© 2002 WIT Press, Ashurst Lodge, Southampton, SO40 7AA, UK. All rights reserved.
Web: www.witpress.com Email witpress@witpress.com
Paper from: *Brownfield Sites*, CA Brebbia, D Almorza & H Klapperich (Editors).
ISBN 1-85312-918-6

Multimedia-modeling integration development environment

M. A. Pelton & B. L. Hoopes
Battelle, Pacific Northwest Division, USA

Abstract

There are many framework systems available; however, the purpose of the framework presented here is to capitalize on the successes of the Framework for Risk Analysis in Multimedia Environmental Systems (FRAMES) and Multimedia Multi-pathway Multi-receptor Risk Assessment (3MRA) methodology as applied to the Hazardous Waste Identification Rule (HWIR) while focusing on the development of software tools to simplify the module developer's effort of integrating a module into the framework. A module in this plug and play framework can be described as one or more codes, models, or databases that meet the framework communication protocol and can be placed in the visual conceptualization as a discrete part of an analysis. A framework such as this can be used to conceptualize and model the unique scenarios brought about by a Brownfields assessment. In a plug and play system users choose modules without having to worry whether the modules can communicate with each other allowing the user to focus on conceptualization.

An Application Programming Interface (API) has been developed for this framework and is implemented as a Dynamic Link Library (DLL), or shared library. The protocol developed for linking modules together is in the form of data dictionaries, which are designed for flexibility. The approach is to focus on developing these protocols (i.e., boundary conditions) between modules using a distributive environment, which allows multiple developers to collaborate on the same boundary conditions between modules in real time. The API is also used for boundary condition input and output (I/O). This enables the developer to consistently access data needed by other modules without the burden of educating their module on multiple file formats, making population of shared-data sources efficient and consistent. System editors are provided to set up the shared-data sources and the information needed to communicate the module's role in the plug and play system, thus easing the developers work load.

© 2002 WIT Press, Ashurst Lodge, Southampton, SO40 7AA, UK. All rights reserved.
Web: www.witpress.com Email witpress@witpress.com
Paper from: *Brownfield Sites*, CA Brebbia, D Almorza & H Klapperich (Editors).
ISBN 1-85312-918-6

1 Introduction

Environmental assessments have grown from a single media being examined to multimedia multi-pathway environmental assessments. To that end, multimedia multi-model frameworks have been developed. These frameworks in the past, like MEPAS (Multimedia Environmental Pollutant Assessment System) and RESRAD (Residual Radioactivity), have always been hard wired together. Efforts such as FRAMES and 3MRA have helped to change the way in which model integration is approached. The general concepts and specifications of FRAMES are provided in Whelan et al. (1997) [1]. The fact that legacy codes exist and are almost always preferable make an applications programmer's interface (API) an unwelcome but necessary solution if the desired outcome is to integrate models. An API means writing wrappers (pre and post processors) for legacy models, which inevitably can change the way in which a model performs. It also means greater flexibility and manageability in the long run.

What really matters in the development of an API is the audience of modelers and scientists that will use it. Experience thus far has shown this audience to use FORTRAN and C with a little Visual Basic of late. The use of data structures is not present in most FORTRAN legacy codes. So more complex ways of process communication, like DCOM (Data Common Object Model), and SOAP (Simple Object Access Protocol), seemed too complex for a scientist who wants to write or wrap a simple model written in FORTRAN. These issues and others have resulted in an API that links and compiles with FORTRAN, C, and Visual Basic. The API, written in C++, has very little overhead compared to the more complex systems and will compile on most platforms.

A host program, the Framework Development Environment (FDE), executes the API. The FDE is a collection of editors and tools that allow users to define, link, select, and interact with a confederation of environmental codes for environmental and human health analyses. The FDE is an environment where developers can easily integrate models and databases as modules and users can link and run those modules as multimedia environmental analysis.

