A Cloud-native Implementation of the *Simulation as a Service*-Concept Based on FMI

Moritz Stüber[1] Georg Frey[1]

[1]Chair of Automation and Energy Systems, Saarland University, Germany,
{moritz.stueber,georg.frey}@aut.uni-saarland.de

Abstract

Providing modelling and simulation capabilities *as a service* promises to increase their value by improving accessibility for non-expert users and software agents as well as by leveraging cloud-computing technology to scale simulation performance beyond the capabilities of a single computer. In order to reach this potential, implementations must align their design with the architectural styles of cloud computing applications and the web in general. We present an open-source, cloud-native Simulation as a Service (SIMaaS)-implementation that gives access to models and allows simulating them on the web. The implementation uses Functional Mockup Units (FMUs) for co-simulation as an executable form of a model and relies on FMPy for simulation. It is realized as a microservice in the form of a REST-based HTTP-API. Functionality and performance are demonstrated by using the service to create ensemble forecasts for PV systems and to search for an optimal parameter set using a genetic algorithm. Conceptual limitations and the resulting opportunities for further work are summarized.

Keywords: simulation as a service, cloud-native simulation, service-oriented software architecture, FMI 2.0

1 Introduction

There exist scenarios in which it is useful to execute simulations on a distributed set of computing resources that can be scaled according to demand and beyond the capabilities of a single machine. Examples for this include simulations which are part of a series of many simulations to be evaluated as a whole; as for example in parameter fitting or sensitivity analysis applications. Also, simulations might be part of a (recurring) larger process, such as providing necessary forecasts for flexibility management in the context of smart grids.

The term *cloud computing* denotes a set of desirable characteristics for accessing a set of computing resources over the internet, as well as characteristic service models and deployment models (Mell and Grance 2011). From a user's point of view, the essential characteristics are that software or computing resources are available *as a service* via the internet, meaning that the resources are readily available without the need for manual installation of hardware and/or software. Consumers can use services without the need for human activity on the side of the provider (*on-demand self-service*), often without apparent limitations, and they have access to metrics for their service usage (*measured service*). Users are billed according to service usage in terms of these metrics (*pays-as-you-go cost model*).

Cloud-based end-user applications usually integrate several services to realize their functionality as it has been found that programmers can effectively realize the desirable characteristics of cloud computing by exposing pieces of functionality as a set of independent services in a so-called Service-oriented Architecture (SOA). In the abstract, SOA is "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains" (OASIS 2006, line 864), where a *service* is defined as the "mechanism by which needs and capabilities are brought together" (OASIS 2006, line 174). More specifically, a service can be seen as the offer to perform work for others; as the service interface which specifies information model, behaviour model and applicable usage policies; and as a specific service instance.

In practice, SOAs can be successfully realized as a set of *microservices* in the form of Representational State Transfer (REST)-based Hypertext Transfer Protocol (HTTP)-Application Programming Interfaces (APIs) which exchange machine-readable representations such as JavaScript Object Notation (JSON) and define their interface according to a formal specification such as the OpenAPI Specification (OAS). The term microservice is used to point out that services best implement exactly one functionality only ("do one thing well"). Representational State Transfer (REST) is the name of the architectural style of the web, in other words a name for its key design principles. Consequently, a REST-based[1] HTTP-API attempts to implement these design principles, acknowledging the sucess of the web and attempting to inherit its positive properties.

Applications which are intentionally designed to work well in the cloud and consequently realize the desired characteristics are called Cloud-native Applications (CNAs). The goal of the presented work is to pro-

[1] Using the term REST-based instead of RESTful indicates that the developers are aware that the term "RESTful" is frequently misused and that their software does not fully realize the REST constraints.

vide a state of the art Simulation as a Service (SIMaaS)-implementation based on an established open standard for model export and to represent the ability to perform simulations as REST-compatible as possible withouth actually realizing the so-called Hypermedia As The Engine Of Application State (HATEOAS) constraint. Furthermore, it should be shown that the desired characteristics implied by the term "cloud-native" are realized.

The remainder of this paper is organized as follows: first, the concepts and abstractions for providing software *as a service* in general and the motivation for providing modelling and simulation as a service (MSaaS) are outlined in section 2. The high-level requirements that follow from the choice of concepts are summarized. Second, the software architecture and software stack for the developed solution are explained in section 3. Restrictions posed on Functional Mockup Units (FMUs) to be used with the implemented software are stated. Third, exemplary use cases for demonstrating functionality and performance are described in section 4. Last, related work is outlined in subsection 5.1 and conceptual limitations of the devised solutions as well as the resulting opportunities for further research are discussed in subsection 5.2.

2 Concepts

Providing MSaaS is a multi-faceted endeavour at the intersection of modelling and simulation (M&S), information science and software development and -operations (DevOps). The core hypothesis of MSaaS is that usability and reuse can be increased by making M&S functionality available to a broader audience via the internet; that functionality can be improved by facilitating the composition of M&S resources; and that performance can be improved by deploying applications in the cloud (see Stüber, Exel, and Frey 2018, section 2 for a detailed explanation and references to original research).

Three recent reviews on MSaaS outline the design space and identify high-level requirements and architectural choices that should guide the design and implementation of specific MSaaS solutions.

