thinking. As already explained, we define our own DSL but it is closely aligned with SysML and we try to diverge only when we see a possibility for improving beyond the standard.

III. Architecture Paradigms

This Section provides background information regarding the two main architecture paradigms that are used today: monolithic software and MSA. Service-oriented architectures (SoA) and serverless architecture [16] are not described in detail as SoA, especially from a deployment perspective, still resembles monolith software [17] and serverless can be seen as taking MSA one step further [18].

A. Monolithic software

[19] defines a monolith as ”a software application whose modules cannot be executed independently”. This architecture is a traditional solution for building applications. A number of problems associated with monolithic applications can be identified:

- Due to their inherent complexity, they are hard to maintain and evolve. Inner dependencies make it hard to update parts of the application without disrupting other parts.
- The components are not independently executable and the application can only be deployed, started and stopped as a whole [20].
- They enforce a technology lock-in, as the same language and framework has to be used for the whole application.
- They prevent efficient scaling as popular and non-popular services of the application can only be scaled together [21].

Nevertheless, monolithic software is still widely used and, except for green-field new developments, there is hardly a way around it. [22] notes that a monolithic architecture is ”often a more practical and faster way to start”. Furthermore, if software from external parties is involved in a tool chain, it is not possible to change its architecture style.

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

B. Microservices

There is no single definition of what a MSA actually is. A commonly used definition by Lewis and Fowler says it is ”an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an Hypertext Transfer Protocol (HTTP) resource Application Programming Interface (API)” [23]. Microservices typically consist of stateless, small, loosely coupled and isolated processes in a ”share-as-little-as-possible architecture pattern” [24] where data is ”decentralised and distributed between the constituent microservices” [25].

The term ”microservices” was first introduced in 2011 [23] and publications on architecting microservices are rapidly increasing since 2015 [26]. In 2016, a systematic mapping study found that ”no larger-scale empirical evaluations exist” [27] and concluded that MSA is still an immature concept.

The following main benefits can be attributed to MSA:

- Relatively small components are easier for a developer to understand and enable designing, developing, testing and releasing with great agility.
- Infrastructure automation allows to reduce the manual effort involved in building, deploying and operating microservices, thus enabling continuous delivery [26].
- It is less likely for an application to have a single point of failure because functionality is dispersed across multiple services [17].
- MSA does not require a long-term commitment to any single technology or stack.

[4] notes the obvious drawback of the current popularity of microservices that ”they're more likely to be used in situations, in which the costs far outweigh the benefits” even when monolithic architecture would be more appropriate.

In a study regarding the challenges of adopting microservices, [3] lists the distributed nature of MSA, which leads to debugging problems, the unavailability of skilled developers with intimate knowledge of MSA and finding an appropriate separation into services.

IV. Deployment Infrastructure

Corporate IT environments imply very strict regularities when it comes to hard- and software architectures and deployments. Bringing in innovation in such an environment requires following a heterogeneous approach.

While it is more challenging to adapt hardware in a corporate context to cope with the latest innovations, service and software developments, e.g., Advanced RISC Machine (ARM) Central Processing Unit (CPU) platform based servers, Graphics Processing Unit (GPU) assisted computing or wide-usage of Field Programmable Gate Arrays (FPGAs), the application platform layer adaption is typically less demanding because almost any state-of-the-art deployment form, like bare-metal, Infrastructure as a Service (IaaS), Platform as a Service (PaaS) or PaaS can be rolled out on standard server hardware.

The rationale for choosing a specific deployment form is based on various constraints imposed by corporate policies and long-term strategy decisions:

- Is the envisaged deployment form available in the corporate infrastructure?
- Has the deployment form limitations due to corporate policies, e.g., restricted internet access, restricted repository access?
- Are there any license limitations?
- Are there geolocation limitations for certain services, e.g., in a multinational company with multinational regulations according to law?
- Is the service available on premise or only on public cloud?
- Does a deployment form for a particular service fit in the long-term corporate IT strategy, e.g., make or buy decisions?

For the SCM modeling prototype, it was necessary to make use of a heterogeneous software and hardware infrastructure provided by the corporate IT. Therefore, the deployment took place on IaaS, PaaS and Function as a Service (FaaS) platforms. Also, end user devices are involved, for example for running the *SCM workbench* (see Figure 14). That variety of platform types was chosen to provide inside information on how a new engineering concept could be supported by different software architecture approaches to be efficient in terms of development time, Continuous Integration (CI), resource efficiency and scalability.

A. Infrastructure as a Service

In the context described above, IaaS is used to describe a hosting platform based on bare-metal and hosted hypervisors. It provides a variety of virtualized operating systems that are in compliance with corporate IT regulations.

For the prototype, the services hosted on classical virtual machines are mainly databases used as persistent layers for distributed Web applications. The main reason for not hosting the web applications together with their respective persistence layer are resource restrictions. Current company policies prevent external access to the databases if they are part of the same microservice image as the hosting environment. This would either limit database management to a web-based command line interface or require the implementation of a Web service deployed in the same container. Also, other external services could not be used to access the databases. This limitation is purely based on a decision made by the company's IT governance, but reflects day to day reality in corporate environments.

For any other Web application around the SCM prototype development, IaaS was avoided as the resource overhead cannot compete with PaaS or FaaS.

B. Platform as a Service

In the following Section, PaaS refers to an on-premise deployment of the *Red Hat OpenShift* [28] platform. It is a platform built around *Docker* [29] containers orchestrated and managed by *Kubernetes* on a foundation of *Red Hat Enterprise Linux*.

In the prototype, PaaS plays a critical role for the continuous integration strategy. The image format used for the deployments follows the Source-to-image (S2I) concept. S2I is a toolkit and workflow for building reproducible container images from source code [30]. S2I produces ready-to-run images by injecting source code into a container image and

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

letting the container prepare that source code for execution. The source code itself is hosted on an on-premise Github Enterprise [31] instance and the dependent resources are provided via an on-premise *Artifactory* [32] deployment that reflects the official sources of the required development environment such as Maven [33], npm, Python or NuGet.

The whole continuous deployment chain is secured via an exchange of keys and certificates to prevent disruptions for example due to company introduced password cycles for the developer and deployment accounts. The deployment speed is improved by using system instances for the S2I chain in the same geolocation of the company to prevent larger inter-site data transfers and round-trip times.

