Using the DDG R Library

Emery Boose & Barbara Lerner

December 2013

1. What is the DDG R Library?

The DDG R library is a collection of R functions that allows a user to annotate (instrument) an R script so that provenance metadata in the form of a Data Derivation Graph (DDG) is created whenever the script executes. The DDG can then be viewed, queried, and stored using DDG Explorer.

2. Installation and Use

In its present form the library is implemented as a single R file (**ddg-library.r**). Download this file and save it to a directory on your computer. To create a DDG, the library must be executed before the main R script is executed. An easy way to do this is to include a **source** statement at the top of the main script and specify the path to the library. Alternatively, an R environment variable can be used to store the library location (see *Using DDG Explorer* for details) or the library can be executed manually before the main script is executed. Note that all variables and functions in the library begin with **ddg.** to avoid confusion with variable and function names in the main script.

The library needs to know the name and path of the R script itself (**ddg.r.script.path**) and the path for the DDG files that will be created (**ddg.path**). These values should be defined in the main script before the **source** statement that reads in the library. For example:

```
setwd("c:/data/r/ddg/quality-control-15min")
ddg.r.script.path <- paste(getwd(), "/quality-control-15min.r", sep="")
ddg.path <- paste(getwd(), "/ddg", sep="")
source("c:/data/r/lib/ddg-library.r")
```

In this Windows example, the working directory is set to *c:\data\r\ddg\quality-control-15min*, the R script (*quality-control-15min.r*) is located on the working directory, the DDG directory is a subdirectory called *ddg* of the working directory, and the name and path for the library is *c:\data\r\lib\ddg-library.r*. We recommend creating a DDG directory separate from the working directory for each R script.

The R user adds calls to the DDG library to collect provenance metadata, as described below. Once the script is properly annotated, provenance metadata will be collected as the R script executes.
This information is stored in memory and written to the DDG file (**ddg.txt**) on the DDG directory when the R script finishes. The DDG file contains information about the computing environment, the number of procedural steps, and the specifications for individual nodes and edges of the DDG. Step and data nodes are each numbered in sequence beginning with one. Simple data values (numbers, strings, URLs) are stored in the DDG itself. More complex data values (vectors, matrices, data frames, lists) are stored as pointers to files created on the DDG directory. Input and output files of the main script are stored as pointers to copies of those files created on the DDG directory. While the DDG file can be viewed using a text editor, its primary purpose is to support exchange of information and it should normally be viewed and queried using DDG Explorer.

3. How to Annotate an R Script

The DDG library contains low-level functions that will allow you to create individual nodes and edges of the DDG. However most users will prefer to use the higher-level functions described below. These functions streamline the annotation process by combining node and edge creation (wherever possible) into a single function.

Annotation of an R script for use with the library involves the following steps:

1. The R script may need to be modified to use functions (or blocks of contiguous code) to implement the key steps (calculations) to be captured in the DDG. The R programmer will need to give some thought to identifying these key steps. The script should be carefully tested to ensure that it is working properly before annotations are added.

2. Data nodes for inputs to the R script are created using the **ddg.data()**, **ddg.url()**, **ddg.snapshot()**, and **ddg.file()** functions, depending on the data type. The first three are used for simple data values, URL addresses, and complex data values (such as data frames), respectively, defined in the R script. The last is used for input files that already exist in the file system.

3. Operational step nodes that perform calculations are created using the **ddg-procedure()** function. This function automatically creates a control flow edge from the previous step. Input and output data are defined with reference to the operational step. In general it is assumed that input data nodes already exist but output data nodes do not. The former are inputs to the R script or created by a previous step. The latter are created by the current step.

4. A data flow edge from an input data node to the current operational step is created with the **ddg.data.in()** function.

5. Output data nodes are created with the **ddg.data.out()**, **ddg.url.out()**, **ddg.snapshot.out()**, **ddg.file.out()**, and **ddg.file.copy.out()** functions. These functions are used to create data nodes for single data values, URL addresses, complex data values (such as data frames) stored in memory, complex data values written to file by the main script, and complex data values in an existing file created by another process of the main script (respectively). A data flow edge is also created from the current operational step to the output data node.

