ARL-TR-9820 ● Oct 2023

User's Guide for the Hierarchical MultiScale Framework (HMS)

by Kenneth W Leiter, Joshua C Crone, and Jaroslaw Knap

ARL-TR-9820 ● OCT 2023

User’s Guide for the Hierarchical MultiScale Framework (HMS)

by Kenneth W Leiter, Joshua C Crone, and Jaroslaw Knap
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE	***Form Approved*** ***OMB No. 0704-0188***

1. REPORT DATE *(DD-MM-YYYY)* October 2023	**2. REPORT TYPE** Technical Report	**3. DATES COVERED (From - To)** January 2016–September 2023
4. TITLE AND SUBTITLE User’s Guide for the Hierarchical MultiScale Framework (HMS)		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Kenneth W Leiter, Joshua C Crone, and Jaroslaw Knap		**5d. PROJECT NUMBER**
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) DEVCOM Army Research Laboratory ATTN: FCDD-RLA-NA Aberdeen Proving Ground, MD 21005-5066		**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-9820
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		**10. SPONSOR/MONITOR’S ACRONYM(S)**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution is unlimited.

13. SUPPLEMENTARY NOTES
Primary author’s email: <kenneth.w.leiter2.civ@army.mil>.
ORCID: Joshua Crone, 0000-0003-0856-9149

14. ABSTRACT
The Hierarchical MultiScale Framework (HMS) facilitates the development of new multiscale models from the assembly of individual at-scale model components. A runtime system in HMS adaptively evaluates the at-scale models and performs data extraction necessary for scale-bridging between components. Existing at-scale models can be incorporated into HMS without modification allowing users to leverage complex and mature software codes. This User’s Guide serves to document the steps necessary for development of new multiscale models with HMS. It includes an overview of the mathematical approach to multiscale modeling that HMS adopts, the steps necessary to incorporate at-scale models into HMS, use of the HMS Broker to perform at-scale model evaluation and data extraction, and use of the adaptive surrogate modeling capabilities implemented in HMS.

15. SUBJECT TERMS
Multiscale Modeling, Scale-Bridging, Network, Cyber, and Computational Sciences, Sciences of Extreme Materials

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Kenneth W Leiter
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	47	**19b. TELEPHONE NUMBER (Include area code)** 410-278-2580

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

Acknowledgments

The Hierarchical MultiScale Framework (HMS) was developed through the financial support of the US Army Combat Capabilities Development Command Army Research Laboratory's Enterprise for Multiscale Research in Materials. Computational support, through a grant of computer time, was provided by the DoD High Performance Computing Modernization Program.

1. Introduction

A multiscale model is a high-fidelity model of a complex system that systematically incorporates relevant processes occurring at multiple spatial and temporal scales. In general, a multiscale model is composed of a number of at-scale model components that each describe a single scale relevant to the multiscale system. At-scale models are linked together via scale-bridging to form a multiscale model. Scale-bridging entails both extraction and communication of data between the at-scale model components. Due to its promise of increased fidelity, multiscale modeling is now relevant to many scientific disciplines including materials science, atmospheric science, biology, and chemical engineering.

For many complex systems, at-scale models are available and have been developed over many years in mature and complex software codes. Therefore, scale-bridging, the process of linking the individual at-scale models into a multiscale model, is the critical step for development of new multiscale models. The US Army Combat Capabilities Development Command Army Research Laboratory has developed the Hierarchical MultiScale Framework (HMS) to significantly simplify the labor-intensive process of scale-bridging for multiscale model development.[1] HMS fits into the category of multiscale computing software.[2] The primary goals of HMS are to 1) enable at-scale models to be incorporated directly into a multiscale model without modification; 2) support adaptive parallel evaluation of at-scale model components on large-scale heterogenous high-performance computing (HPC) resources; and 3) require minimal code modifications to evaluate and extract data from other at-scale model components. Adaptive machine learning methods are also incorporated into HMS to significantly reduce computational expense.[3]

To date, HMS is used in many multiscale modeling applications including the following:

- Finite element squared (FE^2) model of a composite fiber under impact loading[1]
- High-throughput screening of battery electrolyte materials using density functional theory (DFT)[4,5]
- Equation-of-state (EOS) calculations for an energetic material using dissipative particle dynamics (DPD)[3,6]

- Discrete element method (DEM) of soil compaction[7,8]
- Reactive chemistry of an energetic material using DPD[9]
- Crystal plasticity in magnesium by spectral homogenization
- FE^2 model of nanoscale architected trusses[10]

HMS is initially described by Knap et al.,[1] with additional developments in Knap et al.[5] and Leiter et al.[3] This report expands on previous publications to comprehensively describe the design and usage of HMS. Its purpose is to provide a practical guide for the implementation and execution of new multiscale models with HMS.

2. HMS Architecture

We begin with an overview of the mathematical approach for multiscale model development HMS adopts and then describe the corresponding software framework and computational methods the HMS framework uses to implement and evaluate the mathematical model.

2.1 HMS Mathematical Approach

The fundamental assumption of HMS is that a multiscale model may be decomposed into a collection of two-scale model building blocks. A two-scale model, depicted in Fig. 1, consists of an upper-scale model F and lower-scale model f. It is assumed that a separation of scales (both spatial and temporal) exists between F and f. The relationship between F and f closely follows the approach of the heterogeneous multiscale method (HMM),[11] where evaluation of F necessitates the estimation of missing quantities from f, which operates at a finer spatial or temporal scale. When F requires missing data, it supplies boundary conditions $\tilde{u}$ for evaluation of f consistent with some local state of the upper-scale model. Since f may not be able to directly utilize $\tilde{u}$, a mapping G, also called the input filter, transforms $\tilde{u}$ into $\hat{u}$ that can be directly evaluated by f. Following evaluation of f, the needed data for the upper-scale model must be extracted from f, a step performed by the mapping g, also called the output filter.

To make the mathematical model more concrete, we provide an example two-scale model described previously.[1] Consider the boundary value problem of classical linear elastostatics in three dimensions discretized using finite elements. The upper-

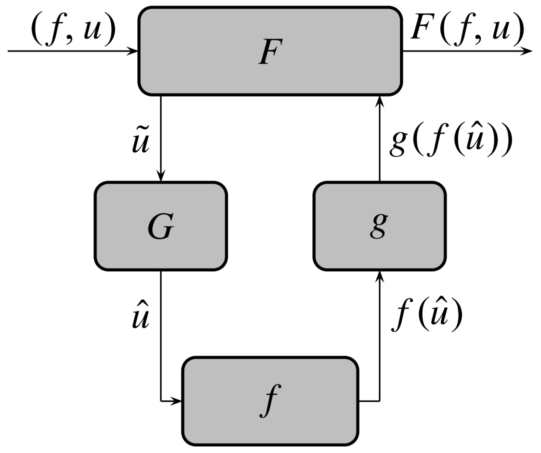

Fig. 1. Schematic of a generic two-scale model, where an upper-scale model (*F*) passes arguments to a lower-scale model (*f*) through input filter (*G*). The relevant output from the lower-scale model is passed back to the upper-scale model through an output filter (g)**.**

scale model treats the equilibrium calculations relating nodal forces to nodal displacements and the lower-scale model computes stresses via an elastic material model. Therefore, the upper-scale model is a function $F(f, u)$ where f is the elastic material model and $u \in \mathbb{R}^{3N_n}$ are nodal displacements, where N_n is the number of nodes in the finite element discretization. The stress returned from the lower-scale model is $f(\hat{u}) \in \mathbb{R}^6$, where $\hat{u} \in \mathbb{R}^6$ is the symmetric part of the displacement gradient. In this example, the input filter G computes the displacement gradient from the nodal displacements and in this example, g is identity as the upper-scale model can directly incorporate stresses into its calculation of nodal forces. However, if conversion between stress measures is required, these could easily be handled by adopting a suitable mapping for g.

In practice, the relationship between F and f may be more complex than the one depicted in Fig. 1. For example, the evaluation of F may necessitate multiple evaluations of f for different arguments u. Fortunately, in many cases the multiple evaluations of f may be performed concurrently. For example, in the linear elastostatics problem described before, evaluation of the elastic material model may be performed concurrently for all elements (or integration points) of the finite element discretization. Another common scenario is that several different f are needed by F to compute multiple quantities. It is also possible that for problems involving more than two scales, f also takes on the role of an upper-scale model and utilizes additional at-scale models at even finer spatial or temporal scales. The aim of HMS is to support development of multiscale models under all these commonly encountered scenarios.

2.2 HMS Computational and Software Framework

Now that the mathematical approach to multiscale modeling that HMS adopts has been established, we can describe the HMS software. HMS adopts a computational approach greatly motivated by the idea of cooperative parallelism[12] in which modeling and simulation applications are assembled from individual simulation components called symponents. A symponent carries out a unit computational task and communicates with other symponents to perform its function. Symponents are created and destroyed dynamically over the course of a simulation by a runtime system. Individual symponents can be serial or parallel (threads, message passing interface [MPI], etc.) and execute across disparate computational resources (central processing units [CPUs], graphics processing units [GPUs], etc.).

