Automatic Bug Detection in R

Luís Miguel da Conceição Rodrigues
Msc Computer Science and Engineering
Instituto Superior Técnico, Lisbon
luis.conceicao.rodrigues@tecnico.ulisboa.pt

ABSTRACT

The debugging capabilities available for R (through its native mechanisms or third-party applications) have not seen many advances in the past decades meaning that, the debug paradigm remains the same: define a desired stopping point, analyze each instruction to ensure its correctness and iterate through them until the problem is found.

The *Automatic Bug Detection in R* implements a non-linear paradigm (timewise): timeless debugging. It provides its users an execution graph based on the data collected during execution, allowing them to analyze each executed instruction at any time. This means that, to analyze the n^{th} instruction, executing the n^{th}-1 instructions is not required because their data was already stored in the first run.

Conceptually, this type of approach has inherent overheads which means that the overall performance will downgrade due to the extra instructions. The ABD_tool also succeeds at this aspect, by providing its complete functionalities with considerably low/medium overheads.

Keywords

R; Debugging; Timeless debugging; Dynamic debugging.

INTRODUCTION

In the United States of America, around 1.25 trillion dollars are spent in all the software development phases combined [1]. Part of that, 156 billion, is used to debug the developed software, which directly translates into 49.9% of the time spent to deliver the product to production. The time consumed in the debugging process is considerably high and this time expenditure can be a result of multiple factors, such as slow language debugger, inadequate tooling, lack of code structure and documentation, etc.

R [2] [3] [4], is one of those programming languages with inadequate tooling due to the fact that, while it is growing, its adoption is not high enough: not many users adopt it and the ones that do, work in areas where the language is not considered the standard, resulting in a lower interest when developing new tools for it. This can be sustained by a study [5] made with machine learning and data science as its main context. It places R in second place (with a noticeable margin) for both of the following questions:

- Programming language used most often
- Programming language recommended by data professionals

Since debugging has a major impact in the software engineering process and the available tools for R rely on wrappers to the commonly known debugging techniques, they can be improved. Instead of using the common approach, what if the programmer had the capability to use a tool that did not require even a second run of the program?

In case that that hypothetical tool was running, the time savings would be immense and with an immediate impact giving the programmer more time to do something else. This can be achieved because the data collected during the program runtime, would provide to the programmer an extensive context of each executed instruction. This data would then be transformed into information to give the programmer the ability to, at any desired instruction, understand why the outputs failed to meet the expected ones. This hypothetical tool is a Timeless Debugger [6] [7].

The *Automatic bug detection in R's* (ABD_tool) implements the paradigm of a timeless debugger. This efficiency improvement in debugging is one of the ABD_tool's main goals because the programmer will have the capability to execute his/her code once and analyze every instruction of it without needing to reconstruct the program context in order to analyze an object value in the previous instruction, for example. To analyze the previous instruction, running the prior n-1 instructions is not required and the previous instruction related data can be accessed and visualized instantaneously. With the ABD_tool, R programmers can visualize branches with statement granularity and, in else if clauses, verify what prior branch conditions failed. Programmers also have the possibility to visualize the data precedence between objects and their usages, the warning messages and thrown errors, function calls arguments mapping and much more.

ARCHITECTURE

Overview

The general architecture consists of two main parts (Figure 4): the ABD_tool's ecosystems and the changes made to the R interpreter.

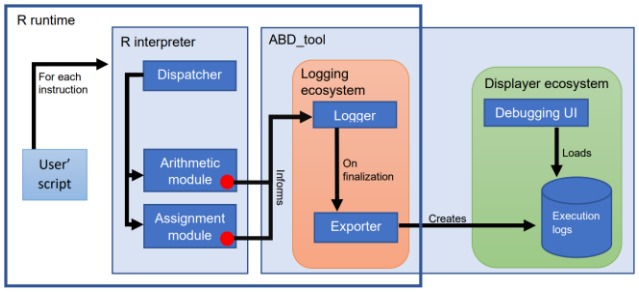


Figure 1 - General architecture overview

The ABD_tool must be notified whenever an object is modified, a function is called, etc., and be provided with all the data relative to those operations. In abstract, ABD_tool needs to be informed by the R interpreter (Figure 1, "Informs" label) whenever supported features instructions are executed, combined with the data relative to their execution, in order to export that data (Figure 1, "On Finalization" and "Creates" label) so that, in a posterior phase, the execution state in which those instructions were executed can be reconstructed (Figure 1, "Loads" label). The R interpreter was modified with calls to the ABD_tool, which will be called signals (represented by Figure 1's red dots).

The ABD_tool inner workings can be "visualized" with a higher-level example:

- The R interpreter process the instructions, reaches the modification and sends the signal (Figure 2, point a).
- The ABD_tool receives the signal and verifies if its context corresponds to the script current execution environment (Figure 2, point b):

  If it does not match - The signal is dropped

  If it matches – the signal is sent to the respective component (Figure 2, point c).
- If the stop command is issued or an error is signaled:

  The ABD_tool base component will verify the user-defined settings (Figure 2, point d), retrieve the saving path and persist the in-memory structures into files in the defined path (Figure 2, point e).

