Developer's Guide for Integrating Models into the Interagency Fuels Treatment Decision Support System (IFTDSS)

H. Michael Rauscher
Tami Haste
Stacy Drury
John Stilley
Jennifer DeWinter

Version: 10/31/12

Terminology

When discussing software, terminology can have different meanings and can be interchangeable and vague. For the purpose of this document, we use the following set of common terms and definitions.

Term	Definition
Software Integration Framework	A framework that integrates scientific models, built for different purposes, into a larger structure capable of addressing a broader question or decision support need. Software integration frameworks may or may not have a graphical user interface (GUI) and are typically platform independent. Software integration frameworks require model or module packages.
Software Application	An application that integrates scientific models, built for a specific purpose, to address a specific and somewhat narrowly defined question. Software applications typically have a GUI that is tightly coupled with its underlying software models and are typically platform-specific. Many applications utilize models or dynamic link libraries (.dll).
Model	The source code, or model calculation software that performs a specific mathematical algorithm. Models are command-line driven and have no GUI.
Module	A collection or grouping of mathematical models.
Model or Module Package	A package of software code containing model calculation software, metadata, and some kind of model implementation (e.g., wrapper) to allow one model (model package) or a collection of models (module package) to communicate with a larger software integration framework.
Scientific Modeling Framework (SMF)	The underlying modeling framework used by the IFTDSS application.
SMF-Modeling Binary Interface (SMF-MBI)	The application binary interface that describes the low-level software interface between a model and the SMF. The SMF-MBI defines how a model can communicate with the SMF.

Introduction

In 2011, an interagency Information Technology (IT) working group was assembled and asked by Kim Thorsen, Deputy Assistant Secretary, Department of the Interior (DOI) and Jim Hubbard, Deputy Chief, USDA Forest Service, to study the interagency IT problems and present a report with recommended actions. The following is taken from *Implementing the National Wildland Fire Enterprise Architecture Blueprint*, submitted by Jim Douglas and John Phipps, accepted by Kim Thorsen and Jim Hubbard, dated July 15, 2011:

> *In summary, at present there is no overall governance of wildland fire investments, no agreed upon vision or strategy for making future investments, and limited standards or protocols for data and management. Each agency maintains separate, parallel organizations. The NWCG provides some coordination of user requirements and voluntary standards. The decision space of each organization is limited. A number of applications provide important support to wildland fire planning and operational activities. But significant inefficiencies exist in sharing of data, project management, and application support. There is no consensus view on business requirements and priorities, nor is there an agreed upon strategy or vision to guide new investments or evaluate the efficacy of current investments. Changing the governance, strategies, and organizations necessary to achieve the stated goal of the Blueprint to operate as "a virtual single agency" will require sustained senior level management commitment as well as investing in significant changes in long-standing cultural norms for the agencies and for interagency structures.*

In short, this report confirmed the "software chaos" problem experienced by the DOI agencies and Forest Service.

Following a strategic assessment, in 2009 the JFSP undertook development and testing of an Interagency Fuels Treatment Decision Support System (IFTDSS) focusing on the inter-agency hazardous fuels reduction (HFR) program. The IFTDSS software integration framework provides a demonstrated technical solution to create a standard and effective process for organizing current standalone scientific software tools so that data management inefficiencies are eliminated, and project management and functional application support is enhanced. There are approximately 500 fire and fuels-related databases, reporting systems, and software applications. Two independent surveys of fire and fuels field users reported that of these 500 databases, reporting systems, and software application, approximately 30 of them have a significant user base and provide important support for wildland fire planning and operational activities at the current time. The first goal of the IFTDSS software integration project was to organize and integrate many of the 30 most commonly used data sources and software applications into a sequence of activities that resulted in the completion of mission critical work as defined by field users.

Furthermore, the IFTDSS software integration framework serves as an example and stepping stone toward a larger interconnected "system-of-systems" vision. The vision is that the management community will access modeling, analysis, information and reporting needs through a small number of interconnected software integration frameworks defined and organized by fire and fuels management business needs. For example, the BlueSky Framework (BlueSky) and Wildland Fire Decision Support

System (WFDSS) can also be considered software integration frameworks, each serving a different business need within the fire and fuels domain. Each software integration framework has access to a common virtual library of component computational models and tools. IFTDSS is intended to demonstrate the viability and value of this concept by integrating access to many of the models and tools available to fuels planners through one GUI organized by workflows identified by the users.

For more detailed information about the JFSP Software Tools and Systems (STS) study, readers are urged to review the many documents that have been generated by the STS study published on the website: www.frames.gov/iftdss.

Document Objective

This document is intended to provide information and guidance to members of the fire and fuels sciences and software tool development community who are interested in linking their existing or new software tools for hazardous fuels reduction planning and management into the IFTDSS software integration framework.

What's In It for Me as a Developer?

Most of the hundreds of existing fire and fuels related scientific models and software applications have been developed by scientists and software developers employed by the Forest Service or a DOI agency. Developing and delivering software applications is motivated by the desire to transform original scientific research results into tools that users can deploy in supporting mission critical work. Software application development, deployment, maintenance, and user support operations are expensive and challenging to do well. Resources that field and support a good software application often compete with resources available to perform new research. Many development teams report that the cost of developing a software application and the overhead to support it (user training and the long-term updates and maintenance costs) are greater than simply coding the underlying mathematical algorithms into software models.

The IFTDSS software integration framework concept will reduce the overhead associated with developing and maintaining software applications and offers developers substantial benefits, including:

- **Software Development Time and Cost Savings**
  - Enables scientists to program new mathematical algorithms into software models (in a variety of programming languages) without the need to develop and maintain a software application to drive the models
  - Allows developers to concentrate available resources on improving the scientific integrity and functionality of software models
  - Provides developers with software-software communications standards, which allow developer-friendly packaging of the software models into model packages that can be used by larger software integration frameworks (i.e., IFTDSS)

- Allows developers to retain responsibility for maintenance and error correction internal to their software models and modules (IFTDSS project team is responsible for overall system maintenance, development of the GUI, and error correction)

- **Access to an Established User Community**
  - Provides developers with instant access to a large user community
  - Provides mechanisms to collect user feedback routed through the framework system
  - Automatically sends usage reports to developers periodically

- **Autonomy and Ownership**
  - Enables developers to maintain control and ownership of the software model or module they place into the IFTDSS framework
  - Provides developers as a group continuous access to the IFTDSS project team and enables great freedom to influence how their software tool is being used within the framework
  - Allows developers to retain the freedom to specify unique and non-standard ways that certain users can use their software tool
  - Allows developers retain responsibility for producing and maintaining documentation specific to the internal operations of their software tool (IFTDSS project team is responsible for user training and coaching of the overall system)
  - Allows developers to retain responsibility for communicating current scientific limitations of their software tool and suggesting what is reasonable and not reasonable for field users to do in their fuels treatment analyses
  - Allows developers to retain the freedom to brand their models with images, logos, or other differentiating information

A Brief Introduction to IFTDSS – Business Needs and Workflows

IFTDSS is a web-based, software integration framework that manages pre-existing and newly developed software models and their required data needs to analyze and support decisions about fuels management to mitigate wildfire risk.

In IFTDSS, software models are organized and made available to users in three ways: (1) by IFTDSS workflows, (2) by model developer(s), and (3) by all models available in the system grouped by model type and the outputs each produces.

IFTDSS Workflows

Prior to the development of the IFTDSS framework, extensive field user input resulted in the identification of four workflows that meet the business needs of fuels treatment planners:

1. The **Hazard Analysis Workflow** is used to identify potentially hazardous areas across a landscape. The focus of this workflow is to identify areas across a landscape where fuels

treatment analysis may be warranted based on potential fire hazard. IFTDSS provides tools that support this workflow.

