An Integrated Optimization and Orchestration Toolchain for Adaptive Optimal Control in Modelica Simulations

Zizhe Wang[1,2]

[1]Boysen-TU Dresden-Research Training Group, Dresden, Germany
[2]Software Technology Group, Technische Universität Dresden, Germany
zizhe.wang@tu-dresden.de

Abstract

This paper introduces a novel Python-based toolchain, "OptiOrch", designed to enhance optimal control in Modelica-based simulations by integrating an optimization framework and an orchestration workflow. OptiOrch leverages the "MOO4Modelica" optimization framework, which supports both single- and multi-objective parameter optimization, and incorporates the "ModelicaOrch" orchestration workflow to dynamically adapt models based on real-time input data and goals. The toolchain features a user-friendly interface, feature model transformation, parallel computing, and automated workflow coordination, making it a powerful and generalized solution for various applications. Practical examples and a case study demonstrate how this toolchain can be effectively applied to Modelica systems for optimal control.

Keywords: Modelica, simulation, optimization, multi-objective optimization, parallel computing, self-adaptive systems, optimal control, feature model

1 Introduction

Digital twins are becoming increasingly important in research and development. These virtual replicas of physical systems allow for real-time monitoring and optimization. The Modelica language, proposed by Fritzson and Engelson 1998, has emerged as the leading equation-based modeling language for multi-domain, multi-physical systems. It has been widely used in industry and academia to build digital twins of complex systems, such as the modeling and simulation of integrated energy systems (Senkel et al. 2021). Modelica-based software, both open-source and commercial, offers user-friendly graphical interfaces and advanced debugging capabilities. Notable examples include OpenModelica (Fritzson, Aronsson, et al. 2005) (Fritzson, Pop, et al. 2022) (Modelica Association 2023) as an open-source option, and commercial environments like Dymola (Elmqvist 1979) (Brück et al. 2002), Modelon Impact, SimulationX, MWorks (Chen and Wei 2008). Additionally, various open-source and commercial Modelica libraries are available in the community, with the Modelica Standard Library (current release v4.0.0 as of July 2024) serving as the foundational library for all Modelica environments.

Modern multi-domain, multi-physical systems are complex entities comprising diverse components, highlighting the importance of advanced modeling languages like Modelica in their development and optimization. Single-objective optimization is often inadequate for these complex products, necessitating multi-objective optimization (MOO) to address various competing objectives. For instance, in the field of renewable energy, optimizing a wind farm might involve balancing energy output and the minimization of environmental impact (Thirunavukkarasu, Sawle, and Lala 2023). However, the current Modelica ecosystem lacks robust support for MOO, particularly in terms of a generalized open-source framework. This work addresses this gap by proposing a comprehensive open-source MOO framework that leverages Modelica and the Python ecosystem using the OMPython API (Ganeson et al. 2012). As systems become more complex, modeling them also becomes more challenging, especially since many systems need to self-adapt based on different contexts/conditions and performance targets. For example, in cloud computing systems, hardware components like CPU cores and frequencies need to be adjusted based on user demands and specific tasks to achieve the optimal energy-performance balance. Modelica, as a powerful modeling language, is particularly useful for optimizing self-adaptive systems, especially in achieving optimal control.

2 Background

2.1 Optimization and MOO in Modelica

According to Sharma and Kumar 2022, optimization techniques can be categorized into three types: exact (classical) methods, heuristic and meta-heuristic methods, and hybrid methods combining elements of both. Exact methods aim to find optimal solutions within a small, manageable solution space but are often impractical for complex real-world applications. Heuristic and meta-heuristic methods, while not guaranteeing optimal solutions, are more feasible and effective for such scenarios. Hybrid methods leverage the strengths of both exact and heuristic approaches to mitigate their weaknesses. MOO techniques, often classified as stochastic meta-heuristic methods, are divided into three classes: evolutionary, swarm-based, and hybrid algorithms.

Different Modelica environments provide support for single-objective optimization tasks. The current state of MOO in Modelica involves various tools. In the commercial sphere, software like Dymola and Modelon Impact offers MOO support. Dymola includes a comprehensive optimization library (Pfeiffer 2012) (current version v2.2.6 as of July 2024) focusing on general optimization algorithms, including the weighted sum method, which converts a MOO problem into a single-objective problem by assigning weights to each objective. Although Dymola supports MOO, it may lack the specialized algorithms of dedicated optimization tools. Therefore, frameworks like the one by Leimeister 2019 have been designed for Dymola. Modelon Impact provides a cloud-based platform with robust optimization features, enabling users to effectively handle complex multi-objective problems in various engineering and industrial applications. OpenModelica, a prominent open-source Modelica environment, integrates with external optimization libraries and tools to facilitate MOO. However, its specific tool, OMOptim (Thieriot et al. 2011), primarily designed for single-objective optimization, has been excluded from the OpenModelica software and is not currently maintained or further developed. Consequently, developers often create custom methods to integrate Python-based libraries tailored to their specific optimization needs.

