INL/EXT-14-33003

Implementation of a Bayesian Engine for Uncertainty Analysis

Leng Vang
Curtis Smith
Steven Prescott

August 2014

The INL is a U.S. Department of Energy National Laborotory operated by Battelle Energy Alliance

INL/EXT-14-33003

Implementation of a Bayesian Engine for Uncertainty Analysis

Leng Vang
Curtis Smith
Steven Prescott

August 2014

Idaho National Laboratory
Idaho Falls, Idaho 83415

Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

SUMMARY

In probabilistic risk assessment, it is important to have an environment where analysts have access to a shared and secured high performance computing and a statistical analysis tool package. As part of the advanced small modular reactor probabilistic risk analysis framework implementation, we have identified the need for advanced Bayesian computations. However, in order to make this technology available to non-specialists, there is also a need of a simplified tool that allows users to author models and evaluate them within this framework. As a proof-of-concept, we have implemented an advanced open source Bayesian inference tool, OpenBUGS, within the browser-based cloud risk analysis framework that is under development at the Idaho National Laboratory.

This development, the "OpenBUGS Scripter" has been implemented as a client side, visual web-based and integrated development environment for creating OpenBUGS language scripts. It depends on the shared server environment to execute the generated scripts and to transmit results back to the user. The visual models are in the form of linked diagrams, from which we automatically create the applicable OpenBUGS script that matches the diagram. These diagrams can be saved locally or stored on the server environment to be shared with other users.

CONTENTS

FIGURES

ACRONYMS

API	Application Programming Interface
BUGS	Bayesian inference Using Gibbs Sampling
CSS3	Cascade Style Sheet version 3
DAL	Data Access Layer
ECMAScript	European Computer Manufacturers Association Script
HTML5	Hypertext Markup Language version 5
HTTPS	Hypertexts Transfer Protocol Secured layer
IDE	Integrated Development Environment
INL	Idaho National Laboratory
MCMC	Markov chain Monte Carlo
PC	Personal Computer
VOBS	Visual OpenBUGS Scripter

Implementation of a Bayesian Engine for Uncertainty Analysis

1. BACKGROUND

As a part of modern risk analysis, there is a need to have a statistics analysis tools installed on a centralized high performance computing environment in which is to be shared with collaborating analysts. Often times, collaborators do not have readily available their own high performance environment and may want to make use of shared computational resources. However they may not have or are not allowed to have direct access INL internal networking servers to access necessary resource tools. Consequently, the implementation of a shared resources in a cloud-based environment can help to improve collaboration.

2. VISUAL OPENBUGS SCRIPTER

2.0 What is OpenBUGS?

OpenBUGS is an open-source software package for performing Bayesian inference Using Gibbs Sampling (Ref 1). The scripting language uses in OpenBUGS is called 'BUGS', a dialect of the open source statistical package language called 'R'. OpenBUGS has a desktop application with limited scripting capability. It also has an accessible batch command line mode. This later function capability is what we used for this project.

In the context of PRA, where we use probability distributions to represent our state of knowledge regarding parameter values in the models, Bayes' Theorem gives a posterior (or updated) distribution for the parameter (which may be a vector) of interest, in terms of the prior distribution and the observed data, which in the general continuous form is written as:

$$\pi_1(\theta \mid x) = \frac{f(x \mid \theta)\pi(\theta)}{\int f(x \mid \theta)\pi(\theta)d\theta}.$$

In this equation, $\pi_1(\theta|x)$ is the posterior distribution for the parameter of interest, denoted by θ. The posterior distribution is the basis for all inferential statements about θ, and will form the basis for model validation approaches. The observed data enters via the likelihood function, $f(x|\theta)$, and $\pi(\theta)$ is the prior distribution of θ.

The denominator of Bayes' Theorem is sometimes denoted f(x), and is called the marginal or unconditional distribution of x. Note that it is a weighted average distribution, with the prior distribution for θ acting as the weighting function. The term f(x) is also referred to as the **predictive** distribution for X.

The **likelihood**, f(x|θ), is most often (in many different PRA applications) binomial, Poisson, or exponential.[a] Below, we describe an OpenBUGS application that is frequently found in PRA applications.

OpenBUGS is able to evaluate very complex multidimensional and hierarchical problems, where closed-form solutions are not possible. OpenBUGS uses a Markov chain Monte Carlo (MCMC) approach to sample directly from the joint posterior distribution. Example problems that are are solvable with OpenBUGS include uncertain data problems, analysis involving non-conjugate prior distributions, and hierarchical models with many parameters.