2. The **Risk Assessment Workflow** provides a first-approximation probabilistic risk assessment for fuels treatment planning.
3. The **Fuels Treatment Workflow** (a) simulates fuels treatment placement in areas of high fire hazard within an area of interest, (b) simulates post-treatment influences on fire behavior and fire effects potentials, and (c) evaluates the temporal durability of fuels treatments, that is, how long, in years to decades, a treatment will continue to reduce adverse fire behavior and fire effects within an area of interest.
4. The **Prescribed Burn Planning Workflow** provides the information needed to plan and document a proposed prescribed fire. IFTDSS provides tools that support this workflow; with these tools, users can
    - calculate the probability of ignition from lightning or a firebrand
    - assess and calculate fire behavior
    - assess and plan fire containment
    - calculate fire effects
    - create a prescribed burn plan (including printing out a Word document with many elements filled in by IFTDSS)

These workflows have been implemented in IFTDSS Version 2.0. An IFTDSS workflow is a user defined sequence of hazardous fuels reduction planning activities that result in the completion of a mission critical task. Many standalone, desktop, software applications are available to field users to assist in hazardous fuels reduction planning; however, these applications are isolated from each other. This means that they cannot easily share data among themselves nor is the data in the same format. The scope of these existing software applications is typically smaller than the scope of the mission critical task users need to perform. This requires users to manually "string together" the existing software applications in an ad-hoc sequence. Users stressed the need to automate this sequencing of software applications in such a way that organizes the tools needed to perform a specific task into a logical workflow. Fuels specialists repeatedly said their job focus is to perform mission critical tasks such as prescribed burn planning, hazard analysis, or risk assessment; however, they do not view their job responsibility as becoming expert in any particular software application.

Model Developer(s)

In our interactions with the scientific model developer community, we have discovered that while the IFTDSS workflows (and the organization of models within each workflow) may serve the fuels planning community, it may or may not meet the needs of the model developers themselves, who often have different use cases for their models. To better meet the needs of the developer community, we have created the option to organize tools by model developer(s). For example, IFTDSS currently provides access to the Fuels Characteristics Classification System (FCCS) and Consume models developed by the

Fire and Environmental Research Applications (FERA) Team with the individual models organized based on feedback and direction from the FERA Team. This provides the FERA Team with more direct access to their models, organized in a way that better meets their needs and use cases. The FERA models are also available to other IFTDSS users within the IFTDSS workflows.

All Available Models

Another set of users expressed the need to be able to have direct access to the individual models from a pick-list to quickly get at a particular output set of data without following the sequence of steps that a workflow would entail.

Despite how the individual models are organized within IFTDSS, the advantage to all users is that IFTDSS provides access to many individual software tools, all in one central place, available through a single user interface, with consolidated and coordinated data management processes. Therefore, IFTDSS has been designed to accommodate a variety of user needs.

For further documentation and reports about the IFTDSS project, see the project website at www.frames.gov/iftdss . The most recent version of IFTDSS can be found at iftdss.sonomatech.com. Developers of existing or new software models that are interested in adding their tools to the IFTDSS software integration framework are encouraged to contact Mike Rauscher at mrauscher@bellsouth.net or Stacy Drury at sdrury@sonomatech.com.

A Brief Introduction to the IFTDSS Software Integration Framework

IFTDSS is a service-oriented architecture (SOA) software integration framework for managing and integrating scientific models and data. The IFTDSS software integration framework consists of three layers: (1) the IFTDSS Web Application, (2) the Scientific Modeling Framework (SMF), and (3) the scientific models available within the system (**Figure 1**).

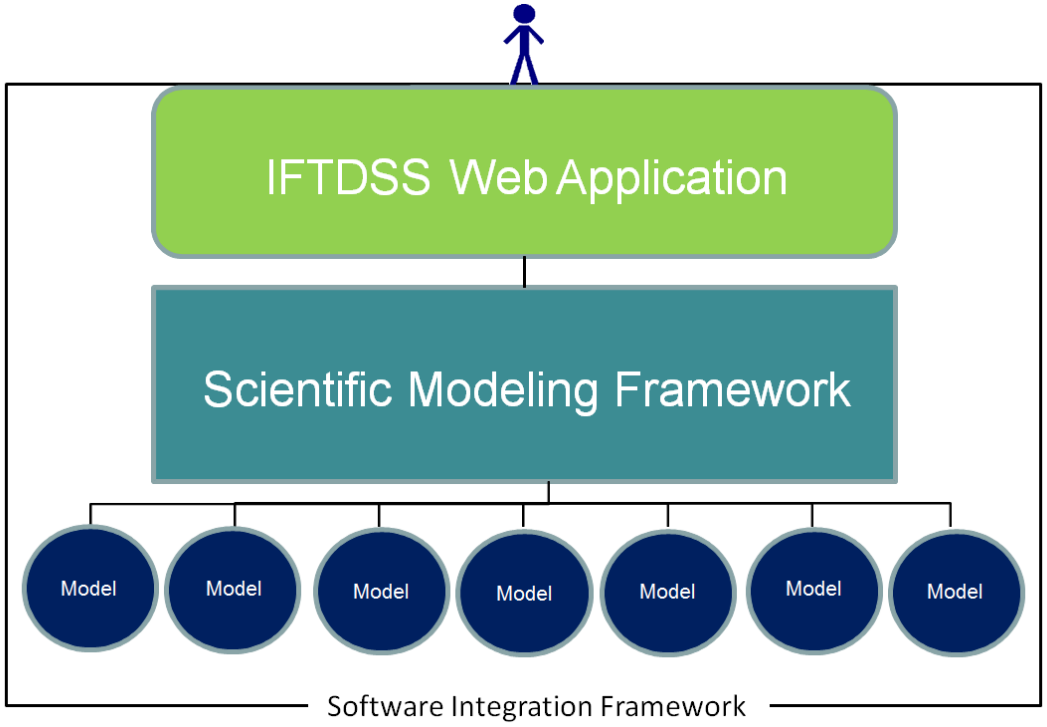

Figure 1. The IFTDSS software integration framework architecture.

The SMF is the backbone of the framework that encapsulates the scientific models and their data. The IFTDSS Web Application allows users to interact with the data and models available within the system and is very loosely coupled with the SMF. The IFTDSS web application takes user input and uses it to drive the SMF services, triggering and monitoring the execution of SMF models, and presents the model's output data to the user in the form of data tables, graphs, and maps.

As models are added to the system, they are bundled as model packages and registered within the SMF. The IFTDSS Web Application exposes the model to the user through the GUI. As new models are added, the corresponding GUI screens that accompany them are generated dynamically, reducing the amount of time required to modify the GUI as models are added or removed.

The core of the SMF is written in Java (based on the Java 6 platform) to be portable across operating systems. The SMF has a comprehensive application programming interface (API) and can natively interact with models written in either Java or Python. The SMF can interact with any other implementation technology through the use of a Java or Python wrapper. For a more detailed discussion of the IFTDSS software integration framework, please refer to **Appendix A**.

Bringing a Model into the IFTDSS Framework

All interaction between the IFTDSS framework and a model is accomplished through the SMF. The SMF is designed to interact with model packages. The SMF model binary interface (SMF-MBI) describes the low-level software interface between a model and the SMF; that is, it defines how a model can

communicate with the SMF. Software code that implements the SMF-MBI is called the SMF Model Implementation (SMF-MI). Generally, a model package contains the compiled model calculation software, the SMF-MI, metadata, and an XML file that describes each model within the model package, the model parameter values and model inputs and outputs.

There are two possible scenarios for how a model can be used by the SMF framework, and thus by IFTDSS. Either:

1. The model calculation software directly implements the SMF-MBI without the use of a wrapper; or
2. A wrapper is developed that implements the SMF-MBI and communicates with the model, either by invoking it as a separate program or by communicating with it via a network service (i.e., web service).

In either case, some piece of software is needed to implement the SMF-MBI. We call this piece of software the SMF-MI. The core of the SMF is written in Java (based on the Java 6 platform) to be portable across operating systems. Currently, SMF can natively interact with model implementations written in either Java or Python, and can interact with any other implementation technology through a Java or Python wrapper.

There are currently two common choices for the SMF-MBI:

1. **For a model coded in Java** – The model package contains a .class file containing bytecode for a Java class that implements the com.sti.smf.core.Model interface. This Java class is the SMF-MI. In the models.xml file, the model is represented as <type>java</type> and the <implementation> tag contains the fully qualified Java class name of the MI. (*Note that this is currently the most commonly used MBI).*
2. **For a model coded in Python** – The model package contains a .py file containing a Python class that derives from the com.sti.smf.core.Model type. This Python class is the MI. In the models.xml file, the model is represented as <type>python</type> and the <implementation> tag contains a reference to the specific class in dotted notation.