There is a critical need for a universal, open-source optimization framework capable of addressing both single-objective and multi-objective optimization tasks. Therefore, a primary objective of this work is to tackle this challenge by developing a comprehensive, open-source optimization framework that robustly supports both single-objective and multi-objective optimization scenarios.

2.2 Optimal Control of Self-adaptive Systems

In real-world, many systems are self-adaptive. For example, a cellphone reduces hardware functionality to conserve power when its battery is low. Similarly, energy systems adjust operations based on user demand, and traffic light systems adapt themselves according to traffic flow. These scenarios require dynamic simulations that can update configurations and parameters based on the real-time conditions and goals. Currently, Modelica environments require developers to manually write scripts for continuous optimization and adaptation. Implementing an automated workflow that optimizes and updates simulations as needed would be significantly more practical and efficient. Such an automated workflow is crucial as it would enable systems to continuously monitor their state and environment, triggering updates to the simulation model with optimized parameters and configurations. This approach ensures that the simulation remains accurate and effective, thereby significantly enhancing the system's performance and reliability. Therefore, another primary objective is to design a robust, automated workflow for generalized orchestration, enabling optimal control for self-adaptive systems in the Modelica ecosystem.

3 The Optimization Framework

Figure 1 illustrates the concept and the structure of the MOO4Modelica optimization framework. Key components of this framework are feature model transformation and optimization operation.

3.1 Feature Model Transformation

This component can be used to transform the Modelica models into feature models. This allows the users to analyze and select parameters and variables that need to be varied and optimized, especially for large-scale models, this would be beneficial. It also allows developers to locate corresponding parameters and variables as well as to identify their relationships in the models quickly.

The method for transforming a Modelica model into a feature model is inspired by the approach described by Zhang et al. 2022. By parsing the selected Modelica model with ANTLR (Parr and Quong 1995), the parameters and variables, along with their types and values, as well as equation sets, can be progressively and recursively obtained. These will then form a feature model, which is saved as a `JSON` file. Still, the users can choose the parameters and variables they want without transforming a Modelica model into a feature model. That is the reason why the two key components are decoupled.

`modelica.g4` This is a grammar file of ANTLR for the Modelica language[1]. It defines the syntax rules that ANTLR uses to generate a lexer and parser for Modelica. Lexer rules specify how to recognize the smallest units (tokens) e.g. keywords, identifiers, and operators. Parser rules define how these tokens are combined to form valid Modelica constructs like expressions, statements, and declarations, specifically in Modelica e.g. classes, components (parameters and variables), and equations. With the help of these rules, the parser generated by ANTLR can understand and process Modelica code.

`parse_model.py` This process parses a Modelica model to extract its components, including parameters and variables, along with their values. Utilizing the ANTLR-generated lexer and parser to construct a parse tree and traverse it to gather the relevant information. Pseudocode 1 illustrates the workflow. By systematically extracting these components, the framework enables users to efficiently identify and manipulate key model elements.

`feature_model.py` This process invokes the `parse_model` function to parse a Modelica model, extracting its components and equations and organizing them into a hierarchical feature model. It includes functionalities to display the feature model and save it to a `JSON` file, which facilitates interoperability with other tools and platforms, enhancing the flexibility and utility of the framework. Pseudocode 2 illustrates the workflow.

[1] https://github.com/antlr/grammars-v4/tree/master/modelica

DOI
10.3384/ecp21757

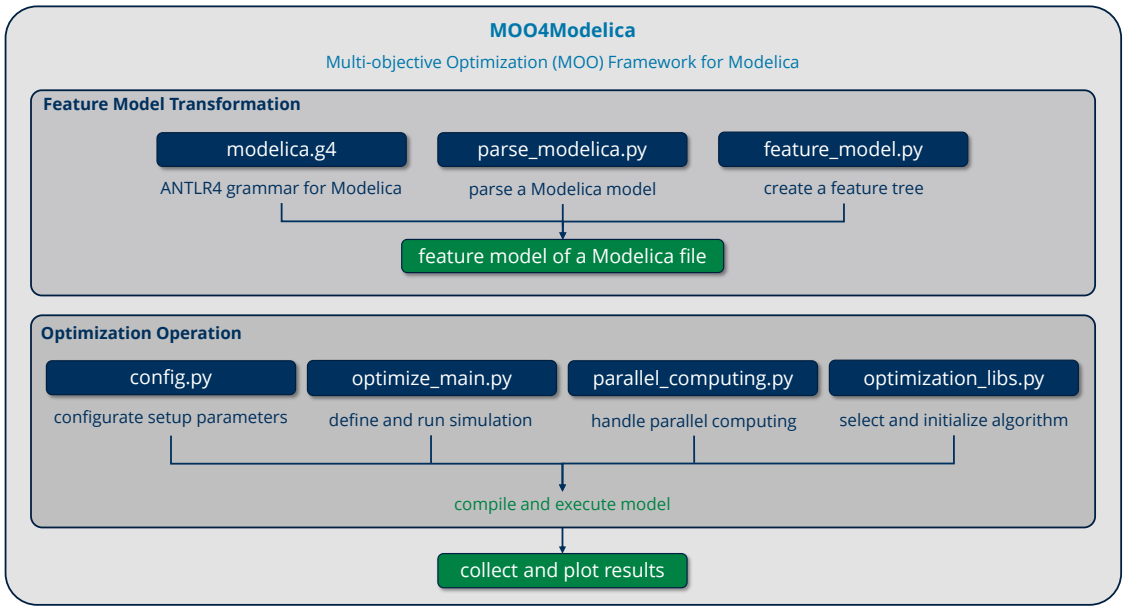

Figure 1. Structure of the framework *MOO4Modelica*.

