Using RDataTracker

Barbara Lerner
Department of Computer Science
Mount Holyoke College
50 College St, South Hadley, MA

Emery Boose
Harvard Forest
Harvard University
322 North Main Street, Petersham, MA

August 3, 2014

Contents

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 (see Using DDG Explorer for more information on this tool).

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, Binding, 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. Binding nodes indicate the process of binding input parameters of a function to its internal representation. 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, as well as other complex data, such as graphical outputs. 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 demarcated in a code box. See Section 6 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). The library depends on **gtools**, so make sure this is installed before attempting installation of RDataTracker. 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)** or the **require(...)** 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 specifying 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.

3.1 Minimal Annotations

This section describes different approaches to directly annotating R commands for provenance collection, either taken from an existing R script file or input directly into the R console.

3.1.1 Short Scripts and Console Sessions

For short scripts or console sessions not exceeding *R_HISTSIZE* (default of 512 lines), a simple DDG can be created with minimal additional commands, as illustrated below:

```
1 # load the library
2 library(RDataTracker)
```

```
3
4  # Set the working directory and obtain the path of the script and the ddg directory
5  setwd("c:/data/r/example")
6  r.script.path <- paste(getwd(), "test.r", sep="")
7  ddgdir <- paste(getwd(), "/ddg", sep="")
8
9  # Initialize the data collection
10 ddg.init(r.script.path, ddgdir)
11
12 # main script or console commands
13 {short script or set of console commands}
14
15 # Save the collected data to disk, and delete the DDG in memory (quit=TRUE parameter)
16 ddg.save(quit=TRUE)
```

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. This is followed by the normal script or console session commands. The final line saves the DDG.

When **enable.console**, an optional parameter to **ddg.init** 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 and is therefore the default. However, the above approach to annotations is limited to short scripts (usually below 512 lines, as specified by *R_HISTSIZE*).

3.1.2 Longer Scripts and Console Sessions

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

```
1  # load the library
2  library(RDataTracker)
3
4  # Set the working directory and obtain the path of the script and the ddg directory
5  setwd("c:/data/r/example")
6  r.script.path <- paste(getwd(), "test.r", sep="")
7  ddgdir <- paste(getwd(), "/ddg", sep="")
8
9  # Initialize the data collection
10 ddg.init(r.script.path, ddgdir)
11
12 # console or script commands go here (approx 512 lines by default, no more)
13 {script or console commands}
14
15 # captures previous set of commands and stores in memory
16 ddg.grabhistory()
17
18 # additional console or script commands
19 {console or script commands}
20
21 # captures previous history, and writes out current ddg to disk
22 ddg.save()
23
24 # more commands
25 {console or script commands}
26
```

```
# Save the additionally collected data to disk, and mark the DDG as completed
ddg.save(quit=TRUE)
```

Here the first line loads the RDataTracker package. The following three lines of code initiate the creation of a DDG and specify where the DDG files will be stored.

The user is then free to input his or her commands, either through the console or from a script. Note that data capture occurs only during calls to functions from the RDataTracker package. This limits the ability to capture data since the state of the environment is unknown between calls to RDataTracker functions. For example, if a variable is ever reused, only the value at the time of data capture is available. This problem can be mitigated with multiple calls to RDataTracker functions. However, it is recommended to see either Section 3.3 or Advanced Annotations 8 as those approaches, while limited only to scripts, avoid this issue entirely.

If advanced annotations are avoided, two functions exist for the purpose of capturing data. The **ddg.grabhistory** may be called more than once during a console session or script execution and will add to the current DDG without writing the output to disk. **ddg.save** without any parameters works similarly, but writes out all data to disk.

A call to **ddg.save** with the *quit* parameter as TRUE will write out all data to disk and clear out the current DDG from memory. A call to **ddg.init** will replace the current DDG with a new DDG without writing or capturing any additional data for the previous DDG.

3.2 Advanced Annotations

A more defined DDG may be created by adding more annotations to the original script or console commands. Here **enable.console** can be set to FALSE in **ddg.init** and only the annotated portions of the script appear in the DDG. However, it is still recommended to leave enable.console set to TRUE and simply annotate functions, as this will allow for automatic provenance capture for the top level as well as detailed provenance of function calls.

The basic strategy for advanced annotations 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.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.url**, or **outs.exception**, **outs.graphic** parameters of **ddg.procedure**, depending on the data type.

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

   A single output data node and corresponding data flow edge may also be created using **ddg.data.out**, **ddg.file.out**, **ddg.graphic.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. Return values from functions can also be captured, and appropriate binding nodes created, using **ddg.return**.

```
# function definition, replace return with ddg.return
f <- function() {
  {R Function Code}
  ddg.return(x)
}
```

The above would create binding nodes with any variable to which the result of f is assigned.

