ARL-TR-9633 ● JAN 2023

User's Guide for the Hierarchical MultiScale - Vectorized User MATerial (HMS-VUMAT) Model 1.0

by Joshua C Crone, Zachary A Wilson, Kenneth W Leiter, Jaroslaw Knap, and Richard Becker

Approved for public release; distribution is unlimited.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-9633 ● JAN 2023

User's Guide for the Hierarchical MultiScale - Vectorized User MATerial (HMS-VUMAT) Model 1.0

by Joshua C Crone, Zachary A Wilson, Kenneth W Leiter, Jaroslaw Knap, and Richard Becker
DEVCOM Army Research Laboratory

Approved for public release; distribution is unlimited.

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

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED (From - To)**
January 2023	Technical Report	October 2019–September 2022

4. TITLE AND SUBTITLE	
User's Guide for the Hierarchical MultiScale - Vectorized User MATerial (HMS-VUMAT) Model 1.0	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	**5d. PROJECT NUMBER**
Joshua C Crone, Zachary A Wilson, Kenneth W Leiter, Jaroslaw Knap, and Richard Becker	**5e. TASK NUMBER**
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	**8. PERFORMING ORGANIZATION REPORT NUMBER**
DEVCOM Army Research Laboratory ATTN: FCDD-RLA-NA Aberdeen Proving Ground, MD 21005-5066	ARL-TR-9633
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

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

primary author's email: <joshua.crone.civ@mail.mil>. ORCIDs: Joshua Crone, 0000-0003-0856-9149

14. ABSTRACT

The HMS-VUMAT library provides a flexible interface for coupling multiscale material models into widely used finite element analysis (FEA) codes. The use of high-fidelity subscale models can improve accuracy and predictive capability, particularly for material systems with limited experimental data or high senstivity to microstructure. However, the implementation and execution of multiscale models is labor intensive and computationally expensive. The HMS-VUMAT library significantly reduces these costs by automating the communication, scheduling, load balancing, concurrent execution, and monitoring of subscale model evaluations with efficient use of high-performance computing resources. To create a new multiscale model, users can select any black-box subscale model and then create input and output filters within the HMS-VUMAT library. The library simplifies the creation of these filters and provides tools for debugging and profiling multiscale simulations. The resulting multiscale model will be compatible with any FEA code that supports the Abaqus VUMAT interface. This enables the use of high-fidelity material models in commercial, government, and other widely available FEA tools.

15. SUBJECT TERMS

multiscale modeling, finite element analysis, high-performance computing, Network, Cyber, and Computational Sciences

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					Joshua C Crone
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER (Include area code)**
Unclassified	Unclassified	Unclassified	UU	43	410-306-2156

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

Contents

List of Figures iv

Acknowledgments vi

1. Introduction 1

2. Overview of the HMS Framework 3

3. Overview of the HMS-VUMAT Library 4

4. Quick-Start Guide to Creating an HMS-VUMAT Multiscale Model 5

5. How to Build 11

6. How to Run 14

7. Additional Details for Creating a New Model 16

8. Debugging 19

9. Profiling 22

10. Description of the HMS-VUMAT Files and Classes 24

11. Conclusion 30

12. References 32

List of Symbols, Abbreviations, and Acronyms 33

Distribution List 34

List of Figures

Fig. 1 Schematic of a generic two-scale model, where an upper-scale model (F) passes arguments to a lower-scale model (f) through a filter (G). Relevant output from the lower-scale model is passed back to the upper-scale model through another filter (g). 2

Fig. 2 (a) Schematic of the four components of a two-scale HMS model where the HMS framework and filters must be directly incorporated into the source code of the upper-scale FEA solver. (b) Schematic of a two-scale HMS model where the HMS-VUMAT interface elminates the need for source code access of the upper-scale FEA solver. 4

Fig. 3 Flow of information in the HMS-VUMAT model. Through the standard VUMAT function call within the FEA solver, HMS-VUMAT receives information such as the deformation (F), loading rate (Δt), state variables (SVs), etc., from a block of material point (MP) calculations. The HMS-VUMAT converts this information to a VumatArgument that can be communicated and evaluated through HMS. The input filter converts the information contained in the VumatArgument to corresponding input files for the lower-scale model. After completion of the lower-scale model, an output filter extracts and processes the output file to determine the Cauchy stress (σ), energies (e), and updated SVs. These quantities are packaged into a VumatValue object and sent back to the HMS-VUMAT, where its content is returned to the FEA solver through the VUMAT function call. The user only needs to provide the lower-scale model and create the input/output filters. All communication and the remaining packing/unpacking of data are handled automatically. 6

Fig. 4 User-defined input filter template file included in the HMS-VUMAT source code. This file can be modified by a user to quickly implement a multiscale material model. 7

Fig. 5 User-defined output filter template file included in the HMS-VUMAT source code. This file can be modified by a user to quickly implement a multiscale material model. 8

Fig. 6 Example implementation of an input filter for a lower-scale model that expects the elastic constants and the deformation gradient in two separate files 10

Fig. 7 Example implementation of an output filter for a lower-scale model that writes the Cauchy stress to an output file 11

Fig. 8 Output from profiling across all time steps for HMS simulations with resource pools consisting of one and four CPU nodes. (a) Simulation time vs. wall-clock time. (b) Wall-clock time to complete each upper-scale time step. 23

Fig. 9 Output from profiling a single time step within 0.02 s of the target simulation time of 1.5 s. Plots in the top row correspond to a simulation with one CPU node of available resources (38 cores). Bottom row corresponds to a simulation with four CPU nodes of available resources (158 cores). The plots in the left column show when the input filter, lower-scale executable, and output filter are active throughout the upper-scale time step for each of the 64 evaluations. The plots in the right column show how many lower-scale executables are running concurrently throughout the upper-scale time step. 25

Fig. 10 Close-up view of Fig. 9a showing full evaluation cycles for the first nine evaluations 26

Acknowledgments

The Hierarchical MultiScale - Vectorized User MATerial (HMS-VUMAT) interface was developed through the financial support of the Army Research Laboratory's Enterprise for Multiscale Research in Materials and the Department of Defense (DoD) High Performance Computing Modernization Program (HPCMP) under User Productivity Enhancement, Technology Transfer, and Training contract no. 47QFSA18K0111, Task Order ID04180146. Computational support, through a grant of computer time, was provided by the DoD HPCMP.

1. Introduction

Multiscale material modeling (MMM) is an approach to developing high-fidelity models by combining various "at-scale models" that capture salient phenomena at disparate spatial and/or temporal scales. This approach is in contrast to analytically and empirically based material models, which rarely consider subscale phenomena and are typically fit through extensive experimental testing. Through the incorporation of high-fidelity subscale models, MMM can provide reliable models for novel materials where experimental data is limited or nonexistent. MMM may also be the only choice when the structure-property relationship of a material is too complex to be captured analytically or empirically. Despite the benefits of MMM, the computational cost can quickly explode, depending on the range of relevant scales and the computational cost of the subscale models. Furthermore, integration of disparate at-scale models is typically labor intensive.

The US Army Combat Capabilities Development Command Army Research Laboratory developed the Hierarchical MultiScale (HMS) framework to significantly simplify the labor-intensive process of developing and implementing multiscale models.[1] The framework enables scale-bridging through seamless combinations of at-scale models. This is accomplished by abstracting out many of the computational science tasks ubiquitous across multiscale modeling, such as scheduling, load balancing, and monitoring of at-scale evaluations, as well as communication between at-scale models. All of these tasks are performed to efficiently utilize high-performance computing (HPC) resources. The framework has been extended to include adaptive machine learning models, which significantly reduce computational costs.[2]

The HMS framework is inspired by the heterogeneous multiscale method,[3] a flexible mathematical framework where an upper-scale model acquires missing data through on-the-fly evaluations of lower-scale models. Constraints are communicated to the lower-scale to be consistent with the local state of the upper-scale model. The data required by the upper-scale model is extracted from the lower scale through homogenization or other data reduction techniques. This multiscale modeling paradigm is illustrated for a generic two-scale model in Fig. 1.