OpenBUGS provides a "programming-style" scripting language to specify the model to be evaluated. The script is then compiled in order to process the model and associated data. Rather than require analysts to write the BUGS script, the diagramming tool described in this document provides an abstraction of a complicated model – this diagram is then translated into BUGS script and run on the server via the cloud-based framework.

As an example of the types of problems, and associated scripts that are required, we will evaluate a simple case of two pumps and assume the following operational (run) data:

Failed Test	***Pump A***	***Pump B***
1	0	1
2	0	1
3	1	0
4	1	1
Total Failures	2	3
Operational time (hours)	920	920

We will need to use this data to estimate a failure rate (this failure rate would be used in a PRA to predict failures of pumps). Since this is Poisson data, we could assume a Jeffreys noninformative prior on the failure rate to obtain the posterior distribution directly. However, to help illustrate the use of OpenBUGS, we will evaluate two cases, one with the Jeffreys noninformative prior and one with a non-conjugate prior.

A general BUGS script usually has three parts: 1) comments, which are preceded by "#", 2) the "model" section, and 3) a "data" section. For our first example, we will need to use the Poisson model and a gamma (Jeffreys) distribution.

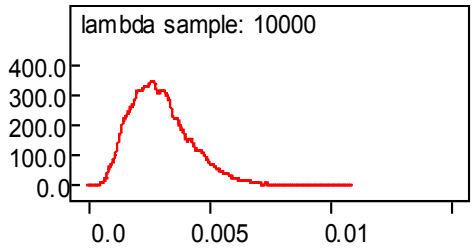
The BUGS script for the first case (Jeffreys prior) appears as:

[a] When dealing with repair times and recovery times, the likelihood function may be lognormal, Weibull, or gamma. If the observed variable is an unavailability, the likelihood may be a beta distribution.

```
# Pump Fail To Run - Five failures out of 1840 hours of operation.
model {
 mu <- lambda * OperatingTime
 pump.ftr ~ dpois(mu)              #Poisson model
 lambda ~ dgamma(0.5, 0.00001)     # Approximate Jeffreys prior for Poisson data
 ftr.post ~ dpois(mu)              #Predicted number of failures from posterior
}
data
list(pump.ftr = 5, OperatingTime = 1840)
```

The **results** of the analysis are:

	mean	s.dev.	median
ftr.post	5.492	3.294	5.0
lambda	0.002986	0.001268	0.002808



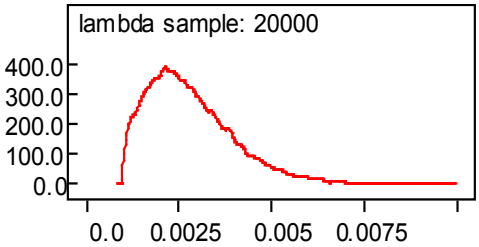
These numerical results would be available for other uses in the PRA model, for example to populate basic events representing pump failures in a simulation or fault tree model.

Now for the second case, let us assume that we believed that the failure rate (prior to collecting any data) was between 0.001 and 0.01 per hour. We did not have any preference over values in this range – thus we could assume that a uniform prior over this range is applicable (note that this is not a conjugate prior). The OpenBUGS script for this case would be constructed by modifying the prior line:

```
lambda ~ dunif(0.001, 0.01)          # Uniform prior
```

Moreover, the results of such a modification are:

	mean	s.dev.	median
lambda	0.002794	0.001172	0.002592



The cloud-based Bayesian engine tool described in this document removes the need for analysts to write BUGS script – instead they would describe a model using the diagramming approach discussed in Section 3. The analysis for the model would then be run automatically using cloud-based resources.

2.1 What is Visual OpenBUGS Scripting?

Visual OpenBUGS Scripting is an Integrated Development Environment (IDE) in a web application that allow scripting of the BUGS script using drag-and-drop visual diagraming tools. The application has built-in a pallet of diagraming shapes with each represents a language's syntax such as keyword, property, group, or relationships that tie the shapes together; all to generate script code block ready to be submitted to OpenBUGS for execution. The application also has built-in capability to manage user's scripts by either stored them locally on user's local drive or to upload for storage on a server.

3. ANALYSIS AND DESIGN

3.1 Bayesian Engine Concept

Visual OpenBUGS Scripter (VOBS) has been designed to be flexible and enables user access from most major computing platforms, such as Microsoft Windows, Linux, Android or Apple OS with a single codebase. Being flexible allows users with limited computing power to offload heavy calculations to a shared high performance computing environment where OpenBUGS is installed and can be much more efficient.

