Automatic Bug Detection in R - Approaching R debugging differently

Luís Miguel da Conceição Rodrigues

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. João Coelho Garcia

Examination Committee

Chairperson: Prof. José Carlos Martins Delgado

Supervisor: Prof. João Coelho Garcia

Member of the Committee: Prof. José Faustino Fragoso Femenin dos Santos

January 2021

Acknowledgements

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.

Firstly, 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.

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.

Abstract

As in a wide variety of programming languages, the debugging capabilities available of R (either through its native mechanisms or via third-party applications) have not seen many advances in the past decades. Due to this lack of innovation, the debug paradigm remains the same since the language was officially launched: define a desired stopping point, analyze each instruction to ensure its correctness and iterate through the instructions until the problem is found. This paradigm constitutes a completely linear debugging process relative to time.

The *Automatic Bug Detection in R* implements a completely different paradigm (non-linear timewise): timeless debugging. With this, it provides to its users an execution graph based on the data collected during the execution, allowing them to analyze each executed instruction at any time (either during or post execution). This means that, to analyze the n^{th} instruction, rebuilding the execution context (by executing the n^{th}-1 instructions) is not required because the data regarding those instructions was already collected and persisted in the first run.

Conceptually, this type of approach has inherent overheads which means that the overall performance (when compared with a clean R interpreter) downgrades, due to the extra instructions. The ABD_tool provides its complete functionalities with low/medium overheads. For example, a 1 million iterations for loop with multiple branches inside it (section 5.6), takes 1 minute more than the clean R but offers the capability of analyzing each iteration separately, and the executed instructions within them on-demand. **Keywords**: R; Debugging; Timeless debugging; Dynamic debugging.

Resumo

Tal como em grande parte das linguagens de programação, as capacidades de depuração disponíveis para o R (quer seja através dos mecanismos nativos ou via aplicações terceiras) nao tiveram grandes avanços nas últimas décadas. Devido a esta lacuna de inovação, o paradigma de depuração continua o mesmo desde o lançamento oficial da linguagem: definir um ponto paragem, analisar cada instrução para assegurar se está correta e iterar sobre todas elas até que o problema seja encontrado. Este paradigma constitui um processo de depuração totalmente linear relativamente ao tempo (para estar num determinado ponto da execução, é obrigatório que as n-1 instruções sejam executadas).

A *Automatic Bug Detection in R* implementa um paradigma completamente diferente (temporalmente não linear): depuração intemporal. Posto isto, a ferramenta fornece aos seus utilizadores um grafo gerado a partir dos dados colecionados (durante a execução), permitindo-lhes analisar, em qualquer altura, cada instrução executada (durante ou após execução). Isto significa que, para analisar a enésima instrução, reconstruir o contexto de execução, executando as n-1 últimas instruções, não é necessário porque os dados relativos ao processamento dessas instruções já haviam sido guardados na primeira execução.

Conceptualmente, este tipo de abordagem implica algumas penalizações de desempenho, o que significa que, no geral, a velocidade de execução diminuirá (quando comparado com o interpretador nativo do R) devido às instruções adicionadas. A ferramenta ABD_tool também providencia todas as suas funcionalidades com uma penalização de desempenho consideravelmente baixa/média. Por

exemplo, um ciclo for com 1 milhão de iterações e múltiplas instruções IF (secção 5.6), consome mais 1 minuto que o R original, mas oferece a capacidade de analisar cada iteração do ciclo separadamente, bem como as instruções executadas em cada iteração, quando desejado/necessário.
Palavras-chave: R; Depuração; Depuração intemporal; Depuração dinâmica

Table of contents

List of figures

List of tables

List of graphs

1 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 debuggers, inadequate tooling, lack of code structure and documentation, etc.

R [2], while it is growing, its adoption is not high: 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 in developing new tools for it. A study [3], made within the context of machine learning and data science, places R in second place for both of the following questions:

- Programming language used most often (Figure 1):
  - 1st place: Python (54%)
  - 2nd place: R (13%)
- Programming language recommended by data professionals (Figure 2):
  - 1st place: Python (75%)
  - 2nd place: R (12%)

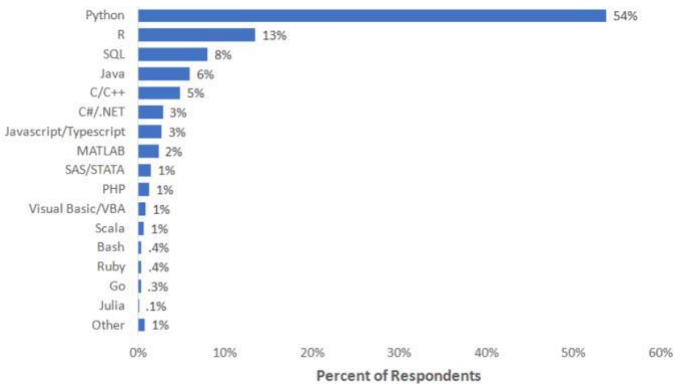


Figure 1 - Study results: Programming language used most often

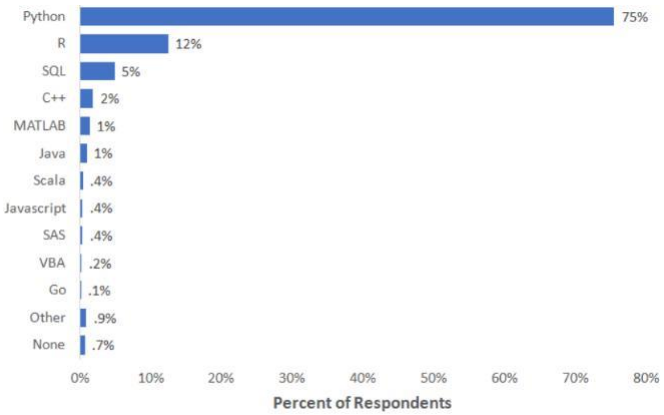


Figure 2 - Study results: Programming language recommended by data professionals

RStudio [4] is the richest Integrated Development Environment (IDE) available for R. Although it is the most powerful, it lacks some functionalities when the need to debug arises. For example, it only implements visual elements (wrappers) for the native R debug commands (breakpoints, step in, etc.) allowing the user to issue the commands via buttons instead of using the console. Since debugging has a major impact in the software engineering process and the available tools for R rely, mostly, on wrappers to the commonly known debugging techniques, their low performance can be easily showcased, with a simple example in order to demonstrate why those techniques may be outdated and how they can be improved with a different approach.

Imagine a simple program section that contains two functions, the `main()` and `compute()` functions. `Compute()'s,` either sums or subtracts the two received arguments depending on a variable value, in other words, a flag variable. To enrich the example, consider that these functions (`add()` and `sub()`) are part of a library and they perform some other calculations before summing or subtracting (it is not important what these calculations are). Also, consider that the programmer wants to debug only this part of the program, but it contains more code before the described portion. After `compute()'s` return, the program performs a time-consuming computation on the returned value. To conclude, the program prints to the console the final value of that computation.

```
function compute(arg1, arg2){
    if(flag)    result = add(arg1, arg2)
    else        result = sub(arg1, arg2)

    return result
}

main(){
    (...)

    result = compute(obj1, obj2)
    result = heavy_processing(result)

    print(result)
}
```

Figure 3 - Example pseudo-code

The programmer, in order to verify if the algorithm is correct, performs tests using a set of well-known inputs that should generate an expected output. After all the tests are concluded, the generated outputs do not match the expected ones, leading the programmer into a debugging phase.

To approach this problem, the programmer places a breakpoint before the heavy computation to view the value returned by the `compute()` function which will then be used to perform the computation and re-run the program. After the debugger hits the declared breakpoint, and also after the programmer analyzes the values of the variables, he/she concludes that the values received do not correspond to the expected ones and the problem should be the values passed to the function as arguments. He/she then proceeds to place another breakpoint before the `compute()`'s function call, to verify if the

arguments passed contain the correct values. The programmer needs to re-run the program to hit the new breakpoint, making it the third program run.

When the new breakpoint is hit, the values passed as arguments match the expected ones and he/she proceeds by clicking `Execute Next Instruction` expecting to view the next instruction that the program would execute. However, the R debugger executes the function call without jumping into it. The semantics does not match the implementation because `Execute next instruction` should mean execute the function call and not the function (as a whole), leading the programmer to re-run the program a fourth time.

He/she proceeds and, instead of `Execute Next Instruction`, he/she steps into the function and verifies that the flag contains an unexpected value which, consequently, makes the function execute the wrong branch leaf. The programmer does not know why and re-runs the code with the intention to hit the first breakpoint (before the function call), to verify the flag variable value.

Finally, when running the code for the fifth time, he/she understands that the flag variable was carrying values from previous code, making `compute()'s` to behave unexpectedly. Since he/she, after several runs of the script, found the issue and patched it, it is now possible to re-run the tests and verify if the outputs match the expected ones (needing to perform the heavy computations again).

This is the common approach to find bugs in code but 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 hypothetical tool was running, and he/she did not had to run the program more than one time, the time savings would be enormous and with an immediate impact which would give the programmer more time to eventually debug other sections of the code or to simply implement new features into the program.

This can be achieved due to the fact that the data collected, by the designated tool during the program runtime, would provide to the programmer an extensive context of each executed instruction during the program execution. 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 during the test phase.

The aforementioned hypothetical tool is, in fact, a Timeless Debugger [5] [6] (TD) which collects information regarding the execution of a given program. The most relevant aspects about an execution are the ones that allow the algorithm to reproduce what were the exact values of the manipulated objects, when those objects changed, how those changes occurred and in what context they were executed on.

The *Automatic bug detection in R's* (ABD_tool for short), which will be the main result of this dissertation, will implement the paradigm of a timeless debugger. This idea intends to give the user the advantage of running his/her script only one time instead of multiple executions, without having to carefully insert

breakpoints at ideal places since it collects and supplies all the execution information for a later analysis. This efficiency improvement 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. This means that, if he/she wants to analyze a given instruction, running the prior n-1 instructions is not required because their execution related data was already collected in the initial execution and its access is instantaneous and on-demand.

The application also intends to present the information in a logical, clear and straightforward form to allow the user to focus more, and in first place, in the problems within the code instead of spending time searching and developing his/her skills with an application to then try to find the problems mentioned before.

With the ABD_tool, R programmers have at their disposal the ability to visualize branches with statement granularity (in order to identify which portion of the whole statement evaluated to true or false), the capability to, in else if clauses, verify what prior branch conditions failed and led the program execution to the current branch being evaluated. The programmer also has the possibility to visualize the data precedence between objects and their modifications (or declarations), the warning messages and errors thrown, function calls arguments mapping (as well as their values) and much more.

2 Related work

The related work chapter explores how the debugging process in the R language is currently handled, what tools the programmers have at their disposal to improve that process (to make it more efficient and effective) and how a new debugging paradigm can be beneficial to the programmers in order to increase the aforementioned efficiency and effectiveness.

The available debugging tools and techniques for R are few and sparse in functionalities, in comparison with other languages and platforms, and therefore, this is also a chapter where tools for other applications are demonstrated in order to learned from them to then, improve the R debugging process.

Programmers debug their code in different ways timewise. For example, debugging a binary executable uses a completely linear debugging technique since it cannot (most of the times) analyze values in the past and at any desired place. Classifying debugging tools according to time, the tool developed in this work can be considered non-linear (timeless) since it allows the programmer to analyze any part of the executed code at any desired moment without the necessity to rerun the code again (considering the exact same input).

2.1 R

The R language is an interpreted language that resulted from the S language [7] implementation combined with lexical scoping semantics (to allow the binding of entities to names, in R case, the binding of values to objects, respectively). Its first, and official, “stable beta” was released in 2000 [8] [9] (version 1.0) and it has improved incrementally until today’s date (version 4.0.3).

The fact that the language is interpreted, gives programmers the flexibility of writing cross-platform code which means that, if they produce/develop one program in a Linux machine, it will run unaltered in any other Operative System’s (OS’s) and Central Processing Units (CPUs). This is possible because the program does not need to be compiled for a specific Instruction Set Architecture (ISA) [10], it is not dependent on the libraries installed in the system nor in the file type recognized as executable by the OS’s. This is possible because the language has an interpreter that will execute each line of the program by translating/parsing the R instructions into already known, implemented and compiled methods. R also has a wide variety of libraries (in the language nomenclature, packages) that can be imported to aid their programmers when extending the programs’ functionalities is needed – i.e. when the base R installation does not have needed features.

2.1.1 Language specifics

The language specifics section showcases the most important aspects of the R language considering what the ABD_tool intends to accomplish and how the R language behaves in certain scenarios. This being said, there are several important aspects when designing an R debugger: the language data types and data structures, how the instructions are structured internally (before being evaluated), the possible

conditions (errors, warnings, messages) and, also, how new contexts are introduced to the currently executing one.

2.1.1.1 Data types and data structures

Like in other languages, R has its own data types as well as data structures. In this case, both of them are strongly adapted to the language' original goals: provide a feature rich and flexible language for statistical usage. This is achieved with:

- The data types and data structures implemented in the R language are oriented to larger and volatile datasets (they change frequently)
- They can be easily manipulated and modified
- The code needed to perform the majority of the tasks is smaller (when compared to other languages)
    - The objects are not strongly associated to any data nor structural types.
    - They are also extremely flexible in terms of modifications
    - The language has a vast amount of built-in functions and a great library repository

The data types nomenclature [2] is mostly self-explanatory, but some key (and very specific) aspects exist and are important (Table 1).

R data type	Values' type
`NILSXP`	NULL value
`SYMSXP`	R objects (usually variables)
`CLOSXP`	Closures (curly brackets delimited blocks)
`ENVSXP`	Environments (execution contexts)
`PROMSXP`	The form that arguments assume until their usage (due to lazy evaluation)
`LANGSXP`	Instructions structure before evaluation
`CHARSXP/STRSXP`	Character strings and character vectors. The `CHARSXP` is used by R internally while the `STRSXP` refers to the user' R code strings.
`LGLSXP/INTSXP/ REALSXP`	Logical, integers and reals respectively

Table 1 - R language' data types

Another important aspect is how the objects are manipulated and what is possible to accomplish with them as well as the restrictions inherited by each type (Table 2).

Data structures	Structure specifications
Vectors	• Can contain any type of object • All the objects must be of the same data type ○ If they are not, type casting is performed and may generate errors
Lists	• Structurally similar to vectors

	• The stored objects can assume different data types as well as other data structures
Matrices	• Only support 2 dimensions • Have the same restrictions as the vector data structure
Data frames	• Support different data types for each column • Support different types for each object in the column ○ **Exception:** on creation, if a vector is used as a column then, all the vectors restrictions are applied. If a list is used as column, lists' restrictions are applied.
Arrays	• Share the same restrictions as matrices ○ Exception: more than 2 dimensions are allowed

Table 2 - Data structures specifications

2.1.1.2 Instructions structure

The R language, in order to obtain actual usable values, parses the user scripts into well-defined LISP [11] dotted-pair lists [12] (also called "S-expressions") which is a result from S direct heritage. All the created instructions are referred as calls in the R internals nomenclature, and they always have the same structure:

- **Head** – can assume two different forms
  - The operation to execute
  - The left-hand side of the operation
- **Tail** – can assume two different forms
  - NILSXP value (indicates that there are no more elements in the list)
  - The right-hand side of the operation (which might be an already evaluated value or another list)

To easily visualize the structure of the calls, a simple example can be taken into analysis with the following R instruction, which assigns a value to two objects (sequentially) with the scalar value 10:

newObject2 <- newObject1 <- 10

This instruction will produce a call with the following structure:

- **Call** - `[ <- , [newObj2, [<-, [newObject1, [10, NILSXP]]]]]`
- **Head** – `"<-"`
- **Tail** – another list
  - `[newObj2, [<-, [newObject1, [10, NILSXP]]]]`

This can be leveraged by extracting (with the accessors provided by R) the tail successively until the NILSXP value is found, providing the object being modified as well as the source of its new value. In this example, the information that the modified object is the `newObj2` can be extracted directly from the initial call structure by accessing the initial tail's head.

2.1.1.3 Conditions

In R, execution complications are called conditions and the code can produce at least three (more severe) types of those conditions which are errors, messages and warnings.

a) Errors

They are the most severe type of condition because they indicate that the function/program cannot continue, and the application must stop immediately. Errors are thrown by the `stop()` function and the most important aspect about them (and their messages' descriptions) is that they must inform the programmer in such a way that he/she can understand where the error has its origin and follow the right direction to solve the issue.

Errors imply that a repair is required because the function/program inevitably stops. Since they immediately stop the code execution, each function can only throw one error condition when executed (if not handled). This also applies to the program itself meaning that, as soon as an unhandled error occurs, the program stops completely. Their messages are only displayed when the program returns to the top level, leading sometimes to misunderstandings when reading the messages because they seem out of causal order, violating the mental code flow assumption made by the programmer.

b) Warnings

Warnings are at an intermediate level (between errors and messages) in terms of importance because they indicate that something went wrong during the execution. Warnings are signaled by the function `warning()`, thrown at return by default, and they have mostly an informative purpose. They are intended to be used when the action cannot be taken or did not end correctly, but, still, the program was able to recover and since they do not interrupt the function/program execution, multiple warnings can be shown/thrown per function.

c) Messages

Messages are the least impactful conditions, since they are mainly used to inform the users about a performed action that they do not know or, explicitly, intended to perform. These conditions are signaled through the function `message()` and they are intended to provide information about tasks that the program is doing on the programmer's behalf. For example, if the program for whatever reason needs to save the current state before some critical action, as a backup, a message should be displayed informing the user about that event.