2 Framework API

The API accomplishes several tasks that would normally be done each time one model is wired to another. In March of 2000, a group of environmental modelers and computer engineers met at the Nuclear Regulatory Commission (NRC) in Washington, D.C. to compile a list of requirements for a multimedia multi-model plug and play framework that facilitated model integration [2]. Most of the requirements from the meeting were used to develop the API. This section gives a very brief overview of the API. Some of the main requirements include but are not limited to:

- handle unit conversions in a consistent and convenient manner

© 2002 WIT Press, Ashurst Lodge, Southampton, SO40 7AA, UK. All rights reserved.
Web: www.witpress.com Email witpress@witpress.com
Paper from: *Brownfield Sites*, CA Brebbia, D Almorza & H Klapperich (Editors).
ISBN 1-85312-918-6

- provide a mechanism that allows for data retrieval of offline as well as online data sources
- provide a mechanism that allows for accessing and running remote models (i.e., remote computing)
- provide a common input/output (I/O) mechanism
- provide error checking for I/O (e.g., variable range, type, units, and cardinality)
- support the latest versions of Borland C++ Builder, Microsoft Visual C++, Lahey FORTRAN-90, and Digital Visual FORTRAN-90 compilers

Whelan et al.[3] provides a complete list of requirements and design. To meet these requirements, a common way to describe data to the system needed to be developed. These meta-data descriptions are at the core of the API. Meta data is the information used to describe data content. The 3MRA system used a set of dictionaries to define the meta-data for module inputs and boundary conditions. The MRA example was used and generalized for the API. These dictionaries are used to describe the meta-data necessary to aid the framework in execution management, variable parameterization, and I/O. The framework uses five types of dictionaries to help define the system architecture.

- The "StartUp" dictionary defines variables for system management (e.g., module list, window settings).
- The "Conversion" dictionary defines the variables for the measurement-conversion library (e.g., measure, unit).
- The "Module" dictionary defines the variables for a module component (e.g., name, version, executable file locations).
- The "Simulation" dictionary defines the variables for a problem conceptualization (e.g. module connectivity)
- The "Data" dictionary defines the variables for module communication (e.g., boundary conditions, module specific inputs).

The API is divided into three categories: Conversion, Dataset, and the System. Each API has an appropriate module or header file for each supported version of compiler. The developer is expected to include the module or header files and compile with the module's processor codes to gain access to the Framework API.

2.1 Conversion API

The Conversion API provides the function calls that allow the developer to work with any linear conversion. The developer is given the capability to add, delete, and retrieve measure and unit labels as well as editing and retrieving the unit conversion. The Convert function is called with a measure (string), an initial value (double), an initial unit (string), and the unit the value is to be converted to (string) as input. The function returns the converted value.

© 2002 WIT Press, Ashurst Lodge, Southampton, SO40 7AA, UK. All rights reserved.
Web: www.witpress.com Email witpress@witpress.com
Paper from: *Brownfield Sites*, CA Brebbia, D Almorza & H Klapperich (Editors).
ISBN 1-85312-918-6

2.2 Dataset API

The Dataset API provides function calls needed to manipulate dictionaries and datasets (see Section 3.2). All classes of modules that are defined are required to register with the framework a set of connection schemes. A connection scheme is a set of dictionaries that define a module's produced and consumed datasets during a simulation (see Section 3.2). When a module is executed during a simulation, one of the defined connecting schemes is passed on to the module through the API. This enables the module the capability to identify its function and access the appropriate datasets. The Dataset API will be the most heavily used and allows the module developer to:

- open and close a session with the API
- retrieve expected dataset names and dictionary types to be produced
- retrieve expected dataset names and dictionary types to be consumed
- read and write data datasets that have been defined by a dictionary.

2.3 System API

The System API provides function calls used mostly by the FDE, and other 'System' modules like sensitivity and calibration modules. These 'System' modules must be defined as such in the Module Editor by setting the module's class type to 'System' (see Section 3.2). Setting the module class flag to 'System' allows the module to gain access to the functions that can manipulate the entire framework at runtime.

3 Framework development environment

The FDE is a suite of editors designed to manage, view, and set up the underlying infrastructure of the framework. The FDE consists of four editors: Conversion Editor, Dictionary Editor, Module Editor, and a Domain Editor. The Conversion Editor creates, edits and manages the measures and units used by dictionaries and datasets to aid in the automatic conversion of units during data retrieval. The Dictionary Editor creates, edits and manages dictionaries that the API uses for defining and categorizing datasets. The Module Editor creates, edits and manages. The Domain Editor is used to define and organize a palette of modules used by the Framework Simulation Editor. The FDE simply displays current system configuration while giving access to the same. The editors have very little logic, treating the API like a black box expecting all verification and consistency checks to be done by the API. When the editors are loaded, all changes made are live, and once changes are saved, they cannot be undone.

Another tool provided but not addressed in this document is the Update Manager that ties to a Linkage Server via http. The Linkage Server is a central information store that facilitates collaboration. The Update Manager provides the capability to acquire and manage the latest updates to conversions, dictionaries, and modules available from the Linkage Server.