In each case, the models.xml file describes the input and output parameters and the spatial "aperture" of the model input and output. The "aperture" defines the size of a model's unit of work. Models may operate on a single coordinate at a time, all of the coordinates in a given data set, or something in between. The SMF-MI implements a fixed programming interface to receive input data and to provide output data. It should be noted that the IFTDSS and the SMF are currently under development and in future there may be additional ways to implement the SMF-MBI as new models are added to the framework.

Here are examples of how the SMF-MI can be applied:

- **Method A** – The SMF-MI may implement the model calculations directly. In this case, the SMF-MI and the model calculation software are the same software.

- **Method B –** the SMF-MI executes a command-line program written in any language. In this case, the command-line program is the model calculation software and the SMF-MI is a wrapper for it. Commonly, the wrapper will write input files for the model calculation software execute the program, and then read in model results from output files created by the model calculation software program. (*Note that this is currently the most commonly used mechanism.*)
- **Method C –** the SMF-MI could contact a remote service across the Internet (e.g., a web service), pass input data to it, and retrieve model results from it. In this example, the remote service is the model calculation software and the SMF-MI is a wrapper for it. In this case, the model package does not need to contain the model; it only needs to be able to communicate with it.

The following guidelines will help streamline the integration of both new and existing models into IFTDSS:

- **Modularize the model and model calculation software**. Separate all of the model calculation code from any input or output data processing steps, and expose as simple an entry-point as possible. For example, if the model calculations can be enclosed in a single function call, a single class, etc., that will make integration with IFTDSS (and other larger software frameworks) much easier.
- **Separate the underlying model calculation software from the GUI**. Implement model calculations to minimize external dependencies (i.e., specific operating systems or external libraries including GUI systems) if possible. Use open source software over proprietary software when possible. Select a platform-independent development language (Java, Python, etc.). When third-party dependencies are necessary, document them thoroughly.
- **Provide a command-line driven model interface.** Provide a way to run all model calculations in batch mode from the command-line and ensure that the model produces results comparable to the interactive mode (i.e., GUI-driven application). Building a model with a command-line interface is useful for testing purposes.
- **Document model interactions.** Thoroughly document any interactions a model will have with other instances of itself running concurrently. If multi-threading or multiple processes are utilized, try to ensure that the code works safely if and when other copies of the model are running at the same time.

Specific Guidelines for Bringing a New Model into IFTDSS

If you intend to develop a new model for integration with IFTDSS, you must first decide whether the model calculation software will natively interact with SMF, or whether there will be a wrapper to interface between the model calculation software and SMF.

If you are comfortable developing in either Java or Python, then simply develop the model calculation software using either of these programming languages and a wrapper will not be needed. If you already have existing libraries of code that you would like to leverage, or if you prefer to code in a programming language other than Java or Python, then the model calculation software can be coded in a different

language (e.g., C or Fortran) and a wrapper can be developed using Java or Python. Please refer to Appendix B for more detailed instructions and pseudo code for bringing a new model into IFTDSS.

To create a new model and corresponding IFTDSS model package, follow these steps:

1. Develop the model calculation software in Java, Python, or a different programming language.
2. Document the input and output parameters and model properties.
3. Create the models.xml file and populate it with information about the model and its parameters (i.e., input and output variables). Refer to **Appendix B** for an example of an .xml file.
4. Create the model implementation (SMF-MI) and test that it works as intended. If the model calculation software is developed in Java or Python, the SMF-MI will be part of the model calculation software code. If the model calculation software is developed in a different language, the SMF-MI will be a wrapper coded in Java or Python.
5. Assemble the model calculation software, the models.xml file, the SMF-MI, and any supporting documentation, as described in **Appendix C**, into a model package.
6. Test the model package with the Model Test Harness[1] to ensure it works as intended.

Specific Guidelines for Bringing an Existing Model into IFTDSS

For discussion purposes, we have used the First Order Fire Effects Model (FOFEM) as an example. Please refer to Appendix A for more detailed instructions and pseudo code for bringing an existing model into IFTDSS. Follow the steps below to convert an existing model into an IFTDSS model package:

1. **Create a command-line driven, batch mode version of the model** - To bring an existing model into IFTDSS, first create a version of the model that can be run in batch mode through a command-line interface. For example, the FOFEM batch mode model is command-line driven and accepts a series of input parameters in a text input file and produces output values in a different text file. The model calculation software is the existing command-line program. If the model only exists as an interactive desktop application with a GUI, then the first step is to separate and extract the model calculation software from the application or build a batch-mode edition of the model. (*Note that in some cases this may be a non-trivial undertaking, but it is a necessary prerequisite.*)
2. **Develop the SMF-MI** - When the command-line driven model calculation software is ready, you will need to develop a new piece of software to implement the SMF-MI. This new piece of software will be the wrapper for the model calculation software.
    a. The first step in developing a SMF-MI is to **determine the number of models that are contained in the model calculation software**. For example, the FOFEM batch mode

[1] The IFTDSS development team is currently working on developing the Model Test Harness, which will be part of a Model Developer's Kit in the future.

program contains multiple models: a model of consumption and emissions, a soil simulation model, a tree mortality model, and a post-fire injury model. Each of these represents a different model from the standpoint of SMF, even though they share implementation details.

b. The next step is to **identify the parameters that the model accepts as input and produces as output**. Model inputs can be divided into two categories: model inputs and model properties. Model inputs are required values that a model needs to produce output data. Model properties are model configuration settings that a user typically selects when setting up model run (e.g., which equations a model will use, how many times a model will be run, etc.). The set of values that the model accepts as input should be identified and documented. IFTDSS currently supports three types of model input parameters: integers, floating point values, and enumerated values. Each model parameter has information associated with it: a name which is used internally by the model calculation software; a display name which is displayed to the user through the IFTDSS GUI; a type which can be "int," "float," or "enum," and a unit; which is the default unit of measure for a parameter. Additionally, a parameter may also have a description that IFTDSS can display as help when a user scrolls over the parameter through the IFTDSS GUI. Parameters may also have a default value, validation rules to determine valid or invalid values (e.g., limiting the range of legal values to a specific minimum and maximum value). Enumerated parameters must be presented as a list of values that can be selected by the user through the IFTDSS GUI through a drop-down selection tool. **The model input parameters, input parameter metadata, and model properties are described in an XML file called models.xml**.

c. The next step is to **create the SMF-MI** using a class in the Java programming language. This is the approach that was used for the FOFEM model. The class must be compiled against the smf-core.jar library and implement the com.sti.smf.core.Model interface. For convenience, this class may also inherit from the com.sti.smf.support.AbstractModel class, which will reduce the amount or simplify the code that needs to be written. The most important part of the SMF-MI is the execute method. This method takes two objects as parameters: a ReadableClump object, which represents the current values of all input parameters, and a WritableClump object, which stores the values of the model output parameters. The execute method is responsible for performing the actual model calculations. In the FOFEM example, the execute method writes values from the ReadableClump to a FOFEM input file, after which it executes the FOFEM batch mode command-line program, and then reads in the results from the FOFEM output file and records those results in the WritableClump.

d. **Compile the SMF-MI** using the Java compiler (javac).

3. **Assemble the IFTDSS model package,** which will contain:

   a. The command-line executable (in this case, the fof-bat.exe program) version of the model calculation software.

b. The models.xml file.

c. The SMF-MI (in this example, a Java class) for each model described in the models.xml file.

d. Any other data files that are needed (e.g. additional Java classes, or other data files used by the command line program, etc.).

e. Metadata such as model information, model developer contact information, user resources and documentation. The required metadata is described in **Appendix C**.

The resulting IFTDSS model package file should be saved as a .zip file and named using the following nomenclature: "modelname-model.zip." In the FOFEM example, the model package filename is "fofem-model.zip."

To facilitate the process of compiling the Java source code for the wrapper, running tests, and assembling the model package, you can use a build tool such as the Maven build tool for Java, which integrates well with a large ecosystem of other Java tools. Alternatively, a simple Makefile may also work. Once the package is created, you can test it using a utility called the SMF Model Test Harness.