Within the HMS framework, the lower-scale models can be treated as black boxes. The only requirements are that inputs associated with the upper-scale state can be

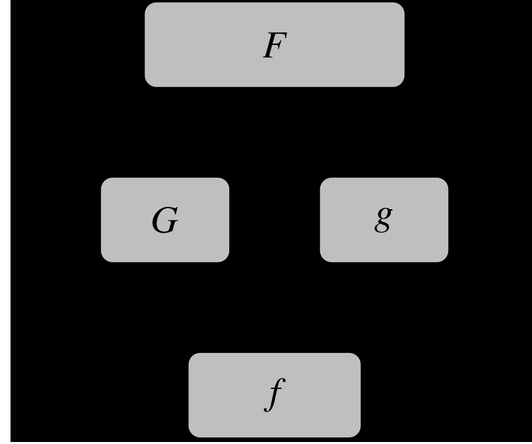

Fig. 1. Schematic of a generic two-scale model, where an upper-scale model (F) passes arguments to a lower-scale model (f) through a filter (G). Relevant output from the lower-scale model is passed back to the upper-scale model through another filter (g).

communicated through the command line or input files and that the quantities of interest from the lower-scale model can be written to an output file. However, the HMS framework requires direct integration into the source code of the upper-scale model. This poses a significant limitation when integrating multiscale models into codes that restrict source access to protect proprietary knowledge or national security. To overcome this limitation in structural mechanics applications, we have developed a library that allows the HMS framework to communicate with an upper-scale model through the Abaqus Vectorized User MATerial (VUMAT) interface.[4] The VUMAT interface has been adopted by many finite element analysis (FEA) tools to enable the integration of user-defined material models. Through the HMS-VUMAT interface, a multiscale material model developed in one upper-scale FEA tool can be seamlessly incorporated into any other FEA tool that supports the VUMAT interface. Through the HMS framework, multiscale materials models have already been developed for a wide range of applications:

- FE^2 model of a composite fiber under impact loading[1]
- Equation of state (EOS) calculations for energetic material using dissipative particle dynamics (DPD)[2]
- Discrete element method (DEM) of soil compaction[5]
- Reactive chemistry of an energetic material using DPD[6]

- Crystal plasticity in magnesium by spectral homogenization
- FE^2 model of nanoscale architected trusses[7]

The variety of applications and lower-scale methods in the above list highlights the flexibility of the HMS framework and the HMS-VUMAT interface. This user guide will aim to highlight the ease at which new multiscale models can be incorporated into commercial and other widely available FEA tools.

2. Overview of the HMS Framework

This section provides a brief overview of the HMS framework. For a detailed description, the reader is referred to Knap et al.[1] and the HMS software documentation (forthcoming). We will limit our discussion to structural mechanics applications, where the upper-scale model is an FEA solver and the information communicated between the upper and lower scales includes stresses, strains, state variables (SVs), temperature, and energy. This is the application domain supported by the VUMAT interface. However, we emphasize that the HMS framework is agnostic to the application, choice of upper-scale model, and the information communicated between scales.

The four components of a typical two-scale HMS model are illustrated in Fig. 2a. The model-specific components are shown in blue and must be defined by the user. The lower-scale model can be anything that produces the output needed by the upper-scale model (stresses and energy within the VUMAT interface), either directly or after additional postprocessing. The execution and monitoring of the lower-scale model are handled by the HMS framework, particularly the HMS broker, which serves as the primary traffic controller for all lower-scale evaluations. The other user-defined component is a pair of input/output filters. The input filter takes the constraints provided by the upper scale and generates input files with the corresponding boundary conditions and simulation parameters for the lower-scale model. The input filter is called by the HMS framework before each lower-scale evaluation. The output filter reads from the output file generated by the lower-scale model and performs any necessary postprocessing to compute the quantities of interest for the upper-scale model. The upper-scale FEA solver is shown in black. Each evaluation request from the upper-scale model must be packaged along with the filters and sent to the HMS framework. This requires modification of the FEA

source code whenever a new model is incorporated. As previously noted, this may not be possible with many restricted-access FEA codes.

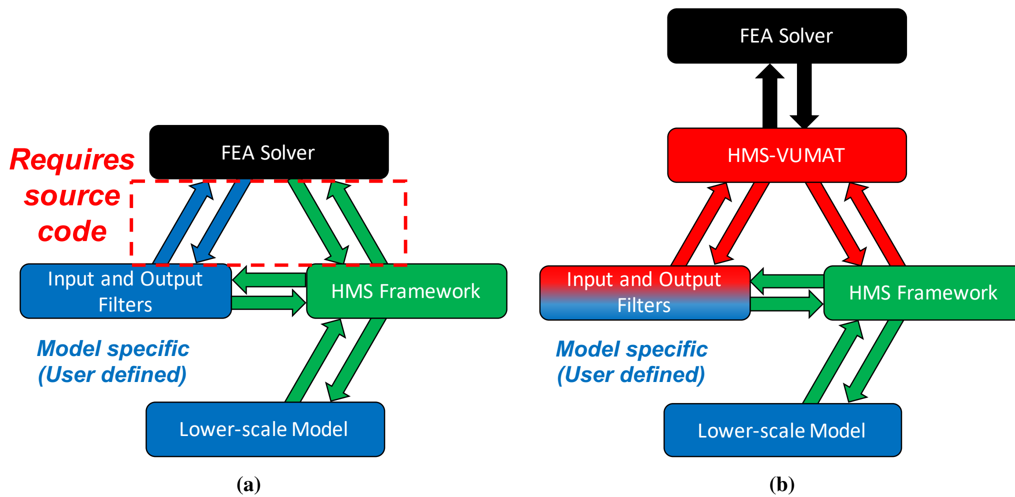

Fig. 2. (a) Schematic of the four components of a two-scale HMS model where the HMS framework and filters must be directly incorporated into the source code of the upper-scale FEA solver. (b) Schematic of a two-scale HMS model where the HMS-VUMAT interface elminates the need for source code access of the upper-scale FEA solver.

3. Overview of the HMS-VUMAT Library

The HMS-VUMAT library can be thought of as an extension of the HMS framework that serves as a middle layer between the FEA solver and HMS, as shown in Fig. 2b. All lower-scale evaluation requests are sent from the upper-scale FEA solver through the standard VUMAT interface, eliminating the need to modify the source code of the FEA solver. The evaluation requests are packaged with the model-specific input/output filters before passing the packages down to the HMS framework, where the evaluations are scheduled, executed, and monitored for completion. Although a user is still required to create input/output filters when developing a new model, the HMS-VUMAT library greatly simplifies this process. The HMS-VUMAT also provides a seamless way to incorporate the newly created input/output filters into the evaluation requests sent to the HMS framework.

The flow of information for an evaluation cycle of the HMS-VUMAT model is shown in Fig. 3. It begins with the standard VUMAT function call within the FEA solver. This sends the deformation, loading time, SVs, and other information provided through the VUMAT interface to the HMS-VUMAT for a block of material

points (MPs). This information is packaged into a structure called the "VumatArgument," which stores the input arguments associated with the VUMAT into an object that can be communicated throughout the HMS framework. This object is combined with the user-defined input/output filters to create a ModelPackage for each MP, which is the evaluation unit of the HMS framework. Once HMS receives the ModelPackages, it schedules each ModelPackage for execution. Input filters are applied for each MP evaluation, where the information contained in the VumatArgument is used to generate the input files required for the lower-scale model. HMS executes the lower-scale models as resources become available and applies the output filters to extract the stress (σ), energies (e), and updated SVs from the information provided in the output files of the lower-scale model. The quantities are packaged into a "VumatValue" object and communicated back to the HMS-VUMAT. The HMS-VUMAT waits to receive these VumatValues and extracts the data to pass back to the FEA solver through its function call to the VUMAT. A user only needs to worry about the mapping from the VumatArgument object to the model input files (the input filter) and the mapping from the model output files to the VumatValue object (the output filter). The rest is handled automatically.

4. Quick-Start Guide to Creating an HMS-VUMAT Multiscale Model

As discussed previously, creating a new multiscale material model within the HMS-VUMAT library involves creating an input filter that converts the inputs from the VUMAT to input files for the lower-scale model and an output filter that converts the output files from the lower-scale model to the quantities returned by the VUMAT. To simplify this process, template files for both filters are provided in the source code in the **model/userDefined** directory. In these files, a commented block is located anywhere a user might need to modify the code for their application. Figures 4 and 5 contain the implementation file templates for the input and output filters, respectively, with the comment blocks highlighted in different colors. The only changes that are required to these templates for most MMM applications would be in the green boxes that define the "apply()" function. This is the function that is called by HMS to apply the filters. Depending on the code implemented in the green boxes, additional headers may be included in the yellow boxes. The red boxes can be ignored in most applications. Once the green boxes have been filled in with the specific user-defined code to communicate with the lower-scale model,

