ORNL/TM-2011/261

THE NUCLEAR ENERGY ADVANCED MODELING AND SIMULATION SAFEGUARDS AND SEPARATIONS REPROCESSING PLANT TOOLKIT

August 15, 2011

Prepared by
Alexander J. McCaskey
Jay Jay Billings
Valmor de Almeida

ORNL/TM-2011/261

THE NUCLEAR ENERGY ADVANCED MODELING AND SIMULATION SAFEGUARDS AND SEPARATIONS REPROCESSING PLANT TOOLKIT

Alexander J. McCaskey, Jay Jay Billings, and Valmor de Almeida

Computer Science and Mathematics Division and
Separations and Materials Research Group
Oak Ridge National Laboratory
Oak Ridge, Tennessee

{mccaskeyaj,billingsjj,dealmeidav}@ornl.gov

Date Published: August 15, 2011

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831
managed by
UT-Battelle, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

1. EXECUTIVE SUMMARY

This report details the progress made in the development of the Reprocessing Plant Toolkit (RPTk) for the DOE Nuclear Energy Advanced Modeling and Simulation (NEAMS) program. RPTk is an ongoing development effort intended to provide users with an extensible, integrated, and scalable software framework for the modeling and simulation of used nuclear fuel reprocessing plants by enabling the insertion and coupling of user-developed physicochemical modules of variable fidelity. The NEAMS Safeguards and Separations IPSC (SafeSeps) and the Enabling Computational Technologies (ECT) supporting program element have partnered to release an initial version of the RPTk with a focus on software usability and utility.

RPTk implements a data flow architecture that is the source of the system's extensibility and scalability. Data flows through physicochemical modules sequentially, with each module importing data, evolving it, and exporting the updated data to the next downstream module. This is accomplished through various architectural abstractions designed to give RPTk true plug-and-play capabilities. A simple application consisting of two coupled physicochemical modules is presented in Section 6, demonstrating this RPTk data flow paradigm.

The remaining sections describe this ongoing work in full, from system vision and design inception to full implementation. Section 3 describes the relevant software development processes used by the RPTk development team. These processes allow the team to manage system complexity and ensure stakeholder satisfaction. This section also details the work done on the RPTk "black box" and "white box" models, with a special focus on the separation of concerns between the RPTk user interface and application runtime. Section 4 and 5 discuss that application runtime component in more detail, and describe the dependencies, behavior, and rigorous testing of its constituent components.

2. OVERVIEW

The NEAMS Safeguards and Separations IPSC has been tasked with the job of developing modeling and simulation capabilities and tools to support nuclear fuel reprocessing technology and research. One aspect of this task is to develop a framework for the integration of safeguards and separations systems, as well as model and simulate processes at different levels of fidelity [1,6]. This makes SafeSeps a unique subset of the NEAMS program, as the physicochemical processes inherent to the operation of a spent nuclear fuel plant are highly diverse and numerous. In contrast to other NEAMS IPSCs that focus on physics functionality for one specific nuclear fuel technology, SafeSeps must incorporate may types of physics functionality, such as dissolution, voloxidation, shearing, and solvent extraction, each of which may be as complex as other IPSC codes. The Enabling Computational Technologies (ECT) program element and SafeSeps have partnered to develop a product that allows the coupling of these diverse processes, as well focus on stakeholder satisfaction. This partnership has produced a development team with expertise in both modern software development and safeguards and separations technology. The Reprocessing Plant Toolkit

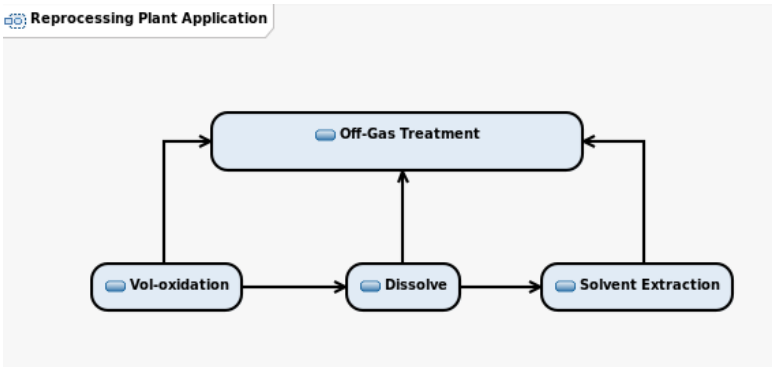


Fig. 1. *Example of RPTk integrated physicochemical processes.*

(RPTk) is the by-product of this dual expertise.

The initial version of RPTk provides a dynamic, integrated, and extensible software framework for the modeling and simulation of the physicochemical processes inherent to spent nuclear fuel reprocessing plants. RPTk will provide stakeholders with a plant simulator capable of designing, monitoring, and controlling the main elements of a reprocessing plant. These capabilities enable users to design dynamic, real-world reprocessing plant flow sheets (Figure 1), conduct simulations of those plant flow sheets, and analyze the results. This enables the exploration of more reprocessing options that aim to accelerate research, as well as reduce cost by focusing experimentation and pilot testing on simulation results. The initial version of RPTk allows the insertion of physics functionality with variable fidelity, and provides stakeholders with a tool to simulate new reprocessing methods, and decrease environmental impact, proliferation, and cost. These capabilities are provided to the user in a way that promotes usability, extensibility, and efficiency.

This vision is inherently complicated, and as such, requires a software development process that efficiently manages the complexities of an extensible, multi-physics framework. Development utilized the industry proven Model Driven Systems Development [4] methodology from IBM Rational to manage this complexity as well as test driven development to ensure a fully tested product. Whenever possible, development leveraged well-known object-oriented software design patterns [2] to reduce workload and increase system reusability, flexibility, and efficiency.

3. SOFTWARE DEVELOPMENT PROCESS

Model Driven Systems Development (MDSD) is a tailoring of the Rational Unified Process (RUP) from IBM that provides developers with a process framework that promotes the rapid design and deployment of complex software, with a focus on requirements, system efficiency, and architectural stability [4]. It is a use case driven, model-based, iterative approach to software development that promotes the design of intricate system models based on the primary uses of the system. Work on RPTk has followed this software development process to manage system complexity and ensure efficient design and implementation of the system.

The iterative nature of MDSD gives developers added flexibility by allowing system development to occur in repeated cycles (Figure 2). This allows developers to take advantage of lessons learned during an iteration and modify the system accordingly in the next. Early iterations place a major focus on the inception and elaboration of a system model, as well as requirements elicitation from stakeholders. As iterations progress and the system model becomes more stable, team focus shifts to system implementation, testing, and deployment [5].