The IFTDSS Model Developer's Kit

In the future a Model Developer's Kit will be prepared that contains more detailed documentation on the model package formats, the SMF API, and a description of the steps necessary to develop and implement models within IFTDSS. The Model Developer's Kit will include:

- **Instructions for developing a model package for integration into IFTDSS** – detailed instructions will be provided describing the different ways in which a model can be integrated within IFTDSS.
- **Model package examples** – examples will be provided showing different options for developing model packages. Model package examples will include:
  - Model calculation software example
  - XML file example
  - SMF-MI example
  - A model creation template – a template will be provided to assist the developers of new models to program them in a format that can be easily integrated with the IFTDSS framework
  - Model metadata example
- **Model test harness** – a command-line driven model test harness will be provided to allow developers to test their model packages prior to integrating them with IFTDSS.
- **User help instructions** – instructions will be provided explaining how to provide user help documentation for integration with the online help system.

Current Models and Data

IFTDSS Version 2.0 currently provides access to models that support these four workflows:

- Hazard Analysis
- Risk Assessments
- Fuels Treatment
- Prescribed Burn Planning

Table 1 lists the models that are currently implemented within the framework and the workflow(s) that each model supports.

Table 1. List of models/modules that are currently implemented in the IFTDSS framework, the workflows they are implemented in, and the method used for the SMF-MI. Method A is direct implementation, method B is through a model wrapper, and method C is a web service call to a remote system.

Models	Model Implementation Method	Workflows			
		Hazard Analysis for Current Conditions	Risk Assessment for Current Conditions	Fuels Treatment Planning	Prescribed Burn Planning
FlamMap Fire Behavior Model	B	X	X	X	X
FBSDK Modules as Implemented in BehavePlus (SURFACE, SIZE, CROWN, SAFETY, SCORCH, IGNITE, CONTAIN, SPOT)	A or B depending on model				X
First Order Fire Effects Model (FOFEM)	B				X
Consume	B	X		X	X
Fuel Characteristic Classification System (FCCS)	A	X		X	X
RANDIG-FlamMap	B	X	X	X	
Fire Emission Production Simulator (FEPS)]	B				X
FlamMap-Minimum Travel Time (FlamMap-MTT)]	B	X		X	X
Open Forest Vegetation Simulator (OpenFVS)	B	X	X	X	X
Stand Visualization System (SVS)	B				X
FireFamily+	A or B depending on model				X
Wind Ninja	B	X	X	X	X
OptFuels	B			X	
VSMOKE via the BlueSky Framework	C				X

The following data are supported in the current version of IFTDSS:

1) LANDFIRE based raster data
    - Landscape (LCP[2]) data
        - Fuel model
        - Canopy Cover
        - Canopy Bulk Density
        - Canopy base height
        - Canopy height
        - Aspect
        - Slope
        - Elevation
    - FCCS[3] standard fuel bed layer
2) FCCS point fuel bed data that are editable and customizable
3) Tree-list data
    - National Tree-List Layer (NTLL)
    - Spatial Data Analyzer output
    - Forest Vegetation Simulator (FVS) output
    - User created tree-lists (stand data)
4) Digital Photo Series (DPS) fuel loading data

[2] The Landscape (.LCP) file is a multi-band raster format used by wildland fire behavior and fire effect simulation models such as FARSITE and FlamMap.

[3] Fuel Characteristic Classification System.

Appendix A

The IFTDSS Software Integration Framework

A1 The IFTDSS Software Integration Framework

The Interagency Fuels Treatment Decision Support System (IFTDSS) is a software integration framework that provides organization of scientific data and models that support the inter-agency hazardous fuels reduction (HFR) program.

A2 What Is the Scientific Modeling Framework?

The IFTDSS software integration framework uses the Scientific Modeling Framework (SMF), a service-oriented software architecture (SOA), for managing and integrating scientific models and data.

The SMF is *service-oriented* in that it is composed of separate units, or services, that are loosely coupled and communicate with each other over nework protocols (**Figure A-1**). The SMF consists of:

- a core software design and library;
- a set of scalable service implementations for managing metadata, data, and models; and
- a suite of tools and support libraries for application and model development.

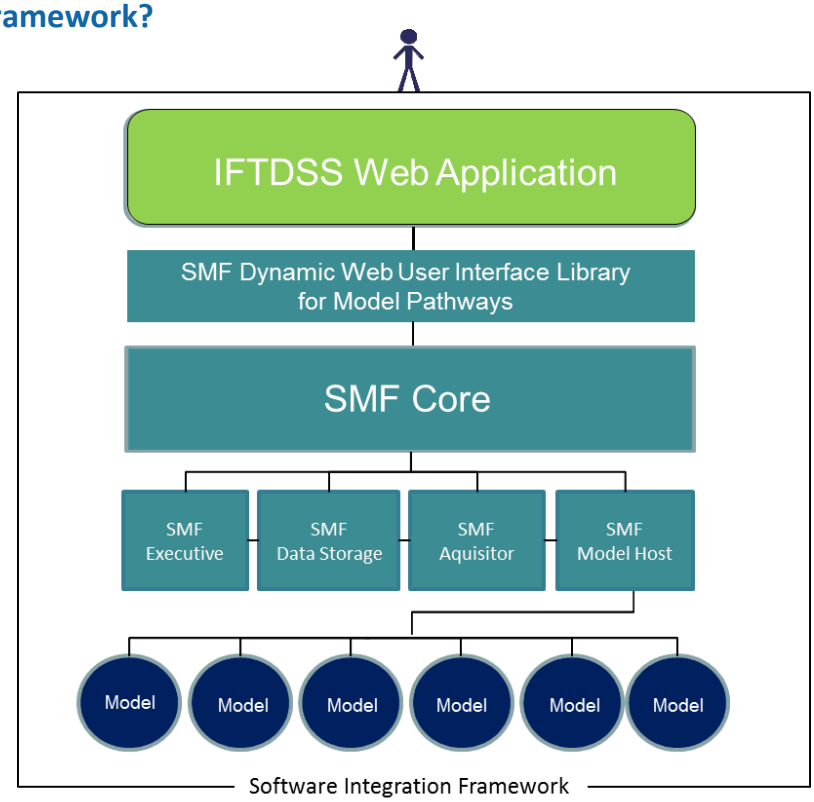

Figure A-1. Diagram of the Scientific Modeling Framework.

SMF software services encapsulate scientific models and their data, compartmentalizing them from each other and from end-user applications such as IFTDSS. SMF services and models operate through programmatic interfaces. The graphical user interface (GUI) is left entirely to the application and only very loosely coupled with the SMF, allowing all user interaction with models to be mediated by the application through the mechanisms of the SMF. For example, the IFTDSS web application takes user input and uses it to drive SMF services, triggering and monitoring the execution of SMF models, and eventually presenting the model output data to the user in the form of data tables, graphs, and maps.

The SMF has four major service components:

- The SMF ***Executive*** is a registry for locating SMF service hosts and models. The Executive also manages other system-wide data about model parameters.
- The SMF ***Data Storage*** server manages and stores multidimensional scientific data.
- The SMF ***Acquisitor*** imports data from external sources, such as uploaded files or the national LANDFIRE database.
- The SMF system also includes one or more ***Model Hosts***, which manage the execution of models.

A2.1 Models and Model Packages

In the SMF, a single ***model*** is defined as the source code, or model calculation software, that performs a specific mathematical algorithm and accepts a well-defined set of input parameters and produces a well-defined set of output parameters. Each model also defines an "aperture," or size of its unit of work. Models may operate on a single coordinate at a time, all the coordinates in the given data set, or something in between. The SMF divides up the input data as needed and performs the model calculations on each "clump" of data, running model executions in a loop or in multiple threads as needed.

In the SMF, a ***model*** is defined as a **calculator** that accepts **input parameters** and produces **output parameters**.

What do we mean by input and output parameters? Put simply, each ***parameter*** defines a single element or layer of data in the SMF. Example parameters include wind speed, elevation, canopy height, and flame length. Variations of similar values—for example, "Flame Length," "Flame Length at Head," and "Flame Length at Back"—are treated as distinct parameters, although the SMF can convert between closely related parameters in some cases. Parameters are not fixed to a specific unit of measure; instead, whenever models and applications interact with data, the units of measure must be specified and the SMF automatically converts between units.

