ORNL/SPR-2021/2402
Revision 0

Uncertainty Quantification in High-Low Dynamic System Coupling Using RAVEN and TRANSFORM

March | 2022

Wesley C. Williams, Vineet Kumar, Dane de Wet, William Gurecky
Oak Ridge National Laboratory

IES
Integrated Energy Systems

ORNL/SPR-2021/2402
Revision 0

Uncertainty Quantification in High-Low Dynamic System Coupling Using RAVEN and TRANSFORM

Wesley C. Williams, Vineet Kumar, Dane de Wet, William Gurecky
Oak Ridge National Laboratory

March 2022

OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283

http://www.ies.inl.gov

Prepared for the
US DEPARTMENT OF ENERGY
OFFICE OF NUCLEAR ENERGY
Managed by UT-Battelle LLC
Contract DE-AC05-00OR22725

ABSTRACT

This report demonstrates new functionality and applications enabled by the development of high-fidelity to low-fidelity (high-low) coupling for system simulations. Additionally, this development allows further exploration of the capabilities of the Risk Analysis Virtual Environment (RAVEN), a novel software framework, in the performance of uncertainty quantification in these high-low coupled system models. The work builds on previous work on high-low coupling that utilized COBRA-TF (CTF), the high-fidelity subchannel analysis code, with a low-fidelity model built in the system analysis code TRANSFORM, utilizing the Functional Mock-up Interface (FMI). Steady-state and transient analysis examples using the high-low coupled models generated from CTF and TRANSFORM/FMI are investigated. The workflows for in-memory and out-of-memory coupling are shown with specific applications to nuclear systems' uncertainty and system characterization studies. This work elucidates some of the potential benefits and future needs of using RAVEN for high-low system coupling analysis of energy systems.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

BWR boiling water reactor

CASL Consortium for Advanced Simulation of Light Water Reactors

CFL Courant–Friedrichs–Lewy

CTF COBRA-TF

DOE US Department of Energy

FMI Functional Mock-up Interface

FMU Functional Mock-up Units

LWR light-water reactor

LWRS Light Water Reactor Sustainability

MSRE Molten Salt Reactor Experiment

NE Nuclear Energy Office

NEAMS Nuclear Energy Advanced Modeling and Simulation

ORNL Oak Ridge National Laboratory

PDF probability distribution (density) function

RAVEN Risk Analysis Virtual Environment

RISA Risk-Informed Systems Analysis

RISMC Risk-Informed Safety Margin Characterization

TRANSFORM Transient Simulation Framework of Reconfigurable Models

VERA Virtual Environment for Reactor Applications

1. INTRODUCTION

1.1 BACKGROUND

An aim of the US Department of Energy (DOE) Office of Nuclear Energy Integrated Energy Systems (IES) program is to develop increasingly coupled simulations of systems with a large diversity of scales and complexity to capture the physical behaviors of mixed electrical and thermal energy usage and storage systems. This desire to couple high- and low-fidelity (high-low) modeling has been an area of active research in the nuclear energy community for many decades and is receiving increased attention due to the potential of digital twins. A digital twin (DT) refers to an integrated multiphysics, multiscale, probabilistic simulation of the physical asset/twin, and it offers a framework to better quantify design margins, parameter uncertainty, and system performance associated with the physical object [2]. An increased computational capacity enabled the ability to bring a more intimate coupling of high-low systems between popular nuclear energy simulation codes. The recently developed software framework called the Risk Analysis Virtual Environment (RAVEN) enables increasingly flexible statistically driven analysis of traditional simulation codes and advanced new codes [3]. The potential use cases are rapidly growing as the capabilities of RAVEN increase. Therefore, it is of great interest to explore use cases of RAVEN for performing High/Low coupled simulations as a way to demonstrate the possibilities.

1.2 OVERVIEW OF RAVEN

RAVEN began in 2012 as a project to create a modernized risk evaluation framework as a part of the Risk-Informed Safety Margin Characterization (RISMC) work, now known as Risk-Informed Systems Analysis (RISA) [3]. These research paths were developed as a part of the Light Water Reactor Sustainability (LWRS) program at the DOE Nuclear Energy Office (NE). Initially, RAVEN was envisioned as a framework for driving dynamic risk assessment capabilities to existing traditional codes such as RELAP5; however, it was built in an agnostic format and has since gained the capability to couple with many standard nuclear and systems codes. The agnostic nature of RAVEN interfaces either directly to the code via software coupling or indirectly to the external code executables through input and output files for the codes. This workflow is described in the examples provided later in the report (2. and 3.). RAVEN contains the following core capabilities inside its framework as shown here [3]:

- **Distributions:** To allow for the exploration of input/output space of a system/physics, RAVEN has the capability to perturb the input space (i.e., initial conditions and/or model coefficients of a system). The input space is generally characterized by one or several probability distribution (density) functions (PDFs), which can be sampled depending on the kind of input desired. In this respect, a large library of PDFs is available.
- **Samplers:** A proper approach to sampling the input space is fundamental for optimizing the computational time. In RAVEN, a ‘sample’ employs a unique perturbation strategy that is applied to the input space of a system. The input space is defined through the connection of uncertain variables (i.e., initial conditions and/or model coefficients of a system) and their relative probability distributions. The link of the input space to the relative distributions allows the sampler to perform a probability-weighted exploration.

- **Optimizers:** Optimizers are tools for optimizing (constrained or unconstrained) the controllable input space (i.e., parameters) to minimize/maximize an objective function of the system/physics under examination. In RAVEN, an optimizer employs an active learning process (feedback from the underlying model, system, or physics) aimed to accelerate the minimization or maximization of an objective function.
- **Models:** A model is the representation of a physical system (e.g., nuclear power plant). Therefore, it can predict the evolution of a system given a coordinate set in the input space. Additionally, it can represent an action on a piece of data to extract key features (e.g., data mining).
- **Data Objects and Databases:** Data objects and databases provide standardized Application Program Interfaces (APIs) for storing the results of any RAVEN analysis (e.g., sampling, optimization, statistical analysis). Additionally, these storage structures represent the common "pipe network" among any entity in RAVEN.
- **Outstreams:** Outstreams export the results of any RAVEN analysis (e.g., sampling, optimization). This entity enables the exposition of analysis results to the user, both in text-based (e.g., XML, CSV) or graphical (e.g., pictures, graphs,) output files.
- **Steps:** Steps provide a standardized way for the user to combine the aforementioned entities to construct a particular analysis. Steps are the core of the calculation flow of RAVEN and are the only system that is aware of any simulation component.
- **Job Handler:** The job handler coordinates and regulate the dispatch of jobs in the RAVEN software. It can monitor and handle parallelism in the driven models to interact with high-performance computing and other similar systems.

This report demonstrates some of the major applications of these components for high-low coupling, as discussed later in the report (2. and 3.).

1.3 OVERVIEW OF CTF

COBRA-TF (CTF) is a subchannel thermal hydraulics code jointly developed by Oak Ridge National Laboratory (ORNL) and North Carolina State University as part of the Consortium for Advanced Simulation of Light Water Reactors (CASL) program and was built from the legacy code COBRA-TF [4]. CTF is based on a two-fluid, three-field (i.e., vapor, continuous liquid, and entrained droplets) semi-implicit modeling approach that can be solved either in a 3D Cartesian representation or by using a subchannel representation (axial and lateral directions). Under the CASL program, extensive capabilities were added into CTF to analyze light-water reactors (LWRs) for nominal operating conditions, departure from nucleate boiling (DNB), and system transients. Further development of CTF was undertaken to improve its modeling and simulation capabilities for boiling water reactors (BWRs) under the Nuclear Energy Advanced Modeling and Simulation (NEAMS) program. CTF was also incorporated into Virtual Environment for Reactor Applications (VERA), a multiphysics core simulator, and provides a coupling framework to couple various codes, such as thermal hydraulics, fuel performance, and neutronics. VERA is being developed in conjunction with CTF to model multiphysics applications such as crud-induced power shift (CIPS) and reactivity-insertion accidents (RIAs) [5].

1.4 OVERVIEW OF TRANSFORM AND THE FMI/FMU

TRANSFORM is a Modelica-based library developed at ORNL to enable rapid development of dynamic, advanced energy systems with an extensible system modeling tool [6]. TRANSFORM is organized as a series of packages, each of which has a general application. The object-oriented nature of Modelica allows users to view, extend, and/or modify any component or add new models to TRANSFORM. The TRANSFORM code, which is built upon the Modelica libraries, solves a time-dependent ordinary differential equations (ODEs) system based on a finite-volume–based staggered grid formulation that is applicable for single- and two-phase flows.

TRANSFORM can model lumped and 1D fluid mechanics, lumped and multi-dimensional heat and mass transfer, control logic and sensors, and simple power systems. It can also construct complex thermal hydraulics systems of mixed components, including nuclear and mechanical subsystems and components. These dynamic models can be used to study the dynamic interdependency of any and all system parameters.

The TRANSFORM Modelica library is currently implemented in the commercial Dymola dynamic modeling laboratory software environment by Dassault Systèmes. Dymola is a GUI that allows for visual drag-and-drop system modeling from the various standard Modelica libraries and from the imported TRANSFORM libraries. Dymola also can also perform text-based editing of models in the Modelica language. More importantly, Dymola provides the translator that converts the code listings of the Modelica models into a system of solvable equations. Once these are translated, Dymola provides a host of numerical time integration methods (e.g., DASSL, Runge–Kutta) that can be used to run model simulations or solutions. Dymola also enables enhanced flexibility in interfacing with other codes implementing the Functional Mock-up Interface (FMI). The FMI defines a standard interface to an exchangeable package that contains a ODE-based system model of a component, or set of coupled components, called Functional Mock-up Units (FMU) [7]. Dymola translates the Modelica models into C-code, and can compile the FMU into a binary format that fully encapsulates the model into a simulator that can be imported for use in a larger system. The FMU then behaves as a standardized black-box simulator that calculates simulated dynamic outputs driven by user-supplied inputs. These inputs and outputs can be driven and accessed by other codes through the standardized interface defined by the FMI, as in this project with CTF or RAVEN.

1.5 COUPLING OF CTF AND TRANSFORM USING THE FMU

Recent work generated an in-memory coupling between the subchannel thermal hydraulics code CTF—which is included in the VERA—and the systems code TRANSFORM by utilizing the FMI standard to create a coupling of the models [1]. The coupling enables an exchange of boundary conditions, run-time setting of system parameters and may be used in a steady state or transient coupled simulation mode. This coupling allows one to trace uncertainties and sensitivities through a range of fidelities and scales, with the fine-fidelity core model supplied by CTF and are coarse system model of the primary, secondary, and tertiary loop supplied by TRANSFORM, which is helpful for detailed investigation of integrated energy systems that would not be possible with a simple core-simulator operating as a standalone core model.

Figure 1 shows a simplified MSRE model example of a potential high-low model from a previous work [1] built by coupling VERA/CTF to the TRANSFORM model. The simplified core of the MSRE reactor was simulated in VERA/CTF, whereas the components external to the reactor core vessel—which includes the primary, secondary, and tertiary components—were modeled in TRANSFORM. Fortunately, only a few

typical parameters are required to be coupled between high-low system models. In this example, only in-memory coupling of six key boundary condition parameters is required. 1 provides the six boundary condition parameters and the direction of the exchange between codes. The interface was achieved through an FMU representation of the TRANSFORM model compiled into an FMU (using the Dymola software package), which was then accessed by CTF with a FORTRAN-composed FMU wrapper. The wrapper enables users to load and interact with co-simulation FMUs that conform to the FMI version 2 standard from a FORTRAN program. The developed wrapper is accessible as an open-source addition to the Futility library [8]. The FMU includes the ODE solver from Dymola for the simulation, called the *co-simulation model*.

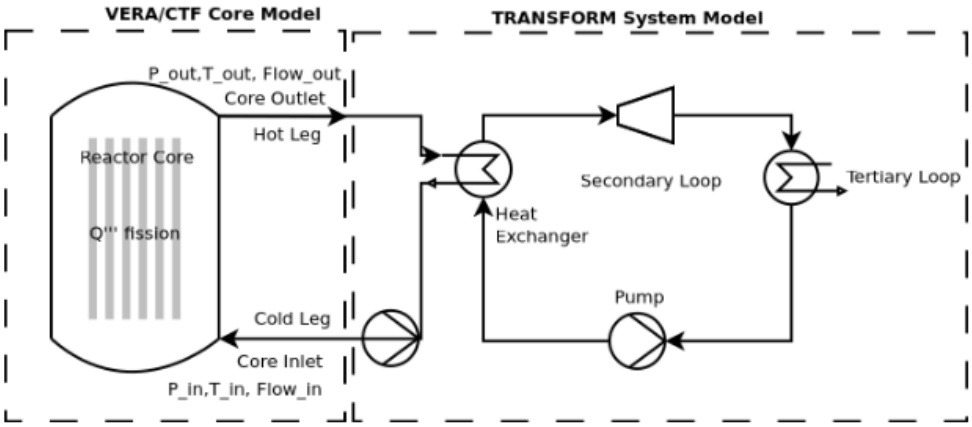


Figure 1. Example of a coupled CTF core model and TRANSFORM FMU system model.

Table 1. Boundary conditions exchanged at core inlet and outlet interface [1].

Variable	Unit	Note	Direction
T_{in}	K	Core inlet temperature	VERA/CTF ← TRANSFORM
P_{in}	Pa	Core inlet static pressure	VERA/CTF → TRANSFORM
$\dot{m}_{in}$	kg/s	Core inlet flow rate	VERA/CTF ← TRANSFORM
T_{out}	K	Core outlet temperature	VERA/CTF → TRANSFORM
P_{out}	Pa	Core outlet static pressure	VERA/CTF ← TRANSFORM
$\dot{m}_{out}$	kg/s	Core outlet flow rate	VERA/CTF → TRANSFORM

As previously mentioned, the MSRE model used in the study is similar to that from a previous work [1]. The TRANSFORM model comprises MSRE piping, pumps, heat exchangers, and heat rejection; the nodalization is shown in 2. The focus of the model developed in Gurecky et al. [1] is not to validate against experimental measurements but to demonstrate the coupled model capabilities of simulating the complex system dynamics and to perform uncertainty propagation, which is the focus of the current study.

A coupled CTF-FMU model requires the following [1]:

1. a CTF input deck,
2. the FMU zip archive extracted to the directory, and

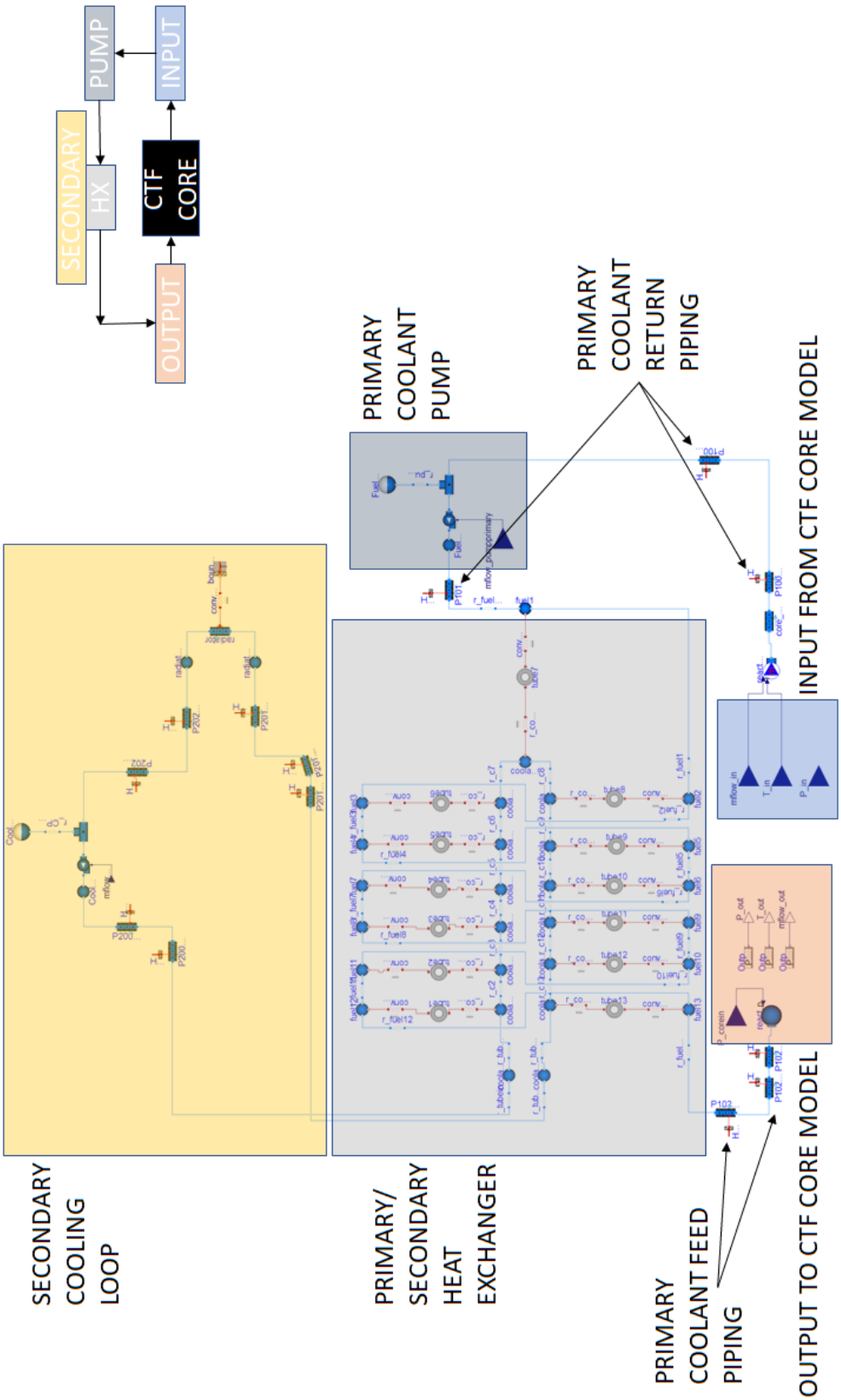


Figure 2. TRANSFORM nodalization of the MSRE external reactor core vessel components.