6. 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
{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.

7. The automatic capture of data, also known as enabling **console mode**, can be turned on and off dynamically using **ddg.console.on** and **ddg.console.off**. Commands captured during an enabled console are encapsulated in a collasible Console node.

```

# turn on the console mode (if alread on, nothing occurs)
ddg.console.on()

# captured R code
{captured R code}

# turn it off
ddg.console.off()

# no automatic capturing occurs here
{R code}

# Nothing occurs since console is already off
ddg.console.off()
```

From the above, a single collapsible console node is created including the captured R code.

8. DDGs generated from sourced files can be nested within the larger DDG of the script or console commands if any calls to **source** are replaced by calls to **ddg.source**. Both commands take similar optional parameters.

```
# source a script
ddg.source("script.r")
```

The above would create a collapsible "script.r" node. Within this node, a DDG that would have normally been generated from copying and pasting the code from "script.r" into the current script. By default, calls from within the sourced script to the library are ignored. For more detail, see the library documentation.

Furthermore, note that any script sourced in this way avoid the issue of data capture discussed in section 3.1.2, as we always have access to the scripts environment.

9. 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** can be set to FALSE.

```
# load the library
library(RDataTracker)

# Set the working directory and obtain the path of the script and the ddg directory
setwd("c:/data/r/example")
r.script.path <- paste(getwd(), "test.r", sep="")
ddgdir <- paste(getwd(), "/ddg", sep="")

# Initialize the data collection
ddg.init(r.script.path, ddgdir, enable.console=FALSE)

# main script or annotated console commands
{short script or set of console commands, annotated}

# Save the collected data to disk, and delete the DDG in memory (quit=TRUE parameter)
ddg.save(quit=TRUE)
```

Alternatively, **ddg.run** can be used instead of **ddg.init** and **ddg.save** if the main program is implemented 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 captured in an Exception node in the DDG.

```
# load the library
library(RDataTracker)

# R script functions
{R script functions}

main <- function {
  # R script main program
  {R script main program}
}

# Set the working directory and obtain the path of the script and the ddg directory
setwd("c:/data/r/example")
r.script.path <- paste(getwd(), "test.r", sep="")
ddgdir <- paste(getwd(), "/ddg", sep="")

# Run the main function and collect data
ddg.run(r.script.path, ddgdir,main, enable.console=FALSE)
```

3.3 Data Provenance from Scripts

If a clean, stand-alone script already exists for the process from which provenance needs to be collected, a final option exists. The advantages of this option are numerous. The approach is:

1. Compatible. This approach can be used in conjunction with the advanced annotations described above, as well as with the minimal annotations.
2. Simple. The process of collecting data provenance can be done with a single command, no annotations required.
3. Minimal. Similar to the **ddg.run** example, calls to **ddg.init** and **ddg.save** are unnecessarsy.
4. Correct. The problem of data collection without calls to the library is solved.
5. Fast.

The script must be either clean (without annotations) or annotated without the use of **ddg.run**, as this will could to issues) Then running the following commands from the console (or an R script) will create a DDG for the script..

```
# Set the working directory (or provide full path to script)
setwd("c:/data/r/example")

# Execute the script and collect data on it
ddg.run("script.r")
```

This executes the script and creates a DDG. By default, calls to **ddg.run** and **ddg.init** in "script.r" are ignored.

3.4 Checkpoint and Restore

Included with the library is a file **DDGCheckpoint.R** which 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.

```
# source the file, assuming it's in the working directory
source("DDGExplorer.R")

# create checkpoint
checkpoint1 <- ddg.checkpoint()

# intervening R code
{intervening R code}

# restore the checkpoint
restore(checkpoint1)
```

Note that in order to use these functions, DDGCheckpoint.R must be sourced. It is **not** included in the RDataTracker library.

4 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.

