Using RDataTracker

Emery Boose & Barbara Lerner

May 2014

1. What is RDataTracker?

RDataTracker is a library of R functions that can be used to annotate (instrument) an R script in order to collect data provenance in the form of a Data Derivation Graph (DDG) as the script executes. It can also be used to collect data provenance during R console sessions. RDataTracker saves the DDG as a text file (ddg.txt), with ancillary files stored in a special DDG directory. The DDG can then be viewed, stored, and queried using a separate tool, DDG Explorer.

2. What is a DDG?

A Data Derivation Graph (DDG) is a mathematical graph that captures the history of a data analysis. The DDG consists of nodes and edges. In DDG Explorer, nodes are shown as ovals and edges are shown as arrows (see below for examples). Different colors (explained in the legend for DDG Explorer) are used to indicate different types of nodes and edges.

There are two major types of nodes—procedural nodes and data nodes—and two major types of edges—control flow edges and data flow edges. Control flow edges indicate how control passes from one procedural node to another procedural node as the script executes. Data flow edges indicate how input data pass from a data node to a procedural node or how output data pass from a procedural node to a data node.

Procedural nodes include Operational, Collapsible, Expandable, Checkpoint, and Restore nodes. Operational nodes perform an operation. Collapsible nodes and expandable nodes provide a level of abstraction by allowing a section of the DDG to be expanded or collapsed. Checkpoint nodes indicate creation of a checkpoint. Restore nodes indicate that a previous checkpoint was restored.

Data nodes include Data, File, Snapshot, URL, and Exception nodes. Data nodes are used for simple values. File nodes are used for files that are inputs to the R script or created by the R script. Snapshot nodes are used for complex data values such as data frames. URL nodes are used for URL addresses. Exception nodes are used for error messages. The values of Data, URL, and Exception nodes are stored in the DDG text file. The values of File and Snapshot nodes are stored as files in the DDG directory.

For more details on DDGs and how to view, store, and query them, please see the DDG Explorer documentation (UsingDDGExplorer.pdf).

3. Installation and Use

The following instructions assume you are using RStudio. Example scripts are shown in blue. See section 9 for DDG examples

RDataTracker is distributed as an R package. Note that R packages must be *installed* to your computer (normally just once) and then *loaded* for use in a particular session.

To install RDataTracker, copy or download the package file to your computer, open RStudio, and use the Tools / Install Packages option to install from a Package Archive File (alternatively you can use the **install.packages** command at the R prompt). Once the library has been installed, select Packages / RDataTracker to see a list of help pages for the various functions. Note that all functions begin with **ddg.** to avoid confusion with function names in the main script.

To load RDataTracker, use the **library(RDataTracker)** command at the R prompt or at the top of your script. Alternatively you can click on the checkbox for RDataTracker in the list of packages in RStudio. The library needs to know the path and name of the R script itself and the path for the DDG files that will be created (see below for how to provide this information). We recommend creating a DDG directory separate from the working directory for each R script.

A DDG is created by adding function calls to RDataTracker as described below. Once the script is properly annotated, data provenance 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 (e.g. numbers) are stored in the DDG itself. More complex data values (e.g. data frames) 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.

The library functions are introduced below. For more details, please see the help pages under Packages / RDataTracker in RStudio.

4. Minimal Annotation

A simple DDG can be created with minimal annotation of an R script, as illustrated below:

```
library(RDataTracker)
setwd("c:/data/r/example")
r.script.path <- paste(getwd(), "test.r", sep="")
ddgdir <- paste(getwd(), "/ddg", sep="")
ddg.init(r.script.path, ddgdir, enable.console=TRUE)

{main script}

ddg.save()
```

Here the first line loads the RDataTracker package. The second line sets the working directory. The third line specifies the path and name of the R script. The fourth line specifies the directory where the DDG files will be stored. The fifth line initiates creation of a DDG. The sixth line saves the DDG.