3. an XML coupling specification file.

An example of a coupling specification file for the coupled CTF–TRANSFORM model of the MSRE model is shown below. The file name should be the same as that specified in the caption, and it should be placed in the working directory. The coupling file contains the coupled solver tolerances, initial values of the exposed variables, boundary mappings between the codes, FMU variables that vary as a function of time, and FMU variables to log to file. A detailed description of the XML coupling specifications file can be found in Gurecky et al. [1]. For brevity, only BC_VAR_NAMES, which contains the mapping from the CTF boundary conditions, is described in detail below in Listing 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<ParameterList name="CASEID">
    <Parameter name="toltemp" type="double" value="0.01"/>
    <Parameter name="tolmf" type="double" value="2.0"/>
    <Parameter name="tolpress" type="double" value="1.0e1"/>
    <Parameter name="toltemp_FMU" type="double" value="1.0e-4"/>
    <Parameter name="tolmf_FMU" type="double" value="1.0e-4"/>
    <Parameter name="tolpress_FMU" type="double" value="1.0e-4"/>
    <Parameter name="FMU_dt_max" type="double" value="2.0e-1"/>
    <Parameter name="ulax_T_corein" type="double" value="1.0"/>
    <Parameter name="ulax_P_coreout" type="double" value="1.0"/>
    <Parameter name="ulax_mflow_corein" type="double" value="1.0"/>
    <ParameterList name="FMU_VAR_INIT">
       <!-- Set FMU Parameters and inital values -->
       <Parameter name="P_in" type="double" value="101.33e3"/>
       <Parameter name="P_corein" type="double" value="101.33e3"/>
       <Parameter name="T_in" type="double" value="907.0"/>
       <Parameter name="mflow_in" type="double" value="171.0"/>
       <Parameter name="mflow_pumpprimary" type="double" value="171.0"/>
       <Parameter name="mflow_secondary" type="double" value="105.745"/>
    </ParameterList>
    <ParameterList name="BC_VAR_NAMES">
       <!-- Parameter name= CTF_name     value= FMU_name -->
       <Parameter name="T_corein" type="string" value="T_out"/>
       <Parameter name="T_coreout" type="string" value="T_in"/>
       <Parameter name="P_corein" type="string" value="P_corein"/>
       <Parameter name="P_coreout" type="string" value="P_in"/>
       <Parameter name="mflow_corein" type="string" value="mflow_out"/>
       <Parameter name="mflow_coreout" type="string" value="mflow_in"/>
       <Parameter name="mflow_pumpprimary" type="string" value="mflow_pumpprimary"/>
    </ParameterList>
    <ParameterList name="FMU_VAR_TRANSIENT">
       <!-- FMU vars that vary as a fn of time -->
       <Parameter name="time" type="Array(double)" value="{0,10,100}"/>
       <Parameter name="mflow_secondary" type="Array(double)" value="{80,90,105.7}"/>
    </ParameterList>
    <ParameterList name="FMU_VAR_LOG">
       <!-- FMU Variables to log to file -->
       <Parameter name="mflow_pumpprimary" type="bool" value="true"/>
       <Parameter name="mflow_secondary" type="bool" value="true"/>
       <Parameter name="T_in" type="bool" value="true"/>
       <Parameter name="T_out" type="bool" value="true"/>
    </ParameterList>
```

```
</ParameterList>
```

Listing 1. XML coupling specification [1] `fmu_param.xml`

- Parameter **T_corein** (required): name of the FMU variable corresponding to the core inlet temperature in Kelvin.
- Parameter **T_coreout** (required): name of the FMU variable corresponding to the core outlet temperature in Kelvin.
- Parameter **P_corein** (required): name of the FMU variable corresponding to the core inlet pressure in pascals.
- Parameter **P_coreout** (required): name of the FMU variable corresponding to the core outlet pressure in pascals.
- Parameter **mflow_corein** (required): name of the FMU variable corresponding to the core inlet mass flow rate in kilograms per second.
- Parameter **mflow_coreout** (required): name of the FMU variable corresponding to the core outlet mass flow rate in kilograms per second.

A separate parameter is defined in the CTF input file under Card 1.5, in which the path of the compiled FMU object should be specified. The FMU should be extracted into the (relative to working directory or absolute) path specified under {**fmu_unzip_directory**} before running the CTF executable with the input file. The shared object libraries and an XML model description file—which are generated during the FMU creation—also reside inside the FMU folder [1]. A snippet of the CTF input file for the simplified MSRE model showing {**fmu_unzip_directory**} is given below in Listing 2.

```
*Card 1.3
**          PREF             HIN            HGIN          VFRAC1          VFRAC2
         3.44738     -633.88889    288.4200000       1.0000000       0.9999000
*Card 1.4
**GTP(1)   VFRAC(3)  GTP(2) VFRAC(4)  GTP(3) VFRAC(5)  GTP(4) VFRAC(6)
     air     0.0001
*Card 1.5
     {fmu_unzip_directory}  "../../transform_fmus/MSRE_HX_wPiping_Full_v7__wFixedAlpha
    "
********************************************************************************
*GROUP 2 - Channel Description
********************************************************************************
**NGR
    2
*Card 2.1
**  NCH NDM2 NDM3 NDM4 NDM5 NDM6 NDM7 NDM8 NDM9 NM10 NM11 NM12 NM13 NM14
   1    0    0    0    0    0    0    0    0    0    0    0    0    0
*Card 2.2
**   I           AN           PW ABOT ATOP NMGP            X            Y         XSIZ
       YSIZ
     1  1.6360e-01  4.1286e+01  0.0  0.0    0  0.0000e+00  0.0000e+00  0.0000e+00
    0.0000e+00
```

Listing 2. CTF input file snippet with the FMU path specified. `system_coupling_transform.inp`

2. STEADY-STATE ANALYSIS OF HIGH-LOW COUPLING IN RAVEN

This section describes the workflow for performing a steady-state co-simulation of the coupled system. The workflow description is followed by two use case demonstrations: parameter estimation and uncertainty analysis. The goal is to demonstrate common use cases for RAVEN in steady-state analysis.

2.1 OVERVIEW OF RAVEN WORKFLOW FOR STEADY-STATE SIMULATIONS

The RAVEN workflow for the CTF–FMU co-simulation is shown in Figure 3. RAVEN has two preferential Python-based APIs for interacting with external applications: 'External Model' and 'External Code' [3]. Both the approaches were used in this study. In the External Model approach, an external Python "entity" is used that can act as a system model, whereas in the External Code approach, an API is used to drive external codes at run time. The External Code approach in this study is based on an existing CTF code coupling interface with two modifications, which are discussed in this section.

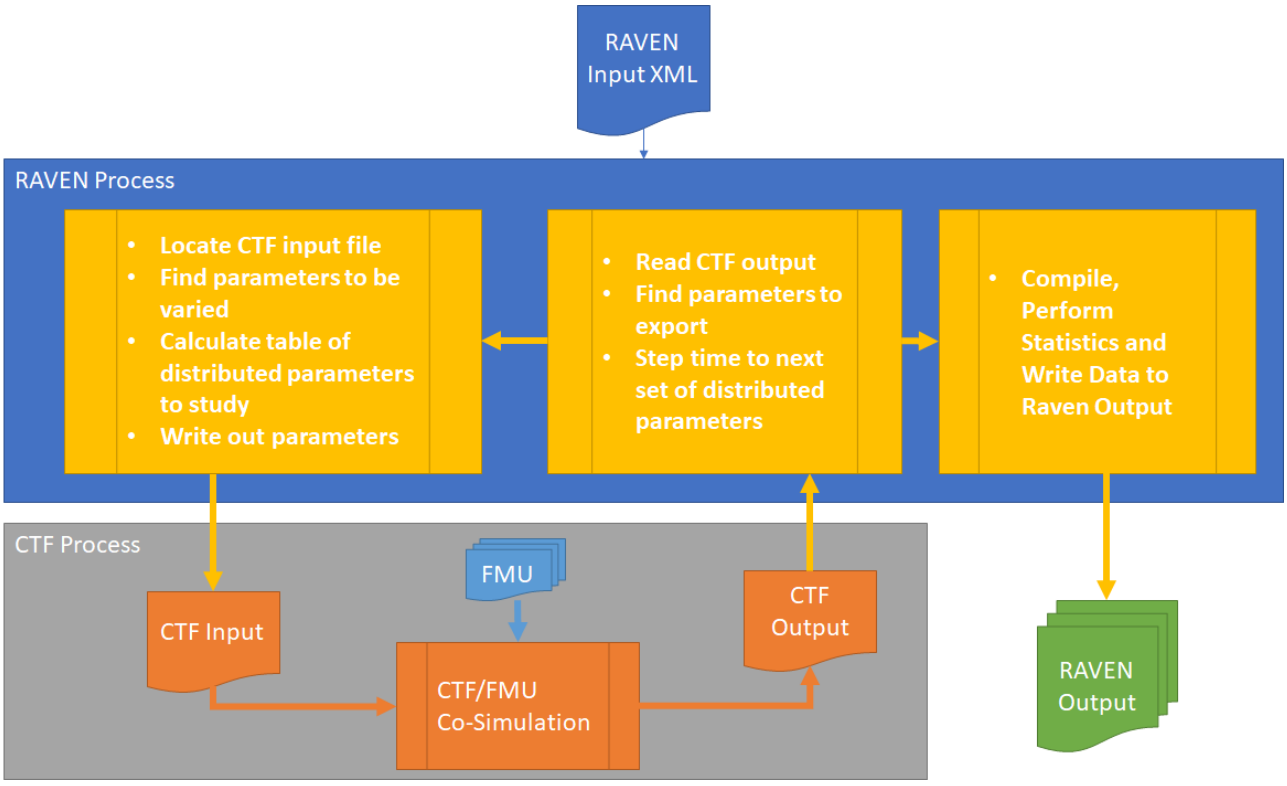


Figure 3. Generalized RAVEN workflow with CTF and TRANSFORM FMU system model coupling.

The workflow in 3 can be enumerated as follows.

1. Search for the appropriate CTF input file(s).
2. Find the parameters to be varied.
3. Write new input file(s) with the sampled variables passed from the RAVEN framework.
4. Perform simultaneous runs with the sampled variables.

5. Read the CTF output for each set of sampled variable(s) and export required output parameters.

6. Obtain RAVEN output, which includes generating data files of the output parameters, generating plots, and compiling statistics.

The code coupling interface can interpret the information from RAVEN and pass the required input to the driven code [3]. The existing CTF code interface can accomplish most of the above steps by:

- using a 'GenericCode' interface to look for parameters to be varied in the CTF input files whose values are passed from the RAVEN framework,
- writing new input files with the substituted variables using the GenericCode interface, and
- writing a Comma-Separated Value (CSV)–based RAVEN output file by reading the generated standard CTF output file for each set of sampled variable(s).

Because the code interface drives only the CTF code, the in-memory CTF–FMU coupling approach was used to run the coupled CTF–FMU simplified MSRE model. The steady-state coupling algorithm for the in-memory coupling approach is shown below in Algorithm 1. For the steady-state model, alternating pseudo-transient calculations of CTF and the transform FMU models were performed using a fixed-point iteration scheme [1]. In the algorithm, x is the vector of temperatures, pressures, and mass flow rates, at the core inlet and outlet; θ_{CTF} are the CTF model parameters; θ_{FMU} are the FMU model parameters; and i is the outer iteration number.

ALGORITHM 1
Solution strategy for steady-state CTF-FMU (in-memory) coupling [1]

Initialization
(1) Set maximum number of outer iterations, N.
(2) Set under relaxation factors, $\omega \in (0, 1]$. Default $\omega = 1$.
(3) Set outer loop convergence tolerance. Default $\varepsilon \cong 0.25$K.
(4) Supply initial guess for $\mathrm{x}_0 = \{T_{0,in}, \dot{m}_{0,in}, P_{0,out}, ...\}$
(5) Initialize CTF and FMU from input
for Outer step: i in $\{0, ...N\}$ **do**
    Execute a pseudo-transient CTF computation, given: x_i:
    $\tilde{\mathrm{x}}_{i+1} \leftarrow \mathscr{G}_{CTF}(\mathrm{x}_i, \theta_{CTF})$
    Execute a pseudo-transient FMU computation:
    $\hat{\mathrm{x}}_{i+1} \leftarrow \mathscr{F}_{FMU}(\tilde{\mathrm{x}}_{i+1}, \theta_{FMU})$
    Update the state vector with under relaxation
    $\mathrm{x}_{i+1} = \omega\hat{\mathrm{x}}_{i+1} + (1 - \omega)\hat{\mathrm{x}}_i$
    if $|\mathrm{x}_{i+1} - \mathrm{x}_i| < \varepsilon$ **then**
        Break
    end if
end for

Two modifications were made to the existing CTF code coupling interface in RAVEN. The first pertains to the output parser. The existing interface reads the '.out' output file, which is a legacy format, and thus a new output parser was written to read the hdf5 output file. A method in the CTF CodeInterface class was

modified to reflect the change to reading the hdf5 file when generated, and the relevant code is shown below in Code 3.

```
  def finalizeCodeOutput(self,command,output,workingDir):
    """
      This method is called by the RAVEN code at the end of each code run to create
    CSV files containing the code output results.
      @ In, command, string, the command used to run the just ended job
      @ In, output, string, the Output name root
      @ In, workingDir, string, current working dir
      @ Out, response, dict, dictionary containing the data {var1:array, var2:array,
    etc}
    """
    if os.path.isfile(os.path.join(workingDir,output+'.native.h5')):
        outfile  = os.path.join(workingDir,output+'.native.h5')
        outputobj= ctfdataHDF5(outfile)
    else:
        outfile  = os.path.join(workingDir,output+'.out')
        outputobj= ctfdata(outfile)
    response = outputobj.returnData()
    return response
```

Listing 3. Snippet of the modifications to the CTF interface module in RAVEN. `CTFinterface.py`

A snippet of the new hdf5 output parser is shown in Listing 4. The current version of the code performs an area averaging across all channels and provides key output variables as a CSV file for only the initial and final states. Furthermore, the current hdf5 output parser is restricted to single-section models. A future update will include support for multi-section models and expand the list of output variables. The hdf5 output parser uses methods from an external hdf5 module, which was developed as part of the CTF SubKit package [9].