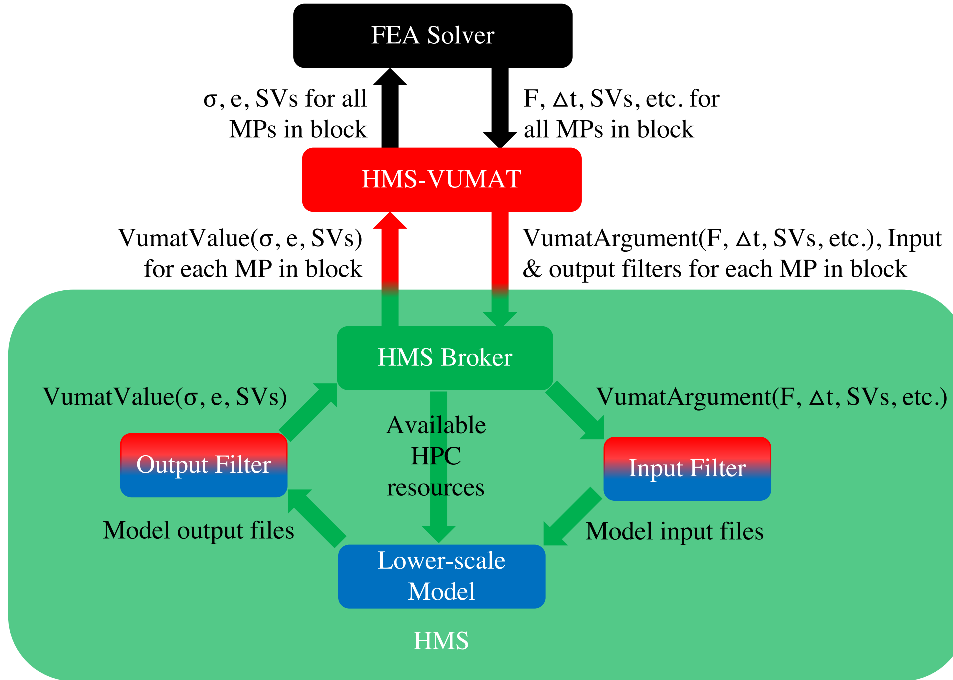

Fig. 3. Flow of information in the HMS-VUMAT model. Through the standard VUMAT function call within the FEA solver, HMS-VUMAT receives information such as the deformation (F), loading rate (Δt), state variables (SVs), etc., from a block of material point (MP) calculations. The HMS-VUMAT converts this information to a VumatArgument that can be communicated and evaluated through HMS. The input filter converts the information contained in the VumatArgument to corresponding input files for the lower-scale model. After completion of the lower-scale model, an output filter extracts and processes the output file to determine the Cauchy stress (σ), energies (e), and updated SVs. These quantities are packaged into a VumatValue object and sent back to the HMS-VUMAT, where its content is returned to the FEA solver through the VUMAT function call. The user only needs to provide the lower-scale model and create the input/output filters. All communication and the remaining packing/unpacking of data are handled automatically.

the HMS-VUMAT library can be compiled (see Section 5) and is ready for use in any FEA code supporting the Abaqus VUMAT interface.

To demonstrate the creation of the input and output filters, we use a simple lower-scale code that computes stress by the isotropic Saint Venant-Kirchoff (SVK) hyper-elastic model. While the model is trivial and would not require the HMS framework in practice, it is a convenient model to demonstrate the basic components of the HMS-VUMAT filters. The SVK code is included with the HMS-VUMAT source in the file **model/svk/***svkModel.cc* and is compiled with the HMS-VUMAT so that the model can be run "out of the box." In practical applications, the lower-scale code will be compiled separately. The SVK code requires two input files, one called *elas-*

```
// File:       UserInputFilter.cc
// Package     model/userDefined
// A template for the implementation file of a new user-defined
// input filter for an HMS-VUMAT model
#if defined(HAVE_CONFIG_H)
#include <arlhmsvumat_config.h>
#endif // HAVE_CONFIG_H
#include "UserInputFilter.h"
/******************************************************************
 * Place for user to include any header files needed to perform   *
 * input filter functions. At a minimum, user will likely need:   *
 ******************************************************************/
#include <fstream>
#include <iostream>
HMS_SERIALIZATION_EXPORT(arl::hms::UserInputFilter)
namespace arl {
  namespace hms {
    // Constructor.
    UserInputFilter::UserInputFilter(const std::string &
                                     hmsConfFileName) :
      VumatInputFilter(hmsConfFileName) {
/******************************************************************
* Place for user to define any behavior that should occur in the  *
* constructor. Most likely task is defining the data members,     *
* either explicitly or by reading values from the hmsConf.ini file*
******************************************************************/
       return;
    }

    // Apply User-defined input filter
    void
    UserInputFilter::apply(const ArgumentPointer & argument,
                           const std::string     & directory) const {
      // HMS sends the argument struct as a generic argument. We
      // need to cast it specfically as a VUMAT argument. This stuct
      // contains all of the input arguments from the VUMAT for this
      // material point.
      VumatArgumentPointer vumatArgumentPtr =
        boost::dynamic_pointer_cast<arl::hms::VumatArgument>(argument);
/******************************************************************
* Place for user to define the mapping from the VumatArgument to  *
* the input files for the lower-scale model.                      *
*                                                                 *
* NOTE: the "directory" argument passed into this function is the *
* unique evaluation directory for this material point. The lower- *
* scale model will be executed in this directory, so the input    *
* files should be written in this directory.                      *
******************************************************************/
      return;
    }
  }
}
```

Fig. 4. User-defined input filter template file included in the HMS-VUMAT source code. This file can be modified by a user to quickly implement a multiscale material model.

```
// File:      UserOutputFilter.cc
// Package    model/userDefined
// A template for the implementation file of a new user-defined
// output filter for an HMS-VUMAT model
#if defined(HAVE_CONFIG_H)
#include <arlhmsvumat_config.h>
#endif // HAVE_CONFIG_H
#include "UserOutputFilter.h"
/*******************************************************************
 * Place for user to include any header files needed to perform     *
 * output filter. A user will most likely need:                     *
 *******************************************************************/
#include <fstream>
#include <iostream>
#include <vector>
#include <sstream>
#include <exception/IOError.h>
HMS_SERIALIZATION_EXPORT(arl::hms::UserOutputFilter)
namespace arl {
  namespace hms {
    // Constructor.
    UserOutputFilter::UserOutputFilter(const std::string &
                                       hmsConfFileName):
      VumatOutputFilter(hmsConfFileName) {
/*******************************************************************
 * Place for user to define any behavior that should occur in the   *
 * constructor. Most likely tasks are defining the data members,    *
 * either explicitly or by reading values from the hmsConf.ini file *
 *******************************************************************/
      return;
    }

    // Apply User-defined output Filter
    ValuePointer
    UserOutputFilter::apply(const std::string & directory,
                            const std::string &,
                            const ArgumentPointer & argument) const {
      // create an empty return value
      VumatValuePointer returnValue(new VumatValue);

      // HMS sends the argument struct as a generic argument. We need
      // to cast it specfically as a VUMAT argument. This stuct
      // contains all of the input arguments from the VUMAT for this
      // material point.
      VumatArgumentPointer vumatArgumentPtr =
      boost::dynamic_pointer_cast<arl::hms::VumatArgument>(argument);
/*******************************************************************
 * Place for user to define the mapping from the lower-scale output *
 * files to the VumatValue struct.                                  *
 *                                                                  *
 * NOTE: the "directory" argument passed into this function is the  *
 * unique evaluation directory for this material point. The lower-  *
 * scale model will be executed in this directory, so the output    *
 * files should be read from this directory.                        *
 *******************************************************************/
      return returnValue;
      }
  }
}
```

Fig. 5. User-defined output filter template file included in the HMS-VUMAT source code. This file can be modified by a user to quickly implement a multiscale material model.

tic_props.txt that contains the elastic constants and another called *defgrad.inp* that contains the current deformation gradient. The model writes an output file called *stress.out* containing the Cauchy stress in the reference frame.

The input filter for the SVK code is presented in Fig. 6. The "apply()" function takes the VumatArgument struct and the full path to the unique evaluation directory for the particular MP as the input arguments. In this example, it is assumed that the elastic constants have been defined in the "props" array from the standard VUMAT interface. The first and second components of the array are extracted from the VumatArgument and written to an input file called *elastic_props.txt* in the format that the SVK code expects. Similarly, the current deformation gradient is extracted from the VumatArgument and written to an input file called *defgrad.inp* in the format expected by the SVK code. Note the indexing of the "defgradNew" array, which matches the ordering employed by the VUMAT interface. The reader is directed to the Abaqus VUMAT documentation[8] for details on the index ordering for full and symmetric tensors. Once both files have been written, the input filter is done.

An example of an output filter for the SVK lower-scale code is shown in Fig. 7. Once again, required changes to the template are isolated to the "apply()" function. First, the output file from the SVK code is opened, using the "directory" variable to get the full path to the output for the specific MP. An error check is added to confirm that the output file was written. The HMS framework includes a number of C++ exceptions, which can be thrown in the filters and are caught by the HMS broker. If an "IOError" exception is thrown, the HMS broker will try to rerun the lower-scale evaluation five times before terminating. This is to provide fault tolerance in case the error was due to file system or other non-repeatable issues. All other exception types will cause the HMS simulation to immediately terminate with the associated error message printed in the standard error stream. After ensuring the output file was opened successfully, the stress from the SVK code is read and stored in a vector. In the SVK code, the stress is written in the order that the Abaqus VUMAT expects for a symmetric tensor. This stress is then used to update the stress in the VumatValue struct, which is what gets passed back to the HMS-VUMAT. In this example there are no SVs to update, and the energy is not calculated. If the user's lower-scale model requires updates to these quantities, their values in the VumatValue struct would get updated here as well. Finally, the stress is rotated to the corotational frame by a built-in function "rotate()". This function extracts the deformation gra-

```
// Get the full path to where the SVK input files will
// be written
std::string fullDefGradFilePath=directory+"/defgrad.inp";
std::string fullPropsFilePath=directory+"/elastic_props.txt";

std::ofstream outputFile;
outputFile.precision(18);

// Print the elastic properties. The VUMAT interface supports
// a variable-sized array of properties that can be specified
// in the input file. We use that "props" array to set the
// properties.
outputFile.open(fullPropsFilePath.c_str());

outputFile <<  "E = "   << vumatArgumentPtr->props[0]
           << std::endl
           <<  "NU = " << vumatArgumentPtr->props[1]
           << std::endl;

outputFile.close();

// Write the deformation gradient to an input file that will
// be read in by the SVK lower-scale model
outputFile.open(fullDefGradFilePath.c_str());

outputFile <<  vumatArgumentPtr->defgradNew[0] << " " // Fxx
           <<  vumatArgumentPtr->defgradNew[3] << " " // Fxy
           <<  vumatArgumentPtr->defgradNew[8] << " " // Fxz
           <<  vumatArgumentPtr->defgradNew[6] << " " // Fyx
           <<  vumatArgumentPtr->defgradNew[1] << " " // Fyy
           <<  vumatArgumentPtr->defgradNew[4] << " " // Fyz
           <<  vumatArgumentPtr->defgradNew[5] << " " // Fzx
           <<  vumatArgumentPtr->defgradNew[7] << " " // Fzy
           <<  vumatArgumentPtr->defgradNew[2] << " ";// Fzz

outputFile.close();
```