Multiscale models are naturally suited to cooperative parallelism, where at-scale models F and f are symponents and scale-bridging is accomplished through data estimation and communication between the symponents. HMS provides the ability for an upper-scale model to dynamically create, execute, and extract data from lower-scale model symponents. Both F and f may be serial or parallel and execute across heterogeneous computational resources.

The HMS software is implemented in the C++ programming language and includes bindings to Python (Section 10). Although HMS is implemented in C++, there are no restrictions on the programming language used to implement lower-scale models since lower-scale models are treated as stand-alone executables in HMS. By treating lower-scale models as stand-alone black boxes, HMS can also incorporate closed-source or proprietary software.

2.2.1 Defining a Lower-Scale Model (f) in HMS

To define a new lower-scale model to be incorporated into HMS, an Argument $\tilde{u}$, Value $g(f(\hat{u}))$, Input Filter G, Output Filter g, and Model f, must be specified. After these objects have been constructed, they are communicated to the runtime system (Broker, see Section 2.2.2) for scheduling and execution. Subclasses for the Argument, Value, Input Filter, and Output Filter must be implemented by the user and their abstract functions defined to be able to create a new lower-scale model in

HMS. The procedure is described in the following subsections, but first we describe the use of shared pointers in HMS.

2.2.1.1 Shared Pointers

HMS uses shared pointers to HMS objects extensively to simplify object ownership and memory cleanup. All public functions in HMS accept and return shared pointers to HMS objects. Typedefs for HMS classes are used to simplify syntax and consist of the class name followed by ”Pointer” (e.g., ArgumentPointer, ModelPointer). The shared pointer typedefs are defined by using the `HMS_SHARED_PTR()` macro located in `HMSMacros.h`. From the user’s perspective, interaction with shared pointers is similar to interaction with bare pointers without the need to worry about memory deallocation.

2.2.1.2 Argument

The Argument class represents $\tilde{u}$ in the two-scale model and contains the data passed to the lower-scale model. Subclasses of Argument are not required to implement any functions other than the serialize() method (see Section 2.2.1.8). The purpose of Argument is to store all necessary input data passed from the upper-scale model to the lower-scale model.

2.2.1.3 InputFilter

InputFilter performs the mapping G in the two-scale model. It includes one function that must be implement by the user, apply(), with the following signature:

```
void
apply(const arl::hms::ArgumentPointer & argument,
      const std::string               & directory) const
```

Two inputs, argument and directory, are supplied by the HMS runtime system. The argument is the ArgumentPointer for the current lower-scale model evaluation and directory is the path of the working directory where the lower-scale model evaluation will take place. The directory is a valid directory created by the HMS runtime system (see Section 4). The purpose of apply() is to perform any required transformation of the Argument for the lower-scale model and to generate the input files necessary to perform the lower-scale model evaluation. After apply() has finished, the lower-scale model is ready for execution inside the working directory.

2.2.1.4 OutputFilter

OutputFilter performs the mapping g in the two-scale model and its purpose is to extract and transform data from the completed lower-scale model evaluation to return to F. It includes one function, apply(), that must be implement by the user with the following signature:

```
ValuePointer
apply(const std::string                 & directory,
      const std::string                 & stdOut,
      const arl::hms::ArgumentPointer   & argument)
```

The three inputs provided to apply() include the directory path where the completed lower-scale model evaluation took place, the stdOut path to the file containing the redirected `stdout` of the lower-scale model evaluation, and the ArgumentPointer for the lower-scale model evaluation. The apply() method must return a Value-Pointer that contains the extracted data required by the upper-scale model. If data cannot be extracted from the completed lower-scale model (e.g., the lower-scale model evaluation encountered an error), then an exception should be thrown to alert the HMS runtime system (see Section 8).

2.2.1.5 Value

The Value class represents $g(f(\hat{u}))$ and is the quantity returned to the upper-scale model. No methods are necessary to implement in subclasses of Value besides serialize() as its purpose is to store the data that will be returned to the upper-scale model.

2.2.1.6 Model

The final component necessary to fully define a lower-scale model is the Model that represents f. The Model stores information about how to execute f. HMS requires the lower-scale model to be implemented as a stand-alone executable. Unlike other classes needed to incorporate a lower-scale model (Argument, InputFilter, Output-Filter, Value) into HMS, it is not necessary to implement a new Model subclass. Rather, Model objects can be directly instantiated by the user. The Model constructor has the following signature:

```
  Model(const std::string                & prefix,
        const std::string                & executable,
        const std::vector<std::string>   & arguments,
        const ResourceType                  & resourceType,
        int                                   resourceAmount)
```

The inputs to the Model constructor consist of two groups. The first three arguments (prefix, executable, and arguments) contain all the command line arguments necessary to execute f. The last two arguments (resourceType and resourceAmount) inform the HMS runtime system the computational requirements for f. Specifically, prefix is a string of command line arguments that are needed prior to the executable command (such as mpirun or python); executable is the path to the lower-scale model executable; arguments is the vector of command line arguments supplied to the executable; resourceType is the type of resource required to run the model (e.g., CPU, GPU); and resourceAmount is the number of resources required to evaluate the model.

Prior to the evaluation of the lower-scale model, the HMS Broker sets the current working directory to the directory where the evaluation will take place and therefore any relative paths supplied to the lower-scale model will be relative to this directory. The HMS Broker also generates an MPI machinefile named `machinefile` in the lower-scale model evaluation directory that contains a list of resources assigned to the lower-scale model. Therefore, when the executable is an MPI application, prefix will be “mpirun -np # -machinefile machinefile”, where # is the resourceAmount. Slight variations of the prefix may be necessary depending on the particular MPI version.

2.2.1.7 Packaging a Model Evaluation in a ModelPackage

Once subclass implementations of Argument, InputFilter, OutputFilter, and Value exist for a particular lower-scale model, instances of these objects may be created and packaged together along with Model into a ModelPackage. A ModelPackage completely specifies a lower-scale model evaluation and is the object communicated to the HMS runtime system by the upper-scale model to perform a lower-scale model evaluation.

2.2.1.8 Data Serialization

HMS uses Boost serialization to transform HMS objects into bytes necessary to communicate lower-scale model data over the network. For extensive details on the use of Boost serialization, we refer the reader to the Boost documentation.[13] Each of the four subclasses implemented to define a new lower-scale model, namely subclasses of Argument, Value, InputFilter, and OutputFilter, must be serializable by Boost. Specifically, a template function serialize() must be implemented in each class having the following signature:

```
template <class Archive>
void
ClassName::serialize(Archive & ar, const unsigned int version)
```

Any data members stored by the class must be serialized into the archive by applying the Boost serialization operator (&). The vast majority of C++ data types (e.g., standard template library data types) may be serialized into the archive in a single line by applying the Boost serialization operator when the required header files are included.

When serializing user-implemented subclasses, it is necessary to first invoke serialization of the base class with the following line of code:

```
ar & boost::serialization::base_object<base_class_of_T>(*this);
```

2.2.1.9 Communicators

Communication between the upper-scale model and HMS runtime system (Broker or Adaptive Sampling module) is performed by HMS CommunicatorPointer objects. A CommunicatorPointer serializes and deserializes ModelPackage objects communicated between HMS and the upper-scale model. From a user's perspective, a CommunicatorPointer is obtained from commands that launch the Broker or Adaptive Sampling Module (See Sections 2.2.2 and 2.2.3 for details) and ModelPackages can be communicated by calling send() and receive() on the CommunicatorPointer. Two types of communication are implemented in HMS: transmission control protocol/Internet protocol (TCP/IP) and MPI and can be selected by settings in the HMS configuration file.

2.2.2 The HMS Broker (Evaluation Module)

The HMS Broker (Evaluation Module) receives ModelPackage objects from the upper-scale model and schedules, executes, and monitors lower-scale model evaluations defined by each ModelPackage on available computational resources. Currently, a single Broker implementation is available in HMS, named BasicBroker. It is possible that other Broker implementations (additional subclasses of Broker) may be developed in the future, but the following discussion assumes usage of BasicBroker.

Before describing the internal operation of the Broker, we first describe how applications use the Broker to perform lower-scale model evaluations. From a user's perspective, the Broker is a black box that accepts ModelPackage objects and asynchronously returns their corresponding Value.

The HMS runtime system is started by the upper-scale model by calling the BrokerLauncher::launch() function. The launch() function takes three arguments, the HMS configuration file path (hmsConf.ini described in Section 3), the rank of the upper-scale model process, and the size of the upper-scale model process. It returns a vector of CommunicatorPointers to the launched Brokers. Typically, the size of the returned vector is one, meaning that a single upper-scale model process communicates to a single Broker, but under unusual circumstances it may be advantageous for an upper-scale model process to communicate with multiple Broker objects, for example when running lower-scale model evaluations on local file systems across multiple nodes.