```
class ctfdataHDF5:
  """
    Class that parses CTF output file and reads in (output files type: .ctf.out) and
    write a csv file
  """
  def __init__(self, filein):
    """
      Constructor
      @ In, filen, string, file name to be parsed
      @ Out, None
    """
    # check file existence
    if ("ctf.native.h5" not in filein):
      raise IOError(
            "Check if the supported hdf5 output file (*.ctf.native.h5) is included.")

    self.majorData, self.headerName = self.getData(filein)

  def returnData(self):
    """
      Method to return the data in a dictionary
```

```
      @ In, None
      @ Out, data, dict, the dictionary containing the data {var1:array,var2:array,etc
    }
    """
    return data

  def testConverged(self, convFile):
    f = open(convFile)
    convergedString = 'Steady state reached'
    if f.read().find(convergedString)>0:
        return True
    else:
        return False

  def getData(self, h5File):
    """
      Method that reads the hdf5 file
      @ In, string, output file (.ctf.native.h5)
      @ Out, (dictArray, header), tuple, tuple containing:
                                        -[0] -> dictionary containing the edit info
                                        -[1] -> header
    """
    h5 = Hdf5Tools.Hdf5Interface(h5File)
```

Listing 4. Snippet of a new CTF HDF5 file parser for the CTF interface module in RAVEN. `ctfdataHDF5.py`

An example of a sampled variable to be substituted into the CTF input file is shown in Listing 5. The sampled variable is defined as follows: $RAVEN-SampledVar$, where 'SampledVar' is the sampled variable name as defined in the RAVEN XML input file. The 'GenericCode' parser—the default option for searching for the sampled variables in the CTF input file(s)—required a modification to accommodate fractional sampled variable parameter values, say, as a percentage of the nominal input parameter.

```
*****************************************************************************
*GROUP 1 - Calculation Variables and Initial Conditions
*****************************************************************************
**NGR
   1
**NGAS  IRFC  EDMD  IMIX  ISOL          GINIT  NOTRN  MESH  MAPS  IPRP  MFLX  IBTM  PPV   NM14
     1     2     0     3     0     171.0*$RAVEN-priMF$+171.0      1     1     0     4     0     0
       7     0
*Card 1.2
**          GTOT           AFLUX          DHFRAC
      171.0*$RAVEN-priMF$+171.0         6.40020         0.99990
*Card 1.3
**          PREF             HIN            HGIN          VFRAC1          VFRAC2
        3.44738      -633.88889     288.4200000       1.0000000       0.9999000
```

Listing 5. CTF input file snippet with the $ RAVEN$ flag for variable substitution from the RAVEN framework. `system_coupling_transform.inp`

The input parser was modified to allow for a pre/post addition/multiplication of the sampled variable in the text-based CTF input file. The current version checks only for the previous or the next character to the

sampled variable and only allows for pre/post addition and multiplication. A future version will be expanded to allow for more complex operations. A code snippet of the modified input parser showing the functions that perform the pattern search is shown below in Listing 6.

```
def addPrefixFloat(keyType, start, line, segments, infileName, var):
    """
      Identifies any floats to be added/multiplied before an input variable
      @ In, keyType, str, the operator type
      @ In, start, int, starting position of input variable in the current line being parsed in the input file
      @ In, line, str, current line being parsed in the input file
      @ In, segments, list, list of parsed line(s) segments
      @ In, infileName, str, input file name
      @ In, var, str, input variable name
      @ Out, segments, list, modified list of parsed line(s) segments
      @ Out, start, int, modified starting position
    """
    endMult = start-1
    index = endMult-1
    while True:
      if index >= 0 and re.match(r'^[+-]|[.]|\d|e|E$',line[index]) is not None:
        index = index - 1
      else:
        break
    startMult = index + 1
    start = start - (endMult - startMult) - 1
    segments[infileName].append(line[:start])
    if keyType == '+':
      segments[infileName].append('ADD')
    if keyType == '*':
      segments[infileName].append('MULT')
    segments[infileName].append(line[startMult:endMult])
    segments[infileName].append(var)
    return (segments, start)

def addPostfixFloat(keyType, flag, start, end, line, segments, infileName, var):
    """
      Identifies any floats to be added/multiplied after an input variable
      @ In, keyType, str, the operator type
      @ In, flag, int, flag to indicate if input variable has been added or not
      @ In, start, int, starting position of input variable in the current line being parsed in the input file
      @ In, end, int, end position of input variable in the current line being parsed in the input file
      @ In, line, str, current line being parsed in the input file
      @ In, segments, list, list of parsed line(s) segments
      @ In, infileName, str, input file name
      @ In, var, str, input variable name
      @ Out, segments, list, modified list of parsed line(s) segments
      @ Out, end, int, modified end position
    """
    startMult = end+1
    index = startMult+1
    while True:
```

```
      if index <= len(line) and re.match(r'^[+-]|[.]|\d|e|E$',line[index]) is not None
    :
        index = index + 1
      else:
        break
    endMult = index - 1
    end = end + (endMult - startMult) + 1
    if flag != 1:
      segments[infileName].append(line[:start])
      segments[infileName].append(var)
    if keyType == '+':
      segments[infileName].append('ADD')
    if keyType == '*':
      segments[infileName].append('MULT')
    segments[infileName].append(line[startMult+1:endMult+1])
    return (segments, end)
```

Listing 6. Snippet of the modified generic input parser for the CTF code interface module in RAVEN flag for pre/post addition and multiplication of sampled variables. `CTFParser.py`

To use the code coupling interface in the RAVEN input file, the **<Code>** XML node must be defined in the Model block with the attribute 'subType', which takes the value of "CTF", as shown below in Listing 7 [3]. The path of the CTF executable is specified in a child node, **<executable>**. Under the **<Files>** XML node, the CTF input file must be specified with the '.inp' extension. In the current study, an additional thermophysical properties file was used to define the salt properties, and an 'fmu_param.xml' XML file was used to define the coupling between CTF and the exposed FMU parameters, shown in Listing 1.

```
<RunInfo>
  <WorkingDir>Testfmu1</WorkingDir>
  <Sequence>testRun</Sequence>
  <batchSize>25</batchSize>
  <NumThreads>1</NumThreads>
  <expectedTime>1:00:00</expectedTime>
  <CustomMode class="MPIEXECSimulationMode" file="%BASE_WORKING_DIR%/mpi_custom.py">
  mpicust</CustomMode>
  <mode>mpicust</mode>
</RunInfo>

<Files>
  <Input name="cobra_input" type="ctf" >system_coupling_transform.inp</Input>
  <Input name="thermophy" type="thermophy" >thermophysical_properties.dat</Input>
  <Input name="fmu_param" type="fmu_param" >fmu_param.xml</Input>
</Files>

<Models>
  <Code name="MyCobraTF" subType="CTF">
    <executable>
      "/home/vk8/build/install/bin/cobratf"
    </executable>
    <csv>True</csv>
  </Code>
```

```
  </Models>
```

Listing 7. RAVEN input file snippet showing the models block using the existing CTF code interface. `test1_ctf_fmu.xml`

The second approach to running an external application with RAVEN is by using an External Model. An *External Model* is an entity that is embedded in the RAVEN framework at run time via a Python module, which is treated as a predefined internal model object [3]. The External Model paradigm provides the ability to interact directly with a user-supplied python module which encapsulates arbitrary physics models and IO routines. This allows for rapid prototyping in cases where a large fixed, compiled, executable program may not be available, but a more flexible python-based physics model is available.

To flex this External Model RAVEN capability, an out-of-memory coupling between the TRANSFORM FMU and CTF was developed aside from the previously developed in-memory coupling approach [1]. This out-of-memory coupling is effectively a python driver script to automate the execution of CTF and TRANSFORM in a coupled manner via boundary condition exchange via file input/output.

One advantage in using an external python based out-of-memory code coupling rather than the in-memory coupling approach is the flexibility that it affords in incorporating powerful third party machine learning (ML) and data processing libraries into the model. The in-memory coupling approach is not well suited to rapidly prototype the inclusion of such ML tools within the FMU or CTF model. The Python script–based coupling code is listed in Appendix A. There are two Python modules: '**fmuCTF.py**' and '**fmuCTFCouple.py**'. The former module is loaded with the RAVEN input file and contains the methods to interface with the RAVEN framework. The latter module contains the procedures to drive the CTF and FMU code. The FMPy package is used to load and run the transform FMU in the script [10]. The steady-state coupled CTF-FMU algorithm is based on a fixed-point scheme, similar to the in-memory coupled algorithm (1).

To use the External Model interface in the RAVEN input file, the **<ExternalModel>** XML node must be defined in the Model block with the attribute 'ModuletoLoad' which in the present case is 'fmuCTF', as shown below in Listing 8 [3]. The path of the CTF executable is specified in the child node, **<executable>**, and the variables that are input/output to the external module are specified under the child node, **<variables>**. Finally, under the **<Files>** XML node, the python scripts are specified along with supplementary files, similar to the code interface coupling approach. One difference between the in-memory and python scripting approach in terms of code predictions is that the predicted temperatures for the python scripting approach were consistently biased lower by $\sim 1\ ^{\circ}\mathrm{K}$ (not shown here). It must be noted that all the cases discussed henceforth are based on the in-memory coupling approach using the CTF code interface in RAVEN.

```
  <RunInfo>
    <WorkingDir>Testfmu1</WorkingDir>
    <Sequence>testRun</Sequence>
    <batchSize>1</batchSize>
  </RunInfo>

  <Files>
    <Input name="fmuCTF.py" type="">fmuCTF.py</Input>
    <Input name="fmuCTFCouple.py" type="">fmuCTFCouple.py</Input>
    <Input name="make_deck.py" type="subKit">make_deck.py</Input>
```

```
    <Input name="thermophysical_properties.dat" type="" >thermophysical_properties.dat
    </Input>
    <Input name="fmu_param.xml" type="" >fmu_param.xml</Input>
  </Files>

  <Models>
    <ExternalModel ModuleToLoad="fmuCTF" name="PythonModule" subType="">
      <variables>HEAT,AVG_ax21_chan_temp</variables>
      <executable>
        /home/vk8/build/install/bin/cobratf
      </executable>
      <fmuDir>transform_fmus</fmuDir>
    </ExternalModel>
  </Models>
```

Listing 8. RAVEN input file snippet showing the models block with an external code coupling interface. `test1_ctf_fmu_ext.xml`

2.2 PARAMETER SWEEP OF PRIMARY FLOW RATES

The first case shown here is a parameter sweep of primary pump flow rates using the steady-state coupled CTF–FMU algorithm of the simplified MSRE. The details of the nominal boundary conditions for the case are discussed in detail in Gurecky et al. [1] and are not repeated here. The advantage of using RAVEN to conduct this sweep is the relative ease of using multiple samplers and distributions for various sampled variables and generating new input files, as previously discussed. A snippet of the RAVEN input file for this case is shown in Listing 9. Twenty samples of the primary pump mass flow rate (fraction), **'priMF'**, were drawn from a uniform distribution with a grid sampler with a ±25% variation with respect to the nominal mass flow rate.

```

  <Distributions>
    <Uniform name="dist_for_priMF">
      <lowerBound>-0.25</lowerBound>
      <upperBound>0.25</upperBound>
    </Uniform>
  </Distributions>

  <Samplers >
    <Grid name="grid">
      <variable name="priMF">
        <distribution>dist_for_priMF</distribution>
        <grid construction="equal" steps="20" type="value">-0.25 0.25</grid>
      </variable>
    </Grid>
  </Samplers>
```

Listing 9. RAVEN input file snippet showing the sampler and distribution for a parameter sweep of primary pump mass flow rates. `test1_ctf_fmu.xml`

The results of the parameter sweep of primary pump flow rates are shown in Figure 4. The nominal result compares well with the actual MSRE operating condition [1]. As expected with an increase/decrease in the

primary pump mass flow rate and with a constant heated rod power, the enthalpy change across the core decreases/increases correspondingly. All the plots generated in this study were generated using external Python scripts. However, advanced plotting tools are available within the RAVEN framework which will be explored in a future study.

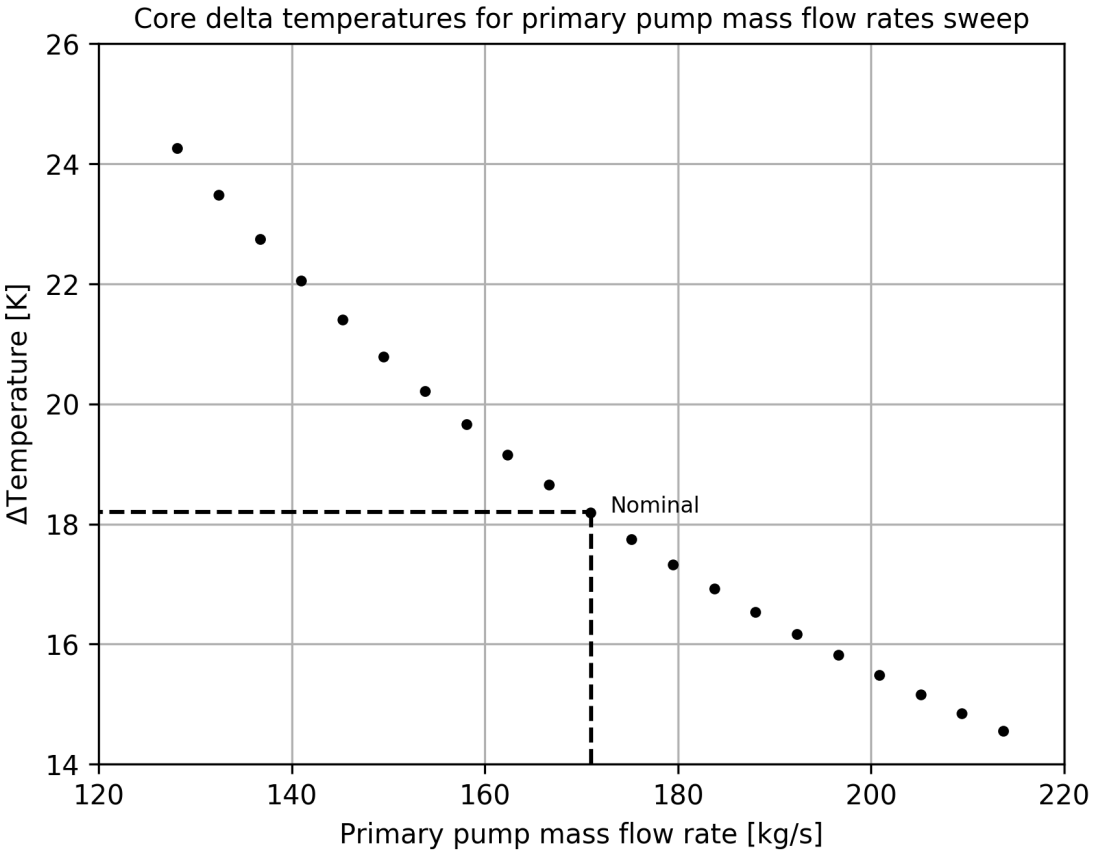


Figure 4. Change in core delta temperatures vs. primary pump flow rates for a parameter sweep of primary pump flow rates of a coupled CTF core model and TRANSFORM FMU system model.

2.3 UNCERTAINTY ANALYSIS OF REACTOR POWER AND PRIMARY FLOW RATES STEADY STATES

An uncertainty analysis of both the heated rod power and the primary pump mass flow rates was conducted using the steady-state coupled approach. These variables were selected for treatment as sampled random variables because they were exposed by the FMU model and easily accessible, in the future other model parameters, such as heat exchanger convective heat transfer coefficients, can be targeted as part of a forward propagation of uncertainty study. A snippet of the RAVEN input file for this case is shown below in Listing 10. Two hundred samples of a combined set of heated rod power (fraction), **'powerFrac'**, and the primary pump mass flow rate (fraction), **'priMF'**, were drawn from individual normal distributions with a Monte Carlo sampler with a ±20% variation relative to the nominal values.

