

A Software Framework to Enable Model-Based Systems Engineering for Accelerator Magnets

Stephan Russenschuck¹, Matthias Bonora¹, Konstantinos Diamantis¹, Melvin Liebsch¹, and Jens Kaeske²,

¹European Organization for Nuclear Research (CERN), 1217 Meyrin, Switzerland

²Institute of Product Engineering (IPEK) at Karlsruhe Institute for Technology (KIT), 76131 Karlsruhe, Germany

Model-based systems engineering (MBSE) is a design methodology for complex systems via models and their interactions, shifting the focus from documents and reports to model implementations and their queries. We present a software framework applying MBSE methodology in the context of designing and optimizing accelerator magnets. We cover model implementation, interaction, dependency resolution, storage, and exchange. In this context, we encapsulate the necessary software to run each model in containers. This provides flexibility in combining different simulation codes while keeping track of dependencies on specific snapshots of the software used. By combining these components, we can fully utilize the MBSE design methodology, react to design changes, and guarantee the future execution of models in systems-generation engineering.

Index Terms—MBSE, Product life-cycle management, Superconducting magnets, Systems modeling

I. INTRODUCTION

MAGNET design is a complex engineering problem in multiple domains (thermal, electric, mechanic, and magnetic) and over long periods. Each magnet project undergoes multiple design iterations, from coil-winding tests, producing short models and prototypes, to series production and commissioning. It is important to support magnet designers during the whole process, provide design tools for each step, and document the design process across the full life cycle of the magnet. In recent years, we have found the adaptation of MBSE [1] and product life-cycle management [2] as a useful approach in retaining knowledge of the design process across product generations. Because MBSE shifts towards models and data instead of documents and reports, a new set of software tools is required to adapt to the methodology and benefit from its concepts.

A first approach to implementing MBSE for magnet design is described in [3]. It focuses on Jupyter notebooks for model executions and report generation, and on GitLab for execution. This works well for smaller projects but is difficult to scale. In addition, it is challenging to separate software codes from model notebooks and execution environments for different models.

We build on this concept and propose solutions to these issues by further generalizing the ideas, relying less on GitLab and Jupyter mechanics and more on customized interfaces.

II. THE SOFTWARE STRUCTURE

The primary goal of the software framework is to decouple simulation codes and applications from the execution and storage of models. We define three layers of integration: model execution, model interaction, and model storage (figure 1). The execution layer handles the integration of simulation software within a container, the parametrization of a simulation, the retrieval and storage of simulation artifacts, and the querying of information about a model. The model-interaction layer handles dependencies between models, the execution order

of multiple models, and the orchestration of the execution layer. Both layers act *locally*, triggered and run by one design engineer. For handling larger collaborations and guaranteeing traceability, the model-storage layer provides routines to publish and retrieve models from a database. This database is accessible to multiple design engineers and thus provides a means of sharing and collaborating on systems and models.

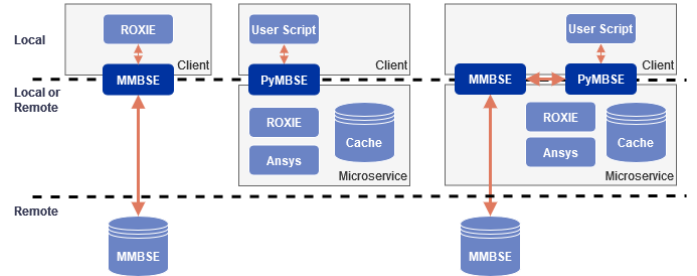


Fig. 1. The software framework applied to three levels of integration.

The goal of this structure is to provide the necessary means of interfacing between models while giving design engineers the freedom to use their preferred simulation codes and tools. Therefore, we aim to minimize the specificity of tools and simulation codes and keep interfaces generic. Each layer is implemented in Python and can be integrated into Python scripts or Jupyter notebooks, providing flexibility to its large user base [4]. New tools and simulation codes can be integrated into the structure with minimal overhead using templates for new applications.

A. Model Interaction

The model interaction layer describes models of a system and their connectivity. A simulation set can consist of different systems, each with multiple models that define their inputs and outputs. Inputs are files, sets of parameters, or references to other models. Model outputs, i.e., the results of a model execution, are associated with data at execution time. In addition,

models can define dependencies to other models, which can be used for model-in-model execution, such as an optimizer calling models with updated parameters.