MDSD forces developers to focus design plans on the development of various system use cases, or actions of value that the system performs for the user [3]. This guarantees that the functionality of the software encompasses all user needs. The system model is constructed from the developed use cases, which in turn trace back to requirements gathered from stakeholders.

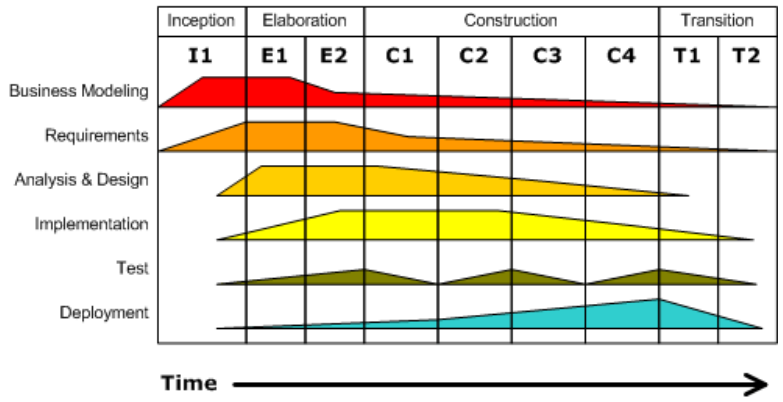


Fig. 2. *MDSD Iterative Approach: Early development focuses on the design of a system model (including use case development, requirements gathering, and black box development). As development progresses, focus shifts to system code implementation, testing, and deployment.*

RPTk development also utilized test driven development, depicted graphically in Figure 3. This development practice ensures the production of fully tested system code by promoting the development of system component tests before the implementation of actual system code. These tests assure that each method of functionality behaves as intended.

The first iteration of RPTk development began with the inception of a system black box. Black box development focuses on the system at the highest level, and details system requirements, system-user interaction, and the specific values the system delivers. The RPTk black box model was driven by stakeholder requirements gathering, and entailed the generation of pertinent system use cases, as well as a high level system context. As iterations progressed, work shifted focus to decomposing the black box model into its constituent components. This decomposition constitutes the RPTk white box model, which focuses on the

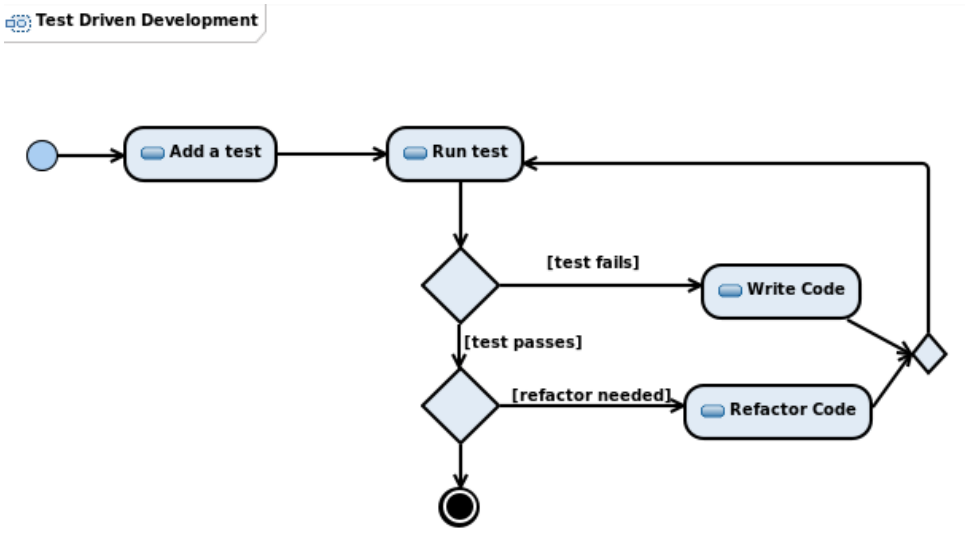


Fig. 3. *Test Driven Development Workflow: First, tests are developed for the various components of a complex system, before the actual component code is written. After these initial tests fail (since there is no system code), the system code is implemented such that it conforms with the designed tests. Upon running the tests again, the test could pass or fail: upon fail, the bug is found, designs are modified, and the code is rewritten; upon success, the code is refactored if needed, and considered complete. This guarantees that all system code produced is fully tested.*

development of a set of components whose mutual dependencies and behavior realize the developed black box use cases.

3.1 RPTK BLACK BOX

RPTk black box development began with requirements gathering through formal interviews with experts in fields such as chemical processing, computational chemistry, and nuclear fuel safeguards and separations. Six interviews were held, gauging user software proficiency, required computational capabilities, and simulation desires. Users requested a variety of features, such as platform independence, reprocessing plant construction, and post-simulation data analysis. These gathered system requirements, in addition to many others, led directly to the inception of six RPTk use cases (Figure 4(a)):

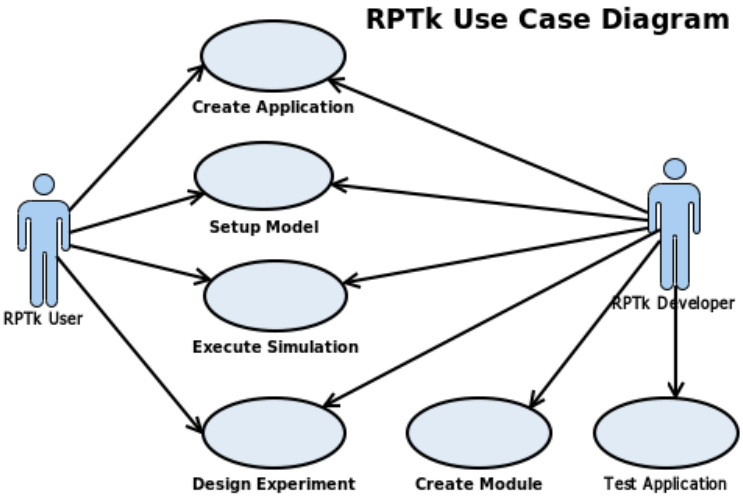
1. *Create Application* - allows a user to construct a collection of dependent physicochemical modules that implements a particular reprocessing technology

2. *Setup Model* - allows a user to set any initial simulation, module, or application configuration options