```
<Distributions>
  <Normal name="dist_for_powerFrac">
    <mean>0.0</mean>
    <sigma>0.05</sigma>
    <upperBound>0.2</upperBound>
    <lowerBound>-0.2</lowerBound>
  </Normal>
  <Normal name="dist_for_priMF">
    <mean>0.0</mean>
    <sigma>0.05</sigma>
    <upperBound>0.2</upperBound>
    <lowerBound>-0.2</lowerBound>
  </Normal>
</Distributions>

<Samplers >
  <MonteCarlo name="MC_samp">
    <samplerInit>
      <limit>200</limit>
    </samplerInit>
    <variable name="powerFrac">
      <distribution>dist_for_powerFrac</distribution>
    </variable>
    <variable name="priMF">
      <distribution>dist_for_priMF</distribution>
    </variable>
  </MonteCarlo>
</Samplers>
```

Listing 10. RAVEN input file snippet showing the sampler and distribution for an uncertainty analysis of reactor power and primary pump mass flow rates. `test2_ctf_fmu.xml`

The results of the uncertainty analysis of reactor power and primary pump flow rates are shown in Figure 5. The system response can be well characterized through this study, especially if the CTF model were also coupled to a neutronics code. Even in the absence of a neutronics solver, the Gaussian density distribution indicates the expected region of operation in terms of the enthalpy change and the primary pump mass flow rate.

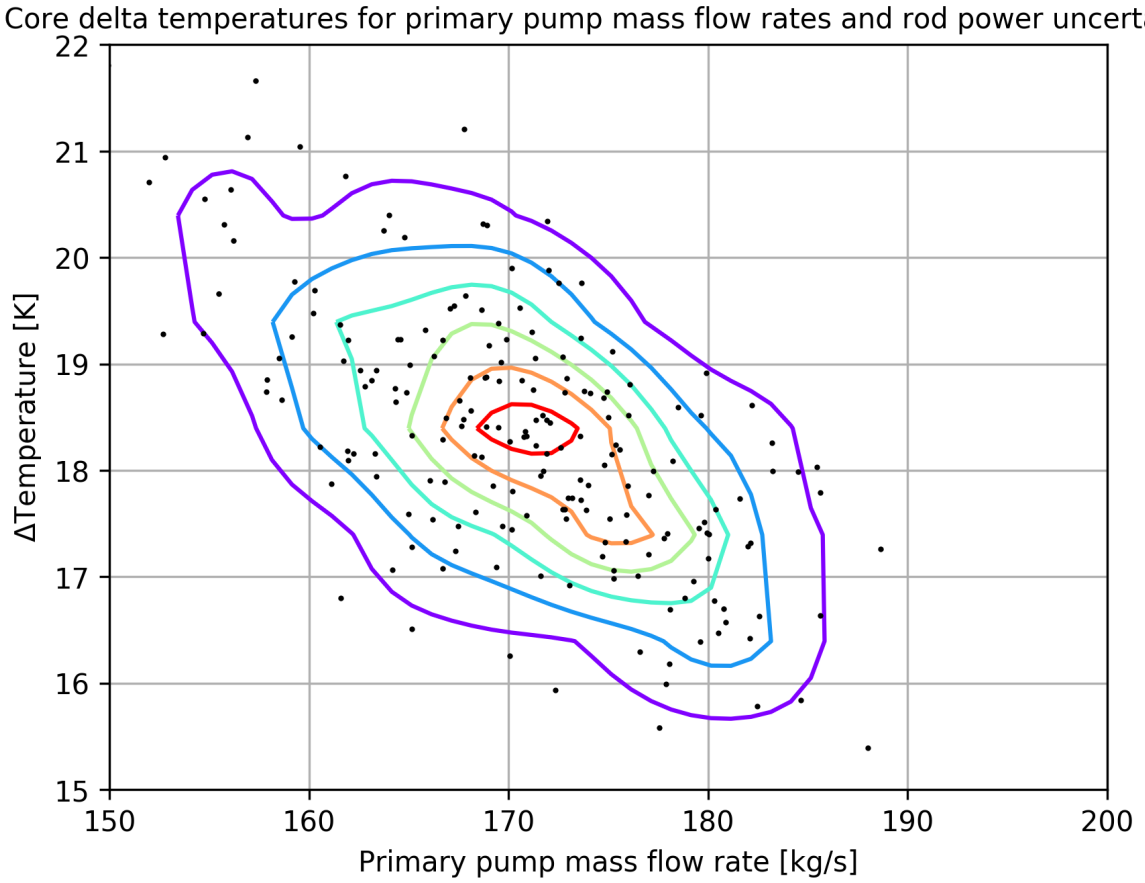


Figure 5. Change in core delta temperatures vs. primary pump flow rates (with Gaussian density contours overlaid) for an uncertainty analysis of reactor power and primary pump flows rates of a coupled CTF core model and TRANSFORM FMU system model.

3. TRANSIENT ANALYSIS OF HIGH-LOW COUPLING IN RAVEN

This section describes the workflow for performing a transient co-simulation of the coupled system. The workflow description is followed by three use case demonstrations: a pump trip sensitivity study, a power ramp sensitivity study, and a periodic perturbation study with frequency analysis. The goal is to demonstrate common use cases for RAVEN in transient analysis.

3.1 OVERVIEW OF RAVEN WORKFLOW FOR TRANSIENT SIMULATIONS

The transient coupled CTF–FMU cases in RAVEN were run in a manner like the steady-state cases using a similar code interface coupling approach. The 'GenericCode' input parser also allows for sampling transient FMU variables, such as the secondary pump mass flow rate, via the **'fmu_param.xml'** file. The transient coupling algorithm is discussed in detail in Gurecky et al. [1]. The transient method, given by algorithm 2, begins by specifying an initial condition and running a steady-state solve, similar to algorithm 1. Once the steady-state solve is completed, the transient CTF solve begins by executing one CTF time step, where the time step size is calculated internally based on a Courant–Friedrichs–Lewy (CFL) stability criteria. The FMU solver then executes a series of smaller substeps to avoid unstable behavior within the FMU model. It should be noted that since the FMU is used in a Co-Simulation mode, the FMU is responsible for implementing its own internal ODE stepping method that could be explicit depending on the FMU implementation. Once the FMU solver has reached the same solution time, the CTF solver advances, and the cycle repeats until the transient is completed. Five transient coupled cases were conducted, starting with a pump trip case.

ALGORITHM 2
Solution strategy for transient CTF–FMU (in-memory) coupling [1]

Initialization
(1) Set the CTF and FMU time steps, dt.
(2) Supply initial guess for $\mathrm{x}_0 = \{T_{0,in}, \dot{m}_{0,in}, P_{0,out}, ...\}$.
(3) Initialize CTF and FMU from input.
(4) Perform an initial steady-state calculation (based on Listing 1).
while $\mathrm{t_{CTF}} <= \mathrm{T}$ **do**
    Set the FMU solution time to previous CTF time for sub-stepping:
    $\mathrm{t_{FMU}} = \mathrm{t_{CTF}}$
    Execute a transient CTF computation, given: $\mathrm{x}_{t_{\mathrm{CTF}}}$:
    $\tilde{\mathrm{x}}_{t+dt} \leftarrow \mathscr{G}_{CTF}(\mathrm{x}_{t_{\mathrm{CTF}}}, \theta_{CTF})$
    Step the transient time step of the CTF solve:
    $\mathrm{t_{CTF}} = \mathrm{t_{CTF}} + \mathrm{dt_{CTF}}$
    while $\mathrm{t_{FMU}} <= \mathrm{t_{CTF}}$ **do**
        Execute a transient FMU computation using the interpolated transient CTF solution:
        $\mathrm{x}_{t+dt} \leftarrow \mathscr{F}_{FMU}(\tilde{\mathrm{x}}_{t_{\mathrm{FMU}}+dt_{\mathrm{FMU}}}, \theta_{FMU})$
        Step the transient time step of the FMU solve:
        $\mathrm{t_{FMU}} = \mathrm{t_{FMU}} + \mathrm{dt_{FMU}}$
    end while
end while

The transient coupled CTF–FMU cases were also tested using the External Model interface via the FMPy package.

One difficulty encountered in the development of the External, FMPy based transient transient coupling method is that CTF does not allow for a fixed time step to be set for its transient solver and instead calculates the timestep based on the CFL number. To make data exchange simpler the time points at which data exchange were passed from CTF to the FMU were pre-specified to be 0.2s, such that CTF was forced to step the core model forward for 0.2s, then write output and restart files that could be parsed for the boundary conditions to subsequently hand off to the FMU model.

The FMU solver used a fixed timestep of ~ 0.076 s that was determined based on the mean timestep used for the internal coupling. The total computational time for the python scripting transient coupling was much higher than the internal coupling solve—by a factor of ~ 8.5. The efficiency of the Python scripting coupling would have to be considerably improved to come closer to the total run time of the internal coupling solve. The reduced performance of the external FMPy based transient coupling was hypothesized to be due, in part, to reading, writing, and parsing the input and output files of CTF and the FMU. In the internal in-memory coupling strategy no such reliance on file exchange at each time step is required. Further investigations to improving performance bottlenecks could be done.

3.2 TRANSIENT PUMP TRIP SENSITIVITY STUDY

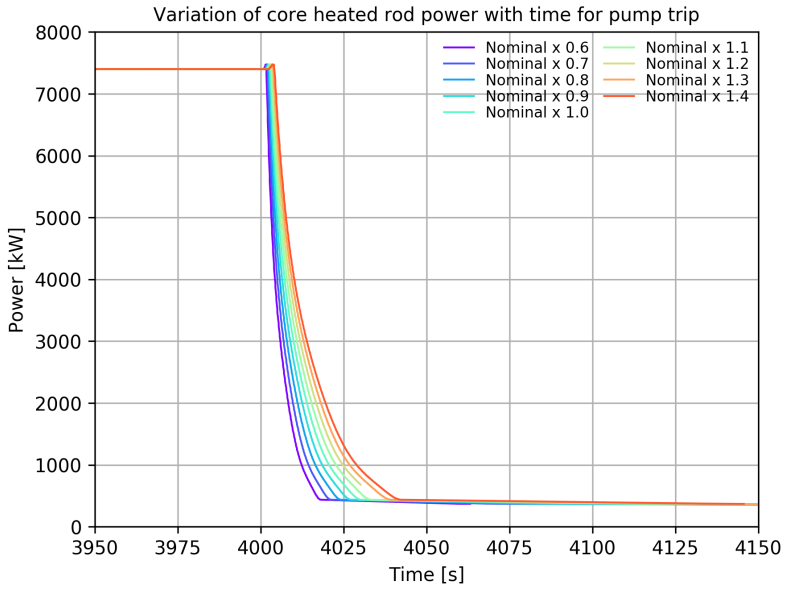
The first coupled CTF-FMU transient case performed was a secondary loop transient with a consequent reduction in heated rod power, simulating a SCRAM event [1]. The transients were all initiated after a lead-in period of 4,000 s for the system to reach thermal equilibrium. Although the pump coast-down rate investigated in Gurecky et al. [1] was based on an MSRE technical report to simulate the transient operating conditions for the 10 s transient in which the secondary mass flow rate drops from nominal to ~5%, in the current study, the coast-down rate was varied while preserving the power reduction curve. A snippet of the RAVEN input file for this case is shown below in Listing 11. Nine samples of the pump coast-down transient time (fraction), **'timeFrac'**, were drawn from a uniform distribution with a grid sampler with a ±30–40% variation with respect to the expected pump coast-down time.

```
<Distributions>
  <Uniform name="dist_for_timeFrac">
    <lowerBound>0.6</lowerBound>
    <upperBound>1.3</upperBound>
  </Uniform>
</Distributions>

<Samplers>
  <Grid name="grid">
    <variable name="timeFrac">
      <distribution>dist_for_timeFrac</distribution>
      <grid construction="equal" steps="8" type="value">0.6 1.3</grid>
    </variable>
  </Grid>
```

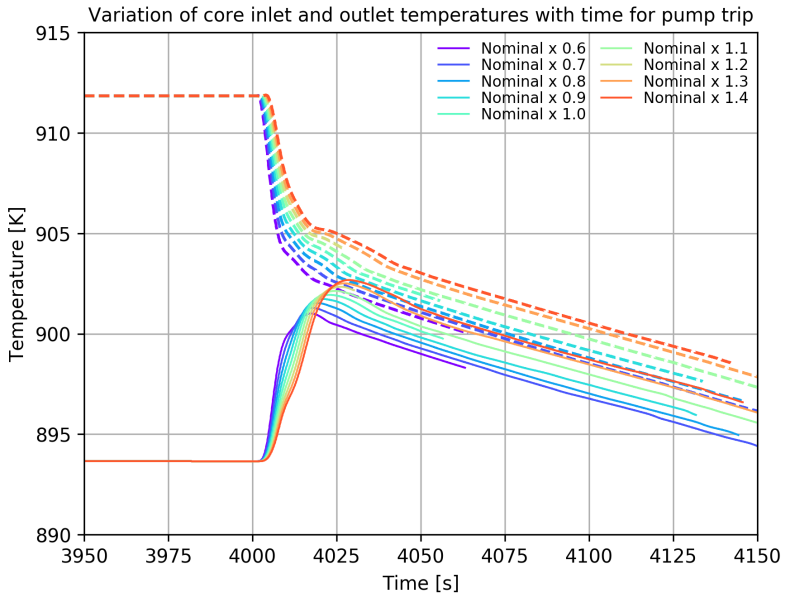
```
16  </Samplers>
```

Listing 11. RAVEN input file snippet showing the sampler and distribution for a transient pump trip sensitivity study. `test3_ctf_fmu.xml`

The results of the pump coast-down analysis are shown in Figure 6. Figure 6a and Figure 6c are essentially boundary conditions. As evidenced in Figure 6b, some cases did not converge all the way to the end of the transient. From a solver point of view, this type of study is useful in understanding the robustness of the coupled solver toward improving its convergence for some of these extreme conditions. For a more accurate analysis, a neutronics solver would be required, as mentioned previously. Using the in-memory coupling approach, the neutronics solver within the VERA toolkit could be used to couple it with CTF.

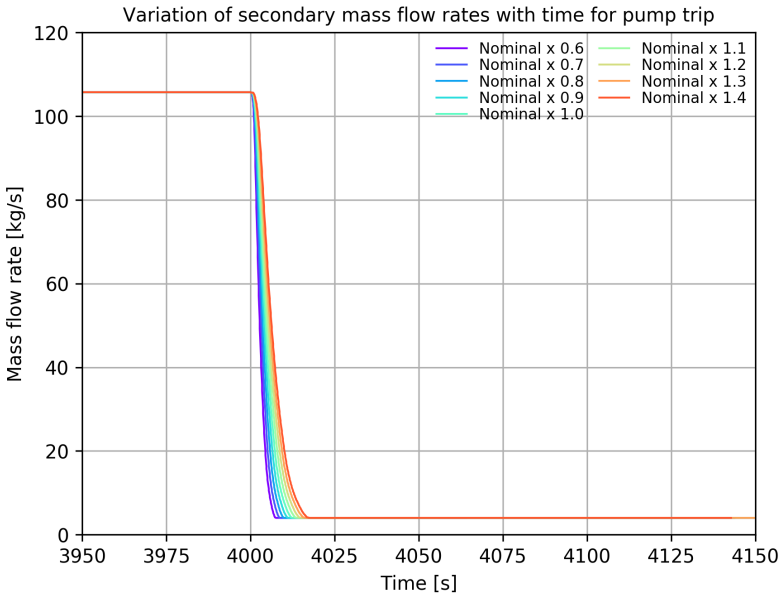


(a) Simulated sweep of pump trip frequency variation with time.



(b) Core inlet (solid lines) and outlet (dashed lines) temperatures vs. time for a sweep of pump trip frequency variation.

Figure 6. Sweep of pump trip frequency variation with time of a coupled CTF core model and TRANSFORM FMU system model.



(c) Pump secondary mass flow rates vs. time for a sweep of pump trip frequency variation.

Figure 6. Sweep of pump trip frequency variation with time of a coupled CTF core model and TRANSFORM FMU system model (contd.).

Results of the core inlet and outlet temperatures variation for the pump trip sensitivity study are shown below (Figure 7) for both the (in-memory) internal coupling interface (solid and dashed lines) and the Python scripting interface (markers). The results of the Python scripting interface are uniformly shifted upward by 1 K. For all cases except for the 0.6 factor case, both the results compare reasonably well across **'timeFrac'** factors for most of the transient. The 0.6 **'timeFrac'** factor case requires greater investigation, although the poor CTF convergence during the transient might be the contributing factor for the difference.

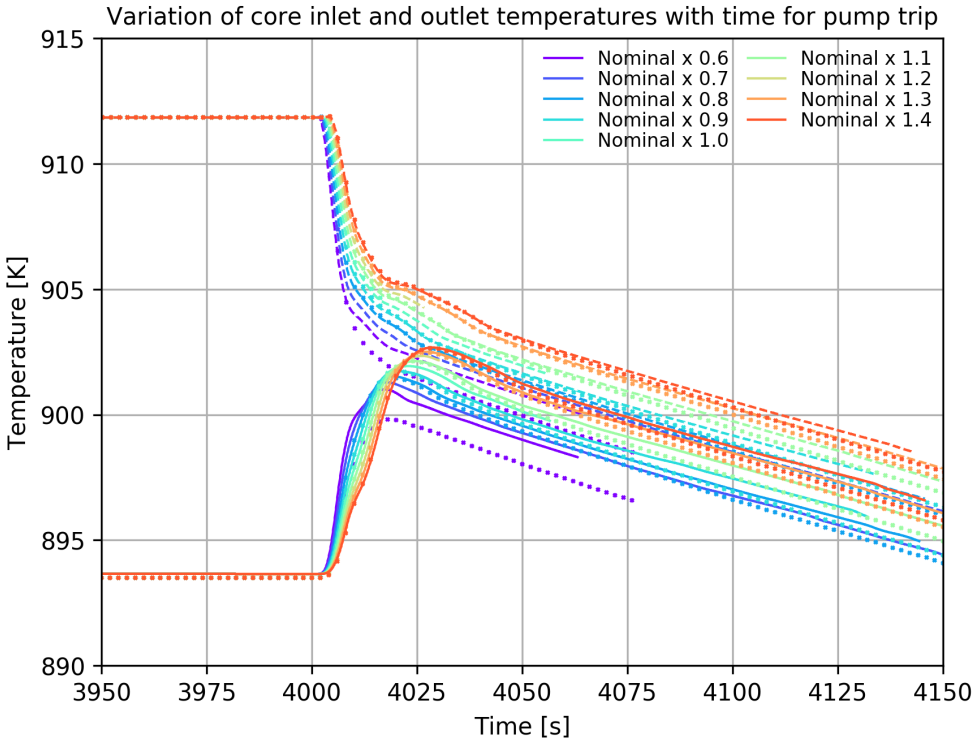


Figure 7. Comparison of core inlet and outlet temperatures with time for in-memory coupling (solid and dashed lines) and Python scripting coupling (markers) for a sweep of pump trip frequency variation.

3.3 TRANSIENT POWER RAMP SENSITIVITY STUDY

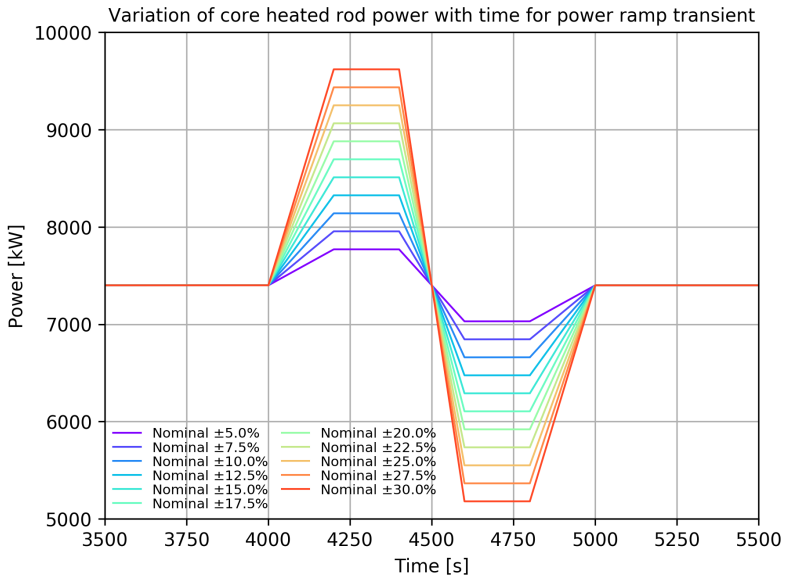
The second coupled CTF–FMU transient case performed was a 200 s heated rod power ramp transient with mirror image rising and falling power profiles for one cycle 8a, similar to Gurecky et al. [1]. Utilizing the capabilities of RAVEN, the amplitude of the profiles were varied to study the system response. A snippet of the RAVEN input file for this case is shown in Listing 12. Eleven samples of the power ramp amplitude (fraction), **'powerFrac'**, were drawn from a uniform distribution with a grid sampler variation of 0–30% with respect to a baseline power ramp. The highest and lowest peak amplitudes were within the expected MSRE operating range.

