Thermo-Fluid Modeling Framework for Supercomputing Digital Twins: Part 2, Automated Cooling Models

Scott Greenwood[1] Vineet Kumar[1] Wesley Brewer[2]

[1]Fusion and Fission Energy and Science Directorate, Oak Ridge National Laboratory, USA,
`{kumarv, greenwoodms}@ornl.gov`
[2]Computing & Computational Sciences Directorate, Oak Ridge National Laboratory, USA, `brewerwh@ornl.gov`

Abstract

The development of digital twins for the purpose of improving the energy efficiency of supercomputing facilities is a non-trivial endeavor that is complicated by the difficulty of creating physics-based thermo-fluid cooling system models (CSMs). Within ExaDigit—an open-source framework for liquid-cooled supercomputing digital twins—a thermo-fluid modeling framework is being developed. This effort has been segmented into two with two companion papers describing each portion of the overall effort. Part 1 focuses on the development of a cooling system library in Dymola for the Frontier supercomputer at Oak Ridge National Laboratory (Kumar et al. 2024). Part 2, this paper, describes an effort to create a template-based auto-generation methodology for CSMs, called *AutoCSM*. In this paper, an overview of the initial AutoCSM architecture and workflow is provided, along with a practical example using the Oak Ridge Leadership Computing Facility's (OLCF) Frontier supercomputer CSM. AutoCSM will (1) improve ExaDigiT's user accessibility by providing a flexible workflow for modularizing the creation of the CSM system and control logic, (2) decrease the development time of CSMs, and (3) standardize the method for incorporating CSMs into the ExaDigiT framework.

Keywords: digital twin, automation, cooling system, supercomputer, architecture, framework

1 Introduction

Across myriad disciplines, high-performance supercomputing facilities have long been a key resource for exploring complex scientific and engineering challenges, spurring technological innovation and opening new avenues of discovery (National Research Council 2005). As the problems being explored increase in complexity, and therefore computational cost, it will become increasingly difficult to make significant advances in energy efficiency.

1.1 Motivation

Fully operational in 2022, the Frontier supercomputer at Oak Ridge National Laboratory became the world's first exascale supercomputer (Atchley et al. 2023). It could be argued that the ability to achieve this milestone was feasible principally because of the enormous gains in hardware optimizations that have been made over the past decade. For example, if the technology of 2009 used in the Jaguar Supercomputer were used for the Frontier facility, the power requirements would have been multiple gigawatts. Instead, Frontier consumes only approximately 20 MW while achieving 1,000 times higher performance than Jaguar. Although enormous strides have been made in the past decade on computational hardware, it is postulated that hardware optimizations have approached their limits of being the primary means of obtaining efficiency improvements; instead, future efficiency gains will be dominated by software innovations such as operational performance (i.e., controls, job staging/prediction) (Brewer et al. 2024). The use of digital twins is one such software innovation that may be a primary means of achieving the necessary performance efficiency improvements for current and future supercomputing systems.

1.2 Digital Twins

A *digital twin* is defined as a virtual representation of a real-world system that synchronizes and exchanges information with the real-world system. In the context of exascale computing facilities, the digital twin is expected to require, at minimum, connections to- and models of- power consumption, the cooling system, and network behavior. A digital twin should also incorporate human–computer interfaces and optimization capabilities. The realization of a digital twin for such computing facilities is non-trivial given the complexity and scale of the facility, components, and data.

1.3 ExaDigiT

To help accelerate the development of exascale facility digital twins and their value in achieving efficiency

This manuscript has been authored by UT-Battelle, LLC under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

gains, Oak Ridge National Laboratory created ExaDigiT, an open-source framework for developing comprehensive digital twins for liquid-cooled supercomputers (Brewer et al. 2024). This framework is intended to streamline the creation and connection of the necessary cross-disciplinary systems and data as well as to provide methods of performing advanced analysis and predictive exploration to build toward sustainable, energy-efficient supercomputing.

1.4 Cooling System Model (CSM) Development

The broader development approach to ExaDigiT takes into account that an effective digital twin of an exascale facility will likely require physics-based system models of the liquid cooling system at various levels of fidelity throughout the system's life cycle—and that it must also be capable of capturing the transient operations of the system for optimization and scenario exploration. In support of ExaDigiT's development, a Modelica model of Frontier's cooling system was completed and is detailed in a companion paper (Kumar et al. 2024). Part of that modeling efforts purpose was to provide insights into the value of the cooling system model (CSM) toward delivering efficiency improvements and into how the CSM would be integrated into ExaDigiT .

1.5 AutoCSM

The Frontier CSM described in (Kumar et al. 2024) adopted a preliminary template architecture based on previous efforts (Greenwood et al. 2017a). However, as the ExaDigiT framework matured and more specific users of the framework were identified, it became apparent that supercomputing facilities—and datacenters more generally—could benefit from a CSM template approach more tailored to the way the systems are hierarchically designed, constructed, and operated. The creation of sufficiently accurate physics-based CSMs is a non-trivial exercise that requires domain-specific knowledge and good modeling experience and best practices. The additional complexities of iterative development and the integration of the model into a broader digital twin exacerbate the difficulty of achieving proper value from a CSM in the broader digital twin. Therefore, the creation of a template system-of-systems modeling approach for automating the development, deployment, and integration of CSMs for supercomputing facilities was proposed. This methodology is called *AutoCSM*.

The remainder of this paper describes the AutoCSM proof-of-concept methodology in the broader ExaDigiT context in which it is situated. A description of the current architecture and general workflow of the AutoCSM is then provided. An illustrative example of the adaption of the original Frontier model to the AutoCSM approach is then provided as well as the extension of that AutoCSM based model for exploration of a parallel datacenter study. Finally, because the users of ExaDigiT are expected to have limited familiarity with Modelica, and to help clearly address the role of AutoCSM in ExaDigiT explicitly, a section of exploring potential questions regarding AutoCSM is provided.

2 ExaDigiT & AutoCSM

The CSM within ExaDigiT consumes telemetry data (e.g., facility operation and job loading data, weather data), couples with the power simulator (RAPS), and ultimately produces operational predictions that can be leveraged for scenario exploration and facility health analysis. The CSM also provides data to reduced-order models for AI/ML facility studies, optimization, and visual analytics (Figure 1). Given the wide adoption of the open-source Functional Mock-up Interface (FMI) standard (*Functional Mock-up Interface* n.d.) by various tools and programming languages—and given its direct application to dynamic simulations at the levels of fidelity that are expected to be valuable for supercomputer digital twins—using Functional Mock-up Units (FMUs) was identified as the primary method of standardizing the incorporation of the CSM into the broader ExaDigiT framework. However, a question was posed about how the process for generating the FMU could be simplified for the user, making it less error prone and less time-intensive, as well as requiring less experience, while remaining agnostic to the underlying technology (e.g., commercial software, programming languages) used to create the CSM. This line of reasoning led to introducing AutoCSM as an optional interface layer for streamlining CSM model generation to FMU deployment. In the framework depicted in Figure 1, AutoCSM provides a means of automating the process for creating a simulation-ready CSM FMU, depicted on the left side.