© 2002 WIT Press, Ashurst Lodge, Southampton, SO40 7AA, UK. All rights reserved.
Web: www.witpress.com Email witpress@witpress.com
Paper from: *Brownfield Sites*, CA Brebbia, D Almorza & H Klapperich (Editors).
ISBN 1-85312-918-6

3.1 Conversion editor

The Conversion Editor provides an interface for creating, modifying, and deleting measures and their associated unit conversions. Conversions provide the framework with the capability to convert units to and from any desired unit as long as the unit's conversion to the measure's base unit is defined. This provides a modeler the ability to produce and consume data in their desired units without worrying about other modeler's unit expectations. The conversions maintained by the framework are also bound directly to the dictionaries in the framework. Therefore, all conversions must be defined before there can be a reference to a unit by a dictionary. Requested dataset values made with a unit reference that is not contained in the list of conversions can only be consumed or produced in that unit. Currently, only linear conversions are supported.

Each measure must have a base unit defined. Therefore, the Conversion Editor automatically prompts the user to define a base unit after adding a measure. When converting units, the base unit is used as a link between all other units within the measure. A measure categorizes a collection of units that inherit the same measuring properties. For example, Time is a measure with seconds, minutes, hours, and days being related units of Time. Each measure has a base unit that all other units must provide a conversion to that base unit. Using Time as a measure, seconds is a logical choice for the base unit. Once a unit has been added the unit is capable of converting to every related unit by reversing the respective unit conversions. For example, if millisecond were added to the collection of Time and seconds, minutes, hours, and days already in the collection, then milliseconds could be converted to or from any other unit in Time by first converting to the base unit then convert to the desired unit. This reduces the amount of complexity in defining relationships between units because not every combination of units needs to be defined. [4]

3.2 Dictionary editor

Dictionaries in the framework describe the layout of how information is stored in a dataset. All dictionaries in the framework must have a unique name and are stored as an ASCII comma-separated file. The file can reside anywhere on the user's system. Dictionaries can be created outside of the editor using any kind of ASCII text editor. However, dictionaries created outside of the Dictionary Editor must be registered and checked for consistency with the Dictionary Editor by simply opening the dictionary file in the editor. The properties for a dictionary are:

- Variable Count—number of variable declarations in the file
- Dictionary Description—self explanatory
- Dictionary Name—self explanatory
- Privilege—dictionary scope – visibility to other modules
- Version—version number of the file
- Updated—flag to indicate whether the file has changed
- Variable Declarations—set of fields that declare a variable.

© 2002 WIT Press, Ashurst Lodge, Southampton, SO40 7AA, UK. All rights reserved.
Web: www.witpress.com Email witpress@witpress.com
Paper from: *Brownfield Sites*, CA Brebbia, D Almorza & H Klapperich (Editors).
ISBN 1-85312-918-6

Privilege, Version, and Update provide the API with the capability to develop the dictionaries in a distributive environment. This allows developers from different parts of the world to share dictionaries (a.k.a., boundary conditions). Once boundary conditions are shared, then modules can communicate.

Currently, there are four intrinsic data types available for variable declaration from the Dataset API: string, integer, float, and logical. Each variable declaration has these properties:

- Name—self explanatory
- Description—self explanatory
- Dimension—number of indices required for retrieval of a value
- Data Type—type of data stream (one of string, float, integer, logical)
- Primary Key—flag to indicate if variable is a data selection key
- Scalar—flag to indicate if variable is scalar for the defined indices
- Minimum— [minimum length of string][minimum value of float/integer][blank for logical]
- Maximum—[maximum length of string] [maximum value of float/integer][blank for logical]
- Measure—string indicating the measure
- Unit—string indicating the unit of measure
- Stochastic—flag to indicate if variable is stochastic
- Preposition—string used to help describe the who, what, when, where, why
- Index 1, Index 2, …—the list of index references defined for the variable

It is expected that module developers will define the appropriate variables for their respective modules. Special attention must be given to indices of variables. Variables are forward packed, that is, when you have an index of 3, then an index of 1 and 2 also exist. The values will be zero filled or blank filled. All indices must be defined with another variable. For example, the vadose-zone variable kd could be stored by location and chemical. Therefore, the variable declaration for location and chemical must exist. Multidimensional variables are allowed as an index provided their indices are provided by the remaining indices in the variable declaration. No circular dependencies are allowed.