```
1
```

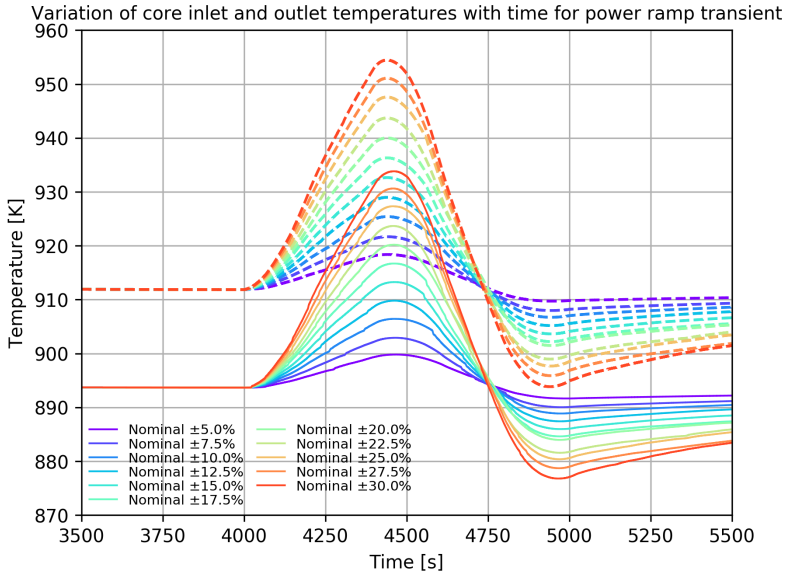
```
<Distributions>
  <Uniform name="dist_for_powerFrac">
    <lowerBound>0.0</lowerBound>
    <upperBound>0.30</upperBound>
  </Uniform>
</Distributions>

<Samplers>
  <Grid name="grid">
    <variable name="powerFrac">
      <distribution>dist_for_powerFrac</distribution>
      <grid construction="equal" steps="10" type="value">0.05 0.30</grid>
    </variable>
  </Grid>
</Samplers>
```

Listing 12. RAVEN input file snippet showing the sampler and distribution for a rod power amplitude and frequency perturbation study. `test4_ctf_fmu.xml`

The results of the reactor power ramp transient study are shown in Figure 8. Interestingly, the system response is sharper for the initial ramp up than the ramp down. The CTF core model is a simplified model, and a more accurate model with heat loss calibration would provide a more accurate representation of the system's thermal inertia.



(a) Simulated sweep of rod power ramp amplitude variation with time.



(b) Core inlet (solid lines) and outlet (dashed lines) temperatures vs. time for a sweep of rod power ramp amplitude variation.

Figure 8. Sweep of rod power ramp amplitudes with time of a coupled CTF core model and TRANSFORM FMU system model.

3.4 PERIODIC PERTURBATION AND FREQUENCY ANALYSIS STUDY

One of the primary goals of this work was to demonstrate the capabilities of RAVEN and how it can be used to study coupled systems consisting of a large diversity of scales, complexities, and physical behaviors. To understand the integral behavior resulting from the coupling of such a diversity of systems, RAVEN can be used to develop an understanding of the coupled configuration by studying the impact that a given input actuation can have on measured outputs as the perturbations propagate through the network of subcomponents and coupled systems. This understanding can be challenging to obtain from first-principles as systems increase in complexity, involving different physical phenomena on different scales. For example, if a nuclear reactor's primary loop is coupled to thermal energy storage, power conversion systems, and electricity distribution lines, then it can be difficult to understand how a change in reactor power could impact the grid frequency measured along the distribution lines, or to understand the impact that a change in electricity demand might have on the conditions inside the thermal energy storage vessels or reactor primary loop. Using RAVEN, it is possible to run a large series of diagnostic tests and systematically sweep across time scales for phenomena of interest to observe the behavior of the coupled systems and provide a concise description of the nature of these coupled systems. This was done by utilizing periodic perturbations at different frequencies and analyzing the resulting data in the frequency domain.

3.4.1 OVERVIEW OF FREQUENCY DOMAIN STUDIES USING RAVEN

Steady-state and transient simulations provide useful information for studying the performance of coupled systems under expected operating conditions and anticipated transient scenarios. However, these simulations have their limits: they might perturb only certain aspects of a given system over a limited range of time scales. To obtain a greater understanding of the nature and behavior of coupled systems, it is often useful to study the behavior in the frequency domain. By perturbing the relevant inputs to the system with periodic input signals at different frequencies, the coupled relationships can be systematically studied over large time scale ranges to reveal any unexpected behavior or instabilities and provide a concise description of how outputs at different locations are impacted by changes in different input conditions. Here, a technique is demonstrated using simulations of the CTF–FMU coupling that could be used to characterize such behavior by using RAVEN to perform sensitivity studies with periodic perturbations and measuring the resulting changes in output variables.

3.4.2 TEST SEQUENCE DESIGN

For these tests, a square wave periodic perturbation was selected as the input sequence (Figure 9). A square wave is a binary sequence that can concentrate around 81% of the signal energy in the fundamental harmonic corresponding to the period of oscillation (Figure 10). They are generated easily by making a binary approximation of a sine wave, where a positive value is equal to the positive amplitude, and a negative value corresponds to the negative amplitude. This is typically calculated by dividing a sine wave of a certain frequency by the absolute value of that same sine wave at the same frequency, multiplying the result by the desired amplitude for the input perturbation, and then adding the steady-state input value to the perturbation value to obtain a time-dependent input sequence for actuating the system.

In this scenario, the parameters that can be changed are the input sequence amplitude and the input

sequence frequency. RAVEN was used to generate the parameters based on different distributions and perform the simulations of the input with the coupled CTF–FMU model.

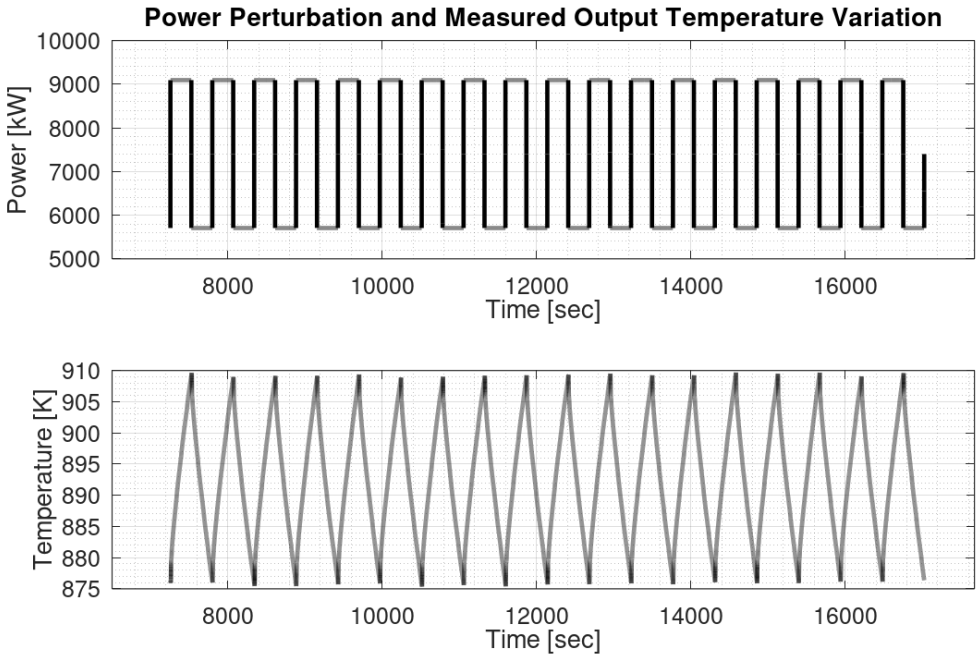


Figure 9. Periodic square wave input perturbation on reactor power and resulting reactor outlet temperature

3.4.3 CHARACTERIZATION OF LOW FREQUENCIES

To characterize the system across a large range of time scales, it was necessary to obtain a characterization of the lower frequency behavior. Lower frequencies require long simulation times, and in this scenario, the low frequencies were far from the time scales for the main phenomena of interest. For this reason, only a single slow transient was simulated to obtain the frequency response at the lower frequencies. The first transient case ran for a total of ~160,000 s, and it is shown in Figure 11.

3.4.4 CHARACTERIZATION WITH DISTRIBUTIONS OF PERTURBATION AMPLITUDES AND FREQUENCIES

3.4.4.1 Distributions with Variable Fraction of Base Value For the higher frequencies, an array of periodic perturbation tests were generated using RAVEN to demonstrate how such a technique can be useful for understanding high-low coupled systems. In the first set of simulations, 40 samples of the heated power amplitude (fraction), **'powerFrac'**, and perturbation frequency (fraction), **'timeFrac'**, were drawn from a uniform distribution with a Monte Carlo sampler. A snippet of the RAVEN input file for this case is shown in Listing 13. The distribution of values produced by RAVEN for this input are shown in Figure 13.

```
1
2  <Distributions>
```

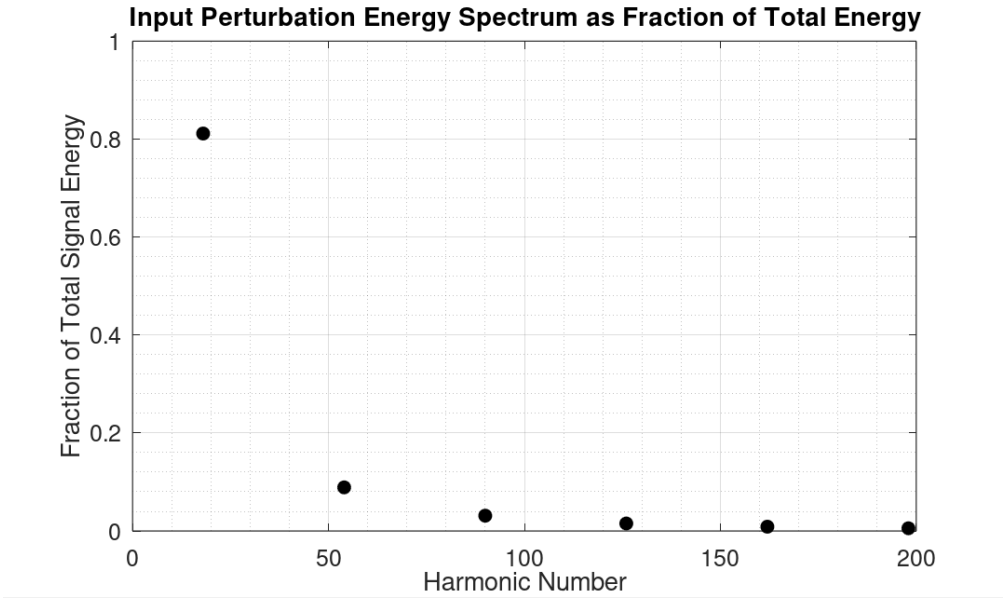
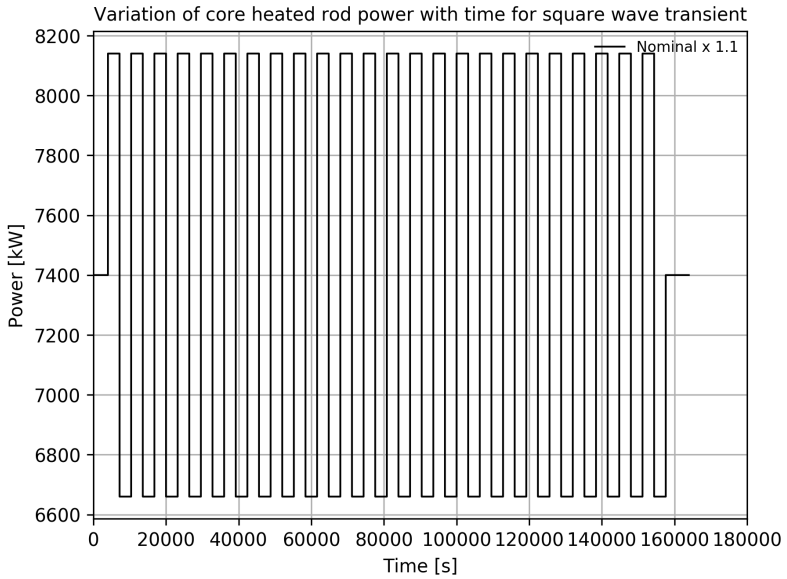


Figure 10. Signal energy spectrum of a square wave perturbation

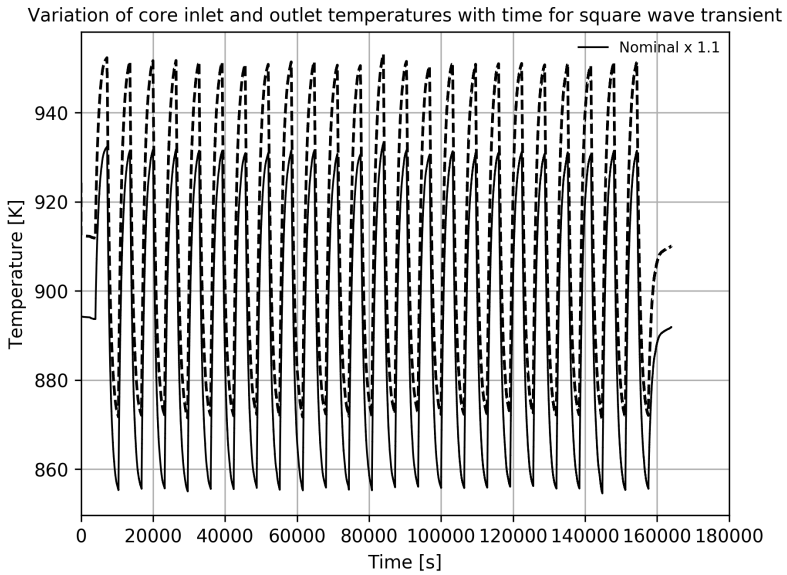
```
    <Uniform name="dist_for_powerFrac">
      <lowerBound>0.0</lowerBound>
      <upperBound>0.30</upperBound>
    </Uniform>
    <Uniform name="dist_for_timeFrac">
      <lowerBound>10.0</lowerBound>
      <upperBound>100.0</upperBound>
    </Uniform>
  </Distributions>