A viable solution is to develop a web-based application as a web service running on a server computer that has a direct communication to OpenBUGS as well as a database management system which is also installed on a high performance computer server. The client application is developed as a single page application using browser technology. The benefit of being a web application is that users do not need to download and install the application in order to use it. The client application also has capabilities to allow the user to build a script visually and to submit it to the server for execution. The result status of the analysis is communicated back to the client application for users' progress monitoring. Users may maintain their own script code by saving locally (client PC) or may opt to save to the server with associated user account. Predefined scripts may be saved as a library for later re-use. These libraries may be saved locally as a private library or uploaded to the server to share with other analysts as shown Figure 1.

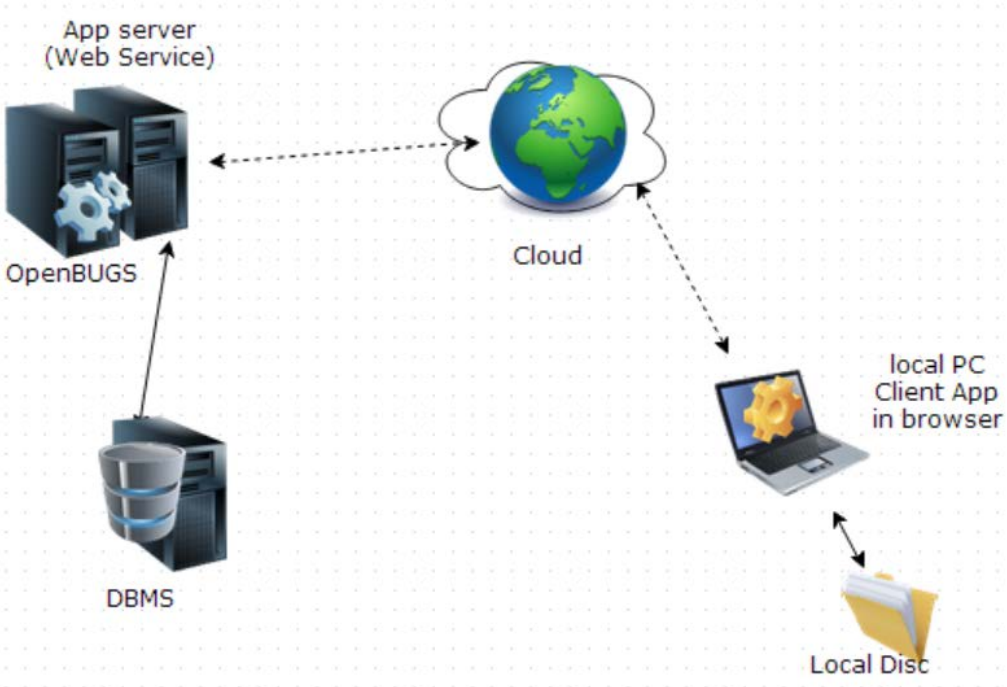


Figure 1: Visual OpenBUGS approach.

3.2 Design

To have a positive user experience with an interactive nature of a drag-and-drop user interface, as well as being able to run inside a web browser, an application framework and supporting libraries are required. To leverage users' experience from the existing desktop OpenBUGS application, a single page web application (SPA) approach is used. In order to facilitate a minimum development time, we used a commercial library called mxGraph (developed by jGraph Inc.) in order to support the application framework and diagramming functionality. The script drawing module of the application is a 100% web-based client side program with limited access for basic data storage. Consequently, it requires a server side program for a more sophisticated data structure storage and to provide direct access to the shared OpenBUGS application (where the analysis is performed). The communications between the client and server application are handled through secured layer of HTTPS protocol as web service calls. This client-side program is illustrated in Figure 2.