First, Shahin, Babar, and Chauhan map out the Architecture Design Space (ADS) for MSaaS by presenting the results of a Systematic Literature Review (SLR) performed with the aim to identify and describe the state of the art (Shahin, Babar, and Chauhan 2020). They categorize the primary studies considered according to different criteria such as the architectural style, the main drivers for architectural decisions, and quality attributes. Additionally, they ponder the implications of the chosen architectures and identify strenghts and weaknesses. The authors conclude that MSaaS-realizations most often use a *layered approach* to build applications; that *containerization* is employed to improve deployability; and that *effective interfaces for end users* that hide complexity and technicalities motivate their development (Shahin, Babar, and Chauhan 2020, section 5).

Second, Hannay, Berg, et al. (2020) reason about the infrastructure capabilities they deem necessary for realizing entire MSaaS ecosystems at scale. Based on "a systematization of concepts from ongoing deliberations on MSaaS" (Hannay, Berg, et al. 2020, section 3), the authors first review the service concepts of the North Atlantic Treaty Organization (NATO) MSaaS reference architecture (Hannay and Berg 2017) and then elaborate on the functionality required for realizing MSaaS ecosystems. Their reasoning is structured around the themes *data management*, *service description and -discovery*, *composition and interoperability* and the *management of different components*. The findings are mostly conceptual in nature, likely useful for verbalizing and contextualizing design questions and -decisions when faced with implementing specific SOAs containing M&S capabilities. The authors also note that solutions for supporting, yet – from an operational perspective – highly relevant functionality such as logging, metering and monitoring are readily available.

Third, Kratzke and Siegfried (2020) focus on the consequences expected from leveraging the cloud for M&S services. Using their work on and definition of CNAs (Kratzke and Quint 2017) as a basis, they propose a definition for what Cloud-native Simulations (CNSs) are in terms of a textual definition (Kratzke and Siegfried 2020, section 4.3), a cloud-native simulation stack and a cloud simulation maturity model. They summarize the software engineering trends in cloud computing as the evolution of deployment strategies to maximize resource utilization (smaller deployment units, elasticity); the use of microservices as an architectural style that supports the aforementioned; and the emergence of microservice engineering ecosystem components for container orchestration, monitoring, et cetera. The authors conclude that the same trends are to be expected for CNS architectures and that, like CNAs in general, CNSs should strive to isolate state in a minimum of stateful components.

In alignment with the findings of Kratzke and Siegfried, section 4.2, we decided to create a microservice realizing the SIMaaS-concept in the form of a REST-based HTTP-API, relying on containers as deployment units to be operated on a clustered elastic platform.

As a consequence of the decision for an interface design based on REST, specifically the *uniform interface constraints*, M&S capabilities need to be represented as *resources* of which *representations* can be transferred when HTTP verbs are applied to them (Verborgh, Hooland, et al. 2015). In other words, a mapping between entities of the application domain, such as a models and simulation results, and uniquely identifiable conceptual resources that constitute the service interface is required.

In the context of the developed SIMaaS-API, the entities of the application domain to be exposed as resources are models, model instances, simulations, and simulation results[2].

[2] The definitions below do not claim to be universally applicable;

Models are well-posed system models that could be simulated once all parameters are set; but the parameters are *not* set yet.

Model instances are system models that *do* have all parameters set (either explicitly or by relying on default values) and could be simulated as soon as initial conditions and input values are provided. The parameters of a model instance cannot be changed; a change in parameters always leads to a new model instance.

Simulations combine a model instance with initial conditions, input trajectories, a solver and the corresponding solver settings. They also have a state, for example `new`, `running` or `finished`, and link to their result if and only if (iff) it exists. Like model instances, simulations cannot be changed once they are created.

Simulation results contain the actual results of exactly one specific simulation. They cannot be updated either.

REST demands that each message must be *self-descriptive*, meaning that it must be actionable independent of any possible prior interaction with the same client. To support this, HTTP provides only a few methods with specified semantics and properties, for example `GET` or `POST`. In combination with the concept of resources, this means that actions which are prevalent in classical M&S environments such as Dymola need to be represented differently (compare Verborgh, Hooland, et al. 2015, section 3.3). For example, there is no such thing as a `SIMULATE` method in HTTP. Assigning a Uniform Resource Locator (URL) to an action contradicts the idea of resources and is therefore incompatible with REST. Thus, the action of starting a simulation is represented instead by `POST`ing a representation of a new `simulation`-resource to the API. Internally, the API starts the simulation as part of the handler that registers the new `simulation`-resource. Once the simulation is finished, a resource exposing the simulation result is created.

As a consequence of this design, the application state (the state of the interaction between service consumer and service instance) is only stored in the state of the resources exposed by the service, including their existence or absence. This is a desired property. However, it also means that clients need to poll the `simulation`-resource by repeatedly sending `GET`-requests in order to know about the existence of a result or the failure of a simulation (compare subsection 5.2).

Note that the developed SIMaaS-API does *not* expose the Functional Mockup Interface (FMI) functions described in the standard document (Modelica Association 2020), but more abstract/high-level functionality as outlined in Table 1.

Based on the choice of resources and a decision on how to represent actions of the application domain in terms of the addition/update/removal of resources, the service interface can be specified. Several specification formats exist, of which the OpenAPI Specification (OAS)[3] has gained widespread support. Formally specifying the service interface has several benefits: first, the service interface description serves as unambiguous documentation both for users and developers of the service. Second, parts of the service implementation can be automatically generated from the service description, such as routines for input validation, the routing of requests or a website rendering the OAS for human users. Third, test cases for verifying that the API behaves as advertised can be generated automatically from the service description.

However, relying on the OAS to specify an interface that exposes models and the ability to simulate them quickly leads to a conceptual problem: the OAS is a *static* interface description written at design time, whereas the parameters for model instantiation and triggering simulations depend on the models to be used with the SIMaaS-instance, which are only added at run time.