The microservice concept, together with PaaS, allows a massive reduction of resource allocations compared to an IaaS deployment, especially if the services are single and independent web applications.

There are still limitations in the corporate environment that currently prevent larger scale use of the technology. The current setup allows a limited number of pods per node, which becomes an issue when a service uses the scaling capability of the *OpenShift* platform. A second limitation is linked to the allocated sub-network and the deployment of the platform. All inter-service communication is routed via a unique company internal network. The PaaS instance does not re-use a network range that is already present in the company for inter-service communications as it would impose other challenges regarding communication from within the PaaS instance towards other company services. The rationale for the chosen PaaS implementation is primarily the reduction of classical virtual machines for simple hosting jobs and only secondarily the creation of a massively scalable infrastructure for new service applications.

To cope with these limitations the prototype furthermore reduces the deployment footprint of single services for certain applications as described below.

C. Function as a Service

FaaS is used for tiny stateless jobs, e.g., rendering of images. These services are monitored by an orchestrator that decommissions containers after idling for a defined time. This reduces resource usage further and has advantages in a scenario with a larger number of services.

The deployment architecture of the FaaS instance allows launching service containers within milliseconds. The applied software stack is *OpenFaaS* based on *Docker Swarm* running on a *Debian* [34] Virtual Machine (VM).

One FaaS instance consumes resources similar to a pod on the above mentioned PaaS environment and hosts numerous services without performance limitations. While PaaS exposes containers under their distinct Internet Protocol (IP) addresses, FaaS comes with a reverse proxy that hides all containers and requires less IP addresses. This reduces the effort for routing name resolution and their documentation.

V. Implementation and Integration

The implementation of the prototype framework is split into different logical bricks as depicted by Figure 5. The services and applications itself can be mapped to their specific deployment paradigm as listed in Table I. The *Architect Cockpit* allows a system architect to use existing models, to schedule the execution of simulations and to review results. The *SCM Workbench* enables SCM developers to create and version SCMs. The *Back End* provides different services such

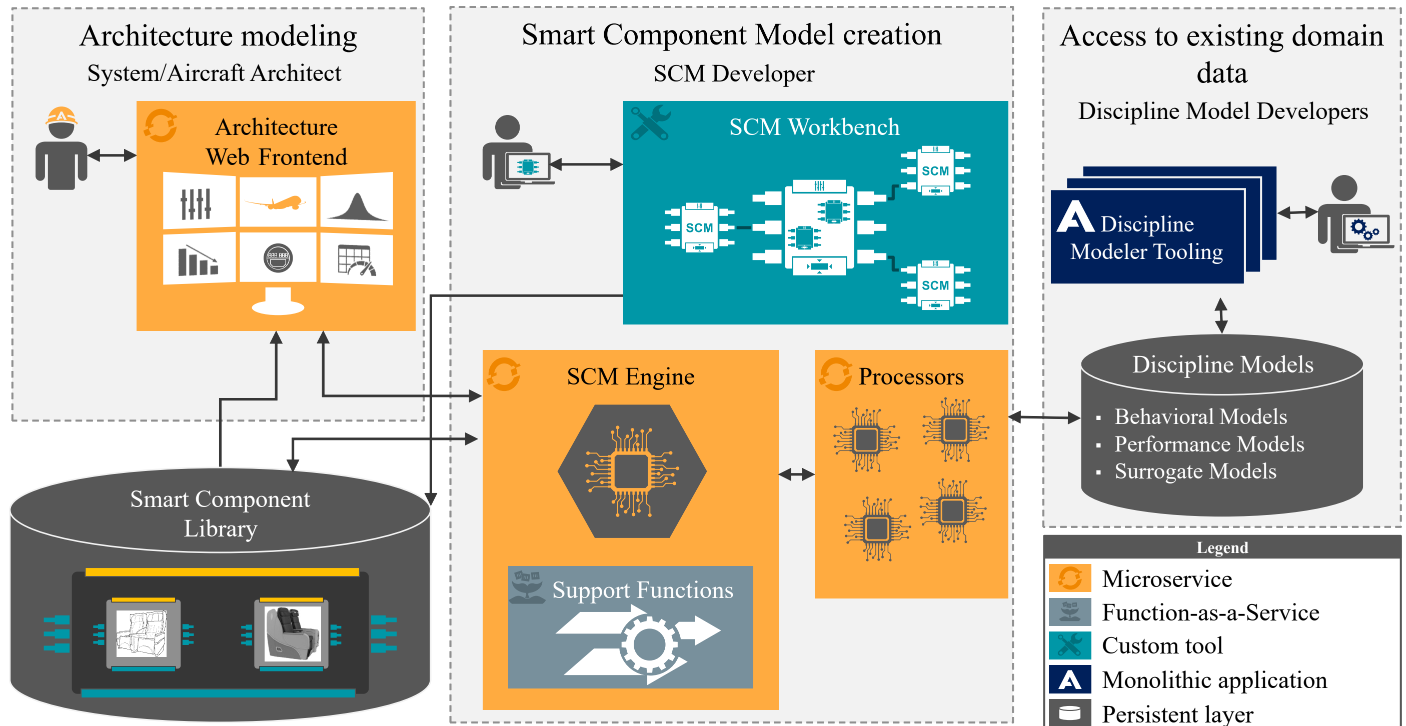

Figure 5. Service Environment & Deployment Infrastructure

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

as the orchestration of different processors to perform the execution of simulations.

TABLE I. Service Mapping to Deployment Paradigm

Deployment Paradigm	Service or Application
IaaS	Persistence Layers (Smart Component Library, MongoDB, Internal GitHub, Internal Artifactory)
PaaS	SCM Engine, Architecture Web Frontend, Service Dashboard, OpenTURNS Sampler, Node-Red, SCM Processors, Jenkins
FaaS	JSONata, Parallel-Coordinates
End User Device	SCM Workbench

A. Architect Cockpit

In order to reduce the workload and make the work for the architects as convenient as possible the interface for the cockpit is setup as an Angular Single Page Application (SPA). This allows using this entity without installing custom software and without bothering the user with update and migration procedures. The site is built using a Jenkins pipeline and then deployed on a specific git repository branch. A webhook on this branch triggers an *OpenShift* instance to build an Express.js server serving the previously build site on a PaaS cluster.