Fig. 6. Example implementation of an input filter for a lower-scale model that expects the elastic constants and the deformation gradient in two separate files

```
// open output file for reading
std::string fullOutputFilePath = directory + "/stress.out";
std::ifstream outputFile(fullOutputFilePath.c_str());

// check whether output file was opened
if(!outputFile) {
  const std::string message("Error opening output file.");
  throw IOError(message);
}

// read cauchy stress from file
std::string line;
std::getline(outputFile, line);
std::stringstream stress_stream(line);
std::vector<double> outputStress(6);
stress_stream >> outputStress[0] >> outputStress[1]
              >> outputStress[2] >> outputStress[3]
              >> outputStress[4] >> outputStress[5];

outputFile.close();

// Update the stress in the value struct that will be returned
// by HMS
returnValue->stressNew = outputStress;

// Rotate into corotational frame, which is what the VUMAT
// expects
rotate(vumatArgumentPtr, returnValue);
```

Fig. 7. Example implementation of an output filter for a lower-scale model that writes the Cauchy stress to an output file

dient and stretch tensor from the VumatArgument to compute the rotation matrix by polar decomposition. The function then extracts the "stressNew" variable from the VumatValue and applies the rotation, replacing the value of "stressNew" with the rotated value. This completes the output filter.

The code blocks shown in Figs. 6 and 7 are included in the template files so that the filters can be compiled and tested with the SVK lower-scale model without any user modification. The blocks of code are surrounded by "#if 1" preprocessor macros that can either be turned off or deleted entirely by the user when they implement their own functions.

5. How to Build

Before extending the HMS-VUMAT for a new lower-scale model, it is recommended to build HMS-VUMAT "out of the box" to troubleshoot any compiling issues prior to introducing new functions and files. The only required dependency

is the HMS framework, which must be built prior to building the HMS-VUMAT interface. See the documentation of the HMS framework (forthcoming) for detailed build instructions. A second dependency is required if the intended upper-scale code is parallelized through the Message Passing Interface (MPI). In which case, the HMS-VUMAT library must also be compiled with MPI, so that each process of the HMS-VUMAT can determine its task rank in order to communicate with the HMS framework.

An "out-of-source build" is recommended, where the configure and compile commands are performed in a separate build directory, rather than in the directory containing the source files. This keeps the temporary files generated during configuration and compiling separate from the source files. When inside the build directory, the following command will perform the configure step and generate the makefiles:

```
$ <HMSVUMAT_SRC>/configure --prefix=<HMSVUMAT_INSTALL>\
  --enable-shared --with-hms=<HMS_INSTALL> \
  --with-mpi=<MPI_INSTALL>
```

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

- <**HMSVUMAT_SRC**> = top-level directory for the source files of the HMS-VUMAT code
- <**HMSVUMAT_INSTALL**> = directory where the headers, libraries, binaries, and wrapper script should be installed
- <**HMS_INSTALL**> = top-level directory where the HMS framework was installed. HMS headers and libraries should be located in <**HMS_INSTALL**>**/include/** and <**HMS_INSTALL**>**/lib/**, respectively
- <**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 also possible to use the "–with-mpi" flag without passing an <**MPI_INSTALL**> path. In which case, 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 is shown below using the intel MPI compilers:

```
$ <HMSVUMAT_SRC>/configure --prefix=<HMSVUMAT_INSTALL>\
  --enable-shared --with-hms=<HMS_INSTALL> \
  --with-mpi CXX=mpiicpc
```

All MPI-related arguments can be ignored in the build process if the upper-scale code is serial. Note that even when the upper-scale code is run in serial, the HMS-VUMAT framework can provide significant parallelization by allowing the numerous evaluations of the lower-scale model to be run concurrently across HPC resources.

After successfully running the *configure* script, the following commands will compile HMS-VUMAT and install the headers, libraries, scripts, and executables into the specified <**HMS_VUMAT_INSTALL**> directory.

```
$ make
$ make install
```

There are two ways that user-defined VUMAT subroutines are incorporated into FEA simulations. One is by linking to a library that defines the VUMAT subroutine. This can be accomplished with HMS-VUMAT by linking to <**HMS_VUMAT_INSTALL**>**/lib/***libhmsvumat.so*. The other method is by compiling the VUMAT subroutine into the FEA code, either during the build process of the FEA code or at runtime. For this method, the *Vumat.cpp* wrapper in <**HMS_VUMAT_INSTALL**> **/share/** should be compiled by the FEA code while linking to the support libraries of the HMS-VUMAT in <**HMS_VUMAT_INSTALL**>**/lib/**.

Abaqus is an example of an FEA code that requires the *Vumat.cpp* file to be compiled at runtime. Defining environment variables is the easiest way to provide the libraries and headers required to compile *Vumat.cpp*. For example,

```
$ setenv CPATH <HMS_VUMAT_INSTALL>/include:$CPATH
$ setenv CPATH <HMS_INSTALL>/include:$CPATH
$ setenv LD_PRELOAD <HMS_VUMAT_INSTALL>/lib/libhmsvumat
  _driver.so
$ setenv LD_LIBRARY_PATH <HMS_VUAMT_INSTALL>/lib:${LD_L
  IBRARY_PATH}
```

```
$ setenv LD_LIBRARY_PATH <HMS_INSTALL>/lib:${LD_LIBRARY
  _PATH}
```

Conflicts may arise if the MPI implementation used to compile Abaqus, HMS, and HMS-VUMAT are not consistent. If any MPI-related errors are present when Abaqus attempts to compile *Vumat.cpp*, it is recommended that HMS and HMS-VUMAT be recompiled without the "–with-mpi" flag. There is no loss of functionality with this configuration, as HMS can still communicate across compute nodes through network sockets and HMS-VUMAT will acquire the necessary MPI functionality when compiled against the Abaqus MPI implementation at runtime.

6. How to Run

Users are referred to the to VUMAT documentation[8] and documentation of their specific FEA code for general information on how to specify a user-defined VUMAT subroutine. The input file for the FEA code should not require further modifications, beyond the requirements for running any user-defined VUMAT.

In addition to the standard input file of the specific FEA code, an HMS configuration file must be included. The file follows the INI configuration file format with "name=value" pairs to specify parameters. A generic example is provided in <**HMS_VUMAT_INSTALL**>**/share/***hmsConf.ini* and an example for the SVK model is provided inside the svk source directory. The [Model] section of the INI file contains all of the parameters that are specific to the HMS-VUMAT interface and the input/output filters. The only required parameter within the [Model] section is "Executable," which must be set to the full path of the lower-scale model executable or to a script that calls the lower-scale executable. Optional parameters include "ExecutePrefix" for any commands that should proceed the executable, such as mpirun, and "ExecuteArguments" for any arguments that come after the executable. If each lower-scale evaluation uses more than one CPU core, the "NumberResources" parameter must be set to the number of required CPU cores per evaluation. SVs can be initialized through the upper-scale code or by providing an array to the "stateInit" parameter. When specifying an array of size n in the INI file, the first n SVs will be initialized to the corresponding array. The rest of the SVs will remain at their default value determined by the upper-scale code. Additional parameters can be added to the [Model] section to be read by the user's input/output filters.

The [Broker], [Communicator], [Logging], and [AdaptiveSampling] sections are described in detail in the documentation for the HMS framework (forthcoming). The only parameters that must change from run to run describe the resources to be used for the particular simulation and the output directory where files for each lower-scale evaluation are created. A Python script is provided to aid in automating the process of filling in these values with the following command:

```
$ <HMS_VUMAT_INSTALL>/share/generateHMSConf_noAS.py \
  hmsConf.ini machinefile N outputDir [M] > \
  hmsConf_final.ini
```

where hmsConf.ini is an HMS configuration file with the [Model] section, [Broker].Library and [Broker].StartBrokerPath filled in. The machinefile should contain a list of all of the resources available for that simulation. For example, $PBS_NODEFILE can be used with the PBS queuing system. N is an integer indicating the number of HMS brokers to launch. Typically, this is equal to the number of processes the upper-scale FEA code is run on (M). However N can be any value less than or equal to M. If N is not equal to M, then M should be specified as the optional fifth argument to the Python script. The HMS documentation provides additional information on situations in which the number of HMS brokers does not match the number of upper-scale processes. The outputDir is the location where files for each lower-scale evaluation are created. Each lower-scale evaluation is executed within the unique subdirectory of outputDir. Depending on the number of finite elements and time steps in the upper-scale simulation, millions of files may be created over the course of a simulation. Therefore, outputDir should be on a file system capable of handing the creation and storage of many files, such as a scratch directory or virtual file system, if available. The size of outputDir can be controlled by specifying the following in the HMS INI file:

```
[Broker]
Cleanup=Fixed
CleanupFixedSize=2000
```

where 2000 can be replaced with any integer corresponding to the maximum number of evaluation directories to be stored in each broker's outputDir. Once all parameters have been defined, the full path to the final INI file must be set in the $HMS_CONF_FILE environment variable. This environment variable is used by

the HMS-VUMAT to find the INI file.