  If the user settings are set to launch the displayer after the execution ends, the ABD_tool will launch the displayer (Figure 2, point f).

Note: When the displayer is launched, the exported files are parsed into memory and the visualization is constructed (Figure 2, point g).

Generating signals in the R interpreter

The R interpreter consists of multiple modules that perform a wide variety of tasks. The signals include several arguments. Some are generic (**G**) (and shared by all) and others are specific (**S**) to the instruction:

Call expression (G) – the call expression before its evaluation by the interpreter (a LISP list).
Left-hand side (LHS) (G) – the left part of an instruction. In an assignment, it is the modified/created object. The value sent in the signal is post evaluation.
Right-hand side (RHS) (G) – the right part of an instruction and it can assume another expression or a value. The value sent in the signal is post evaluation.

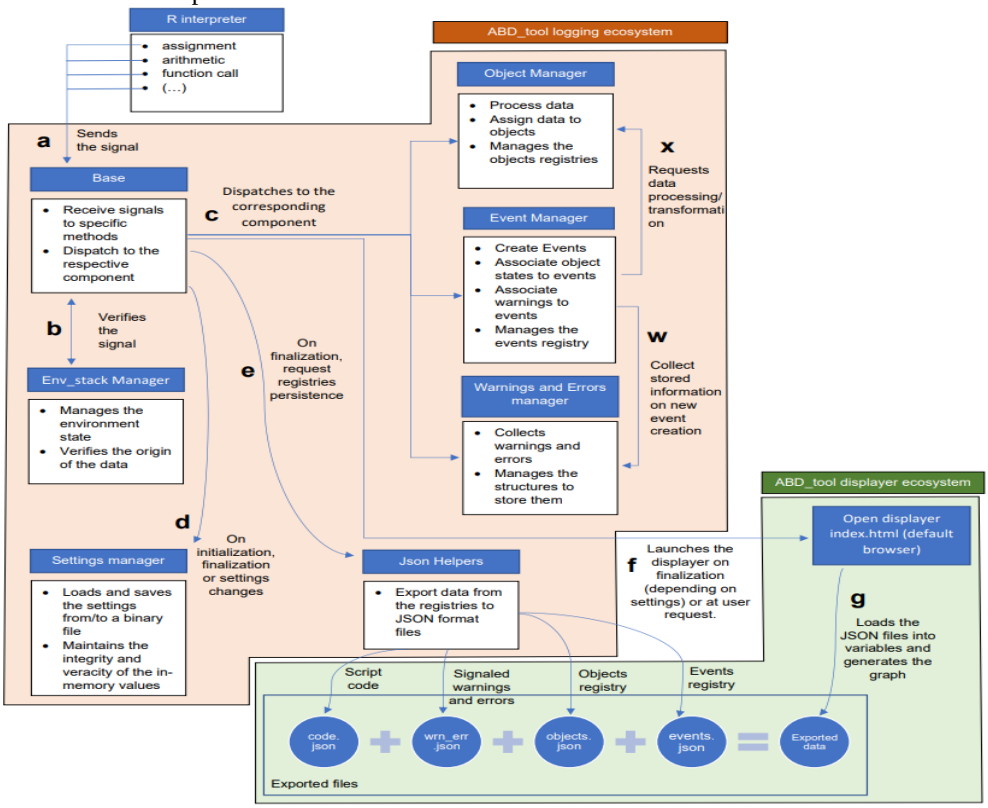


Figure 2 - Architecture overview

Environment (RHO) (G) – Represents the execution context in which the signal was sent, it allows the verification of the instruction and the decision regarding the signal processing:

RHO matches the user code current execution environment: Process the signal.

RHO does not match: The signal is discarded.

Arguments, Results, etc. (S) – The specific arguments are instruction specific, and so, the signal includes them when created.

The placement of this signaling is very important because if, for some reason, the expression evaluation generates an error, and the ABD_tool is notified before that evaluation, there is the possibility that the tool processes erroneous data. The majority of the signals are sent after the instructions are successfully processed and applied by R to the corresponding variables/environment avoiding the persistence of erroneous data and the display of misinformation to the user by the UI.

ABD_tool logging ecosystem

The ABD_tool, in order to achieve the intended information disclosure to its users, needs to convert the signals into data that, when exported, can be understood and processable by the displayer component. These signals, when reaching the ABD_tool "endpoint", will be processed and transformed into well-known and well-defined structures named events. Only one component is responsible for the literal creation of events (Event manager) but it needs to communicate with other components, depending on the event it is creating (Figure 2 point x and w), to request data (already processed and stored) or to process new data (transforming it to known structures). The ABD_tool components are:

Base - The base component contains all the methods that are intended to be called from outside the tool, giving the ABD_tool an API-like structure.