From a functional point of view the *Architect Cockpit* gives a reduced view on SCMs. Only information, which is necessary for the work of an architect is available and can be modified. This results in a nearly full intuitive usage of the interface and prevents faulty configurations. For example, some parameters can only be changed within a certain range. Ranges are defined by the model developer who knows the limitations best. The architect does not need to have a deep understanding of these limitations when using the predefined models.

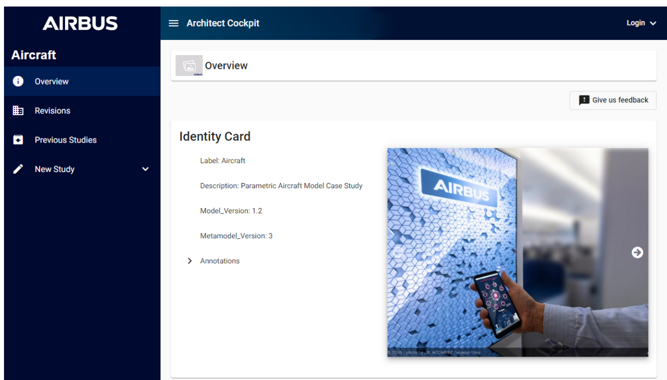

Figure 6. Architect entry point for gathering information on an SCM

Figure 6 shows the Architects entry point into using a SCM. In this case it is an aircraft architect opening a parametric aircraft model. He can then start a new study and set or change parameters of this model which Figure 7 shows. After running a calculation in the mixed-paradigm back-end the front-end renders an executive result and presents it to the architect. Figure 8 shows the result. If deeper insights into the calculation chain is required the architect can open a more

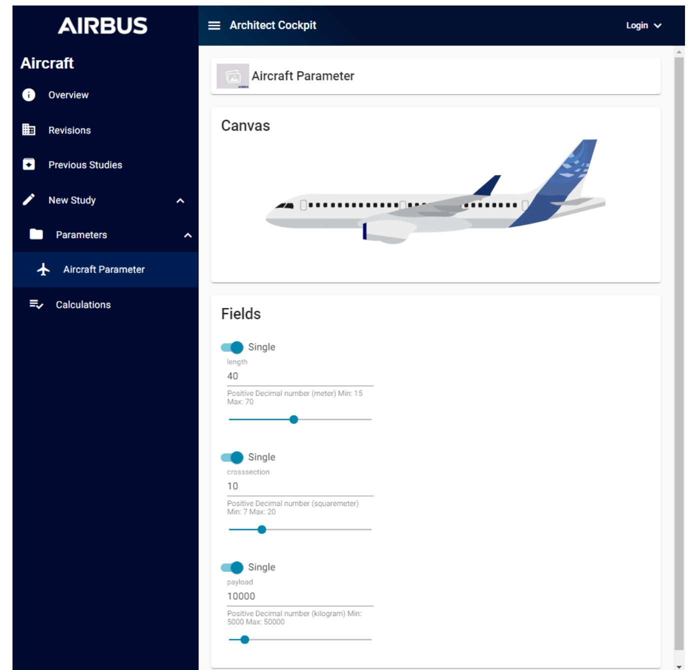

Figure 7. User interface for defining parameter values

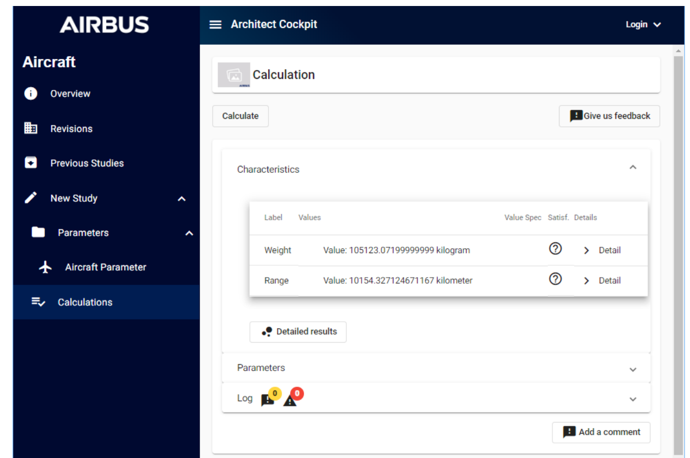

Figure 8. User interface for launching calculation with a SCM

detailed interactive report rendered as a bubble chart as shown in Figure 9. When selecting a range of values for a set of parameters an additional representation appears rendering all distinct runs of a study into a parallel coordinates plot. Figure 10 shows this interactive diagram. It is a data analytics tool that allows highlighting specific runs, filtering specific parameter and characteristic values as well as removing parameters from the diagram.

B. SCM Workbench

The *SCM Workbench* is a full-fledged graphical editor to work with SCMs implemented as a monolithic rich-client application. It is implemented in an Eclipse Rich Client Platform (RCP) and based on the Eclipse Modeling Framework (EMF) [35]. It is a modeling framework and code generation facility for building tools and other applications based on a

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

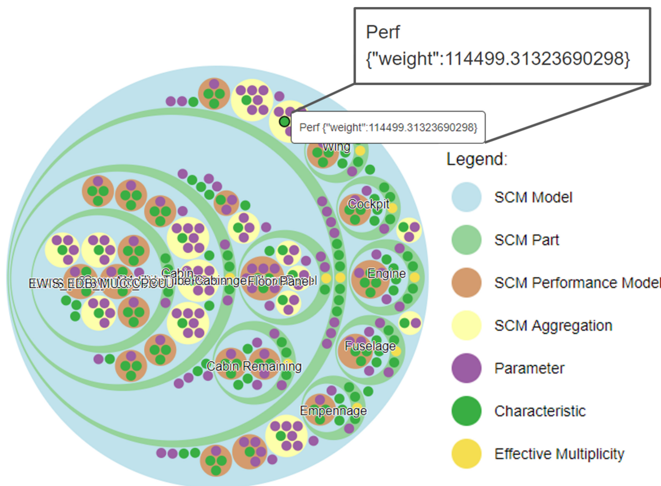

Figure 9. Bubble chart for browsing through the propagation of calculated values

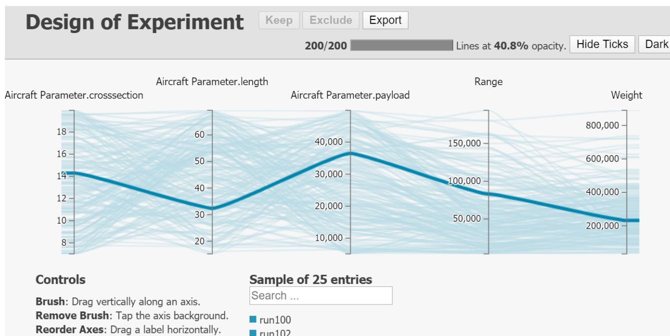

Figure 10. Design of Experiment explorer for navigation through large sets of sampled simulation runs

structured data model. EMF provides tools and run-time support to produce a set of Java classes from a model specification, along with a set of adapter classes that enable viewing and editing of the model, and a basic editor.