  <Samplers >
    <MonteCarlo name="MC_samp">
      <samplerInit>
        <limit>40</limit>
      </samplerInit>
      <variable name="powerFrac">
        <distribution>dist_for_powerFrac</distribution>
      </variable>
      <variable name="timeFrac">
        <distribution>dist_for_timeFrac</distribution>
      </variable>
    </MonteCarlo>
  </Samplers>
```

Listing 13. RAVEN input file showing the sampler and distribution for a rod power frequency perturbation study. `test5_ctf_fmu.xml`

3.4.4.2 Distribution with Fixed Fraction of Base Values An additional coupled CTF–FMU transient case to characterize the system frequency response involved a perturbation of frequency and heat rod power



(a) Power perturbation for low-frequency transient



(b) Resulting temperature variation for low-frequency transient

Figure 11. Low-frequency square wave power perturbation and resulting temperature changes.

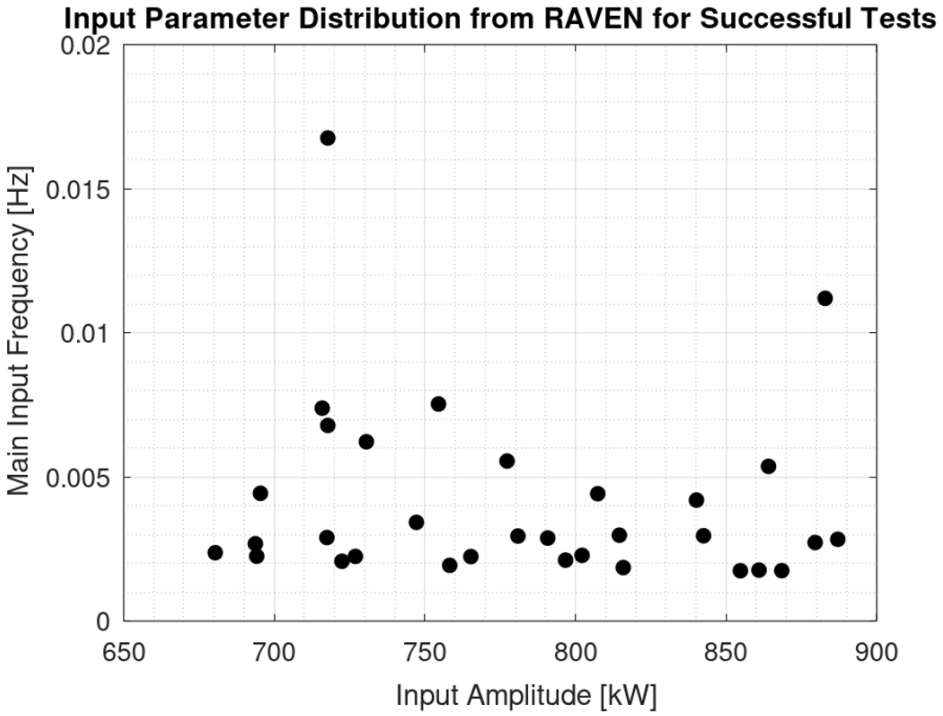


Figure 12. Overview of the selected wave frequencies and amplitudes from RAVEN for test cases sampled with uniform distribution with the Monte Carlo sampler, for variable fractions of the base value.

magnitude for a fixed perturbation (±10%) of amplitude variation. A snippet of the RAVEN input file for this case is shown in Listing 14. Forty samples of the heated rod power magnitude (fraction), **'powerFrac'**, and perturbation frequency (fraction), **'timeFrac'**, were drawn from a uniform distribution with a Monte Carlo sampler. The distribution of values produced by RAVEN for this input are shown in Figure 13.

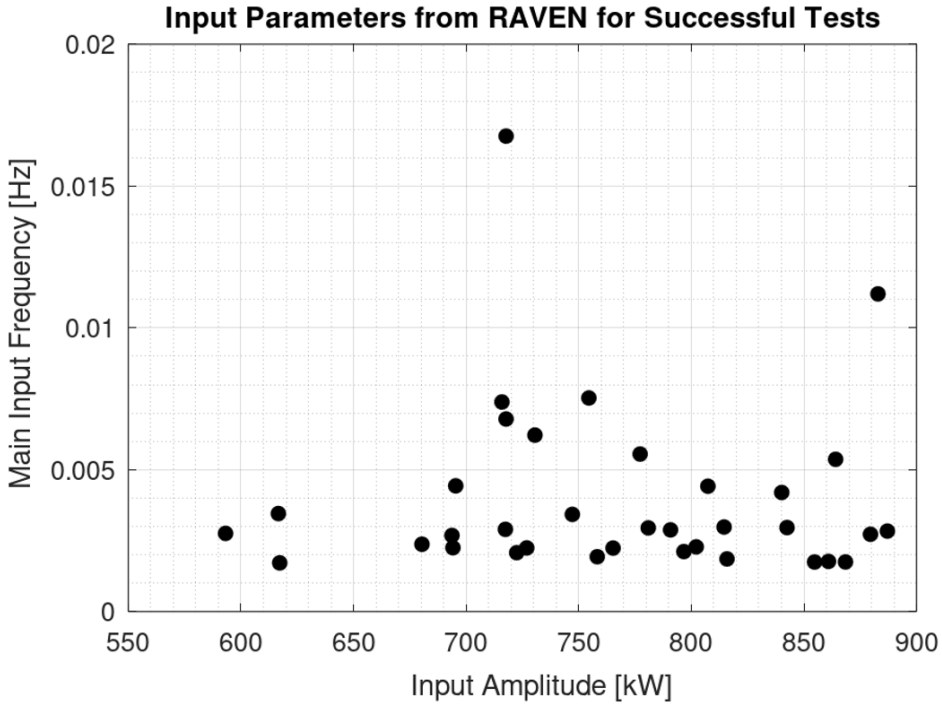


Figure 13. Overview of the selected wave frequencies and amplitudes from RAVEN for test cases sampled with uniform distribution with the Monte Carlo sampler, for fixed fractions of the base value.

```
<Distributions>
  <Uniform name="dist_for_powerFrac">
    <lowerBound>-0.20</lowerBound>
    <upperBound>0.20</upperBound>
  </Uniform>
  <Uniform name="dist_for_timeFrac">
    <lowerBound>10.0</lowerBound>
    <upperBound>100.0</upperBound>
  </Uniform>
</Distributions>

<Samplers >
  <MonteCarlo name="MC_samp">
    <samplerInit>
      <limit>40</limit>
    </samplerInit>
    <variable name="powerFrac">
      <distribution>dist_for_powerFrac</distribution>
    </variable>
    <variable name="timeFrac">
      <distribution>dist_for_timeFrac</distribution>
    </variable>
  </MonteCarlo>
</Samplers>
```

Listing 14. RAVEN input file showing the sampler and distribution for a rod power frequency perturbation study. `test6_ctf_fmu.xml`

3.4.5 ANALYSIS AND RESULTS

After the simulations were performed using RAVEN, the data underwent frequency analysis. First, the time segment containing the periodic perturbations was isolated from the larger dataset that included an approach to steady state and any post-perturbation transients. From this segment, the starting points and end points of the individual periods were identified, enabling the removal of the first few periods before the system stabilized to a steady-state periodicity about a stable mean without significant variable drift away from a mean value. The system frequency response is not defined by the absolute value of the variable values, but rather the change in the input and the resulting measured change in the output about some steady state achieved before the perturbations were implemented. Before frequency analysis could be performed on the data, the steady-state value was subtracted from the snipped data to obtain the change in input and change in output as a function of time.

The discrete Fourier transform was calculated for both the input and output perturbation data. The frequency response of the system was then calculated by dividing the discrete Fourier transform of the output perturbation by the discrete Fourier transform of the input perturbation. This gives a complex valued function with the magnitude providing the gain and the argument providing the phase angle. The bin values of the Fourier transform were then correlated to frequency values using the sample time of the data with an upper limit given by the Nyquist frequency, and the lower limit was given by the fundamental harmonic. During the analysis of this work, it became clear that initial simulations had floating time step values

instead of a constant time step. Frequency domain analysis requires a constant sampling rate. This variable sampling rate was an artifact of the CTF tool and not a result of RAVEN. Luckily, it is possible to use RAVEN to break the simulation into constant time step values and run them as separate simulations if fixed interval values are needed.

Input perturbations are typically designed to study certain frequencies in a system by spreading signal energy across chosen frequencies. The basic square wave that was used for this demonstration allowed for roughly 81% of the signal energy to be concentrated in the fundamental harmonic while still allowing for a binary input sequence that is easy to implement in a large variety of systems. The frequency response characteristics were filtered to retain frequencies with the highest signal energy calculated from the measured input perturbation. The RAVEN simulations ran sequences with different fundamental harmonics. The frequency domain data from all these tests were combined to provide a frequency-dependent description of the coupled system's frequency response characteristics, providing a concise description of the transient behavior of this coupling across a large frequency range.

These analysis steps were performed externally to RAVEN; however, if the analysis methods were included within RAVEN, it could make these techniques readily available for quickly performing this type of characterization with a large number of actuated inputs and measured outputs.

The resulting Bode plots for the variable-fraction and fixed-fraction distributions are shown in Figure 14 and 15, respectively.

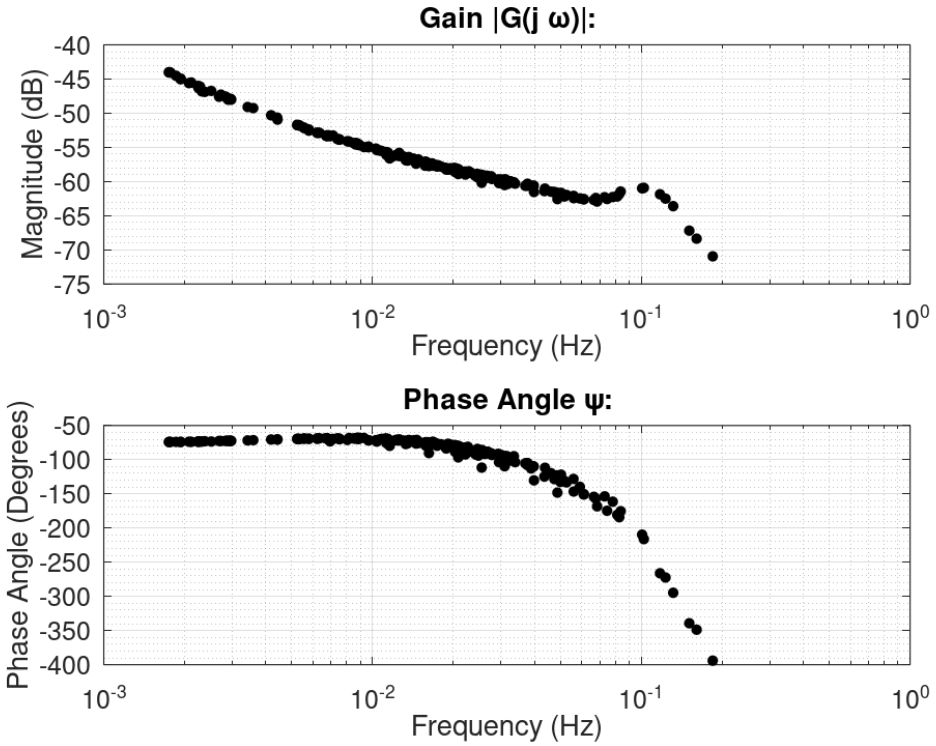


Figure 14. Bode plot of gain and phase angle from variable-fraction distribution.

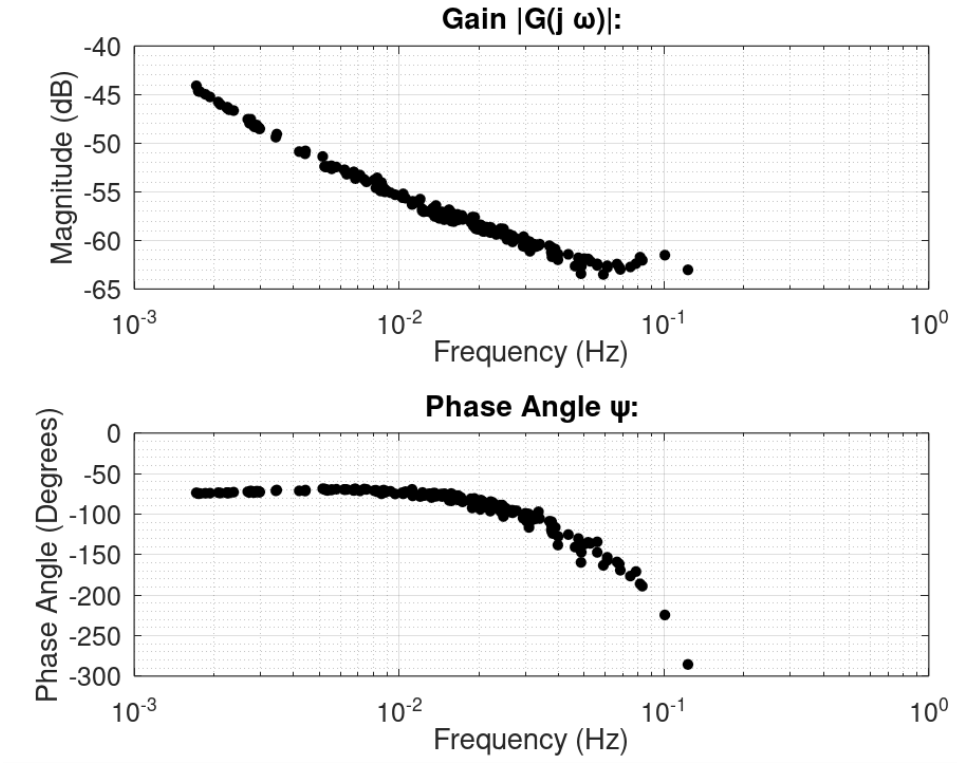


Figure 15. Bode plot of gain and phase angle from from fixed-fraction distribution.

4. CONCLUSIONS

This work demonstrated new functionality and applications that stemmed from the development of high-fidelity to low-fidelity (high-low) coupling for system simulations. RAVEN is a useful platform for coordinating these simulations and performing uncertainty quantification in this kind of high-low coupled system models. The FMU is an excellent methodology for interfacing with more complex system models, but there are limitations. Time step handling for the exchange of data between models can be affected for codes that allow for floating time steps. This can significantly increase the amount of processing time by the need to perform more information exchange between the models. This can result in a much finer time discretization in the lower fidelity model than what is needed, which slows calculation speed. Additionally, in-memory coupling of the codes was found to provide for much better computational performance. However, this may be due to the unoptimized usage of FMPy in Python. The additional need to exchange data in the out-of-memory implementations could make high-low coupling expensive.

Demonstrating RAVEN's ability to perform high-low coupled uncertainty quantification is an important step toward modeling increasingly more complex integrated energy systems in which overall system behavior will become more intertwined and performance effects of individual components are hard to quantify at the system scale. This tool should enable further exploration of these systems-of-systems in which mixed fidelity and mixed models prevail. Future work will continue to explore and expand these capabilities with RAVEN and FMUs.

Periodic perturbations were used to demonstrate techniques that can be used with RAVEN to investigate the transient nature of coupled systems across various time scales. It was possible to use RAVEN to create a distribution of test simulations that studied coupled system behavior at a range of time scales. Future work could incorporate these capabilities into RAVEN to make it possible to complete frequency analysis and characterization of the transient behavior of complex coupled configurations within the current framework. This would allow for rapid assessment and iteration of design configurations to achieve desired transient capabilities in systems. This could be valuable for validating load-following capabilities of integrated energy systems or evaluating potential resonance behavior that might emerge when merging different systems, among other applications.

5. BIBLIOGRAPHY

[1] William Gurecky, Dane de Wet, Michael Scott Greenwood, Robert Salko Jr, and Dave Pointer. Coupling of CTF and TRANSFORM using the Functional Mockup Interface. Technical Report ORNL/TM-2020/1872, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2020.

[2] Shannon Flumerfelt, Katherine G Schwartz, Dimitri Mavris, and Simon Briceno. *Complex systems engineering: Theory and practice*. American Institute of Aeronautics and Astronautics, Inc., 2019.

[3] Cristian Rabiti, Andrea Alfonsi, Joshua Cogliati, Diego Mandelli, Robert Kinoshita, Sonat Sen, Congjian Wang, Paul W Talbot, Daniel P Maljovec, and Jun Chen. RAVEN user manual. Technical report, Idaho National Lab.(INL), Idaho Falls, ID (United States), 2017.

[4] R. Salko, M. Avramova, A. Wysocki, A. Toptan, J. Hu, N. Porter, T. Blyth, C. Dances, A. Gomez, C. Jernigan, J. Kelly, and A. Abarca. *CTF Theory Manual*, 2019.

[5] B. Kochunas et al. VERA core simulator methodology for pressurized water reactor cycle depletion. *Nuclear Science and Engineering*, 3:13–15, 2017.

[6] Michael S Greenwood, Richard Hale, Lou Qualls, Sacit Cetiner, David Fugate, Thomas Harrison, et al. TRANSFORM-TRANsient Simulation Framework of Reconfigurable Models. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2017.

[7] Torsten Blockwitz, Martin Otter, Johan Akesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, et al. Functional Mockup Interface 2.0: The standard for tool independent exchange of simulation models. In *Proceedings of the 9th International Modelica Conference 173, September 3-5, 2012, Munich, Germany*, 2012.

[8] Futility Development Group. *Futility Fortran utility*, 2020.

[9] Robert K Salko, Jr. and USDOE. Subkit, version 0.1, 2 2019.

[10] Dassault Systems. Fmpy, 2021.

APPENDIX A. RAVEN-CTF External Model Code

APPENDIX A. RAVEN CTF-FMU external model procedures

The main external model Python script is listed below, followed by the supplementary code procedures script. The procedure in the main script interface with RAVEN includes reading from the external model block in the RAVEN input file, relevant variable(s) initialization, receiving inputs coming from the RAVEN framework, passing the inputs to the supplementary code procedures script, and returning the output to RAVEN. For simultaneous runs of the external code, the necessary input files along with the supplementary code procedures script are copied to new folders created in the working directory. The supplementary module is then loaded from the working folder, and the relevant procedures are called.