Pseudocode 1 Modelica Parsing

```
1. Import Libraries and Setup
    Import antlr4 library and lexer/parser;
2. Define FeatureExtracton Class
    Create lists for components and equations;
    Define method to handle component clauses
        Extract component and add to list;
    Define helper method to extract the value
        Extract and add values to the list;
    Define method to handle equation sections
        Extract and add equations to the list;
3. Define parse_model function
    Read and parse Modelica file into parse tree;
    Initialize FeatureExtractor, traverse tree;
    Return components and equations;
4. Main execution block
    Call parse_model and print results;
```

Pseudocode 2 Feature Model Extraction

```
1. Import Libraries and Setup
2. Define FeatureModel Class
    Define method to add components/equations
        Create node for component type and name;
        Add parameters as children nodes;
        Add equation as a node;
    Define method to convert to dictionary
        Convert tree to dictionary format;
3. Main Execution Block
    Call parse_model to get components;
    Initialize FeatureModel with model name;
    Add components and equations to feature model;
    Display and save feature model to JSON;
```

3.2 Optimization Operation

The second component is used for operating optimization tasks. This can be used for both single-objective and multi-objective optimization. This component uses OMPython (Ganeson et al. 2012) as the bridge to connect the simulation of Modelica models to Python. All the global settings have been abstracted into the `config.py`. The main workflow of the optimization operation is outlined in the following steps:

- Step 1: Basic settings
  - Set model name, path, and simulation time.
  - Import external library (if needed).
  - Configure plot diagram settings.
- Step 2: Selection of parameter(s) to be varied
  - Set the range and data type.
- Step 3: Selection of objectives to be optimized
  - Set precision (decimal places) of the results.
- Step 4: Optimization options
  - Select optimization type and algorithm.
  - Set population size and number of generations.
- Step 5: Parallel computing options
  - Enable or disable parallel computing.
  - Set the number of CPU cores to be used.

`optimize_main.py` This process sets up and executes the optimization using configured algorithms and parameters. It involves defining the optimization problem, initializing the algorithm, running the optimization, and subsequently printing and plotting the results. As the main driver for conducting and analyzing the optimization, it ensures a streamlined and efficient workflow. Pseudocode 3 illustrates the workflow.

Pseudocode 3 Optimization

1. Import Libraries and Configuration
2. Define `OptimizationProblem` Class
  Define superclass initializer
    Initialize with `PARAMETERS` need to be varied;
    Initialize with `RESULTS` need to be minimized;
    Initialize with `RESULTS` need to be maximized;
    Set bounds for the parameters;
    Call superclass initializer
  Implement `evaluate` method;
    Convert parameter values to list of dictionaries;
    Parallel processing for evaluation with `n_jobs`;
    Negate objectives that need to be maximized;
    Store results;
3. Initialize Algorithm
  Initialize algorithm based on configuration;
4. Run Optimization and Handle Clean Up
  Define problem instance;
  Cleanup temporary directories;
5. Collect, Print, and Plot Results
  Iterate through results;
  Negate back maximized objectives;
  Print each solution with formatted results;
  Create scatter plot with results;

`parallel_computing.py` This process enhances computational efficiency by facilitating parallel execution of simulations. It defines functions for running simulations with different parameter sets concurrently using the `joblib`[2] library, significantly speeding up data processing and model evaluations. This is essential for handling computationally intensive tasks by leveraging parallel processing capabilities. Pseudocode 4 shows the workflow.

`optimization_libraries.py` This module provides a unified interface for initializing and configuring various optimization algorithms for the optimization operation. By abstracting the complexity of setting up different optimization libraries and algorithms, it simplifies the process of switching between them and configuring their parameters. This module includes the powerful open-source framework *pymoo*, introduced by Blank and K. Deb 2020 which offers state-of-the-art algorithms and features for visualization and decision-making. In this script, users can easily extend its capabilities to meet their specific needs.