Once the INI file is complete and the $HMS_CONF_FILE environment variable is set, launching of the FEA executable is performed in the same way as any simulation with a user-defined VUMAT subroutine. Within the outputDir, a separate directory is created for each broker and within each broker directory, separate directories are created for each process ID. This allows for the same outputDir to be used for multiple simulations while keeping the evaluation directories separate.

7. Additional Details for Creating a New Model

Section 4 discusses the easiest way to develop a simple multiscale model with the HMS-VUMAT interface. In this section, additional details are provided for developing more sophisticated models. Yet this section only scratches the surface of what can be done with the HMS-VUMAT. The extensibility of the HMS-VUMAT framework provides tremendous flexibility in the operations of the filters and the interactions with the lower-scale model.

The first step in developing a new multiscale material model is determining the lower-scale model. This model should be capable of taking the relevant parameters from the VUMAT input arguments (contained in the VumatArgument object) and writing an output file containing values that can be postprocessed to determine the relevant output parameters of the VUMAT interface (i.e., corotational Cauchy stress, internal energy, dissipated inelastic energy and/or updated SVs). The simplest mode for running the lower-scale model is where the command-line arguments are the same throughout the entire simulation and the inputs that change for each evaluation are communicated to the lower scale through input files. This mode of operation will be assumed in the remainder of this section. Dynamically changing the command-line arguments for each lower-scale evaluation (e.g., passing the deformation gradient through the command line) is possible, but requires modifications to the ModelFactory and is therefore an advanced feature. Instructions of how to create models that operate in this mode will be forthcoming in a later version of the user's guide. For the input filter's base class (VumatInputFilter), the apply function simply writes the entire VumatArgument object to a file named *argument.txt*, which could then be read in and parsed by a corresponding lower-scale executable. Similarly, the output filter's base class (VumatOutputFilter) reads a file named *value.txt*, which should be written by the lower-scale executable and should

already contain the output parameters expected by the VUMAT interface. No additional postprocessing is performed in the base class of the output filter. While it is possible to operate the HMS framework with the filters' base classes, it is expected that the vast majority of lower-scale models will require a derived version of both input and output filters with customized operations. Construction of these derived filters is expected to constitute the majority of the effort required for a user of HMS to implement a new model.

Examples of derived classes for the input and output filters are contained in the **model/svk/** directory. These filters are compatible with the SVK executable described in Section 4 and perform the same basic operations as the examples contained in the **model/userDefined** templates. However, they introduce additional sophistication supported by the HMS-VUMAT interface. First, these filters demonstrate how to create filters outside of the **model/userDefined** directory. This may be desired if the user wants to rename their filters to something more descriptive or if they want to create multiple models. Users can create new directories within the source code in the **model/**, such as **model/svk/**, to organize their filters. Each input and output filter must have a header file ("*.h") containing the declarations of the functions and data members of the class and a corresponding implementation file ("*.cc"). The naming conventions for files and filter classes are up to the user. A good starting point is to copy the contents of **model/userDefined** and replace all instances of "UserDefinedInputFilter" and "UserDefinedOutputFilter" with the desired names.

The filters in **model/svk** demonstrate many additional features compared to those in **model/userDefined**. One such feature is the reading of the HMS INI file to specify parameters at runtime, rather than hard coding them into the filters. For example, the SVK executable defined in *svkModel.cc* can take file names for the input and output files as command line arguments. The example INI file in **model/svk/***hmsConf_svk.ini* shows how these command line arguments can be passed to the lower-scale executable. However, the modified file names must also get passed to the filters so that the files can be written during the input filter and read during the output filter. In the **model/svk** filters, this is accomplished by setting parameters, such as "ElastPropsInputFile", in the INI file. Both numeric and non-numeric (i.e., strings) can be passed between the HMS INI file and the filters, providing tremendous flexibility in modifying the behavior of the filters at runtime. A verbose flag is

also used by the SVK filters to indicate whether additional debugging information should be printed to files. Any number of flags can be added in this manner to turn on or off certain features of the user-defined filters.

Reading of the INI file can be performed directly in the filters' "apply()" functions. However, this can lead to numerous file I/O operations since the functions are called for each MP evaluation. In the SVK filters, the reading of the INI file is performed in the constructor, which is only called once per upper-scale VUMAT function call. The values are stored as data members of the filter classes so that they can be accessed in the "apply()" functions. This design choice is purely an optimization and does not provide additional functionality over a filter that reads the INI file during "apply()". But it provides a demonstration of how data members can be added to the filter classes. First, the data members must be declared in the header file. In this example, variables with the prefix "d_" indicate data members. Then, additional "archive" commands must be added to the "serialize()" function, which is a templated member function of the filter classes defined directly in the header files. The "serialize()" function enables these objects to be passed between compute nodes within the HMS framework through serialization and deserialization. The order of the archive commands must match the order that the data members are declared. The archive command for the base class remains unchanged. Finally, the constructor, defined in the implementation file, must be modified to assign values to the new data members. These variables can now be accessed in the "apply()" function like any other variable defined within the scope of the function.

A single SV is used in this model, although it is not required by the SVK algorithm. This is purely to demonstrate how an SV array can be created and updated. The "stateOld" vector is provided by the upper scale through the VUMAT with the expectation that they will be updated in a "stateNew" vector that is returned to the VUMAT. It is important to ensure that the size of stateNew is consistent with the incoming stateOld. In the SvkOutputFilter, stateNew is copied from stateOld and the single SV is incremented by one. Initialization of SVs is discussed in Section 6.

After a user creates a pair of filters for their model, they need to modify the "getInputFilter()" and "getOutputFilter()" functions in the ModelFactory. First, add "#include" commands for the header files to the new input and output filters. Then create new objects of the input and output filters in "getInputFilter()" and "getOutputFil-

ter()", respectively. This can be achieved by simply replacing instances of "UserDefined" in *ModelFactory.cc* with the name chosen by the user.

The final step is to update the list of files to be compiled. In the *Makefile.am* file located within the **model/** directory, the source files for the new input and output filters must be added to the "libhmsvumat_model_la_SOURCES" list. All header (*.h) files must be added to the "nobase_hmsvumatmodelinclude_HEADERS" list. The files associated with the SVK model can be repeated or replaced with the new file names to simplify this step. Once **model/***Makefile.am* has been updated, the "automake" command must be executed at the top level of the source directory. Depending on the user's version of the GNU Autotools suite, it may be necessary to run aclocal, autoheader, autoconf, automake. This should be the first course of action if the automake command produces any errors. Finally, "make" and "make install" should be called from the build directory as described in Section 5.

8. Debugging

Debugging an HMS simulation can be a challenge due to the distributed and asynchronous nature of HMS. The HMS framework performs error checking for the launching of the broker to aid in debugging issues when launching HMS and connecting the broker to the upper-scale code. However, errors in the input filter, lower-scale executable, and output filter may be difficult to identify. In this section, various tips and tools are discussed to assist developers and users in debugging lower-scale evaluations in HMS simulations.

When developing new input and output filters, it is beneficial to include error handling directly in the filters. The HMS framework includes a number of C++ exceptions that can be thrown in the filters and are caught by the HMS broker. If an "IOError" exception is thrown, the HMS broker will try to rerun the lower-scale evaluation five times before terminating. This is to provide fault tolerance in case the error was due to file system or other non-repeatable issues. All other exception types will cause the HMS simulation to immediately terminate with the associated error message printed in the standard error stream. Users may want to include the evaluation directory path in the error message to easily identify which lower-scale evaluation triggered the error.