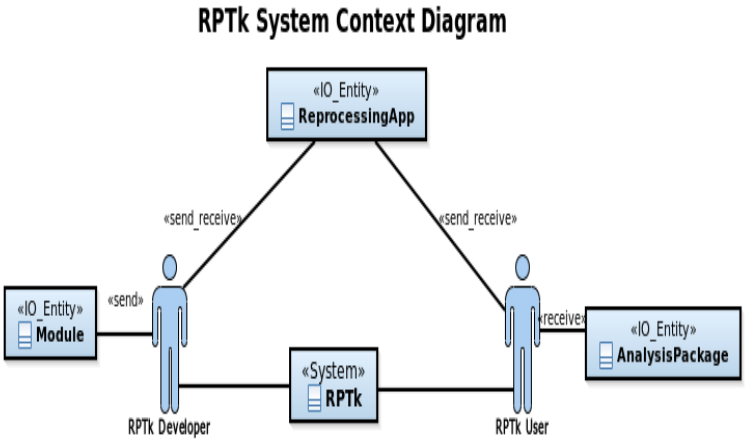
3. *Execute Simulation* - allows a user to couple a constructed application with a configured input model and run a plant-level simulation

4. *Design Experiment* - allows a user to conduct domain-specific analysis by iterating over a simulation with varying initial input configurations

5. *Test Application* - allows a user to perform validation, verification, and uncertainty quantification tests on a constructed application



(a) RPTk Use Case Diagram showing current system use cases.



(b) RPTk System Context Diagram detailing how the system is used.

Fig. 4. *RPTk Black Box.*

6. *Create Module* - allows a developer to create and insert custom physicochemical modules

The first three use cases stand out amongst the others, as they constitute an actual reprocessing plant simulation. They are performed by the RPTk User, which represents a user with analytical skills. The RPTk Developer is similar to the RPTk User in that the Developer can perform these same use cases; however, the Developer can also design and insert new modules that endow RPTk with additional physicochemical functionality.

Black box development next shifted focus to the inception of the RPTk system context (Figure 4(b)), which allows one to consider what the system should look like at the highest level. This diagram is driven by the use cases and describes RPTk in the context of its environment. It details all system users, inputs, and outputs. RPTk takes new modules of physicochemical functionality as input, as well as applications, which are simply collections of those modules with defined inter-dependencies. It outputs resultant simulation data for user analysis. Both actors, the Developer and the User, can input an application to the system and receive pertinent data for analysis. However, only the Developer can create new modules to extend RPTk plant functionality.

3.2 RPTK WHITE BOX

The RPTk white box model consists of two additional levels of abstraction inside the black box. The first level constitutes a decomposition of the RPTk black box level, and continues to be driven by the developed use cases. This decomposition considered a simple separation of concerns: the separation of the user interface and the application runtime (Figure 5).

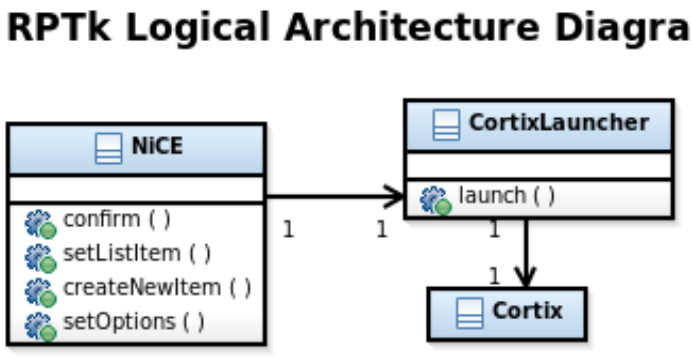


Fig. 5. *RPTk Logical Architecture: At this level, the system composed of the user interface NiCE, the application runtime Cortix, and a system launcher, CortixLauncher, that acts as a mediator between the two.*

RPTk uses the NEAMS Integrated Computational Environment (NiCE) as its user interface. This software tool is developed by the Oak Ridge National Laboratory ECT team, and provides RPTk with a well-supported graphical interface, as well as job launch and problem setup capabilities. NiCE provides the functionality needed by the three primary RPTk use cases: Create Application, Setup Model, and Execute Simulation. NiCE defines these use cases as follows:

1. *Create Application* - NiCE provides a way for developers to insert custom modules of physicochemical functionality into the RPTk framework. It also provides an interface for users to assemble collections of those modules and define their inter-dependencies.

2. *Setup Model* - NiCE provides an interface for users to set desired configuration options. Additionally, it provides functionality to create custom geometries, as well as define meshes and material properties if required by a particular software package.

3. *Execute Simulation* - NiCE provides functionality that enables the coupling of an application and a set of input configurations to launch a simulation, remotely or locally. [5]

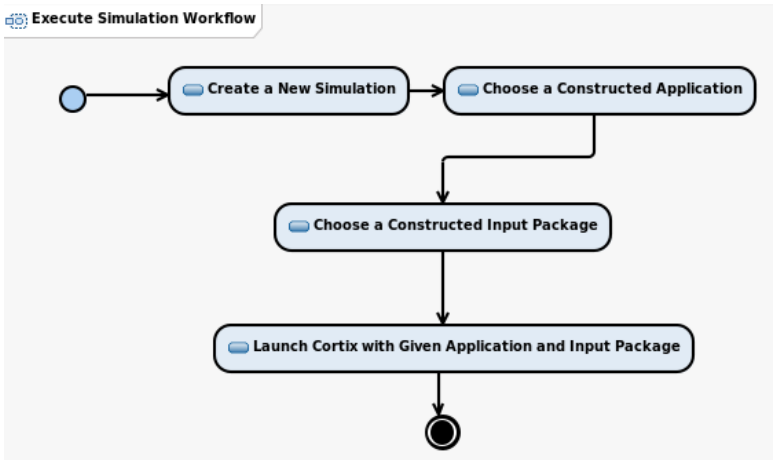


Fig. 6. *RPTk Execute Simulation Primary Flow: Using NiCE, a user creates a new simulation by selecting both a constructed application and an input package. NiCE then launches Cortix on a separate process, with files describing the application and input package.*

The application runtime component, Cortix, acts as the core of the entire RPTk system and is responsible for the flow and evolution of data through a virtual reprocessing plant. Cortix is launched as a separate process through the job launch capabilites of NiCE (Figure 6). Cortix remains coupled to NiCE through the exchange of application specification and input configuration files, which are passed from NiCE to Cortix upon launch. These files are constructed by NiCE, through user input, and detail the user's desired reprocessing application, as well as all simulation and module configuration options. This file coupling between NiCE and Cortix is vital, as it gives users the choice between using RPTk as a stand-alone application, or using the data flow and evolution functionality of Cortix in other applications. The composition of Cortix is significantly complex, and as such, requires a second level of white box modeling, to be described in Section 4.