3 AutoCSM

The motivating philosophy behind AutoCSM is to remove as many barriers as possible to incorporating CSMs into a facility's digital twin. Therefore, attention was given to identifying the broad architecture, identifying functional requirements, and determining how AutoCSM development could be compartmentalized and focused to achieve near-term impact. This section touches on each of these topics and provides a pseudo-code example of the AutoCSM workflow.

3.1 Architecture

AutoCSM is intended to be an optional interface that can be used to automate the generation of system models that have adopted a template architecture for rapid deployment to simulation-ready FMUs. The AutoCSM interface, or AutoCSM API (Figure 2), relies on the implementation of an API that has a language- or modeling-specific extension. That extension will be used to script the generation of the CSM in that language using a user template strategy that can be unique to their facility and language. Finally, the scripted model will be exported as an FMU using ex-

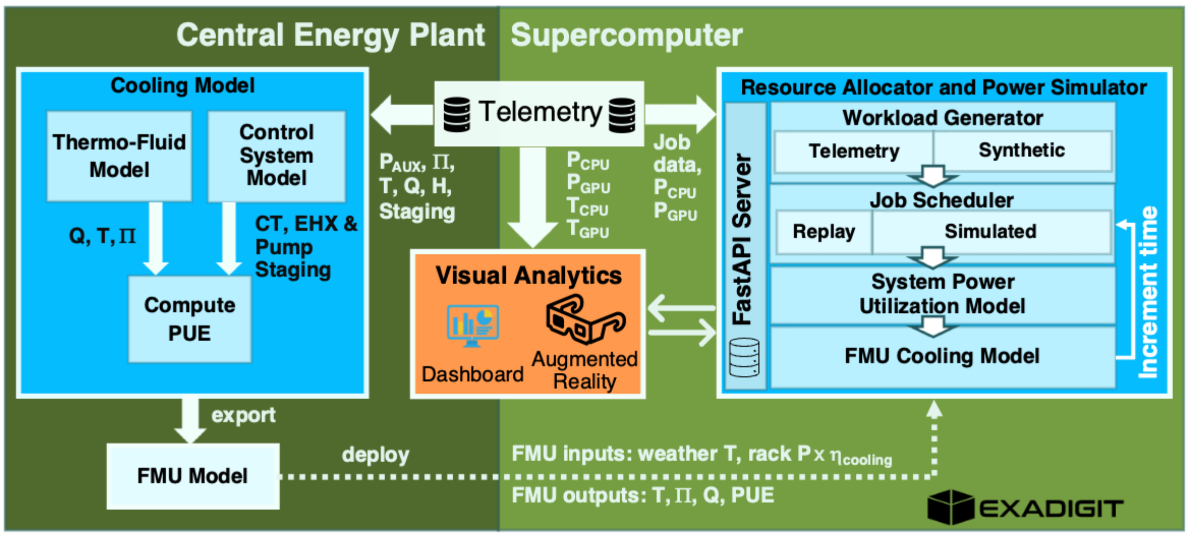


Figure 1. The base ExaDigiT framework, incorporating telemetry data, a power simulator (RAPS), visual analytics, and a CSM. (Brewer et al. 2024)

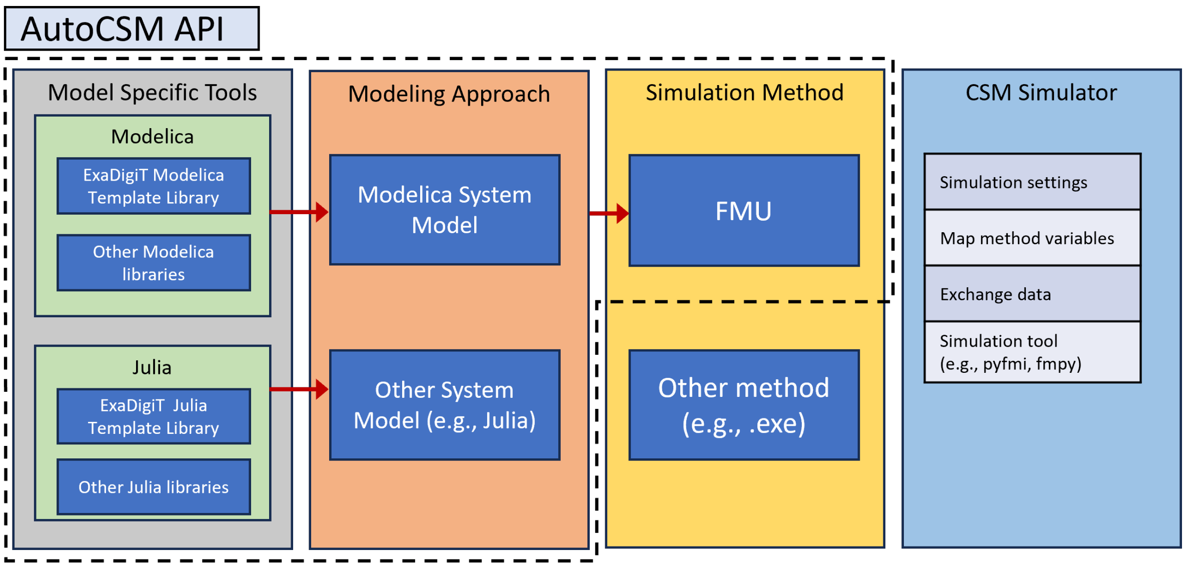


Figure 2. AutoCSM API in the broader ExaDigiT procedure. Dotted line indicates scope of AutoCSM.

isting tools. Again, the tool used will be user-definable and can be chosen according to the user's preferred modeling approach and tool (e.g., integrated development environment or language FMU export library). This FMU will then be consumed within the broader ExaDigiT CSM Simulator. At each step, AutoCSM is intended to remain as agnostic to specific facility requirements as possible by abstracting and reducing modeling requirements to the input specification (to be discussed).

Internal to the AutoCSM API is a specific procedure that is used to create an FMU export from user-input specification. Figure 3 illustrates the process of consuming input specifications. Additional details of the principal focus areas of the API are discussed in a subsequent section.