Three solutions to this problem suggest themselves. First, the interface description could be kept so generic that the differences between models are abstracted. This would severly diminish the advantages of using a formal service description outlined in the penultimate paragraph and is therefore undesirable.

Second, the OAS could be regenerated dynamically each time a model is added to or removed from the SIMaaS-instance. This allows the OAS to be specific enough to fully realize its potential. The translation of constraints on parameters and inputs such as maximum or minimum values or unit specifications can be automated as long as these constraints are present in the model. This approach is realized in the SIMaaS-implementation presented in this paper.

The third approach would be to *not* use a service interface description at all and let users explore the capabilitites of the server dynamically by following links. This is how human users navigate websites as they are good at finding a way to achieve their goal on a web page and understand possible consequences of clicking links without actually following them. However, this is a challenging tasks for software agents and subject to ongoing research under the term *"hypermedia API"*. The possibilities for turning the presented SIMaaS-implementation into a hypermedia API will be discussed further in subsection 5.2.

3 Implementation

Below, the software architecture and software stack chosen to realize our goal of providing a state of the art SIMaaS-implementation based on FMI as an established open standard for model export are described. For practical reasons, some requirements are posed on FMUs to

they should be seen as specific to the developed software.

[3] `https://github.com/OAI/OpenAPI-Specification`

Table 1. Overview of the service interface in terms of HTTP methods, exposed resources and their meaningful combinations.

Method	Resource	Description
`POST`	/models	Add a new model to the API-instance
`GET`	/models/{model-id}	Retrieve a model representation from the API
`DELETE`	/models/{model-id}	Delete a model representation from the API
`POST`	/models/{model-id}/instances	Instantiate a model for a specific system
`GET`	/models/{model-id}/instances/{instance-id}	Get a representation of a specific model instance
`POST`	/models/{model-id}/instances/{instance-id}/experiments	Trigger the simulation of a model instance by defining an experiment
`GET`	/models/{model-id}/instances/{instance-id}/experiments/{experiment-id}	Retrieve a representation of a specific experiment definition and its status
`GET`	/models/{model-id}/instances/{instance-id}/experiments/{experiment-id}/result	Retrieve a representation of the results of a specific simulation run

be used with the service, which are explained in subsection 3.2. In order to get the exact same simulation results from simulation of the FMU as when simulating the model in a Modelica environment, the sequence of calling the FMI methods implemented in FMPy had to be changed as explained in subsection 3.3.

3.1 Software Architecture

Figure 1 shows the high-level software architecture. Exactly one component (A) provides the service interface in the form of a HTTP-API and stores the state, in other words the resources exposed. At least one, but potentially tens or even hundreds of stateless workers (W) run simulation jobs that they pull from a task queue (Q1). The simulation results are propagated back to component A through a second queue (Q2). Both queues do not store data permanently, and neither do the workers. Requests from users do not reach the API component directly but instead arrive at a reverse proxy (R). This reverse proxy is responsible for providing an encrypted Hypertext Transfer Protocol Secure (HTTPS) connection to the outside world.

The specific software components and libraries used for implementation were chosen based on the criteria that they are well-suited for the job; Free/Libre and Open-source Software (FLOSS); represent the state of the art; and that their use avoids re-implementing functionality that already has stable implementations.

The HTTP-API (A) possibly receives many requests at once, most of which result in requests to storage or other services which are operations that are very slow compared to pure computations, meaning that significant amounts of time are spent waiting. Therefore, Node.js[4] was chosen as the programming language as it provides excellent support for non-blocking input-output (IO) operations using `promises` and the `async/await`-syntax. Also, it is commonly used for implementing HTTP-APIs and consequently offers many useful libraries that support implementation, such as the Express-framework[5] and the `openapi-backend`[6]. Incoming requests are checked for validity against their schema in the OAS. Valid requests are then propagated to the appropriate handlers, which alter or retrieve resource state and enqueue simulation requests if necessary.

The internal representation of a simulation job couples the worker implementation to the API. Workers retrieve these representations from the task queue, which is implemented using Celery[7], using RabbitMQ[8] as the message broker (Q1). As a result of only coupling API and worker through the task representation, the workers can use Python[9] as programming language. This enables the use of the pandas[10] package for representation and manipulation of time series, and allows using FMPy[11] for simulating FMUs. FMPy was chosen because it can be used natively from within a Python environment; because it is actively maintained and developed under an open-source license; and because FMPy and its dependencies can be installed easily, also as part of a container image. Worker instances can be added or destroyed according to demand and jobs are automatically distributed across all worker instances that are available. Upon finishing a simulation job, the results are propagated back to component A using Redis[12] as the result backend (Q2).

API and worker are implemented according to the *twelve-factor app*[13]-method, which is the name of a set of best practices for developing, operating and maintaining Software as a Service (SaaS).