In cases where the directory is not included in the error message, or when the error

occurs within the lower-scale executable, it can be difficult to find which lower-scale evaluation caused the error, particularly when there are thousands to millions of separate lower-scale evaluations. To aid in identifying the failed evaluation, the *findStalledJobs.py* Python script is provided.

```
$ <HMS_VUMAT_INSTALL>/share/findStalledJobs.py \
  <outputDir> <filename>
```

For this script to work, there must be a file that the script can look for to indicate a successful evaluation. This filename is passed as the second command line argument. For example, if the lower-scale executable prints *stress.out*, then running *findStalledJobs.py* with this filename will find any evaluation directory where the lower-scale executable did not finish. However, this will not find evaluation directories where the lower-scale executable completed but the subsequent output filter failed. During development, it is recommended that a small file is written at the end of the output filter. This file can then be used by the Python script to easily find directories that did not complete the output filter phase of the lower-scale evaluation. The example below shows the command and output from running findStalledJobs.py on the outputDir (**/tmp/hmsOutputDir**) from an HMS simulation with two brokers. In this model, a file called *PostOutputFilterVals.txt* was created at the end of each output filter evaluation, containing the values returned to the upper-scale code.

```
$ python <HMS_VUMAT_INSTALL>/findStalledHMS.py \
  /tmp/hmsOutputDir/ PostOutputFilterVals.txt

Expected: 688 in /tmp/hmsOutputDir/r4i0n20/37443
    STALLED: /tmp/hmsOutputDir/r4i0n20/37443/683
    STALLED: /tmp/hmsOutputDir/r4i0n20/37443/656
    STALLED: /tmp/hmsOutputDir/r4i0n20/37443/686
    STALLED: /tmp/hmsOutputDir/r4i0n20/37443/685
    STALLED: /tmp/hmsOutputDir/r4i0n20/37443/687
    STALLED: /tmp/hmsOutputDir/r4i0n20/37443/684
  Found 682 / 688
Expected: 732 in /tmp/hmsOutputDir/r4i0n21/8298
    STALLED: /tmp/hmsOutputDir/r4i0n21/8298/727
    STALLED: /tmp/hmsOutputDir/r4i0n21/8298/731
```

```
  STALLED: /tmp/hmsOutputDir/r4i0n21/8298/729
  STALLED: /tmp/hmsOutputDir/r4i0n21/8298/730
  STALLED: /tmp/hmsOutputDir/r4i0n21/8298/698
  STALLED: /tmp/hmsOutputDir/r4i0n21/8298/728
Found 726 / 732
```

In this case, there are two subdirectories in **/tmp/hmsOutputDir** for the two brokers. Each broker had one evaluation that failed, evaluation number 656 on the first HMS broker and 698 on the second. The failed evaluations were rerun five times before terminating, corresponding to the five additional evaluation directories that are numbered sequentially. With the failed evaluation identified, additional work is required to determine the cause of the failure. Note, as the file name suggests, this script can also be used on a running simulation that has stopped making progress. The most likely cause is that a lower-scale execution has stalled; the script will quickly find where that has occurred.

If the failure of a lower-scale evaluation occurs within the lower-scale executable, debugging actions will be specific to the lower-scale code. A good first step is to attempt to manually run the code in the evaluation directory, outside of the HMS framework. When the failure is due to the input or output filter, a user can employ the *testInputFilter* and *testOutputFilter* binaries located in <HMS_VUMAT_INSTALL>/bin. These executables apply the input and output filters outside of the HMS framework. They are recompiled every time the HMS-VUMAT library is recompiled. They can be run through a debugger, or additional code, such as print statements, can be added in the filters to aid in the debugging process. These executables require that the VumatArgument struct was written to a file that can be read back in. Since these filter tests are executed outside of HMS, they do not receive the argument data from the upper-scale. Therefore, the input filter should write the contents of the VumatArgument to a file using the provided "print()" member function during development. This can be removed once the model is stable, or the user can create a flag to turn this on and off for added flexibility (see the "Verbose" flag in the SVK filters as an example). The filter tests can be run with the following command:

```
$ <HMS_VUMAT_INSTALL>/bin/testInputFilter \
  -d <directory-to-apply-filter> \
  -a <argument-file> -c <hmsConf-file>
```

9. Profiling

Two Python scripts are provided to generate basic profiling information of the lower-scale evaluations. These scripts make use of the*Timestamp_I.log* files, where I is the process count of the broker. These timestamp files are written by default in HMS simulations. Simple profiling of all time steps for multiple simulations can be obtained through the *profile_allTimesteps.py* script by the following command:

```
$ <HMS_VUMAT_INSTALL>/share/profile_allTimesteps.py \
  -f <list of all Timestamp_I.log files>
```

while more in-depth profiling of a single time step can be obtained through *profile_singleTimestep.py* using the following command:

```
$ <HMS_VUMAT_INSTALL>/share/profile_singleTimestep.py \
  -f <Timestamp_I.log file> \
  -t <Desired simulation time of time step> \
  -dt <Search window for desired simulation time>
```

Two example problems were run to demonstrate the use and output from the profiling scripts. A mesh consisting of 64 elements was deformed over a simulation time of 2 s. Both simulations employed one HMS broker and one CPU core for the upper scale. In one simulation, the total resource pool was a single CPU node containing 40 cores. One core was devoted to the upper-scale code and another core was devoted to the HMS broker. This left the resource pool for the lower-scale evaluations at 38 cores. A second simulation was run on four CPU nodes, leaving the resource pool for lower-scale evaluations at 158 cores. Each lower-scale execution was parallelized and run on four cores. The *Timestamp_0.log* files for the one node and four node runs were copied to files called *1node_time.log* and *4node_time.log*, respectively.

The output plots from profiling all timesteps are shown in Fig. 8 and were generated with the following command:

```
$ <HMS_VUMAT_INSTALL>/share/profile_allTimesteps.py \
  -f 1node_time.log 4node_time.log
```

In Fig. 8a, the simulation time versus wall-clock time is plotted. From this plot, we see that the simulation took approximately 13 min to complete on one CPU

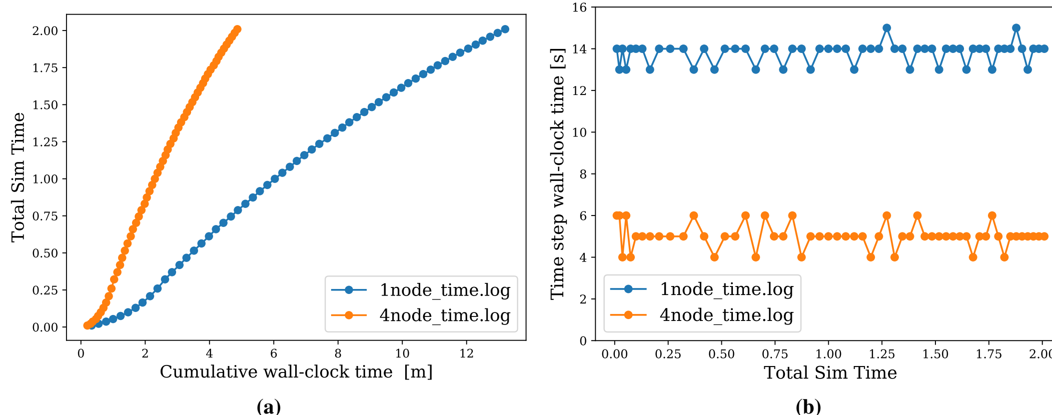

Fig. 8. Output from profiling across all time steps for HMS simulations with resource pools consisting of one and four CPU nodes. (a) Simulation time vs. wall-clock time. (b) Wall-clock time to complete each upper-scale time step.

node and approximately 4 min to complete on four CPU nodes. The plot in Fig. 8b shows the wall-clock time of each upper-scale time step with a resolution of 1 s. Not surprisingly, each time step is quicker in the four node simulation since there are more resources to run the lower-scale evaluations concurrently. If spikes are observed in this plot, it may be indicative of filesystem issues or unique conditions causing the lower-scale executable to take longer than normal. This plot is a good first check if HMS simulations are taking longer than expected.

Profiling plots for a single time step around the simulation time of 1.5 are shown in Fig. 9 and were generated with the following commands:

```
$ <HMS_VUMAT_INSTALL>/share/profile_singleTimestep.py \
  -f 1node_time.log -t 1.5 -dt 0.02
$ <HMS_VUMAT_INSTALL>/share/profile_singleTimestep.py \
  -f 4node_time.log -t 1.5 -dt 0.02
```

The plots on the left of Fig. 9 show when the input filter, lower-scale executable, and output filter are running for each of the 64 evaluation requests within the time step. The plots on the right show the number of concurrent instances of the lower-scale executable that are running at various times throughout the time step. On a single CPU node, nine lower-scale evaluations start almost immediately, using 36 of the 38 available CPU cores. New evaluations start shortly after another one finishes. With four CPU nodes, the switching of tasks within the broker is evident.

Once the input filters for 17 evaluations have been completed, the broker pauses the application of input filters and switches to starting the lower-scale executable for those evaluations. After launching those executables, the broker goes back to finish the rest of the input filters. Once input filters have been applied for all evaluations, the rest of the resources are committed to additional lower-scale evaluations to achieve 39 concurrent evaluations, filling up 156 of the 158 CPU cores in the resource pool. This highlights the asynchronicity of HMS, where HMS is constantly switching between receiving evaluation requests, applying input filters, scheduling and launching lower-scale executables, applying output filters, and sending results back to the upper scale. A close-up of the first 15 evaluations for the one-node simulation is presented in Fig. 10. The black bars show that each lower-scale model takes 1.5 to 2 s. A new lower-scale executable is launched when resources become available, after the completion of another execution. The red and cyan bars show that the input and output filters require a relatively insignificant amount of time, even with file I/O. If simulations are taking longer than expected, particularly if spikes are observed in Fig. 8b, then profiling of a single time step may aid in identifying the bottlenecks.

10. Description of the HMS-VUMAT Files and Classes

The HMS-VUMAT library is written in C++ following the object–oriented programming paradigm. This allows maximum flexibility and extensibility of the library. Objects are communicated between the HMS-VUMAT library and the HMS framework using BOOST's serialization feature.[9] This is the same serialization method used by the HMS framework to pass objects amongst its various modules. Below is a brief description of the files and classes contained in the HMS-VUMAT source code. Directories are indicated by bold text with down arrow bullet markers. Files are indicated by italicized text and solid black circles. If a file extension is not provided, there are separate header (*.h), implementation (*.cc) and template implementation (*_t.cc) files associated with the file name.