3.2 Functional Requirements

The functional requirements for the development of AutoCSM are identified below. Some have been mentioned previously but are repeated here for completeness. In general, the broad theme of the requirements is to provide the framework and avoid encroaching on a user's methods. AutoCSM will:

1. conform to the deployment requirements of ExaDigiT (e.g., open-source),
2. be language/tool agnostic (both across and within tools),
3. support custom specification extensions,
4. leverage existing solutions/methods where possible (e.g., third-party Python libraries),

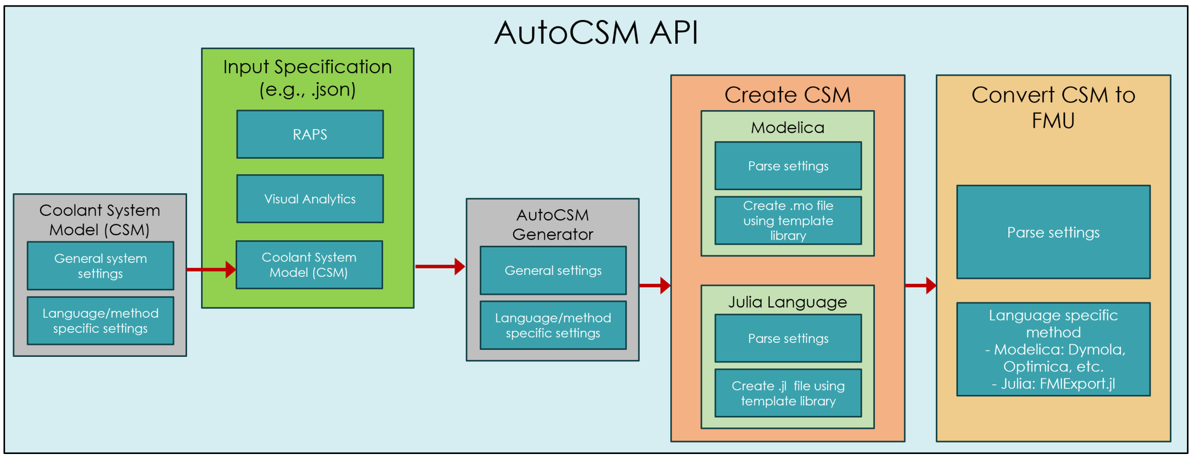

AutoCSM API
Coolant System Model (CSM)
General system settings
Language/method specific settings
Input Specification (e.g., .json)
RAPS
Visual Analytics
Coolant System Model (CSM)
AutoCSM Generator
General settings
Language/method specific settings
Create CSM
Modelica
Parse settings
Create .mo file using template library
Julia Language
Parse settings
Create .jl file using template library
Convert CSM to FMU
Parse settings
Language specific method
- Modelica: Dymola, Optimica, etc.
- Julia: FMIExport.jl

Figure 3. Internal to AutoCSM API procedure for CSM creation.

5. produce an FMU consistent with the requirements of ExaDigiT,
6. assemble pre-configured systems and subsystems (i.e., it will not generate the internals to a system model itself, such as connecting or creating pipes, volumes, etc.), and
7. not dictate the modeling template approach.

3.2.1 Modeling Approach Requirements

As shown in Figure 2, AutoCSM requires a modeling approach (e.g., Modelica, Julia, or other software/languages and libraries) that supports the creation of a formal modeling structure or template approach for the assembly of the FMU. To use AutoCSM for a selected modeling approach, three requirements have been identified.

1. Be templatable (i.e., able to organize models into system-subsystem architectures)
2. Be scriptable at the level of system assembly (i.e., not binary or otherwise inaccessible from outside tools)
3. Support FMU export

Satisfying these three requirements allows AutoCSM to support virtually any modeling approach a user may want to employ. If a requirement is not met with a selected tool, then the user will need to decide if that is an insurmountable issue with that tool (e.g., commercial software unable to be adapted) or if investment in an AutoCSM extension for that tool is feasible. As an open-source project, AutoCSM will be open to community contributions, and users may develop proprietary extensions that are not shared with the broader community.

3.3 AutoCSM Focus Areas

The development of the AutoCSM API consists of three distinct focus areas: input specification, the automated CSM generator, and an example template architecture for demonstration and development. Figure 3 provides an illustration of these areas within the internal AutoCSM methodology, from input specification to model creation to FMU generation. The following subsections elaborate on the role of each of these focus areas.

3.3.1 Input Specification

AutoCSM requires a standardized means of collecting the necessary information into a form that can be incorporated into the broader ExaDigiT input requirements. As referenced in Figure 3, the specification that proceeds to the CSM generation stage is an amalgam of the CSM-specific settings and additional information from other ExaDigiT pieces that a CSM may indirectly require. From a user's perspective, the CSM settings are what a modeler would define, and the CSM generator would use those CSM settings after they have undergone a settings compilation step. Potential examples of CSM input include the number of various systems used, modifications on parameters and input, and/or specification of the different versions of systems to be used.

3.3.2 AutoCSM Generator

The AutoCSM Generator provides the necessary logic to automatically translate the input specification into a complete system model that can be used within the ExaDigiT framework. The initial demonstration uses the Modelica language, but the API is extensible so that other languages or tools can be added as needed by users (e.g., Julia). To limit scope creep and edge cases, the initial development focused on the creation of co-simulation FMUs as the product of the automation process for use in the digital twin.

3.3.3 ExaDigiT Modelica Library

To enable automated generation of a CSM, a formalized architecture or template approach that compartmentalizes the facility into standardized systems, subsystems, and control logic is required. This is achievable because of how supercomputer facilities are constructed and organized and because of the level of fidelity necessary from the CSM for ExaDigiT's purposes. Therefore, work in this focus area involved creating a proof-of-concept template approach for assembling the facility.

As previously noted, a CSM for Frontier was developed (Kumar et al. 2024) to provide support to its operations and contribute to the development of ExaDigiT (**??**). The model outlined in the aforementioned work leveraged a template approach previously developed for advanced energy systems studies contained within the TRANSFORM Modelica library (Greenwood et al. 2017b). Although the adopted template approach has been adequate for its original purposes, the CSM would benefit from an adapted template approach that aligns better with a nested structure that could be leveraged by external programming languages and be better aligned with ExaDigiT. Therefore, an ExaDigiT specific Modelica template prototype was developed using other Modelica template approaches (Modelica Asscotiation 2024; Greenwood et al. 2017a) for inspiration and the Frontier CSM modeling efforts as a use case demonstration. Figure 4 shows the outline of the developed Modelica template system. This leverages identical underlying structures–*BaseClasses*–and replaceable components to create a common template–*TemplateSystem*–that can be duplicated by AutoCSM or a user and serve as the foundational structure to create an AutoCSM compliant model.