Each component is intended to be deployed as a container. Containers are a lightweight packaging format for an application *and* all of its dependencies. They are guaranteed to run on any host that runs a compatible container

[4] `https://nodejs.org`

[5] `https://expressjs.com`

[6] `https://github.com/anttiviljami/openapi-backend`

[7] `https://github.com/celery/celery`

[8] `https://www.rabbitmq.com`

[9] `https://www.python.org`

[10] `https://pandas.pydata.org`

[11] `https://github.com/CATIA-Systems/FMPy`

[12] `https://redis.io`

[13] `https://12factor.net`

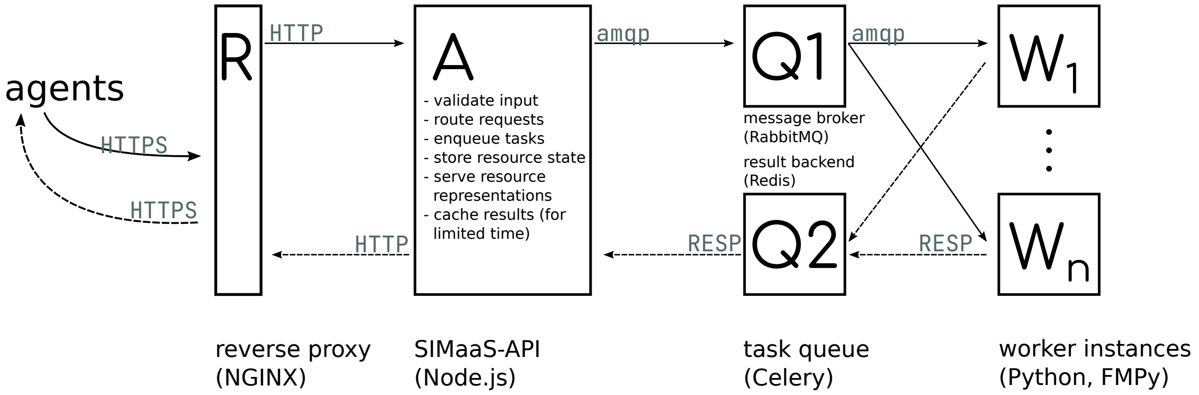

Figure 1. Software architecture of the SIMaaS-implementation.

engine, for example the Podman[14] engine. Containers are also the basic building blocks for deploying on clustered elastic platforms such as Kubernetes[15]. As such, using containers as the deployment unit for the components of the developed SIMaaS-implementation enables their use on such platforms, which in turn enables using advanced operation strategies such as automatic scaling in response to demand or load-balancing requests between several containers running the same component.

The source code for the SIMaaS-API and the workers are available subject to the conditions of the MIT license[16] at `https://github.com/UdSAES/simaas-api` and `https://github.com/UdSAES/simaas-worker`, respectively. For message broker and result backend, stock container images can be used, which are for example available on Docker Hub[17]. Consult the `README` documents in the API and worker-repositories for details.

3.2 Requirements on FMUs

FMI 2.0 for co-simulation can be seen as a way to export models *and* the corresponding solver in an open, widely supported way. In other words, a FMU can be seen as a standalone executable format of a single model using a single solver for simulation. Obviously, the capabilities and intended usage of FMI are more diverse; but for the purpose of this work, we adopt this limited view.

In order to facilitate the implementation of the envisioned software, we impose additional restrictions on the FMUs concerning their parameterization, the supported platforms for which binaries must exist, and the definition of inputs, outputs and parameters to be exposed via the API. Note that none of these restrictions impose limits on the actual models or their simulation; they merely represent a concretization of the format supported by the developed software.

Schmitt et al. (2015) investigated different possibilities to parameterize models in Dymola with respect to their subsequent export as FMU. In section 3.3 of their paper, they describe a method that "becomes favorable if the user wants to exchange whole data sets of one and the same model" (Schmitt et al. 2015, section 3.3), which is exactly the case for the SIMaaS-implementation. In short, parameters inside the model are set by inter-component references to a record. This record has a parameter `filename`, which must be set to the path of a file containing the values. All actual model parameters are set by reading this file during model initialization. This can, for example, be achieved using the `DataFiles` package distributed with Dymola or one of the functions provided in `Modelica.Utilities`.

The second requirement is that inputs and outputs of the models must be listed as such in the `modelDescription.xml` file of the FMU because the schemata for the trajectories that a service user needs to supply/can expect as a result, which are part of the OAS, are derived from this information.

Last, the FMU must contain binaries for GNU/Linux as the containers are intended to be deployed on GNU/Linux host systems.

3.3 FMI Calling Sequence

In FMI 2.0 for co-simulation, direct feedthrough is forbidden[18]. FMPy ensures this by calling the FMI functions in the order `fmi2GetXXX()`, `fmi2SetXXX()`, `fmi2DoStep()` in the `simulateCS()`-function[19].

In some cases, this leads to significant differences between the simulation results of a model simulated natively (for example in Dymola) and the simulation of the corresponding FMU. As an example, consider the simulation of a model that calculates the power generated by a photovoltaic (PV) module as a function of ambient conditions (irradiance in the horizontal plane, temperature, wind speed) and the orientation of the PV module. For this calculation, the irradiance in the horizontal plane has

[14] https://podman.io

[15] https://kubernetes.io

[16] https://spdx.org/licenses/MIT.html

[17] https://hub.docker.com

[18] Compare https://github.com/CATIA-Systems/FMPy/issues/89#issuecomment-522949757

[19] https://github.com/CATIA-Systems/FMPy/blob/69ec43813e6d5f8eb79da0d17c181fe57271f8ac/fmpy/simulation.py#L1171

to be converted to the plane of array (POA), which requires the sun's position relative to the module. At 07:00 a.m., the sun's position at 07:00 a.m. is calculated because the calculation is part of the model. At this time instant, inside the FMU, the irradiance data available is the data for 06:45 a.m. (assuming an output interval of 15 minutes) because of the calling sequence chosen by FMPy. When using Dymola to natively simulate the Modelica model, the irradiance data for 07:00 a.m. is used as intended.

For the special case of using FMUs as a portable export format of single models that are ready to be simulated using a single solver contained in the FMU, this *unnecessarily* introduces systematic errors. Therefore, we use a *fork* of FMPy which calls `fmi2SetXXX()`, `fmi2DoStep()` and `fmi2GetXXX()` in this order for the implementation of SIMaaS-workers.