6. Collapse and expand step nodes may be created using the **ddg.start()** and **ddg.finish()** functions. These functions create start and finish step nodes, respectively. A control flow edge is also created from the previous step. Note that these functions must be used in pairs and must be correctly nested.

7. The **ddg.save()** function should be used at the end of the main script. This function inserts attribute information and the number of procedure steps at the top of the DDG. It also creates the DDG file (**ddg.txt**) and nodes file (**nodes.txt**) on the DDG directory.

8. As an alternative to #7, the entire main program can be contained in a single function followed by a call to **ddg.run()** with the function name as parameter (see below for example). If no R errors are generated, the script will execute in the normal way and the DDG will be saved. If an R error is

generated, the DDG will be saved with a final operation node called "tryCatch" and final error node called "error.msg" whose value is the R error message.

4. Function Definitions

Note: Names used in function parameters must be enclosed in double quotation marks. File names used in data nodes of type "snapshot" or "file" should include the file extension.

The following functions may be used to create data nodes for input data to the R script. For data created or modified by the R script, see the next section.

- **ddg.data** (*dname, dvalue*). Creates a data node of type "Data" called *dname* with value *dvalue*. Use for single data values defined in R script. Example: ddg.data("number", 15)
- **ddg.url** (*dname, dvalue*). Creates a data node of type "URL" called *dname* with address *dvalue*. Use for URL addresses defined in R script. Example: ddg.url("hf", "http://harvardforest.fas.harvard.edu")
- **ddg.snapshot** (*dname, data*). Creates a data node of type "Snapshot" called *dname*. Writes contents of *data* to file *dname* (with numerical prefix) on DDG directory. Use for complex data values defined in R script. Example: ddg.snapshot("calibrated-data.csv", calibrated.data)
- **ddg.file** (*dname, dloc*). Creates a data node of type "File" called *dname*. Path to original file is optionally specified in *dloc* (otherwise current working directory is assumed). Copies input file called *dname* to new file called *dname* (with numerical prefix) on DDG directory. Use for existing input file. Example: ddg.file("archive-data.csv")

The following functions may be used to create an operational step and associated data nodes and edges. These functions assume that input data nodes already exist and that output data nodes must be created.

- **ddg.procedure** (*pname*). Creates a procedure node of type "Operation" called *pname*. Example: ddg.procedure("read.data")

- **ddg.data.in** (*pname, dname*). Creates a data flow edge from data node *dname* to procedure node *pname*. Data node type may be "Data", "URL", "Snapshot", or "File". Example: ddg.data.in("read.data", "archive-data.csv")

- **ddg.data.out** (*pname, dname, dvalue*). Creates a data node of type "Data" called *dname* with value *dvalue*. Creates a data flow edge from procedure node *pname* to data node *dname*. Use for simple data values. Example: ddg.data.out("calculate-square-root", "sqr-root", x)
- **ddg.url.out** (*pname, dname, dvalue*). Creates a data node of type "URL" called *dname* with address *dvalue*. Creates a data flow edge from procedure node *pname* to data node *dname*. Use for URL addresses. Example: ddg.url.out("get.url", "hf", "http://harvardforest.fas.harvard.edu")
- **ddg.snapshot.out** (*pname, dname, data*). Creates a data node of type "Snapshot" called *dname*. Creates a data flow edge from procedure node *pname* to data node *dname*. Writes contents of *data* to file *dname* (with numerical prefix) on DDG directory. Use for complex data values stored in memory. Example: ddg.snapshot.out("read.data", "data.file", xx)
- **ddg.file.out** (*pname, dname, data, dloc*). Creates a data node of type "File" called *dname*. Path to original file is optionally specified in *dloc* (otherwise current working directory is assumed). Creates

a data flow edge from procedure node *pname* to data node *dname*. Writes contents of *data* to file *dname* (with numerical prefix) on DDG directory. Use for complex data values written to file by the main script. Example: ddg.file.out("save.data", "final-data.csv", xx)

- **ddg.file.copy.out** (*pname, dname, dloc*). Creates a data node of type "File" called *dname*. Path to original file is optionally specified in *dloc* (otherwise current working directory is assumed). Creates a data flow edge from procedure node *pname* to data node *dname*. Copies existing output file called *dname* to new file called *dname* (with numerical prefix) on DDG directory. Use for output file already created by another process in the main script. Example: ddg.file.copy.out("plot.data", "plot.jpeg")

The following functions may be used to create collapse and expand nodes. These functions must be used in pairs and must be correctly nested.