In addition to a common structure, at each system level the models are implemented as arrays such that all inputs/outputs to the simulation can be readily associated with ExaDigiT input/output requirements and the ExaDigiT user can easily customise the structure of their facility. For example, to access the summary output of a particular model a path of the following form can be utilized: *system[i].systemA[j].systemB[k].summary.VAR* where *i*, *j*, and *k* are the index of the instance reference. Finally, if desired, parallel flow logic may be implemented for situations where computational performance is more important than model fidelity. If employed, the parallel logic treats an array of model system as a single representative model (i.e., without parallel logic an array may have 10 instances but with it included and enabled an array will be reduced to a 1 instance). This feature is built into the input specification and Modelica template library structure.

Figure 5 shows an example of a system model using the ExaDigiT Modelica template library. One of the advantages of this approach is that no matter which level in the hierarchy a system exists it has the same foundational components shown in the figure–i.e., structure, summary, inputs, control system, data, and the sensor bus. Everything beside those components are definable by the user for that system–e.g., alternative fluid ports, or none at all. Below is a description for each of these components.

- *structure*: instances of the same name as system-models used at that level. This component contains information that requires recompilation of a model such as the number of instances and a flag to enable a parallel model.
- *summary*: user-defined variables or calculations of interest to users of the model that are not readily accessible or desired to be highlighted–e.g., some type of calculation of many variables.
- *inputs*: replaceable model containing time-dependent variables for external access.
- *control system*: replaceable control system model for that model.
- *data*: replaceable data for containerizing information for that model.
- *sensorBus*: An expandable model for collecting signals for communicating to/from inputs, control systems, or other levels in the hierarchy.

This template approach directly informs the input specification development and is used by the automation process. Although AutoCSM does not require the use of this specific template approach, this library can also serve to accelerate the development of the system models themselves, as well as modified template approaches that may be more appropriate for a specific supercomputing facility. Additional features of the ExaDigiT Modelica Library are templates for testing system models for verification purposes. Leveraging the overall template approach and the practices demonstrated therein should provide significant efficiency gains for supercomputer facility modelers and analysts. To help orient a new user to using AutoCSM, a simplified Modelica library—*GenericCSM*—demonstrating the use of the template library is included with the AutoCSM source code. It is expected that new adopters of AutoCSM using Modelica would take that model and then adapt it to their system.

3.4 Example AutoCSM Workflow

Figure 6 provides a step-by-step pseudo-code workflow of AutoCSM's use, along with a description of each step. The current version of AutoCSM is written in Python and adopts a RedFish-style (*REDFISH | DMTF* n.d.) JSON input specification. The FMU is generated using Dymola's Python API, and simulation of the generated FMU is performed via FMPy (CATIA-Systems 2024). To reiterate, the use of tool-specific choices (i.e., Dymola) is not dictated by AutoCSM.

4 Frontier AutoCSM Demonstration

The Frontier CSM detailed in (Kumar et al. 2024) was modified and updated to use the ExaDigiT Modelica template library as described above and AutoCSM API for model generation. This section will first briefly discuss key aspects of that conversion and FMU auto-generation process. The ability to then extend this approach to extension of that base model to a preliminary exploration of

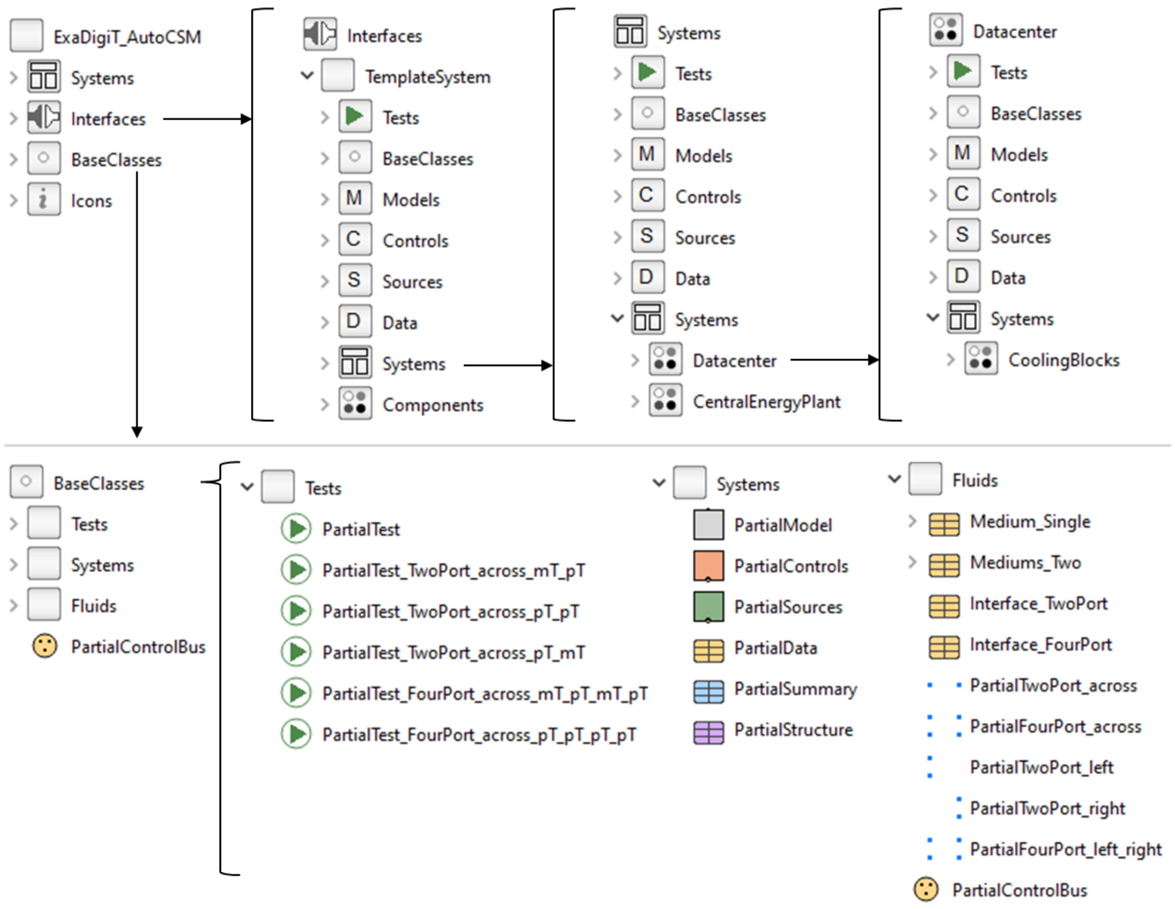


Figure 4. ExaDigiT Modelica template library. All systems use a common *TemplateSystem* that is then modified for the specific system. The *TemplateSystem* relies on replaceable components built on the (BaseClasses) models.