Messages are also useful when describing the behavior of functions as well as the arguments they receive because, sometimes, the functions become unreadable due to their complexity and size (a vast number of parameters) or they are too generic (variable size arguments).
Furthermore, for long running tasks, it is interesting to give information to the programmer about the task progress and here, messages are very useful too because they are displayed immediately.

2.1.1.4 Environments

The environments were designed to associate/bind a set of names to a set of values providing context to an instruction execution. This translates to the R language usage as the concept of local variables and stack frames.

For instance, when a function is called, a new stack frame (for the new environment) is created and the very first bindings that are performed are related to the arguments that the function receives. This indicates that, when assigning a new value to the arguments' objects, their name already exists in that environment and thus it does not perform an object creation but a redefinition instead. Environments are organized as a list where each node points to its parent (the one from which the new environment creation was requested).

a) R initial environment setup

When the programmer wants to run a script, initial/default environments are created in order to provide a new execution context for it. These initial environments are composed by:

- **Empty Environment:** the root/base of the environment list. It does not contain anything (as the name suggests)
- **Base Environment:** contains the base R language functionalities that are native to the language such as functions, accessible global variables that modify certain aspects of the language, etc.
- **Global environment: i**nitially it is an empty environment, but it is the environment that the programmers' script executes over (creating objects) and from (calling functions) and thus, it is modified over time

R sets/arranges these initial environments in a hierarchical form, with the following structure:

Empty Environment (root) <- Base <- Global Environment (current)

b) Environments bindings

The objects bindings present in any of the environments can be extracted using a specific function called `envprint(env)`. This function prints the environment memory address, its parent and all the objects (more specifically, the bindings) that it contains. All the R execution information can be extracted recursively by selecting the current environment' parent, until the Global Environment or the Empty Environment are reached (if the programmer wants to access all the environments in the hierarchy).

c) Parents

The parent is what is used to achieve and implement lexical scoping (name-spacing) which consists in looking for names when they are not present in the current environment. For instance, if an object is not found in an environment, then R will make the same check in the parent of the current environment and will keep doing this procedure until it finds the object or reaches the Global Environment (it can also search until the Empty Environment, if the programmer forces it to).

d) Package loading and function calling

R groups environments and packages in the same structure. This occurs because packages also contain bindings, either for their functions or their objects (internally defined), and thus, in their essence and the way they are managed, they are identical to environments.
When attaching a new package to the execution, R will start including them, in the list, between the Base and the Global environments. This way, if the programmer attaches the packages A and B, sequentially and in this order, the list will have following structure:

Empty Environment <- Base <- **Package A** <- **Package B** <- Global Environment

Also, when a function is called, a new environment is generated to create a new context of execution to perform the bindings and the name-spacing over that new context (objects assignment, objects lookup, etc.). These new environments' names assume the form of their memory address contrary to the packages and the initial/default environments (which are represented by their name).
Their insertion procedure in the environment hierarchy is also different than the one described for packages because, and since these environments will eventually be popped (removed) from the list when the function execution finishes, they're pushed to the very bottom of it. For a call to the function named `fun1()`, as an example, the list will have the following structure (the address is fictitious):

Empty Environment <- Base <- **(packages)** <- Global Environment <- **0x82313255** (`fun1()` address)

The parenting for the function calls environments works as described for the packaging, having their parent pointing to the caller environment, which is the Global Environment (usually the `main` function) or another function environment (in this case, its parent is the previous function environment memory address).

e) Options settings

R has the capability to provide to its programmers more functionalities by changing settings regarding multiple aspects through the function `options()`. For instance, if the user needs to visualize the printed numbers with only four decimal digits, he can modify the default value (which is 7 decimal digits) to meet that requirement, with the command: `options(digits=4)`. The options function can also provide a great functionality when conditions are signaled as shown in the section below (Section 2.1.2 b).

The list of options is extensive and, theoretically, infinite because the programmer can create new options and apply them as if they were native to the language and.

2.1.2 Debugging techniques

A syntactically correct program may produce incorrect results. Although the program' grammar is correct, its logic is not and thus, it is important to analyze each one of its contexts (as well as their objects) to verify where the logic misaligned with the expected results. This can be achieved with debugging techniques which can be performed in two ways: through the more traditional methodologies (interactive debugging) or through automated tasks when such inconvenient occur during execution (non-interactive debugging).

a) Interactive debugging

Interactive debugging, as the name suggests, consists in using the IDE' commonly known features (or the R native debugging commands) to, in an iterative way, visualize how the contexts evolve during the instructions execution. This can be achieved by initially setting a breakpoint on the desired starting point for the iterative process (as done in every IDE, with a red dot on the side of the desired line of code or through the native function calls), followed by the usage of the IDE' buttons (or the R native commands) to iterate over the instructions - executing next instruction, step into the function call, etc.

This methodology of debugging is helpful when the problem' search space can be considerably reduced/constrained to a smaller portion of code, but it fails to provide the same help when the search space for the problem increases. This occurs because, iterating over each instruction and analyze them (the instruction and the context) can be time consuming and prone to errors. Due to the fact that, most of the times the exact line of code that triggers the error is not where the error has its origin. It is common for the programmer to conclude that the error originated elsewhere (the value of a determined object propagated through other pieces of code) making him repeat the process.

b) Non-interactive debugging

Contrary to interactive debugging, non-interactive debugging is used when some portion of the code is generating error conditions and it cannot be run interactively due to pipeline executions or, in some cases, the condition simply does not occur when debugging interactively.

R gives the possibility to change the options regarding what happens when an error is signaled. For example, it allows the programmer to dump the frames from desired environments when that happens. Those frames can later be loaded into the interpreter in order to analyze and understand what happened when the error occurred. This is possible because they contain all the information that the environments had when the error occurred.
Another way to overcome this problem is with "Print debugging" [13]. This method is commonly used to verify which code is in fact executed and delimit the analysis scope by reducing its search space. This technique consists in simply printing values to understand the mutations in the object's values or, for example, to identify what a flag contained before it is tested (at a branch operation).

2.1.3 Available debugging tools

In order to apply the available debugging techniques, and find the program's logic problems, programmers have several options when choosing the tool to achieve that: they have at their disposal native tools by using debugging-specific function calls (included in base R installation), third party applications and community produced packages (installed directly from R code).

a) Native tools

Natively, R offers debugging functionalities to its programmers. If they need to analyze some problem in their code, the programmers can use already built-in functions (Table 3) without needing to rely on

third-party tooling which, in many scenarios, can have their access limited/restricted (company policies, remote and/or field work, etc.).

R function	Functionality
`browser()`	The `browser()` function gives the programmer the ability to suspend/interrupt the execution and the inspection of the environment from which `browser()` was called from. When the execution is interrupted, the programmer can enter commands in order to perform several actions: • **c/cont** – exit the browser and continue execution • **f** – finish the current loop or function • **n** – evaluate next statement, stepping over function calls • **s** – evaluate the next statement, stepping into function calls • **where** – prints stack trace of all active function calls • **r** – resume the execution from the interruption line • **Q** – exits the browser (returns to top-level prompt)
`traceback()`	`traceback()` prints the sequence of function calls that led to the actual error and it also identifies the script line where the functions were called from (in this case, only user-defined functions contain the script line). This function can also be used to display the current call stack without the existence of an error in the execution nor the need to be in debug mode.
`debug()`	The function `debug()` allows the user to step through the execution of a function, line by line. Usually it is used to investigate what instruction generated the error, based on the output from the `traceback()` function (since it does not indicate where the error actually occurred, it needs to be "discovered" line by line). It can be used as simply as: `debug(func_name)`. When the debug mode is launched, the programmer can issue the same commands as when `browser()` is issued.
`recover()`	`recover()` allows the programmer to use `browser()` directly on any of the active function calls (functions that are present in the `traceback()'` result). This function can also be used as an option so, for example, when an error occurs, R will call the `recover()` function.

Table 3 - Native debugging tools' functionalities

b) GNU Debugger

The GNU Debugger (gdb) [14] is a program created in 1986, by Richard Stallman, and its widely known in the binary debugging community. The reason behind its popularity resides in the functionalities it provides to the programmers which are hardly matched by any other debugger available:

- **Calling functions while debugging**
- **Watch Points** – It allows setting watch points in variables, registers, address (etc.) and breakpoint the execution when those' values change.
- **Debugging Preprocessor Macros**

- **Automation** – Execute any desired commands at any reached breakpoint, reducing the programmers' typing necessity and repetitiveness.
- **Attaching to child/parent process** – The debugger can automatically attach to the desired threads that the process has meaning that, it debugs the child thread instructions instead of main' thread ones or vice-versa. Detaching is also possible.

R is not a primary use case for gdb, but it can still be used although the difficulty and the time that this technique requires is extremely higher than the native ones. To debug R code using gdb, the debugger needs to be attached to the R language interpreter process and, after that, start decompiling and disassembling the interpreter code in order to identify the desired code to be debugged (the actual C code that corresponds to the executing R script code).

c) Flow package

Since R native debugging tools are simple and, in some way, naïve, and using gdb is extremely difficult in the R language scenario, R programmers/users can develop extensions (through packages or code modification) to make the debugging process easier and more efficient. Flow [15] is such a package. It is developed and maintained by Antoine Fabri, under the username moodymudskipper, on GitHub.

As stated in the package repository "About" section, its objective is to "View and Browse Code Using Flow Diagrams" and it runs against a script or a function. This means that, given a code base, the package will create a static code flow graph based on the branches it contains. After it completes the run, exporting the results is a feature that the package supports.

The exported graph contains all the possible paths the script may end up following (with the followed path having a different drawn line), which means that it identifies all the leaves that a branch can eventually follow and the instructions that each of those leaves contain. What it does not do is consider the content of declared functions and the branches inside them, if they implement any.

This translates to a drawn graph that includes instructions that were not effectively run since it generates a static analysis UML syntax diagram of the code skeleton. The graph can then be exported to a collection of supported file types (HTML, png, jpeg and pdf). Although the Flow package exports the generated graph to HTML, its content is not interactive and does not display any information regarding the objects' stored data at each instruction (the values, types and used data structures) as well as their data precedence meaning that, the correlation between the objects in one instruction (and their data, as well as their structure) and its last usage/modification, is not visible, mentioned nor quickly identifiable.

While this package is an enormous improvement over the tools and functionalities currently available to R, the probability of it generating unreadable graphs (due to their density and complexity as a result of a static analysis) grows rapidly while the script code base increases slowly.

```
f1 <- function(){
  a <- 99
  b <- 98
  z <- 0

  if(z == 0){
    print("equal")
  }else if(z>0){
    print("bigger")
  }else if(z<0){
    print("smaller")
  }else{
    print("NA?")
  }
}
```

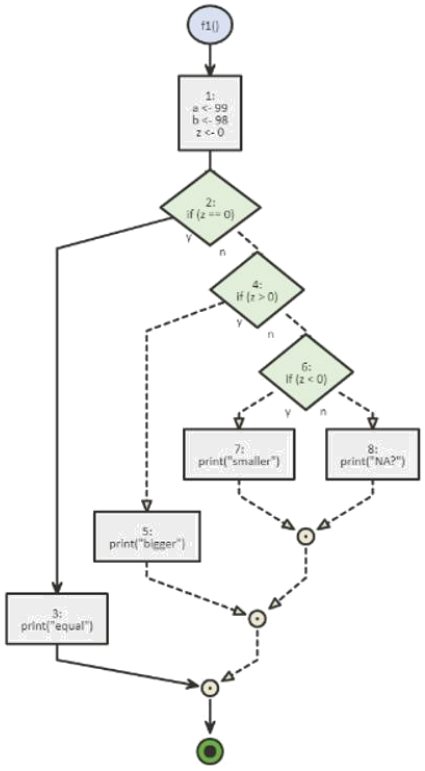


Figure 4 - Graph generated with small code base

Also, the diagram becomes less readable when functions are declared (Figure 5) because the package does not identify their declaration and does not create an independent "box" to describe those functions' content (it is processed like a normal assignment). This implies that the body of the function will be a simple assignment line and the package does not process the function instructions to generate its graph elements as it would normally do (appear just as plaintext even when branches are present).

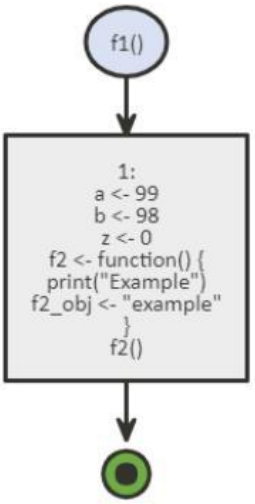


Figure 5 - Flow package functions' problems

2.2 Timeless debuggers

Timeless debuggers [5] [6] (TDs) provide to programmers a greater power when analyzing the changes that occurred to programs' variables (in R, objects), which functions were called at a determined context/environment (and the arguments/parameters passed to them), branch results by expression (the result for each part of the branch), and so on.

The great power of TDs resides in programmers not needing to find the ideal place to insert a breakpoint (which in many cases it is time consuming and difficult when the code base is complex and long), not needing to step through each instruction and analyze each object individually, removing the inconvenience to enter, without any intention, in functions that are declared in libraries and are not relevant to the runtime, etc. In other words, TDs offer what static analyzers and traditional debuggers offer with some differences - the programmer has effective data produced by every executed instruction when a set of inputs is given (the script provided). This data can be analyzed at any given time, without any iterative requirements (in order to build the execution context for the instruction to be analyzed) and it is composed by the correlations between the variables' data and its origin and the actual action/task executed.

In any post-execution moment, the collected data allows the reconstruction of every instruction and its components independently:

- **The exact moment each instruction was executed** - This implies a causal-ordered record of actions/tasks and the co-relation between executed instructions and the script (usually through the line of code)
- **The variables involved** - Identifying their role (if target of modification or used as data source) as well as their states (usually their identifier)
- **The context of execution for each action/task** - Usually based on the stack frames and allows to describe function calls, their arguments and local variables, etc.

Aside from the advantages given to the programmer by TDs, there are costs that should be mentioned (it depends greatly on the actual TD approach and how much granularity is intended to be displayed to the user):

- Only displays executed instructions
    - For example: instructions within a false evaluated branch block are not presented to the user since they will create visual pollution (their relevance is null, literally)
- Does not follow library calls
- Displaying inner works of the memory is not important (being it stack or heap).
- Execution performance degradation (due to the overhead addition inherent to the data' collection and persistence)

Basically, they do not show information that is too specific and irrelevant to the user. Again, this depends on what the debugger is supposed to achieve. For instance, an assembly language TD (as shown in the

section below) should have the ability to display what happened to registers as well as their values at a given time, where those values came from (more specifically, the memory addresses), etc. In contrast to that, an R timeless debugger does not need that granularity because R is a higher-level language and does not have any reference to that information in its normal usage meaning that the user does not need to view and know information with such specificity.

2.2.1 QEMU Interactive Runtime Analyzer (QIRA)

An example of a binary timeless debugging is the QEMU Interactive Runtime Analyzer (QIRA). QIRA is an open source application, developed by George Hotz [5], that competes with well-known and established applications such as strace (which follows system calls) and gdb. QIRA is a timeless debugger which means that all the information is tracked and collected while the program is running, giving the programmer the power to debug “in the past” which consists of analyzing what was in memory/registers at any given time, during the program runtime (this may be right after execution or at any time after it).
QIRA was initially developed during an internship at Google with the project code name "Project Zero". Hotz decided to develop it because, as a Capture The Flag (CTF) challenges participant, he noticed that binary reversing challenges, which consists in understanding the inner workings of a program without its source code and somehow exploit it, consumed a lot of his time mainly because of the available and commonly used tools (mostly gdb). He noticed this due to two main factors: if he didn’t place (or misplaced) a breakpoint and he needed to rerun everything again or if he just passed a breakpoint unintentionally (and going back in the execution was not feasible). This worsened his performance and cost him points in competitions, making him understand that the approach that is commonly used to debug binaries is outdated and not efficient at all.

With this in mind, Hotz took the paradigm of version control software (VCS) and applied it to binaries. Basically, a VCS, such as git [16], is an architectural layer running on top of the file system and, in a version controlled directory, watches for the entire history of the files within, for the order that the changes are made and for who does those changes. Also, VCSs have functionalities to save those changes to the persistent repository through commits, pushes and pulls in order to sync all the user’s data. Applying this to binaries, every register or memory can be seen as a file in the VCS, every instruction can be translated to a commit (for example, mov rax, 0x0 can be comprehended as a change regarding the value 0x0 being written in the file rax), etc..

Hotz also states that the difficult part, and the most critical, about this type of debugger is the user interface that is implemented to make sense of all the collected data because it is the part that makes the application usable and an improvement over existing methodologies. If a tool makes the programmer’s life harder (by needing to study it and the displayed information), then it is not beneficial and should have never been created.