4. CORTIX

4.1 CORTIX LOGICAL FLOW

The primary purpose of Cortix is to take the user constructed application and input configuration files that NiCE generates and execute a plant-level simulation. This is achieved through the close collaboration of a number of components: MetaDataReader, Broker, ModuleFactory, and Module (Figure 7). These components aid Cortix in the creation of user-specified modules, the registry of those module's inter-dependencies,

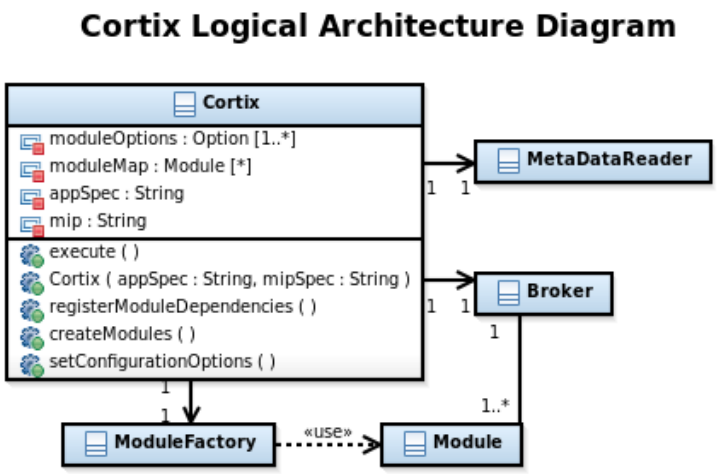


Fig. 7. *Cortix Logical Architecture.*

the setting of any user defined configuration options, and the evolution and flow of data during simulation execution.

The MetaDataReader parses the application specification and input configuration files for requested modules, their inter-dependencies, and initial configuration. The application specification file follows a simple format:

```
/** Specify desired Modules with the keyword **/
/** Module followed by the Module's name **/
Module Module1
Module Module2

/** Specify a Producer-Consumer Module dependency with **/
/** the keyword Connection, followed by the Producer **/
/** Module's name, then the Consumer Module's name **/
Connection Module1 Module2
```

This metadata file, constructed by the user through NiCE, tells Cortix to create an instance of *Module1* and *Module2*, and register a data dependency between the two, with *Module2* consuming any data *Module1* produces. The input configuration file is similar:

```
/** Specify any Module initial configuration options **/
/** with the keyword Option followed by the Module's **/
/** name, the option name, and the option value **/
Option Module1 optionName1 value1
Option Module2 optionName2 value2
```

This format tells Cortix to set the *optionName1* configuration option of *Module1* to *value1*, and similarly for *Module2*.

Figure 8 shows the logical flow of Cortix execution. Modules that are requested through the application specification files are created by the ModuleFactory. Once created, Cortix sets the Module's configuration options according to the input configuration file. Next, the created Modules are registered with the Broker as producers and consumers of data, according to the *Connection* keyword in the application specification.

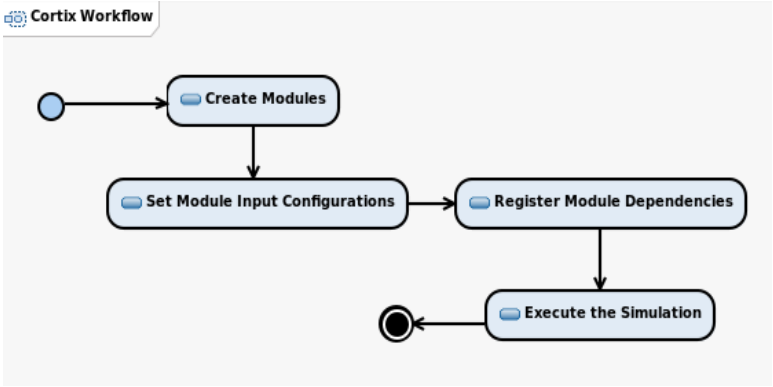


Fig. 8. *Cortix Primary Flow: Cortix first creates the requested Modules, sets those Module's initial configurations, registers the Module inter-dependencies, then executes the simulation.*

This registry constructs a directed graph, with Modules as the vertices and module connections as the edges. Cortix then executes the simulation by walking the graph and executing each Module's physicochemical process on a separate thread or process.

4.2 BROKER

Cortix utilizes a tailored version of the *Observer* object-oriented design pattern [2]. In this pattern, objects observe other objects for changes in data state and react accordingly. The objects being observed are referred to as Subjects, while those observing are called Observers. This pattern, in effect, defines a one-to-many dependency between objects, such that when one object changes state, all dependents are updated automatically. It is applicable to Cortix since Modules can act as both Consumers (Observer) and Producers (Subject) of data during plant runtime. This can lead to sufficiently complex update semantics, so Cortix introduces a mediator, or Broker, to regulate all Module data distribution. Figure 9 illustrates the Broker's various relationships and behavior.

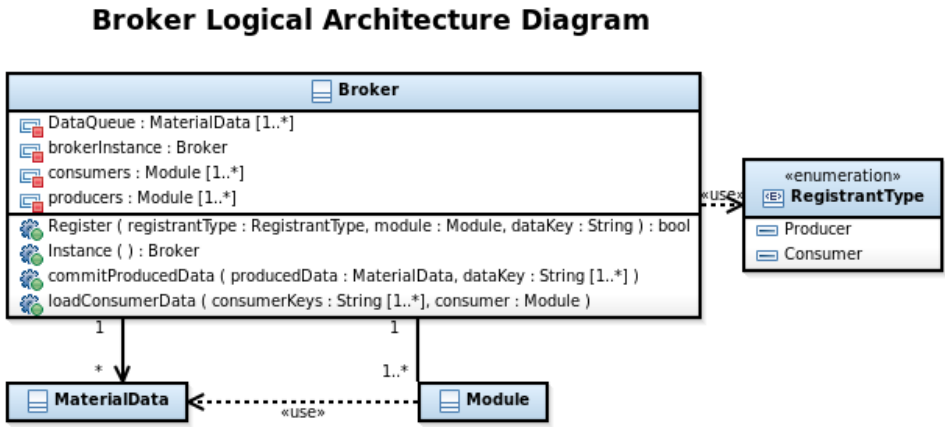


Fig. 9. *Broker Logical Architecture.*

The Broker is notified any time a Module is done with data, or ready for more. This notification triggers

either a commit to Broker data storage, or a data load onto the notifying Module. This Broker behavior facilitates the import/export nature of each Module and ensures true RPTk extensibility and integration. Modules can be inserted into any application without knowledge of the other application Modules, since communication flows solely through the Broker. Additionally, Module encapsulation is preserved by relinquishing the requirement that Modules directly request data from another Module. From the Module's perspective, data should simply flow in, be processed, and flow out.