3.3 Module editor

The Module Editor provides the developer the ability to define a module for execution in the framework. A module class determines the level at which modules communicate with the API and how the module execution handled.

There are four classes of modules:

- System—Read/Write/Execute privileges, dataset completeness expected
- Database—Read/Write privileges, dataset completeness not expected
- Model—Read/Write privileges, dataset completeness expected
- Viewer—Read Only privileges

© 2002 WIT Press, Ashurst Lodge, Southampton, SO40 7AA, UK. All rights reserved.
Web: www.witpress.com Email witpress@witpress.com
Paper from: *Brownfield Sites*, CA Brebbia, D Almorza & H Klapperich (Editors).
ISBN 1-85312-918-6

System modules are expected to know a lot about the inner workings of the API and framework. System modules have the capability to do completely manipulate the framework of conversions, dictionaries, modules, and domains. Database and Model classes have the same privileges, except they are prevented from accessing system functions. The only real difference between Database and Model is the expectation on the boundary conditions. Boundary conditions must be complete when output by a Model and/or a System module. Viewers are just that; they provide read-only access to the data.

Module properties consist of five types of information: executable, reference, company, developer, requirements, and connection schemes, most of which is for documentation purposes. Critical for module integration is the executable and connection scheme information. Modules manipulate data and do so by specifying the dictionaries that define the datasets they will manipulate, this declaration is known as a connection scheme. Executable information gathered includes location of interface and model executable, version, and command-line options.

Connection-scheme information is the most critical part necessary for module connectivity and execution management. Connection schemes and the class of the module determine how the framework allows the module to connect and the level at which the module can access the API. Modules need to have connection schemes defined to describe the type of information a module expects to consume and produce. A scheme should be defined for every combination of dictionaries the module being developed can consume and produce. A developer could think of these schemes as function calls to the module.

3.4 Domain editor

The Domain Editor allows the user to create tiers of icons that have the same or similar physical meaning. The developer as well as the end user can choose where modules should appear in their constructed domains. Domains consist of four predefined module classes: Database, Model, System, and Viewer. Under each of these classes, the user can create a group of modules or a grouping of sub-groups of modules. Modules can be placed in more than one tier, but can only be assigned to one class of module. Each domain, class, group, and sub-group can be assigned a unique icon. The icon should convey the physical aspects of the respective tier they represent. By assigning meaningful icons to the tiers, users provide their own understanding and meaning to the visual conceptualization of the simulation displayed by the Simulation Editor.

4 Framework module tester / dataset editor

This tool is designed to assist module developers in testing their modules within and outside of the Simulation Editor, as it pertains to consuming and producing datasets defined by module connection schemes. It is significant in that this tool allows for modules to be developed and tested in parallel with other modules, alleviating the need to develop and test dependent modules sequentially. In other

© 2002 WIT Press, Ashurst Lodge, Southampton, SO40 7AA, UK. All rights reserved.
Web: www.witpress.com Email witpress@witpress.com
Paper from: *Brownfield Sites*, CA Brebbia, D Almorza & H Klapperich (Editors).
ISBN 1-85312-918-6

words, modules that consume data produced by another can be tested without having to wait for the producing module to be completed. The editor runs in three modes determined by command-line arguments. It can be run standalone outside of the Simulation Editor, or within the Simulation Editor as a model user interface (UI) or model executable when specified as such in the Module Editor.

In any case, the tool assumes responsibility for assessing the modules' mode and generating an environment complete with a selected connection scheme (if standalone) and to populate those datasets specific to the module scheme in question. Produced datasets are populated with random values. For every variable in the dataset dictionary, values are generated using the intrinsic random function with the specified range and data type; random lengths are used for strings.

In all modes, the layout of the editor includes a hierarchical display of the applicable datasets and variables accompanied by a tabular display of actual values and indices. The dataset display lists dataset names and identifies each dataset as model specific input, consumed, produced, or referenced. Each variable is identified by description and name. The user clicks on a variable description to view its value(s). The user can also change any value within the data type and range.

4.1 Standalone mode

In Standalone mode, the editor facilitates testing a module's functionality and behavior without the additional complexity of interacting with the Simulation Editor and generating an appropriate simulation. As a standalone component running outside the framework environment, the editor displays the Simulation Editor icon palette from which the user selects a domain, class, group, sub-group, module, and scheme of interest. Given the selected scheme, the editor interacts with the API to generate a simulation (in the background) consistent with that scheme so the module executables will "get the right information" when later they query the API about connections and datasets. The tool then populates all consumed and referenced datasets with random data and displays the dataset contents to the user for viewing and optional editing. To test the module executables, there are buttons on the editor to invoke the UI and/or model as defined in the module editor. When a model executable is invoked and it runs successfully, the user can choose to view the datasets produced by that model.