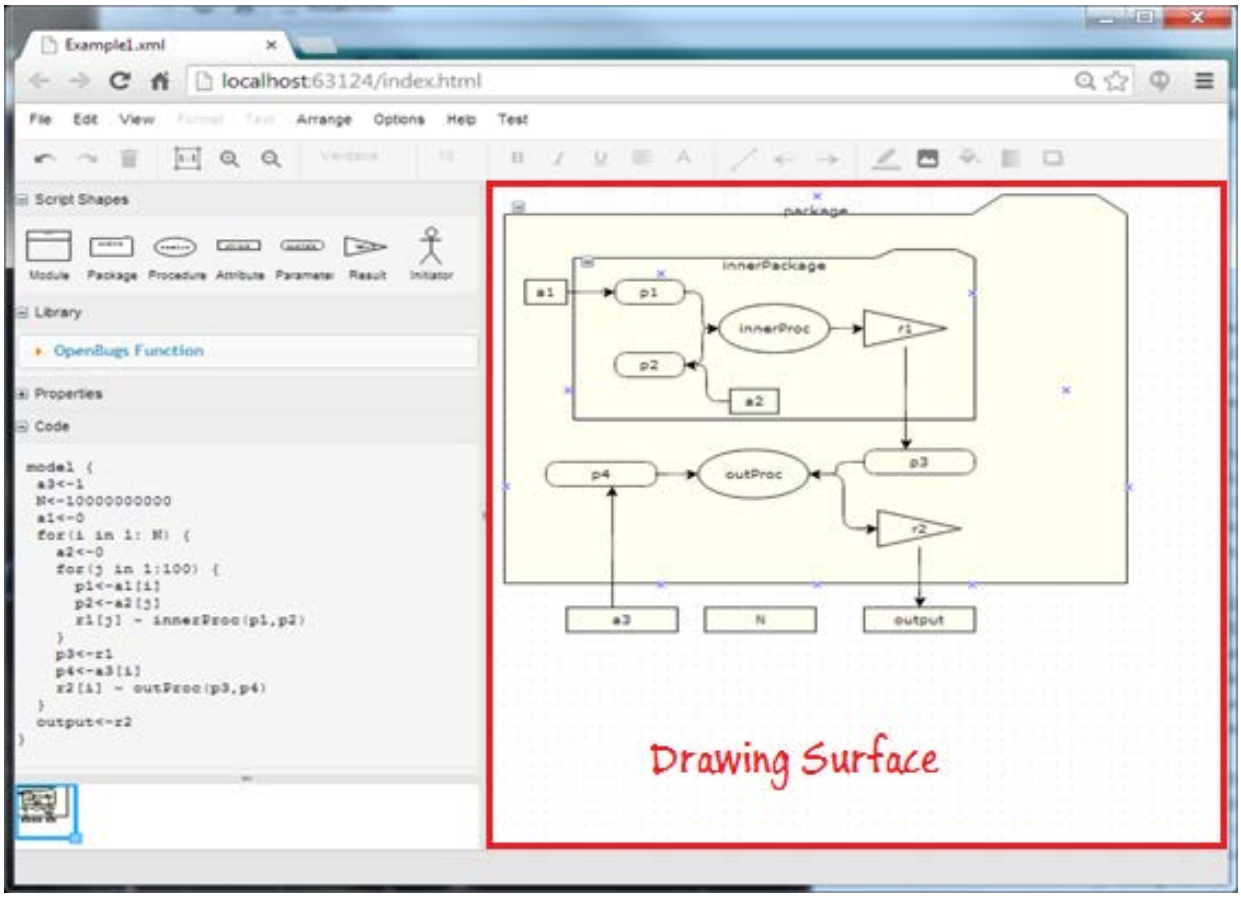


Figure 2: Client-side application for creating the Bayesian analysis models.

3.2.1 Server Application Program

The server program consists of database management system for maintaining user accounts and associated scripts, diagrams representing models, and the analysis results of calculations when necessary. It requires a server application for managing the database and is responsible for handling of data retrieval and saving user data. It also serves to communicate all requests to and from the client application. In our prototype, the database storage is designed for Microsoft SQL Server 2012. The server application is a web-service based and is developed using Microsoft .NET with C# as the programming language.

3.2.1.1 Data Access Layer

The Data Access Layer (DAL) is a module that is responsible for handling data query requests into the underlying database and relays the queried data to the Service Layer. It also is responsible for handling saving and updating requests from the Service Layer to update or save data records into the database.

3.2.1.2 Service Layer

The Service Layer handles requests to and from the client application. It also is responsible to query the Data Access Layer for retrieve, update, and to save user data.

3.2.1.3 Exposed functions

The Service Layer defines and exposes functions that are accessible by the client application. These functions are specific and are called by the client for handling individual requests, for example get_userAccount, update_UserAccount, save_Library, etc.

3.2.2 Client Application Stack

The client application represents the bulk of the development. It uses the mxGraph library extensively to create the entire user interface. It is written with 100% JavaScript, including the dependent libraries. The user is not required to pre-install any dependent tools in order to use it other than the minimum requirement mentions below. However, a user must create an account with the server if he/she wishes to upload libraries to be stored or shared with the community.

The client user interface resembles a desktop application with multi-document windows, menus, tools bars, and side panels with pallets of diagraming shapes, properties, and generated OpenBUGS script code.