When **enable.console** is set to TRUE, the library captures assignment statements as Operational nodes and associated data values as Data nodes. This approach is useful for getting a quick view of a data analysis.

5. Console Sessions

A similar approach can be used to capture console sessions in R.

```
>library(RDataTracker)
>ddg.init(ddgdir="c:/data/r/console/ddg", enable.console=TRUE)

{console commands}

>ddg.save()
```

Here the first line loads the RDataTracker package. The second line initiates creation of a DDG and specifies where the DDG files will be stored. The third line saves the DDG. Note that **ddg.save** may be called more than once during a console session to add to the current DDG, while calling **ddg.init** again replaces the current DDG with a new DDG.

6. Annotating an R Script

A more detailed DDG may be created by adding more annotations to the original script. Here **enable.console** is set to FALSE in **ddg.init** and only the annotated portions of the script appear in the DDG.

The basic strategy for annotating a script is described below. See the help files for RDataTracker for additional details on individual functions.

1. Data nodes for input data to the original script are created using **ddg.data**, **ddg.file**, **ddg.snapshot**, **ddg.url**, or **ddg.exception**, depending on the data type. If only the variable name is supplied, the library will look up its value in the current environment.

```
ddg.data(x)
```

2. Operational nodes are created using **ddg.procedure**. If **ddg.procedure** is called from inside an R function and the name of the node is omitted, the library will use the name of the function.

```
ddg.procedure()
```

3. Input data nodes are assumed to exist when **ddg.procedure** is called. Data flow edges to one or more existing input data nodes can be created using a list of values and the **ins** parameter of **ddg.procedure**.

```
ddg.procedure(ins=list("x1", "x2"))
```

If **ddg.procedure** is called from inside an R function and if **lookup.ins** is set to TRUE, the library will create a data flow edge to each argument passed to the function, if a corresponding data node exists.

```
ddg.procedure(lookup.ins=TRUE)
```

A data flow edge to an existing data node may also be created using **ddg.data.in**. The name of operational node may be omitted if **ddg.data.in** is called from within an R function.

```
ddg.procedure()
ddg.data.in(x)
```

4. One or more output data nodes and corresponding data flow edges may be created using a list of values and the **outs.data**, **outs.file**, **outs.snapshot**, **outs.url**, or **outs.exception** parameters of **ddg.procedure**, depending on the data type.

```
ddg.procedure(outs.data=list("z1", z2"))
```

A single output data node and corresponding data flow edge may also be created using **ddg.data.out**, **ddg.file.out**, **ddg.snapshot.out**, **ddg.url.out**, or **ddg.exception.out**, depending on the data type. The name of the operational node may be omitted if **ddg.data.out**, etc is called from within an R function.

```
ddg.procedure()
ddg.data.out(x)
```

5. Expandable and collapsible nodes, implemented through calls to **ddg.start** and **ddg.finish**, may be used to create levels of abstraction in the DDG. These functions must be correctly nested with matching arguments.

```
ddg.start("calculate.square.root")

{intervening R code}

ddg.finish("calculate.square.root")
```

Clicking on the start or finish node in DDG Explorer will collapse the intervening nodes to a single node, while clicking on a single collapsed node will expand to reveal the intervening nodes.

6. A DDG is created and saved using the **ddg.init** and **ddg.save** functions. The annotation is essentially the same as for the minimal DDG described above, except that **enable.console** is set to FALSE.

```
library(RDataTracker)
setwd("c:/data/r/example")
r.script.path <- paste(getwd(), "test.r", sep="")
ddgdir <- paste(getwd(), "/ddg", sep="")
ddg.init(r.script.path, ddgdir, enable.console=FALSE)

{main script}

ddg.save()
```