The number of Brokers launched and the number of Communicators returned from BrokerLauncher::launch depend on settings contained in the HMS configuration file described in detail in Section 3. Specifically, the Broker.Hosts setting contains a list of hosts on which Brokers are launched. If the number of hosts on which Brokers are launched is equal to or fewer than the size of the upper-scale model process, then a single communicator is returned from BrokerLaucher::launch() because each upper-scale model process will only communicate to a single Broker. On the other hand, if Broker.Hosts contains more hosts than the size of the upper-scale model process, more than one Communicator will be returned from BrokerLauncher::launch(). In all cases, Brokers are divided as evenly as possible among upper-scale model processes.

After the vector of CommunicatorPointers is returned from BrokerLauncher, ModelPackage objects can be communicated to the Broker by calling send() on the Communicators to begin the lower-scale model evaluation procedure inside the Broker. The procedure to evaluate the lower-scale model inside the Broker is depicted in Fig. 2.

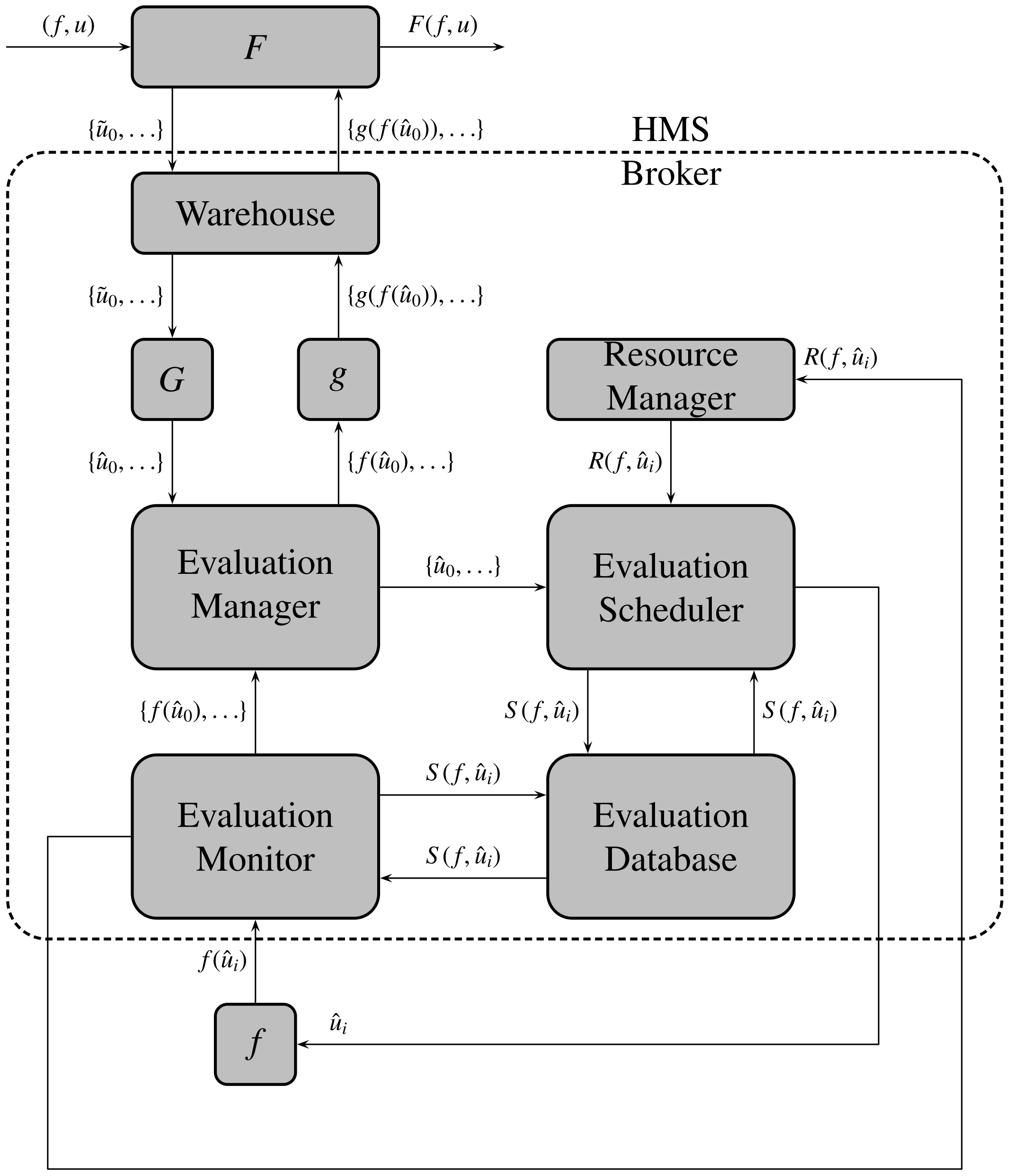

Fig. 2. A schematic of the process of the evaluation of f by Broker (Evaluation Module). $S(f, \hat{u}_i)$ denotes the current state of the $f(\hat{u}_i)$ evaluation. $R(f, \hat{u}_i)$ represents the resources required for the $f(\hat{u}_i)$ evaluation. The dashed line encloses all the components of the Broker.

After the Broker receives the ModelPackage from F, it prepares the ModelPackage for evaluation by creating a working directory for the model evaluation and applying the InputFilter G to setup the evaluation in the working directory. Next, the evaluation is added to the Scheduler to determine when the evaluation may begin, for example when sufficient computational resources are available from the Resource

Manager. Computational resources assigned to the Resource Manager are defined from settings in the HMS configuration file. Once the evaluation of the lower-scale model is started, an Evaluation Monitor checks the status of the evaluation to determine the current state of the evaluation (running, completed, aborted due to error, etc.). Following completion of the lower-scale model evaluation detected by the Evaluation Monitor, the Broker applies the OutputFilter to obtain the Value, attaches the Value to the ModelPackage, and communicates the ModelPackage back to the upper-scale model F. The upper-scale model receives the ModelPackage for completed evaluations by calling receive() on the Communicator to the Broker. The receive() function is asynchronous and if no ModelPackage objects are ready to be returned, the receive() call will return an empty vector. It is the responsibility of the upper-scale model to continue to call receive() on Broker communicators until all evaluations are returned.

2.2.3 The HMS Adaptive Sampling Module

HMS includes a capability to perform adaptive surrogate model evaluation and this functionality is implemented in the Adaptive Sampling module. The adaptive sampling algorithm is based on work by Knap et al.[14] and Barton et al.[15] with the implementation in HMS described in detail by Leiter et al.[3] In adaptive sampling, a surrogate model is trained on-the-fly from lower-scale model evaluation data. The surrogate model prediction is substituted for the lower-scale model prediction in regions where the predicted surrogate model error is below a user-defined threshold. As the multiscale model evaluation proceeds and more data is acquired from the lower-scale model, the surrogate model is expected to obtain greater accuracy and fewer lower-scale model evaluations will be necessary, greatly reducing the computational cost of the multiscale model. The surrogate model of choice in HMS has traditionally been Gaussian process regression since an error estimate can be readily obtained from the model, but the implementation of the adaptive sampling algorithm in HMS allows for incorporation of any surrogate modeling or machine learning algorithm, including neural networks.

The Adaptive Sampling module is launched from the upper-scale model by calling AdaptiveSamplingLauncher::launch() that returns a vector of CommunicatorPointers to the launched Adaptive Sampling modules and is typically of size one. ModelPackages can be sent to the Adaptive Sampling module via these Communicators to initiate the model evaluation process. From the perspective of the upper-scale model, interaction with the Adaptive Sampling module is identical to interaction with the Broker module: ModelPackages are sent over the Communicator and completed ModelPackages are received back from the Communicator.