Data values for input or output parameters are stored as ***data sets***, which are managed by the SMF ***Data Storage*** server. The Data Storage server provides a database optimized for storing scientific data, including spatial information.

The SMF system can have one or more ***Model Host*** servers, each of which provides an execution environment for the model calculations. Model Host servers are services within the SOA framework of the SMF, which communicate with other SMF components across a network. This way, Model Host servers may be located in the same machine as the rest of the application, or distributed across the network at other locations.

The SMF organizes models into ***model packages***. A model package is a specially-formatted ZIP file containing everything necessary to run the model within the SMF, plus a special XML file describing the contents of the package. The XML file lists the input and output parameters that each model uses, as well as instructions for how the Model Host should communicate with the model calculation software. Currently, the SMF can only communicate natively with model calculation software written in Java, JavaScript, or Python, but it also supports the use of a *wrapper* written in one of those languages that can interface with any other program, such as a command-line batch version of a model. In the future, other languages and execution environments may be supported as well.model package

A2.2 Pathways and User Interface

So far, we have discussed how the SMF interacts with *models*, that is, with the actual model calculation routines. But how do users interact with SMF models? Ultimately, the user experience is controlled by the *application* using the SMF. In the case of the IFTDSS application, we use the SMF ***Web UI library***, an optional component of the SMF that provides a set of SMF-aware user interface components for use in web applications (**Figure A-2**). Web UI's interface components and user-triggerable actions interact with SMF elements such as data sets and models, while leaving the application with total control over layout, data access, and model execution.

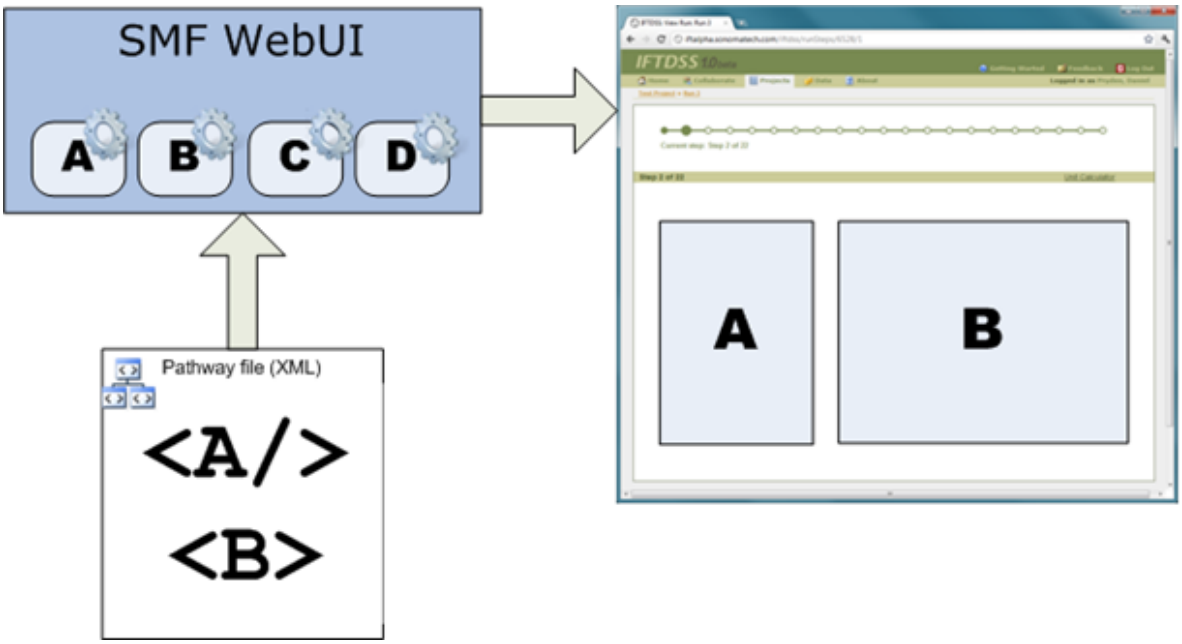

Figure A-2. The SMF Web UI library builds user interface screens for each step by reading pathway XML files.

Web UI components and actions are organized into scripted workflows called ***pathways***, written in XML. A normal pathway might help the user prepare input data, run a single model, and view output data. Complex pathways might execute multiple models in sequence and crosswalk data between them.

Simple pathways can even be used for data preparation or analysis, without running any model calculations at all. Pathways can also be used to prepare documents.

Pathways comprise one or more ***steps***. Each step consists of one or more *components* that define the user interface appearing on a page, as well as any *actions* that occur when the user completes that page (normally by clicking the "Next" button). In most pathways, the user must complete the steps in strict order; in others, such as document preparation pathways, the user can navigate freely to any step.

A2.3 The Purpose of the SMF Design

The design of the IFTDSS application is focused on meeting the needs of the ***end users*** – the people who want to run the models and use the resulting data. The SMF was designed to support the same goals as IFTDSS, but its design focuses on the needs of two other groups of users: model authors and pathway authors.

By ***model author***, we mean the scientist or scientists developing the actual model calculations, as well as their colleagues. We have designed the SMF model package format to be easy to work with, providing a standardized way of packaging all the files necessary to run the model. We have also developed a prototype ***model test harness***, a desktop program for opening and running modules outside of the IFTDSS application. This means that model authors can focus on developing the science behind the models, and can quickly test model calculations in a research environment. Long term, our goal is to provide a suite of tools that enable model authors to develop model packages with less effort than it currently takes to develop a new software application.

A ***pathway author*** may or may not be the same person as a model author. Pathway authors should be people with knowledge of how various models can be used together, so they can design an appropriate user experience for the pathway as a whole. The SMF allows pathway authors to create a small XML file, combining existing user interface components to construct the desired user experience. In the future, we would like to develop additional tools to simplify the process of developing new pathways.

Ultimately, the goal of the SMF is to make the process of maintaining the IFTDSS application as easy as possible. By isolating the model-specific functionality to pluggable model packages and pathway files, the core of the application does not have any dependency on any specific model software. This means that we can easily deploy new models into a running IFTDSS installation, without any modifications to the IFTDSS application or SMF code.

A2.4 What's Next?

The SMF is designed for reuse in other similar projects so other tools can be given the capability to use SMF model packages or other SMF functionality. Likewise, the SMF model package system is designed for extensibility, to allow models to be written in a variety of programming languages and for a variety of computing platforms. This also allows the SMF to coexist peacefully with other model development efforts, providing tools to connect different kinds of components together.

Rather than attempting to replace existing modeling software, the SMF is designed to become a general-purpose “glue” to stick existing software components together. And, by separating the parts of the system that perform calculations from the parts that handle user interactions, the SMF also encourages a new way of thinking about model development, one that will make possible many more “mash-up” applications of different models and data.

The SMF is the platform on which IFTDSS is being built. Our hope is that the SMF platform will prove to be a new foundation for the next level of modeling software.

Appendix B

Bringing a New or Existing Model into IFTDSS

Introduction

This appendix provides stepwise instructions for implementing a new or existing scientific model into IFTDSS. The instructions are intended to provide an overview of the software components that are required to develop a model for the Scientific Modeling Framework (SMF), but they are not comprehensive and do not cover all the ways in which a model could be developed for IFTDSS. The Appendix first describes a set of conceptual steps to be considered prior to development. The Appendix then describes six steps for developing a model for IFTDSS. The description of each step is followed by an example, where applicable, from the IFT-RANDIG model that has already been added into IFTDSS. In IFTDSS, the burn probability module (IFT-RANDIG) is a spatial fire behavior tool in which an entire landscape is analyzed using a single set of wind and moisture conditions and user-defined flame length classes. The module creates raster maps of overall burn probabilities across a landscape and burn probabilities at defined flame length classes.

The following steps should be considered before development to help determine the best way to incorporate a model into IFTDSS.