The advantage of not needing to use the “standard”/”traditional” approach repeatedly (set breakpoints, analyze, re-run, etc.), allowed Hotz to win a second place at a CTF challenge as a one-man army. This

was possible in a major part due to the time leverage/advantage given by the used tool to reverse the binary challenges: QIRA.

3 Architecture

This section describes the structure of the ABD_tool and its relationship with the R interpreter. The main purpose is to identify how this relationship is established, scrutinizing its inner workings, describing the implemented components and how they achieve the desired goals: simplify, enhance and save time to R programmers when debugging their scripts.

3.1 Motivation

It is crucial to remember that the ABD_tool aims to collect information regarding the program being executed and, afterwards, provide an User Interface (UI) that allows the programmer to visualize, understand and identify bugs within his/her code more easily, with a single program run. This potentiates the capability of discovering problems, and their origin, by analyzing any instruction at any desired time with a complete context of their execution.

The programmer, when presented with the UI, has at his/her disposal a directional graph with sets of executed instructions grouped by environments constituting nodes (each node represents a function and its consequent execution). He/she can explore and review any of the executed instructions with an optimized and debugging-oriented UI at any time, the number of times he/she desires.

Initially, the displayer centers the graph view in the starting function but, since the scripts are allowed to contain an unlimited number of functions (thus creating an unlimited number of environments which corresponds to an unlimited number of nodes in the graph), when a function call is issued, a clickable `follow` label appears besides the instruction line of code in order to center the graph in the called environment. The same applies when the function returns and the user wants to focus on the caller function again (a `follow` label appears on the side of the return instruction, as seen in Figure 6).

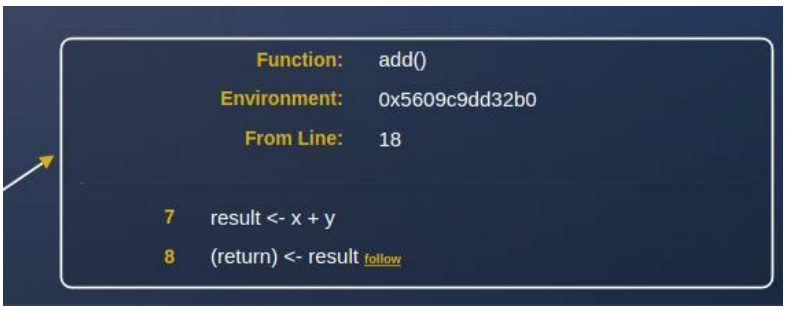


Figure 6 - Graph node for function add()

ABD_tool also allows exploring the program' branches and their conditions. When a branch is present, not only the statement that evaluated to true is displayed but also all the failed branches are shown and are accessible through posterior tested statements. For instance, if a branch has three tested statements and the one that evaluated to true is the final one, the programmer can backtrack from the last to the first in a historic-like approach. Furthermore, what makes this branch analysis so powerful is the fact that the programmer can visualize which portions of the tested statement failed or passed and, in case

arithmetic operations are present in the statement, the values produced are also shown. Another important aspect is that, if the statement contains objects, the user can click their name and access the instruction that led them to that state. This can be seen in Figure 7, which is presenting the branch analysis custom UI.

Figure 7 - Branch analysis custom UI

The ABD_tool implements many more features which have an extended explanation and demonstration later in this chapter (Section 3.4 ABD_tool logging ecosystem), including a detailed description of the process to collect and later display the data.

3.2 Overview

The general architecture consists of two main parts (Figure 8): the ABD_tool' ecosystems and the changes made to the R interpreter. Both of these parts are extremely important because, without the other, one would not achieve the full set of capabilities provided by their combination.

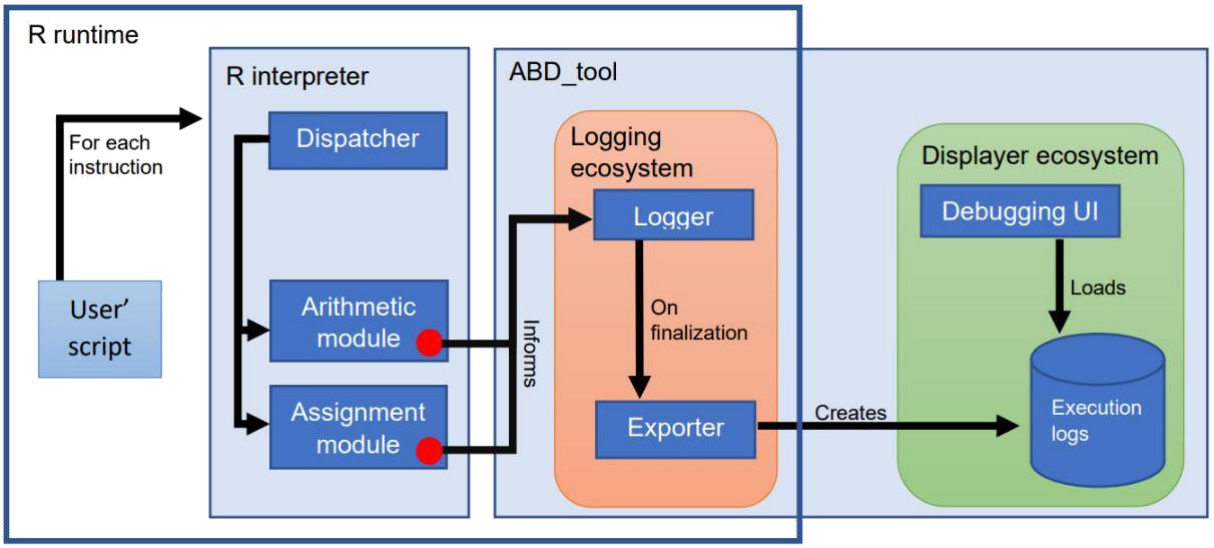


Figure 8 – General architecture overview

The idea behind this architecture is: 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 8, `Informs` label) whenever supported features instructions are executed combined with the data relative to their execution (Figure 8, `Assigns` label), in order to export that data (Figure 8, `On Finalization` and `Creates` label) so that, in a posterior phase, the execution state in which those instructions were executed can be reconstructed (Figure 8, `Loads` label). To achieve this conceptual idea, the R interpreter was modified with method calls to the ABD_tool, which will be called signals (represented by Figure 8 `red dots`).

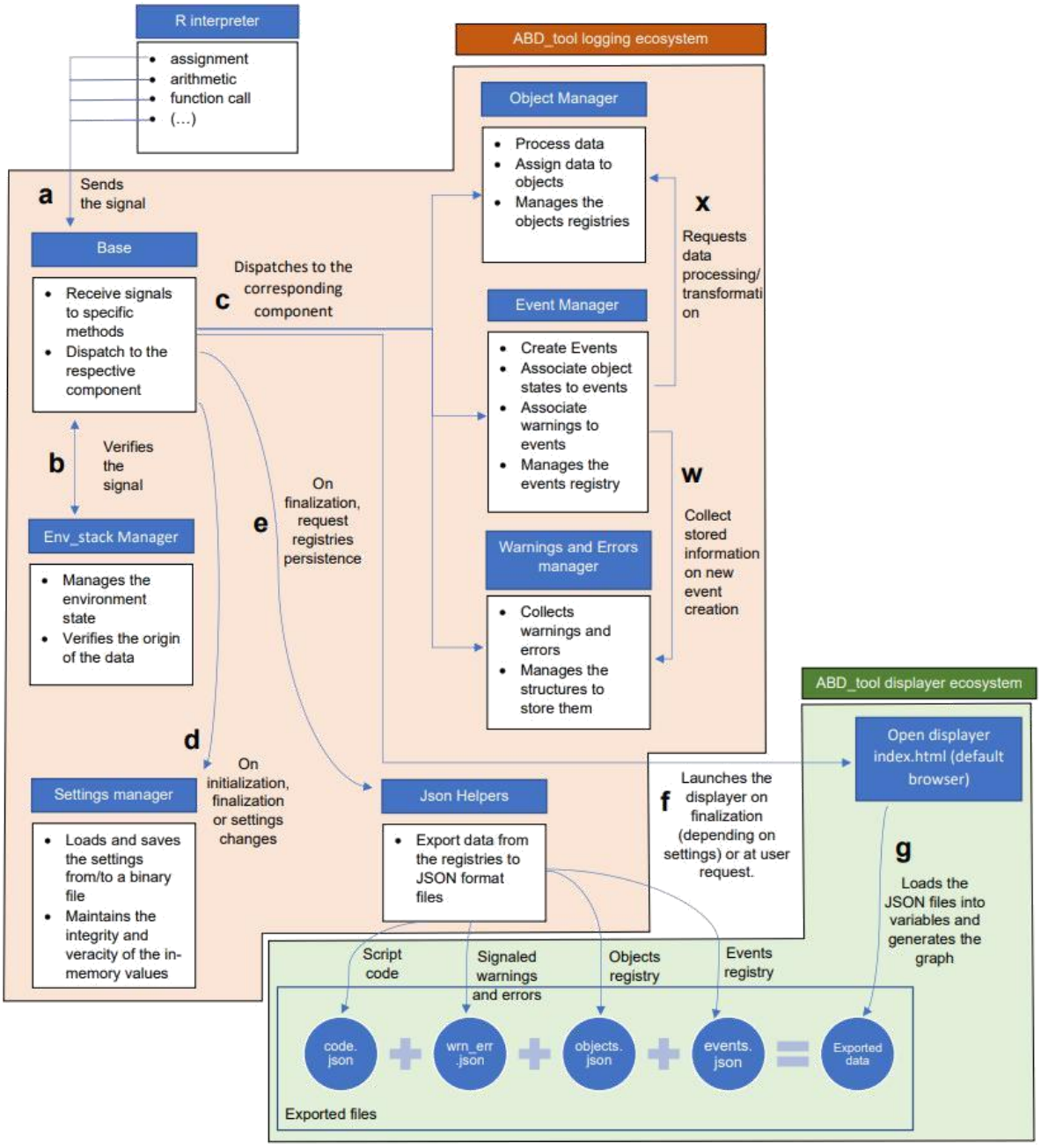


Figure 9 - Architecture overview

The ABD_tool inner workings can be explained with a high-level example:

- The R interpreter process the instructions, reaches the modification and sends the signal (Figure 9, point a).
- The ABD_tool receives the signal and verifies if its context corresponds to the script current execution environment (Figure 9, point b):
    - **If it does not match** - The signal is dropped.
    - **If it matches** – the received data is assigned to the component that handles the designated signal (Figure 9, point c).
- If `stop()` command is issued or an error is signaled:

- The ABD_tool base component will verify the user-defined settings (Figure 9, point d), retrieve the saving path and persist the in-memory structures into files in the defined path (Figure 9, point e).
- If the settings are set to launch the displayer after the execution ends, the ABD_tool will launch the default browser (Figure 9, point f).
    - When the displayer is launched – The exported files are parsed into memory and the visualization is constructed (Figure 9, point g)

3.3 Generating signals in the R interpreter

The R interpreter consists of multiple modules that perform a wide variety of tasks. For example, when an arithmetic operation occurs, the designated methods/components to process that operation are encapsulated in the `arithmetic` module. When errors and/or warnings are signaled, the module that processes them is the `errors` module. For every instruction execution that occurs at a specific place in the interpreter, that module sends a signal to a specific ABD_tool base component method. These signals comprise all the essential information that allows the instruction execution context to be rebuilt.

They 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)** – Given a R instruction, the LHS is the left part of it meaning that, for example, in an assignment, the LHS is the modified/created object. The value sent in the signal is post evaluation.
- **Right-hand side (RHS) (G)** – The RHS is, contrary to the LHS, the right part of an instruction and it can assume a language expression (`LANGSEXP`) or a value (either scalar or not). The value sent in the signal is post evaluation.
- **Environment (RHO) (G)** – The environment represents the execution context in which the signal was sent, making it one of the most important elements. It allows the signal verification (comparing the received context with the user's script current context) and to decide on the signal further 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, as the name suggests, are instruction specific, and so, the signals includes them when created. For example, an arithmetic signal appends the final result for the whole expression, a function call signal includes the arguments, etc.

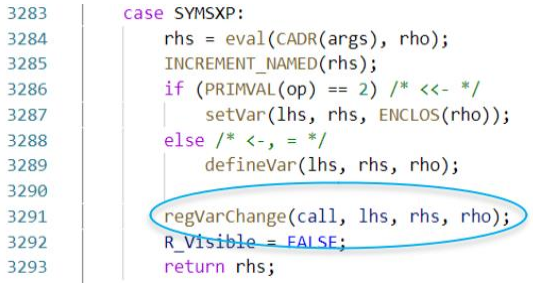


Figure 10 - Object modification in eval.c (ABD signal highlighted)

The placement of the aforementioned signals is very important because if, for some reason, the evaluated value produced by that expression generates an error, and the ABD_tool is notified before that evaluation, there is the possibility that the tool processes erroneous data. If that scenario occurs in an assignment, the signal must not be issued from the call present in Figure 10, but from the one present in the `errors` interpreter module (Figure 11).

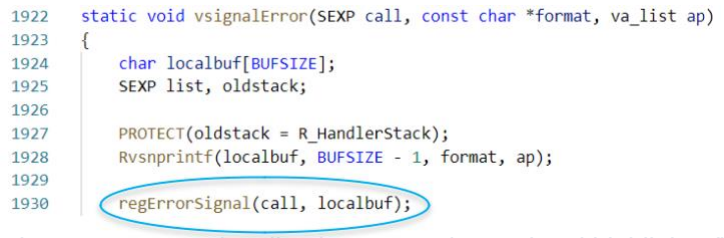


Figure 11 - Error signaling in errors.c (ABD signal highlighted)

Although some exceptions are present in the tool implementation, the majority of the calls to the ABD_tool base component are done after the instructions are successfully processed and applied by R to the corresponding variables/environment avoiding the persistence of erroneous data and, consequently, the display of misinformation to the user by the UI. Those exceptions exist due to the fact that, identifying the exact method that performs some action (for example: process a data frame cell change with empty indices, which uses the entirety of the data structure) is not always possible because R relies on external loaded libraries to perform multiple actions which makes the execution flow unidentifiable and untraceable and thus, its modification impossible.

To mitigate this, two signals must be issued by the interpreter in order to achieve the overall desired functionalities of the tool. The modifications, contrary to the aforementioned one-line addition (to send the signals from the R interpreter to the ABD_tool base component), must comprise two signals:

- First: notifies that some instruction is going to be executed (pre-evaluation)
- Second: notifies the result of that instruction (post-evaluation)

The first signal allows the ABD_tool to collect information relying on the call LISP list (basically parsing part of it, not the entire list) and to store it in temporary structures. Upon receiving the result (the post-evaluation signal), if the calls lists are the same match (by memory addresses comparison) the result is processed into the designated structures (if needed). Otherwise the temporary variables are reset, and the second signal is discarded.

3.4 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 have one characteristic that must be detailed in order to clarify how they are, in fact, processed: their nature/behavior. Since they are intended to describe actions that took place in the R core, they can assume one of two types:

- **Program state modifier**: Signals that, when issued, will modify the program state by modifying an object, modifying the current execution environment when a function is called, etc. For example, an object assignment - it takes any evaluated value and sets the target object current value to it.
- **Informative**: Some signals will not modify the program state by themselves, instead, they act as a data source to other signal. For instance, an arithmetic signal will not modify anything in the whole environment but the signal that uses the arithmetic instruction evaluated value will (normally an assignment, an index change, etc.).

This behavior is important because the signals, when reaching the ABD_tool endpoint, will be processed and transformed into well-known and well-defined structures named events and thus, what they are and do, is critical in this transformation process. Only one component is responsible for the literal creation of events (`Event manager`) but it communicates with other components, depending on the event it is creating (Figure 9 point x and w), to request data (already processed and stored) or to process new data (transforming it to known structures).

The components in the ABD_tool are named according to their functionality, as shown in Table 4, for a better process to component association. For instance, when processing and assigning data to an object, the work is done/assigned by/to the `Object manager` component. When creating and registering the events, the `Event manager` is the component processing it. This granularity allows a better understanding of the code by a programmer that wants to install and/or extend the tool functionalities (since it will be open source).

Component	Functionality
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 with the objects and their states. 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 association to be made). The event manager also manages an events registry that stores all the created events in a list format.
Events	Contains the definitions and declarations for all the events that can be created by the `Event manager` component. In summary, it describes the supported R language features by ABD_tool.
Warnings and errors manager	Stores warnings and errors that were signaled to the ABD_tool and has methods to store, get and clear the currently stored signals.
Environment stack manager	Its functionalities are meant to prevent the ABD_tool to store data from different execution contexts by verifying if the context of the signals correspond to the current script context (discarding mismatches).
Settings manager	The programmer can modify how the tool performs regarding multiple aspects. For example, he/she can enable the verbose mode, make the tool launch the UI at stop, and much more. These settings need to be saved and that is the purpose of this component. It also verifies paths validity and checks the settings' integrity.
JSON helpers	Exports the data, from the memory stored structures, to human readable files that are then used by the UI to display the information. This component translates/transforms all the data (the structures) to information. Similar to an ETL process, this is the middleware that ensures the transform phase.