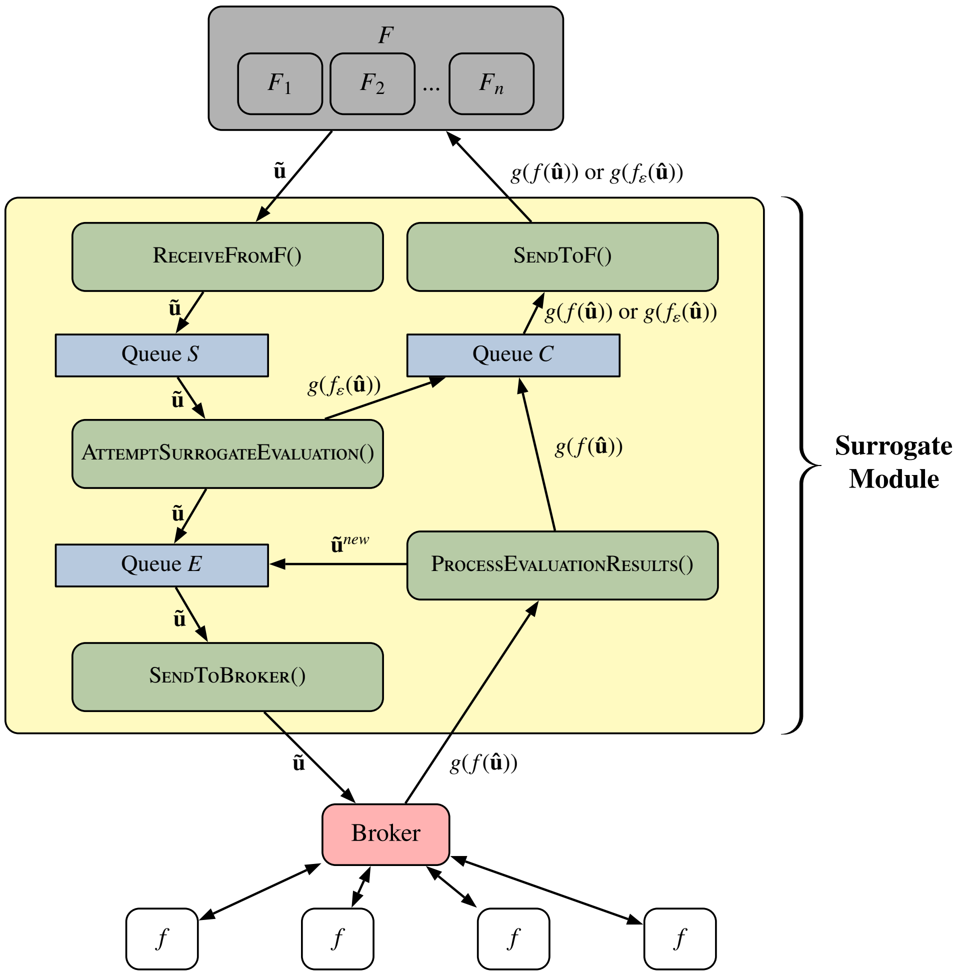
F
F_1
F_2
...
F_n
$\tilde{\mathbf{u}}$
$g(f(\hat{\mathbf{u}}))$ or $g(f_\varepsilon(\hat{\mathbf{u}}))$
RECEIVEFROMF()
SENDTOF()
Queue S
Queue C
$g(f_\varepsilon(\hat{\mathbf{u}}))$
ATTEMPTSURROGATEEVALUATION()
$g(f(\hat{\mathbf{u}}))$
Surrogate Module
Queue E
$\tilde{\mathbf{u}}^{new}$
PROCESSEVALUATIONRESULTS()
SENDTOBROKER()
Broker
f

Fig. 3. Operation of the Surrogate Module highlighted in yellow, between the upper-scale model F, and the Broker (Evaluation Module). The Surrogate Module is composed of several interacting functions, colored in green. Data is passed between functions in the Surrogate Module using shared queues, colored in blue, to permit use of pipeline parallelism for execution on multicore architectures. HMS can utilize multiple Surrogate Modules that execute concurrently. In such cases, each Surrogate Module communicates to a subset of the F processes. Only one Surrogate Module is depicted in this figure for clarity.

The AdaptiveSampling module is implemented as a multi-threaded application as depicted in Fig. 3 consisting of five threads: ReceiveFromF(), AttemptSurrogateEvaluation(), SendToBroker(), ProcessEvaluationResults(), and SendToF(). For more detail on the implementation of these threads see Leiter et al.,[3] but we provide a summary of their operation here. The ReceiveFromF() thread receives ModelPackages from the upper-scale model and places them into Queue S. The AttemptSurrogateEvaluation() thread removes a ModelPackage off Queue S and attempts to use the surrogate model to evaluate the ModelPackage. If the surrogate model evaluation is successful, the result is added to Queue C to return it to the upper-scale model; otherwise, the ModelPackage is added to Queue E to be routed to the Broker. The SendToBroker() thread sends ModelPackages in Queue E to the Broker for evaluation by the lower-scale model. The ProcessEvaluationResults() thread receives completed evaluations from the Broker and updates (retrains) the surrogate model with the data. In some implementations of the adaptive sampling algorithm, the update of the surrogate model may generate additional lower-scale model evaluations necessary to initialize a new surrogate model, which are added to Queue E to be routed to the Broker. Following the surrogate model update, completed ModelPackages are added to Queue C by the ProcessEvaluationResults() thread. Lastly, the SendToF() thread removes completed ModelPackages from Queue C and communicates them to the upper-scale model.

Surrogate models are implemented in HMS as subclasses of the InterpolationDatabase base class. The KrigingInterpolationDatabase uses the surrogate modeling method implemented in the ASF library[14,16] consisting of an M-Tree database of Kriging (Gaussian process regression) interpolators. Other implemented InterpolationDatabase surrogate models include PythonSurrogateModel, which can incorporate Python-based surrogate models, and NullInterpolationDatabase, which does not use any surrogate model and simply forwards all ModelPackage evaluations to the Broker, allowing the AdaptiveSampling module to act as a round-robin distributor of lower-scale model evaluations.

Subclasses of InterpolationDatabaseInputFilter and InterpolationDatabaseOutputFilter must be implemented by the user to allow lower-scale model evaluations defined in a ModelPackage to be evaluated by a surrogate model in the AttemptSurrogateEvaluation() thread. Both filters are passed to AdaptiveSamplingLauncher::launch and are required to start the Adaptive Sampling Module. The purpose of the InterpolationDatabaseInputFilter is to convert between the Argument and its representation as a vector of doubles. Similarly, the InterpolationDatabaseOutputFilter converts between the Value and its representation as a vector of doubles. The filters precisely define the quantities that get passed into and out of the surrogate model, as depicted in Fig. 4.

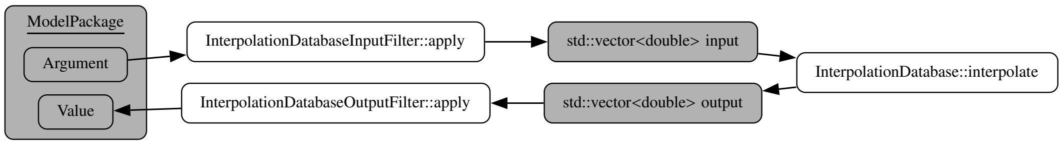
ModelPackage
Argument
Value
InterpolationDatabaseInputFilter::apply
std::vector<double> input
InterpolationDatabase::interpolate
std::vector<double> output
InterpolationDatabaseOutputFilter::apply

Fig. 4. Flowchart depicting the transformation of ModelPackage data by InterpolationDatabaseInputFilter and InterpolationDatabaseOutputFilter for surrogate model evaluation

3. HMS Configuration File (hmsConf.ini)

The HMS configuration file, typically named hmsConf.ini, defines all settings required to execute HMS applications, and is the primary means for user interaction with HMS applications at runtime. A path to the hmsConf.ini file is provided as an argument to BrokerLauncher::launch() and AdaptiveSamplingLauncher::launch() to start the HMS runtime system and is therefore required to use the HMS software. The format of the HMS configuration file follows the Python ConfigParser format consisting of sections of key/value pairs. Individual settings are typically referred to by using the Section.Key syntax. The configuration file contains four sections: AdaptiveSampling, Broker, Communicator, and Logging. Additional settings are included in the HMS configuration file when using the Hierarchical MultiScale Vectorized User MATerial Model (HMS-VUMAT) and we refer the reader to its documentation.[17]

A template hmsConf.ini file that contains default settings is provided in the top level directory of the HMS source code and is reproduced in the Appendix. The settings that must be configured by the user are denoted in the template with FILL_IN for the value, and includes settings for AdaptiveSampling.StartAdaptiveSampling and Broker.StartBroker, which need to be set to the respective StartAdaptiveSampling and StartBroker executable locations in the HMS installation `/bin` directory.

When HMS is run as a batch job through a batch queuing system such as PBS or SLURM, it is often necessary to generate a new HMS configuration file within the batch job script that is specific to each run. For example, Broker.Hosts may need to be populated with the nodes that are allocated to the job by the batch queuing system.

In the documentation within the following subsections, the default settings for options in the HMS configuration file are highlighted in bold. For some settings, such as Broker.ResourceNames and Broker.ResourceTypes, a space-separated list is needed. However, if the setting is the same for each entry of the list, one entry can be provided and it will be duplicated for each entry of the list by HMS.

3.1 Adaptive Sampling Configuration Settings

The AdaptiveSampling section contains settings for the Adaptive Sampling Module. If Adaptive Sampling is not used, the settings are ignored. Most settings are specific to the standard Kriging (or Gaussian process) regression and can be ignored if the Kriging interpolation is not being used.

Key	Description
AgingThreshold	Time threshold for object aging in adaptive sampling database **Default: 600000000**
DeltaH	Distances in each dimension used to generate new points around a sample point necessary to build a Kriging model when interpolating without derivatives (must be size of point dimension) **Default: 0.1 (for each dimension)**
Hosts	Hosts of adaptive sampling modules - a space-separated list of hosts on which adaptive sampling modules are launched
Library	Path to libraries loaded by adaptive sampler in order to deserialize objects, see Section 6
MaxKrigingModelSize	Maximum number of point/value pairs in a single Kriging model **Default: 50**
MaxNumberSearchModels	Maximum number of Kriging models to be pulled out of the database during the search for the best Kriging model **Default: 4**
MaxQueryPointModelDistance	Maximum distance between the query point and the model for which interpolation is still attempted **Default: 1000.0**
MeanErrorFactor	The value of the coefficient multiplying the mean square error **Default: 1.0**
MtreeDirectoryName	Name of the directory to use for storage of disk MTree data
NumberInterpolationThreads	Number of threads performing interpolation in the adaptive sampling module **Default: 1**
Scheduler	Scheduler for load balancing of brokers *Options: **RoundRobin**, Load*
StartAdaptiveSampling	Path to executable that launches adaptive sampling, typically in the HMS installation /bin directory
Tolerance	Error tolerance to switch from interpolation to model evaluation **Default: 10^{-4}**
Type	Interpolation type *Options: **Kriging**, Null, Python*

3.2 Broker Configuration Settings

The Broker section contains settings for the Broker (Evaluation Module).