1) **Identify the intended workflow for the model within IFTDSS.** Determine the needs of the intended users and how the scientific model will be used by fuels treatment planners or the model developer community. The intended workflow may or may not include other models already available in IFTDSS.
2) **Identify whether the model will be added to an existing model package, or set of models, or whether a new model package will be created.** Within IFTDSS, model packages can be specific to the agency responsible for the models in the package. For example, IFTDSS includes a Fire and Environmental Research and Applications (FERA) model package containing models developed by FERA such as the Consume fuel consumption model.
3) **Identify whether the model will be developed natively for SMF, or implemented in SMF via a wrapper class or web service call.** There are two ways to develop a model for SMF: (1) natively (i.e., the model class that implements the SMF-Modeling Binary Interface [SMF-MBI] performs the scientific model calculations directly, without a wrapper), or (2) by developing a wrapper class that implements the SMF-MBI and calls the model calculation application either by invoking an executable or via a web service. This decision is based upon the needs and capabilities of the model developer, as well as the future development plans for the scientific model.
4) **Identify the inputs and outputs required for the model.** Determine what parameters and values must be supplied by the user for the model to run. Determine what outputs are relevant to provide and display to the user after the model has run.

Development Steps to Bring a Model into IFTDSS

The following six steps describe how to develop a new or existing model for implementation into IFTDSS by building each component that is required for an IFTDSS model package. Generally, a model package contains the documentation about the model; the compiled model calculation software; the SMF-MI (a Java or Python wrapper class); auxiliary data (e.g., lookup tables, .csv files); and an XML file that describes each model within the model package, the model parameter values, and model inputs and outputs. The model package can be built by using the NetBeans Integrated Development Environment (IDE). NetBeans is a freely available, open-source IDE that can be used for application development in Java, PHP, and C++, and has been used to develop the IFTDSS modeling framework and incorporate scientific models into the modeling framework.

For each development step, an example is provided or described from the IFT-RANDIG fire behavior model. IFT-RANDIG is an example of implementing a scientific model into IFTDSS via a wrapper class rather than natively. This is the most common method for bringing a model into IFTDSS. The wrapper class, RandigModel.java, is written in Java and implements the SMF-MBI via a Java class called RandigModel which implements the com.sti.smf.core.Model interface. This Java class is the SMF-MI for the Randig model. The RandigModel class obtains input parameters from the input "clump" object,[4] executes the IFT-RANDIG model calculation software via calls to two software executables (*.exe files), and sets the modeled burn probability values to the output "clump" object for use by other models and for display in the IFTDSS web application.

Other models and model packages that have already been incorporated into IFTDSS can also be useful examples for model developers. The Consume scientific model is an example of a model package containing multiple models written in Python that are implemented via several Java wrapper classes. The Digital Photo Series (DPS) models in IFTDSS, which determine fuel loadings for the fuels in a user-selected photograph based on a lookup table, are an example of an algorithm that was developed natively for SMF.

1) Set up the model package

The first development step is to create the Java application for the model package and the Java source package for the new scientific model. The Java source package contains the SMF-MI wrapper class, any associated classes that are required, and the model XML file. There may be multiple Java source packages if more than one scientific model will be included in the model package; however, each source package typically utilizes a unique namespace. Each model package is typically contained within a single folder directory that includes at least two subfolders: (1) a folder containing the model calculation software that performs the scientific modeling, and (2) a folder containing the software to connect the

[4] Data is delivered to and from the SMF-MI in simple pieces called "clumps," which contain values for one or more parameters across a set of one or more coordinates and sometimes also contain structured data stored within SMF features. Coordinates might represent multiple simulations (using the SMF "N dimension") or spatial coordinates across the X, Y, and Z dimensions.

model to SMF. As shown in **Figure B-1**, the model package "My Scientific Models" contains a folder labeled "My Model" with the executable "RunMyModel.exe" (model calculation software), as well as a folder labeled "src" which contains the wrapper class "MyModel.java" and the XML file "models.xml" to connect the model to SMF. The "src" folder could also contain additional classes as required by the wrapper class (i.e., "MyModel.java"). Further, the "MyModel" folder could also contain any libraries or auxiliary files required by the model calculation software (i.e., "RunMyModel.exe").

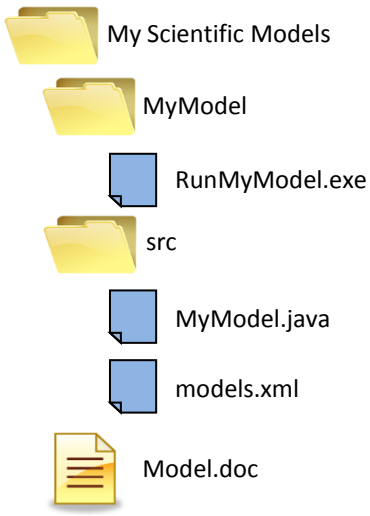

Figure B-1. Example of folder structure for the model package.

The model package must also contain the documentation required for each model in the model package that can be placed in the main folder directory (i.e., the "Model.doc" file in the "My Scientific Models" directory).

2) Develop the model calculation software

The scientific model calculation software contains the algorithms and/or calculations that the model performs. The scientific model must be able to run in batch mode via a command-line interface. If the model exists only as an interactive desktop application with a GUI, then the first step is to separate and extract the model calculation software from the GUI application or build a batch-mode edition of the model calculation software. For example, IFT-RANDIG is a command-line implementation of the FlamMap model that supports batch-mode runs of the minimum travel time module (MTT) with a different random ignition each time. The command-line implementation of IFT-RANDIG within IFTDSS utilizes two software executable files, randig3d.exe and flammapx21.exe, to perform the fire behavior modeling. Each executable requires multiple input arguments; the arguments include the required input files (landscape file, fuel moisture file, weather scenario file, etc.) and input parameters (number of fire ignitions to simulate, etc.), as well as the output file name.

3) Develop the model wrapper class

As described above, all models in IFTDSS require a class, the SMF-MI, which implements the SMF-MBI to enable the model to communicate with SMF. The class must be compiled against the smf-core.jar library and implement the com.sti.smf.core.Model interface. The most important part of the SMF-MI is the execute method, which is responsible for performing the actual model calculations. This method takes two objects as parameters: a ReadableClump object, which represents the current values of all input parameters, and a WritableClump object, which stores the values of the model output parameters. If the model calculation software is to be wrapped, then the SMF-MI will also serve as a wrapper to call the model calculation software. Currently, SMF can interact with wrapper classes written in Java or Python. By convention, the wrapper class file name includes the keyword "Model". For example, the IFT-RANDIG wrapper class is labeled "RandigModel.java".

An example from the IFT-RANDIG model is provided below. The Java class RandigModel is the SMF-MI that implements the SMF-MBI (the com.sti.smf.core.Model interface). Because of the length of the class, only a portion of the RandigModel class is provided below. However, the portion provided includes sections of the "execute" function, which is required by the SMF-MBI.

```
package com.sti.iftdss.models.randig;

import com.sti.justice.StringUtil;
import com.sti.forest.lcp.Landscape;
import com.sti.forest.lcp.MoisturePoint;
import com.sti.smf.core.*;
import com.sti.smf.support.AbstractModel;
import com.sti.forest.lcp.LandscapeFile;
import java.io.*;
import org.apache.commons.io.FileUtils;

public class RandigModel extends AbstractModel
{
   private static final String LCP_FILE_NAME = "Landscape.lcp";
   private static final String FUEL_MOISTURE_FILE_NAME = "Moisture.fms";
   private static final String CUSTOM_FUEL_MODEL_FILE_NAME = "Model.fmd";
   private static final String WEATHER_SCENARIO_FILE_NAME = "Scenario.scn";
   private static final String FLAME_LENGTH_PROBABILITY_FILE_NAME = "out_FLP.txt";
   private static final String OUTPUT_FILE_NAME_PREFIX = "out";
   private static final String IGNITION_GRID_PROBABILITY_FILE = "x";
   private final LandscapeFile lcpHelper;
   private static final int SCENARIOS_PER_FIRE = 1;
   private static final int UNITS = 0;

   public RandigModel() {
      super();
      lcpHelper = new LandscapeFile(getWorkingDir(), "randig.");
   }

   public void execute(ReadableClump input, WritableClump output) throws ModelException {
      int cores = Runtime.getRuntime().availableProcessors();

      Randig randig = new Randig.Builder(this.getProperty("crownFireMethod"),
            this.getPropertyAsDouble("mediumMinFeet"),
            this.getPropertyAsDouble("highMinFeet"),
```

Identifies the model package and namespace for registration with SMF.