The Broker registers all inter-Module dependencies by associating keys with each data connection. Each Module can produce *N* data keys, depending on the number of Modules consuming that data. Similarly, each Module can consume as many data keys as requested through the application specification file. For example, the following connections in the application specification file

```
Connection Module1 Module2
Connection Module1 Module3
Connection Module2 Module3
```

would associate *Module1* as a producer of data associated with *dataKey1*, and *Module2* as a consumer of *dataKey1* data. *Module1* would also produce data associated with *datakey2*, and *Module3* would be registered as a consumer of that data key. *Module2* would be a producer of data associated with *dataKey3*, with *Module3* consuming *dataKey3* data. This data key registration allows the Broker to store data in an organized manner, as well as ease Module data requests. Broker storage organization constitutes a mapping *D*:

$$D(dataKey) = (dataObject1, dataObject2, \ldots, dataObjectN), \tag{1}$$

where each vector in the range, referenced by a unique data key, represents a queue of data objects, with a first-in first-out structure. This allows Modules to request data in a time-organized manner; data requested is the data that has been in storage the longest. Modules can process data at different rates, and still not compromise the integrity of the plant simulation.

This behavior and data storage scheme provides a solution to the unique challenges facing the SafeSeps effort. The Broker enables the coupling of any desired physicochemical process by taking data flow out of the control of individual Modules. The diverse set of possible processes can be fully managed and incorporated into one software framework by the Broker.

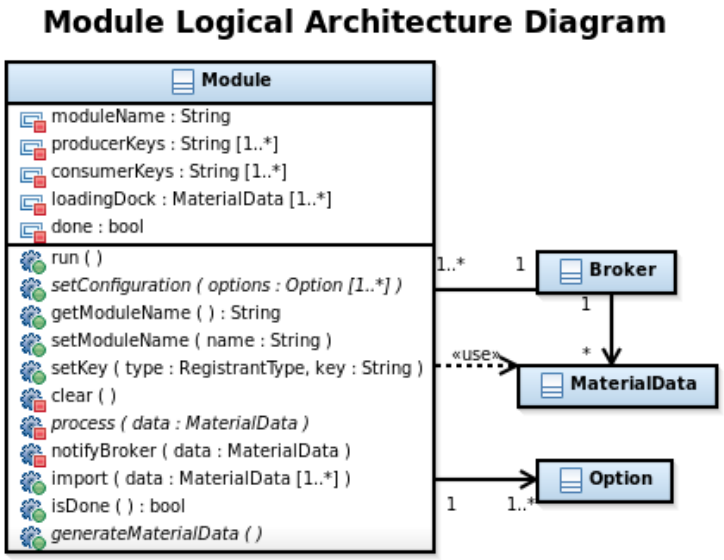


Fig. 10. *Module Logical Architecture.*

4.3 MODULE

Modules are the central abstraction in the Cortix architecture, as they are used to represent any unit of physicochemical functionality inherent to spent nuclear fuel reprocessing [7]. It is an abstract class that must be implemented by developers. Figure 10 illustrates the Module's dependencies and behavior.

Eleven methods specify the behavior of Module (Table 1); however, only three of these define the interface that gives RPTk its extensibility: *setConfiguration*, *process*, and *generateMaterialData*. Primary among these is *process*, which contains the developer's mathematical model that is used to evolve an incoming data object. Cortix executes a Module through the *run* method, which in turn calls *process* when the Module has been given data by the Broker.

The other two methods are fairly straightforward. To set any needed Module options, developers implement the *setConfiguration* method. This method takes a list of options that the developer can use to set any initial configurations, represented as developer-created Module attributes. The last method, *generateMaterialData*, to be fully detailed in Section 4.4, facilitates data flow by providing reference data for mergers with imported data.

The *run* method is the heart of every Module, and is invoked by Cortix as a separate thread of execution. Figure 11 shows the workflow for this method.

Table 1. Table of Module Methods

Method	Description
run	Loops over the available consumerKeys, notifying the Broker for data storage or data loading. When data is present, this method calls *process* to execute the developer's physicochemical processing.
process	Developer-implemented method that contains the Module's specific physicochemical mathematical model.
setConfiguration	Developer-implemented method that sets any module configuration attributes needed before the *process* method can be invoked.
notifyBroker	Notifies the Broker that this Module is done with a data object, and ready to commit it to storage, or if this Module needs more data to be loaded.
import	Imports a new set of data for the Module to process.
generateMaterialData	Returns a developer-implemented subclass of MaterialData. This subclass encapsulates all state variable Entries this Module should need. Used for Modules that are sources of data (no consumer keys), and to perform a data merger.
clear	Removes the next data object from this Module's queue of data.
getModuleName	Returns this Module's name.
setModuleName	Sets this Module's name.
setKeys	Sets which data keys this Module produces and consumes.