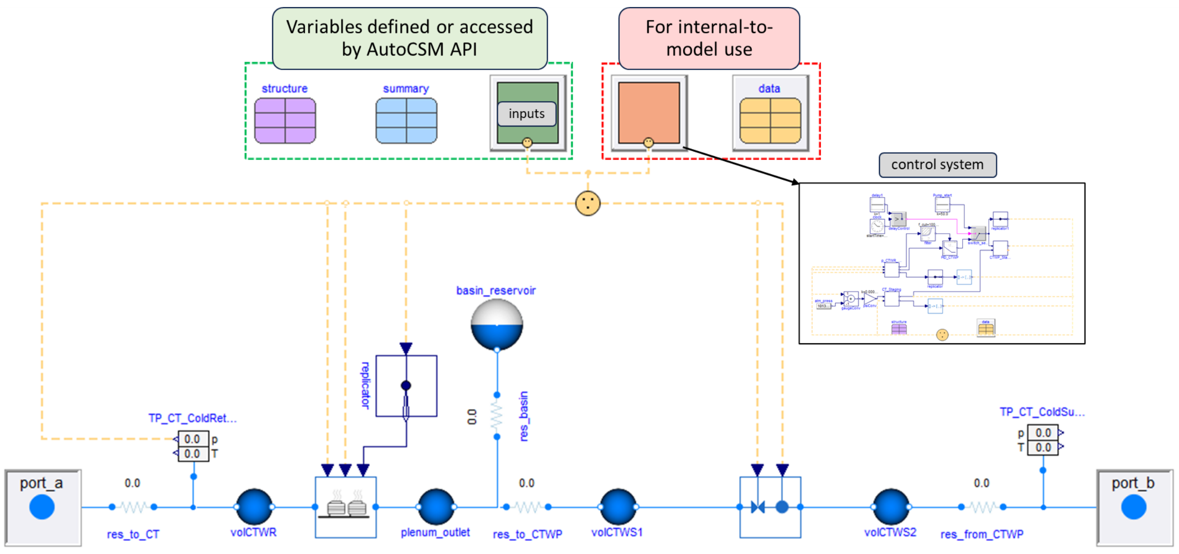
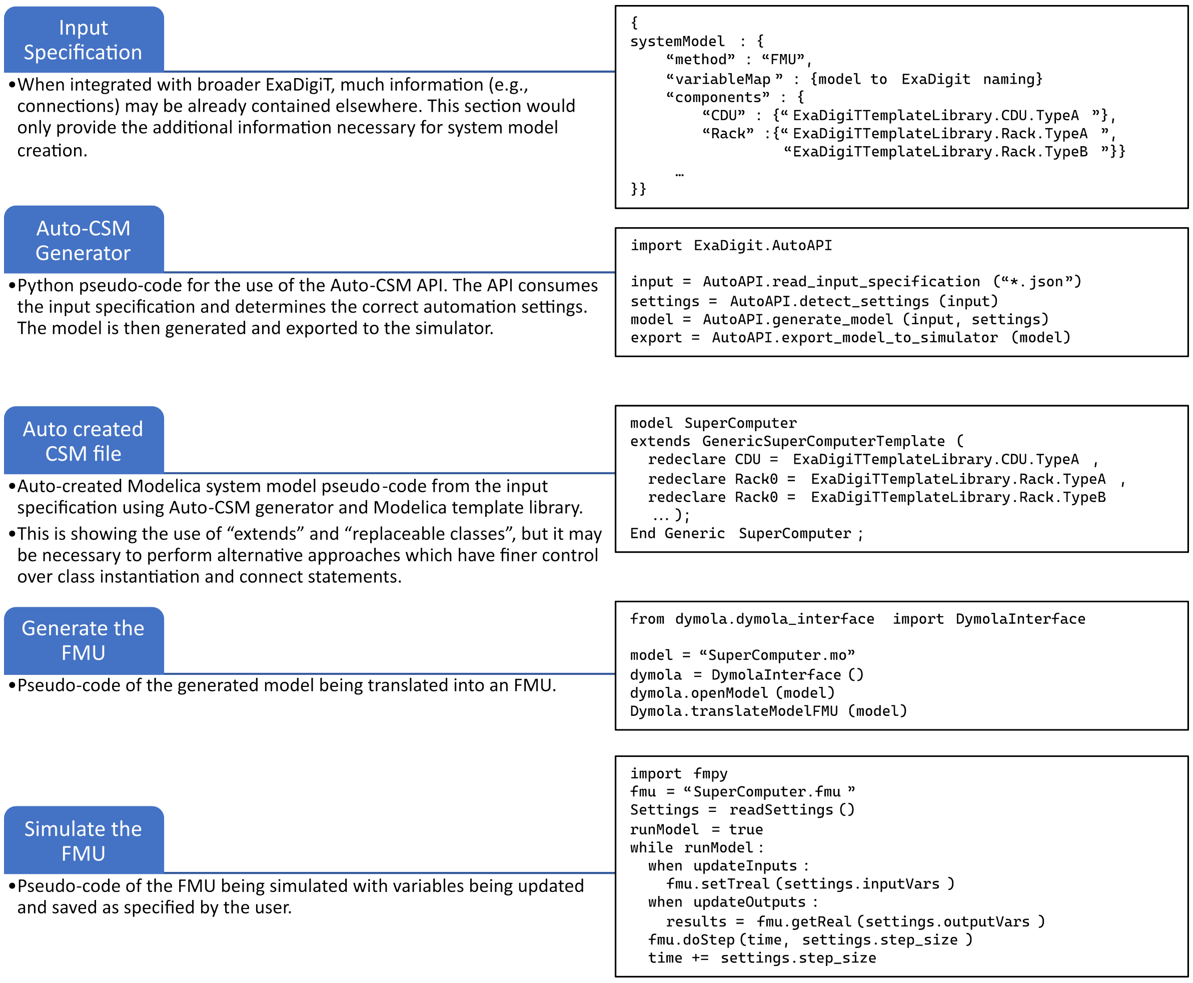


Figure 5. Example of a system model using the ExaDigiT Modelica template library. This is the Frontier AutoCSM model cooling tower loop. Every system model has a common set of components for use by the AutoCSM API or for internal-to-model usage.

two parallel datacenters with a single heat rejection system will be presented.