Object Manager - Processes data, associate's data to objects and creates the structures needed for that data to be associated to the objects. It also manages the objects registries: the common and the code flow registries for data objects and code flow objects (functions), respectively.

Event Manager - Creates new events and associate's objects' states to them (in case there is any). The event manager also manages an events registry that stores all the created events in a list.

Events - Contains the definitions/structures for the events.

Warnings and errors manager – Manages warnings and errors that were signaled to the ABD_tool.

Environment stack manager - Prevents the ABD_tool to store undesired data by verifying if the context of the calls made to the API correspond to the current script context.

Settings manager – Manages the user-defined settings (tool settings, output paths validity and settings integrity).

JSON helpers - Exports the collected data to files that are then used by the UI to display the information.

ABD_tool displayer ecosystem

The displayer is one of the most crucial ABD_tool' components but, the whole functionality of the tool is not dependent on it. Although the displayer is a replaceable component, it is one of the most important pieces of the whole architecture because it translates the recorded data into accessible, understandable and explorable information to the programmer, through the UI, in order to achieve an easier, more efficient and effective debugging.

Graph generation procedure

Using the events file created from the processing phase, the displayer generates a graph based on the environments that it encounters. In short, when the graph is generated, it will constitute a directional and interactable graph where each node represents an executed environment and is identifiable by its memory address. Its contents have the instructions that were executed during the environment life cycle, associated to the effective script line number.

Displayer capabilities

The displayer (Figure 3) is what will give value to the

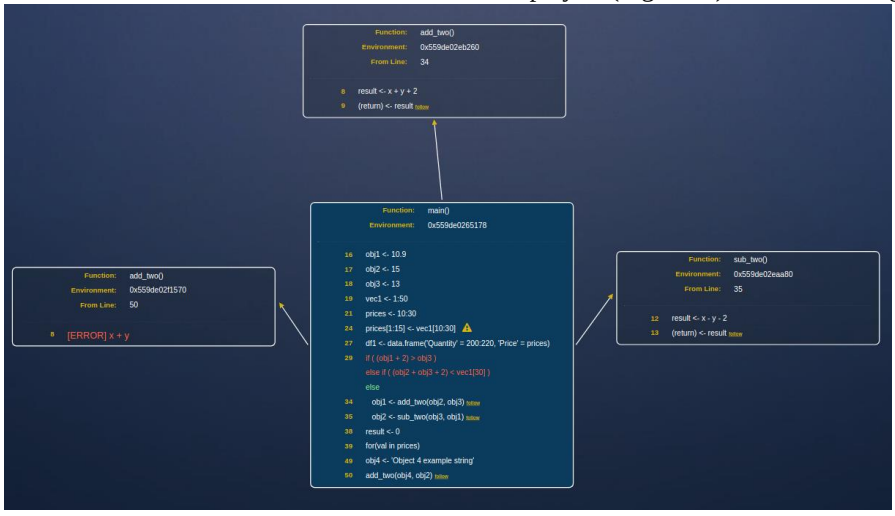


Figure 3 - Example graph

timeless debug paradigm implemented by the ABD_tool, making its functionalities the most important aspect of this work mainly because they are what the user will effectively experience during the tool's usage. Its functionalities are:

Function calls - By environment, directional graph visualization

- Clickable labels to center in the target node.
- Mouse over to highlight all the edges to and from the desired node.
- Custom function call instruction visualization window showing passed arguments, the names used to receive them as well as their values.

Branching - Custom window showcasing:

- The whole statement and its global result.
- Statement breakdown and the results for each evaluated portion of it and object state links.
- The previous failed statements if it applies and a "view" link to show that statement.

Assignment - Visualization of assignments consists of:

- General object information (name, data type, data structure).
- New values information (origin, size, values).
- Object modifications history with the events in which they occurred (with "links" to visualize them).

Assignment (from events) - If the origin of the assignments' data is an event, the node content corresponds to two labels:

- **LHS**: described above (assignment).
- **RHS**: link to the event which originated the new values

Index/cell changes - Custom visualization of the modified indexes/cells of a given object:

- General object information (name, final values)
- Changes information
    - **Indexes:** number of changes
    - **Cells:** number of rows and columns
- Data source (with link to visualize its state)
- Changes breakdown
    - Modifiable visualization window.
    - Navigation through the defined window
- **Note:** the visualization only comprises the new values and the indexes/cells they were associated to.

Index/cell changes (from events) - Uses the same approach as the one used and described in the above "Assignments (from events)".

Looping - Depending on the loop used, the user can visualize:

- The condition used (while loops)
- The expected/effective number of iterations (for loops)
- The executed instructions per iteration with navigation capabilities to access the desired iteration

Conditions – Warnings and errors support

- **Warnings** - Symbol appended to the instructions (on the side). Its message is displayed inside a tooltip and can contain more than one warning message.
- **Errors -** The instruction that caused the error is displayed in its complete form, with the [ERROR] prefix. It also contains the message in a tooltip

Arithmetic operations - The arithmetic operations visualization follows the same structure as the one used in the branch analysis.