- **ddg.start** (*pname*). Creates a procedure node of type "Start" called *pname*. Example: ddg.start("main")
- **ddg.finish** (*pname*). Creates a procedure node of type "Finish" called *pname*. Example: ddg.finish("main")

One of the following functions should be used at the end of the main program:

- **ddg.save** (). Inserts attribute information and the number of procedure steps at the top of the DDG. Writes the DDG and nodes table to the DDG directory. Example: ddg.save()
- **ddg.run** (*fname*). Executes the function *fname* which contains the main program. If an R error is generated, captures the error message and saves it in the DDG. Note: this function includes a call to ddg.save(). Example: ddg.run(main)

5. Troubleshooting

The most common annotation error is mistyping the name of a data or procedure node. This will result in an error message of the form "No data node found for …" or "No procedure node found for …" that will appear in the R command window.

The DDG library contains a debugging feature that may be helpful if your script generates other error messages upon execution. This feature is turned off by default. It may be turned on and off by entering **ddg.debug.on()** or **ddg.debug.off()** at the R command line after the library has been executed at least once.

When the debugging feature is turned on, the nodes and edges of the DDG will be listed in the R command window as the script executes and the DDG is created. The following abbreviations are used:

proc.node = procedure node
data.node = data node
CF, proc2proc = control flow edge
DF, proc2data, data2proc = data flow edge
d1 = data node number 1
p1 = procedure node number 1

Once your script executes without errors, try loading the DDG file (**ddg.txt**) into DDG Explorer. If DDG Explorer discovers an error in the DDG, an error message will be displayed in the main window.

The **ddg.run()** function described above may also be used to capture R error messages (arising from the main script or the annotations) and store them in the DDG.

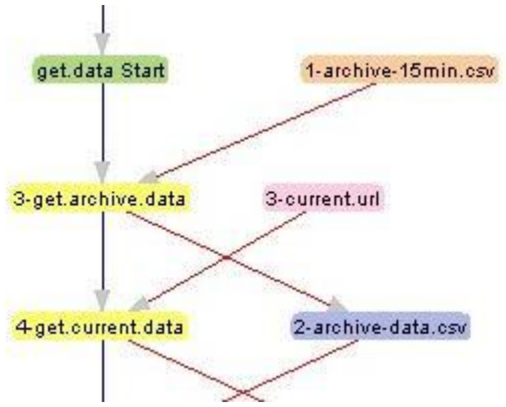
6. Examples

Simple operation (R function):

```
get.archive.data <- function() {
	archive.file <<- "archive-15min.csv"
	zz.col <- c("type","year","jul","hm","airt","rh","dewp","prec","slrr","parr","netr","bar","wspd",
	  "wres","wdir”,"wdev","gspd","s10t")
	zz <- read.csv(archive.file,col.names=zz.col,header=FALSE)

	ddg.procedure("get.archive.data")
	ddg.file(archive.file)
	ddg.data.in("get.archive.data",archive.file)
	ddg.snapshot.out("get.archive.data","archive-data.csv",zz)

	return(zz)
}
```

This R function reads an existing data file into a data frame, assigns column names, and returns the data frame. In the annotation, **ddg.procedure()** creates an operational step node for the R function, **ddg.file()** creates an input data node for an existing file, **ddg.data.in()** creates a data flow edge from the input data node to the step node, and **ddg.snapshot.out()** creates an output data node for the data frame as well as a data flow edge from the operational step node to the output data node. Note that the DDG function calls follow the computational steps but precede the return statement.