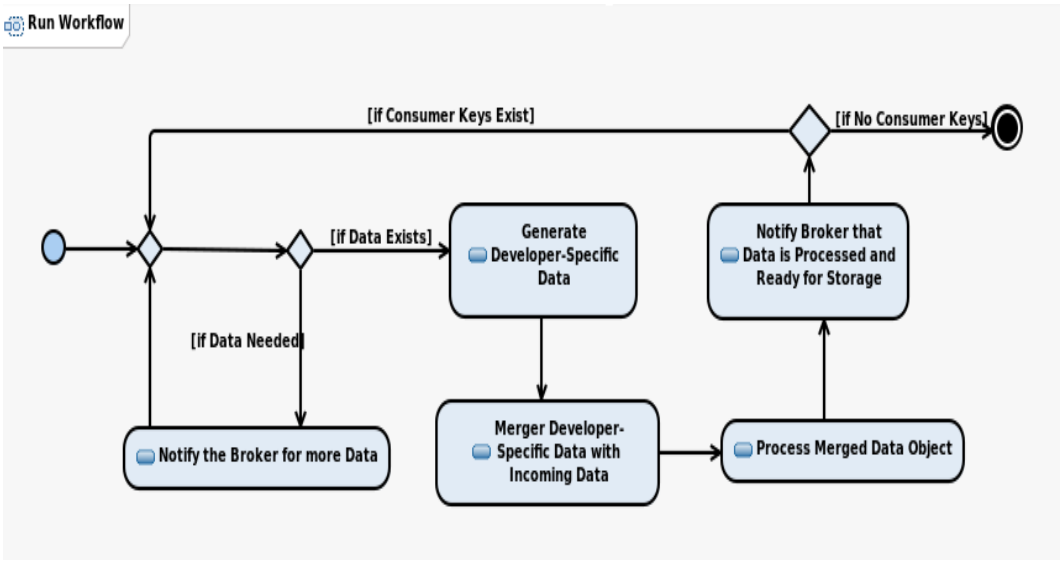


Fig. 11. *Module* run() *Workflow: This method begins by requesting data from the Broker corresponding to the Module's consumer keys. The Module then performs a merger of the incoming data with a developer-specific reference data. The merged data is processed according to the developer's mathematical model and exported to the Broker for storage. This cycle repeats until data has been processed for all consumer keys.*

4.4 MATERIALDATA AND ENTRY

Data flow and coupling in integrated frameworks is inherently difficult. Module developers do not know what other Modules will exist in a given application when they design and implement their Module. This problem can be overcome through an abstract data flow model. Cortix has achieved this through an inheritable MaterialData structure, a data merger mechanism, and a customizable interpolation system.

MaterialData encapsulates the time evolution of N quantities of interest, such as the temperature of nitric acid. These time evolutions are called Entries and are further indexed, within the MaterialData structure, by a SpeciesName and an EntryType (Figure 12). SpeciesName constitutes a list of possible species present in any given MaterialData, and EntryType lists of the possible state variables that a given SpeciesName can have. For example, the concentration of nitric acid is described in MaterialData as an Entry indexed by the SpeciesName *NitricAcid* and the EntryType *Concentration*. This structure implies that MaterialData constitutes a mapping from $\mathbb{N}x\mathbb{N}$ to the space of available Entries. Any species or entry type can be added to MaterialData by simply adding to the respective enumerations (see Section 7 for plans to improve this mechanism). This mapping of Entries allows a developer to process an incoming MaterialData object by simply querying the proper Entry and updating it according to the Module's mathematical model. There are a number of methods on MaterialData to enable this evolution, such as *evolveEntry* (Figure 12).

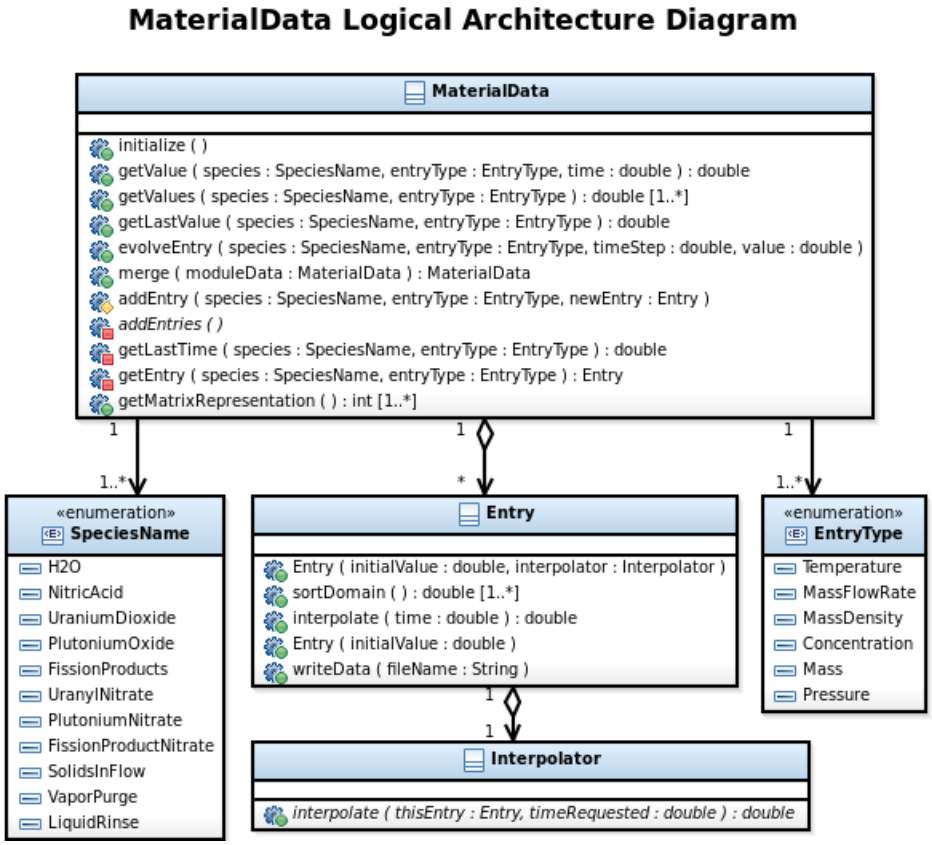


Fig. 12. *MaterialData Logical Architecture.*

Since the time domain of an Entry is a discrete set, it is necessary to address the situation where a processing Module asks for an Entry value at a time that does not exist. Inevitably, a downstream Module will request data at a time that the upstream Module did not record. Cortix handles this situation through an

inheritable interpolation structure. Developers can implement their own interpolation routines by defining a custom Interpolator class that provides a precise interpolant value in the event that data is requested at a time index that is not available. A cubic spline interpolation routine is used if a developer does not wish to create a custom interpolation method. This interpolation scheme keeps RPTk plant simulations time independent and allows developers to focus on their own Module implementations.

Data coupling is difficult in multi-physics frameworks because developers only know what data they need for their physicochemical process, not what other Modules will provide, or what downstream Modules will need. This lack of knowledge eliminates the possibility for a Module to request data from another during runtime, because that data is not guaranteed to be there. Cortix tackles this problem in two ways:

1. Making MaterialData an abstract, inheritable structure that developers define to fit their Module's needs

2. Providing a MaterialData merge mechanism to combine an imported, upstream data object with that Module developer's specific MaterialData implementation.

To subclass MaterialData, developers simply implement the private *addEntries* method. This method should be a series of calls to the protected *addEntry* method, detailing the SpeciesName, the EntryType, and the initial value of the Entry. Each time an instance of the subclassed MaterialData is created, the object's *initialize* method must be invoked to fill the object with the specified Entries.