The client application is developed with forward compatibility features that dependent on current technologies to function well. A minimum requirement is a browser that support:

- HTML5
- CSS3
- JavaScript ECMAScript 5.1

In Appendix A, we provide examples of use cases scenarios in order to demonstrate how the client application is used.

3.2.2.1 Menus Navigator

The menu (see Figure 3) is the main navigator for most functionality within the application. It contains options and sub-navigational options:

1. File – Document functions to create New, Open, Save, Save As, Save as Library, Import, Export, Page Setup and Print diagram.
2. Edit – Allows basic editing of a shape including: Undo, Redo, Cut, Copy, Paste, Delete, Edit Data, Select Vertices, Select Edges, Select All
3. View – Basic diagram zooming and page fitting.
4. Format – Built-in shape shading and visual decoration functions.
5. Text – Allows modification to fonts, colors, alignments and text displacement.
6. Options – Allows setting of interaction options.

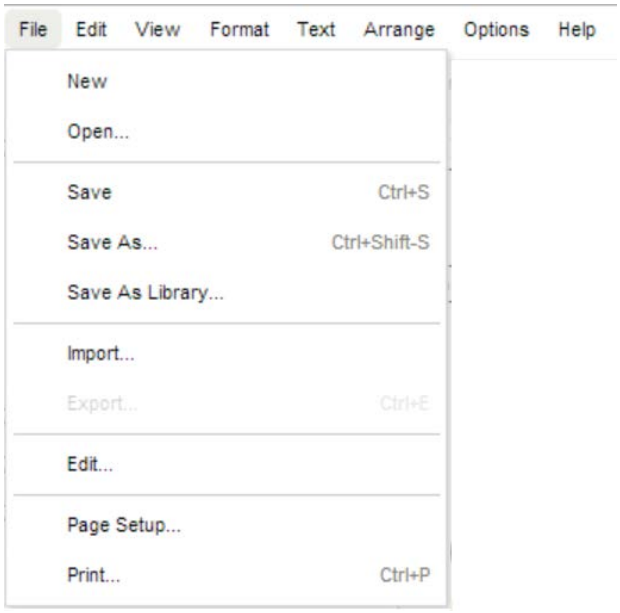


Figure 3: Menu Navigation

3.2.2.2 Tools Bar Navigator

Tools Bar contains duplicated navigational functions found in the menu navigation. The purpose of the Tools Bar is to list those function that are in frequent used as well as to provide quick access to them. Figure 4 shows an example of the Tools Bar containing options found in various locations in the menu for Undo, Redo, Delete (trash bin), Zoom 1:1, Zoom in, Zoom out, Font family, Font size, Font Style (Bold, Italic, Underline, alignment and color), connection line style, connection start node, connection end node symbols, line color, insert image, flood fill, fill shading and shape shadowing.

The Tools Bar is dynamically context sensitive dependent on users' activities on various part on the screen or functionality being performed.

Figure 4: Tools Bar navigator

3.2.2.3 Drawing Surface

The main panel (to the right of the client window) is the Drawing Surface where to the user can place or drop a shape or make connections. The Drawing Surface accepts items dropped from the Shapes pallet and the Library list. It also accept user direct interactions as discuss later in the User Cases under the Appendix section. The Drawing Surface is the large blank area of the screen (see Figure 2).

3.2.2.4 Shapes Pallet

The Shapes Pallet contains various shapes, each representing a specific role corresponding to the BUGS scripting language. Each shape may have multiple distinct rules governing connecting and dependencies.

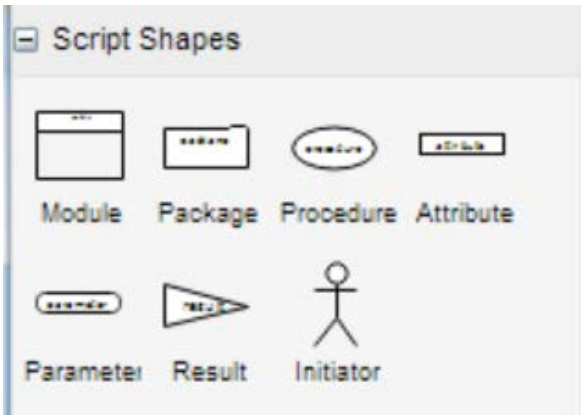


Figure 5: Pallet of script shapes

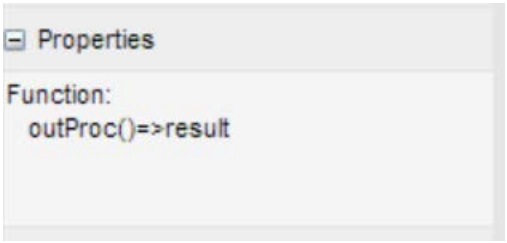
The following defines shapes and rules:

1. Module shape – a container shape for grouping of shapes. Containers do not accept incoming connector nor do they have outgoing connector. Module can be nested within another container but it does not generate any code.
2. Package shape – another scope/container shape, however if the <lower bound> and/or <upper bound> properties are provided, it generate loop code. Containers do not accept incoming connector nor do they have outgoing connector. Package may be nested within another Package or Modules.
3. Procedure shape – a function representation shape. It may have zero or more incoming connectors but those connector's source can only be either Attribute or Parameter shapes. It may also has one outgoing connector, but the target must be a Result shape.
4. Attribute shape – a shape to declare a variable. Attribute may have zero or one incoming connector and the source must be either another attribute or Result shape. It may have zero or more outgoing connectors but the connecting target must be either another attribute shape, a parameter shape or a Procedure shape.
5. Parameter shape – a shape to represent attributes while passing information into functions. A parameter may have zero or exactly one incoming connector with the source from either an Attribute shape or a Result shape and it may have zero or one outgoing connector with the target must be a Procedure shape.

6. Result shape – it represent a returning value from a function call. As return value, it may have zero or one incoming connector with the source a Procedure shape. It may also have one or more outgoing connector with the targets either an Attribute shape or a Parameter shape.
7. Initiator shape – a start state indicator. Initiator may not have any incoming connector and may have one outgoing connector and the target must be a Procedure shape.

3.2.2.5 Attribute Properties

The Properties panel on the Side bar displays the selected shape's type and associating properties with their current assigned default values.



3.2.2.6 Script Code

The Code panel (also on the Side Bar) displays the generated BUGS code block based on the current selected shape or the entire diagram if no shape is selected. The code for the entire diagram is what will be sent to the server for execution or to be saved.

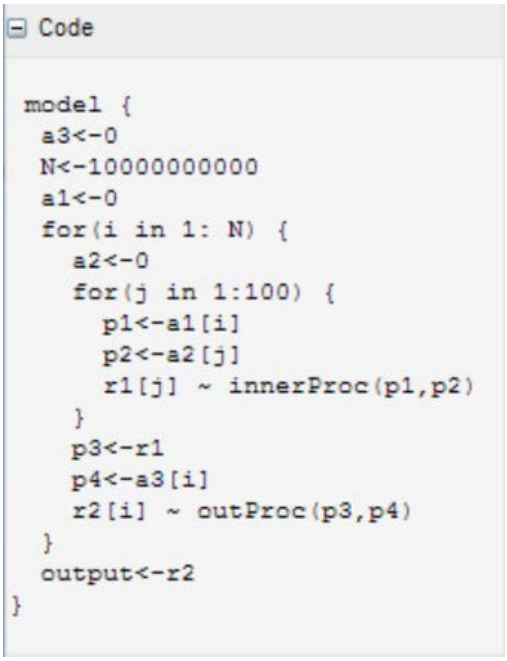


Figure 6: Script Code panel

4. FUTURE WORK

The application at this stage is a prototype version to prove the concept of implementing a Bayesian analysis engine in a cloud-based shared environment. In order to make the application fully functional and in a usable quality, there are some features and capabilities, both in the server and the client, which have yet to be implemented.

4.0 Server Application

All of the following features for the server application are necessary to make the application functional in a cloud-based framework.

1. Design database schema for user account management (account, code, libraries, results, etc.)
2. Identify and implement necessary exported functions as interface to the client app.
3. Create a module to interface to OpenBUGS runtime for scripts execution and update monitor purposes.
4. Create a data access layer module to allow the server application to query and store data to the database.

4.1 Client Application

Some features listed here are required; others are optional to make the client app functional. Specifically features 1 through 6 are required, but the rests are features that would make the application more efficient or just easier to use.

1. Support the full features of BUGS language needed for risk assessment.
2. Support external "Data" inclusion in order to import data structures into the Bayesian analysis.
3. Add a menu item [Run] and create progress view to monitor updates from server while processing script.
4. Create authentication user interfaces and management.
5. Upload the script library to the server.
6. Save/load script/diagram to and from server respectively.
7. Allow editing custom scripts and verification (i.e., error checking) of these scripts.
8. Support multiple custom library modules (currently it supports a single library module).
9. Script generation optimization.
10. Abstracting diagram layers. Viewing diagram at different level of complexity, i.e. zoom out shows only groups and container versus zoom in to show detail of each container group.
11. Allow to edit the selected shape's attributes value directly in the Properties panel on the Side Bar.
12. Allow editing of the generated code or add custom code on the Code panel.
13. Ability to draw a diagram from an existing script.