Key	Description
Cleanup	Method to cleanup lower-scale model evaluation directories *Options:* ***None*** *(no clean up), Custom (clean up when specified by lower-scale model argument), Fixed (retain a fixed total number of lower-scale model evaluation directories)*
CleanupFixedSize	Number of lower-scale model evaluations to retain when Fixed Cleanup method is specified
Hosts	Hosts of brokers - a space-separated list of hosts on which brokers are launched to facilitate lower-scale model evaluations
Library	Path to libraries loaded by broker in order to deserialize objects, see Section 6
Monitor	Monitor type used to detect status of lower-scale model evaluations *Options:* ***Basic***
OutputDirectory	Path where lower-scale model evaluation scripts are stored and executed
Resources	A space-separated list of resources assigned to brokers - one per broker host. When Broker.ResourceTypes is Machinefile, the Resources are paths to machinefiles containing resources assigned to each broker. When Broker.ResourceTypes is List, the Resources are comma-separated list of resources assigned to each broker with spaces separating resources assigned to each broker
ResourceNames	A space-separated list of resource names - one per broker host *Options:* ***CPU****, GPU*
ResourceTypes	A space-separated list of resource types - one per broker host. For Localhost, all CPUs on the localhost of Broker are assigned to the Broker. For Machinefile and List, resources specified in Broker.Resources are assigned to the Broker *Options:* ***Localhost****, Machinefile, List*
Scheduler	Scheduler type for scheduling lower-scale model evaluations *Options:* ***Basic****, PBS*
StartBrokerPath	Path to executable that launches broker, typically in the HMS installation /bin directory
Type	Type of broker *Options:* ***Basic***

3.3 Communicator Configuration Settings

The Communicator section contains settings for communication between the upper-scale model and Broker and/or Adaptive Sampling modules. Typically, TCP/IP socket communication is used, but MPI communication is sometimes necessary, notably when performing fan-in communication where multiple upper-scale model processes communicate to a single Broker or Adaptive Sampling module.

Key	Description
Type	Type of communicator used to communicate between upper-scale model, broker, and adaptive sampling *Options:* ***TCPIPSocket****, UnixSocket, MPI*
UnixSocketFileName	Unix socket file name when Communicator Type is set to UnixSocket

3.4 Logging Configuration Settings

The Logging section contains settings for HMS logging when HMS is compiled with logging enabled (see Section 5)

Key	Description
Level	Logging level *Options: Trace,* ***Debug****, Info, Warning, Error, Fatal*
File	Logging file path

4. Broker OutputDirectory Structure

Lower-scale model evaluations occur inside the directory specified by the Broker.OutputDirectory setting in the HMS configuration file. It is often necessary for users to navigate within this directory to diagnose problems encountered when running lower-scale model evaluations (See Section 8). The directory is structured to ensure that multiple brokers running on a shared file system do not interfere with each other. Specifically, each broker creates a subdirectory inside Broker.OutputDirectory having the path `<Broker.OutputDirectory>/<hostname>/<pid>/` where `<hostname>` is the hostname where the Broker is running and `<pid>` is the process ID of the Broker. Each lower-scale model evaluation takes place in an individual directory inside `<Broker.OutputDirectory>/<hostname>/<pid>/`. The lower-scale model evaluation directories are numbered in increasing order beginning at 0.

HMS can automatically remove lower-scale model subdirectories after their evaluations finish by specifying a Broker.Cleanup policy in the HMS configuration file. Otherwise, the number of directories contained inside Broker.OutputDirectory can become unwieldy.

5. How to Compile HMS

The HMS library and executables must be compiled and installed before HMS can be used and models developed by users. HMS uses the GNU Autotools build system and follows the standard `configure`, `make`, and `make install` build procedure. The only required dependency is the compiled Boost C++ libraries.

An "out-of-source build" is recommended, where the `configure` and `make` commands are performed in a separate build directory, rather than in the directory containing the source files. When inside the build directory, the following command will perform the `configure` step to generate the makefiles:

```
<HMS_SRC>/configure --prefix=<HMS_INSTALL> --enable-shared
--with-mpi=<MPI_INSTALL>
```

where the paths inside of the angled brackets must be replaced with the locations on the specific system such that

- `<HMS_SRC>` = top-level directory for the source files of the HMS code
- `<HMS_INSTALL>` = directory where the headers, libraries, and binaries should be installed
- `<MPI_INSTALL>` = top-level directory where the MPI implementation was installed. MPI headers and libraries should be located in `<MPI_INSTALL>/include/` and `<MPI_INSTALL>/lib/`, respectively

It is possible to compile HMS entirely without MPI, but MPI communicators and fan-in communication between upper-scale models and HMS Broker or Adaptive Sampling module will be disabled and a runtime error thrown if attempted. It is also possible to use the `--with-mpi` flag without passing an `<MPI_INSTALL>` path so that the configuration step relies on specialized MPI-aware compilers (e.g., mpicxx, mpiCC, mpiicpc). The C++ compiler can be explicitly set by defining the "CXX" variable on the command line. An example using the intel MPI compilers is shown as follows:

```
<HMS_SRC>/configure --prefix=<HMS_INSTALL> --enable-shared
--with-hms=<HMS_INSTALL> --with-mpi CXX=mpiicpc
```

HMS can be compiled with adaptive sampling support by adding `--with-asf=<ASF_INSTALL>` to the `configure` command, where `<ASF_INSTALL>` is the top-level directory where the Adaptive Sampling Framework (ASF) library is installed. The ASF library can be obtained from https://github.com/exmatex/aspa. [14–16]

To compile the Python bindings for HMS, the `--enable-python-bindings` option should be added to the `configure` command. The build system will search the current environment for a suitable Python installation to use.

After `configure` is successfully executed, `make` will compile HMS and `make install` will install the headers, libraries, and executables into the specified `<HMS_INSTALL>` directory.

6. Compiling a User-Developed HMS Model into a Shared Library

Multiscale models developed with HMS must be compiled into shared libraries before they can be executed using the HMS runtime system. The HMS runtime system requires a shared library to be able to serialize and deserialize the Argument, InputFilter, OutputFilter, and Value implemented for specific lower-scale models. The location of the shared library must be set in hmsConf.ini under the Broker.Library and AdaptiveSampling.Library settings. If the shared library setting in Broker.Library and AdaptiveSampling.Library is inconsistent with the ModelPackage contents communicated by the upper-scale model, deserialization errors will occur and execution will be terminated.

7. Example HMS Application: Monte Carlo Estimation of π

In this section, an example HMS application is described that estimates the value of π via Monte Carlo. Although it is not strictly a multiscale modeling problem, the application fits the pattern of the two-scale HMS mathematical model and can therefore be easily implemented in HMS. The application demonstrates all steps required to implement models in HMS and has the advantage of being extremely simple.

The application is a two-scale model composed of F and f. The lower-scale model $f : [-1, 1] \times [-1, 1] \rightarrow \{0, 1\}$ takes a point in a 2×2 square centered at the origin and returns 1 if the point is contained in the unit circle and 0 otherwise. The upper-scale model F uses f to estimate pi. Specifically, $P(X \in \text{unit circle}) = \frac{\text{area of unit circle}}{\text{area of square}} = \frac{\pi}{4}$ when X is uniformly distributed in the square. Therefore, π is estimated in F as $\pi = 4 \times \sum_{i=1}^{n} f(x_i^1, x_i^2)/n$, where x_i^1 and x_i^2 are the coordinates of a point uniformly sampled in the square and n is the number of samples.