Data visualization - The data visualization uses the same methodology of the index/cell changes feature. The only difference is relative to the number of elements (n):

- **n < 5**: The values are shown in-place
- **5 < n < 15**: A tooltip is available with the values in normal ordering (for one-dimensional objects)
- **15 < n**: The values are shown with the format and visualization capabilities as described in the index/cell changes language feature.

IMPLEMENTATION

Supported features

Considering the complexity and extent of a programming language and also the time and human resources constraints for the development of this work, the current version of ABD_tool does not support all the R language features or data structures. The main focus was to develop the language aspects that would show the potential and the capabilities of the architected approach for the debugging problem.

Data structures – Vectors (single dimensions) and Data Frames (multiple dimensions) .

Data types – Integer (INTSXP), Real (REALSXP), Strings (STRSXP), Closures (CLOSXP, functions), Symbols (SYMSXP, objects).

Data structures operations – Distinction between assignment and modification, index and cell changes with data precedence linkage, comprehension and object type changes.

Branches – Complete implementation (No limitations regarding length, depth and nesting).

Looping – Complete implementation (all loops, no limitations).

Function calls – Local variables, returns linked to objects' modifications, environments/contexts distinction.

Warnings collection – Association warnings-events (allows multiple warnings per event, without limitation).

Error detection – Error detection and script line association.

Events

The available events describe what R language features the ABD_tool supports. These events are maintained in memory, in a registry named eventsRegistry, until the data export phase. This eventsRegistry structure consists of multiple ABD_EVENT structures connected as a single linked list and can assume the following types:

MAIN_EVENT - Used to initialize the events registry.

IF_EVENT - Describes an if statement.

FUNC_EVENT - When a user-defined function is called.

RET_EVENT – Created when a user-defined function returns. Also stores the destination of the returned values: An object and state combination or NULL (when the value was not used for an association).

ASGN_EVENT - Registers when an assignment occurs.

ARITH_EVENT – Created when arithmetic operations are performed.

VEC_EVENT - Used when an object is assigned from a new vector (hardcoded vector declaration).

IDX_EVENT – When a vector index is modified. Specifies the object, the new state and the value origin.

FOR_EVENT, REPEAT_EVENT, WHILE_EVENT - All maintain a list of iterations, each containing a list of events that occurred during that loop-iteration.

NEXT_EVENT, BREAK_EVENT - The next and break events determine when a loop interrupted their current iteration. The break can also be part of an if statement.

FRAME_EVENT - Registers a new data frame creation. Also stores information regarding the data source.

CELL_EVENT - Same as IDX_EVENT, but for multi-dimensional objects.

Displayer

The displayer was developed using mainly HTML [8] and JavaScript [9] and the data it needs originating from the four JSON [10] files produced by the JSON helpers component. These files represent the script (code.json), the generated events (events.json), the used objects (objects.json) and the thrown warnings and errors file (wrn_err.json). With these files, the dynamic execution graph is generated by iterating over all the events. The processing of each of those events results in the combination of an HTML template (according to the event type) and the event specific (and relevant) data.

Graph generation

To generate the graph, every time the displayer notices FUNC_EVENT it creates an entry with the environment address as the key in a Map object variable named envirContent. The executed code, that is associated to the environment memory address in the envirContent, is maintained under the form of a Map object (another) where the key is the script line and the value is a list with the generated HTML for the events that occurred at that line. This approach allows the ABD_tool to have, theoretically, an unlimited number of events per line of code. With this, the final structure holding the created events by the ABD_tool are two nested Map objects (Figure 4).

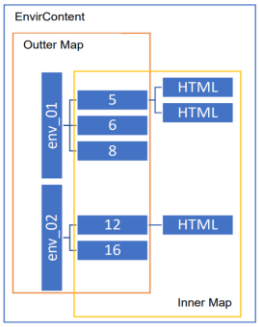


Figure 4 - EnvirContent abstract overview

Object current values

The objects' current values resolution is done in a backtrack mode starting from the requested state and finalizing when it reaches an object (re)definition state. It creates an auxiliary Map variable that contains the modified indexes (or the cells) as its keys, and the most recent seen values as the key's value. Considering this, when resolving the current values, every time the displayer encounters an object modification it verifies if the indexes used in that modification are found in the auxiliary map:

- **Indexes not found**: they are added to it.
- **Indexes are found**: they are skipped.

When the algorithm finds the defining state, it will copy the complete structure with the initial values and patch it with the values contained in the Map for the found addresses generating the current values for a determined state.

Constraints

Due to web browsers recent security policies, they cannot load files directly from the file system without the user's explicit intention. To mitigate this, the exported files need to be duplicated into the displayer folder in order to include them in a JavaScript file (with hard coded instructions to the specific file names), making the exportation performance twice as worse.

The second limitation is the JavaScript capabilities when parsing JSON files. It limits the size of the heap to 1.9 Gb and, since the allocated memory to store the data parsed from the JSON file will reside in the heap, it limits the loading size of the exported data to that quantity.