[2]https://joblib.readthedocs.io

Pseudocode 4 Parallel Computing

1. Import Libraries and Configuration
2. Initialize Variables
  Initialize `temp_dirs` for temporary directories;
3. Define `optimization_function`
  Create `temp_dir` for each worker;
  Attempt for each worker
    Create OpenModelica session `omc`;
    Copy, load and build model in `omc`;
    Set parameters and simulate model in `omc`;
    Retrieve and return simulation results;
    Shutdown `omc`;
4. Define `shutdown_omc`
  Quit and close `omc`;
  Print success or error message;
5. Define `cleanup_temp_dirs`
  Attempt to remove `temp_dir`;
  Print success message and break loop if successful;
  If `PermissionError` occurs, sleep for backoff;

3.3 Examples

The first example features a simple heating system modeled using Modelica. In this model, increasing the **heating power** will raise the room temperature more quickly, thereby enhancing **human comfort** compared to slower heating. However, this approach results in higher **energy consumption**. Additionally, setting the **target temperature** too high can also decrease **human comfort**. In this context, the key parameters to be adjusted are **heating power** and **target temperature**. The objective is to find the optimal settings that maximize **human comfort** while minimizing **energy consumption**. Table 1 summarizes this scenario, including the parameters to be adjusted, objectives, and the goal.

Parameters	**Objectives**
Heating Power	Human Comfort
Target Temperature	Energy Consumption
Goal	
Maximize Human Comfort	
Minimize Energy Consumption	

Table 1. Parameters, objectives, and goal-setting of example 1.

A feature tree is not required in this simple heating system. The configurations and parameters are directly set in the `config.py`. The default configuration has been used for this example, and the simulation time has been set to 3000 seconds. The ranges for **heating power** and **target temperature** are 1000 - 5000 Watts and 280 - 310 Kelvin, respectively. The NSGA2 algorithm (Kalyanmoy Deb et al. 2002) has been chosen for the optimization. The result shown in Figure 2 displays the corresponding Pareto front of **human comfort** versus **energy consumption**.

DOI
10.3384/ecp21757

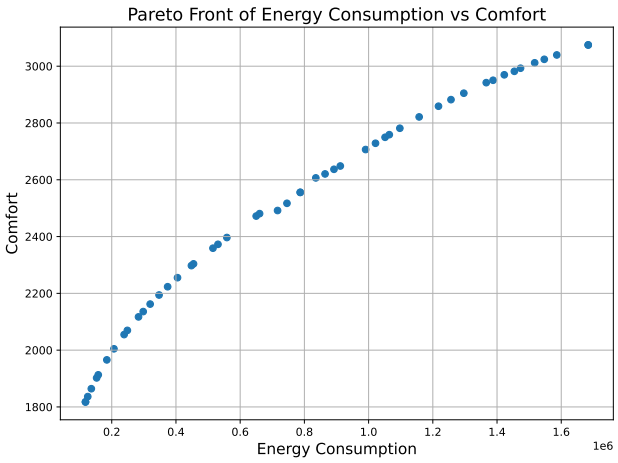

Figure 2. Pareto front of the simple heating system.

Another example involves an electric driving robot modeled using Modelica. This model focuses on a robot where the challenge is to find an optimal balance between **travel distance** and **energy consumption** at various **driving speeds**. In this scenario, **driving speed** is the key parameter to be adjusted. The objective is to fine-tune the speed to achieve the longest possible **travel distance** while minimizing **energy consumption**. Table 2 summarizes this scenario, including the parameter to be adjusted, objectives, and the goal.

Parameter	**Objectives**
Driving Speed	Travel Distance Energy Consumption
Goal	
Maximize Travel Distance Minimize Energy Consumption	

Table 2. Parameters, objectives, and goal-setting of use case 2.

The default configuration was used for this example, and the simulation time was set to 3000 seconds. The range for **driving speed** is 3 - 15 m/s (10.8 - 54 km/h). The NSGA2 algorithm has been chosen for the optimization. Since the diagrams of the Pareto front are similar, the result for the second use case is not shown here.

The results can inform decision-making processes for various applications. For instance, the first example helps decision-makers efficiently develop a strategy for heating a room. The second example involves multiple scenarios: (1) Optimizing an electric robot or electric vehicle for the target range and performance etc. (Dharumaseelan et al. 2021); (2) Optimizing electric car-sharing systems by analyzing vehicle states to select the most suitable car for a client, balancing environmental and economic considerations (Hamroun, Labadi, and Lazri 2020).

4 The Orchestration Workflow

Figure 3 shows the four components of the "ModelicaOrch" orchestration workflow.

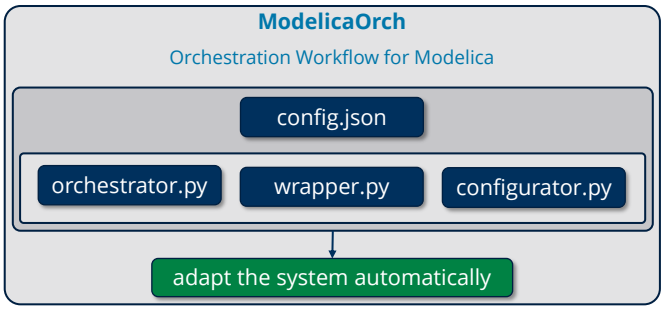

Figure 3. Structure of the workflow *ModelicaOrch*.

`config.json` The configuration file. It also dynamically adapts the MOO4Modelica configuration file.

`orchestrator.py` It initializes components, reads data, runs the optimization, and manages the entire simulation and evaluation loop. It acts as the central orchestration unit, ensuring that each component functions correctly and in sync with the others.