4.1 Conversion of Frontier to AutoCSM

The conversion process of taking the pre-AutoCSM Frontier model and converting it to the AutoCSM approach involved four general steps. In each of these steps, as-



• When integrated with broader ExaDigiT, much information (e.g., connections) may be already contained elsewhere. This section would only provide the additional information necessary for system model creation.

```
{
systemModel  : {
    “method” : “FMU”,
    “variableMap ” : {model to  ExaDigit  naming}
    “components” : {
        “CDU” : {“ ExaDigiTTemplateLibrary.CDU.TypeA  ”},
        “Rack” :{“ ExaDigiTTemplateLibrary.Rack.TypeA  ”,
                 “ExaDigiTTemplateLibrary.Rack.TypeB  ”}}
    …
}}
```

• Python pseudo-code for the use of the Auto-CSM API. The API consumes the input specification and determines the correct automation settings. The model is then generated and exported to the simulator.

```
import ExaDigit.AutoAPI

input = AutoAPI.read_input_specification  (“*.json”)
settings = AutoAPI.detect_settings (input)
model = AutoAPI.generate_model (input, settings)
export = AutoAPI.export_model_to_simulator  (model)
```

• Auto-created Modelica system model pseudo-code from the input specification using Auto-CSM generator and Modelica template library.
• This is showing the use of “extends” and “replaceable classes”, but it may be necessary to perform alternative approaches which have finer control over class instantiation and connect statements.

```
model SuperComputer
extends GenericSuperComputerTemplate (
  redeclare CDU =  ExaDigiTTemplateLibrary.CDU.TypeA  ,
  redeclare Rack0 =  ExaDigiTTemplateLibrary.Rack.TypeA  ,
  redeclare Rack0 =  ExaDigiTTemplateLibrary.Rack.TypeB
  … );
End Generic  SuperComputer ;
```

• Pseudo-code of the generated model being translated into an FMU.

```
from dymola.dymola_interface  import DymolaInterface

model = “SuperComputer.mo”
dymola = DymolaInterface ()
dymola.openModel (model)
Dymola.translateModelFMU (model)
```

• Pseudo-code of the FMU being simulated with variables being updated and saved as specified by the user.

```
import fmpy
fmu = “SuperComputer.fmu ”
Settings = readSettings ()
runModel = true
while runModel :
  when updateInputs :
    fmu.setTreal (settings.inputVars )
  when updateOutputs :
    results = fmu.getReal (settings.outputVars )
  fmu.doStep (time, settings.step_size )
  time += settings.step_size
```

Figure 6. Pseudo-code workflow of the AutoCSM workflow with descriptions of each of the steps.

sociated variables for control, summary, or external input where also updated accordingly. Once all steps were complete, the Frontier model was properly formatted into a hierarchical structure Figure 8 and accessible by AutoCSM for the auto-generation of the FMU.

1. Identify and convert models to arrays and then create template-based *System* packages accordingly–e.g., the compute blocks that make up the datacenter Figure 7.

2. Break models into compartmentalized sections and convert, as with the first step, or turn into component-only models.

3. In parallel two the previous two steps, create test simulations for dynamic and steady-state to verify the models returns the expected results. Typically these tests only use a small number of instances of a particular system model to be tested sufficient to verify the behavior and performance–e.g., 2 compute blocks instead of 25. The input JSON file facilitates creation of the complete model.

4. Improve numerical or structural issues uncovered via the tests to reduce or eliminate numerical issues (e.g., numerical Jacobians and non-linearities).

4.2 Frontier to AutoCSM FMU

A nested hierarchical modeling approach for Modelica was implemented in AutoCSM API. Therefore, with the Modelica model updated to this approach the input JSON specification was created that reflects the desired overall model structure including instances of each model and parallel logic flags Figure 10. The JSON file is then processed and an FMU is generated using the AutoCSM Python API Figure 10. If needed, the intermediate *.mo* file generated in this process is accessible and can be loaded into a Modelica IDE for simulation, debugging, or manual FMU generation. After the Frontier model was converted to the AutoCSM approach, and various numerical issues were resolved, the simulation time was cut approximately by one-third while significantly improving the tractability and robustness of the model.

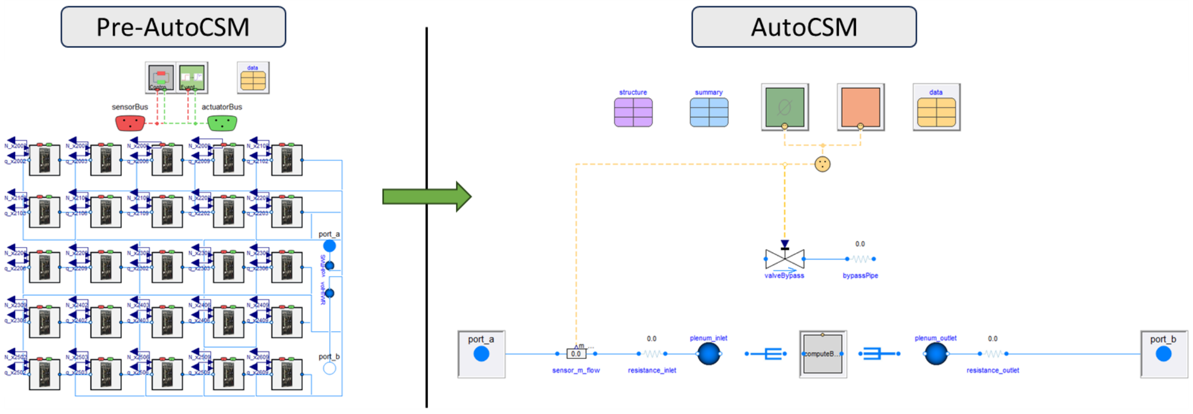


Figure 7. Example of the datacenter portion of the Frontier CSM before and after using the ExaDigiT Modelica AutoCSM template library where the compute blocks were able to be modified to an array.