```
"""
Created on October 15, 2021

@author: Vineet Kumar, ORNL
"""
import numpy as np
import os
import sys
import shutil
import importlib.util

def _readMoreXML(self,xmlNode):
    """
        Method to read additional params from the RAVEN XML input file
        @ In, object, xml object, XML object
        @ Out, None
    """
    self.ctfExe = None
    self.fmuDir = None
    for child in xmlNode:
        if child.tag == 'executable':
            self.ctfExe = child.text
        if child.tag == 'fmuDir':
            self.fmuDir = child.text
    if self.ctfExe is None:
        raise IOError("CTF executable not provided")
    if self.fmuDir is None:
        raise IOError("FMU Directory not provided")
    if not isinstance(self.ctfExe, str):
        raise IOError("CTF executable should be a (string) path " + self.ctfExe)
    if not isinstance(self.fmuDir, str):
        raise IOError("FMU directory should be a (string) dirname " + self.fmuDir)

def initialize(self,runInfo,inputs):
    """
        Method to define all variable initializations defined in ExternalModel
        @ In, runInfo, dict, Dictionary of the run parameters
        @ In, inputFiles, file object, File object of all input files
        @ Out, None
    """
    self.inputFiles = inputs
    self.workDir = runInfo['WorkingDir']
    self.batchSize = runInfo['batchSize']
    return

```

```
def createNewInput(self,inputs,samplerType,**Kwargs):
    """
        Method to pass the information coming from the RAVEN framework
        @ In, inputs, dict, Dictionary of other input variables
        @ In, samplerType, file object, File object of all input files
        @ In, **Kwargs, dict, variable dictionary of sampled variables
        @ Out, Dictionary of sampled variables
    """
    self.caseNo = Kwargs['prefix']
    # return sampled variables as a dict
    return Kwargs['SampledVars']

def run(self,Input):
    """
        Method to define all variable
        @ In, Input, dict, Dictionary of the run parameters
        @ Out, None
    """
    caseDir = 'case' + self.caseNo
    printOutput = True
    # Move input scripts into a seperate directory
    for dirName in os.listdir(self.workDir):
        # Does not clear working directory. User has to manually clear.
        if dirName == caseDir and os.path.isdir(dirName):
            shutil.rmtree(dirName, ignore_errors=True)
    os.makedirs(caseDir)
    files = os.listdir(self.workDir)
    for file in files:
        if os.path.isfile(file):
            if file.endswith(".xml") or file.endswith(".dat") or file.endswith(".txt")
     or file.endswith("fmuCTFCouple.py") or file.endswith("make_deck.py"):
                shutil.copy(file,caseDir)
    workDir = os.path.join(self.workDir,caseDir)
    # Load the python module in the directory
    spec = importlib.util.spec_from_file_location('fmuCTFCouple',os.path.join(workDir,
    'fmuCTFCouple.py'))
    fmuCTFCouple = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(fmuCTFCouple)
    if self.batchSize > 1:
        printOutput = False
    assert(len(Input.keys())==1)
    fmu = fmuCTFCouple.solveFMUCTF(self.fmuDir, self.ctfExe, self.inputFiles, workDir,
     printOutput, Input)
    self.AVG_ax21_chan_temp = fmu.solveSteadyFMU()
```

Listing 15. Main RAVEN external code procedure fmuCTF.py

The supplementary module contains the procedures that perform the steady-state CTF–FMU coupling using the Picard iteration scheme and then returns the output to the main module.

```
"""
Created on October 15, 2021

@author: Vineet Kumar, ORNL
"""
from __future__ import division, print_function, unicode_literals, absolute_import
import os
import sys
import shutil
import subprocess as sp
from fmpy import read_model_description, extract, dump
from fmpy.fmi2 import FMU2Slave
from fmpy.util import plot_result, download_test_file
import numpy as np
import csv
import pandas as pd
import xml.etree.ElementTree as ET
from collections import OrderedDict
mypath = os.path.dirname(os.path.abspath(__file__))
from ctfdataHDF5 import ctfdataHDF5
from GenericCodeInterface import GenericParser

class solveFMUCTF:
    """
      Class that runs the coupled CTF-FMU steady state solve to convergence
    """
    def __init__(self, fmuDir, ctfExe, inputFiles, workDir, printOutput, Input):
        """
            Constructor
            @ In, fmuDir, string, FMU working directory name
            @ In, ctfExe, string, CTF executable
            @ In, inputFiles, file objects, list of input file objects
            @ In, workDir, string, Working directory path
            @ In, printOutput, bool, print output to screen.
            @ In, Input, dict, dictionary of perturbed var(s).
            @ Out, None
        """
        # Define the simulation parameters for a steady state run
        # These are hardcoded for now
        self.start_time = 0.0
        self.threshold = 1.0
        self.step_size = 0.2
        self.nstepMax = 25000

        # Unit conversions
        self.t_K_F = 1.8
        self.t_F_K = 1.0/self.t_K_F
        self.t_Pa_bar = 1.e-5
        self.t_bar_Pa = 1.0/self.t_Pa_bar
        self.t_bar_psi = 14.5038
        self.t_psi_psf = 144.0
        self.t_bar_psf = self.t_bar_psi * self.t_psi_psf
        self.t_psf_bar = 1.0/self.t_bar_psf
```

```
        self.t_lbm_kg = 0.453592
        self.t_kg_lbm = 1.0 / self.t_lbm_kg

        # read the model description
        fmu_filename =  None
        self.workDir = workDir
        self.fmuDir = os.path.join(self.workDir, '../', fmuDir)
        self.printOutput = printOutput
        self.ctfExe = ctfExe
        self.inputFiles = inputFiles

        if not os.path.isdir(self.fmuDir):
            raise IOError("fmu directory is incorrect " + self.fmuDir)
        for file in os.listdir(self.fmuDir):
            if file.endswith('.fmu'):
                fmu_filename = os.path.join(self.fmuDir,file)
                break
        if fmu_filename is not None:
            model_description = read_model_description(fmu_filename)
        else:
            raise IOError("fmu object not found in " + self.fmuDir)
        if self.printOutput:
            dump(fmu_filename)

        # collect the input and output value references
        self.fmuInputs = OrderedDict()
        self.fmuOutputs = OrderedDict()
        for variable in model_description.modelVariables:
            if variable.causality == 'input':
                self.fmuInputs[variable.name] = variable.valueReference
            if variable.causality == 'output':
                self.fmuOutputs[variable.name] = variable.valueReference
                # Make sure the FMU only supplies mflow_corein, T_corein, and
P_coreout
                assert(variable.name in ['T_out', 'P_out', 'mflow_out'])

        # Make sure the FMU takes in mflow_coreout, T_coreout, and P_corein, amongst
optional args
        # Better variable name checking feature to be implemented.
        assert(all(var in list(self.fmuInputs.keys()) for var in ['T_in', 'P_in', '
mflow_in']))
        # FMU can only output 3 variables currently. To be modified.
        assert(len(list(self.fmuOutputs.keys()))==3)

        # Error checking for CTF input files.
        self.qPrime = None
        varFound = None
        self.checkCTFInp(self.inputFiles)
        fmuParamPertFlag = 0 # Flag to check if fmu_param.xml is perturbed.
        if 'HEAT' in Input.keys():
            self.qPrime = Input['HEAT']
        else:
            fmuParamPertFlag = 1
            keys = list(Input.keys())
            pertName = "$RAVEN-" + keys[0] + "$"
            pertValue = Input[keys[0]]
```

```
107
108         # Parse the xml file
109         if not os.path.isfile(os.path.join(self.workDir, 'fmu_param.xml')):
110             raise IOError("fmu_param.xml not found in working directory")
111         filenameXML = os.path.join(self.workDir, 'fmu_param.xml')
112         xmlDict = self.readXML(filenameXML)
113         # Remove the fmu_param file from the work dir because it is read by CTF
114     # and if it contains a perturbed variable CTF would throw up an error
115         # Future capability to be added to use the raven generic parser to
116         # write to the fmu_params file.
117         os.remove(os.path.join(self.workDir, 'fmu_param.xml'))
118         # Get list of inputs and outputs from XML file
119         rowNames = list(xmlDict.index)
120         indicesInit = [index for index, strings in enumerate(rowNames) if '
    FMU_VAR_INIT' in strings]
121         indicesBC = [index for index, strings in enumerate(rowNames) if 'BC_VAR_NAMES'
     in strings]
122         indicesTol = [index for index, strings in enumerate(rowNames) if 'tol' in
    strings]
123         indicesLog = [index for index, strings in enumerate(rowNames) if 'FMU_VAR_LOG'
     in strings]
124
125         fmuInit = OrderedDict()
126         fmuBC = OrderedDict()
127         self.fmuTol = OrderedDict()
128         self.fmuLog = OrderedDict()
129
130         for index  in indicesInit:
131             if fmuParamPertFlag == 1 and xmlDict.iloc[index]["value"] == pertName:
132                 xmlDict.iloc[index]["value"] = str(pertValue)
133                 varFound = True
134             try:
135                 float(xmlDict.iloc[index]["value"])
136             except ValueError:
137                 print(xmlDict.iloc[index]["value"] + " cannot be converted to a float
    in fmu_param.xml")
138             fmuInit.update({xmlDict.iloc[index]["name"]: \
139                         float(xmlDict.iloc[index]["value"])})
140         if fmuParamPertFlag == 1 and not varFound:
141             raise IOError("Incorrect perturbed variable found in fmu_param.xml file")
142         # Currently unused
143         for index in indicesBC:
144             fmuBC.update({xmlDict.iloc[index]["name"]: \
145                           xmlDict.iloc[index]["value"][0]})
146
147         for index in indicesTol:
148             try:
149                 float(xmlDict.iloc[index]["value"])
150             except ValueError:
151                 print(xmlDict.iloc[index]["value"] + " cannot be converted to a float
    in fmu_param.xml")
152             self.fmuTol.update({xmlDict.iloc[index]["name"]: \
153                                 float(xmlDict.iloc[index]["value"])})
154         for index in indicesLog:
155             self.fmuLog.update({xmlDict.iloc[index]["name"]: \
156                                 xmlDict.iloc[index]["value"]})
```

```
157
158         for key in self.fmuLog.keys():
159             assert(key in self.fmuInputs.keys() or key in self.fmuOutputs.keys())
160
161         assert(all(var in list(self.fmuTol.keys()) for var in ['toltemp', 'tolmf', '
    tolpress', 'toltemp_FMU', 'tolmf_FMU', 'tolpress_FMU']))
162         # Add relaxation params if not specified in fmu_params
163         if not 'ulax_T_corein' in self.fmuTol.keys():
164             self.fmuTol.update({'ulax_T_corein':1.0})
165         if not 'ulax_P_coreout' in self.fmuTol.keys():
166             self.fmuTol.update({'ulax_P_coreout':1.0})
167         if not 'ulax_mflow_corein' in self.fmuTol.keys():
168             self.fmuTol.update({'ulax_mflow_corein':1.0})
169         # Set initial values based on the values provided in the XML file
170         self.initInputs = []
171         self.initOutputs = []
172         for input in self.fmuInputs.keys():
173             assert(input in fmuInit.keys())
174             self.initInputs.append(fmuInit[input])
175
176         for output in self.fmuOutputs.keys():
177             if output == 'P_out':
178                 self.initOutputs.append(fmuInit['P_in'])
179             if output == 'T_out':
180                 self.initOutputs.append(fmuInit['T_in'])
181             if output == 'mflow_out':
182                 self.initOutputs.append(fmuInit['mflow_in'])
183
184         # extract the FMU
185         self.unzipdir = extract(fmu_filename)
186
187         self.fmu = FMU2Slave(guid=model_description.guid,
188                              unzipDirectory=self.unzipdir,
189                              modelIdentifier=model_description.coSimulation.modelIdentifier
    ,
190                              instanceName='instance1')
191
192         # initialize
193         self.fmu.instantiate()
194         self.fmu.setupExperiment(startTime=self.start_time)
195         self.fmu.enterInitializationMode()
196         self.fmu.exitInitializationMode()
197
198     def readXML(self, xmlFileName):
199         """
200           Method to parse the XML params file
201           @ In, filename, string, FMU XML params file
202           @ Out, data, dict, the pandas dataframe containing the data
203         """
204         with open(xmlFileName, 'r') as xml_file:
205             # read the data and store it as a tree
206             tree = ET.parse(xml_file)
207
208             # get the root of the tree
209             root = tree.getroot()
210
```