Table 4 – Components' functionalities

3.5 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 the it meaning that, changing or replacing it with a newer technology (or a completely new implementation), will not affect the way that the ABD_tool collects and processes the signals that arrive from the R interpreter. 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.

3.5.1 Initialization

When initializing, the ABD_tool' displayer loads the exported data into memory, which resulted from the core component `JSON helpers` exporting phase (Figure 9, point e), and processes that information

based on the created events. The displayer component will process the events based on the events file while the remaining files play a supporting role meaning that, the only structure that is actively iterated over is the one stored in the events file while the others (code, wrn_err and objects), are accessed on-demand.

This is important because if, for example, during the script execution a huge data frame was created and several modifications were done to it, and the user does not intend to visualize them, the current value of this data frame does not need to be resolved on the displayer launch. Thus, it is assumed a lazy-evaluation approach in order to reduce the displayer spawn time while it increases its responsiveness because memory is not consumed by non-used data.

3.5.2 Graph generation procedure

Using the events file created from the processing phase, the displayer generates a graph based on the environments that it encounters, which means that, for every function call that occurred (and, thus, an environment creation), the displayer will create a node represented by the environment memory address. Meanwhile, while it creates the nodes, it also creates directional edges to identify that, at a certain point in time during the execution of that environment, one line of code resulted in another environment creation.

The nodes' contents are representative of the actual executed R code and are identified by the line where they effectively are in the script (to allow the programmer to look at the script and understand what piece of code he/she is looking at more easily). 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.

3.5.3 Displayer capabilities

The displayer is what will give value to the timeless debug paradigm implemented by the ABD_tool, making its functionalities/capabilities the most important aspect of this work mainly because they are what the user will effectively experience during the tool's usage. These functionalities are scrutinized and exemplified during the remaining of this section in the following format:

- Descriptive (Table 5)
- Demonstrative (Section 3.5.4)

Language Features	**Displayer capabilities**

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 instructions 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, the results for each evaluated portion of it and object links (to visualize their state when they were used in the branch). • If it applies, the previous failed statements (regarding the branch as a whole) and a `view` link to load those statements into analysis.
Assignment	Visualization of the assignments consists of: • General object information (name, data type, data structure). • New values information (origin, size, values). • List of prior events (history) in which the modified was also modified.
Assignment from events	If the origin of the assignments' data is an event the node content corresponds to two labels: ○ **Left-hand side:** assumes what was described above. ○ **Right-hand side:** assumes a link to the event which originated the new values (arithmetic, function call, etc.).
Index/cell changes	Custom visualization of the modified indexes/cells of a given object: • General object information (name, final values) • Applied changes information ○ **Indexes:** number of changes ○ **Cells:** number of rows and columns • New values source information (object name with a link to view its when it was used) • Changes breakdown ○ Possibility to change the visualization window and its size ○ Navigation through the defined window (using directional arrows) **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-mentioned "Assignments from events".
Looping	Depending on the loop used, the user can visualize: • The condition used in the branch (while loops) • The expected and the effective number of iterations (for loops)

	• The executed instructions per iteration, with navigation capabilities to access the desired iteration
Conditions	**Warnings:** Symbol reminiscing a warning road sign, appended to the instructions (on the side). Their messages are 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, as the warnings, their messages inside a tooltip.
Arithmetic operations	The arithmetic operations visualization follows the same structure as the branch analysis one.
Data visualization	The objects data visualization uses the same methodology of the index/cell changes feature. The only difference is that it depends on the size of the object data structure: • **Less than 5 elements:** The values are shown in-place • **Between 5 and 15:** A tooltip is available with the values in normal ordering (for one-dimensional objects) • **More than 15:** The values are shown with the format and visualization capabilities as described in the index/cell changes displayer capabilities.

Table 5 - Displayer capabilities by language features

3.5.4 Displayer capabilities usage example

To give a better visualization regarding the ABD_tool functionalities, and how it improves the debugging capabilities of its users, this section presents a small script, as an example, in a walkthrough fashion in order to showcase the provided features when using the displayer.

a) The script

The script (Figure 12), starts by initiating the tool (at line 6) followed by two function declarations (`add_two()` and `sub_two()`, at line 7 and 11 respectively). Proceeding, five objects are created where three of them (`obj1`, `obj2` and `obj3`) are single-indexed vectors and the fourth (`vec1`) and fifth (`prices`) are vectors with 50 and 10 indexes, respectively. The `prices` vector is also modified at line 22, with the new values originating from the `vec1` vector.

Then, a two-column data frame is created where (`df1`):

- The first is an in-place declared (hard-coded) vector.
- The second is the previously declared `prices` vector.

After the data frame creation some branches, containing arithmetic operations in their statements, are performed. An object named `result` is created and initialized to 0 (line 36).

Also, a for loop is performed over the previously defined prices vector which performs some branching with the values that are being iterated over, in order to modify the loop's natural execution sequence. To conclude the script, one more object is created (`obj4`) containing a `STRSXP` that is later passed as argument to the `add_two()` function (which expects only numeric values and, thus, throws an error). After that, the ABD_tool is stopped (line 50).

```
1  options("keep.source"=TRUE) #options modification
2  run <- function(){ #wrapper function
3      abd_stop(0) #change abd settings
4      abd_verbose(0) #change verbose
5
6      abd_start() #ABD_tool startup
7      add_two <- function(x, y){ #function declaration
8          result <- x + y + 2 #arith & assignment
9          result #explicit return
10     }
11     sub_two <- function(x, y){ #function declaration
12         result <- x - y - 2 #arith & assignment
13         result #explicit return
14     }
15
16     obj1 <- 10.9 #REALSXP obj declaration
17     obj2 <- 15 #INTSXP obj declaration
18     obj3 <- 13 #INTSXP obj declaration
19     vec1 <- 1:50 #INTSXP vector object declaration
20
21     prices <- 10:30 #INTSXP vector object declaration
22     prices[1:15] <- vec1[10:30] #vector index changes (throws warning)
23
24     #two-col data frame creation
25     df1 <- data.frame("Quantity" = 200:220, "Price" = prices)
26
27     if((obj1 + 2) < obj3){ #branch with arith
28         obj1 <- add_two(obj1, obj3) #add_two call and return assignment
29     }else if((obj2 + obj3 + 2) < vec[30]){ #branch with arith
30         obj3 <- sub_two(obj1, obj2) #sub_two call and return assignment
31     }else{
32         obj1 <- add_two(obj2, obj3) #add_two call and return assignment
33         obj2 <- sub_two(obj3, obj1) #sub_two call and return assignment
34     }
35
36     result <- 0 #INTSXP obj declaration
37     for(val in prices){ #for-loop over INTSXP vector
38         if(val < 12){ # branch
39             next #loop iteration skip
40         }else if(val < 15){ #branch
41             result <- val * 2 #assignment & arith
42         }else{
43             break #loop exit
44         }
45     }
46
47     obj4 <- "Object 4 example string" #STRSXP obj declaration
48     add_two(obj4, obj2) #add_two call (throws error)
49
50     abd_stop() #ABD_tool shutdown
51 }
52 run()
53
```

Figure 12 - Example script

b) The displayer

After running the script, the generated execution graph (Figure 13) consists of 4 nodes:

- The `main()` function node.
- Two `add_two()'s` function node.
- The `sub_two()'s` node.

To demonstrate the displayer capabilities, the graph usage and analysis will follow the script execution natural sequence.

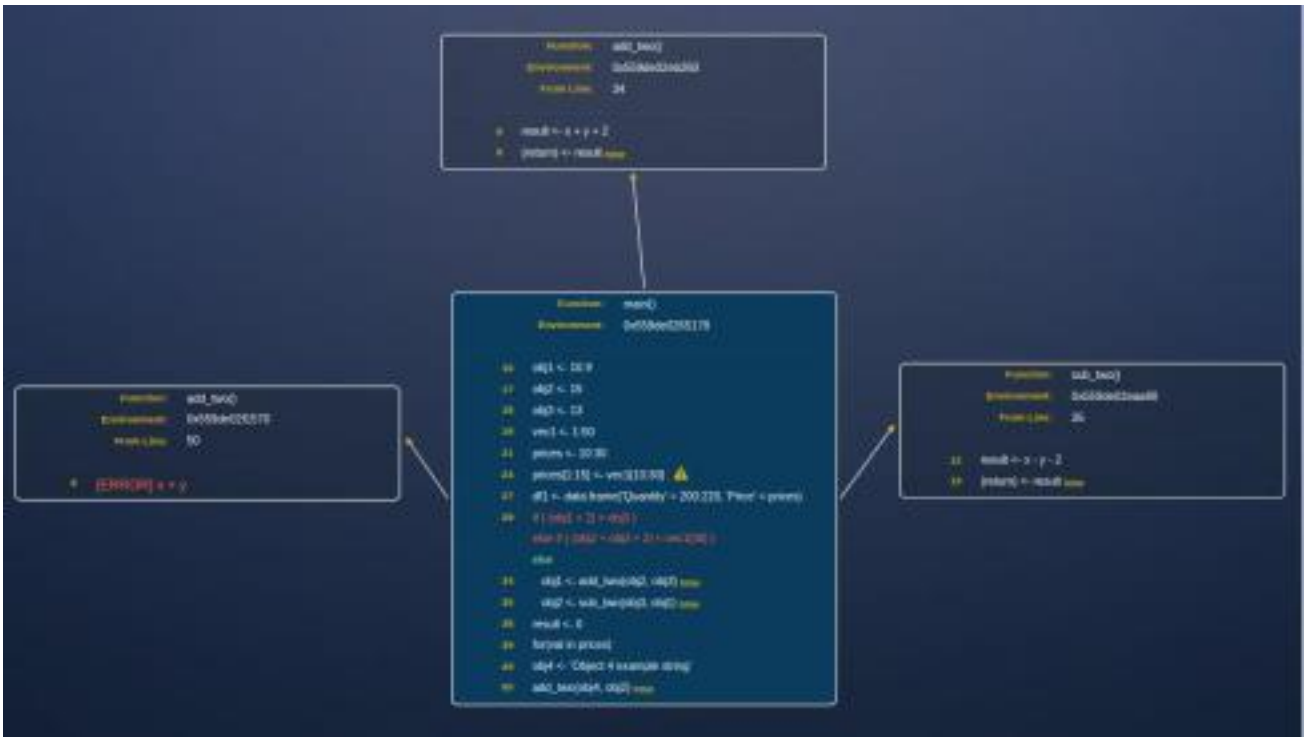


Figure 13 - Example script generated graph

As described before, the script starts (after declaring the functions) by creating some objects, either single-indexed vectors or multiple-indexed vectors.

If the programmer clicks on one of those declared objects labels, either one of the windows in Figure 14 is shown (depending on the object):

- Single-indexed vector (Figure 14, left side window):
  - The values are shown in place.
  - The new values origin, since they were declared in the script, is labelled as `User-Typed`.
    - The same occurs when the source is a non-implemented feature or a function which is not user-defined.
- Multiple-indexed vector (Figure 14, right side window):
  - The values extrapolate the bounds for in-place (size 5) and tooltip (size 15) visualization so they can be seen through the custom visualization window (accessible by clicking the `Values` label).
  - Their origin, and since the object was created with a vector creation mechanism, it is set to `Vector creation`
- Size-independent information:
  - The size of the vector is displayed (1 and 50 in this case)
  - The data type as well as the data structure is shown

  The object name and the line the assignment occurred
  - The history of the object (in this case, an empty history record)

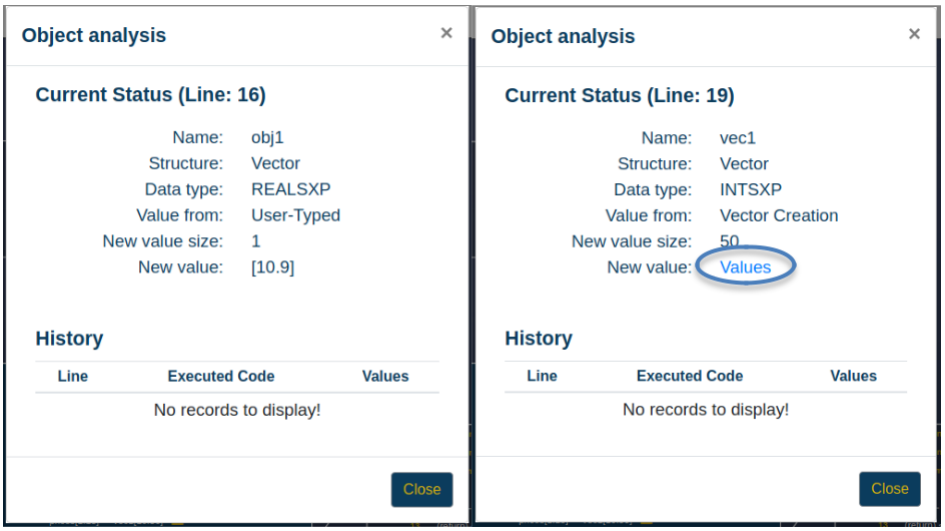


Figure 14 - Example script object analysis windows

If the `Values` label is followed (Figure 14 - right side window, ellipse form), the user can visualize multiple things regarding the object in analysis (as seen in Figure 15):

- The object size's
- The values it holds

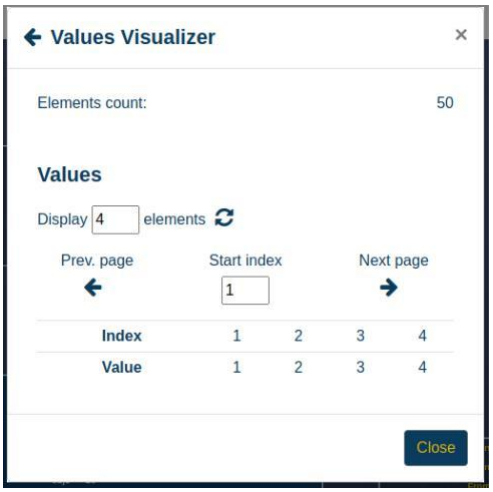


Figure 15 - Example script values visualizer window

By default, the number of shown elements is set to 4 and starts from the very first index. The user has the capability of changing these values as he intends. Furthermore, it is possible to navigate through the values using the pointing arrows.

Since one of the ABD_tool's main goal is simplify and improve the debugging process, the user can use the "go back" arrow (at the upper left corner) to return to the window that lead him to the current window being displayed. This avoids having to repeat the process to achieve something that he/she already had previously.

At line 24, when the prices vector is modified, the presence of a different element in the UI is visible. This new element indicates that a warning was generated in this exact line (Figure 16).

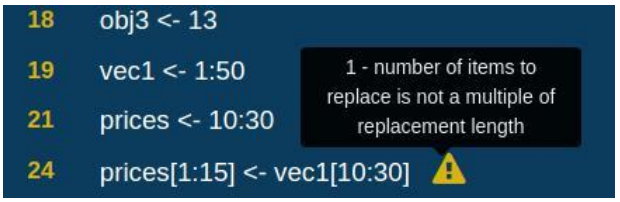


Figure 16 - Example script line 24 warning

Analyzing this event (Figure 17), the user has at his/her disposal several information starting by general information regarding the event, such as:

- Target object (the object that suffered the modification)
- The number of changes
- The source of the data as a clickable label visualize its state at the moment of its usage, if the source object is a user-defined object (in this case, `vec1` object)
- A `values` label, to the target object final values (after the modifications, as showcased before).

Below this initial section, the `Changes breakdown` section displays to the user the mapping between the source and the target index as well as the actual value. This section can be used as described before, in the vector visualization portion.