5. CONCLUSIONS

Once this application is fully developed, it provides analysts the capability to focus on models and submit them to be executed in a shared high performance computing environment. It will also provide a diagraming approach to Bayesian model creation without the steep learning curve of the underlying scripting language. The infrastructure of the framework to support this application is designed not only to support BUGS scripting language, but is also applicable to support other scripting languages if needed. For example, there are other generalized open source statistical software package, such as R, that with this diagramming tool could be made to support it as well.

6. REFERENCES

1. **Kelly, Dana and Smith, Curtis.** *Bayesian Inference for Probabilistic Risk Assessment: A Practitioner's Guidebook.* s.l. : Springer, 2011. ISBN 978-1-84996-186-8.

APPENDIX A

A.1 Use Case Scenarios

There are many tasks that started with the user initiating an element on the application, such as menus navigator, Tools bar and Side bar. Some tasks may be a simple click, while others required the user to click + hold and move the mouse to certain location then release (drag-and-drop). All menu items are single click actions. The Shapes pallet supports either single click or drag-and-drop. The Library panel supports drag-and-drop only. Still other panels do not response to user inputs.

Saving and loading uses HTML5 File API capability and it is effecting files on the user's local PC. In the case of script libraries, the local database is governed by the browser user's profile setting. The effect on the locally stored libraries is always volatile. It means the local database is controlled by the browser's history cache and if users were to switch to a different browser, log onto the PC as different user, or clear out browser history, the previous locally saved database, namely the libraries, will no longer be available. It is a built-in security feature of the browser itself. For this volatile effect, there are built-in functions to allow import and export the library stored in local database into a non-volatile location as a library file (*.libx) on the user's local hard drive.

Also in order to save file with a custom name, the browsers "always prompt to download" feature must be turned on.

NOTE: saving a file on user's local PC is saved onto the browser's default download path configurable through the browser's settings.

A.1.1 Save/Load

To save current diagram:

Click on menu [file] -> [Save] -> dialog open: Type name -> click [Save].

Press Ctrl + S -> if new diagram -> dialog open: Type name -> click [Save]

If existing diagram ->automatic update.

Open a saved diagram:

Click on menu [File] -> [Open…] -> file dialog open: locate a file [.xml] -> click [Open].

A.1.2 Save as library

Click on menu [File] -> [Save As Library…] -> dialog open: Type in name -> click [Save].

The new library appears in the Library panel list on the Side Bar. Note: the save to library will save the entire diagram as a library not just the selected shapes.

NOTE: The library is stored in the user's local PC using a local client database called IndexedDB. As mentioned above, this client database is volatile to browser history management.

A.1.3 Import/Export library

To import a saved library from file: importing a library from file will replace currently loaded ones.

Expand the Side Bar/Library panel then right click on a library name like “OpenBUGS Functions” [Library Name -> Import Library -> dialog open: select a file [.libx] -> [Open].

To export the current libraries to a file:

Expand the Side Bar/Library panel then right click on a library name like “OpenBUGS Functions” [Library name -> Export Library to File -> dialog open: Type in name -> click [Save].

A.2 Create diagram

To create diagram with shapes:

Creating diagram is done through shapes pallet on the Side Bar (Figure 7). Clicking a shape will create the shape and placing it at the upper left corner on the Drawing Surface. Drag-N-drop a shape onto a specific location on the Drawing Surface to create the shape at that location or within another container shape. Once the shape is created (Figure 8), it will prompt user for a name. Click [OK] to accept and [Cancel] to abort the shape. Noted that a shape created or move inside a container such as a Module shape or a Package shape, it cannot be moved outside of the container. This function limitation will be changed in the future.

Figure 7: select a shape out on the Shapes pallet

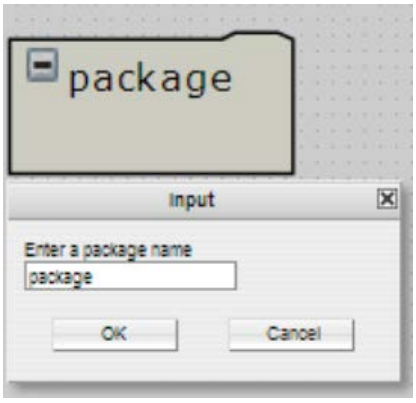


Figure 8: Create shape prompting for a name

On clicking [OK] the application checks the following:

1. Check to ensure that the new name is properly named; it checks that a name cannot start with numeric or any symbol characters except underline (_).
2. Make sure there isn't already another same name with same shape on the same scope as the one being created – duplication.
3. If 1 or 2 is violated, it prompts user for a new name and repeat check.