`wrapper.py` It manages the optimization process using MOO4Modelica, handles optimization results, and provides parameter sets for simulation. It effectively bridges the optimization and optimal control processes, ensuring that the best possible configurations are tested.

`configurator.py` The configurator updates the configuration based on the current status. It also prepares and sets parameters for the simulation.

Figure 4 shows how the orchestration workflow operates. After reading input data it enters the adaptive control loop that iterates over defined time units. In each iteration, the `orchestrator` calls the `configurator` to update optimization configurations and the `wrapper` to assign and retrieve optimized parameter sets found by MOO4Modelica. The `configurator` simulates and evaluates these sets to check if the goal is satisfied. If satisfied, the result is added to the final report; if not, the system tries the next parameter set until all options are exhausted. This process repeats until all time units are iterated, culminating in a comprehensive final report of the optimization results. This report can be used to refine the system and guide future adjustments. The adaptive nature of this workflow allows for continuous improvement and ensures that the system can respond effectively to changing conditions and requirements. By automating the optimization and adaptation process, the workflow significantly reduces the need for manual intervention, allowing for more efficient and reliable system management. This capability is particularly beneficial in complex systems where numerous parameters and configurations need to be considered.

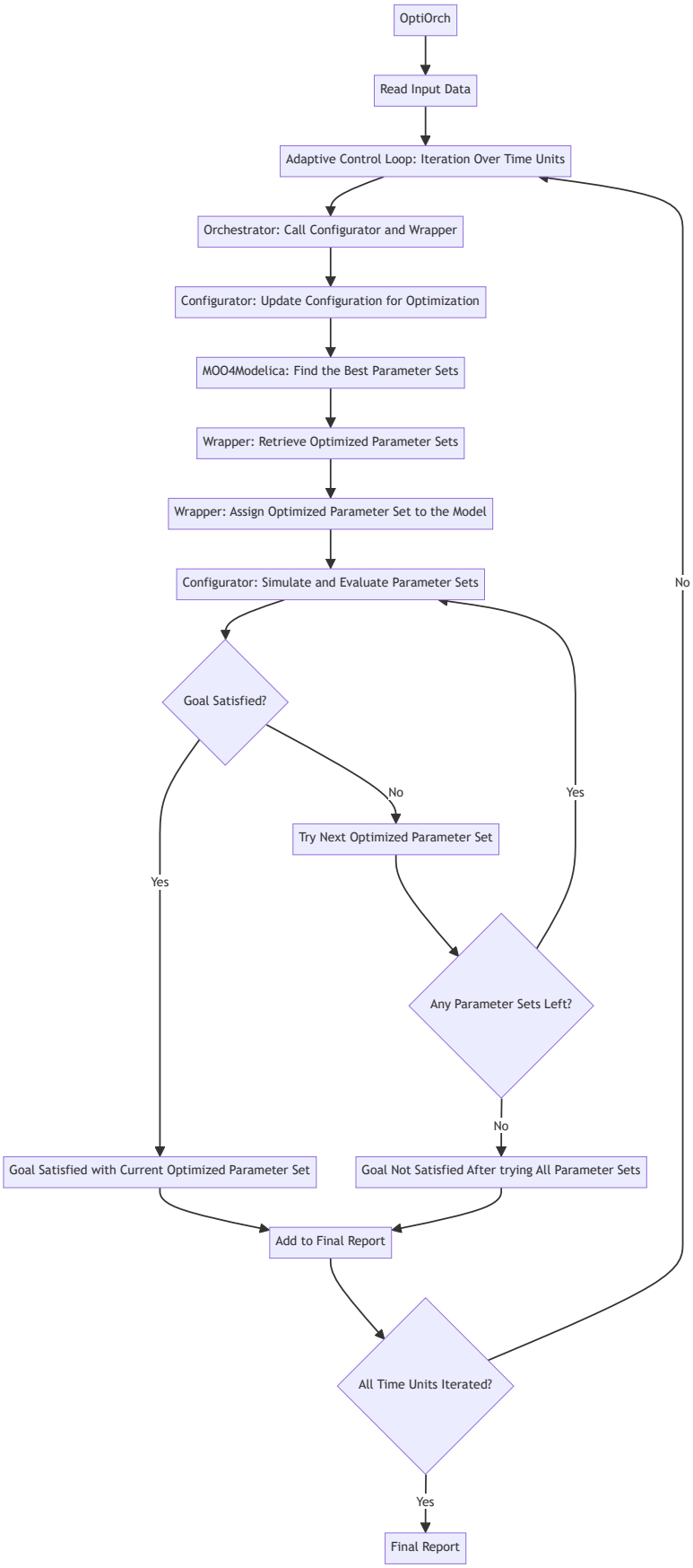

Figure 4. Flowchart of the orchestration workflow: The adaptive control loop iteratively optimizes parameters, updates configurations, evaluates models, and checks goal satisfaction over defined time units, systematically documenting results into a final report.