In HMS applications, f is implemented as a stand-alone executable. For this example, f is implemented in Python. The model f, `unit_circle_indicator_function.py`, reads the point from a file whose path is given as a command line argument and outputs 1 to `stdout` if the point is contained in the unit circle and 0 otherwise:

unit_circle_indicator_function.py

```
import sys

if __name__ == "__main__":

    if len(sys.argv) < 2:
        print("Usage: " + str(sys.argv[0]) + " <input-file-with-point>")
        sys.exit(-1)

    inputFile = sys.argv[1]

    f = open(inputFile)
    if not f:
        print("File containing point not found: " + inputFile)
        sys.exit(-1)

    x1 = float(f.readline())
    x2 = float(f.readline())

    squaredDistance = x1*x1 + x2*x2
    if squaredDistance < 1.0:
        print("1")
    else:
        print("0")
```

To incorporate f into HMS, an Argument, InputFilter, OutputFilter, and Value must be implemented. We start by implementing PiArgument and PiValue, subclasses of Argument and Value. PiArgument stores the point (x^1, x^2), which is given to the lower-scale model, and PiValue stores the return value of the lower-scale model, which is 0 or 1:

PiArgument.h

```
#include <base/Argument.h>

#include <boost/serialization/base_object.hpp>

using namespace arl::hms;

class PiArgument : public Argument {

public:

  PiArgument();
  PiArgument(double x1, double x2);
  virtual ~PiArgument();

  template <class Archive>
  void serialize(Archive & archive,
                 const unsigned int version) {
    using namespace boost::serialization;
    archive & base_object<Argument>(*this);
    archive & d_x1;
    archive & d_x2;
  };

  double getX1() const;
  double getX2() const;

private:

  PiArgument(const PiArgument &);
  PiArgument & operator=(const PiArgument &);
  double d_x1, d_x2;

};

HMS_SHARED_PTR(PiArgument);
```

PiArgument.cc

```
#include "PiArgument.h"

#include <utils/HMSMacros.h>

HMS_SERIALIZATION_EXPORT(PiArgument)

PiArgument::PiArgument()
{
}

PiArgument::PiArgument(double x1, double x2) :
  d_x1(x1), d_x2(x2)
{
}

PiArgument::~PiArgument()
{
}

double
PiArgument::getX1() const
{
  return d_x1;
}

double
PiArgument::getX2() const
{
  return d_x2;
}
```

PiValue.h

```
#include <base/Value.h>

#include <boost/serialization/base_object.hpp>

using namespace arl::hms;

class PiValue : public Value {

public:

  PiValue();
  PiValue(int value);
  virtual ~PiValue();

  template <class Archive>
  void serialize(Archive & archive,
                 const unsigned int version) {
    using namespace boost::serialization;
    archive & base_object<Value>(*this);
    archive & d_value;
  };

  int getValue() const;

private:

  PiValue(const PiValue &);
  PiValue & operator=(const PiValue &);

  int d_value;

};

HMS_SHARED_PTR(PiValue);
```

PiValue.cc

```
#include "PiValue.h"

#include <utils/HMSMacros.h>

HMS_SERIALIZATION_EXPORT(PiValue)

PiValue::PiValue()
{
}

PiValue::PiValue(int value) :
  d_value(value)
{
}

PiValue::~PiValue()
{
}

int
PiValue::getValue() const
{
  return d_value;
}
```

The PiInputFilter apply() method writes the point stored in PiArgument to the file that is read in by the lower-scale model:

PiInputFilter.h

```
#include <base/InputFilter.h>

#include <boost/serialization/base_object.hpp>
#include <boost/serialization/string.hpp>

using namespace arl::hms;

class PiInputFilter : public InputFilter {

public:

  PiInputFilter();
  PiInputFilter(const std::string & pointFileName);
  virtual ~PiInputFilter();

  virtual void
  apply(const ArgumentPointer & argument,
        const std::string     & directory) const;

  template <class Archive>
  void serialize(Archive & archive,
                 const unsigned int version) {
    using namespace boost::serialization;
    archive & base_object<InputFilter>(*this);
    archive & d_pointFileName;
  }

private:

  PiInputFilter(const PiInputFilter &);
  PiInputFilter & operator=(const PiInputFilter &);

  std::string d_pointFileName;

};

HMS_SHARED_PTR(PiInputFilter);
```

PiInputFilter.cc

```
#include "PiArgument.h"
#include "PiInputFilter.h"

#include <fstream>
#include <iostream>
#include <sstream>

#include <utils/HMSMacros.h>

HMS_SERIALIZATION_EXPORT(PiInputFilter)

PiInputFilter::PiInputFilter()
{
}

PiInputFilter::PiInputFilter(const std::string & pointFileName) :
  d_pointFileName(pointFileName)
{
}

PiInputFilter::~PiInputFilter()
{
}

void
PiInputFilter::apply(const ArgumentPointer & argument,
                     const std::string     & directory) const
{
  PiArgumentPointer piArgument =
    boost::dynamic_pointer_cast<PiArgument>(argument);
  const double x1 = piArgument->getX1();
  const double x2 = piArgument->getX2();

  std::stringstream filePath;
  filePath << directory << "/" << d_pointFileName;

  std::ofstream file(filePath.str().c_str());
  file << std::setprecision(16);
  file << x1 << std::endl;
  file << x2 << std::endl;
}
```

The PiOutputFilter parses the `stdout` generated from the lower-scale model evaluation to read its return value. The apply() method creates a new PiValue that stores the return value:

PiOutputFilter.h

```
#include <base/OutputFilter.h>

#include <boost/serialization/base_object.hpp>
#include <boost/serialization/string.hpp>

using namespace arl::hms;

class PiOutputFilter : public OutputFilter {

public:

  PiOutputFilter();
  virtual ~PiOutputFilter();

  virtual ValuePointer
  apply(const std::string     & directory,
        const std::string     & stdOut,
        const ArgumentPointer & argument) const;

  template <class Archive>
  void serialize(Archive & archive,
                 const unsigned int version) {
    using namespace boost::serialization;
    archive & base_object<OutputFilter>(*this);
  }

private:

  PiOutputFilter(const PiOutputFilter &);
  PiOutputFilter & operator=(const PiOutputFilter &);

};

HMS_SHARED_PTR(PiOutputFilter);
```

PiOutputFilter.cc

```
#include "PiValue.h"
#include "PiOutputFilter.h"

#include <fstream>
#include <iostream>

#include <utils/HMSMacros.h>

HMS_SERIALIZATION_EXPORT(PiOutputFilter)

PiOutputFilter::PiOutputFilter()
{
}

PiOutputFilter::~PiOutputFilter()
{
}

arl::hms::ValuePointer
PiOutputFilter::apply(const std::string     & /* directory */,
                      const std::string     & stdOut,
                      const ArgumentPointer & /* argument */) const
{

  std::ifstream file(stdOut.c_str());

  // f only outputs one line to stdout
  int value;
  file >> value;

  PiValuePointer returnValue(new PiValue(value));
  return returnValue;

}
```

The last step is to implement the upper-scale model F, which takes two command-line arguments: the path to the hmsConf.ini file and the number of Monte Carlo samples to evaluate. Once the command line arguments are parsed, F launches the Broker and then generates ModelPackages to define the lower-scale model evaluations. Each ModelPackage contains a PiArgument that contains a random point in $[-1, 1] \times [-1, 1]$. After each ModelPackage is generated, the ModelPackage is sent by F to the HMS Broker by calling send() on the Communicator, which begins the lower-scale model evaluation procedure in HMS. After all of the ModelPackages are sent to the Broker, F begins a loop to receive the completed evaluations from the Broker Communicator and stores a running sum of f values. Once all ModelPackages have been evaluated and their results received by F, the estimate for π is calculated by F and printed.

main.cc

```
#include "PiArgument.h"
#include "PiInputFilter.h"
#include "PiOutputFilter.h"
#include "PiValue.h"

#include <broker/BrokerLauncher.h>

#include <cstdlib>
#include <iostream>

const std::string unitCircleIndicatorDirectory("/p/app/projects/hms/ken/hms_mc_pi/src/");

int
main(int ac, char * av[])
{

  if(ac < 3) {
    std::cout << "Usage: " << av[0] << " <hmsConf.ini> <num-samples>" << std::endl;
    exit(-1);
  }

  const std::string configFile(av[1]);
  const int numSamples(std::atoi(av[2]));
  const std::string pointFileName("point.txt");

  // The upper-scale code is serial, so the rank and size are 0 and 1,
  // respectively
  std::vector<arl::hms::CommunicatorPointer> communicators =
    arl::hms::BrokerLauncher().launch(configFile, 0, 1);
  arl::hms::CommunicatorPointer communicator = communicators[0];

  std::stringstream unitCirclePath;
  unitCirclePath << unitCircleIndicatorDirectory << "unit_circle_indicator_function.py";

  // The Model, InputFilter, and OutputFilter are the same for all
  // samples and can be created outside the loop over sample points
  arl::hms::ModelPointer
    model(new arl::hms::Model("python",
                              unitCirclePath.str().c_str(),
                              std::vector<std::string>(1, pointFileName),
                              arl::hms::CPU,
                              1));
  arl::hms::InputFilterPointer inputFilter(new PiInputFilter(pointFileName));
  arl::hms::OutputFilterPointer outputFilter(new PiOutputFilter);
  // Create random Arguments, build the ModelPackages, and send to HMS
  // Broker for evaluation.
  for(int i=0; i<numSamples; ++i) {
    double x1 = 2.0 * static_cast<float>(rand())/static_cast<float>(RAND_MAX) - 1.0;
    double x2 = 2.0 * static_cast<float>(rand())/static_cast<float>(RAND_MAX) - 1.0;

    PiArgumentPointer argument(new PiArgument(x1, x2));

    arl::hms::ModelPackagePointer
      modelPackage(new arl::hms::ModelPackage(model,
                                              inputFilter,
                                              outputFilter,
                                              argument));
    communicator->send(modelPackage);
  }

  // Continue to check HMS Broker for returned values until all
  // pending evaluations are received back
  int counter = 0;
  double valueSum = 0.0;
  while(counter < numSamples){
    std::vector<arl::hms::ModelPackagePointer> modelPackages =
      communicator->receive<arl::hms::ModelPackagePointer>();
    counter += modelPackages.size();
    for(std::vector<arl::hms::ModelPackagePointer>::const_iterator iter =
          modelPackages.begin(); iter != modelPackages.end(); ++iter) {
      arl::hms::ModelPackagePointer modelPackage = *iter;
      PiValuePointer value =
        boost::dynamic_pointer_cast<PiValue>(modelPackage->getValue());
      valueSum += value->getValue();
    }
  }

  const double piEstimate = 4.0 * valueSum / static_cast<float>(numSamples);
  std::cout << "Estimate for pi: " << piEstimate << std::endl;

}
```

8. Debugging

Debugging HMS applications can be a challenge because lower-scale models, the upper-scale model, and the HMS modules (Broker and Adaptive Sampling) are executed as separate processes that do not share the same execution environment (call stack, allocated memory, etc.) as the upper-scale model. Adding to the challenge is that lower-scale model evaluations can often take place on remote compute resources that are separate from resources used to run the upper-scale model. In this section we describe how HMS handles lower-scale model evaluation failures and provide some guidance on how to track down problems in HMS applications.