4 Demonstration

The ensemble forecast for the power produced by a PV system and the search for an optimal parameter set by means of a genetic algorithm (GA) serve as examples for demonstrating the use of the SIMaaS-implementation.

The code that executes the necessary requests is written in Python in a concurrent fashion using Python's `async/await` mechanism and an asynchronous HTTP library[20]. Request sequences such as the sequence for triggering and retrieving the results of a specific simulation (`POST`ing a new simulation resource, polling its status using repeated `GET` requests, `GET`ting the result) are obviously still executed in order for each individual simulation, but they are executed in parallel for several different simulations, thereby testing/showing the ability of the SIMaaS-API to handle many requests at once.

Like the API and worker implementations, the demonstration code is available under the MIT license on GitHub: `https://github.com/UdSAES/simaas-demo`. Please refer to the `README` and the code itself for details.

4.1 Ensemble Forecast for PV Systems

A single trajectory of values such as those obtained as the result of simulating a model implemented in Modelica implies a level of exactness that does not fairly reflect the uncertainties inherent to modelling process, parameterization, and simulation.

One way to better understand and/or communicate the actual meaning of a simulation is to perform a number of simulation runs with slight variations in parameterization, input trajectories and/or initial conditions as an ensemble forecast. Ensemble forecasts created by varying the input trajectories of each simulation run are representative of situations where multiple simulations of the same model instance are required.

Here, we use the repeated simulation of a model of a photovoltaic (PV) system with the members of an ensemble weather forecast as input trajectories as an example. The PV system model is exported from the `pv-systems` library (Stüber 2020) according to the requirements on the FMUs given in subsection 3.2. The FMU is then added to the SIMaaS-instance and the request bodies to be sent for triggering the individual simulation runs are prepared by collecting the different weather forecasts. Once they are ready, all request sequences are started in parallel. Depending on the number of workers that are started, the simulations are either carried out in sequence (exactly one worker) or in parallel (more than one worker).

Admittedly, executing one ensemble forecast consisting of nine individual forecasts based on a computationally lightweight model does not require the computing resources of several nodes in the cloud. However, the advantage of using the SIMaaS-API instead of executing the FMU locally quickly becomes visible when considering that realistic real-world users for such ensemble forecasts would be utility companies responsible for stabilizing a section of the electricity grid. In this scenario, ensemble forecasts for *all* renewable energy sources in the relevant grid section would be required as part of flexibility management processes, likely to be re-calculated several times per day.

4.2 Component Selection Using a Genetic Algorithm

The second example for demonstration purposes is the search for an optimal set of values for the components of a temperature-dependent electrical circuit as shown in Figure 2, given the desired voltage at `p2` over the temperature range from $-10\,°C$ to $60\,°C$.

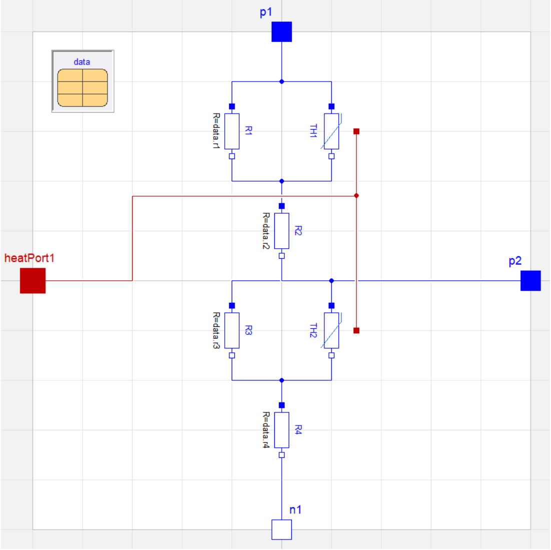

Figure 2. Thermistor network.

Suppose the resistors can each take one of the 70 values of the E24-series between 300 Ω and 220 kΩ and suppose there are nine possible values for both the resistance at reference temperature and the temperature coefficient `B` of the two thermistors. Then, there are $70^4 * 9^2 * 9^2 =$

[20] `https://github.com/aio-libs/aiohttp`

157 529 610 000 different solutions, each resulting in a different voltage over temperature-curve. A rough estimate, assuming a Central Processing Unit (CPU) time of 0.05 s per simulation, puts the total time for testing *every* permutation at around 250 years.

One possible alternative, suggested in an article on edn.com (EDN 2008), is to use a genetic algorithm (GA)[21] to search for a good solution without trying every permutation. Each possible combination of component values is seen as an individual. The fitness of an individual is evaluated by how close the voltage at p2 matches the desired voltage over the relevant temperature range, for example in terms of the Root Mean Square Error (RMSE). Because the determination of the fitness of an individual is independent of other individuals, the fitness values for an entire generation can be evaluated in parallel. After the fitness of each individual in a population is determined, the best results are recorded and the next generation is created by cross-over and mutation (subject to user-defined probabilities). The algorithm is terminated by setting a threshold for either an acceptable fitness value or a fixed number of generations.

This example represents a situation where different model instances are simulated with the same input. For finding good solutions, a few hundred simulations are likely required, many of which can be executed in parallel given a sufficiently high number of worker instances. Using a SIMaaS-instance deployed on a clustered elastic platform instead of running the GA locally becomes really beneficial iff the number of individuals in a generation is higher than the number of CPUs available locally.