The RandigModel class implements the SMF-MBI by extending the AbstractModel class. The AbstractModel class implements the com.sti.smf.core.Model interface which requires that an "execute" method be implemented.

The "execute" function is responsible for running the model calculation software. The "execute" function parameters include the input and output clumps.

```
            this.getPropertyAsDouble("veryHighMinFeet"),
            input.getValueAt("randig.numFireIgnitions", RelativeCoordinate.CENTER, "count"),
            (int) input.getValueAt("randig.wind.speed", RelativeCoordinate.CENTER, "mi/h"),
            (int) input.getValueAt("randig.wind.direction", RelativeCoordinate.CENTER, "deg"),
            (int) input.getValueAt("randig.duration", RelativeCoordinate.CENTER, "min"),
            input.getValueAt("randig.probability", RelativeCoordinate.CENTER, "count"))
            .build();

        File workingDir = getWorkingDir();
        WeatherScenario scenario = new WeatherScenario(randig.getWindSpeed(), randig.getWindDirection(),
randig.getDuration(), randig.getProbability());
        MoisturePoint mp = lcpHelper.readMoisturePoint(input);

        Landscape landscape = lcpHelper.createLandscape(input);

        double resolution = landscape.getResX();

        try {
            createScenarioFile(WEATHER_SCENARIO_FILE_NAME, scenario);
            createMoistureFile(FUEL_MOISTURE_FILE_NAME, mp);
            createCustomFuelModelFile(CUSTOM_FUEL_MODEL_FILE_NAME);
            lcpHelper.createLandscapeFile(LCP_FILE_NAME, landscape, "Created by SMF Randig wrapper");

            // randig and flammap executables need to reside  in the same folder as the input files

            final File randigExe = new File(getPackageDir(), "randig3d.exe");
            final File flammapExe = new File(getPackageDir(), "flammapx21.exe");
            final File randigExeCopy = new File(workingDir, "randig3d.exe");
            final File flammapExeCopy = new File(workingDir, "flammapx21.exe");
            FileUtils.copyFile(randigExe, randigExeCopy);
            FileUtils.copyFile(flammapExe, flammapExeCopy);

            if (!randigExe.canExecute()) {
                throw new ModelException("Unable to execute randig3d.exe");
            }
```

The IFT-RANDIG model wrapper utilizes the "getValueAt" method to determine input parameter values, such as the number of fire ignitions, that are accesible in the input clump. Parameter values in the input clump are configured by the user from the IFTDSS web application, and from the model properties in the models.xml file.

The model calculation software, including randig3d.exe and flammapx21.exe, is specified relative to the package directory.

```
        log.info("Randig: begin executing model");
        execute(randigExeCopy.getAbsolutePath(),
              new File(workingDir, LCP_FILE_NAME).getAbsolutePath(),
              new File(workingDir, FUEL_MOISTURE_FILE_NAME).getAbsolutePath(),
              new File(workingDir, CUSTOM_FUEL_MODEL_FILE_NAME).getAbsolutePath(),
              new File(workingDir, WEATHER_SCENARIO_FILE_NAME).getAbsolutePath(),
              String.valueOf(SCENARIOS_PER_FIRE),
              String.valueOf(resolution),
              String.valueOf(randig.getNumIgnitions()),
              String.valueOf(cores),
              new File(workingDir, OUTPUT_FILE_NAME_PREFIX).getAbsolutePath(),
              IGNITION_GRID_PROBABILITY_FILE,
              String.valueOf(UNITS),
              randig.getCrownFireMethod());
        log.info("Randig: finished executing model");
```

Execute the randig3d.exe via the "execute" function in the AbstractModel class. The arguments required by the IFT-RANDIG command-line program must also be provided.

```
        readProbabilityValues(output, randig);
      } catch (Exception ex) {
        throw new ModelException("Error executing Randig model", ex);
      }
    }
```

The IFT-RANDIG model output file is read by the readProbabilityValues method, shown below.

```
private void readProbabilityValues(WritableClump output, Randig randig) throws IOException,
ModelException {
      log.info("Randig: reading probability from flame length probability file");
      File workingDir = getWorkingDir();

      try {
        BufferedReader br = new BufferedReader(new FileReader(
                              new File(workingDir, FLAME_LENGTH_PROBABILITY_FILE_NAME)));
        br.readLine();
```

The readProbabilityValues method utilizes the burn probabilty estimates from the IFT-RANDIG model results that were written to the flame length probability output file.

```
            String line = br.readLine();

            int i = 0;
            while (line != null) {
               String[] tokens = line.split(",");
               Double x = Double.parseDouble(tokens[0]);
               Double y = Double.parseDouble(tokens[1]);
               Double burnProbability = Double.parseDouble(tokens[2]);
               FlameLengthBurnProbability[] flameLengthBurnProbabilities = new
FlameLengthBurnProbability[20];

               RelativeCoordinate coord = RelativeCoordinate.newBuilder()
                     .plus(Dimension.EASTING, x - 15, Units.METERS)
                     .plus(Dimension.NORTHING, y - 15, Units.METERS)
                     .build();
               output.setValueAt("randig.burnProbability", coord, burnProbability, "count");

               /* Code to determine the low, medium, high, and very high burn probabilities based on the

                * burn probability as modeled by Randig has been omitted.
                */

               output.setValueAt("randig.burnProbability.low", coord, lowBurnProbability, "count");
               output.setValueAt("randig.burnProbability.medium", coord, mediumBurnProbability, "count");
               output.setValueAt("randig.burnProbability.high", coord, highBurnProbability, "count");
               output.setValueAt("randig.burnProbability.veryHigh", coord, veryHighBurnProbability, "count");

               line = br.readLine();
               i++;
            }
            br.close();
         } catch (IOException e) {
            throw new ModelException("Error reading from Randig output file: " + e.getMessage(), e);
         }
      }
```

Burn probabilties are set to the IFTDSS output clump.

The IFT-RANDIG results include low, medium, high, and very high burn probabilities. The values are set to the output clump and can be then be displayed on the IFTDSS web interface for the user to view.

4) Develop the model XML file

Every model package within IFTDSS contains a models.xml file that describes the configuration of the inputs and outputs for each model in the model package. It is a resource file within the Java application. If the model is being added to an existing package, the developer must modify the existing models.xml file; otherwise, a new models.xml file must be created. The NetBeans IDE provides a variety of tools for creating and visualizing XML documents. Three major sections are specified in the models.xml file for each model in the model package: (1) the model properties, including global input properties, the model name, type, and description; (2) the model inputs, including the aperture and input parameters; and (3) the model outputs, including aperture and output parameters. Global input properties are global settings for executing the model. Model inputs should be implemented as model properties if they are not expected to vary across multiple simulations within a single model run or across different coordinates within a landscape. However, values that vary between simulations or coordinates should be defined as model input parameters. The <model>, <input>, and <output> tags in the models.xml file are used to build the "ReadableClump inClump" and "WritableClump outClump" objects, which are parameters of the "execute" method in the Model wrapper class (Model.java). Any model package that implements the com.sti.smf.core.Model interface must use the input clump and set results to the output clump via the "execute" method. Portions of the models.xml file are provided below for the IFT-RANDIG scientific model. Some of the input and output parameters have been omitted.

```
<!DOCTYPE modelPackage SYSTEM "http://localhost/smf_models.dtd">
<modelPackage version="1.1" name="randig">
  <model name="randig" displayName="Random Ignition" version="1.0"
  subprocessTimeoutSeconds="7200">
    <type>java</type>
    <description></description>
    <implementation>com.sti.iftdss.models.randig.RandigModel</implementation>
    <property name="crownFireMethod" default="Finney" displayName="CrownFireCalculationMethod">
      <description>Crown Fire Calculation Method</description>
      <propertyValue displayName="Finney Method" value="Finney"/>
      <propertyValue displayName="Scott &amp;amp; Reinhardt Method" value="ScottReinhardt"/>
    </property>
    <property name="mediumMinFeet" default="4" displayName="Min Flame Length for Medium (feet)">
      <description>Minimum Flame Length for Medium classification, in feet</description>
    </property>
    <property name="highMinFeet" default="8" displayName="Min Flame Length for High (feet)">
      <description>Minimum Flame Length for High classification, in feet</description>
    </property>
    <property name="veryHighMinFeet" default="11" displayName="Min Flame Length for Very High
    (feet)">
      <description>Minimum Flame Length for Very High classification, in feet</description>
    </property>
    <input>
      <!-- Input Aperture -->
      <aperture>
        <dimension name="EASTING" stepMeasure="30" unit="m"/>
        <dimension name="NORTHING" stepMeasure="30" unit="m"/>
      </aperture>
      <!-- Landscape parameters -->
      <parameter name="fuel.model" displayName="Fire Behavior Fuel Model" type="enum" default="1"
      kind="FuelModel-ScottBurgan40">
        <description> A fuel model is a set of fuelbed information needed by fire behavior or fire effects
        models. In the IFTDSS FlamMap SFB the 13 Anderson Fire Behavior Fuel Model and the more
        recently produced Scott and Burgan 40 fire behavior fuel models are supported.  Each fire behavior
        fuel model represents the characteristic biomass or "fuels" that are available to burn for a given
        vegetation type such as grasslands, shrublands, forests and slash. </description>
      </parameter>
      <parameter name="elevation" displayName="Elevation" default="2500" unit="ft" type="int"
      kind="Elevation">
        <description>Elevation is the height above sea level of the area of interest. In the IFTDSS elevation
```