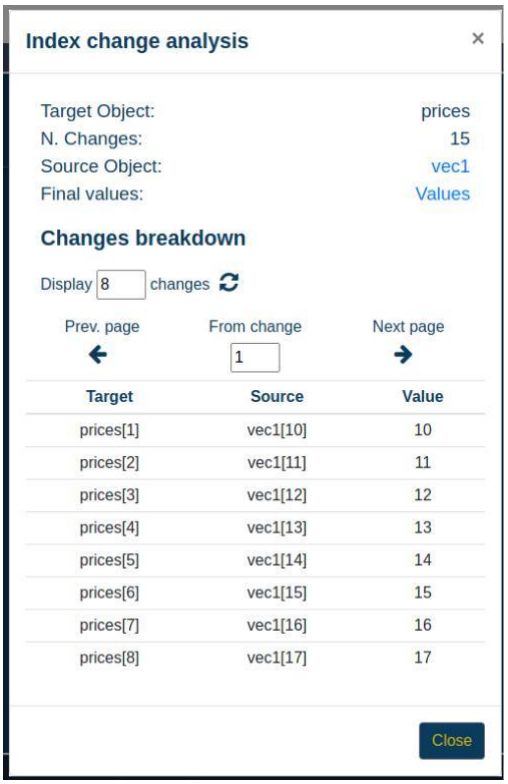


Figure 17 - Example script index change analysis window

The next event available (script line 27) relies on a data frame creation consisting of two columns:

- **Quantity** – An in-place declared vector with 21 positions from 200 until 220 (both bounds are inclusive)
- **Price** – Uses the values from the previously declared vector `prices`

This event, contrary to the ones analyzed before, generates two labels (the left-hand and the right-hand sides of the operation):

- **Left-hand side** – a clickable label to use the `Object analysis` window on the created data frame (`df1`) object
- **Right-hand side** – a clickable label to the `Data frame creation` window (Figure 18)

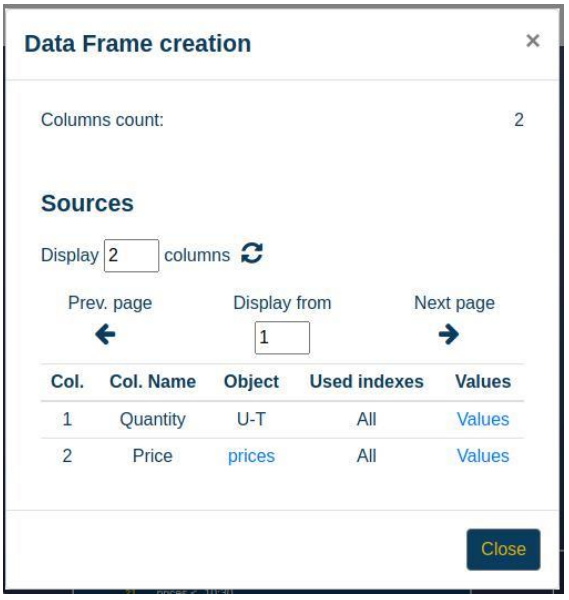


Figure 18 - Example script data frame creation window

At this window the user can visualize where the values, used to create the data frame, originated from, having the ability to control how many and from which the visualization window displays. If any of the columns sources are objects, the `Data frame creation` window offers the user the same functionalities as in the other windows: use the label to visualize the object at the state which it was used to create the data frame column. Complementing this, the values are also available through the `Values` label (which depends on the size of the vector) for quick access. In this case it was not applicable but, if only a portion of the `prices` vector was used, the used indexes would also be displayed (through the same mechanism as the `Values` label).

Following the data frame creation, a branch is present (Figure 19). For this, the user can visualize in a quick form which statements failed and which passed, without the need to open the `Branch analysis` window for each one of them. To easily visualize what code was executed under the true-evaluated statement, the instructions are indented depending on the depth of the branch (nested statements will increase the indentation incrementally). The branch is displayed as without the line number for each statement because, even them being in different script lines, they constitute an integrated block.

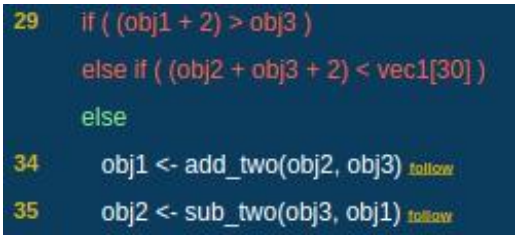


Figure 19 - Example script branch display

By clicking a branch statement label, the user is presented with a window named `Branch analysis` which is divided in three sections:

- **Top section**
  - The condition being tested is displayed.
  - The global result of the expression.
- **Middle section** (Condition breakdown)
  - The statements are displayed individually, as well as their relationship.
  - The values used in the evaluation are shown inside parenthesis.
    - If it is an object, the ABD_tool offers a label to jump to the state that object was in that moment.
  - The results for each individual statement.
  - The operator used in each individual statement.
- **Bottom section** (Current branch failed conditions)
  - The previous failed conditions in the branch block with chronological order.
    - The first entry was the first false-evaluated statement.
  - Each entry offers to the user a clickable label to visualize the `Branch analysis` window for that particular statement.

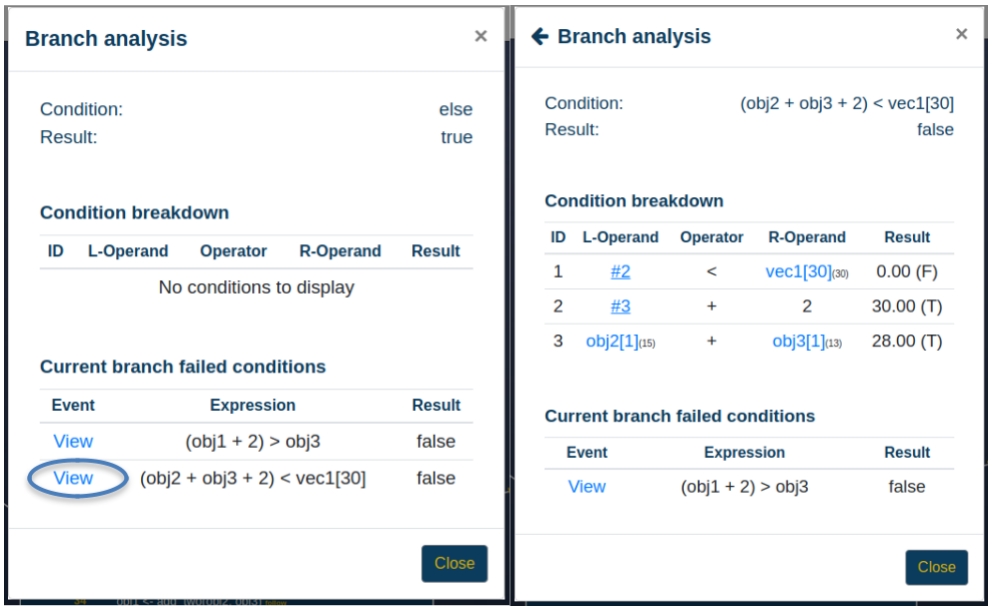


Figure 20 - Example script else/else if clause analysis

If we analyze the true-evaluated statement, in this case, the else clause (Figure 20, left window), these sections can be visualized. Since the else clause does not contain any statements, the only relevant information that can be extracted from this is the previous failed conditions.

If the second `View` label is clicked on the `Current branch failed conditions` records, the `else if` clause entry, its `Condition breakdown` can be visualized (Figure 20, right window), with the content:

- **ID #1** - The final statement to be evaluated (which determines the branch global result)
  - Verifies if the statement with ID #2 is smaller than the value, at the 30th index, of `vec1`.
    - By the object label's side is the value for that specific index (30).
- **ID #2** - The statement performs an arithmetic operation
- **ID #3** – The `obj3`'s first index is added to the `obj2`'s first index.
  - Both values, for those indexes, are by the object's sides.

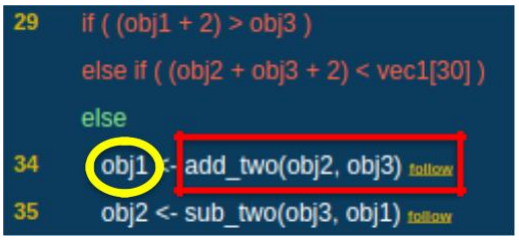


Figure 21 - Example script LHS and RHS showcase

As seen in Figure 19, two functions were called after the branch evaluated to true, at the else clause. These events, the calling of the function and the assignment of the returned value to the `obj1` and `obj2` (line 34 and 35 respectively), generate two labels:

- **Left-hand side** (Figure 21, **yellow** circle)– clickable label to the, previously seen, `Object analysis` window
- **Right-hand side** (Figure 21, **red** rectangle)– clickable label to the `Function call analysis` window (Figure 22).

Another important aspect is the presence of the `follow` clickable label (by the function call label's side), which will center the graph (when clicked) in the node that corresponds to function call execution environment.

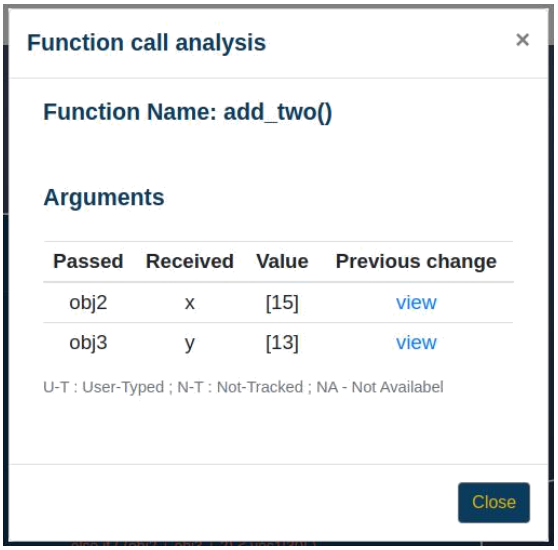


Figure 22 - Example script function call analysis window

At the `Function call analysis` window the user has at his disposal the mappings between the values passed to the function and which object received those values. The `Object analysis` window, for the objects used as arguments, is also reachable from the `previous change` column `view` labels (when applicable).

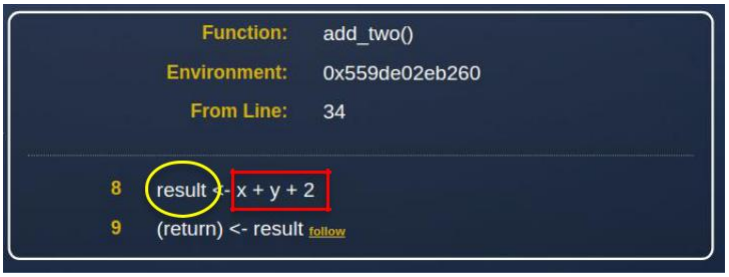


Figure 23 - Example script add_two() generated node

From this point on, the script called the `add_two()` function, which sums the received arguments and adds 2 to the value that resulted from that sum. This will generate two labels:

- **Left-hand side** (Figure 23, **yellow** circle) – clickable label to the `Object analysis` window for the object `result`
- **Right-hand side** (Figure 23, **red** rectangle)– clickable label to the `Arithmetic analysis` window.
    - The `Arithmetic analysis` window (Figure 24), provides the same functionalities as the `Branch analysis` window but does not provide the bottom section, since it does not apply to this case.

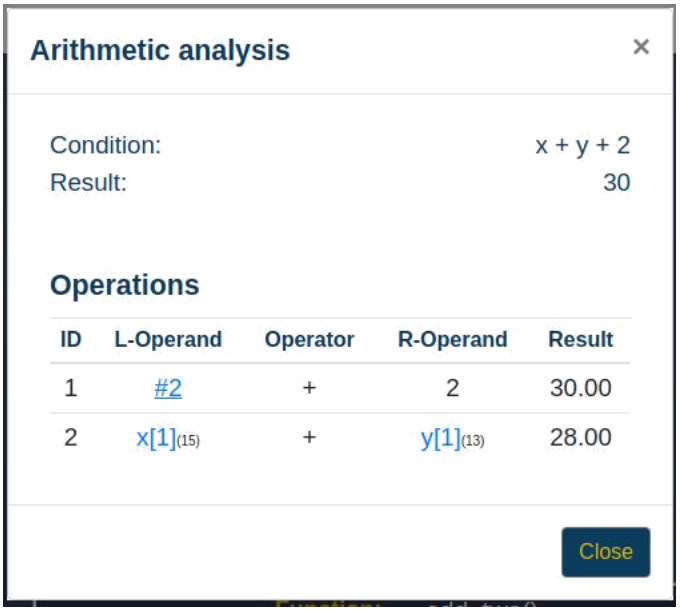


Figure 24 - Example script add_two() arithmetic analysis window

After performing the arithmetic and the assignment, the function returns and, also, generates a label for that event. In this case, it only generates one label and provides to the programmer a `Return analysis` window (Figure 25) which contains:

- The name of the returning function.
- The returned values (which may assume multiple forms such as the `Values` label, a tooltip or in-place values, as described before).
- The object that received the returned value.
    - **If applicable** (the return value was stored in some object) – a clickable label to visualize the object state in the `Object analysis` window is available
    - **If not applicable** – indicates that the value was not stored.

As part of the node generated label, another `follow` clickable label is appended to the return event in order to facilitate the re-contextualization to the caller environment, by centering the graph on it (this becomes more useful when the number of nodes is higher).

Figure 25 - Example script add_two() return analysis window

Post return, the `main()` function declares an object named `result` and initiates a for-loop, right after that object definition. The loop iterates over the previously declared `prices` vector object and provides a `For-loop analysis` window (Figure 26).

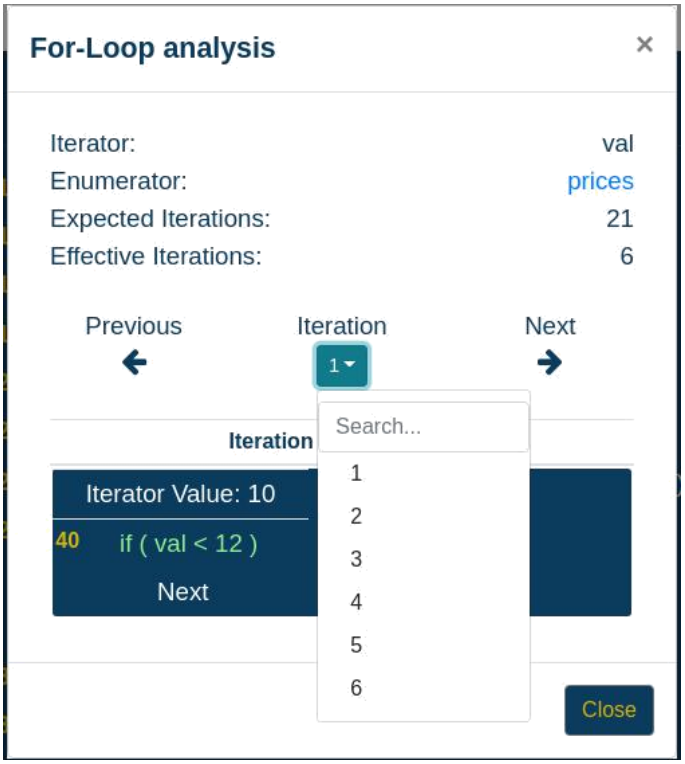


Figure 26 - Example script For-loop analysis window

At this window (Figure 26), the user has at his/her disposal 3 sections:

- **Top section** - Miscellaneous information regarding the whole loop execution (depends on the loop)
  - The iterator
  - The enumerator (with clickable label to access the `Object analysis` window for the current enumerator state)
  - The expected and the effective iterations
    - Which indicates beforehand if the loop found a break event or an error leading its execution to halt.
- **Middle section** - Navigation
  - Pointing arrows to navigate through the loop iterations
  - Iteration indicator with a search field (to jump to a desired instruction more easily)
- **Bottom section** – Executed instructions
  - This section provides to the user interactable content with the same functionalities, as the ones found in the nodes' content, for the code that each iteration executed.

For other types of loop, the concept and structure are the same, the only change being the miscellaneous information that is displayed with the `While-Loop analysis` and `Repeat-Loop analysis` windows (Figure 27).

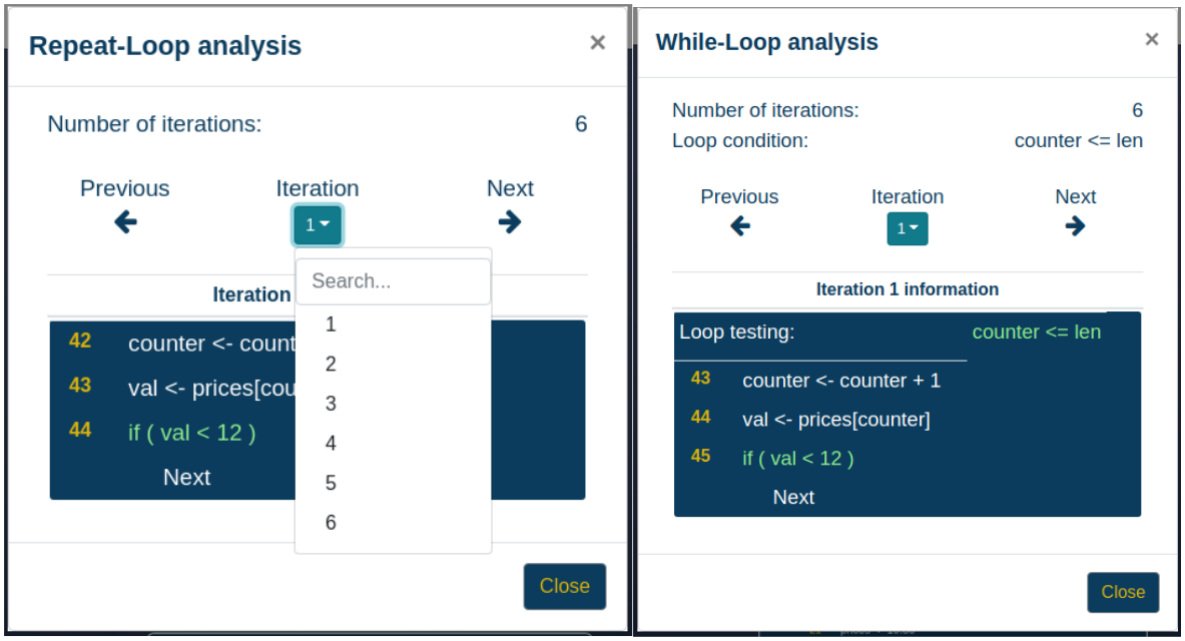


Figure 27 - Example script Repeat and While loops implementations

To finalize the script analysis through the ABD_tool displayer, an object (`obj4`, at line 49) holding a `STRSXP` value is defined and later passed to the `add_two()` function (at line 50). Since the `add_two()` function is expecting two numeric objects (in order to perform the arithmetic seen before), and the `obj4` is passed, an error is signaled and displayed at the node corresponding to the line 50 function call environment.