Each model is associated with model executions. A model execution represents an instance of a model, frozen with a given set of inputs and associated with outputs generated from this model. Executions keep track of their inputs and outputs in a local cache. Future executions can use the cache to retrieve inputs from other model executions or skip simulations if inputs do not change. For example, a coil head and its local force distribution might not change, while the material characteristic of the iron yoke must be updated.

The interaction layer defines the interface to query a simulation's outputs, either by retrieving simulation as artifacts (data files generated by the simulation), figures of merit (parameter dictionaries stored by the simulation), or queries (more complicated postprocessing routines defined on a specific model or simulation type). It is implemented in a client part in Python and a microservice providing a REST API for interaction.

B. Model Execution

The model-execution layer handles the execution environment for a simulation code. It loads all inputs and parameters and stores results. In addition, it distributes each model execution to the appropriate software environment. Each software environment registers a set of workers, and the execution layer distributes executions on each worker node.

Different simulation codes require different setups and versions of packages, while codes and operating systems change over time. It is necessary to keep a snapshot of any given execution environment linked to a model. We achieve this by utilizing docker containers for execution environments [5]. Each container exposes one or more versions of a specific execution environment. For example, a generic Jupyter Notebook execution provides different versions of the environment, each with a custom set of packages and versions installed.

We separate the functionality of the model execution layer from tool implementations by providing templates for setting up applications. The templates provide callback hooks in which a folder with all input-file dependencies is prepared, and all input parameters are forwarded to the tool. From there, the tool can be executed, and a list of outputs to be stored is returned. The actual storage and transfer to the cache are handled internally by the execution layer.

C. Persistency

The persistency layer ensures that models, dependencies, and associated artifacts are stored permanently so they can be retrieved for future reference and design iterations. In addition, it enables collaboration on a design by providing access control and a history of model versions. Each model version stores associated authors to keep track of ownership.

Each model iteration is associated with its metadata, parameters and generated outputs. A generated report summarizes and visualizes the outputs, providing a consistent synopsis of the results.

Users can retrieve and query model iterations and their results from a web-based user interface (UI) for easy access or using the framework's client API for automated pipelines and scripted data processing.

III. USING THE SOFTWARE

Putting all layers together results in a complex system of microservices, databases, and application containers. However, the interfaces between each layer are clearly defined, and no global environment to run each component is imposed. For local use, a *docker-compose* describes all environments as multi-container applications and resolves dependencies to run a system locally. This compose file, in addition to local input files (inputs to simulations or links to models stored on a persistent model storage), describes the execution flow and is easily shared and synchronized between collaborators.

As microservices, execution environments can also be set up on a computing cluster, and provided to multiple users, reducing computing requirements on individual machines.

On the client side, a Python package handles the management of systems and models, file uploads, and triggers executions. A minimal installation would only require this package to schedule model executions and retrieve results.

As model dependencies are tracked for each model step of a system, all results can be extended with a complete list of authors they depend on, facilitating a comprehensive ownership tracking of models and their data on each design step.

IV. CONCLUSION

We present a software framework for distributed execution of simulation codes based on the MBSE methodology. Designers can focus on their simulation codes and model descriptions with minimal overhead for the integration into the framework.

The chosen structure fully separates the concerns of model interactions and model execution from the tools themselves. This simplifies the knowledge required to add new tools to the framework.

The full paper will contain an example of two generations of a superconducting accelerator magnet with different coil and yoke designs and their pipelines of simulations to derive manufacturing tolerances, bending invariances in the coil heads, dynamic effects in the superconductor and iron yoke, and quench propagation.

REFERENCES

- [1] S. Russenschuck, *Field Simulation for Accelerator Magnets, Volume 2*. Wiley, 2025.
- [2] J. Kaeske, E. Wagner, A. Albers, and S. Russenschuck, "Tool support for implementing a methodology in magnet development projects at cern," *Proceedings of the Design Society*, vol. 4, p. 2615 – 2624, 2024.
- [3] M. Maciejewski, B. Auchmann, D. M. Araujo, G. Vallone, J. Leuthold, and J. Smajic, "Model-based system engineering framework for superconducting accelerator magnet design," *IEEE Transactions on Applied Superconductivity*, vol. 33, no. 5, pp. 1–5, 2023.
- [4] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. development team, "Jupyter notebooks ? a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (F. Loizides and B. Schmidt, eds.), pp. 87–90, IOS Press, 2016.
- [5] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, Mar. 2014.