DOI
10.3384/ecp21757

5 Case Study

In the future, autonomous driving vehicles need distributed edge computing systems for low latency as well as high security and privacy, such as computing and data sharing locally. Based on this background, the case study built a simple edge computing system powered by Photovoltaic. Key parameters in this model are active CPU cores and frequency; different combinations of these parameters will result in various combinations of performance (computing power) and remaining energy, which are the key variables. In real applications, traffic flow varies at different times. For example, during the morning rush hour, available energy is still low, but user demand (the required computing power) is high. From 10 AM to 4 PM, user demand is low, and available energy is medium. During the afternoon rush hour, user demand is high, and the available energy is also high. Therefore, the system needs to adapt to different available energy levels and user demands to meet the user demand while keeping the remaining energy ≥ 0 for each defined time unit.

Input

- Energy Available (hourly)
- User Demand (hourly)

Goals

- **Meet User Demand:** The system aims to provide the necessary performance to meet user demand. User demand is considered satisfied when the system's performance (computing power the system provides) $\geq$ user demand.
- **Maximize Energy Efficiency:** The system seeks to optimize energy consumption to prolong operation and maintain efficiency, ensuring that the remaining energy ≥ 0 at the end of the simulation.

Listing 1 shows the configuration for this case study. The defined time unit is the hour, and the adaptive control loop runs for each hour in the simulation. For convenience, the time range has been selected between 8 AM and 12 AM, despite the input data having a time range of 24 hours. The model will be simulated for one hour (3600 seconds). Both objectives are set to be maximized, and the bounds and data types for parameters to be tuned are set to 1 - 4 (integer) and 1.0 - 3.0 (float), respectively. The goal expressions are set such that performance needs to be greater than or equal to user demand, and the energy should not run out (remaining energy should not be negative). `N_JOBS` has been assigned as "-1", which means that parallel computing is enables for the optimization, using all CPU cores. The `CONFIG_PATH` is the configuration file of the MOO4Modelica optimization framework. For each time unit, the orchestration configuration file will also update MOO4Modelica's configuration file to run optimization.

Listing 1. The configuration file for the case study.

```
"DATA_FILE_PATH": "data.txt",
"CONFIG_PATH": "config.json",
"MODEL_FILE": "ITSystem.mo",
"SIMULATION_TIME": 3600,
"TIME_CONFIG": {
    "START_TIME": 8,
    "END_TIME": 12,
    "TIME_UNIT": "hour"
},
"OBJECTIVES": [
    {"name": "remainingEnergy",
        "maximize": true},
    {"name": "performance",
        "maximize": true}
],
"TUNABLE_PARAMETERS": {
    "PARAMETERS": [
        "activeCores",
        "cpuFrequency"],
    "PARAM_BOUNDS": {
        "activeCores": {
            "bounds": [1, 4],
            "type": "int"},
        "cpuFrequency": {
            "bounds": [1.0, 3.0],
            "type": "float"}
    }
},
"INPUT_PARAMETERS": {
    "available_energy": "
       availableEnergy",
    "user_demand": "userDemand"
},
"CRITERIA": {
    "GOAL_EXPRESSION": [
        "evaluation_results['
           performance'] >=
           simulation_inputs['
           user_demand']",
        "evaluation_results['
           remainingEnergy'] >= 0"
    ]
},
"OPTIMIZATION_CONFIG": {
    "USE_SINGLE_OBJECTIVE": false,
    "ALGORITHM_NAME": "nsga2",
    "POP_SIZE": 10,
    "N_GEN": 10
},
"LIBRARY_CONFIG": {
    "LOAD_LIBRARIES": false,
    "LIBRARIES": [
        {"name": "", "path": ""}
    ]
},
"N_JOBS": -1
```

Figure 5 demonstrated the visualized result of the case study. At 8 AM, neither goal is satisfied. At 9 AM, the first goal is not satisfied. From 10 AM to 12 PM, both goals are satisfied. Based on these results, it is clear that for 8 AM and 9 AM, we need more power and updated hardware to meet the system requirements. For the remaining periods, we have already found the best configurations, which we can now implement into the real hardware.