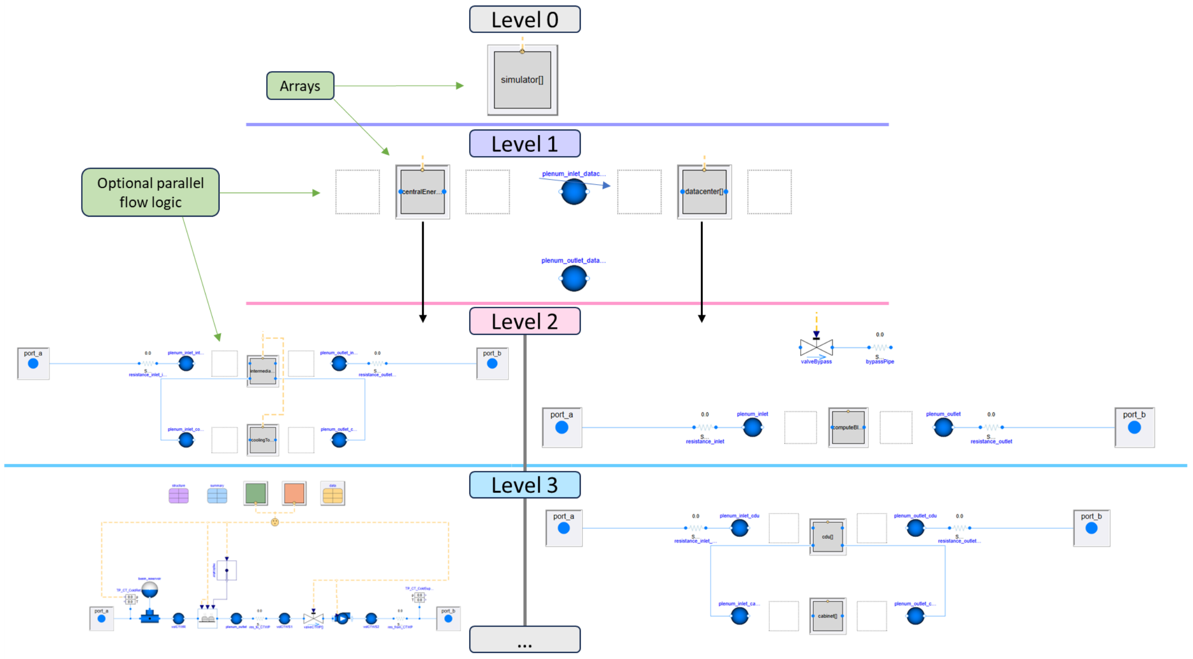


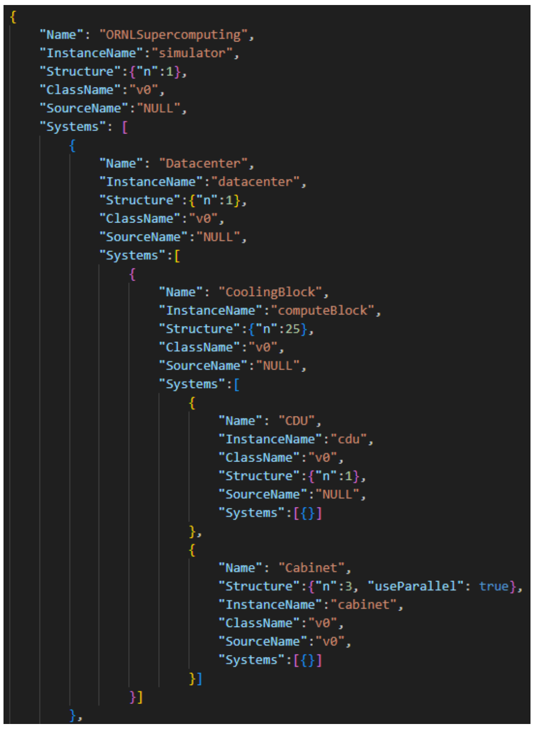
Figure 8. Subset of the Frontier CSM model implemented using the ExaDigiT Modelica AutoCSM template library demonstrating the nested hierarchy of the model structure. The dots at the bottom indicate that the levels of the hierarchy may continue as needed.

4.3 Extension to Alternative Studies

The migration to an AutoCSM approach enabled the exploration of a secondary test at Oak Ridge National Laboratory (ORNL). ORNL is beginning to assess whether its heat rejection system can support both Frontier and the next flagship supercomputer simultaneously. Although a detailed performance analysis of this system is beyond the scope of this work, it's worth discussing how AutoCSM could be adapted for that study.

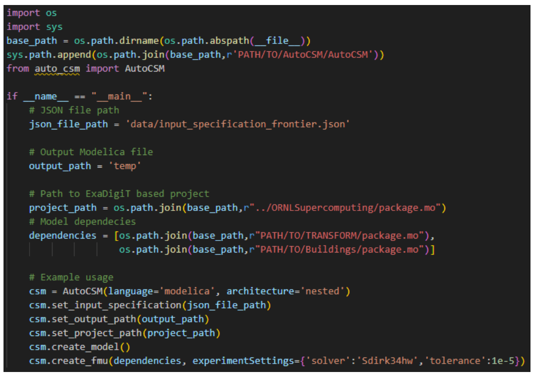
A key requirement for this effort is that the new supercomputer will be structurally differ from Frontier. Since Modelica doesn't support arrays of replaceable models from different classes, a second datacenter was added in parallel to Frontier instead of simply increasing the instance number by one. The datacenter-level structure component was modified to include this addition. With the new model and associated sub-models created, the JSON file was updated to include a new *Datacenter* system under the *ORNLSupercomputing* node, named *datacenter_new*. The entries for the new datacenter were populated, and an FMU was generated without changes to the Python code.

This exercise demonstrated the flexibility of the AutoCSM approach to meet the needs of various datacenter studies with minimal effort. Originally designed for a supercomputer facility, AutoCSM is also likely applicable to most datacenter modeling activities and could likely serve as a starting point for any system modeling effort with a hierarchical architecture.



```json
{
    "Name": "ORNLSupercomputing",
    "InstanceName":"simulator",
    "Structure":{"n":1},
    "ClassName":"v0",
    "SourceName":"NULL",
    "Systems": [
        {
            "Name": "Datacenter",
            "InstanceName":"datacenter",
            "Structure":{"n":1},
            "ClassName":"v0",
            "SourceName":"NULL",
            "Systems":[
                {
                    "Name": "CoolingBlock",
                    "InstanceName":"computeBlock",
                    "Structure":{"n":25},
                    "ClassName":"v0",
                    "SourceName":"NULL",
                    "Systems":[
                        {
                            "Name": "CDU",
                            "InstanceName":"cdu",
                            "ClassName":"v0",
                            "Structure":{"n":1},
                            "SourceName":"NULL",
                            "Systems":[{}]
                        },
                        {
                            "Name": "Cabinet",
                            "Structure":{"n":3, "useParallel": true},
                            "InstanceName":"cabinet",
                            "ClassName":"v0",
                            "SourceName":"v0",
                            "Systems":[{}]
                        }]
                }]
        },
```

Figure 9. Part of the Frontier input JSON specification file for AutoCSM used to auto-generate the FMU.



```python
import os
import sys
base_path = os.path.dirname(os.path.abspath(__file__))
sys.path.append(os.path.join(base_path,r'PATH/TO/AutoCSM/AutoCSM'))
from auto_csm import AutoCSM

if __name__ == "__main__":
    # JSON file path
    json_file_path = 'data/input_specification_frontier.json'

    # Output Modelica file
    output_path = 'temp'

    # Path to ExaDigiT based project
    project_path = os.path.join(base_path,r"../ORNLSupercomputing/package.mo")
    # Model dependecies
    dependencies = [os.path.join(base_path,r"PATH/TO/TRANSFORM/package.mo"),
                    os.path.join(base_path,r"PATH/TO/Buildings/package.mo")]

    # Example usage
    csm = AutoCSM(language='modelica', architecture='nested')
    csm.set_input_specification(json_file_path)
    csm.set_output_path(output_path)
    csm.set_project_path(project_path)
    csm.create_model()
    csm.create_fmu(dependencies, experimentSettings={'solver':'Sdirk34hw','tolerance':1e-5})
```

Figure 10. The python code required to process the input JSON file to create a Frontier FMU.