Optimizations

The first optimization was the creation of separated object registries to differentiate objects that store data and objects that store functions. This reduces the time spent when searching for objects to register a new value and also when verifying if the function being called was defined by the

programmer. To optimize the registries searches even more, the objects are ordered by their usages.

The entry of a new object usage, for an object that was not defined, is also optimized due to the fact that for each registry a pointer is maintained to the least used object so, when a new object is detected and needs to be registered, it is directly appended to the end of the list without the need to traverse it. Its ordering can also be skipped.

To minimize the memory usage, the complete values of the common objects are only stored when it is defined or redefined. The index modifications were optimized by registering only the new indexes and the corresponding new values (the delta's), which reduces the memory used and the time complexity to register the changes. As new values are assigned to an object, they are appended to the modifications list (double linked list) in the object structure. To optimize the accessing and insertion of new values in the list, pointers to the head and tail of the list were created to avoid the traverse of the list every time it needs to be extended or accessed. In this case, the head represents the very first modification made to the object and the tail represents the state that the object is currently at. This approach also optimizes the data exportation phase time complexity.

EVALUATION

Test environment

Virtual machine running over VMWare Player
Operating System: Manjaro 20.0.3 KDE linux (Arch based)
Number of CPU's (cores): 4
Memory capacity: 8192 Megabytes
Video configuration: Accelerated 3D graphics enabled
Graphics memory: 768 Megabytes

Performance measures

In order to understand how the ABD_tool impacted the R execution time, multiple benchmarks were made related to different aspects of the language. To ensure consistent results, the tests were conducted with five time-complexity incremental tests for each section. Each of those tests were run five times and the final value of that individual test was achieved with the average of the five runs. Their results are presented in two forms:

- Table-based (Contain the detailed information for each time-complexity increment)
- Graph-based (logarithmically scaled graphs for a better visualization of the overall performance disparities).

Micro-benchmark 1 – Recursion

R limits the depth of recursion to 999 calls, throwing an error at the 1000th call. To mitigate this, multiple recur() calls were made (to generate more recursive calls) instead of increasing the depth of the recursion (Figure 5).

```
recur <- function(x){
    if(x < 999){
        x <- x + 1
        x <- recur(x)
    }
    x
}
```

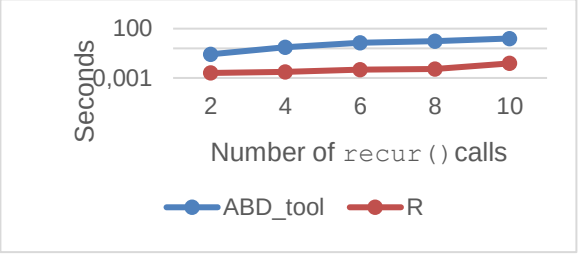
Figure 5 - Micro-benchmark 1 R code

This micro-benchmark stresses the following ABD_tool functionalities:

- Function calls (FUNC_EVENT, EVENT_ARGS, ABD_OBJECT, ABD_OBJ_MOD, local variables and the concept of contexts)
- Simple branch conditions (IF_EVENT)
- Arithmetic expression (ARITH_EVENT)
- Assignment (ASSGN_EVENT)
- Return (RET_EVENT)

Number of `recur()` function calls	Results (seconds)	
	ABD_tool	**R**
2 calls (1998 recursions)	0,25s (+7713%)	0,0032s
4 calls (3996 recursions)	1,296s (+32300%)	0,004s
6 calls (5994 recursions)	3,6s (+51329%)	0,007s
8 calls (7992 recursions)	5,37s (+68746%)	0,0078s
10 calls (9990 recursions)	10,05s (+33400%)	0,03s

Table 1 - Micro-benchmark 1 results



Graph 1 - Micro-benchmark 1 results

As described and shown by the micro-benchmark results (Table 1 and Graph 1), the overhead added is moderate to high but completely usable in real world scenarios due to the fact that recursion is not highly used and when it is, the depth is not, usually, on the levels of the ones benchmarked.

Micro-benchmark 2 – Data import and processing

Since R is more commonly used in data science, this micro-benchmark (Figure 6) verifies how much the ABD_tool impacted the loading of a data set from a data source, and

its consequent storing. The file structure consisted of multi-type columns accounting for a total of fourteen columns.

```
df <- read.csv(file = "file.csv")
```

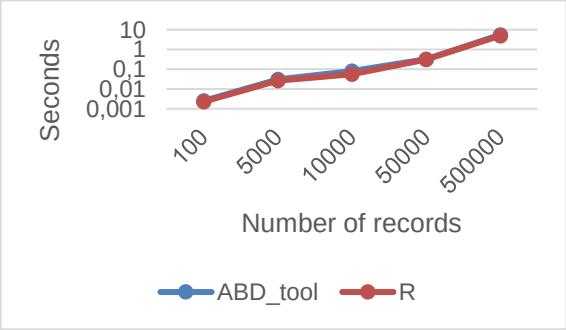
Figure 6 - Micro-benchmark 2 R code

The multiple actions performed by the single line of code can be translated to the following ABD_tool' functionalities:

- Process the data:
  - Each column (ABD_VEC_OBJ)
  - Memory management (ABD_OBJECT, ABD_OBJ_MOD)
- Data frame creation (ABD_FRAME_EVENT)

Loaded file content size	Results (seconds)	
	ABD_tool	R
100 records (1400 cells)	0,0025s (+10%)	0,00228s
5K records (70K cells)	0,03s (+15%)	0,026s
10K records (140K cells)	0,08s (+43%)	0,056s
50K records (700K cells)	0,32s (+4%)	0,308s
500K records (7M cells)	5,53s (+11%)	4,96s

Table 2 - Micro-benchmark 2 results



Graph 2 - Micro-benchmark 2 results

The micro-benchmark 2 results show, in Table 2 and Graph 2, that the ABD_tool did not increase in any substantial quantity the time consumed to perform all the tasks needed to store the sourced data. The time consumed increased in the same proportion as in the standard R interpreter and the results, most of the times, overlap when visually analyzed in the graph.

Micro-benchmark 3 – Looping and branching

One of the most time-consuming instructions in programming is branching, mainly because the compiler ,most of the times, cannot predict the outcome of those branches and, thus, cannot optimize the code to run faster. The quantity of branching needed to produce measurable and relevant values is humanly impossible to produce. To mitigate this, the branching micro-benchmark had to be based on a loop that increases the processing power required at each iteration. The for-loop, for each phase of evaluation, will increase the number of iterations (non-linearly) and consequently the number of performed branch conditions test (Figure 7).

```
for(i in 1:1000000){
    if(i<1000){
       print("smaller 1")
    }else if(i<10000){
       print("smaller 2")
    }else if(i<400000){
print("smaller 3")
    }else if(i<600000){
       print("smaller 4")
    }else if(i<700000){
print("smaller 5")
    }else if(i<900000){
       print("smaller 6")
    }else if(i<1000000){
print("smaller 7")
    }else{
       print("equal")
    }
}
```

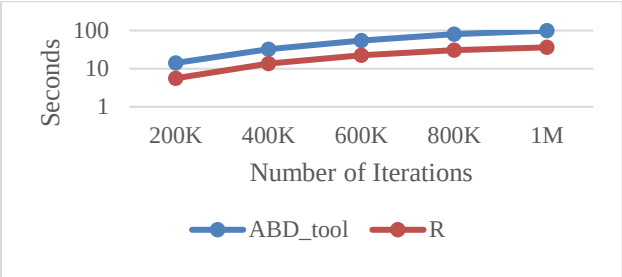
Figure 7 - Micro-benchmark 3 R code

The ABD_tool functionalities targeted by this micro-benchmark are the following:

- Looping (FOR_LOOP_EVENT and VEC_CREATION)
  - Create lists with the multiple iterations (ABD_ITERATION) and assign the created events to each iteration
- Complex branch conditions (IF_EVENT)

Number of for loop iterations	Results (seconds)	
	ABD_tool	R
200K iterations	14,16s (+152%)	5,61s
400K iterations	32,52s (+141%)	13,48s
600K iterations	54,71s (+145%)	22,35s
800K iterations	80,4s (+164%)	30,45s
1M iterations	99,24s (+174%)	36,21s

Table 3 - Micro-benchmark 3 results



Graph 3 - Micro-benchmark 3 results

The ABD_tool for the first four tests, as shown by the results in Table 3 and Graph 3, performs roughly 2.5 times slower than the standard R interpreter and the performance downgrades closer to 3 times when the number iterations increases to one million.

Macro-benchmark

In order to have a closer comparison to real usage, the benchmark was performed taking into consideration factors that were not considered previously (Figure 8).

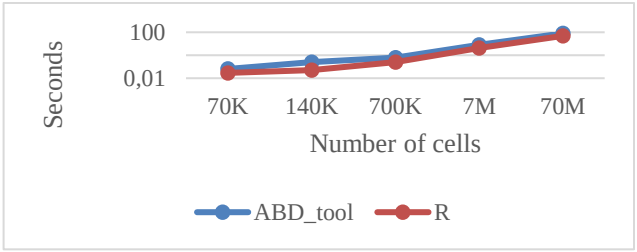
- Load a much bigger file

- Perform calculations over the data loaded from the file (Total profit column): average, sum, minimum and maximum

- Post-execution debug data export (ABD_tool's data).

```
df <- read.csv(file = "file.csv")

#calculates the total profit
total_profit <- sum(df[,"Total.Profit"])
print(paste0("Total profit: ", total_profit))

#calculates the average of the total profit
average_profit <- mean(df[,"Total.Profit"])
print(paste0("Average profit: ", average_profit))

#find the row with the lowest profit
min_index <- which.min(df[,"Total.Profit"])
print("Record with lowest profit: ")
print(df[min_index, ])

#find the row with the highest profit
max_index <- which.max(df[,"Total.Profit"])
print("Record with highest profit: ")
print(df[max_index, ])
```