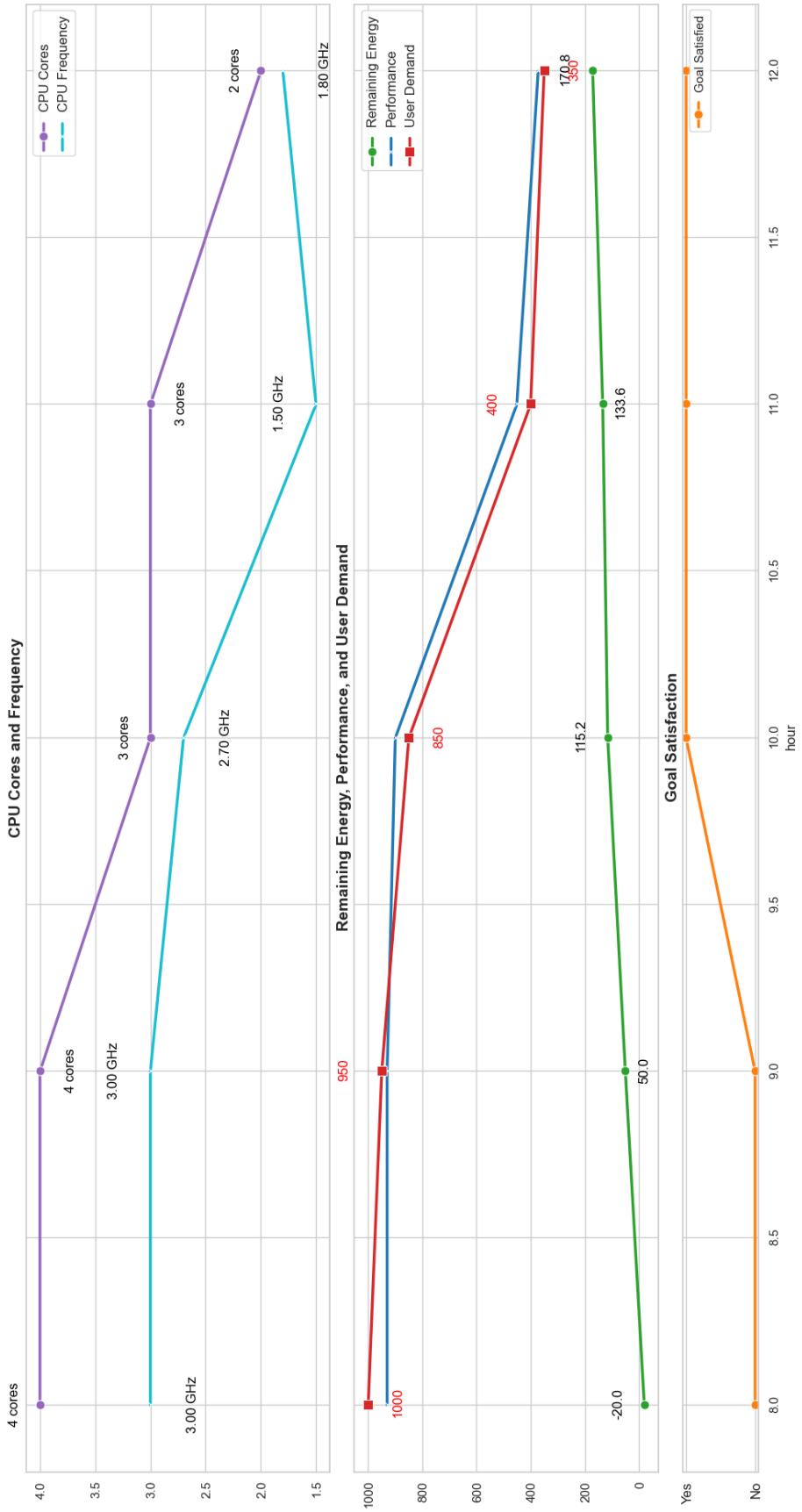

Figure 5. Visualized result of the case study

DOI
10.3384/ecp21757

6 Conclusion and Future Work

OptiOrch[3][4] is a toolchain integrating the MOO4Modelica optimization framework and the ModelicaOrch orchestration workflow. MOO4Modelica facilitates both single- and multi-objective optimization in Modelica-based simulations, featuring user-friendly setup configurations and practical feature model transformations. It leverages parallel computing to enhance performance. ModelicaOrch orchestrates the entire workflow, coordinating the optimization process and updating configurations dynamically. Together, they enable efficient and optimal control in complex Modelica simulations. Additionally, this toolchain is designed to be flexible and extensible, allowing users to adapt it to a wide range of optimization and orchestration scenarios.

Despite its robust capabilities, optimizing and orchestrating large-scale models can be both resource-intensive and time-consuming. To address this challenge, it would be interesting to investigate strategies such as using surrogate models for Modelica-based simulation and optimization (Costa Paulo et al. 2023) or implementing adaptive instance reduction (automatic search space reduction) to reduce the computation complexity. How these two concepts work in the Modelica ecosystem presents interesting research topics. In real-life applications such as edge computing systems, tasks can vary significantly, requiring the system to dynamically allocate resources (CPU, memory, etc.) based on their current demands and priorities. A future goal of this work is to integrate the toolchain with real-time hardware configurators. To achieve this, an architecture will be developed that combines simulation-based optimal control with real-time hardware configurators. By incorporating software like MQuAT (Multi-Quality Auto-Tuning by Contract Negotiation) by Götz 2013 and BRISE (Benchmark Reduction via Adaptive Instance Selection) by Pukhkaiev 2023 into the architecture shown in Figure 6, we can effectively fine-tune and configure real hardware systems to maximize performance and energy efficiency.