A lower-scale model evaluation can fail for many reasons, including the presence of software bugs in the lower-scale model, inputs supplied to the model are outside its domain of suitability, the execution environment for model evaluation is incorrect (e.g., shared libraries cannot be found), as well as intermittent hardware failures. Failures in lower-scale model evaluations should be detected by the HMS

OutputFilter::apply() method that reads the output of the lower-scale model. When errors are detected, the OutputFilter::apply() method should throw an exception. If the error is thought to be caused by an intermittent hardware failure or a random error where a retry of the evaluation may be successful, then an IOError exception should be thrown. When an IOError exception is thrown by OutputFilter::apply(), the HMS Broker will retry the lower-scale model evaluation five times until the multiscale model execution is halted. If other exception types are thrown by OutputFilter::apply(), then the HMS Broker will cease execution immediately and the exception will cause a core dump file to be generated, which is typically written into one of the Broker.OutputDirectory subdirectories.

When an error is encountered in a lower-scale model evaluation that causes model evaluations to stop, an error message containing the contents of the thrown exception is written to `stderr` by the Broker, which is the same `stderr` inherited from the upper-scale model application. The exception message is the best place to begin debugging lower-scale model evaluation failures. Typically, the strategy to debug lower-scale model evaluation failure is to locate the failed evaluation inside the Broker.OutputDirectory. The directory of the failed lower-scale model evaluation will contain script.sh used to execute the lower-scale model, output.out containing the `stdout` of the lower-scale model, error.err containing the `stderr` of the lower-scale model, as well as any input files generated by InputFilter to prepare for the lower-scale model evaluation. After examining output.out and error.err for error messages of the lower-scale model evaluation, it is also possible to rerun the model entirely outside of HMS by manually executing script.sh. It may be necessary to generate a new MPI machinefile inside the lower-scale model evaluation directory in order to run script.sh if the lower-scale model is a parallel application.

It is also possible that errors will occur in the Argument, InputFilter, OutputFilter, and Value implementations as well as from possible bugs in the HMS implementation itself. Typically, these errors will result in exceptions being thrown with error messages printed to `stderr` before exit, or in catastrophic cases, with a segmentation fault and core files being dumped that can be examined with a debugger.

9. Performance Considerations

HMS records time stamps associated with its various actions, including application of input and output filters, lower-scale model execution start and end, and communication of the ModelPackage, and stores the time stamps inside the ModelPackage. The time stamps can be very helpful to understand bottlenecks in HMS model evaluation and can be obtained by the upper-scale model when it receives a completed ModelPackage by calling getTimestampLog() on the ModelPackage.

Typically, the most challenging scenario for HMS model performance is when lower-scale model evaluations are fast, since overhead to apply the input and output filters, start evaluation of the lower-scale model, and interaction with the file system can dominate the runtime. When lower-scale model evaluations are fast, an effective strategy to increase performance is to run a Broker on each node and assign all compute resources on the node to the locally running Broker. Given this configuration, the Broker can use an in-memory file system for Broker.OutputDirectory, such as a tmpfs often located in /dev/shm. Using an in-memory file system reduces HMS overhead substantially because all input/output required to set up the lower-scale model and perform its evaluation take place on a very fast file system. It is also effective to move any large input files required by the lower-scale model from a networked file system to the in-memory file system ahead of time to reduce overhead. In extreme cases, it can also be helpful to compile lower-scale models statically and move them to in-memory file systems to eliminate accessing networked file systems entirely. For Python-based lower-scale models, bundling the Python application using PyInstaller and moving it to an in-memory file system can help reduce startup overhead.

10. Python Interface

The Python interface to HMS allows for its use in upper-scale models written in Python as well as implementation of application-specific Argument, InputFilter, OutputFilter, and Value classes in Python. The Python interface is generated by the `--enable-python-bindings` option when invoking `configure` during HMS compilation. Once compiled and installed, HMS can be imported in Python applications by calling `import hms`. The HMS Python bindings are generated by SWIG.

One necessary detail when implementing HMS applications in Python is that application-specific filters need to be serializable. This can be accomplished by inheriting from a class named PickleableSWIG that contains special `__setstate__()` and `__getstate__()` methods (see `pi_filters.py` example).

When running Python-based HMS applications, the shared library specified in hmsConf.ini for AdaptiveSampling.Library and Broker.Library should be set to _hms.so located in the Python site-packages directory.

A Python implementation of the Monte Carlo estimation of π implemented in C++ in Section 7 is as follows:

pi_filters.py

```
import hms
import os
import sys

#
# Allow SWIG type objects to be pickled
#
class PickleableSWIG:

    def __setstate__(self, state):
        self.__dict__ = state

    def __getstate__(self):
        state = self.__dict__
        if "this" in state:
            del state["this"]
        return state

class PiArgument(hms.Argument, PickleableSWIG):

    def __init__(self, x1, x2):
        hms.Argument.__init__(self)
        self.x1 = x1
        self.x2 = x2
        return

class PiInputFilter(hms.InputFilter, PickleableSWIG):

    def __init__(self, pointFileName):
        hms.InputFilter.__init__(self)
        self.pointFileName = pointFileName
        return

    def apply(self, argument, directory):
        path = os.path.join(directory, self.pointFileName)
        f = open(path, "w")
        f.write(repr(argument.x1) + "\n")
        f.write(repr(argument.x2) + "\n")

class PiOutputFilter(hms.OutputFilter, PickleableSWIG):

    def __init__(self):
        hms.OutputFilter.__init__(self)
        return

    def apply(self, directory, stdOut, argument):
        f = open(stdOut)
        value = float(f.readline())
        return PiValue(value)
```

```
class PiValue(hms.Value, PickleableSWIG):

    def __init__(self, value):
        hms.Value.__init__(self)
        self.value = value
        return
```

main.py

```
import hms
import random
import time
from pi_filters import *

if __name__ == "__main__":

    if len(sys.argv) < 3:
        print("Usage: " + str(sys.argv[0]) + " <hmsConf.ini> <num-samples>")
        sys.exit(-1)

    configFile = sys.argv[1]
    numSamples = int(sys.argv[2])
    pointFileName = "point.txt"

    communicator = hms.BrokerLauncher().launch(configFile,
                                               0,
                                               1)
    communicator = communicator[0]

    model = hms.Model("python",
                      "/Users/kleiter/software/hms/build/unit_circle_indicator_function.py",
                      hms.StringVector([pointFileName]),
                      hms.CPU,
                      1)

    for i in range(0, numSamples):

        x1 = 2.0 * random.random() - 1.0
        x2 = 2.0 * random.random() - 1.0

        argument = PiArgument(x1, x2)
        inputFilter = PiInputFilter(pointFileName)
        outputFilter = PiOutputFilter()

        modelPackage = hms.ModelPackage(model,
                                        inputFilter,
                                        outputFilter,
                                        argument)

        communicator.send(modelPackage)

    counter = 0
    valueSum = 0.0
    while counter < numSamples:
        modelPackages = communicator.receive()
        counter = counter + len(modelPackages)
        for modelPackage in modelPackages:
            value = modelPackage.getValue()
            valueSum += value.value

    piEstimate = 4.0 * valueSum / float(numSamples);
    print("Estimate for pi: " + str(piEstimate))
```

11. HMS-VUMAT

A vectorized user material model (VUMAT) that incorporates HMS named the HMS-VUMAT has been developed that allows for the quick development of HMS multiscale models in finite element method (FEM) codes that adopt the VUMAT interface.[17] The HMS-VUMAT implements specialized Argument and Value classes that store the inputs and outputs of a VUMAT material model function call. The only required steps to implement a new multiscale model with HMS-VUMAT is to implement the application-specific input and output filters that set up and extract data from lower-scale models. The principal advantage of the HMS-VUMAT is that no code modifications are needed in the FEM upper-scale model and the implemented material model can be used interchangeably in any FEM code that supports the VUMAT standard. We refer the reader to the HMS-VUMAT user manual for further details of the HMS-VUMAT implementation.[17]

12. Conclusion

HMS enables the creation of adaptive distributed multiscale models from individual at-scale model components. A runtime system implemented in HMS coordinates evaluation of at-scale model components and allows for data communication between them. Many multiscale modeling applications have been developed using HMS, primarily for modeling of materials.