Figure 8 - Macro-benchmark R code



Graph 4 - Macro-benchmark results

Number of Cells	Tool	Execution timing results (seconds)			Memory usage
		Process	Export	Total	
70K	ABD_ tool	0,03s +3%	0,034s	0,064s +55%	0,7% ~57MB
	R	0,029s	-	0,029s	0,7% ~57MB
140K	ABD_ tool	0,066s +21%	0,192s	0,258s +80%	0,8% ~65MB
	R	0,052s	-	0,052s	0,8% ~65MB
700K	ABD_ tool	0,28s +7%	0,33s	0,62s +58%	1.3% ~106MB +0.2%
	R	0,26s	-	0,26s	1.1% ~90MB
7M	ABD_ tool	4,38s +5%	3,46s	7,84s +47%	2.6% ~212MB +0,5%
	R	4,18s	-	4,18s	2.1% ~172MB
70M	ABD_ tool	50,12s +3%	29,45s	79,57s +64 %	37,1% ~3039MB +50%
	R	48,46s	-	48,46s	18,6% ~1523MB

Table 4 - Macro-benchmark results

From the results (Table 4 and Graph 4), as also stated in the micro-benchmark 2, the data processing speed of ABD_tool does not add a noticeable overhead to the R interpreter performance. What does in fact add time complexity to the execution is the data exportation with roughly thirty seconds increase to the overall consumed time (on the 70 million cells file). It can also be seen that the total amount of memory that ABD_tool requires over R is mainly due to the initial data duplication meaning that, the used structures by the tool, do not consume a significant amount of system' resources when comparing with R.

User-study

Performing a detailed ABD_tool user study was not possible at the current world' state due to the COVID-19 pandemic. To mitigate this roadblock, a post in the Reddit

platform was made to assess the users first impressions as well as the ABD_tool potential. The post was constructed with a small presentation of the project and a list of the most relevant features that are not present in other R debugging environments.

To finalize the post, a small survey with two options were provided to users (no question):

- Answer 1: You got my attention
- Answer 2: I do not think it is useful

The survey was intended to be small but direct in order to make its answering fast and doable (people tend to avoid answering surveys). The results to the survey are:

- Answer 1: 73 (seventy-three)
- Answer 2: 11 (eleven)

Besides the survey, some users commented and appreciated the effort and inquired about the difference to some existing tools. They, after the explanations to their doubts, commented that the ABD_tool was something worthwhile for the community.

Discussion

Considering the obtained results, with the micro and macro-benchmarks and the user-study, a statement can be made: the ABD_tool has the potential to become one of the primary debugging tools to debug R programs. Although some benchmarks might discourage this statement at a first glance, when further analyzed, they showcase that the performance penalties are not problematic when considering the value given by its functionalities.

The micro-benchmark 1 is a proof of that. When analyzing the results superficially (Table 1 and Graph 1), it is visible the significant execution time overhead added by the tool to the R interpreter but, when taking a closer look into the data, it is visible that the performance penalty is acceptable when considering some key factors:

- The ABD_tool scales considerably well (Graph 1)

- Recursion is a programming technique that does not suit R very well as seen by the recursion depth limitation (dynamic programming does suit)

- The amount of recursion used is extremely unlikely to be seen in an R program

Another important aspect regarding this micro-benchmark is the value that ABD_tool brings to the programmer. The possibility of using the debugging UI to visualize each of the recursive calls, and analyze their arguments as well as all their instructions (including the value returned by each of them) with a single program run, at any time post execution, is not measurable when comparing to debugging recursive programs using the traditional approach. Using a traditional approach, if a programmer wants to debug the 998th call, he/she has to either script the debugger to stop on that particular call or skip all the unwanted calls manually. With ABD_tool, the information was already collected in one run.

The micro-benchmark 2 results (Table 2) clearly support the validity of the ABD_tool as a debugger, since the primary applicability of R relies on huge datasets usage. The results are promising, showing an overhead addition ranging from 4% (+0,012s) to 15% (+0,004s) with an outlier of 43% (+0,024s) which, considering the speed of execution and the actual time increase, is not visually understandable when running the program. The ABD_tool also overlaps the R interpreter growth curve most of the times (Graph 2) showing that they scale identically.

When micro-benchmarking the overheads added by the ABD_tool to the R interpreter performance regarding looping and branching (Micro-benchmark 3), it is also possible to maintain the confidence in the tool's potential. The results seen in Table 3, show a ~2.5 (+141%) to ~3 (+174%) times execution performance downgrade when using ABD_tool but, when the productivity gains are considered, the values become acceptable and overlookable. Based on Graph 3, although they have a difference in performance, both applicants scaling curves are very similar.