The Cortix data merger mechanism ensures that every Module developer has the required data to perform their Module's specific physicochemical processing. Developers provide a custom MaterialData object to Cortix through Module's abstract *generateMaterialData* method (briefly mentioned in Section 3.2). This method constructs an instance of the developer's MaterialData subclass, initializes that object, and then returns it. With that in place, Cortix can merge the imported data object with the Module developer's data implementation by invoking MaterialData's *merge* method. This method converts the two data objects into corresponding matrix representations, sums the two matrices, analyzes the result, and then returns a merged data object. The data matrix is generated by MaterialData's *generateMatrixRepresentation* method, and converts the MaterialData discrete Entry map into a matrix containing only ones and zeros. The $(i, j)^{th}$ matrix component is a one if an Entry exists at the i^{th} EntryType and j^{th} SpeciesName, and a zero if it doesn't. The resultant matrix analysis is shown graphically in Figure 13. This merger ensures that data exists for any Entry a developer may need, and decouples the development of a Module from any future application environment.

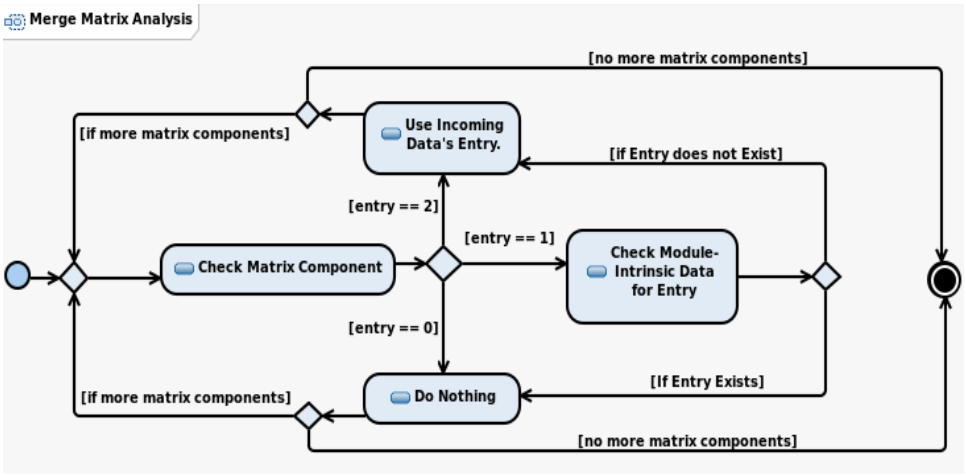
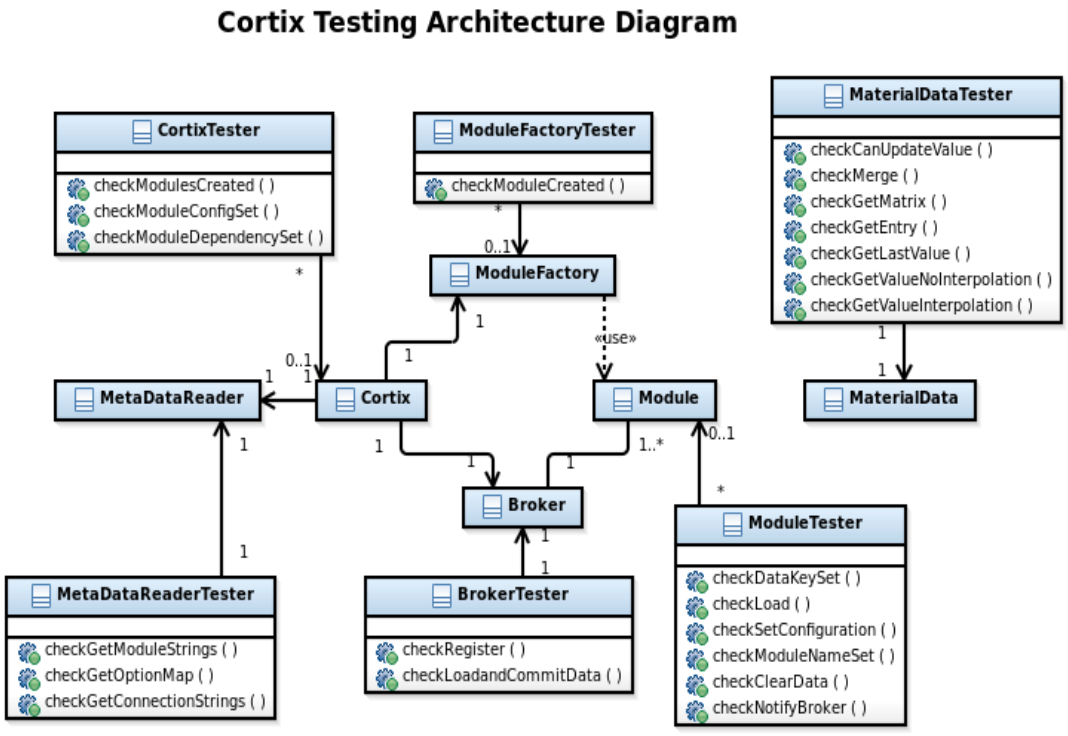


Fig. 13. *Merged Matrix Analysis: If a component of the merged matrix is 2, then both data objects had an Entry for that SpeciesName and EntryType, so the incoming Entry is retained. If the resultant component is a 0, then neither had the Entry and nothing is done because neither data object needed that Entry. If the resultant component is a 1, the module-intrinsic data object is checked to see if that Entry existed before the merger. If not, then Cortix adds the incoming data's Entry at that SpeciesName and EntryType to the data object being returned.*