EMF is the basis for the Obeo Designer tool [36], which builds on the Eclipse Sirius project [37] and allows definition of graphic editors based on a defined EMF metamodel. This enables rapid prototyping of modeling solutions, which is ideal for a research/prototyping environment such as Airbus Central R&T. Changes to the metamodel are almost instantly available in the *SCM Workbench*, our prototype SCM modeling tool. On the other hand, EMF and *Obeo Designer* are mature and have been proven in industrial practice, e.g., *Capella*, the modeling tool from Thales that implements the *Arcadia* method is built with EMF and *Obeo Designer* as well [38].

Starting the *SCM Workbench* opens an application that is shown in Figure 11. The left toolbar allows browsing through the SCMs that exist in a project. A project is split into SCMs and representations. While the SCMs contain all functionality required for computing it, the representation adds additional information on how to render the SCMs within the workbench. Figure 12 shows the parameter view that allows the *SCM Developer* to model the propagation of parameters and characteristics through the SCM. Selecting entities in this view leads to its properties to show up in an editor at the bottom of the window. This example shows the parameter

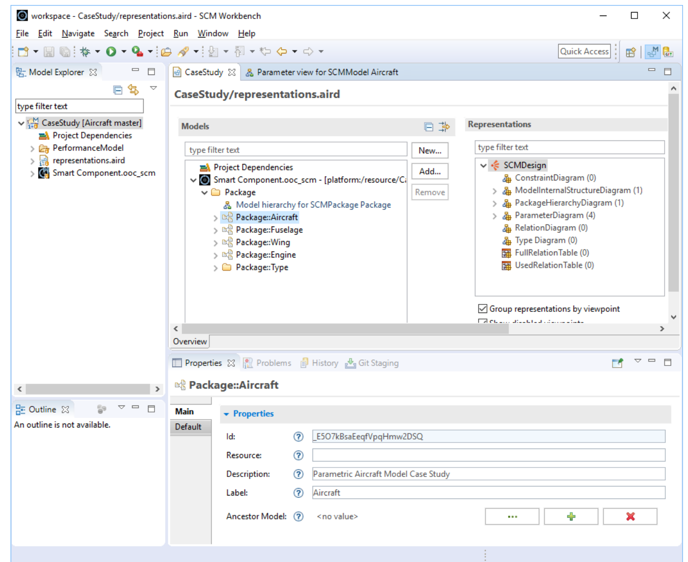

Figure 11. SCM Workbench showing the representations

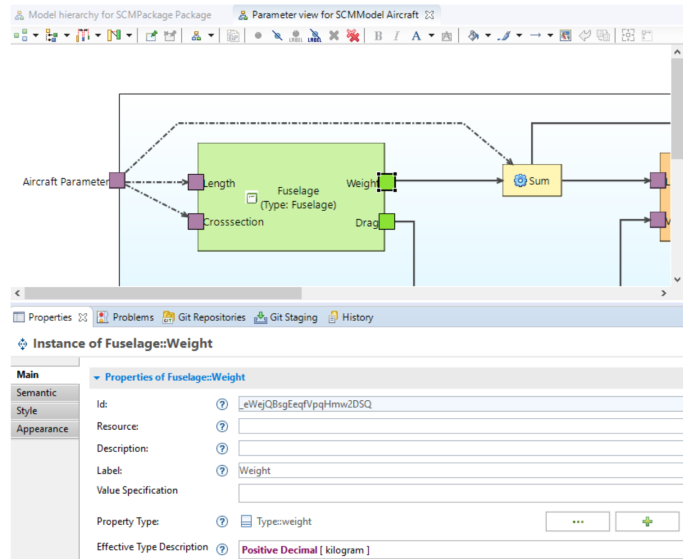

Figure 12. Parameter view in SCM Workbench

"Weight" of the SCM "Wing". The structure view shown in Figure 13 describes the architectural interdependency between underlying SCMs. In this case the "Engine" is attached at the "Pylon" to the "Wing" and the "Wing" is attached to the "Fuselage" at the "Belly".

Using such a rapid prototyping approach for the *SCM Workbench* can be easily misunderstood as just a proof-of-concept study. The final look and feel of the graphical editor for the SCMs is only limited by the amount of development time used for user experience (UX) polishing. The workflow and information accessibility as well as the connection to a versioning system is comparable to other commercially available modeling tools, which are well known by the developers. It is assumed that a SCM developer has to take a short on-boarding training before using the *SCM Workbench*.

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

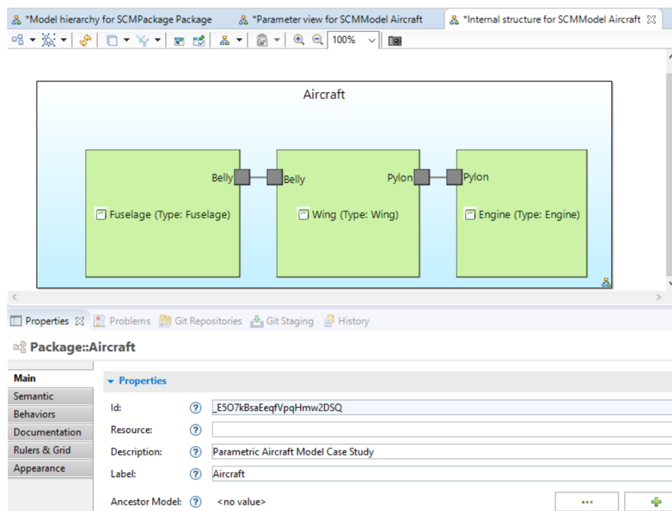

Figure 13. Structure view in SCM Workbench

C. Back End

The *back-end* is built from several different entities that are based on different paradigms. These entities are described in the following paragraphs.