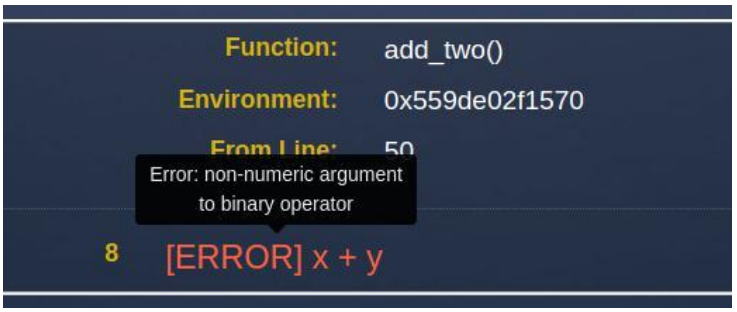


Figure 28 - Example script add_two() call error message

Since the environment throws an error in its first line, the only thing available to analyze in it is the error message, which is presented to the user using a tooltip and an attached prefix (`[ERROR]`) to the statement which signaled the error (Figure 28).

4 Implementation

This chapter covers the technologies used in the ABD_tool's implementation. Due to the nature of the work (somewhat similar to instrumentation, which introduces huge overheads), some optimizations were implemented to alleviate the overheads and thus they are also described in this chapter.

The structures and components that will be described and scrutinized in this chapter are available in the project repository, on GitHub. On the ABD_tool's repository, the progress, the modifications and the improvements made during the tool development cycle can also be seen (through the commit messages and their contents).

4.1 Employed technologies

R provides Dynamically Linked Libraries (DLLs) [17] to access the data that its interpreter currently has in memory from languages like JAVA and C# but, as stated in the documentation and by some users that experimented with them, the performance was extremely poor, which, in the context of this work, is not something that can be considered as acceptable. As the R interpreter is mainly written in C, and some portions of it in C++, the ABD_tool (regarding its collection and exportation functionalities) is implemented using the C language. There are advantages and disadvantages regarding this choice.

The C language has some huge advantages that suit this project perfectly. Its performance is extremely good because it runs on a considerable lower level to hardware (not as close as assembly, but much closer than JAVA, for example), the memory manipulation is entirely done by the programmer (which allows for improvements in how the structures can be designed and created/allocated).

In contrast to that, C does not provide any type of object-oriented functionalities to abstract the used structures leading to larger code bases, which introduces complexity. Also, its total memory manipulation advantage can be considered a disadvantage because the chance of memory leaks and errors is prone to be exponentially higher.

To compile and link the ABD_tool components with the R interpreter, ABD_tool takes advantage of the R `make` process simply by placing the files in any interpreter directory and importing the base component at the desired R interpreter files/components. This approach worked flawlessly because the `make` process identified that those modified files have a link to the base component, leading to its consequent inclusion and further compilation and linkage process (to generate the binaries) of the interpreter. Since the base component has access to all the other components, those are also included and linked to the interpreter by the same mechanism.

Regarding the displayer portion of the developed tool, initially (in a design phase) the Shiny R package was considered since it can access R objects directly. Nevertheless, after further discussion, the decision led towards a pure web development stack in order to have a better cross-platform support, a better lifetime for the tool (because those are established technologies and not an emerging one, as

Shiny is), better and larger documentation and also greater performance. Another important factor when deciding to move over to the current software stack was the stronger support for graph visualization libraries.

The technological stack chosen for the ABD_tool displayer was:

- HTML [18] [19]: Creates the structure and the visuals for the debugging UI.
- JavaScript [20] and jQuery [21]: Provide dynamic functionalities to the UI components.
- CSS [19] and Bootstrap 4 [22]: Enhances the visuals and its elements' responsiveness

As for the browser, the tool was developed and tested using Chromium's base since it is widely used, translating in a wide variety of supported browsers. To generate the execution graph, a JavaScript open source library named D3.js [23] was used, not only to generate the graph but also to have zoom and view scaling features in it, which are very important functionalities when the graph is considerably bigger.

4.2 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 nor data structures. The main focus was to develop the ones that would show the potential and the capabilities of the architected approach for the debugging problem.

Feature	Implemented functionalities
Data structures	● Vectors (single dimension) ● Data Frames (multiple dimensions)
Data types	● Integer - `INTSXP` ● Real - `REALSXP` ● Strings - `CHARSXP`/`STRSXP` ● Closures - `CLOSXP` (functions) ● Symbols - `SYMSXP` (objects)
Data structures operations	● Distinction between object assignment and modification ● Index (single dimension) and cell (multiple dimensions) changes with data precedence linkage ○ From complete/partial object ○ Hardcoded values ○ From events (arithmetic, return, etc.). ● Comprehension ● Object type changes
Branches	● Without expression length and depth limitations ● Without nesting depth limitations

Looping	• All available loops: ○ For, Repeat and While loops • Capability for multiple loop types nesting • Theoretically, unlimited loop depth nesting
Function calls	• Local variables • Returns can be associated to objects or part of them (indexes/cells) • Environments/contexts (besides the one used by R, to aid on a namespace-like architecture when processing assignment events that require objects in a certain environment)
Warnings collection	• Warnings to events association allowing multiple warnings per event and multiple events with warnings
Error detection	• Error detection and association to the respective instruction and line of code • Persistence when R halts, to avoid data loss.

Table 6 - ABD_tool implemented features

4.3 Events

As stated before, the available events describe, in a concise and well-defined form, what R language features the ABD_tool supports and, this way, an event describes an action performed by the R interpreter. For example, an arithmetic operation performed by the R interpreter, generates an `ARITH_EVENT`.

These events are maintained in memory, in a registry named `eventsRegistry`, until the data export phase. This `eventsRegistry` structure consists of multiple `ABD_EVENT's` structures connected in a single linked list form. To use this registry, a pointer to its last structure is maintained in order to have the capability of only appending to the last stored event and to persist the events from the first to the last. This is important because, if only the last event pointer was saved, the entire list would need to be traversed, until the beginning, in order to persist the events chronologically.

Each element of the `eventsRegistry` also maintains a structure that allows the ABD_tool to associate warnings to events. This is possible because the events are processed after the code is evaluated and applied and thus, the warnings are already stored when a signal to create a new event arises, making the association between events and warnings reliable and straightforward.

```
struct abd_event
{
    int id;
    int scriptLn;
    ABD_EVENT_TYPE type;
    ABD_WARNINGS * warns;
    short branchDepth;
    union
    {
        ABD_IF_EVENT *if_event;
        ABD_FUNC_EVENT *func_event;
        ABD_RET_EVENT *ret_event;
        ABD_ASSIGN_EVENT *asgn_event;
        ABD_ARITH_EVENT *arith_event;
        ABD_VEC_EVENT *vec_event;
        ABD_IDX_CHANGE_EVENT *idx_event;
        ABD_FOR_LOOP_EVENT *for_loop_event;
        ABD_REPEAT_LOOP_EVENT *repeat_loop_event;
        ABD_WHILE_LOOP_EVENT *while_loop_event;
        ABD_FRAME_EVENT *data_frame_event;
        ABD_CELL_CHANGE_EVENT * cell_change_event;
    } data;
    ABD_OBJECT *atFunc;
    SEXP env;
    struct abd_event *nextEvent;
};
```

Figure 29 - ABD_EVENT structure

The `ABD_EVENT` structure (Figure 29) encapsulates the actual event in a union data structure (variable `data`) and uses a variable `type` to identify which pointer of that data union is in use. The `type` variable is assigned based on an enumerator variable that declares all the event types (`ABD_EVENT_TYPES`) available. This approach emulates and gives an object-oriented style to the `ABD_EVENT` structure.

Each `ABD_EVENT` detains a pointer (`env`) to the environment where it was created, as well as a pointer to the function (`atFunc`) which was referenced as the current function in the `Environment stack manager` component. It also stores the script line (`scriptLn`) in which the event is being created at and the branch depth (`branchDepth`) (this is only important for the displayer portion, in order to indent labels accordingly).

The enumeration type that defines the event types is constituted by:

ABD_EVENT_TYPE	Description
MAIN_EVENT	A general-purpose event used to initialize the events registry.
IF_EVENT	Used to describe an if statement.
FUNC_EVENT	Created when a function call is performed. This only records calls to functions declared by the program in his code, to avoid following libraries.
RET_EVENT	When a user declared function returns, a `RET_EVENT` is created.

	This event 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 or an object redefinition occurs.
`ARITH_EVENT`	Every time an arithmetic operation is processed, to be used as a new value for an object or inside a branch statement, a new `ARITH_EVENT` is created.
`VEC_EVENT`	Used when an object is assigned from a new vector (hardcoded vector declaration).
`IDX_EVENT`	Every time a vector index is modified, an `IDX_EVENT` will be created specifying the object that received the value as well as the new state of the object and the new value origin (object, hard-coded, etc.).
`FOR_EVENT, REPEAT_EVENT, WHILE_EVENT`	Represents the occurrence of a loop: a for, repeat or while loop. All maintain a list of iterations, each containing a list of events that occurred during that iteration.
`NEXT_EVENT, BREAK_EVENT`	The next and break events exist to determine when a loop interrupted their current iteration execution either by interrupting (break) or skipping it (next). The break event can also be part of an if statement.
`FRAME_EVENT`	Registers when a new data frame is created. This event also stores information (like the assignment event) regarding the source of the new data frame values.
`CELL_EVENT`	As a `FRAME_EVENT` is for the `ASGN_EVENT`, the `CELL_EVENT` is for the `IDX_EVENT`. It stores information regarding a cell modification in a multidimensional object (in this case, only supporting data frames).

Table 7 - Events and their applicability

4.4 Components details

It is important to detail some key aspects of the ABD_tool' inner workings in order to understand why some components exist and to relate them, and their importance, to the overall working of the tool.

4.4.1 Environment stack manager (env_stack.h)

The `Environment stack manager` is the critical component to ensures the correctness of the tool. This component has such an important role because many actions trigger underlying mechanisms that make calls to the ABD_tool base component when they are not part of the user code.

For example, when an object value is modified (or the object is created), the R method that will handle that call is called `do_set` (in the eval.c file). This method is also where the R interpreter signals a new object change to the ABD_tool base component indicating that an object was created or modified. Although this seems a perfectly working mechanism, internally the R interpreter calls this method to perform assignments when a function is called (to set the local bindings for the arguments), to set temporary objects (as when a vector is modified), etc.. The calls made during these background processes have the same structure as the ones performed by the user, the only distinction between them being the environment they were signaled from.

To mitigate this behavior, and to ensure data correctness when correlating the collected data with the programmer's script, the `Environment stack manager` was created. It ensures that the environment of the call corresponds to the current user' execution environment. This component acts as a common stack implementation, pushing new frames every time a user defined function is called (which consequently creates a new environment) and popping them when those functions return. With this, the ABD_tool ensures that the calls belong to the script by comparing the environments (which are memory address and thus, their addresses comparison is sufficient) and that the collected data does not come from the R interpreter internals nor from a library instruction.

The environment stack manager also stores variables that might be environment specific to avoid their propagation. As an example, a task may need a two-step processing phase and between those two steps exists a window where a function can be called. This would lead to the first step values being used in the new environment instead of being stored to be used when the function returns.

4.4.2 Object Manager (obj_manager.h)

The objects, alongside with the events, are one of the most important aspects of the ABD_tool due to the fact that, without them, reconstituting a certain state would not be possible. Since objects are meant to store data, this must be done in the most efficient and less memory consuming form possible, in order to allow a better scaling and accessing/storing speed.

```
typedef struct abd_obj
{
    int id;
    char *name;
    unsigned int usages;
    OBJ_STATE state;
    SEXP createdEnv;
    char *createdAt;
    ABD_OBJECT_MOD *modList;
    ABD_OBJECT_MOD *modListStart;
    struct abd_obj *prevObj;
    struct abd_obj *nextObj;
} ABD_OBJECT;
```

Figure 30 - ABD_OBJECT structure

All the objects, both code flow and common objects, share the above structure. The only difference is that the code flow objects (the `CLOSEXP's`) do not contain memory allocated in both `modList` fields. The `modList` field is paired with a `modListStart` pointer in order to allow an easier manipulation of the linked list. The `modList` points to the memory location of the last modification that the object suffered (its current state) and the `modListStart`, as the name suggests, points to the very first which is the object definition (more information on this approach in Section 4.5 Optimizations). These fields intend to create a timeline of the multiple states that the object has assumed so far.

These states are named `ABD_OBJECT_MOD` (object modifications) and are demonstrated in Figure 31 and explained afterwards.

```
typedef struct abd_obj_mod
{
    int id;
    ABD_OBJ_VALUE_TYPE valueType;
    union
    {
        ABD_VEC_OBJ *vec_value;
        ABD_FRAME_OBJ *frame_value;
    } value;
    OBJ_STATE remotion;
    struct abd_obj_mod *prevMod;
    struct abd_obj_mod *nextMod;
} ABD_OBJECT_MOD;
```

Figure 31 - ABD_OBJECT_MOD structure

Each object modification contains an identifier (`id`), which correlates directly to the number of modifications that the object had made to it. For example, when an `IDX_EVENT` is signaled over a certain object, the `toObject` and `toState` event fields will be populated with the object identifier (`object.id`) and the current modification identifier (`object.modlist.id`) correspondingly (not the values themselves, but pointers to the `ABD_OBJECT` and `ABD_OBJECT_MOD` corresponding structures).

The previously used polymorphic approach was used again (under the `value` variable) to describe the data structure that the object is using in a determined state (and its values, consequently). This mitigates data type and data structure changes meaning that, each object is not strongly bound to any of those and thus, they may assume multiple types during their lifetime. To aid in a faster access to the right structure in use, a variable named `valueType` is used to describe the structure, based on an enumeration data structure that contains all the supported data structure types.

Another important aspect, that is useful to reconstruct the current values for a given state at the displayer stage is that, each structure under the `value` variable contains a flag named `isMod` to indicate if the `ABD_OBJECT_MOD` is either a definition/redefinition or a modification to a past definition.

4.4.3 Displayer

The displayer, as stated before, is one of the most important components in the ABD_tool architecture although its functionalities are not, and cannot be, directly requested by any signal. This means that it is not directly accessible through the implemented API-like structure which makes it independent from the remaining components (the ones within the logging environment) but not from the data exported from them (more specifically, the `JSON helpers` component).

The displayer was developed using mainly HTML and JavaScript, and the data it needs originating from the four JSON [24] 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, since they contain all the needed information to reconstruct the state of the program at a given time, 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.

a) Graph generation

The execution graph nodes maps directly to each environment executed by the programmer's script, meaning that, each box (Figure 32**),** represents a called function. When applicable, the graph also contains directed edges indicating the connection direction between two nodes to identify the parent and the child environments.

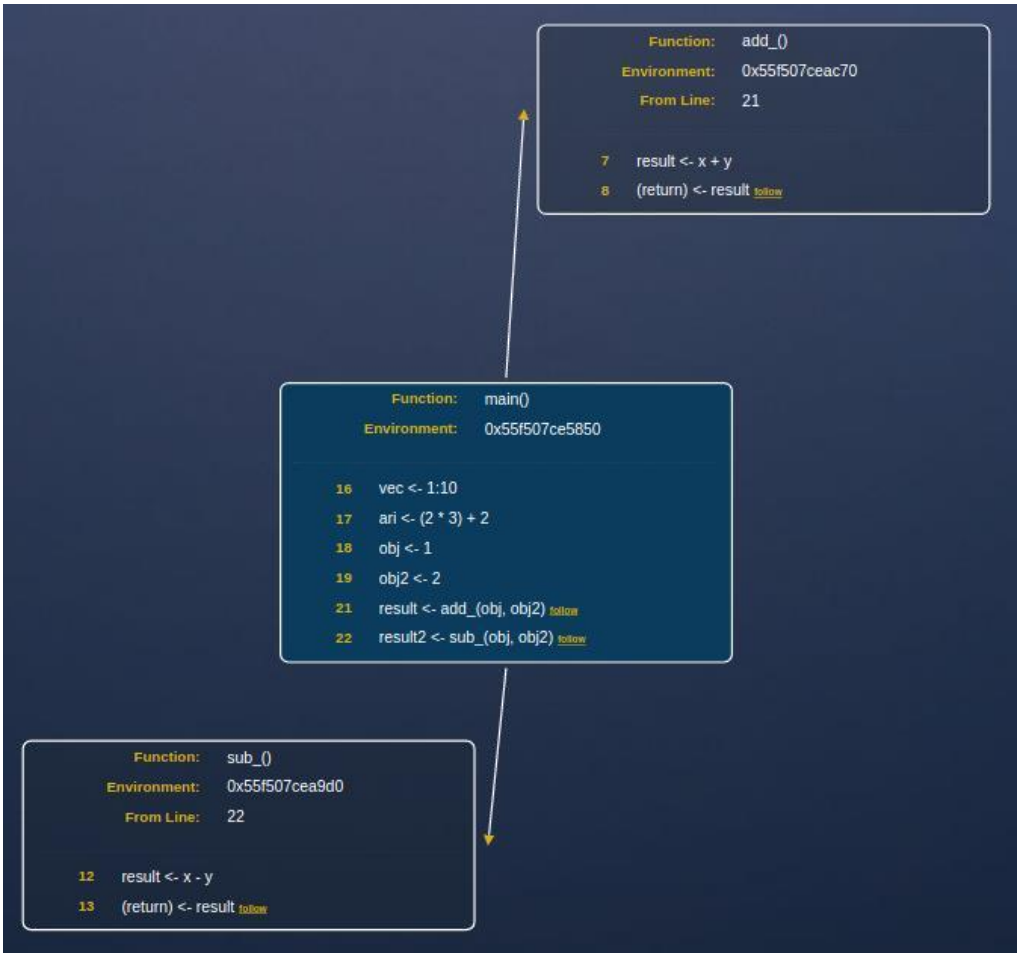


Figure 32 - Graph two nodes example

To generate the graph, every time the displayer notices a `FUNC_EVENT` (or at the very start, where the event is the `MAIN_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 33).