In the most severe overhead addition micro-benchmark run (micro-benchmark 3, 1 million iterations), the actual time difference between the ABD_tool and the R interpreter is ~1 minute. The ability to analyze each of those instructions on-demand at any time, compensates the time difference. For example, if a programmer wants to analyze the 999 999th for-loop iteration using the traditional R debugging techniques, he/she needs to have the same approach as the one mentioned afore. Using the ABD_tool the programmer just needs to launch the ABD_tool' displayer and select the iteration he/she wants to analyze.

The final performance benchmark (macro-benchmark) took in consideration new factors: exporting the generated structures and the loaded data usage.

With the results (Table 4 and Graph 4), it is visible that the major performance penalty that the ABD_tool results suffered originated from the generated data exportation. This is mostly due to web browsers privacy policies constraints. Also, another impactful aspect for the data export by the ABD_tool is the disks read/write speed and, with the test environment being a virtual machine, it made the time consumed by persisting the information worse. Memory usage was also tested with the highest disparity in the results residing in the 70 million cells file run (which used a .csv file of ~500MB). The difference (+50%) is explainable by the initial data duplication made by the ABD_tool.

The final performed test was not performance related, but user related. It partly mitigates the absence of a full user study that was supposed to be performed. The questionnaire

focused on users' opinions on the ABD_tool potential and it is possible to conclude that the users were interested in the tool and its potential.

CONCLUSION

The *Automatic Bug Detection in R* work implements the concept of timeless debuggers in order to achieve a more efficient and effective debugging process which is divergent from the available techniques in R (and the most used third-party applications).

The ABD_tool implements several functionalities and mechanisms to achieve, and to provide to its users and R programmers, the initial projected goals:

- **Simplify the debugging process** (application usage simplicity; focus on the code and not the process of debugging)

- **Enhance the capabilities of the debugging tools** (data analysis at any given time; debugging oriented UI; graph visualization by environment and custom display for each supported R instruction type).

- **Time savings** (one run collection; no breakpoint placement guessing; low/medium overheads added)

As demonstrated, the ABD_tool collects, processes and provides the UI for the programmer to visualize and understand what occurred during an R program execution at a given/desired time. Considering these goals and the overall results obtained with the performed benchmarks, the ABD_tool offers a good leap in productivity and easiness when debugging because the added overheads can be considered overlookable given the provided functionalities.

For instance, when working with big datasets (Macro-benchmark, 70 Million cells result), the ABD_tool only adds a 7,8% overhead (on average) to the processing phase of the script which, given the actual clock time, is extremely low considering the collected information. Even if exporting the data is considered, a 60,8% overhead addition continues to be very acceptable performance degradation because, if the programmer needs to re-run the program several times to place breakpoints at ideal places, that extra spent time by the ABD_tool is already surpassed by the standard techniques procedures.

Since the ABD_tool has its displayer implemented with the web-browser experience in mind, the data visualization is very flexible allowing experienced programmers to expand the displayer functionalities, use a completely different graph visualization library or use the exported data to create a completely new UI based upon it.

The developed work coverage is in an advanced state with most of the structures and data types supported and most of the remaining functionalities do not constitute a greater challenge to implement, with the only resource needed being time.

ACKNOWLEDGMENTS

I would like to begin by acknowledging the institution and the direct participants that helped in the development of this work, *Automatic Bug Detection in R*. This dissertation was completed due to their support in myriad forms.

I would like to express my gratitude to Instituto Superior Técnico for allowing its students to study such topics and make them better at creating meanings to mitigate missing knowledge by reinventing and improving themselves. Also, i would like to show appreciation to my supervisor, Professor João Coelho Garcia, for all the guidelines and the countless hours revising and improving the developed application in order to have a successful and the best possible "product" that meets the expected outcomes.

Last, but not least, I want to thank my course colleague and friend João Maria Tiago for an initial introduction to some web-application technologies used in this work and, also, to a friend, João Paulo Ferreira, for helping revising the semantics and syntax of this document.

REFERENCES

[1] T. Britton, L. Jeng, G. Carver, P. Cheak and T. Katzenellenbogen, "Reversible debugging software," *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[2] R Core Team, "R language definition," *Vienna, Austria: R foundation for statistical computing*, 2000.

[3] H. Wickham, Advanced R, CRC press, 2019.

[4] R. C. Team, "R internals," *R Foundation for Statistical Computing*, vol. 3, 1999.

[5] Kaggle, *2018 Kaggle ML & DS Survey*, 2018.

[6] G. Hotz, "Timeless debugging," 2016.

[7] A. Di Federico, "Compiler techniques for binary analysis and hardening," 2018.

[8] I. S. Graham, The HTML sourcebook, John Wiley & Sons, Inc., 1995.

[9] R Core Team, "Writing R extensions," *R Foundation for Statistical Computing*, 1999.

[10] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte and D. Vrgoč, "Foundations of JSON schema," in *Proceedings of the 25th International Conference on World Wide Web*, 2016.

[11] D. Crockford, JavaScript: The Good Parts, " O'Reilly Media, Inc.", 2008.