Because the example is merely intended to serve as a proof of concept, no detailed analysis of the performance was carried out – neither with respect to the speed-up achieved, nor with respect to the overhead introduced by the additional software layers and the exchange of data over the network.

For implementation of the GA, the Distributed Evolutionary Algorithms in Python (DEAP) framework (Fortin et al. 2012) was used. A tiny Modelica package containing the necessary models and an example ready to be simulated in Dymola is included in the simaas-demo-repository.

5 Discussion

While the use cases described in the previous section illustrate that there are scenarios for which the developed software works and has benefits over other approaches, there are some conceptual limitations inherent to its design that should be discussed. Before summarizing these issues, we outline related work within the Modelica community.

5.1 Related Work

The concept of MSaaS has been discussed extensively in the literature. We refer the interested reader to key publications (Cayirci 2013; MSG-131 2015; Hannay, Berg, et al. 2020; Shahin, Babar, and Chauhan 2020; Kratzke and Siegfried 2020) and focus this section on previous work within the Modelica community.

Tiller (2014) motivates the use of web technologies for the design of engineering tools in general, detailing potential benefits for non-expert users. The FMQ platform and its HTML5-based interface for human users are outlined; the use of a hypermedia API as the backend of the FMQ platform is hinted at, but no details are given. The FMQ platform also uses FMUs as an executable form of a single model to be simulated using a single solver. It is a proprietary product of Xogeny, Inc.

In their 2017 presentation of the modelica.university platform, Tiller and Winkler motivate the high-level requirements for and architecture of the Aperion platform by Xogeny, Inc (presumably the successor of the FMQ platform) in addition to presenting the modelica.university website itself. With regard to concepts and the chosen technology stack, the Aperion platform seems to be very similar to the work presented in this paper, but it is a commercial product and specifics are consequently not available publicly. With regard to the use of truly RESTful technologies such as hypermedia representations and generic software clients that use them, Aperion seems to already have realized some of our plans for further work as outlined in subsection 5.2.

Bittner, Oelsner, and Neidhold (2015) outline work on a web application based on FMI 1.0 for co-simulation. Compared at a high level, the application's architecture is similar to what is presented in section 3 as there is also at least a conceptual separation between API, storage, simulation components and front-end. The provided user interface (UI) directly supports ranges for setting parameter values and seems to be intended for use by engineers. Details or source code are not readily available.

FMIGo![22] is a set of software tools for executing several coupled FMUs over the internet. It is described in a paper by Lacoursière and Härdin (2017) and available[23] under the MIT license. In contrast to the concepts of the SIMaaS-implementation explained in section 2 and the design concepts of the FMQ platform, FMIGo! choses not to make use of the design principles of the web (REST) and instead expose the capabilites of FMI withouth further abstraction through the use of (low-level) message passing protocols. Lacking the simplification of seeing an FMU as nothing but an executable form of one model with one solver, FMIGo! allows/demands chosing master algorithms for co-simulation and exposes necessary numerical details; but this clearly makes it a specialized tool for simulation experts.

Elmqvist, Malmheden, and Andreasson (2019) present the Web Architecture for Modeling and Simulation (WAMS) in terms of several use cases, the correspond-

[21] See Pelikan (2011) for an excellent introduction.

[22] https://www.fmigo.net

[23] https://mimmi.math.umu.se/cosimulation/fmigo

ing web application aimed at engineers with training in M&S and a very brief overview of its software architecture. FMUs are used for simulation using PyFMI and it is claimed that a REST API is used for exposing capabilities of the server, but the extent to which the REST constraints are realized remains unclear. It appears that WAMS is an internal project of Modelon AB.

Since version 0.2.25 (released in November 2020), FMPy features the possibility to expose the UI of FMPy including the ability to simulate FMUs as a web application. Consequently, this approach falls in line with the WAMS web application, also intended to be used by engineers, and thus caters to use cases different to those that motivate the work presented in this paper.

5.2 Results, Limitations and Outlook

The SIMaaS-implementation presented in section 3 has several desirable characteristics. First, its design reflects best practice and the state of the art for creating SaaS as identified by Kratzke and Siegfried (2020). Second, the decision to decouple API and workers, only linking them by the internal representation of tasks in the task queue, means that the implementation of how models are simulated can be changed without having to change the public service interface. This includes adding support for FMUs for model exchange or for non-FMI-based models as long as they can be expressed in terms of the resources exposed by the API (models, model instances, simulations, simulation results). For example, a user might have legacy models written in Matlab or scientific computing routines written in a general-purpose programming language which cannot be exported as a FMU. Third, the chosen software stack consists of proven and widely used open-source components.

By deploying it on Kubernetes as an example of a clustered elastic platform, horizontal scalability of the workers was shown. Graceful handling of the addition or removal of worker instances is ensured by using Celery as the task queue implementation. Implementing automatic scaling of worker instances in response to demand is only a question of properly configuring Kubernetes resources.

From our point of view, the additional restrictions posed on the FMUs to be supported by the software (subsection 3.2) are reasonable.

Using FMUs for co-simulation according to version 2.0 of the FMI standard allows using the proven solvers provided by native M&S-environments such as Dymola, but the workaround for avoiding systematic errors described in subsection 3.3 is problematic because it is not compliant with the FMI standard and because it requires maintenance work to synchronize the fork of FMPy with upstream development. Therefore, support for FMUs for model exchange and/or the upcoming FMI 3.0 standard should replace the workaround in the future.