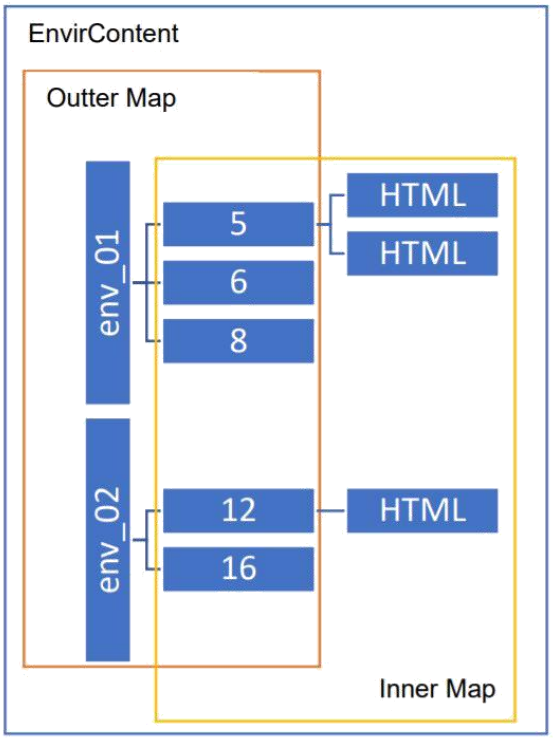


Figure 33 - EnvirContent abstract overview

The inner Map object values is a list of HTML snippets to allow more granularity and functionality. For instance, if an assignment value originates from an arithmetic operation, the right-hand side of the operation will have a clickable label (that displays the dedicated arithmetic analysis UI), and the left-hand side of the operation another clickable lable (to the dedicated object analysis UI, with the displayed data based on the [object,state] tuple). As a consequence of this behavior, the inner Map will have two distinct entries for the designated line of code.

If, on the other hand, the value of that assignment originates from a user typed value (hard-coded), the generated HTML will only comprise one clickable label and thus, the inner map will consist of only one entry for the designated line of code.

b) Object current values

Since the exported data only contains the deltas (the difference between states) and not the objects complete values, the current values for those specific states need to be resolved in order to present the correct information to the user.

This reconstruction is done in a backtrack mode (over the object `modList` variable) starting from the requested state and finalizing a state flag `isMod` is set to false (until it reaches an object definition/redefinition state). Furthermore, it will create an auxiliary Map variable that contains the modified indexes (or the cells, in case it is a multidimensional object), as its keys, and the modification values as the key's values.

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 variable:

- The indexes are not found: they are added to it as keys, with the corresponding new values as values.
- The indexes are found: they are skipped (newer values were already found for them).

When the algorithm finds an object defining state (when the flag `isMod` is set to false), 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, thus, the current values for a determined state.

c) Constraints

One of the primary ABD_tool intentions is to offer the users ease of use and flexibility as well as a simple software setup and requirements. Due to this, the displayer relied on a serverless architecture which answers to all the previously mentioned needs and intentions, although it has some specific downsides inherited from the browser-based approach.

Due to web browsers recent security policies, they cannot load files directly from the file system without the user's explicit intention (a file browsing functionality needs to be implemented and the user needs to select the files manually). To achieve this automatically, a server needs to be running and a request for the desired file must be issued to it. To mitigate this constraint, 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), having the tradeoff of making the exportation performance, basically, twice as worse.

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

4.5 Optimizations

Since the overheads of this tool were expected to be enormous due to the extra processing needed to collect, store and process the data signaled from the R interpreter, some optimizations needed to be done to minimize them as much as possible.

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, or a new change, and also when verifying if the function being called was defined/declared by the programmer. To optimize the registries searches even more, the objects are ordered by their usages (as it is done with registries in operating systems virtualization scenarios) because if, an object has many usages the probability of it being used again (without code analysis) is higher and thus, it being better ranked in the registry, will make its look-up faster.

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 (a pointer to the tail of the linked list), 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 and (since it has only one usage) its ordering can also be skipped.

Also, to minimize the memory usage, the complete values of the common objects, for example a vector, are only stored when it is defined or redefined. This means that values to all positions are only stored once instead of on every modification. The index modifications were optimized by registering only the new indexes and the corresponding new values (the delta's), thus greatly reducing the consumed memory space and the time complexity to register the changes.

As an example, if the programmer declares a vector and then modifies the n^{th} index, the new memory block allocated to express that change will only contain the new value assigned to that n^{th} position and not all the previous data as when the vector was declared (with the new change). When working with small vectors this would not cause a noticeable improvement, but when thousands of indexes are considered, the impact in memory consumption/usage is high because it would grow significantly due to values duplication. The processing time is also reduced due to the fact that, for instance, if the programmer modifies 10 indexes of the 1000 that the vector initially allocated, instead of copying the new vector state with 1000 values, the ABD_tool just needs to process those new 10 values. Those modifications are then resolved (in a backtrack mode) at visualization time (when they are actually requested and for a determined state) to reduce the exportation time and to reduce, also, the size of the file that contains the exported data.

As new values are assigned to an object, either modifications or redefinitions, they are appended to the `modList` field in the object structure (as a double linked list). To optimize the accessing and insertion of new values in the mentioned `modList`, 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 beginning, or head, represents the very first modification made to the object and the end, or tail, represents the state that the object is currently at. This approach also improves the data exportation phase because, if only the final state pointer was saved, either the states would be exported from the last to the first (reverse order), which would worsen the resolution of a determined state, or the complete list would need to be traversed to reach the very first modification and then traversed again to export it in the correct order.

4.6 – Installation and configuration

In order to provide to its users an easier usage of the it, the ABD_tool provides an option to its manual installation through a Python 3 [25] script. The script provides the following features:

- Verifies if the ABD_tool is already installed (patches the needed files if it is not)
    - This filter can be bypassed to force a new installation
- Installs R recommended base packages
- Verifies the R source files integrity through a `checksum` (SHA1 [26]) against the specific version tested and supported (version 4.0.2)
    - This verification can be bypassed
- Verifies if the R source files folder has the standard structure
- Configures the R source files through the `./configure` command
- Builds the R source files through the `make` command
- Modifies the `PATH` [27] [28] environment variable (so the R and Rscript binaries are available through the common `PATH` lookup mechanism)
    - The user is prompt to decide on this behavior (Yes/No answer).

Note: the script assumes that the machine has a `bash` shell [27] [28] installed.

The users have the following arguments available for usage:

- **-h**: Displays the help information
- **-r** : Displays the requirements to use ABD_tool
- **-p*** : Specifies the path to the R source files folder in which the user desires to install the ABD_tool
- **-c** : Bypasses the checksum verification
- **-m** : Configures and build after installation
- **-f** : Force installation (since it assumes already modified R source files, it also bypasses the checksum verification)

Note: the * indicates mandatory arguments the remaining being optional.

An installation command (using the script) will have the following structure:

./abd_inst -p /path/to/R/source/files [options]

The optional arguments must be placed before, or after, the -p argument, never in-between. The following usage will cause an error:

./abd_inst -p [options] /path/to/R/source/files

If the user does not have access to Python a manual installation can be performed (Figure 34).

```
cp *.h /rs/src/include/
cp *.c /rs/src/main/
cp -r abd_tool /rs/src/include/
cd /rs/tools/
./rsync-recommended
cd ..
./configure
make

echo "export PATH='/rs/bin/:$PATH'" >> ~/.bashrc
source ~/.bashrc
```

Figure 34 - ABD_tool manual installation

Note: in Figure 33, **/rs/** assumes the path to the R source files folder location.

In order to take full advantage of the ABD_tool functionalities, users can configure its settings through the R command line interface, having at their disposal the following commands:

- abd_help() – Displays all the available commands
- abd_start() – Initializes the tool
- abd_stop(x) – Tool stopping behavior
  - x = empty: Stops the tool immediately
  - x = 0: The displayer is not launched on stopping
  - [default] x = 1: The displayer is launched on stopping
- abd_path(path, x) – File output folder configuration
  - path and x = empty : Shows the path for all the files
  - With a valid path:
    - x = 0: Sets all files' output folder to the designated path
    - [default] x = 1: Sets object file output to the designated path
    - x = 2: Sets events file output to the designated path
    - x = 3: Sets warnings_errors file output to the designated path
  - Default path: /home/<user>/Documents/ABD_tool
    - The provided paths must be absolute.
- abd_verbose(x) – Change verbosity of the ABD_tool output
  - [default] x = 0: Disables verbose mode
  - x = 1: Enables verbose mode
- abd_display() – Launch displayer immediately
  - The program execution waits for a user input to proceed
- abd_clear() – Loads default settings

5 Evaluation

Evaluating a tool is crucial to identify if the intended goals, and the way it was implemented, were in any form beneficial to the final user of that particular tool. In the ABD_tool case, there exist multiple forms that this evaluation can be performed.

One very important aspect that should not be overlooked, regarding the ABD_tool's usage and its evaluation, is the fact that its users may give more value to the functionalities and the overall debugging easiness than to the actual computation power needed and the added overheads. Of course, this may be assumed within a certain degree because if the tool were extremely slow and consumed a lot of resources, its practicality and usability downgrades greatly and the users might not consider using it.

Because the tradeoff between debuggers' performance and the added functionalities is difficult to evaluate absolutely, the ABD_tool' evaluation aims to distinguish between its performance and user satisfaction.

The initial plan was to measure the overheads introduced by the tool to the R interpreter and, also, to measure the programmer's efficiency in order to compare it to the current available debugging techniques. This last evaluation methodology was not possible due to multiple factors: the global pandemic (due to the COVID-19 outburst) which prohibited any user study, and due to the time constraints that the developed work had.

The user study was a big evaluation factor because it intended to identify difficulties that the users had when using the tool (without any intervention) and also to discuss, with them, what in their point of view could be improved and what they appreciated the most.

5.1 Test environment

The machine used to perform the tests/benchmarks consisted of:

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

5.2 Performance measures

To understand how the ABD_tool impacted R execution time, multiple benchmarks were made related to different aspects of the language. The very first aspect was recursion, the second big data usage and the final looping and branching.

The time consumed by each benchmark run was recorded with a R native function called `time()`, located in the Sys (system) library. It was used to create a `start_time` and `end_time` object to then show the time consumed by printing the difference between the `end_time` and the `start_time`. These calculations give the result in seconds, with a 60 seconds maximum so, in certain scenarios, manual calculations were made to get the actual result.

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).

5.3 Micro-benchmark 1 - Recursion

Recursion programming approach relies on using a function that calls itself in order to divide problems into smaller ones (eventually reaching an atomic problem) and thus, making their resolution easier. This programming approach has inherited overheads due to the call stack size increase making it, most of the times, slower and more demanding (in terms of resources, more specifically, RAM) than iterative/dynamically programmed approaches.

R limits the depth of recursion to 999 calls, throwing an error at the 1000th call. This translates to multiple `recur()` calls approach (to generate more recursive calls), instead of increasing the depth of the recursion in order to mitigate this implementation limitation.

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

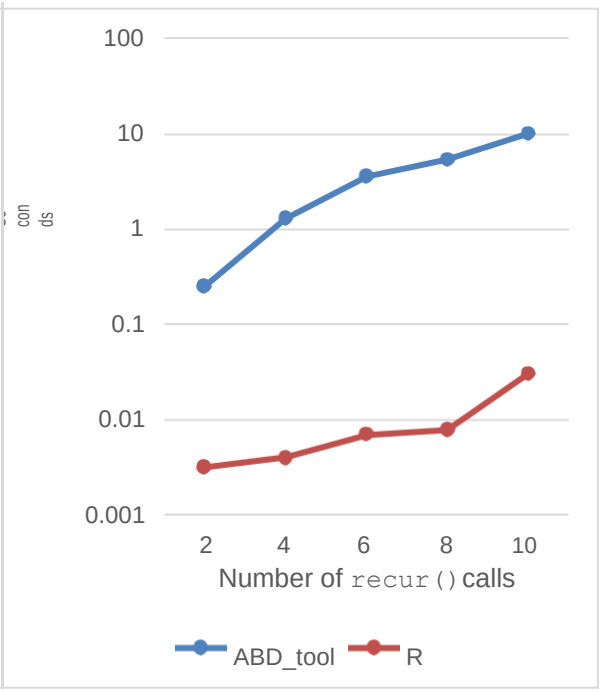
Figure 35 – Micro-benchmark 1 R code

This micro-benchmark stresses the following ABD_tool functionalities:

- Function calls (ABD_FUNC_EVENT, ABD_EVENT_ARGS, ABD_OBJECT, ABD_OBJ_MOD, local variables and the concept of contexts)
- Simple branch conditions (ABD_IF_EVENT)
- Arithmetic expression (ABD_ARITH_EVENT)
- Assignment (ABD_ASSGN_EVENT)
- Return (ABD_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 8 – Micro-benchmark 1 results



Graph 1 - Micro-benchmark 1 results

As described and shown by the micro-benchmark results (Table 8 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.

5.4 Micro-benchmark 2 - Data import and processing

Since R is more commonly used in the area of data science, which usually works over big sets of data, this micro-benchmark intended to verify how much the ABD_tool impacted the loading of a set of data from a data source (in this case a file), and its consequent storing.

To achieve this, and since the data size was the key and not the content itself, the source values for the current performance micro-benchmark were pulled from a website [29] containing different sized files (but with the same structure). The file structure consisted of multi-type columns accounting for a total of fourteen columns.

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

Figure 36 – Micro-benchmark 2 R code

Although the script has just one line of code, it will perform multiple actions:

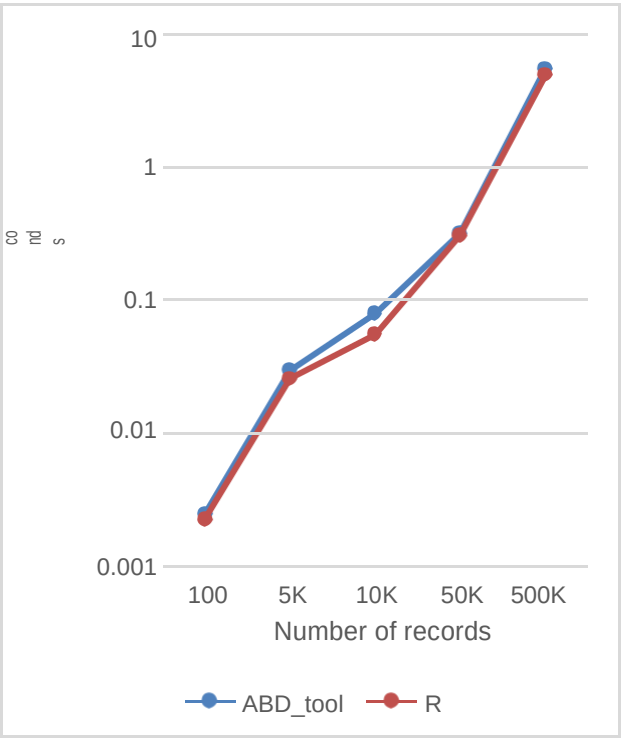
- Read the data from the comma-separated values (CSV) file
- Process the data
    - Identify each column file type
    - Allocate memory
    - Perform data type verifications and modifications on the used data structure (if needed)
- Store the data in the df object (which abbreviates the data structure used: Data Frame).

The multiple actions performed can be translated to the following ABD_tool' functionalities:

- Process the data
    - Each column (ABD_VEC_OBJ)
    - Memory management (ABD_OBJECT, ABD_OBJ_MOD)
- Dataframecreation (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 9 – Micro-benchmark 2 results



Graph 2 - Micro-benchmark 2 results

The micro-benchmark 2 results show, in Table 9 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.

5.5 Micro-benchmark 3 - Looping and branching

One of the most time-consuming instructions in programming is branching, due to the fact that the compiler most of the time cannot predict the outcome of those branches and, thus, cannot optimize the code to run faster. Although, measuring the speed of branching is not trivial because the quantity of branching needed to produce measurable and relevant values implies a very code extensive script (also, it is humanly impossible to produce).