- *Vumat.cpp* – A wrapper function that extracts code-specific MPI communication information (MPI rank and size) and evaluates the VumatDriver. This wrapper is either compiled with the rest of the library as a stand-alone library or can be compiled by the parent FEA software at runtime and linked against the driver library, as described in Section 5.

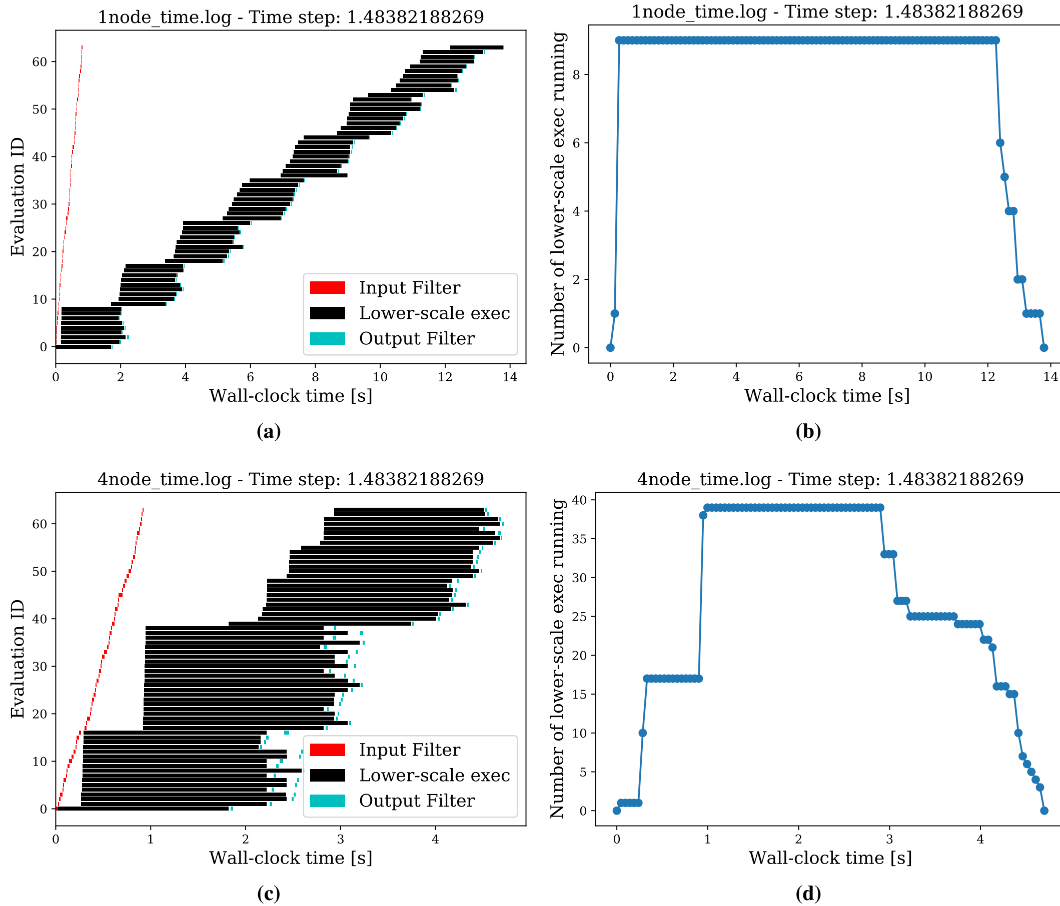

Fig. 9. Output from profiling a single time step within 0.02 s of the target simulation time of 1.5 s. Plots in the top row correspond to a simulation with one CPU node of available resources (38 cores). Bottom row corresponds to a simulation with four CPU nodes of available resources (158 cores). The plots in the left column show when the input filter, lower-scale executable, and output filter are active throughout the upper-scale time step for each of the 64 evaluations. The plots in the right column show how many lower-scale executables are running concurrently throughout the upper-scale time step.

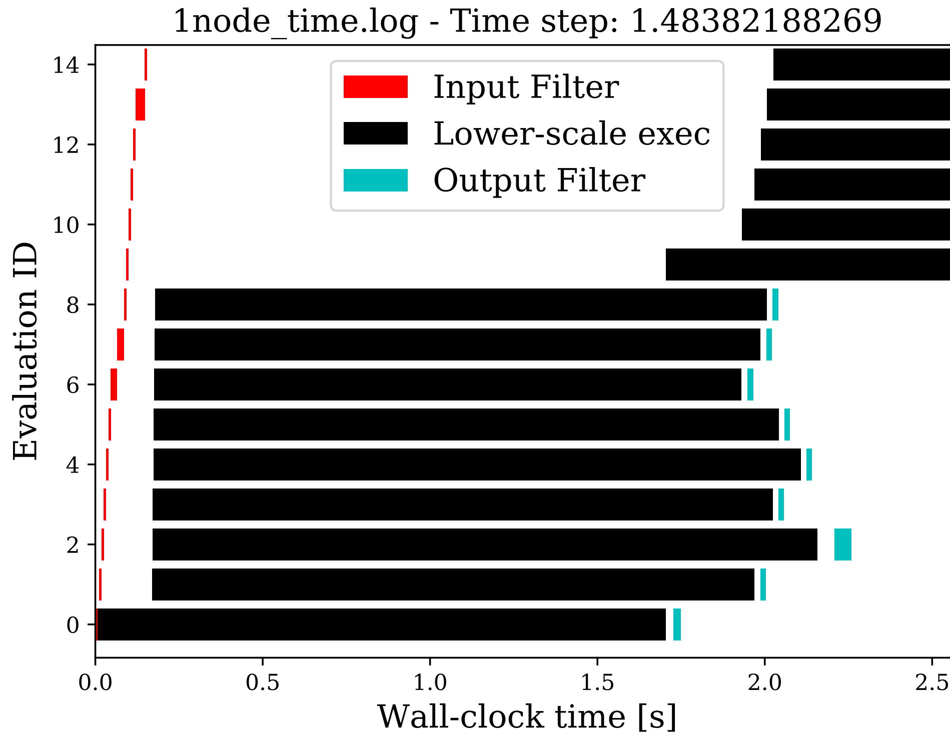

Fig. 10. Close-up view of Fig. 9a showing full evaluation cycles for the first nine evaluations

↓ **base** - Contains classes and functions that should not need to change when introducing new models. The base classes for the VUMAT input and output filters will work "out-of-the-box" if the lower-scale model is modified appropriately. However, in practical applications it is more likely that new derived filter classes will be created to accommodate the input and output file formats of the lower-scale model.

- *VumatArgument* – This class holds the data sent from the VUMAT interface to perform constitutive evaluations at a given MP. This data includes information such as the current simulation time, time step, strain increment, deformation gradient, temperature, and SVs. The full list of data sent via the VUMAT interface can be found in Section 1.2.22 of the Abaqus User Subroutines Reference Guide.[8] Empty and copy constructors are provided, as is a constructor for setting the data members from a file. Accordingly, a print() member function is included such that the argument can be written out to file and reconstructed seamlessly.
- *VumatValue* – This class holds the data to be returned to the VUMAT such as the updated stress tensor, energies, and SVs. Like with the VumatArgument, a

constructor and print() member function are provided to reconstruct the value object through file I/O.

- *VumatInputFilter* – The VumatInputFilter base class and its derived classes are responsible for taking the VumatArgument data and filtering it such that the data is staged as input for the evaluation of the lower-scale model through the "apply()" member function. An HMS configuration file (see details in Sections 6 and 7) can be read by the constructor to provide information, such as file names, directory locations, and parameters to the filter that may be desirable to specify at runtime, instead of hard coded into the filters' source code. There is also an initialize_state() member function that can use the HMS configuration file to initialize state history variables, which are data members of the VumatArgument object.

- *VumatOutputFilter* – The VumatOutputFilter base class and its derived classes are responsible for processing the output of the lower-scale execution and returning the pertinent data to the VumatValue object. The VumatOutputFilter constructor also takes the HMS configuration file if needed. One nuance of the VUMAT interface is that it expects constitutive models to be evaluated in a frame that rotates along with the material (corotational frame), and thus the stress tensor returned to the VUMAT should be rotated accordingly. Since many lower-scale models operate in the global reference frame, a "rotate()" member function, which rotates the stress tensor from the reference to the material frame, is included in the VumatOutputFilter base class for convenience.

- *VumatUtils* – Contains various utility functions for reading to and writing from files, converting between Fortran and C array indexing, and redirecting the output streams.

↓ **model** – All files that need to be modified by a user when creating a new model are isolated to this directory.

- *ModelFactory* – Simple functions used by the VumatDriver to obtain the input and output filter objects specific to the user's model. These functions must be modified when adding a new model. Other information, such as the model executable, command-line arguments, and resource types and numbers, are read from the HMS configuration file via the Model Factory to construct the Model object.

- *Makefile.am* – File containing instructions for GNU Automake. This file must be modified when new filter classes are created by a user (see Section 7 for details).