As the presented software is predominantly research software and not a commercial product, some features that are expected of the latter are not implemented yet. For example, there is currently no access control (any user that can reach the API can send all requests); but it would be straightforward both from a conceptual and practical persepective to add this, most likely using JSON Web Tokens (JWTs). The pay-as-you-go cost model is not currently supported, but it represents mostly an organizational and operational aspect that most likely should be implemented using sidecar containers that form a service mesh (Morgan 2019) and not as part of the application code, anyway.

There exists neither an explicit threat model nor a subsequent detailed consideration of security aspects. However, basic measures against misuse were implemented. First, all requests by users are validated against the schemata defined in the OAS. These schemata are as specific as possible, including the use of regular expressions for string fields. Second, the process inside a container is run as a non-privileged user to prevent easy privilege escalation.

In contrast to the aforementioned aspects, which are of practical nature, there also exist *conceptual* limitations that keep the SIMaaS-implementation from reaching its full potential. The first limitation is that clients are required to poll resources in order to find out about changes of their state. This leads to potentially many requests that would not have been necessary and raises the question of how to set polling frequencies and eventual timeouts. Moreover, the answer to this question depends on both the current server load and the amount of workers available which cannot be known a priori. It would be possible to instruct the service to notify the consumer as soon as the resource state changes, but this would mean that the consumer has to become its own server for accepting such notifications.

The second (and more important) limitation concerns the required use of a static service interface description against which clients hardcode requests, meaning that client code will break in case the service interface changes. Dynamically re-generating the service interface description to account for the different inputs, outputs and parameters of models can, from an academic perspective, only be seen as a workaround because a static service interface description should not be necessary in the first place. Instead, clients should construct their requests at run time, based on information present in the representation received (starting at the root path of the service). This requires an alternative format for resource representation, for example the JSON-based Serialization for Linked Data (JSON-LD), because JSON lacks support for natively encoding hyperlinks (Kellogg, Champin, and Longley 2020, section 3) as well as the ability to explicitly encode the semantics of data.

Ideally, software clients should be enabled to reason about the options for advancing the application state at each step of the interaction in order to make informed decisions about which state transitions allow them to reach their goal (a desired resource state) by following links; just like humans navigate websites. The technical feasibility of this has been shown by Verborgh, Arndt, et al. (2017). Realizing such services opens exciting opportunities for building generic clients that can achieve a class of similar, yet distinct objectives while being robust against changes of the service interface as requests are assembled at run time and not constructed at design time.

We are actively working towards realizing this vision for the presented SIMaaS-implementation.

6 Conclusion

Previous work within the Modelica community to provide M&S capabilities *as a service* can be categorized in three different directions of work: providing model development and/or simulation environments as web applications in the browser for use by engineers; providing tools for distributed co-simulation; and transferring M&S functionality to the web by translating them to the architectural style REST, hoping to make use of the positive aspects of the web's design principles and cloud computing technology for M&S tooling as well as facilitating the use of M&S by software agents in a distributed setting.

The SIMaaS-implementation presented in this paper falls into the last category. It exposes models in the form of FMUs for co-simulation and the ability to simulate them as a REST-based HTTP-API that consists of an API and several worker components which exchange data using queues. Worker instances can be scaled horizontally when deployed on an clustered elastic platform such as Kubernetes.

The design concepts, software architecture and software stack are summarized and put into context by outlining related work and the achieved characteristics. Two exemplary use cases are shown. Development continues with the aim to fully support the HATEOAS principle by adding a format for resource representations that enables client-side reasoning, and to demonstrate the abilities gained by doing so.

As of this moment, the presented SIMaaS-implementation represents a functional building block for SOAs, intended to be used in combination with other services. The software is available under a permissive open-source license, readily available for testing.

Acknowledgements

This work was supported by the SINTEG-project "Designetz" funded by the German Federal Ministry of Economic Affairs and Energy (BMWi) under grant 03SIN224.

The authors would like to thank Lukas Exel and Florian Wagner for sharing their views on and experience with providing modelling and simulation *as a service*, as well as for their work on early prototypes of the presented software.

References

Bittner, Stefan, Olaf Oelsner, and Thomas Neidhold (2015-09-18). "Using FMI in a cloud-based Web Application for System Simulation". In: *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*. Linköping University Electronic Press. DOI: 10.3384/ecp15118845.

Cayirci, Erdal (2013-12). "Modeling and simulation as a cloud service: A survey". In: *2013 Winter Simulations Conference (WSC)*. IEEE. DOI: 10.1109/wsc.2013.6721436.

EDN, ed. (2008-03-19). *Genetic algorithm solves thermistor-network component values*. URL: https://www.edn.com/genetic-algorithm-solves-thermistor-network-component-values (visited on 2021-05-08).

Elmqvist, Hilding, Martin Malmheden, and Johan Andreasson (2019-02-21). "A Web Architecture for Modeling and Simulation". In: *Proceedings of the 2nd Japanese Modelica Conference Tokyo, Japan, May 17-18, 2018*. Linköping University Electronic Press. DOI: 10.3384/ecp18148255.

Fortin, Félix-Antoine et al. (2012-07). "DEAP: Evolutionary Algorithms Made Easy". In: *Journal of Machine Learning Research* 13.70, pp. 2171–2175. URL: http://jmlr.org/papers/v13/fortin12a.html.

Hannay, Jo Erskine and Tom van den Berg (2017-10). "The NATO MSG-136 Reference Architecture for M&S as a Service". In: *Proceedings of the NATO modelling and simulation group symposium on M&S technologies and standards for enabling alliance interoperability and pervasive M&S Applications* (Lisbon, Portugal, October 19–20, 2017). STO-MP-MSG-149, p. 3.