Define the model package name and version.

Define the model name, type, description, and implementation elements. The implementation element must match the package name as defined in the IFT-RANDIG wrapper class. It is the fully qualified Java class name of the SMF-MI.

Define model input property elements which are global model settings.

A set of inputs are described including the aperture and input parameters. These input parameters will be accessible in the input clump parameter of the "execute" function within the RandigModel class in the wrapper.

```
            can have units of meters or feet.</description>
        </parameter>
        <parameter name="slope" displayName="Slope"  default="15" unit="percent" type="int"
        kind="Slope">
            <description>The slope theme is one of the raster themes required to generate a Landscape (.LCP)
            File. It should be integers. Slope can have units of degrees or percent of inclination from the
            horizontal.</description>
        </parameter>
        <validation>
            <minMax parameters="moisture.1hr moisture.10hr moisture.100hr" min="1" max="60"
            unit="percent"></minMax>
            <minMax parameters="moisture.liveHerb moisture.liveWoody" min="30" max="300"
            unit="percent"></minMax>
            <minMax parameters="wind.speed" min="0" max="80" unit="mi/h"></minMax>
            <minMax parameters="wind.direction" min="0" max="360" unit="deg"></minMax>
            <minMax parameters="probability" min="0" max="1" unit="count"></minMax>
        </validation>
    </input>
    <output>
        <!-- Output Aperture -->
        <aperture>
            <dimension name="EASTING" stepMeasure="30" unit="m"/>
            <dimension name="NORTHING" stepMeasure="30" unit="m"/>
        </aperture>
        <parameter name="burnProbability.low" displayName="Burn Probability at Low Flame Lengths"
        unit="count" kind="BurnProbabilityAtLowFlameLengths">
            <description>Burn Probability at Low Flame Lengths</description>
        </parameter>
        <parameter name="burnProbability.medium" displayName="Burn Probability at Medium Flame
        Lengths" unit="count" kind="BurnProbabilityAtMediumFlameLengths">
            <description>Burn Probability at Medium Flame Lengths</description>
        </parameter>
        <parameter name="burnProbability" displayName="Overall Burn Probability" unit="count"
        kind="BurnProbability">
            <description>Burn probability</description>
        </parameter>
    </output>
  </model>
</modelPackage>
```

Validation criteria are defined for any of the input parameters.

A set of output characteristics are defined including the aperture and output parameters. These output parameters must be set to the output clump by the “execute” function within the RandigModel class in the wrapper.

5) Develop the pathway file

The pathway XML file describes the details of the IFTDSS pathway that will include the new model. The elements in this XML file relate to the IFTDSS web application's display for setting up the model and displaying the model output. The requirements in this XML file depend on what is required for the pathway. For example, the pathway file can include various actions, configuration steps, datasets, input, output, etc. For this reason, an example from IFT-RANDIG is not included in this Appendix. The pathway file is not part of the IFTDSS model package, but is required for the model to be included in the IFTDSS web application.

6) Consider the units for the unit set file

The unit set file is a Comma Separated Values (CSV) file that describes the U.S. Customary and metric units currently available for use within IFTDSS. The unit set file is not part of the IFTDSS model package and may be unnecessary for some scientific models. For this reason, an example from IFT-RANDIG is not included in this Appendix.

Summary of Requirements for Adding a Model to IFTDSS

1. Set up the model package.
2. Develop the model calculation software (such as an executable that enables a wrapper class to call the command-line software calculation application).
3. Develop the wrapper class in Java or Python that connects the model to SMF and implements SMF requirements (SMF-MBI).
4. Develop the Models.xml file to configure the model properties, inputs, and outputs.
5. Develop the pathway.xml file to set up the interaction with the model from the web application.
6. Consider what units are required for the model parameters.

Appendix C

Sample Model/Module Information

Supporting Documentation for a Software Model Package

This section describes the information that data or model developers need to provide to the IFTDSS system administrator. This information is provided to users of IFTDSS so that they understand the data and models/modules that they work with.

Name of Software Model: provide the full name and acronym of the software model/module.

Current Version Description/Date: identify the most recent version and its date of release. If there are multiple versions that may be used for different purposes, then describe each of them.

Software Code and History: specify the current software code used to implement the model/module and provide a brief background, if applicable, of the migration from the original coding language, such as FORTRAN, to the current coding language. Please be specific about version or date of the coding language(s) used.

Developer(s) Names, Organization, and Contact Information: identify a key contact who is knowledgeable about the development of the model/module.

> **Note to Users: Only contact the developer(s) of this software model/module with questions relating to the internal functional operations of this tool. Questions regarding how this tool is used within IFTDSS should be directed to the IFTDSS Team by using the Feedback Function available on every page of IFTDSS.**

Scientist Researcher(s) Names, Organization, and Contact Information: identify a key contact knowledgeable about the scientific basis of the model/module.

Availability of the version of record: identify where potential users and other developers can find the version of record, how to obtain an executable copy, and whether the source code is available.

Primary Funding Sources: identify the major funding sources that have contributed to the development of this model/module.

Application Purpose (General): describe the functions in all problem areas that the model/module can be used to perform.

Application Purpose (Fuel Treatment): describe the functions specifically in the fuels treatment problem area that the model/module can be used to perform.

User/Application Documentation: describe where user documentation explaining how to use the model/module is found.

User Application Guidance: a description of the problem situation, geographical location or ecological situation, and the data ranges where the model/module is applicable and can be correctly used. A description of any known problem situations, ecological situations, or data ranges where the model/module is known are suspected not to be applicable. The intent is to provide the user with guidance on how to appropriately use the model/module and how to recognize and avoid inappropriate uses.

Scientific Foundations of the Model/module: describe the research and development history of the scientific foundations upon which the model/module rests. Users and managers need to understand the level of independent review and validation/evaluation that the underlying science at the heart of the model/module has achieved.

- Degree of validation/evaluation and availability of written results:
  - No validation/evaluation has been conducted.
  - Developer(s) have used independent data sources and field trials to validate/evaluate the model/module.
  - Other scientists have used independent data sources and field trials to validate/evaluate the model/module.

- Publication History:
  - List peer reviewed publications concerning the science behind the model/module. Identify specifically those publications that received anonymous peer reviews.
  - List peer reviewed publications focusing on the application of the model/module to relevant problem areas such as case study experiences. Identify specifically those publications that received anonymous peer reviews.
  - List non-peer reviewed publications such as user guides, online tutorials, agency publications, etc.

Training Availability: describe what type of training possibilities exist for the potential users of the model/module and how to obtain such training.

Literature Cited

Include any peer-review publications related to the model/module.