The testing advantage here is that an environment supportive of the module scheme is guaranteed to exist when the UI or model is executed, and any datasets consumed or referenced by that UI or model will exist and be fully populated. This provides for testing dictionaries and module executable interactions with those dictionaries and corresponding datasets.

© 2002 WIT Press, Ashurst Lodge, Southampton, SO40 7AA, UK. All rights reserved.
Web: www.witpress.com Email witpress@witpress.com
Paper from: *Brownfield Sites*, CA Brebbia, D Almorza & H Klapperich (Editors).
ISBN 1-85312-918-6

4.2 Model input mode

The dataset editor can be assigned as a model user interface (MUI). In this mode the editor limits the functionality to populating the model specific input dataset and allowing the user to view/edit/save that dataset. This capability is useful if there is no MUI for a module that requires user input or the MUI is in development but the model is ready to integrate and test.

4.3 Model run mode

The dataset editor can be assigned as a model executable. In this mode, the editor limits the functionality to populating the datasets produced by the module connection scheme identified by the simulation connectivity. This capability is useful for testing modules that expect to consume datasets that are not available through other modules, for whatever reason.

5 Future tools

The following is a list of recommendations on future additions and updates to the set of tools used to integrate modules and access the API. This list is not meant to be all-inclusive, but has been suggested by users and developers. This list can be used to help prioritize modifications of the software system. Recommended additions and modifications are:

- Define operators that perform on variables defined by dictionaries. Could prove to be very powerful for future model development. Also would cut down on communication traffic with the API reading and writing of variables.
- Flat file string substitution tool to aid in user input through a generic interface. The Multimedia Integrated Modeling System (MIMS) uses this tool to aid in its legacy code integration.
- Create variable declaration mapping for legacy code, a very interesting idea. The object here is to have UI that allows the developer to generate a mapping of legacy code variables to dictionary variables. After the mapping is complete, the program would then generate a file to be included with the legacy code for compiling. Then the developer is able to remove or comment out the legacy variable declaration and start using the API directly with the original variable names.

6 Summary

The API and model integration tools presented here were designed to automate tasks normally done when hard wiring models together and provide engineers the tools needed to integrate legacy models required by regulations and/or regulators to conduct contaminant exposure and risk assessments at brownfield sites. The power of the API and tools provide developers the opportunity to seamlessly link their model with others and gives them access to a platform for comparison.

© 2002 WIT Press, Ashurst Lodge, Southampton, SO40 7AA, UK. All rights reserved.
Web: www.witpress.com Email witpress@witpress.com
Paper from: *Brownfield Sites*, CA Brebbia, D Almorza & H Klapperich (Editors).
ISBN 1-85312-918-6

Acknowledgments

The U.S. Environmental Protection Agency (EPA), U.S. Department of Energy (DOE), U.S. Army Corps of Engineers, U.S. Nuclear Regulatory Commission, and Pacific Northwest National Laboratory (PNNL) under Contract DE-AC06-76RL01830 support this work. A special thank you to K.J. Castleton, K. E. Dorow, G. Whelan, and F.C. Rutz from PNNL for their guidance and support and W.C. Cosby of PNNL for editing the manuscript. PNNL is operated for DOE by Battelle.

References

[1] Whelan, G., Castleton, K.J., Buck, J.W., Gelston, G.M., Hoopes, B.L., Pelton, M.A., Strenge, D.L., & Kickert, R.N., *Concepts of a Framework for Risk Analysis in Multimedia Environmental Systems (FRAMES)*. PNNL-11748. Pacific Northwest National Laboratory, Richland, Washington. 1997.

[2] Whelan, G. & Nicholson, T., *Proceedings of the Environmental Software Systems Compatibility and Linkage Workshop*. PNNL-13654. Pacific Northwest National Laboratory, Richland, Washington, 2001.

[3] Whelan, G., Pelton, M. A., Dorow, K.E., Gelston, G.M., & Castleton, K.J., *Merger Between 3MRA and FRAMES-V1: Design*, PNWD-3145. Pacific Northwest National Laboratory, Richland Washington.

[4] Elder, M., & Pelton, M.A., *Framework Development Environment*, to be published. Pacific Northwest National Laboratory, Richland Washington.