Alternatively, **ddg.run** can be used instead of **ddg.init** and **ddg.save** if the main program is implanted as a single function. One advantage to this approach is that if an R error occurs during execution of the function, the error will be capture in an Exception node in the DDG.

```
library(RDataTracker)
setwd("c:/data/r/example")
r.script.path <- paste(getwd(), "test.r", sep="")
ddgdir <- paste(getwd(), "/ddg", sep="")

{R script functions}

main <- function() {

    {R script main program}

}
ddg.run(main, r.script.path, ddgdir)
```

7. Checkpoint and Restore

The library contains functions that can be used to save and restore the R state and file system state, allowing a user to return to the environment present at an earlier point in the data analysis. **ddg.checkpoint** adds a checkpoint node and a snapshot node and returns the full path to the file containing the saved state. **ddg.restore** adds a restore node with an input edge from the snapshot node for the saved checkpoint file. A given checkpoint can be restored more than once.

```
checkpoint1 <- ddg.checkpoint()

{intervening R code}

restore(checkpoint1)
```

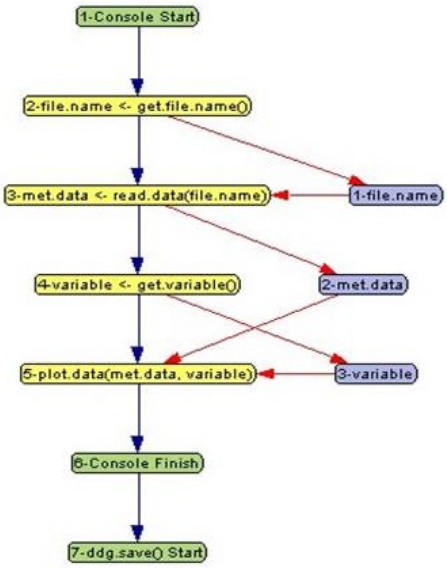
8. Troubleshooting

Annotation errors are generally captured by the library and stored as error nodes in the DDG. The same is true for error messages from the R interpreter if the **ddg.run** function is used as described above. These features can be useful for troubleshooting the original script and the associated annotations.

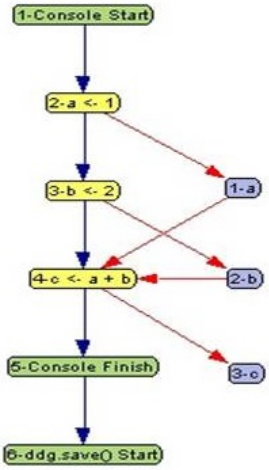
The **ddg.debug.on** and **ddd.debug.off** functions may be used (in a script or at the console) to turn debugging on and off. Debugging is off by default. When debugging is turned on, details related to creation of the DDG are displayed in the console as the script executes.

The contents of the current DDG directory (if different from the working directory) may be deleted by calling the **ddg.flush.ddg** function at the R prompt.

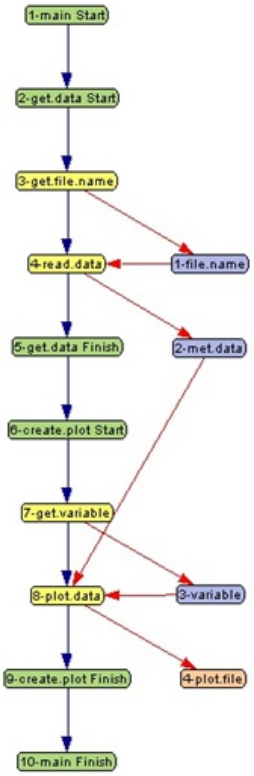
9. DDG Examples



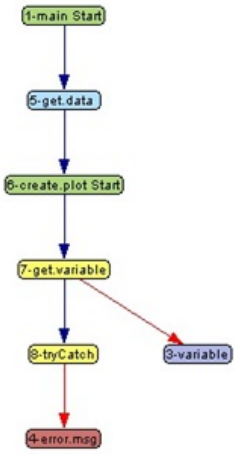
Minimal Annotation



Console Session



Detailed Annotation



Error Message

10.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”