1) SCM Library: The *SCM Library* stores the models that have been created using the *SCM Workbench*. It is based on Connected Data Objects (CDO) a Java model repository for EMF models and metamodels. The specific implementation in use is the *Obeo Designer Team Server (ODTS)*, which enables concurrent engineering of EMF models. A custom plug-in allows other services and applications to access the model repository through a Representational State Transfer (REST) interface. Due to its complex deployment strategy the *SCM Library* is deployed in an IaaS environment, which allows more user interaction during updates.

2) SCM Engine: The *SCM Engine* can interpret SCMs, check constraints and run parametric calculations either as a single simulation run or as a Design of Experiments (DoE) setup with multiple samples. It is a Java application executed in an OpenJDK VM. Access to the engine is established through REST interfaces that are hosted on a Jetty server. The endpoints are described and documented using the Jersey framework. The *SCM Engine* is hosted on a PaaS instance and allows rolling updates, automated builds and scaling.

3) Model Processors: The *Performance Model API* serves as a glue between external domain-specific models with their own solver or simulation engine and the *SCM Engine*. A *Model Processor* is an application that implements this API to execute a specific model type. The API enables the *SCM Engine* to orchestrate simulations tools in a unified way and guides developers through the process of integrating additional simulation tools into this environment. In order to include a new model type in the SCM application framework, a model type specific *Model Processor* has to be implemented that implements the *Performance Model API* and connects to the model type specific solver or simulator. A reference implementation shows how this works for Excel models. An Excel model is processed by a Java application running in an OpenJDK VM using the Apache POI framework. Depending on the type of model and, e.g., the license and installation requirements of the model solver or simulator, the *Model Processor* can be deployed in any of the available deployment options IaaS, PaaS and FaaS.

Figure 14 depicts how the components of the SCM tool framework prototype are deployed in our infrastructure.

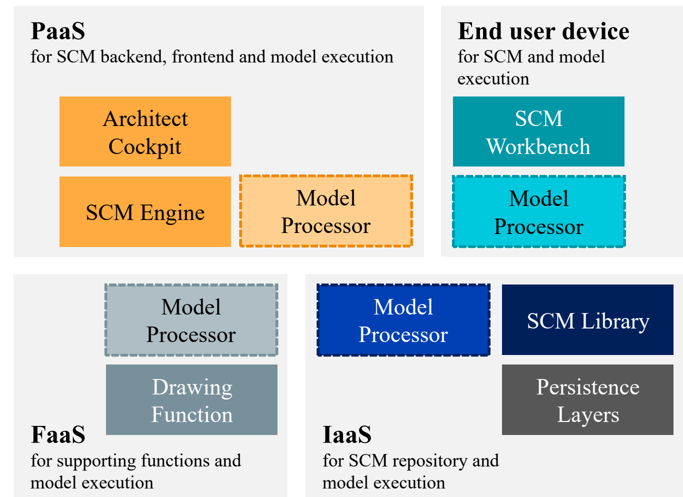

Figure 14. Prototype tool deployment

To make the polyglot approach of the MSA work and integrate each service all participating entities need to agree to a commonly understood interface. For the prototype REST over HTTP was chosen as the default interface combined with JavaScript Object Notation (JSON) as serialization format. REST over HTTP is a de facto standard since almost every technology stack provides at least an HTTP API if not specialized REST frameworks and clients such as Java API for RESTful Web Services (JAX-RS). JSON as a serialization format is accepted and provides solid tooling on all integrated technologies. In addition many front-end frameworks natively support JSON such as *JavaScript* or *Ruby*. This eases the integration work needed to be done for the implementation of our demonstrators mainly the *Architect Cockpit*. As an added bonus it is easily digestible by human user, which helped tremendously with debugging. To build up process chains utilizing the deployed microservices we selected *Node-RED*. It provides all the tools necessary to handle HTTP based REST APIs and JSON based message bodies and is integrated well into the existing environment.

If we dissect the service environment infrastructure shown in Figure 5 we can see what protocols are being used in the communication between the different services. As Figure 15 shows the communication through the HTTP RESTful web services is the predominate form of communication in our prototype. The only deviation from this paradigm occurs in the communication between the *SCM Workbench* and the *SCM Library* where the tool provider specifies Transmission Control Protocol (TCP) as interface. We did not challenge this implementation since it is provided by the *Obeo Designer Team Server*; however, we implemented a *Mapper Plugin* that provides access to the stored SCM Models via a REST interface to incorporate it in our service environment. The HTTP REST approach is especially useful for incorporating the various Discipline Models Processors into the overall pro-