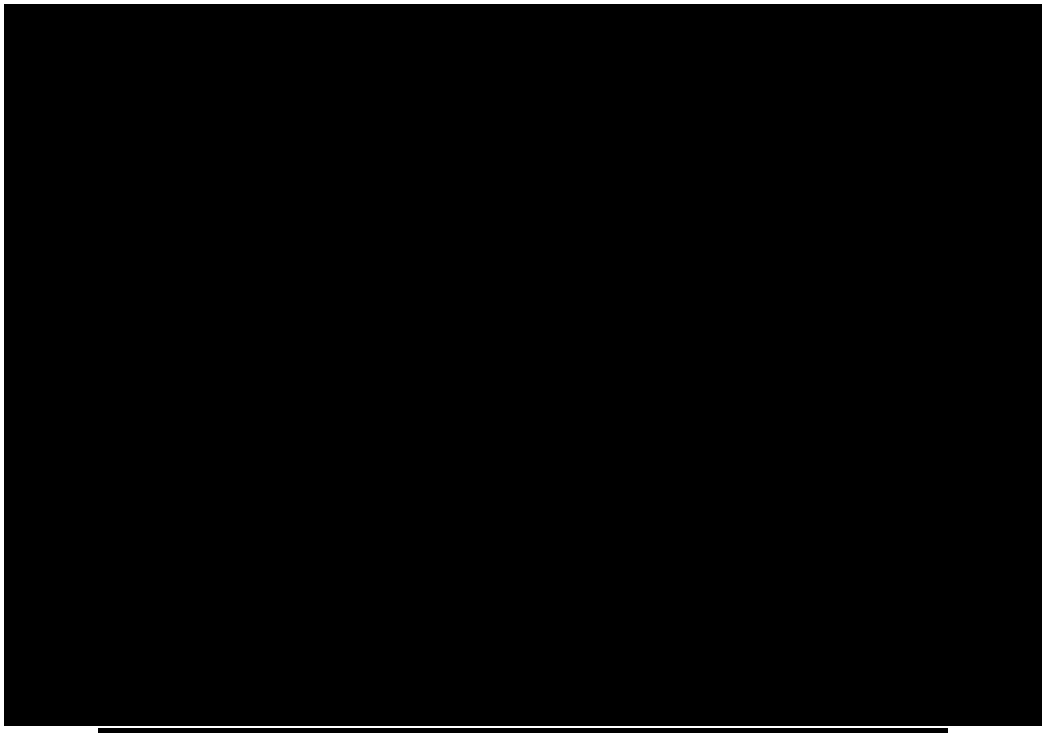
The contents of the current DDG directory (if different from the working directory) may be deleted by calling the **ddg.flush.ddg** function at the R prompt.

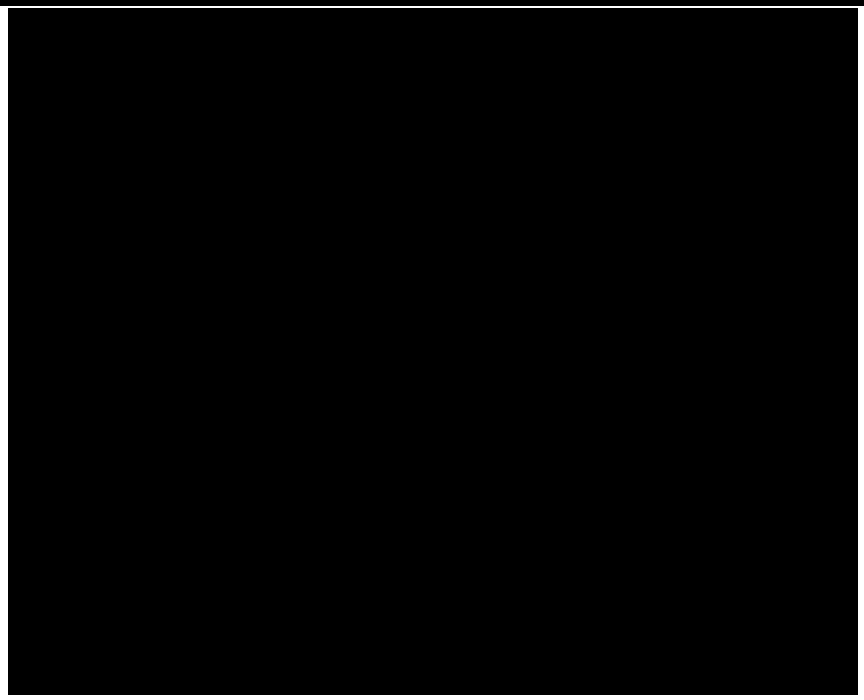
5 Acknowledgements

This material is based upon work supported by the National Science Foundation under Awards No. CCR-0205575, CCR-0427071, and IIS-0705772, the National Science Foundation REU grants DBI-0452254 and DBI-1003938, the Mount Holyoke Center for the Environment Summer Leadership Fellowships, and the Charles Bullard Fellowship in Forest Research at the Harvard Forest. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, Mount Holyoke College or Harvard University.

Numerous students have been involved in the research and tool development through the REU program at Harvard Forest. They are Shay Addams (2013 REU), Vasco Carinhas (2013 REU), Xiang Zhao (University of Massachusetts, Amherst), Luis Antonio Perez (2014 REU), and Nicole Hoffler (2014 REU).

6 DDG Examples





7 Technical Details

7.1 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> ["Value" "="<Value> ] ["Time
    " "=" <Timestamp> ][";"]

<ProcedureNodeType> -> "Operation" | "Start" | "Finish" | "Binding" | "Checkpoint" | "
    Restore"

<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"
```