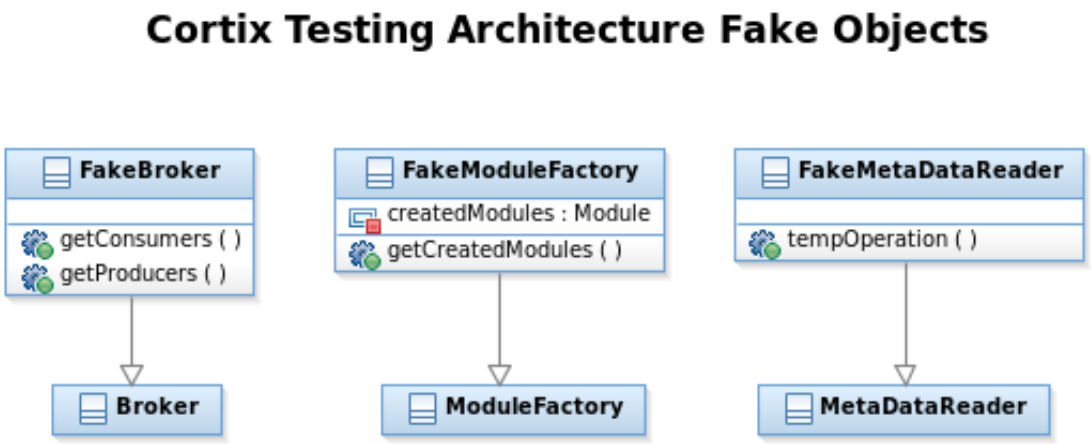
5. CORTIX TESTING ARCHITECTURE

A rigorous testing architecture has been developed for Cortix to produce fully tested code. This architecture was constructed following the test driven development process mentioned in Section 3. The testing model was designed in parallel with RPTk development, and utilized principles found in MDSD. The architecture, shown in Figure 14(a), was designed to perform unit testing on each Cortix component. It was subsequently pushed to code and provided a testing harness for future Cortix code implementation. Every component of Cortix has a corresponding Tester class that tests all major component methods.

Additionally, for a majority of Cortix components, the testing architecture introduces a fake component. These fakes (Figure 14(b)) are used to break component dependencies. Every component relies on other components for its own functionality. This is disadvantageous during unit testing, as each test is only concerned with a single component. These fakes effectively simulate the behavior of their real counterpart, allowing execution to complete without touching another component's code. This gives the developer certain knowledge that any failure must exist within the component being tested, and not elsewhere in the system.



(a) Cortix Tester Components.



(b) Cortix Fake Objects.

Fig. 14. *Cortix Testing Architecture.*

6. SIMPLE TEST APPLICATION

A simple plant application has been developed to demonstrate the capabilities of Cortix, specifically its data flow architecture and evolution during plant simulation runtime. This application consists of two coupled Modules. The first Module's *process* method was implemented to model the concentration versus time of H_2O as a sine wave. The second Module consumes this data and evolves it further, this time modeling the evolution as a simple sinc function

$$f(x) = \frac{sin(x)}{x}, \tag{2}$$

which is calculated using the sine data generated from the previous Module. The application had to take advantage of the Cortix data merger mechanism, as these Modules were intentionally not given the same Entries. Additionally, these Modules were not calculating over the same time steps, which forced the second Module to make significant use of the default interpolation scheme provided by Cortix.

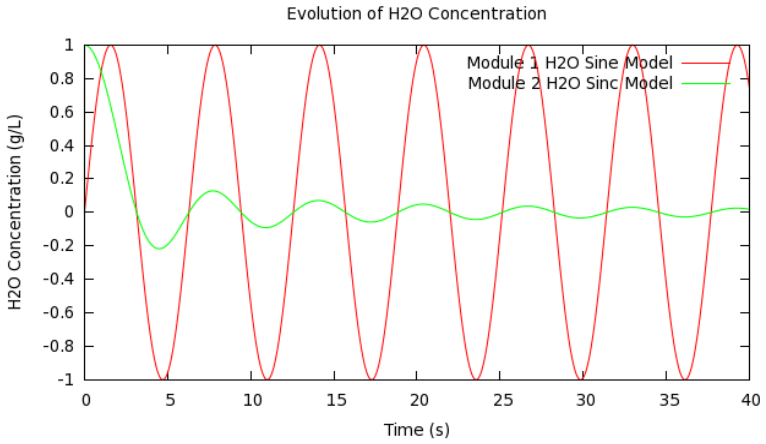


Fig. 15. *Simple test of Cortix: Sine wave evolved into Sinc wave.*

Figure 15 shows the end result of this simulation. Cortix is able to properly simulate the real-world flow of material through a number of physicochemical components. At each Module, data is evolved based on previous processing and the current Module's mathematical model.

7. COMMENTS AND FUTURE WORK

The development of Cortix has reached a satisfactory state. It successfully connects custom Modules, regulates data flow, and performs physicochemical evolutions in a sequential manner. It is tested with a testing architecture that can be easily extended to include new RPTk components. Despite these successes, there are pending work items for the RPTk development team to consider in the near future:

1. The SpeciesName and EntryType indices for MaterialData are not easily extensible. Adding to the enumerations in the source code is the only way to extend MaterialData with new species or entry types. Future work should include developing an extensible enumeration structure to accomodate these desires during plant runtime.

2. The application and input configuration specification files are currently read in as plain text. The MetaDataReader reads in these plain text files and passes that data back to Cortix. There is no file output functionality in this component. Future work should focus on developing an improved file IO system that allows users to read and write files in a wide variety of formats.

3. There is no error reporting system in Cortix. If an error occurs, and is severe enough, the system simply exits. It should be able to catch any error and react accordingly, so that simulation runtime does not need to exit. The RPTk development team plans on constructing an error reporting system model to encapsulate this behavior.

4. The current version of Cortix places every Module on its own thread. This is great for small workstations, but to accomodate larger HPC clusters, a new distributed computing system is needed. This system should be able to query the number of processors on the system, calculate the requirements of each Module, and then allocate the appropriate number of processors to each Module.

REFERENCES

[1] A. Larzelere. *Nuclear Energy Advanced Modeling and Simulation*, http://science.energy.gov/media/ascr/ascac/pdf/meetings/nov09/larzelere.pdf.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, 1995.

[3] K. Bittner and I. Spence. *Use Case Modeling*. Addison-Wesley, Boston, 2003.

[4] B. Nolan, B. Brown, L. Balmelli, T. Bohn, U. Wahli. *Model Driven Systems Development with Rational Products*. February 2008 IBM.

[5] J. J. Billings et al. *Designing a Component-Based Architecture for the Modeling and Simulation of Nuclear Fuels and Reactors*. CompFrame 2009, Portland, OR, 15-16 November 2009.

[6] de Almeida, V. F., McCaskey, A., and Billings, J. J., *Safeguards and Separations Modeling and Simulation: Reprocessing Toolkit Development*. 2011, Oak Ridge National Laboratory Letter Report, ORNL-LTR-2011-115.

[7] de Almeida, V. F., *Progress on Plant-Level Components for Nuclear Fuel Recycling: Commonality*. 2011, Oak Ridge National Laboratory Letter Report, ORNL-LTR-2011-176.