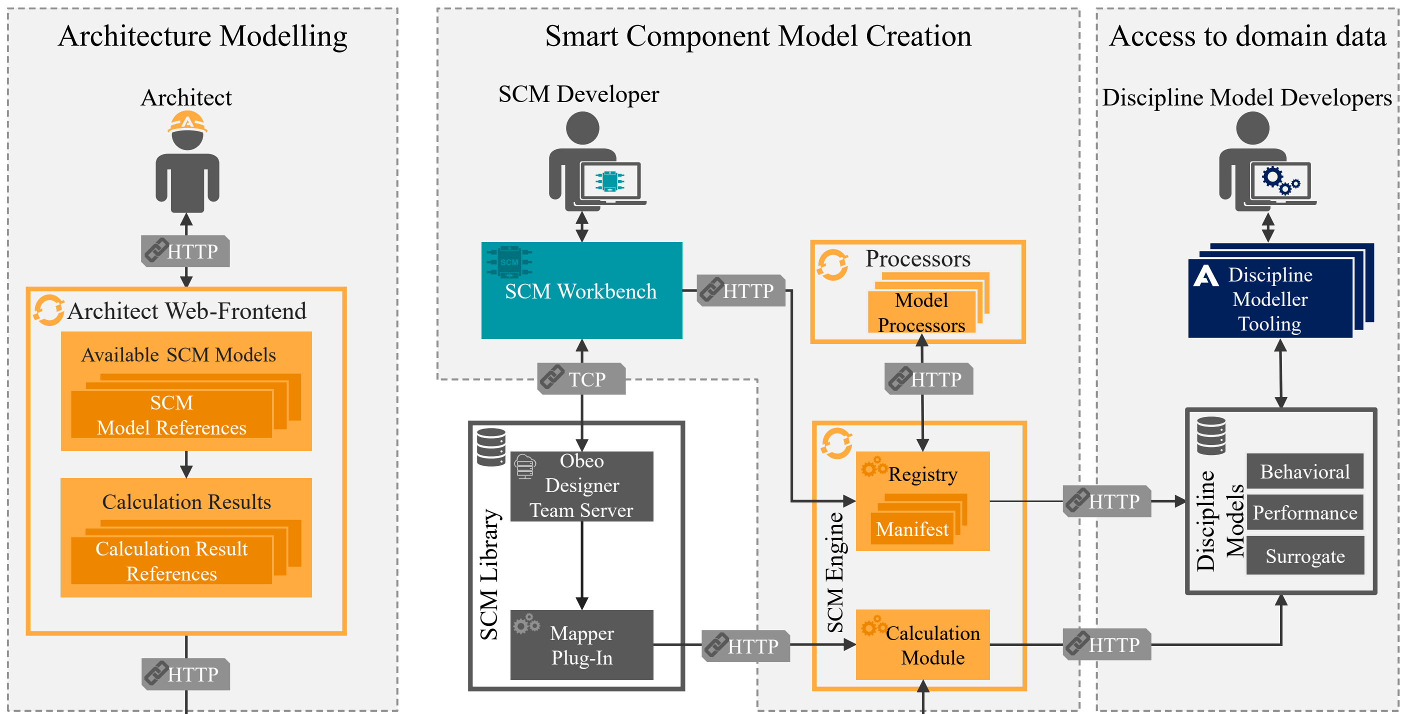

Figure 15. The use of REST interfaces in our prototype

cess. Since every processor works on a dedicated technology stack designed for his task utilizing a unified interface makes integration into the overall process network easy.

After the explanation of all the building blocks we will present a case study to demonstrate the framework in action.

VI. Case Study

This section describes a case study that has been made using the SCM Workbench with the purpose of showing individual features of the tool-chain.

The described model has been provided showing an Aircraft consisting of Fuselage, Wing and Engine. The hierarchical view, depicted in Figure 16, of the SCMs shows that all models are placed in one package. The hierarchical view is equivalent to SysML's class diagram. This view shows that the Aircraft is decomposed into Fuselage, Wing and Engine. However, it does not show their interdependency.

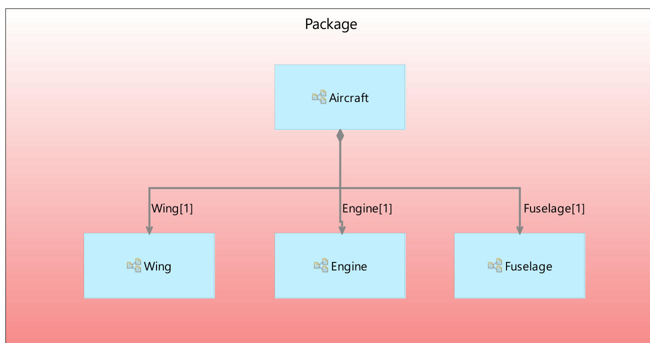

Figure 16. Case study: Hierarchy view

All models are SCMs. The Aircraft is a special case, because it is decomposed into other SCMs.

In order to describe the interdependency within the Aircraft SCM the internal structure diagram was used, shown in Figure 17. It shows that the Fuselage and Wing are connected with each other at the Belly. Wing and Engine are connected at the Pylon. This diagram is similar to SysML's internal block diagram. The blue box shows the scope of the SCM. The green boxes represent the decomposition into other SCMs.

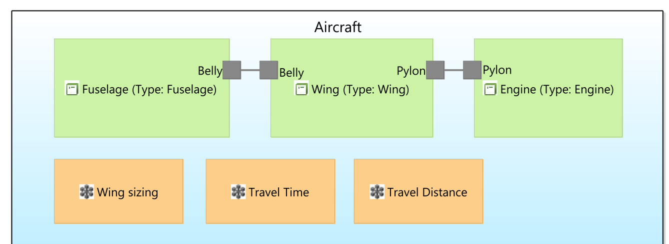

Figure 17. Case study: Structure view

Additionally there is the parameter view, shown in Figure 18, which describes in-transient calculation of the model characteristics from the set of parameters. All parameters appear in purple color and all characteristics in green color. This specific SCM describes the calculation of the Aircraft's weight and range from its length, cross-section and payload. Light yellow color represents an aggregation function that generates one value from a list of values. In this case both aggregation nodes compute a sum. Direct calculations on performance models are represented by the orange boxes. They refer to a domain model manifest, describe machine-readable how these domain models

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

shall be executed and how parameters and characteristics are transmitted to and from them.

In order to ensure compatibility between parameters and/or characteristics a common type system is available. It allows to specify the data to be exchanged during the simulation between the components. Besides the typical basic types it allows structured types like lists and key-value-pairs. An additional feature is to nest types as references into other types. As an example the Aircraft parameters are shown in Figure 19. As an example for a basic type the weight is shown in Figure 20.

VII. Evaluation

Evaluating the mixed-paradigm approach, we experienced that developers where able to create a working deployment much faster compared to the traditional approach using virtual machines. This also includes the amount of times that a new version of the service was built from once a week to several times a day using the automated CI pipeline. This increased the general development velocity as well as the prototypes feature set, which helped us to tailor the application to our stakeholder needs.

The raised deployment speed increased the number of times we experienced broken client applications. This was due to a violated interface contract between the services if the new features where not integrated properly. A well-defined and adhered to interface specification is paramount for the success of introducing this mixed-paradigm approach.

In general, we noticed a greater sense of ownership of single developers over their service/code, which lead to a hike in the overall implementation quality. The mandatory usage of the git version control system increased the maintainability of the code base. The combination of git and the *OpenShift* framework made it easy to recover from failures and faulty builds, which lead to a constant up-time of all services. In the future the introduction of additional agile software development principles like *Test Driven Development* could further increase the code quality.

However, the deployed solution is marked as a proof of concept or prototype which lead to the conclusion that it is not ready for operational use for various reasons. The main focus of the development lay on the proof of feasibility of the SCM modeling methodology as described in [6]. As soon as first parts of the prototype were available selected engineering departments started trials with the solution, which lead to further improvements of the underlying methodology as well as the overall usability. The overall perception was very positive, which lead to the conclusion that the developed methodology points in the right direction as well as the performance of the proposed tool set based on the described approach in this paper.