Hannay, Jo Erskine, Tom van den Berg, et al. (2020). "Modeling and Simulation as a Service infrastructure capabilities for discovery, composition and execution of simulation services". In: *The Journal of Defense Modeling and Simulation. Applications, Methodology, Technology*. DOI: 10.1177/1548512919896855.

Kellogg, Gregg, Pierre-Antoine Champin, and Dave Longley (2020-07). *JSON-LD 1.1*. W3C Recommendation. https://www.w3.org/TR/2020/REC-json-ld11-20200716/. W3C.

Kratzke, Nane and Peter-Christian Quint (2017). "Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study". In: *Journal of Systems and Software* 126, pp. 1–16. ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.01.001. URL: http://www.sciencedirect.com/science/article/pii/S0164121217300018.

Kratzke, Nane and Robert Siegfried (2020). "Towards cloud-native simulations – lessons learned from the front-line of cloud computing". In: *The Journal of Defense Modeling and Simulation. Applications, Methodology, Technology*. DOI: 10.1177/1548512919895327.

Lacoursière, Claude and Tomas Härdin (2017-07-04). "FMI Go! A simulation runtime environment with a client server architecture over multiple protocols". In: *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*. Linköping University Electronic Press. DOI: 10.3384/ecp17132653.

Mell, Peter and Timothy Grance (2011). *The NIST Definition of Cloud Computing*. NIST Special Publication 800-145. National Institute of Standards and Technology. DOI: 10.6028/NIST.SP.800-145. URL: https://www.nist.gov/publications/nist-definition-cloud-computing.

Modelica Association (2020-12). *Functional Mock-up Interface for Model Exchange and Co-Simulation Version 2.0.2*. Tech. rep. Linköping: Modelica Association. URL: https://fmi-standard.org.

Morgan, William (2019-11-09). *The Service Mesh: What Every Software Engineer Needs to Know about the World's Most Over-Hyped Technology*. URL: https://buoyant.io/service-mesh-manifesto (visited on 2021-05-07).

MSG-131, Specialist Team (2015). *Modelling and Simulation as a Service: New Concepts and Service-Oriented Architectures*. Final Report AC/323(MSG-131)TP/608. North Atlantic Treaty Organization NATO. DOI: 10.14339/STO-TR-MSG-131. URL: https://www.sto.nato.int/publications/STO%20Technical%20Reports/STO-TR-MSG-131/$$TR-MSG-131-ALL.pdf.

OASIS (2006). *Reference Model for Service Oriented Architecture 1.0*. Tech. rep. URL: http://docs.oasis-open.org/soa-rm/v1.0/.

Pelikan, Martin (2011). "Genetic Algorithms". In: *Wiley Encyclopedia of Operations Research and Management Science*. American Cancer Society. ISBN: 9780470400531. DOI: 10.1002/9780470400531.eorms0357. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470400531.eorms0357. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470400531.eorms0357.

Schmitt, Thomas et al. (2015-09-18). "A Novel Proposal on how to Parameterize Models in Dymola Utilizing External Files under Consideration of a Subsequent Model Export using the Functional Mock-Up Interface". In: *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*. Linköping University Electronic Press. DOI: 10.3384/ecp1511823. URL: https://2015.international.conference.modelica.org/proceedings/html/errata/errata_SchmittAndresZieglerDiehl.pdf. Revised version.

Shahin, Mojtaba, M. Ali Babar, and Muhammad Aufeef Chauhan (2020-07-24). "Architectural Design Space for Modelling and Simulation as a Service: A Review". In: *Journal of Systems and Software* 170, p. 110752. ISSN: 0164-1212. DOI: 10.1016/j.jss.2020.110752. URL: http://www.sciencedirect.com/science/article/pii/S0164121220301746.

Stüber, Moritz (2020-12-24). *UdSAES/pv-systems: v0.9.0*. Version 0.9.0. DOI: 10.5281/zenodo.4392849. URL: https://github.com/UdSAES/pv-systems.

Stüber, Moritz, Lukas Exel, and Georg Frey (2018). "Using Modelling and Simulation as a Service (MSaaS) for Facilitating Flexibility-based Optimal Operation of Distribution Grids". In: *Proceedings of the 15th International Conference on Informatics in Control, Automation and Robotics - Volume 2: ICINCO*. INSTICC. SciTePress, pp. 613–620. ISBN: 978-989-758-321-6. DOI: 10.5220/0006899106230630.

Tiller, Michael (2014). "Vehicle Thermal Management – A Case Study in Web-Based Engineering Analysis". In: *Proceedings of the 10th International Modelica Conference; March 10-12; 2014; Lund; Sweden*. 96. Linköping University Electronic Press, pp. 1073–1079. DOI: 10.3384/ecp140961073.

Tiller, Michael and Dietmar Winkler (2017-07-04). "modelica.university: A Platform for Interactive Modelica Content". In: *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*. Linköping University Electronic Press, pp. 725–734. DOI: 10.3384/ecp17132725.

Verborgh, Ruben, Dörthe Arndt, et al. (2017-01). "The Pragmatic Proof: Hypermedia API Composition and Execution". In: *Theory and Practice of Logic Programming* 17.1, pp. 1–48. DOI: 10.1017/S1471068416000016. URL: http://arxiv.org/pdf/1512.07780v1.pdf.

Verborgh, Ruben, Seth van Hooland, et al. (2015). "The fallacy of the multi-API culture: Conceptual and practical benefits of Representational State Transfer (REST)". In: *Journal of Documentation* 71.2, pp. 233–252. DOI: 10.1108/JD-07-2013-0098.