This document is a practical guide for using HMS to develop new multiscale modeling applications. We expect to continue development of HMS to provide additional functionality including more advanced adaptive surrogate modeling techniques, computational algorithms to handle internal microstructure evolution, and to treat stochasticity in multiscale models.

13. References

1. Knap J, Spear C, Leiter K, Becker R, Powell D. A computational framework for scale-bridging in multi-scale simulations. International Journal for Numerical Methods in Engineering. 2016;108(13):1649–1666.

2. Groen D, Knap J, Neumann P, Suleimenova D, Veen L, Leiter K. Mastering the scales: a survey on the benefits of multiscale computing software. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences. 2019;377(2142):20180147.

3. Leiter KW, Barnes BC, Becker R, Knap J. Accelerated scale-bridging through adaptive surrogate model evaluation. Journal of Computational Science. 2018;27:91–106.

4. Borodin O, Olguin M, Spear CE, Leiter KW, Knap J. Towards high throughput screening of electrochemical stability of battery electrolytes. Nanotechnology. 2015;26(35):354003.

5. Knap J, Spear CE, Borodin O, Leiter KW. Advancing a distributed multi-scale computing framework for large-scale high-throughput discovery in materials science. Nanotechnology. 2015;26(43):434004.

6. Barnes BC, Leiter KW, Larentzos JP, Brennan JK. Forging of hierarchical multiscale capabilities for simulation of energetic materials. Propellants, Explosives, Pyrotechnics. 2020;45(2):177–195.

7. Chen G, Yamashita H, Ruan Y, Jayakumar P, Knap J, Leiter KW, Yang X, Sugiyama H. Enhancing hierarchical multiscale off-road mobility model by neural network surrogate model. Journal of Computational and Nonlinear Dynamics. 2021;16(8):081005.

8. Chen G, Yamashita H, Ruan Y, Jayakumar P, Gorsich D, Knap J, Leiter KW, Yang X, Sugiyama H. Hierarchical MPM-ANN multiscale terrain model for high-fidelity off-road mobility simulations: a coupled MBD-FE-MPM-ANN approach. Journal of Computational and Nonlinear Dynamics. 2023;18(7):071001.

9. Leiter KW, Larentzos JP, Barnes BC, Brennan JK, Becker R, Knap J. Temporal scale-bridging of chemistry in a multiscale model: application to reactivity of an energetic material. Journal of Computational Physics. 2022;472:111682.

10. Crone JC, Knap J, Becker R. Multiscale modeling of 3D nano-architected materials under large deformations. International Journal of Solids and Structures. 2022;252:111839.

11. Weinan E, Engquist B, Li X, Ren W, Vanden-Eijnden E. Heterogeneous multiscale methods: a review. Communications in Computational Physics. 2007;2(3):367–450.

12. Barton NR, Bernier JV, Knap J, Sunwoo AJ, Cerreta EK, Turner TJ. A call to arms for task parallelism in multi-scale materials modeling. International Journal for Numerical Methods in Engineering. 2011;86(6):744–764.

13. Boost C++ Libraries Documentation. https://www.boost.org/doc/.

14. Knap J, Barton N, Hornung R, Arsenlis A, Becker R, Jefferson D. Adaptive sampling in hierarchical simulation. International Journal for Numerical Methods in Engineering. 2008;76(4):572–600.

15. Barton NR, Knap J, Arsenlis A, Becker R, Hornung RD, Jefferson DR. Embedded polycrystal plasticity and adaptive sampling. International Journal of Plasticity. 2008;24(2):242–266.

16. Dorr M. Adaptive sampling proxy application. US Department of Energy Office of Scientific and Technical Information (OSTI). 2012. https://www.osti.gov/servlets/purl/1231623.

17. Crone JC, Wilson ZA, Leiter KW, Knap J, Becker R. Users guide for the Hierarchical MultiScale-Vectorized User Material (HMS-VUMAT) Model 1.0. DEVCOM Army Research Laboratory; 2023 Jan. Report No.: ARL-TR-9633.

Appendix. HMS Configuration File Template (hmsConf.ini)

A template for the HMS configuration file is provided in this appendix. Settings that must be filled in by the user are indicated with FILL_IN. The file is configured to run the Broker or Adaptive Sampling module along with the lower-scale model evaluations on the localhost where the upper-scale model executes.

```
[AdaptiveSampling]

# Time threshold for object aging in adaptive sampling database.
AgingThreshold = 600000000

# Used by adaptive sampling to generate new points around a sample
# point in order to build a full kriging model when interpolating
# without derivatives (must be size of point dimension)
DeltaH = 1.0e-1 1.0e-1

# Hosts of adaptive sampling modules - an adaptive sampling module is
# launched on each of these hosts to perform interpolation
Hosts = localhost

# Path to libraries loaded by adaptive sampler in order to deserialize objects
Library = FILL_IN

# Maximum number of point/value pairs in a single kriging model.
MaxKrigingModelSize = 50

# Maximum number of kriging models to be pulled out of the database
# during the search for the best kriging model.
MaxNumberSearchModels = 4

# Maximum distance between the query point and the model for which
# interpolation is still attempted.
MaxQueryPointModelDistance = 1000.0

# The value of the coefficient multiplying the mean square error.
MeanErrorFactor = 1.0

# Name of the directory to use for storage of disk MTree data"
MtreeDirectoryName = .

# Number of threads performing interpolation in adaptive sampling
# module
NumberInterpolationThreads = 1

# Scheduler for load balancing of brokers
# Values: Load RoundRobin
Scheduler = RoundRobin

# Path to executable that launches adaptive sampling
StartAdaptiveSampling = FILL_IN (Located in HMS install /bin directory)

# Tolerance for interpolating
Tolerance = 1.0e-4

# Type
# Values: Null Kriging RandomForest
Type = Kriging

[Broker]

# Cleanup policy for broker
# Values: None, Custom, Fixed
Cleanup = None
CleanupFixedSize = 5

# Hosts of brokers - a broker is launched on each of these hosts to
# facilitate lower scale model evaluations
Hosts = localhost
```

```
# Path to libraries loaded by broker in order to deserialize objects
Library = FILL_IN

# Monitor type
# Values: Basic
Monitor = Basic

# Path where model scripts are stored and executed
OutputDirectory = FILL_IN

# Files used to populate broker resources - must be one set per broker host
Resources = localhost

# Resource types - must be one per resource file
# Values: CPU
ResourceTypes = CPU

# Resource list types - must be one per resource file
# Values: Localhost, Machinefile, List
ResourceListTypes = Localhost

# Scheduler type
# Values: Basic, PBS
Scheduler = Basic

# Path to executable that launches broker
StartBrokerPath = FILL_IN (Located in HMS install /bin directory)

# Type of broker
# Values: Basic
Type = Basic

[Communicator]

# Type of communicator used to communicate between algorithm, broker,
# and adaptive sampling
# Values: UnixSocket, TCPIPSocket, MPI
Type = TCPIPSocket

# Unix Socket File Name
UnixSocketFileName = /tmp/socket_file

[Logging]

# Logging level
# Values: Trace, Debug, Info, Warning, Error, Fatal
Level = Debug

# Logging file
File = /tmp/hms.log
```

List of Symbols, Abbreviations, and Acronyms

TERMS:

ASF –	Adaptive Sampling Framework
CPU –	central processing unit
DEM –	discrete element method
DFT –	density functional theory
DPD –	dissipative particle dynamics
EOS –	equation of state
FE^2 –	finite element squared
FEM –	finite element method
GPU –	graphics processing unit
HMM –	heterogeneous multiscale method
HMS –	Hierarchical MultiScale framework
HMS-VUMAT –	Hierarchical MultiScale framework Vectorized User MATerial model
HPC –	high-performance computing
MPI –	message passing interface
TCP/IP –	transmission control protocol/internet protocol
VUMAT –	vectorized user material model

1	DEFENSE TECHNICAL
(PDF)	INFORMATION CTR
	DTIC OCA
1	DEVCOM ARL
(PDF)	FCDD RLB CI
	TECH LIB
15	DEVCOM ARL
(PDF)	FCDD RLA NA
	J CRONE
	J KNAP
	K LEITER
	FCDD RLA CB
	R BECKER
	A TONGE
	FCDD RLA MB
	Z WILSON
	FCDD RLA TD
	R DONEY
	FCDD RLA TE
	J LLOYD
	M LOVE
	D HORNBAKER
	G VUNNI
	FCDD RLA WA
	B BARNES
	J BRENNAN
	J LARENTZOS
	B RICE
2	DEVCOM GVSC
(PDF)	P JAYAKUMAR
	Y RUAN
1	JOHNS HOPKINS UNIV
(PDF)	M SHIELDS