The mixed-paradigm approach that was used to develop and deploy the prototype discussed in this paper led to reduced complexity, lower coupling, higher cohesion and a simplified integration. This in turn enabled agile collaboration for continuous delivery and integration of the solution.

VIII. Outlook and future challenges

In the previous sections, we described how MSA can support the chosen polyglot approach utilizing a variety of different technology stacks and storage solutions. This enabled us to select the most fitting technical solution for the required functionality. Additionally the network based architecture provides an environment that is well suited for a multinational company like Airbus with sites scattered throughout different sites and IT domains. It also provided a commonly understood deployment layer for our cross-functional project team.

MSA supports us with the agility and velocity needed to convince our customers of our approach and implement a prototype that can handle the complexity of our SCM modeling approach. However, during the development we found stumbling blocks that need awareness once the scale changes from a research project prototype to a full scale industrial roll out.

Corporate IT – The proposed environment builds and hosts microservices in an agile and automated way. This requires the setup and maintenance of a CI pipeline (in our case *OpenShift*/*GitHub*), which results in additional costs as well as an IT department that is capable of dealing with those investments. Additionally setting up certificate chains and firewalls to allow for secure communication inside the corporate network need to be accounted for. On the developer side roadblocks like proxy server hindering communication and enabling cross-origin resource sharing (CORS), which allows for communication between different domains need to be taken care of.

Service discovery – Once we reached a critical mass of microservices environment we discovered that it is hard to keep track of what services have already been implemented and what functionality each service provides. Even in our research project this point was reached rather quickly. Thus, we introduced *Swagger* [39] as a Web based documentation for all our services and implemented a simple dashboard where services could be registered against. This allowed for manual service discovery across the team. In the future automated service discovery through bots and processable service descriptions will bring more value to the MSA approach by handling the sprawling service environment.

Now that we optimized the CI pipeline in the first half of the project we experience a rapid increase in deployed services. This allowed us to swiftly introduce new functionality as microservices, boosting the capabilities of our proof of concept prototype. It shows that MSA can initially speed up the implementation velocity of a new project. Once we continue with the project more efforts will go towards managing the volume of services as well as (network) performance and reliability.

IX. Conclusion

Past experience shows that current aircraft and aircraft system development processes are not suitable for keeping up with the rising complexity of products. Those processes are under pressure from market-driven demands for faster, and from business-driven demands for cheaper aircraft programs. In this paper, we presented a proposal for a change from a traditionally linear development approach to one that includes a parallel, OOC component development phase. This approach required a supporting IT infrastructure that was built as a prototype at Airbus in the frame of a research project. To reduce both time and resources required for building this prototype state-of-the-art architecture and deployment paradigms were

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

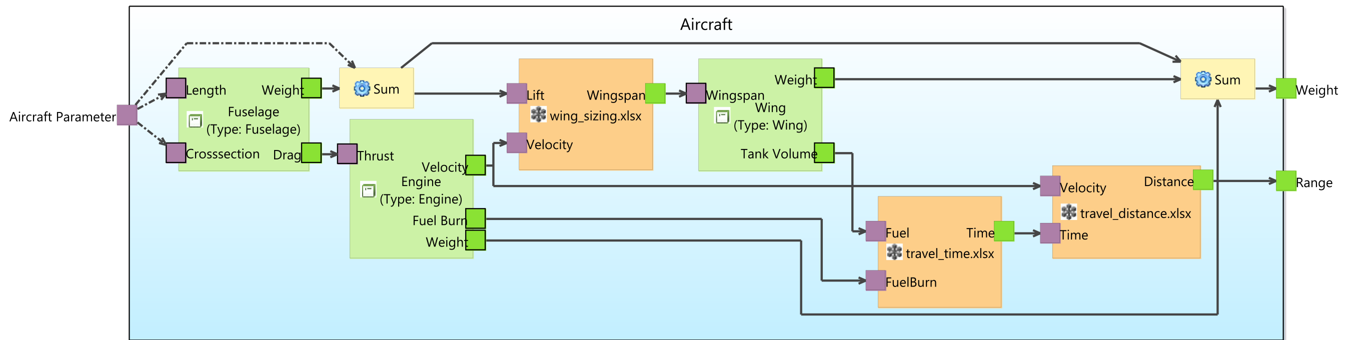

Figure 18. Case study: Parameter view

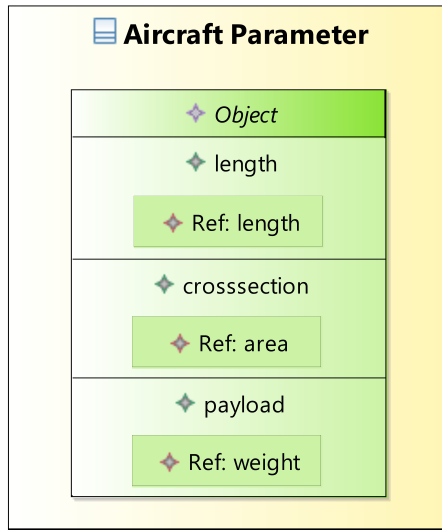

Figure 19. Case Study: Aircraft parameter modeled in the type system

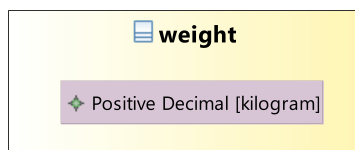

Figure 20. Case Study: Basic parameter representation in the type system

used and mixed with more classic approaches to get the best of both worlds.

A direct, specific and measurable comparison between the described mixed-paradigm and a classical approach is not possible as it would have required the same infrastructure landscape to have been developed and deployed multiple times using different concepts. Nevertheless, implementers were given the freedom to decide for every distinct artifact to freely choose the paradigm used for implementation. Furthermore, developers were allowed to split artifacts, which enables to select the right paradigm for each problem within. Later the interface documentation allowed the developers to easily re-implement an artifact using a different paradigm in case the initial decision for a specific paradigm reveals to have been not an optimal choice. Therefore, the selection of the right paradigm appears to be inherent and native. To support a newly developed MBSE approach called SCM modeling, a supporting application framework prototype had to be developed. Instead of a single architecture and deployment paradigm, a mixed-paradigm approach was followed to take the advantages of the different options and to consider external constraints coming from the IT governance. The software bricks were implemented in monolithic, SoA, microservice and serverless architecture glued together by REST interfaces over HTTP. The deployment took place on desktop-PC, IaaS, PaaS and FaaS platforms. It provided insight into how a new engineering concept could be supported by different software architecture approaches to be efficient in terms of development time, continuous integration, resource efficiency and scalability.