5 Potential Questions

During the exploration and development of AutoCSM, several common questions were identified, especially from the perspective of a potential user unfamiliar with Modelica. Below is a non-exhaustive list of questions that may be relevant to a potential AutoCSM user. While some of these questions or their answers are addressed in part within this paper, they are repeated here for completeness.

What is the value of an AutoCSM API in ExaDigiT? Creating CSMs is a time-consuming process that typically requires a deep understanding of the facility and expertise across various domains, such as thermal-hydraulics and controls. Additionally, integrating the model demands another specialized skill set, further complicating CSM development. AutoCSM accelerates this process by offering a step-by-step framework that allows for greater compartmentalization between the CSM developer and the user (in ExaDigiT). Another key time-saving benefit of AutoCSM lies in its support for exploratory studies, from analyzing subsystems at different levels of fidelity to exploring what-if scenarios to understand how design or operational changes might affect facility performance. Ultimately, time savings is the core value of AutoCSM.

What is the value of having a CSM in ExaDigiT? The benefit of having a CSM in ExaDigiT is to allow a user to understand how all aspects of their facility interact—for example, how job loading, facility cooling, and facility usage all respond to each other. This type of information can then be leveraged for exploration and optimization during all facility life cycles—across the processes of design, upgrades/downgrades, and operation. Thus, a CSM within ExaDigiT enables improvements and changes in facility design and operational efficiency that otherwise may not be possible.

If a user's digital twin needs do not require the use of the same or all subsystem levels as those of the template example library, how will the template system handle this scenario? The template system assumes top-down assembly of the CSM, allowing users to abstract lower-level facility components. The automation process adapts to the user's desired level of detail. For instance, if individual blades don't need modeling, the user can specify this in the input, omitting lower-level template subsystem models. The CSM modeler must only ensure that the models necessary for an ExaDigiT study are included.

As many Modelica integrated development environments (IDEs) are commercial software, is there concern that the template library will become tool specific? Although this work will use the commercial Modelica IDE Dymola to accelerate development, the proof-of-concept development will use pedantic mode to strictly enforce language standards such that the library should be usable with any Modelica-compliant IDE (e.g., Dymola, Modelon Impact, OpenModelica, ANSYS Twin Builder). Although this requirement will not be enforced upon users of ExaDigiT, the template library and any components used to create the example facility will be cross-tool compliant. To satisfy this requirement, components from existing libraries will be used, if possible; otherwise, modified components that satisfy the requirement will be created. However, this work prioritizes the input specification and API over the Modelica library, so the initial version will only include essential model development.

Is the limited availability of Modelica expertise a concern in AutoCSM value and adoption? While the methodology is language-agnostic, Modelica is chosen for initial demonstration due to its established capabilities. Alternatives like Julia's ModelingToolkit may be viable in the future, but uncertainties exist regarding problem size handling, solver availability, language stability, and domain-specific libraries for CSM-relevant dynamic system modeling. Lessons learned from Modelica are expected to be transferable to other approaches. More generaly, Modelica has a proven track record of signficantly decreasing model development time as compared to other approaches. Therefore, with AutoCSM and other open libraries available to users, it is expected that the creation of CSM subsystems for use in ExaDigiT, even by novice Modelica users, will not be a barrier to using ExaDigiT.

What is the advantage to using Modelica over alternative modeling languages? The advantages of using Modelica for this type of application derives from the maturity of the Modelica Language Standard and of the integrated development environments which implement it. The three primary relevant aspects is the importance of supporting *extends*, *replaceable*, and the language being acausal. The acausal nature assists in rapid and flexible model creation. The other two are foundational for creating architecture based implementations.

How will the AutoCSM process know how to model a user's facility? The template architecture provides the framework for connecting systems. This system-of-systems abstraction is therefore abstracted to a level where the method of defining the interfaces is the critical enabler for automating CSM creation. The architecture will not create the internal logic of individual subsystems. For example, the specific way to model the manner in which a blade or GPU is cooled will not be automated. The user must create the subsystem internal model by using Modelica components and then connect that internal model to the subsystem template.

6 Conclusions

The development of digital twins is a non-trivial endeavor. Methods that can help standardize and streamline the process for model development and incorporation into a framework used to operate a digital twin are critical. AutoCSM is one such methodology that aims to accelerate CSM development for integration into ExaDigiT's digital twin framework for supercomputing facilities. It strives to enhance speed, robustness, and the quality of results by enabling users to focus more on specific problems by abstracting out of the workflow, to the greatest extent possible, the infrastructure requirements for connecting models. This paper provides an overview of the AutoCSM methodology and workflow and provides an example overview of AutoCSM being used on the ORNL Frontier supercomputer facility. Future efforts in AutoCSM development will be driven by community adoption and feedback to the open source code base. Therefore, if AutoCSM is relevant to a potential user's needs, they are highly encouraged to provide feedback to the authors or directly via the code repository (`https://code.ornl.gov/exadigit/AutoCSM`).

Acknowledgments

This research was sponsored by and used resources of the Oak Ridge Leadership Computing Facility (OLCF), which is a DOE Office of Science User Facility at the Oak Ridge National Laboratory (ORNL) supported by the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

Atchley, Scott et al. (2023). "Frontier: Exploring Exascale". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: Association for Computing Machinery, pp. 1–16. DOI: 10.1145/3581784.3607089.

Brewer, Wesley et al. (2024-11). "A Digital Twin Framework for Liquid-cooled Supercomputers as Demonstrated at Exascale". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. To be published. Atlanta, GA.

CATIA-Systems (2024). *CATIA-Systems/FMPy*. https://github.com/CATIA-Systems/FMPy. Published 2017.

Functional Mock-up Interface (n.d.). https://fmi-standard.org/. Accessed April 19, 2024.

Greenwood, Scott et al. (2017a-08). *A Templated Approach for Multi-Physics Modeling of Hybrid Energy Systems in Modelica*. Technical Report 10.2172/1427611. DOI. URL: https://www.osti.gov/biblio/1427611.

Greenwood, Scott et al. (2017b-09). *TRANSFORM - TRANsient Simulation Framework of Reconfigurable Models*. DOI: 10.11578/dc.20171025.2022. URL: https://www.osti.gov/biblio/1503596.

Kumar, Vineet et al. (2024-10). "Thermo-Fluid Modeling Framework for Supercomputing Digital Twins: Part 1, Demonstration at Exascale". In: *Proceedings of the America Modelica Conference*. Storrs, CT.

Modelica Asscotiation (2024-05). *VehicleInterfaces Library*. https://github.com/modelica/VehicleInterfaces.

National Research Council (2005). *The Future of Supercomputing–Conclusions and Recommendations*. Washington, DC: The National Academies Press. DOI: 10.17226/11148.

REDFISH | DMTF (n.d.). https://www.dmtf.org/standards/redfish. Accessed April 19, 2024.