To mitigate this, the branching micro-benchmark had to be based on a loop (which wraps the branches instructions) that increases the processing power required at each determined phase of its execution. Considering this, the benchmark was structured with a for-loop that for each phase of evaluation will increase the number of iterations (non-linearly). This loop statement served as a wrapper to a long branch statement that would progressively perform more comparisons while the loop iterations number increases. For example (Figure 37), for the first 1000 thousand iterations of the loop 1000 comparisons are made but when the for-loop iterations are increased to 10 thousand, the number of comparisons increase to 11 thousand. This ensures that, when the number of loop-cycles are increased, the computation power needed to perform the micro-benchmark also increases.

```
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")
    }
}
```

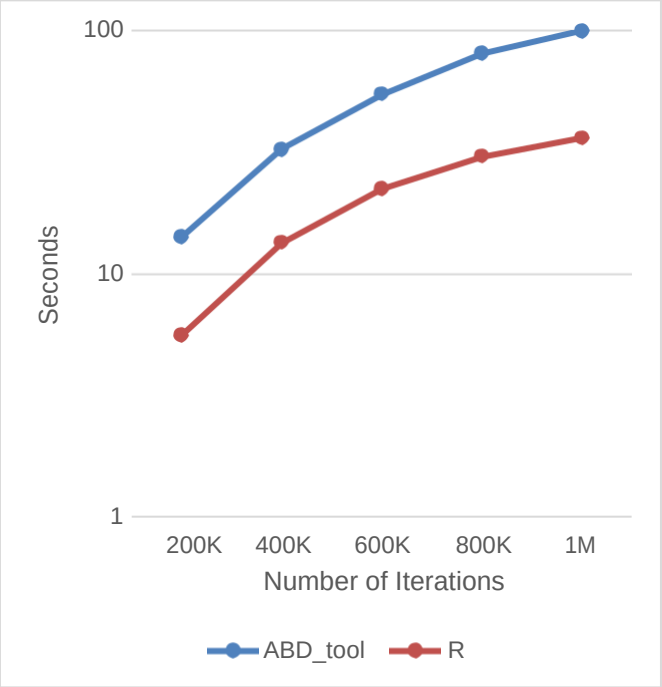
Figure 37 – Micro-benchmark 3 R code

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

- Looping (ABD_FOR_LOOP_EVENT and ABD_VEC_CREATION) ○
  The loop event comprises multiple tasks:
  - Create lists with the multiple iterations (ABD_ITERATION)
  - Assign the created events to each iteration
- Complex branch conditions (ABD_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 10 – 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 10 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 increased to one million.

5.6 Macro-benchmark

The benchmarks performed until now did considered some factors.

1. The process of persisting the data was not taken into consideration in order to have a more normalized execution time between the tested applications (ABD_tool and the R interpreter). This way the actual time to finalize the script execution could be effectively compared between the scripts without the added factor of data exportation.
2. The data used in the micro-benchmark 2 could be significantly bigger (bigger data sources were not used to reduce the benchmark extensiveness).
3. The performed micro-benchmarks did not took into consideration data usage/manipulation (no data accessing, calculations over values, etc.).
    a. Performing tasks over the data requires its values traverse, which increases the script execution time.

In order to have a closer to real usage comparison (where the data exportation is used and must be a contributing factor), a new execution test was performed taking into consideration the factors that were not considered previously.

Due to this, the test had the following approach (Figure 38):

- Load a much bigger file (with the same column count as the one used in the synthetic benchmark)
    - 5 (five) million records * 14 (fourteen) columns = 70 (seventy) million cells
    - Cells with different types: strings, reals, integers.
    - **Note:** the previously benchmarked data sources were also used to view the data usage and exportation overheads.
- Perform calculations over the data loaded from the file (Total profit column)
    - Average
    - Sum
    - Minimum
    - Maximum
- Post-execution debug data export
    - Persist the data kept in memory by the ABD_tool to the designated files to then be accessed by the displayer functionality.

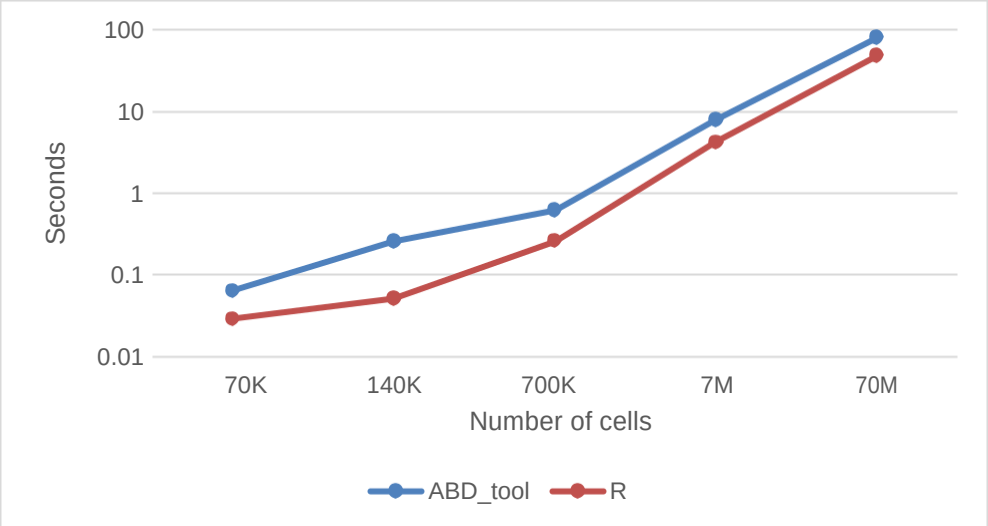
```
df <- read.csv(file = "file.csv")

#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 38 – Macro-benchmark R code

Number of cells	Running method	Execution timings results (seconds)			Memory usage
		Process	Export	Total	

70K	ABD_tool	0,03s (+3%)	0,034s	0,064s (+55%)	0,7% ~57MB (+0%)
	R	0,029s	-	0,029s	0,7% ~57MB
140K	ABD_tool	0,066s (+21%)	0,192s	0,258s (+80%)	0,8% ~65MB (+0%)
	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 11 – Macro-benchmark results



Graph 4 - Macro-benchmark results

From the above results (Table 11 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 structures used (and managed) by the tool, do not consume a significant/noticeable amount of system' resources when comparing with R.

5.7 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 [30] platform was made to assess the users first impressions as well as the ABD_tool potential.

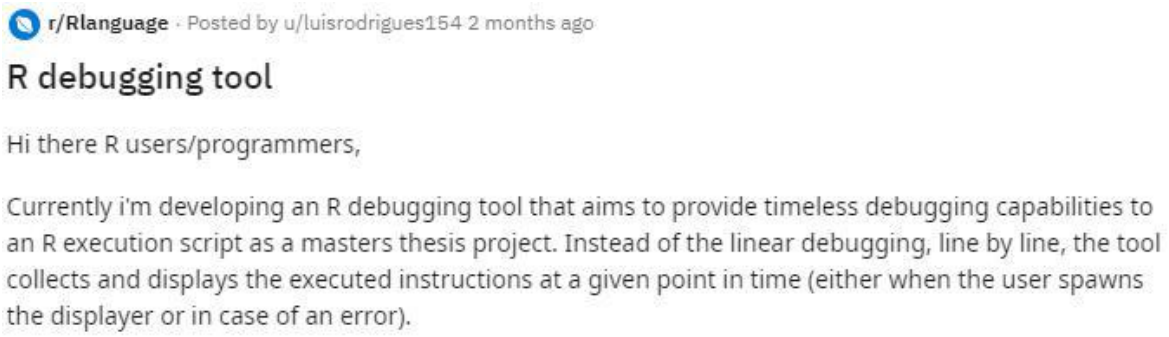


Figure 39 - Reddit post presentation [31]

The post was constructed with a small presentation of the project, underlining that it was an academic project and that has some limitations as is stated in this document. Also, the post contained a list of the most relevant features that are not present in other debugging/developing environments such as:

- Data precedence with granularity for each executed instruction.
    - Data origin.
    - Custom UI depending on the above origin.
- Branch breakdown and results by statement
    - If a branch has more than one statement, the results for each are presented together with the branch evaluation order.
    - Example:
        - (1+2)>(3+4).
        - This example constitutes 3 statements.
        - Statement 1#:(1 + 2).
        - Statement 2#: (3 + 4).
        - Statement 3#: 1# > #2.
- Interactable graph visualization of the executed code.
- Warnings and errors displayed by the side of the corresponding instruction (event) that generated them.
- Custom data presentation and navigation.

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

Note: the survey was intended to be small but direct in order to make its answering fast and doable (normally, people tend to avoid answering surveys due to the time they consume).

The results to the survey are:

- R language section reddit users: 21.4 thousand
- Post views: Not available
- Number of comments: 11 (eleven)
- Total answers: 84 (eighty-four)
- 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 (RStudio and Flow package) which I responded accordingly. The users, after the explanations to their doubts, commented that the ABD_tool was something worthwhile for the community.

5.8 Discussion

Considering the obtained results, with the micro and macro-benchmarks as well as with the user-study, a pretty bold, but sustainable, statement can be made: the ABD_tool has the potential to become (in a future iteration where more features are supported and implemented) one of the primary debugging tools choice to debug R programs. Although some benchmarks might discourage this statement at a first glance, when further analyzed and applied to real world usage examples, they showcase that the performance penalties with the current ABD_tool implementation is not problematic at all when considering the value given by its functionalities. Another really important aspect when analyzing the results is the correlation of the performed performance benchmarks with the R language applicability.

The micro-benchmark 1 is a proof of that. When analyzing the results superficially (Table 8), 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 has to either script the debugger to stop on that particular call or skip all the unwanted calls. With ABD_tool, the information was already collected in one run. In the micro-benchmark 1 case, it is a tradeoff of 10 seconds for recording 9990 function calls and all their instructions results or using the traditional approach with the much faster (and standard) R interpreter.

Although it might be acceptable, it was the most problematic micro-benchmark due to the amount of structures and events needed to recreate all the context. When a function is called, a new `env_stack` frame is pushed to update the current ABD_tool environment, then a new `FUNC_EVENT` is created. The function arguments need to be processed and local variables need to be created. Upon return, the `RET_EVENT` needs to be created, the value stored and the new environment that was pushed into the stack need to be popped and so, the current frame needs to be updated to the previous. If the return value is used in any assignment then, the `RET_EVENT` created before needs to be modified to update the data precedence for the object that is receiving the value. All this behavior constitutes a greater that desired overhead but, as stated afore, acceptable analyzing the results differently.

The micro-benchmark 2 results (Table 9) 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% (+0,004) with an outlier of 43% (+0,024) 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 10 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. Considering the quantity of iterations and the branch conditions tested during those iterations, the ability to analyze each of those instructions on-demand at any time, completely washes away the time difference. Using an 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 (scripting the debugger or manual approach). 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 (which are used by the displayer)
- The loaded data usage

With the results (Table 11), it is visible that the major performance hit that the ABD_tool results suffered originated from the generated data exportation. This is mostly due to the already mentioned limitations of web browsers privacy policies (thus, the ABD_tool needs to essentially write to 2 files and consequently worsen the performance). 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. In this performance benchmark, memory usage was also tested with the highest disparity in the results residing in the 70 million cells file run (which represented a .csv file of ~500MB). The difference (+50%) is explainable by the initial data duplication made by the ABD_tool to the objects structure.

The final performed test was not performance related, but user related. The user online questionnaire partly mitigates the absence of a full user study that was supposed to be performed but became impossible to execute, due to the global pandemic and the inherent prohibitions imposed to avoid its spread. The questionnaire focused on users' opinions on the ABD_tool potential – if it is worthwhile to the community and the features show a better debugging paradigm and approach than the currently available techniques. From the user study results, it is possible to conclude that the users were interested in the tool and its potential.

6 Conclusions

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, conceptually, divergent from the currently available techniques in the R language (and the most/commonly used third-party applications, such as RStudio).

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
    - General application usage simplicity
    - The programmer just needs to focus on the code and not the process of debugging
- Enhance the capabilities of the debugging tools
    - Data analysis at any given time
        - During the script execution (ABD_tool hangs the execution and launches the displayer)
        - Any time after by loading the exported data
    - Debugging oriented UI
        - Graph visualization by environment
        - Customized display for each supported R instruction type
            - Example:
                - Branches display each statement evaluation result
                - Function calls UI display arguments and which name they were received at (mapping between the values passed and the corresponding local variables)
                - UI oriented to the data (more functionalities to navigate through bigger data sets)
                - Etc.
- Time savings
    - One run collection (no re-runs needed)
    - No breakpoint placement guessing (just start and stop the tool to collect data regarding the desired section of code)
    - Overall benefits
        - The custom UI
        - The usage simplicity
        - The relatively low added overheads

As demonstrated throughout the document, 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, by him/her. Considering these goals and the overall results obtained with the performed benchmarks (and, also, the user study that was possible to perform at the world current

state), 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 (section 5.6, 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 is already surpassed by the standard techniques procedures.

Another important benchmark, that showcases the functionalities and the time savings, is described in the section 5.5. If a special attention is taken over the last performed run of this benchmark, it is visible that the ABD_tool requires roughly 3 times more CPU time to conclude. Considering the actual time taken over the R interpreter (+1 minute), the number of iterations performed by the loop (1 million), the number of branch conditions as well as the fact that the programmer can access each of these on-demand, the time addition can be easily consumed when trying to debug using the available debugging techniques.

Since the ABD_tool has its displayer implemented with the web-browser experience in mind, the data visualization (as well as its usage) 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 file types supported. Although it does not cover all the R language, most of the remaining functionalities do not constitute a greater challenge to implement (and expand the tool) with the only resource needed being, effectively, time.

7 Future work

The future work on the ABD_tool should mostly focus on expanding of the supported features, improvement of the displayer ecosystem and performance optimizations.

One aspect that can be improved in the future is the displayer and the form it is currently implemented. The major fact regarding the displayer is that, to alleviate the users initial requirements to use the ABD_tool, it is a serverless implementation which means that the user does not need to be running a webserver to have the web-browser experience and the debug oriented UI provided by the tool. Although this reduces drastically the requirements to use the tool, the overheads added when exporting the collected data are extremely high and thus, to mitigate this, there are two identifiable approaches:

- Develop an independent UI without relying in web-browsers (the generated files have such a structure that allow their usage by any other application)
- Instead of writing to files, the ABD_tool could implement a completely different approach by using sockets to communicate directly with a webserver and reduce the overheads of writing to disk by working solely on memory (this webserver can be a local instance, making the round trip times fast).
    - It can go even further by creating a serializer to other language and interchange the structures held on memory directly.

Also, optimizations regarding the events created by the ABD_tool should be made to reduce, especially when working with function calls (to avoid the recursion performance problems seen in the micro-benchmark 1), to achieve closer performances to the standard R interpreter. These optimizations can go further by performing general code profiling and optimizations in order to visualize what portions of the implementation consume more memory and CPU time and, thus, reduce them.

Due to the extent of the R programming language interpreter, it was unfeasible and impossible to implement all the features that a language which grew incrementally supports/implements. Because of this, one of the concerns is the expansion of the supported features in order to reach more users and offer better debugging capabilities.

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

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

[4] J. Allaire, "RStudio: integrated development environment for R," *Boston, MA,* vol. 770, p. 394, 2012.

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

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

[7] J. M. Chambers, Programming with data: A guide to the S language, Springer Science & Business Media, 1998.

[8] R. Ihaka, "R: Past and future history," *Computing Science and Statistics,* vol. 392396, 1998.

[9] D. Smith, *Over 16 years of R Project history,* 2016.

[10] D. Page, A practical introduction to computer architecture, Springer Science & Business Media, 2009.

[11] J. R. Anderson, R. Farrell and R. Sauers, "Learning to program in LISP," *Cognitive Science,* vol. 8, p. 87–129, 1984.

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

[13] H. Wickham, Advanced r, CRC press, 2019.

[14] R. Stallman, R. Pesch, S. Shebs and others, "Debugging with GDB," *Free Software Foundation,* vol. 675, 1988.

[15] A. Fabri, *Flow.*

[16] J. Loeliger and M. McCullough, Version Control with Git: Powerful tools and techniques for collaborative software development, " O'Reilly Media, Inc.", 2012.

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

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

[19] D. Goodman, Dynamic HTML: The Definitive Reference: A Comprehensive Resource for HTML, CSS, DOM & JavaScript, " O'Reilly Media, Inc.", 2002.

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

[21] J. Chaffer, Learning jQuery, Packt Publishing Ltd, 2013.

[22] S. Moreto, Bootstrap 4 By Example, Packt Publishing Ltd, 2016.

[23] N. Q. Zhu, Data visualization with D3. js cookbook, Packt Publishing Ltd, 2013.

[24] 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.

[25] G. Van Rossum and F. L. Drake, The python language reference manual, Network Theory Ltd., 2011.

[26] A. J. Menezes, P. C. Van Oorschot and S. A. Vanstone, Handbook of applied cryptography, CRC press, 2018.

[27] M. Mitchell, J. Oldham and A. Samuel, Advanced linux programming, New Riders Publishing, 2001.

[28] C. Newham and B. Rosenblatt, Learning the bash shell: Unix shell programming, " O'Reilly Media, Inc.", 2005.

[29] E. For Excel, *Downloads 18 - Sample CSV Files / Data Sets for Testing (till 5 Million Records) - Sales | E for Excel | Awakening Microsoft Excel Student Inside You.*

[30] *Reddit.*

[31] L. M. Rodrigues, *r/Rlanguage - R debugging tool.*