↓ **userDefined** – Templates for a user-defined input and output filter that can be easily modified by a user to create their own multiscale material model. See Section 4 for additional information.

  - *UserInputFilter* – Template for a user-defined input filter. Comment blocks are located anywhere a user may want to modify code for their specific application. *UserInputFilter.h* should only be modified when adding data members to the class. This should be unnecessary in most applications. An example implementation of "apply()" is provided that is compatible with the *svkModel.cc* executable described in Section 4. This block of code can be removed or ignored with the "#if" preprocessor macro when implementing a new user-defined function.
  - *UserOutputFilter* –Template for a user-defined output filter. Comment blocks are located anywhere a user may want to modify code for their specific application. *UserOutputFilter.h* should only be modified when adding data members to the class. This should be unnecessary in most applications. An example implementation of "apply()" is provided that is compatible with the *svkModel.cc* executable described in Section 4. This block of code can be removed or ignored with the "#if" preprocessor macro when implementing a new user-defined function.

↓ **svk** – A simple example of creating the input/output filters for a lower-scale model. This example uses an isotropic SVK hyperelastic constitutive model as the lower scale. While the model is trivial and would not require the HMS framework in practice, it demonstrates the flexibility the filters provide. The SVK model can also be used as an "out-of-the-box" model for testing the integration of the HMS-VUMAT interface with upper-scale FEA codes. These filters are not an exhaustive demonstration of the full capabilities supported by the HMS-VUMAT. Complex filters with sophisticated branching to determine input conditions and extensive postprocessing of the output files to extract the necessary quantities of interest are possible.

  - *svkModel.cc* – Code for the lower-scale executable. The model requires two input files, one containing the elastic constants and another containing the current deformation gradient. The model writes an output file contain-

ing the Cauchy stress in the reference frame. The default paths for all files are hardcoded in the source code, but can be changed through command-line arguments. In this simple example, the lower-scale executable is compiled within the HMS-VUMAT framework for convenience. In most practical applications, the lower-scale model will be compiled separately.

- *SvkInputFilter* – The input filter must write the two input files expected by the svkModel executable. The filenames have defaults within the input filter that are consistent with the defaults in the svkModel source code. However, the filenames can be changed through parameters specified in the HMS configuration file. The deformation gradient is extracted directly from the VumatArgument. The elastic properties are passed to the filters through the "props" array, which is part of the VUMAT interface and must be specified by the user in the upper-scale input file.
- *SvkOutputFilter* – The output filter reads the stress file written by the svkModel executable and rotates it to the corotational frame. An SV is incremented by 1 to show how SVs can be updated. This is purely for demonstration purposes, as no SVs are required for this simple elastic model.
- *SvkDefines.h* – List of macros for the indexing into property and SV arrays. This file is not required when developing a new model, but improves readability of the input and output filters and makes it easier to add, remove, or reorder the parameters within these arrays.
- *hmsConf_svk.ini* – Example HMS configuration file with example values for the [model] section that correspond to the SVK model.

↓ **driver**

- *VumatDriver* – The driver performs the following functions:
  1. Launches the HMS broker at the beginning of the simulation.
  2. Constructs VumatArgument objects for each MP in the block sent to the VUMAT.
  3. Constructs the necessary input and output filters using the ModelFactory.
  4. Constructs an HMS ModelPackage object by reading the HMS configuration file and packaging the VumatArgument, input filter, and output filter objects for each MP.
  5. Sends the ModelPackage objects to the HMS broker for evaluation.

6. Waits for the ModelPackage objects to return with VumatValue objects.
7. Once returned, stuffs the data contained in the VumatValue object into the variables returned to the upper-scale code by the VUMAT.

↓ **scripts** – various scripts used to run, debug, profile, and postprocess HMS-VUMAT simulations. See Sections 6, 8, and 9 for more information on the use of each script.

- *HmsConf.ini* – Template HMS configuration file used by the HMS framework and HMS-VUMAT. Uses the INI configuration file format.
- *generateHMSConf_noAS.py* – Python script used at runtime to fill in the HmsConf.ini file.
- *findStalledJobs.py* – Python script that searches the HMS output directory for any jobs that failed to complete.
- *gatherHmsResults_MatId.py* – Python script that searches through the HMS output directory and consolidates evaluation files associated with a particular MP ID.
- *profileEvalTimes_allTimesteps.py* – Python script that uses the timestamp log files to track the simulation and wall-clock times.
- *profileEvalTimes_singleTimestep.py* – Python script that uses the timestamp log file to profile all lower-scale evaluations at a single time step.

The HMS-VUMAT library employs the GNU Autotools build system to compile and install all needed libraries, headers, and source code files. Each directory contains a *Makefile.am* file, which is interpreted by the GNU Autotools build system to generate Makefiles for compilation. The *Makefile.am* contained in the **model** directory is the only file highlighted in the above list, as it is the only *Makefile.am* that the user needs to modify.

11. Conclusion

The HMS-VUMAT provides a flexible interface for coupling multiscale material models into robust FEA tools. The interface has been successfully used with lower-scale methods such as FEA, particle-based methods, and FFT-based spectral homogenization methods. HMS-VUMAT was designed to significantly reduce the

programming and computational science expertise required to develop hierarchical multiscale material models, enabling computational mechanicians to rapidly develop high-fidelity models for FEA tools used throughout the engineering community.

Use of the VUMAT interface ensures transferrability of multiscale material models to any FEA tool that supports the VUMAT interface. This library serves as an example of how the capabilites of FEA codes can be drastically improved through support of the VUMAT interface. HMS-VUMAT has already been successfully tested in Abaqus, ALE3D, and EPIC.

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

3. E W, Engquist B. The heterogeneous multiscale methods. Communications in Mathematical Sciences. 2003;1(1):87–132.

4. Hibbitt D, Karlsson B, Sorensen P. Abaqus standard user's manual. 1997.

5. 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.

6. 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.

7. 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.

8. Dassault Systèmes. Abaqus user subroutines reference guide. 2014 [http://130.149.89.49:2080/v6.14/books/sub/default.htm?startat=ch01s02asb20.html#sub-rtn-uexpmat.

9. Ramey R. Boost serialization tutorial. 2004 [https://www.boost.org/doc/libs/1_75_0/libs/serialization/doc/tutorial.html.

List of Symbols, Abbreviations, and Acronyms

TERMS:

DEM – discrete element method

DPD – dissipative particle dynamics

EOS – equation of state

FE – finite element

FEA – finite element analysis

HMS – Hierarchical MultiScale

HPC – high-performance computing

MMM – multiscale material modeling

MP – material point

MPI – Message Passing Interface

SV – state variable

SVK – Saint Venant-Kirchoff

VUMAT – Vectorized User MATerial model

MATHEMATICAL SYMBOLS:

e – energy

F – deformation gradient

σ – stress

δt – time step

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CI TECH LIB

FCDD RLA NA
J CRONE
J KNAP
K LEITER

FCDD RLA CB
R BECKER
A TONGE

FCDD RLA M
E CHIN

FCDD RLA MA
J SANDS
D SCHESSER

FCDD RLA MB
A GAYNOR
G GAZONAS
E HERNANDEZ
B POWERS
Z WILSON

FCDD RLA MG
J LENHART

FCDD RLA TA
M COPPINGER

FCDD RLA TB
J CLAYTON
S SATAPATHY
S WOZNIAK

FCDD RLA TD
R DONEY
S SCHRAML
J STEWART

FCDD RLA TE
J LLOYD
M LOVE
D HORNBAKER
J HOUSKAMP
G VUNNI

FCDD RLA TF
J CAZAMIAS
R LEAVY
C MEYER
J O'GRADY

FCDD RLA TG
D FOX
S KUKUCK

FCDD RLA WA
B BARNES
J LARENTZOS
B RICE

FCDD RLR ET
B LOVE

DEVCOM AC
D CARLUCCI
A HAYNES
D PFAU

DEVCOM GVSC
P JAYAKUMAR
Y RUAN

ERDC
M ADLEY
M CHANDLER
K DANIELSON
R MOSER

AIR FORCE RESEARCH LAB
K VANDEN
E WELLE

NAVAL AIR SYSTEMS COMMAND
B BLAZEK

NAVAL AIR WARFARE CENTER
D ABELN
M GROSS

NAVAL SURFACE WARFARE CENTER
S DWIVEDI

NAVAL UNDERSEA WARFARE CEN-
TER
M HOPSON

IDAHO NATIONAL LAB
B AYDELOTTE
L MUNDAY
A RECUERO
D SCHWEN

LAWRENCE LIVERMORE NAT LAB
M BARHAM
N BARTON
W ELMER
E GLASCOE
E HERBOLD
M HOMEL
R MCCALLEN
C WOJNAR

LOS ALAMOS NAT LAB
K BENNETT
R LEBENSOHN
M LEWIS
E MAS
H MOURAD
D ZHANG

SANDIA NATIONAL LAB
S MILLER
J PLEWS
J SERRANO
T SHELTON
J THOMAS
L TUTTLE
M WONG

JOHNS HOPKINS UNIV
L GRAHAM-BRADY
K RAMESH
M SHIELDS

UNIV OF WISCONSIN
C BRONKHORST

SOUTHWEST RESEARCH INST
S BEISSEL
C GERLACH
T HOLMQUIST