To connect shapes.

Click on a shape to select it. A yellow arrow (Figure 9) is shown to the right while the shape is selected. Some shapes do not allow connection and no edge-icon is shown. Click and hold on the edge-icon and move the mouse to another shape (Figure 10). Release the mouse and the connection is established (Figure 11), but before the connection is made permanent, it checks the following:

1. Check for any shapes involve the connection for rule violation.
2. If violation occurred, only the first violated rule message is shown and the connection is removed.
3. If no violation occurred, it makes the connection permanent.

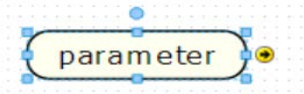


Figure 9: Selecting a shape to show the connection icon

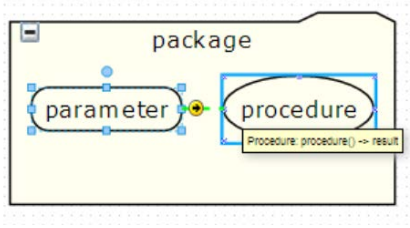


Figure 10: Connecting shapes together

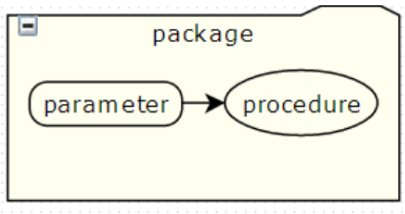


Figure 11: completing the connection

Container shapes may be nested and can be contain as a module or group as loop block in a Package shape as show in Figure 12. These group and block control code generation.

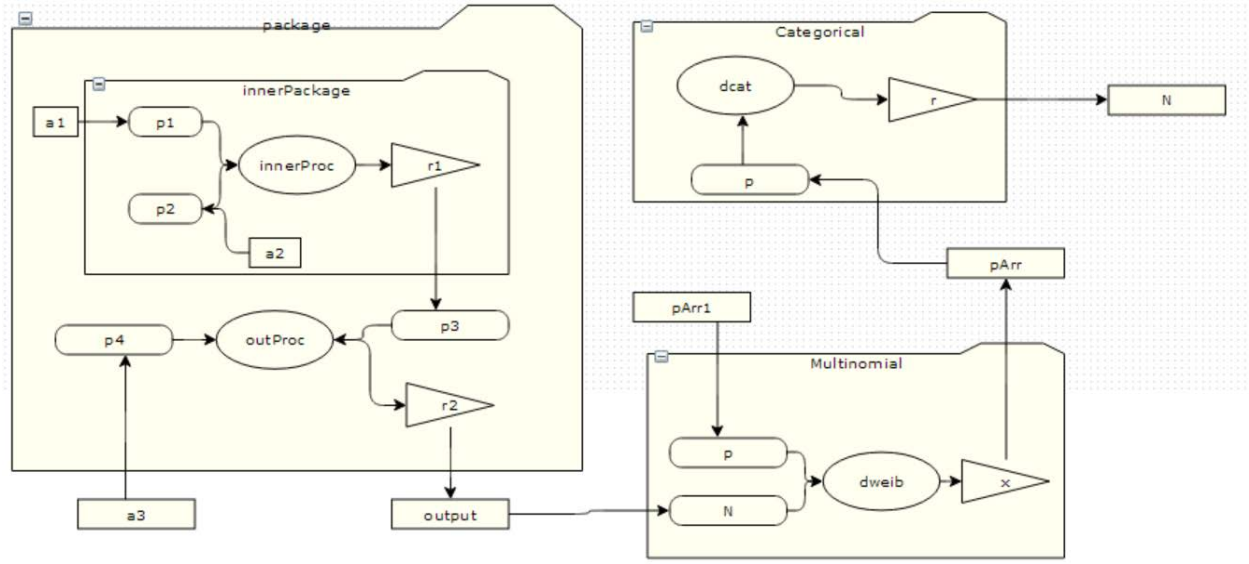


Figure 12: Example of nested and code block script diagram

NOTE: as each shape is added or removed and a connection is established or removed, the Code Panel is updated to show the resulting script generated.

A.3 Force script generation

Clicking on a shape or a connection will show the script for that specific shape. Deselect by clicking on the Drawing Surface results in generating the script for the entire diagram.

A.4 Execution

At any point, if there is code generated successfully, clicking on the [Run] on the menu or Tools Bar will submit the script to the server. It also open a monitor viewer window to show the progress status.