In the resulting DDG, the operational step is shown in yellow (get.archive.data), the input data file is shown in salmon (archive-15min.csv), and the internal data frame is shown in dark blue (archive-data.csv):

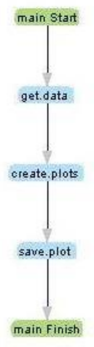


Expandable Nodes:

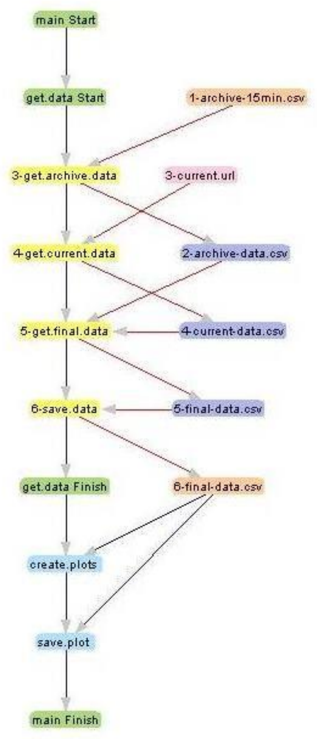
```
ddg.start("get.data")

archive.data <- get.archive.data()
current.data <- get.current.data()
final.data <- get.final.data(archive.data,current.data)
save.data("final-data.csv",final.data)

ddg.finish("get.data")
```

In this section of the main program, the functions **ddg.start("get.data")** and **ddg.finish("get.data")** are used to create collapsible start and end nodes that encompass four operational steps implemented as R functions: get.archive.data(), get.current.data(), get.final.data(), and save.data(). Note that the start and finish functions must be used in pairs and must be correctly nested.

In the resulting DDG, the get.data node (left) can be expanded into its corresponding start, finish, and intervening nodes (right):



Collapsed

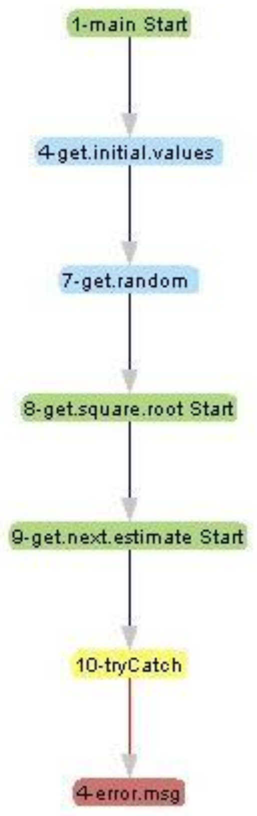


Expanded

Capturing R Error Messages:

```
main <- function() {
            ...
            foobar()  # intentional error
            ....
}

ddg.run(main)
```

Here the entire main program is enclosed in a single function (main) which is then called by the **ddg.run(**) function. The script terminated when an intentional error (an unknown function called "foobar") is encountered. The DDG is saved with final operation step "tryCatch" and final error node "error.msg" whose value is the R error message.



7. Technical Details

Syntax of DDG Text File:

```
<DDG> -> <Attributes>*<PinCounter> <Declaration>*
<Declaration> ->  <EdgeDecl> | <NodeDecl>
<EdgeDecl> -> <ControlFlowDecl> | <DataFlowDecl>
<ControlFlowDecl> -> <CF_TOKEN> <ProcedureNodeID><ProcedureNodeID>
<DataFlowDecl> -> <DF_TOKEN> <DataFlowEdgeDecl>
<DataFlowEdgeDecl> -> <DataNodeID><ProcedureNodeID> | <ProcedureNodeID><DataNodeID>
<NodeDecl> -> <DataNode> | <ProcedureNode>
<ProcedureNode> -> <ProcedureNodeType> <ProcedureNodeID> <NAME>";"
<ProcedureNodeType> -> “Operation” | “Start” | “Finish” | “Interm” | “Leaf” | “SimpleHandler” | “VStart” |
            “VFinish” | “VInterm”
<DataNode> -> <DataNodeType> <DataNodeID> <NAME> ["Value" "="<Value> ] ["Time" "=" <Timestamp> ]
            [“Location” “=” <FILENAME>] ";"
<DataNodeType> -> “Data” | “Exception” | "URL" | "File" | “Snapshot”
<Value> -> <URL> | <FILENAME> | <STRING>
<Timestamp> -> <YEAR>"-"<MONTH>"-"<DATE>["T"<HOUR>":"<MINUTE>[":"<SECOND> ["."<FRACTIONAL>]]]
<Attributes> -><NAME>["="]<AttrValue>
<AttrValue> -> <STRING>
<PinCounter> -> <INT>
<DataNodeID> -> ‘d’ <INT>
<ProcedureNodeID> -> ‘p’ <INT>
<CF_TOKEN> -> “CF”
<DF_TOKEN> -> “DF”
```