References

[1] P. Helle, S. Richter, G. Schramm, and A. Zindel, "Mixed-Paradigm Framework for Model-Based Systems Engineering," in FASSI 2019, The Fifth International Conference on Fundamentals and Advances in Software Systems Integration. IARIA, 2019, pp. 8–13.

[2] N. Dragoni et al., "Microservices: yesterday, today, and tomorrow," in Present and ulterior software engineering. Springer, 2017, pp. 195–216.

[3] J. Ghofrani and D. Lübke, "Challenges of microservices architecture: A survey on the state of the practice." in ZEUS, 2018, pp. 1–8.

[4] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," IEEE Software, vol. 35, no. 3, 2018, pp. 24–35.

[5] D. D. Walden, G. J. Roedler, K. Forsberg, R. D. Hamelin, and T. M. Shortell, Eds., Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, 4th ed. Hoboken, NJ: Wiley, 2015.

[6] P. Helle, S. Feo-Arenis, A. Mitschke, and G. Schramm, "Smart component modeling for complex system development," in Proceedings of the 10th Complex Systems Design & Management (CSD&M) conference, forthcoming.

[7] A. Reichwein and C. Paredis, "Overview of architecture frameworks and modeling languages for model-based systems engineering," in Proc. ASME, 2011, pp. 1–9.

[8] Object Management Group, OMG Systems Modeling Language (OMG SysML), v1.2. OMG, Needham, MA, 2008.

[9] M. H. Sadraey, Aircraft design: A systems engineering approach. John Wiley & Sons, 2012.

[10] Object Management Group, OMG Unified Modeling Language (OMG UML), v2.3. OMG, Needham, MA, 2010.

[11] J. A. Estefan et al., "Survey of model-based systems engineering (mbse) methodologies," Incose MBSE Focus Group, vol. 25, no. 8, 2007, pp. 1–12.

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[12] A. Albers and C. Zingel, “Challenges of model-based systems engineering: A study towards unified term understanding and the state of usage of sysml,” in Smart Product Engineering. Springer, 2013, pp. 83–92.

[13] R. Karban, R. Hauber, and T. Weilkiens, “Mbse in telescope modeling,” INCOSE INSIGHT, vol. 12, no. 4, 2009, pp. 24–31.

[14] A. Van Deursen, P. Klint, and J. Visser, “Domain-specific languages: An annotated bibliography,” ACM Sigplan Notices, vol. 35, no. 6, 2000, pp. 26–36.

[15] J. Gray and B. Rumpe, “Uml customization versus domain-specific languages,” 2018.

[16] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, “Serverless programming (function as a service),” in 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2017, pp. 2658–2659.

[17] A. Karmel, R. Chandramouli, and M. Iorga, “Nist definition of microservices, application containers and system virtual machines,” National Institute of Standards and Technology, Tech. Rep., 2016.

[18] I. Baldini et al., “Serverless computing: Current trends and open problems,” in Research Advances in Cloud Computing. Springer, 2017, pp. 1–20.

[19] N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, “Microservices: Migration of a mission critical system,” arXiv preprint arXiv:1704.04173, 2017.

[20] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, “From monolithic to microservices: an experience report from the banking domain,” Ieee Software, vol. 35, no. 3, 2018, pp. 50–55.

[21] M. Villamizar et al., “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” in 2015 10th Computing Colombian Conference (10CCC). IEEE, 2015, pp. 583–590.

[22] ——, “Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures,” Service Oriented Computing and Applications, vol. 11, no. 2, 2017, pp. 233–247.

[23] M. Fowler and J. Lewis. Microservices a definition of this new architectural term. [Online] http://martinfowler.com/articles/microservices.html [Accessed: 11 September 2019].

[24] T. Cerny, M. J. Donahoo, and M. Trnka, “Contextual understanding of microservice architecture: current and future directions,” ACM SIGAPP Applied Computing Review, vol. 17, no. 4, 2018, pp. 29–45.

[25] D. Shadija, M. Rezai, and R. Hill, “Towards an understanding of microservices,” in 2017 23rd International Conference on Automation and Computing (ICAC). IEEE, 2017, pp. 1–6.

[26] P. Di Francesco, I. Malavolta, and P. Lago, “Research on architecting microservices: trends, focus, and potential for industrial adoption,” in 2017 IEEE International Conference on Software Architecture (ICSA). IEEE, 2017, pp. 21–30.

[27] C. Pahl and P. Jamshidi, “Microservices: A systematic mapping study.” in CLOSER (1), 2016, pp. 137–146.

[28] RedHat, “Openshift,” https://www.openshift.com/, 2019, [Online; accessed 3-February-2020].

[29] Docker Inc., “Docker,” https://www.docker.com/, 2019, [Online; accessed 3-February-2020].

[30] A. Lossent, A. R. Peon, and A. Wagner, “PaaS for web applications with OpenShift origin,” Journal of Physics: Conference Series, vol. 898, oct 2017, p. 082037.

[31] GitHub, Inc., “Github,” https://github.com/, 2019, [Online; accessed 3-February-2020].

[32] JFrog Ltd, “Artifactory,” https://jfrog.com/artifactory/, 2019, [Online; accessed 3-February-2020].

[33] The Apache Software Foundation, “Maven,” https://maven.apache.org/, 2019, [Online; accessed 3-February-2020].

[34] Software in the Public Interest, Inc., “Debian,” https://www.debian.org, 2019, [Online; accessed 3-February-2020].

[35] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, EMF: eclipse modeling framework. Pearson Education, 2008.

[36] Obeo, “Obeo designer,” https://www.obeo.fr/en/, 2019, [Online; accessed 3-February-2020].

[37] V. Viyović, M. Maksimović, and B. Perisić, “Sirius: A rapid development of dsm graphical editor,” in IEEE 18th International Conference on Intelligent Engineering Systems INES 2014. IEEE, 2014, pp. 233–238.

[38] P. Roques, “MBSE with the ARCADIA Method and the Capella Tool,” in 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), Toulouse, France, Jan. 2016.

[39] SmartBear Software, “Swagger,” https://swagger.io/, 2019, [Online; accessed 3-February-2020].

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org