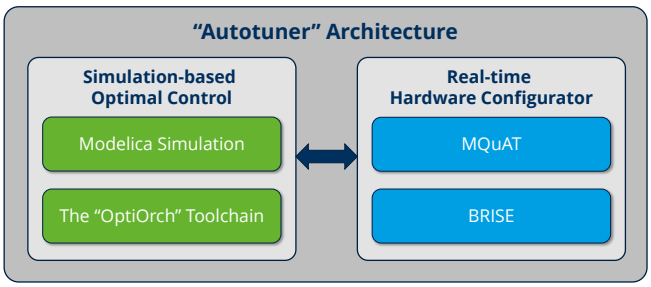

Figure 6. The "autotuner" architecture will define the interface between simulation-based optimal control and real-time hardware configurator.

[3]Repository: https://git-st.inf.tu-dresden.de/wang/OptiOrch
[4]Documentation: https://wangzizhe.github.io/OptiOrch

Acknowledgements

The author would like to thank the Boysen–TU Dresden–Research Training Group for the financial and general support that has made this contribution possible. The Research Training Group is co-financed by the Friedrich and Elisabeth Boysen Foundation and the TU Dresden.

References

Blank, J. and K. Deb (2020). "pymoo: Multi-Objective Optimization in Python". In: *IEEE Access* 8, pp. 89497–89509.

Brück, Dag et al. (2002). "Dymola for multi-engineering modeling and simulation". In: *Proceedings of modelica*. Vol. 2002. Citeseer.

Chen, Xia and Zhongchao Wei (2008). "A new modeling and simulation platform-MWorks for electrical machine based on Modelica". In: *2008 International Conference on Electrical Machines and Systems*. IEEE, pp. 4065–4067.

Costa Paulo, Breno da et al. (2023). "Surrogate model of a HVAC system for PV self-consumption maximisation". In: *Energy Conversion and Management: X* 19, p. 100396.

Deb, Kalyanmoy et al. (2002). "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE transactions on evolutionary computation* 6.2, pp. 182–197.

Dharumaseelan, Elavarasan et al. (2021). "Model Based Analysis and Multi-objective Optimization of an Electric Pickup truck for Range, Acceleration, Drivability, Handling and Ride Comfort Performances". In: *2021 IEEE Transportation Electrification Conference (ITEC-India)*. IEEE, pp. 1–6.

Elmqvist, Hilding (1979). "DYMOLA-a structured model language for large continuous systems". In.

Fritzson, Peter, Peter Aronsson, et al. (2005). "The OpenModelica modeling, simulation, and development environment". In: *46th Conference on Simulation and Modelling of the Scandinavian Simulation Society (SIMS2005), Trondheim, Norway, October 13-14, 2005*.

Fritzson, Peter and Vadim Engelson (1998). "Modelica—A unified object-oriented language for system modeling and simulation". In: *ECOOP'98—Object-Oriented Programming: 12th European Conference Brussels, Belgium, July 20–24, 1998 Proceedings 12*. Springer, pp. 67–90.

Fritzson, Peter, Adrian Pop, et al. (2022). "The OpenModelica integrated environment for modeling, simulation, and model-based development". In: Mic.

Ganeson, Anand Kalaiarasi et al. (2012). "An OpenModelica python interface and its use in PySimulator". In.

Götz, Sebastian (2013). "Multi-Quality Auto-Tuning by Contract Negotiation". In.

Hamroun, A, K Labadi, and M Lazri (2020). "Modelling and performance analysis of electric car-sharing systems using Petri nets". In: *E3S Web of Conferences*. Vol. 170. EDP Sciences, p. 03001.

Leimeister, Mareike (2019). "Python-Modelica framework for automated simulation and optimization". In.

Modelica Association (2023-03). *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification Version 3.6*. Tech. rep. Linköping: Modelica Association. URL: https://specification.modelica.org/maint/3.6/MLS.pdf.

Parr, Terence J. and Russell W. Quong (1995). "ANTLR: A predicated-LL (k) parser generator". In: *Software: Practice and Experience* 25.7, pp. 789–810.

Pfeiffer, Andreas (2012). “Optimization library for interactive multi-criteria optimization tasks”. In.

Pukhkaiev, Dmytro (2023). “A Software Product Line for Parameter Tuning”. In.

Senkel, Anne et al. (2021). “Status of the transient library: Transient simulation of complex integrated energy systems”. In: *Modelica Conferences*, pp. 187–196.

Sharma, Shubhkirti and Vijay Kumar (2022). “A comprehensive review on multi-objective optimization techniques: Past, present and future”. In: *Archives of Computational Methods in Engineering* 29.7, pp. 5605–5633.

Thieriot, Hubert et al. (2011). “Towards design optimization with OpenModelica emphasizing parameter optimization with genetic algorithms”. In: *Proceedings of the 8th International Modelica Conference*. Vol. 63, pp. 756–762.

Thirunavukkarasu, M, Yashwant Sawle, and Himadri Lala (2023). “A comprehensive review on optimization of hybrid renewable energy systems using various optimization techniques”. In: *Renewable and Sustainable Energy Reviews* 176, p. 113192.

Zhang, Congcong et al. (2022). “A Multi-objective Optimization Algorithm and Process for Modelica Model”. In: *2022 4th International Conference on Artificial Intelligence and Advanced Manufacturing (AIAM)*. IEEE, pp. 9–13.

DOI
10.3384/ecp21757