```
            # return the DataFrame
            return self.iterXML(root)

    def iterXML(self, root):
        """
          Method to return the XML data in a pandas DataFrame
          @ In, xml root, object, parsed XML root
          @ Out, data, dict, the dictionary containing the data
        """
        # temporary dictionary to hold values
        temp_dict = {}
        flag = 0
        for child in root:
            # iterate through all the fields
            if child.tag == 'ParameterList':
                index = 0
                for var in child:
                    temp_dict.update({child.attrib['name'] + str(index):var.attrib})
                    index +=1
            else:
                temp_dict.update({'tol'+ str(flag):child.attrib})
                flag += 1

        return(pd.DataFrame.from_dict(temp_dict, orient="Index"))

    def checkCTFInp(self, inputFiles):
        """
           Performs error checking for CTF input files.
          @ In, inputFiles, list, list of the input files
          @ Out, None
        """
        vuqParam = []
        vuqMult = []
        subKit = []
        # otherInp = []
        # inputDict = {}
        for inputFile in inputFiles:
            if inputFile.getFilename() == 'fmuCTF.py' or inputFile.getFilename() == '
fmuCTFCouple.py' or inputFile.getFilename() == 'thermophysical_properties.dat':
                continue
            if inputFile.getType().strip().lower() == "vuq_param":
                vuqParam.append(inputFile)
            elif inputFile.getType().strip().lower() == "vuq_mult":
                vuqMult.append(inputFile)
            elif inputFile.getType().strip() == "subKit":
                subKit.append(inputFile)
            # else:
            #     otherInp.append(inputFile)

        if len(subKit) != 1:
            raise IOError('One subKit based input file must be defined.')
        # raise error if the names of the vuq files are defined different than hard
coded ones (not allowed)
        hardCodedInputNames = ["make_deck.py", "vuq_mult.txt", "vuq_param.txt"]
        # multiplier modifier name check
        if (len(subKit) == 1) and (subKit[0].getFilename() != hardCodedInputNames[0]):
```

```
265             raise IOError("Name of the subKit input file must be " +
    hardCodedInputNames[0] + " in RAVEN input and placed in the working directory. No
    other name is accepted!")
266         if (len(vuqMult) == 1) and (vuqMult[0].getFilename() != hardCodedInputNames
    [1]):
267             raise IOError("Name of the multiplier modifier file must be " +
    hardCodedInputNames[1] + " in RAVEN input and placed in the working directory. No
    other name is accepted!")
268         # parameter modifier name check
269         if (len(vuqParam) == 1) and (vuqParam[0].getFilename() != hardCodedInputNames
    [2]):
270             raise IOError("Name of the parameter modifier file must be " +
    hardCodedInputNames[2] + " in RAVEN input and placed in the working directory. No
    other name is accepted!")
271
272     def genCTFInput(self, **Kwargs):
273         """
274             Method to return the passed simulation working directory
275             @ In, Kwargs, dict, variable inputs
276             @ Out, ctfName, string, ctf input file name
277             @ Out, nLev, int, Number of nodes
278         """
279         cmd = []
280         cmd.extend(['python2', 'make_deck.py'])
281         if 'inletTemp' in Kwargs.keys():
282             cmd.extend(['--inletTemp', str(Kwargs['inletTemp'])])
283         if 'inletFlow' in Kwargs.keys():
284             cmd.extend(['--inletFlow', str(Kwargs['inletFlow'])])
285         if 'qPrime' in Kwargs.keys():
286             cmd.extend(['--qPrime', str(Kwargs['qPrime'])])
287         p1 = sp.Popen(cmd, stderr=sp.PIPE, stdout=sp.PIPE, universal_newlines=True,
    cwd=mypath)
288         stdout, stderr = p1.communicate()
289         if p1.returncode != 0:
290             raise IOError("Failed to generate CTF input file %d %s %s" % (p1.
    returncode, stdout, stderr))
291         # Fixed params. To be modified in the future.
292         ctfFname = 'system_coupling_transform.inp'
293         nLev = 20
294         return (ctfFname, nLev)
295
296     def runCTF(self, ctfFname):
297         """
298             Method to return the passed simulation working directory
299             @ In, ctfFname, float, CTF input file name
300             @ Out, None
301         """
302         p1 = sp.Popen([self.ctfExe, ctfFname], stderr=sp.PIPE, stdout=sp.PIPE,
    universal_newlines=True, cwd=mypath)
303         if self.printOutput:
304             while True:
305                 line = p1.stdout.readline()
306                 if line != '':
307                     print(line)
308                 else:
309                     break
```

```
        stdout, stderr = p1.communicate()
        if p1.returncode != 0:
            raise ValueError("Failed CTF solve %d %s %s" % (p1.returncode, stdout,
    stderr))

    def extractCTFOutput(self, output):
        """
            This method is used to extract the code output results.
            @ In, output, string, the Output name root
            @ Out, response, dict, dictionary containing the data {var1:array, var2:
    array, etc}
        """
        if os.path.isfile(os.path.join(self.workDir,output+'.ctf.native.h5')):
            outfile  = os.path.join(self.workDir,output+'.ctf.native.h5')
            outputobj = ctfdataHDF5(outfile)
        else:
            raise IOError("h5 file not generated. Check CTF input file.")
        response = outputobj.returnData()
        return response

    def K2C(self, T):
        """
            Converts Kelvin to Celsius
            @ In, T, float, Temperature in Kelvin
            @ Out, T, float, Temperature in Celcius
        """
        return T - 273.15

    def C2K(self, T):
        """
            Converts Celcius to Kelvin
            @ In, T, float, Temperature in Celcius
            @ Out, T, float, Temperature in Kelvin
        """
        return T + 273.15

    def writeCSV(self, couplingCTFDict, couplingFMUDict, couplingFMULogDict):
        """
            Write output BC dicts to csv files
            @ In, couplingCTFDict, dict, Dict of BCs from FMU to CTF
            @ In, couplingFMUDict, dict, Dict of BCs from CTF to FMU
            @ In, couplingFMULogDict, dict, Dict of FMU log vars from fmu_params
            @ Out, None
        """
        couplingCTF_File = os.path.join(self.workDir, "coupling.ctf.out")
        couplingFMU_File = os.path.join(self.workDir, "coupling.transform.out")
        couplingFMULog_File = os.path.join(self.workDir, "coupling.fmu.out")
        dfCTF = pd.DataFrame.from_dict(couplingCTFDict, orient="Index")
        dfFMU = pd.DataFrame.from_dict(couplingFMUDict, orient="Index")
        dfFMULog = pd.DataFrame.from_dict(couplingFMULogDict, orient="Index")
        dfCTF.to_csv(couplingCTF_File, index=False, sep=' ',quoting=csv.QUOTE_NONE,
    escapechar=' ',float_format='%13.4e')
        dfFMU.to_csv(couplingFMU_File, index=False, sep=' ',quoting=csv.QUOTE_NONE,
    escapechar=' ',float_format='%13.4e')
        dfFMULog.to_csv(couplingFMULog_File, index=False, sep=' ',quoting=csv.
    QUOTE_NONE, escapechar=' ',float_format='%13.4e')
```

```
361
362     def solveSteadyFMU(self):
363         """
364             Method to return the converged solution of the steady state CTF-FMU case
365             @ In, None
366             @ Out, float, Core outlet temperature
367         """
368         simTime = self.start_time
369         outDictCTF = {}  # Dict to record BC from FMU sent to CTF
370         outDictFMU = {}  # Dict to record BC from CTF sent to FMU
371         outDictFMUlog = {} # Dict to FMU lo vars from fmu_params
372         vr_inputs = list(self.fmuInputs.values())
373         inputs = self.initInputs
374         input_keys = list(self.fmuInputs.keys())
375         vr_outputs = list(self.fmuOutputs.values())
376         outputs = self.initOutputs
377         output_keys = list(self.fmuOutputs.keys())
378
379         # FMU does not compute the core outlet pressure correctly
380         presFromFMU = outputs[output_keys.index('P_out')] # Core inlet pressure
381         tempFromFMU = outputs[output_keys.index('T_out')] # Core outlet temperature
382         mfFromFMU = outputs[output_keys.index('mflow_out')] # Core outlet mass flow
    rate
383
384         couplingConv = False
385         flag = 0
386
387         # Have to run system code for a few time steps. Not implemented.
388         cool_temp_inlet = self.K2C(tempFromFMU) # C
389         cool_mflow_inlet = mfFromFMU # kg/s
390
391         # Coupling loop
392         while (couplingConv == False):
393             tmp_cool_temp_inlet = cool_temp_inlet
394             cool_temp_inlet = self.fmuTol['ulax_T_corein']*self.K2C(tempFromFMU) +
    (1.0 - self.fmuTol['ulax_T_corein'])*cool_temp_inlet # C
395             tmp_cool_mflow_inlet = cool_mflow_inlet
396             cool_mflow_inlet = self.fmuTol['ulax_mflow_corein']*mfFromFMU + (1.0 -
    self.fmuTol['ulax_mflow_corein'])*cool_mflow_inlet # kg/s
397             # Generate CTF inputs
398             if self.qPrime:
399                 ctfFname, nLev = self.genCTFInput(inletTemp=cool_temp_inlet, inletFlow
    =cool_mflow_inlet, qPrime=self.qPrime)
400             else:
401                 ctfFname, nLev = self.genCTFInput(inletTemp=cool_temp_inlet, inletFlow
    =cool_mflow_inlet)
402
403             if self.printOutput:
404                 print("Setting CTF core inlet temperature [K] from ", str(self.C2K(
    tmp_cool_temp_inlet)), " to ", str(self.C2K(cool_temp_inlet)))
405                 print("Setting CTF core inlet mass flow rate [kg/s] from ", str(
    tmp_cool_mflow_inlet), " to ", str(cool_mflow_inlet))
406
407             # Run the CTF Code
408             self.runCTF(ctfFname)
409
```

```
            # Extract CTF Output data
            ctfOutput = self.extractCTFOutput('system_coupling_transform')
            presFromCTF = ctfOutput['AVG_ax1_chan_pressure'][-1]*self.t_bar_Pa # Pa
            tempFromCTF = self.C2K(ctfOutput['AVG_ax'+str(nLev+1)+'_chan_temp'][-1]) #
     K
            mfFromCTF = ctfOutput['AVG_ax'+str(nLev+1)+'_chan_mdot_liq'][-1] # kg/s
            cool_pres_outlet = ctfOutput['AVG_ax'+str(nLev+1)+'_chan_pressure'][-1] #
    bar
            ctfTime = ctfOutput['time'][-1]

            # Set FMU inputs from CTF Outputs
            for key in input_keys:
                # Core outlet mass flow rate
                if key == 'mflow_pumpprimary' or key == 'mflow_in':
                    tmp_mfFromCTF = inputs[input_keys.index(key)]
                    inputs[input_keys.index(key)] = mfFromCTF
                # Core outlet pressure
                if key == 'P_in':
                    inputs[input_keys.index(key)] = cool_pres_outlet*self.t_bar_Pa
                # Core inlet pressure
                if key == 'P_corein':
                    tmp_presFromCTF = inputs[input_keys.index(key)]
                    inputs[input_keys.index(key)] = presFromCTF
                # Core outlet temperature
                if key == 'T_in':
                    tmp_tempFromCTF = inputs[input_keys.index(key)]
                    inputs[input_keys.index(key)] = tempFromCTF

            # Run the FMU Code
            if self.printOutput:
                print("Setting FMU core outlet temperature [K] from ", str(
    tmp_tempFromCTF), " to ", str(tempFromCTF))
                print("Setting FMU core outlet mass flow rate [kg/s] from ", str(
    tmp_mfFromCTF), " to ", str(mfFromCTF))
                print("Setting FMU core inlet pressure [bar] from ", str(
    tmp_presFromCTF*1.e-5), " to ", str(presFromCTF*1.e-5))
                print("****************************************")
                print("****************************************")
                print("***        start FMU SS run          ***")
                print("No  ", "   TempDiff ", "   presDiff ", "   mfDiff ",
                      "   tempFMU ", "   tempCTF ", "   pressFMU ", "   pressCTF ", "
      mfCTF ", "   mfFMU ")
            nstep = 0

            # FMU loop
            while True:
                # set the input
                self.fmu.setReal(vr_inputs, inputs)

                # Assign previous time-step values
                oldPressure = presFromFMU
                oldTemperature = tempFromFMU
                oldMassFlow = mfFromFMU

                # perform one step
                self.fmu.doStep(currentCommunicationPoint=simTime,
```

```
    communicationStepSize=self.step_size)

                # get the values for 'outputs'
                outputs = self.fmu.getReal(vr_outputs)
                presFromFMU = outputs[output_keys.index('P_out')]
                tempFromFMU = outputs[output_keys.index('T_out')]
                mfFromFMU = outputs[output_keys.index('mflow_out')]

                # Check of the FMU steady state
                presDiffT = abs(oldPressure - presFromFMU)
                tempDiffT = abs(oldTemperature - tempFromFMU)
                mfDiffT = abs(oldMassFlow - mfFromFMU)
                nstep += 1

                wf = "{0:3d} {1:11.3E} {2:11.3E} {3:11.3E} {4:11.3E} {5:11.3E} {6:11.3E} {7:11.3E} {8:11.3E} {9:11.3E}"

                if (nstep == 1 and self.printOutput):
                    print(wf.format(nstep, tempDiffT, presDiffT, mfDiffT, tempFromCTF, tempFromFMU, presFromFMU, presFromCTF, mfFromCTF, mfFromFMU))

                if (nstep % 100 == 0.0 and self.printOutput):
                    print(wf.format(nstep, tempDiffT, presDiffT, mfDiffT, tempFromCTF, tempFromFMU, presFromFMU, presFromCTF, mfFromCTF, mfFromFMU))

                if (tempDiffT < self.t_F_K*self.fmuTol["toltemp_FMU"]):
                    if (presDiffT < self.t_psf_bar*self.t_bar_Pa*self.fmuTol["tolpress_FMU"]):
                        if (mfDiffT < self.t_lbm_kg*self.fmuTol["tolmf_FMU"]):
                            simTime += self.step_size
                            break

                if (nstep > self.nstepMax):
                    if self.printOutput:
                        print("FMU: max. iteration reached")
                    simTime += self.step_size
                    break

                # advance the time
                simTime += self.step_size

            # append the results
            flag += 1
            outDictCTF.update({flag : {'    Time [s]': ctfTime, ' Temperature [K]': tempFromFMU, ' Massflow [kg/s]':mfFromFMU, ' Pressure [Pa]':presFromFMU}})
            outDictFMU.update({flag : {'    Time [s]': ctfTime, ' Temperature [K]': tempFromCTF, ' Massflow [kg/s]':mfFromCTF, ' Pressure [Pa]':presFromCTF}})
            subDict = {}
            for key in self.fmuLog.keys():
                if key in input_keys and self.fmuLog[key]=="true":
                    subDict.update({key:inputs[input_keys.index(key)]})
                if key in output_keys and self.fmuLog[key]=="true":
                    subDict.update({key:outputs[output_keys.index(key)]})
            outDictFMUlog.update({flag:subDict})

            # Temporary variable defined because FMU outlet pressure not computed
```

```
    correctly
509             presFromFMUTemp = cool_pres_outlet*self.t_bar_Pa
510
511             # Check convergence tolerances
512             if (abs(self.C2K(cool_temp_inlet) - tempFromFMU) < self.t_F_K*self.fmuTol[
    "toltemp"]):
513                 if (abs(cool_pres_outlet*self.t_bar_Pa - presFromFMUTemp) < self.
    t_psf_bar*self.t_bar_Pa*self.fmuTol["tolpress"]):
514                     if (abs(cool_mflow_inlet - mfFromFMU) < self.t_lbm_kg*self.fmuTol[
    "tolmf"]):
515                         couplingConv = True
516                         if self.printOutput:
517                             print("Steady State simulation converged")
518                         break
519
520         # Terminate FMU
521         self.fmu.terminate()
522         self.fmu.freeInstance()
523
524         # Write CSV Files
525         self.writeCSV(outDictCTF, outDictFMU, outDictFMUlog)
526
527         # clean up
528         shutil.rmtree(self.unzipdir, ignore_errors=True)
529
530         # return core outlet Temperature
531         return tempFromCTF
```

Listing 16. Supplementary RAVEN external code procedure `fmuCTFCouple.py`