UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Effective code maintenance with continuous data collection

by

David Chenaux

Thesis for the Master of Arts in Information Management
Supervised by Prof. Dr. Philippe Cudré-Mauroux and Dr. Roman Prokofyev

eXascale Infolab
Department of Informatics - Faculty of Economics and Social Sciences -
University of Fribourg

March 21, 2017

UNIVERSITY OF FRIBOURG
Faculty of Economics and Social Sciences
Department of Informatics
eXascale Infolab

Thesis for the Master of Arts in Information Management
Supervised by Prof. Dr. Philippe Cudré-Mauroux and Dr. Roman Prokofyev

by David Chenaux

Abstract

Code analysis has been around for a few decades already, yet it is mostly limited to static analysis without considering the actual data stream flowing through the code. The intention of this thesis is to implement a proof-of-concept system for effective code maintenance with continuous data collection.

The results of the development are detailed in the present report. In order to understand the field of Dynamic Program Analysis, this work first dedicate an entire chapter about related work which includes a definition of the program analysis, several approaches and a brief overview of available tools. Then, a presentation of the coded system is made and finally we propose at the end of the thesis some future work which could be implemented in the developed system.

Acknowledgements

First, I would like to thank Prof. Dr. Philippe Cudré-Mauroux for the creation and the lead of the eXascale Infolab. This research group is a great asset for the University of Fribourg and gives amazing opportunities to explore technical and experimental fields in many informatic sectors.

Then, I would particulary like to thank Dr. Roman Prokofyev who supervised my work and who has been really supportive and comprehensive with my time schedule. I thank as well Michael Luggen who came in backup and gave a fresh insight on the developed proof-of-concept system.

Finally, I send all my gratitude to Prof. Marino Widmer for helping me with my administrative struggles and Gaëtan Vieux for his final corrections and suggestions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Code analysis has been around for a few decades already, yet it is mostly limited to static analysis without considering the actual data stream flowing through the code. Ability to display the actual data running through the system promises many potential benefits, including easier debugging and code recall, which results in significantly lower code maintenance costs.

1.1 Problem definition

Every developer is more or less feared about the debugging and code reviewing phase of their software. Obviously, this process can sometimes take several painfully hours and each programmer knows how frustrating it can be to search for a hidden bug in thousands lines of codes. In order to support the programmers in this hated task, debuggers are the most useful existing tools which are part of the so called *static program analysis* (SPA).

With the apparition of object-oriented programming language, searching for syntactic errors in the code is not anymore sufficient. Therefore, a new research field was pushed forward which is called the *dynamic program analysis* and consists in analyzing the software during its execution. This procedure allows to take in account some possible inputs which were not probed with the SPA. Yet none of the modern code analysis tools was able to successfully integrate their source code editors with the actual data stream flowing through the code.

1.2 Objectives

The goal of this project is to design a proof-of-concept system in one programming language that allows full code instrumentation. This system should be able to :

- seamlessly capture some or all values for all variables in source code and store them somewhere ;
- easily retrieve the saved values by providing an API to the storage in order to make the data accessible for navigation in third-party applications;
- provide a basic visualizing interface in order to allow an easy analysis of the results.

Finally, an evaluation of the system performances will be established through different experiments.

1.3 Organization

The thesis is organized into four sections :

1. **Related work**: In this first chapter of the thesis, we present an insight of the existing work on the field *program analysis* and in particular the DPA. This is including a definition of the field and its particularities, an overview of some available solutions side by side with the current restrictions.
2. **Development**: This part describes the architecture of the proof-to-concept system with a presentation of the proposed solution and detailed information about its structure.
3. **Installation guide**: An installation guide of the software which describes the required environment, the package installation and the compilation of the system.
4. **Experiments**: In this section, we conduct a number of experiments in order to test and check the performance and results of the software.
5. **Conclusions** : The thesis concludes with some outputs and is proposing some future improvements which could be relevant.

Chapter 2

Related work

"Sharing is good, and with digital technology, sharing is easy."
Richard Stallman

The intention of this thesis, as brought up in the introduction, would be to implement a dynamical program analysis system. In order to meet this goal, it is necessary to build a theoretical understanding of "Program Analysis" and therefore the present chapter will endeavor to do a presentation of the subject. The first part propose a definition of the field, then the second suggest several technical approaches. Following, the third section introduces some popular analysis tools, to finally discuss the actual limitations of dynamic analysis in the fourth part.

2.1 What is Program Analysis ?

Programming environments are an essential key for the acceptance and success of a programming language. After Ducassé and Noyé [1994], without the appropriate developments and maintenance tools, programmers are likely to have a bad software understanding and therefore produce low-quality code. They will be therefore reluctant to use a language without appropriate programming environments, however powerful the programming language is.

As already introduced in the previous chapter, program analysis is an automated process which aims to analyze the behavior of a software regarding a property such as correctness, robustness, safety and liveness. Program analysis can be separated in two methods : the SPA which is performed without actually executing the software and the DPA which is performed during the runtime. [Wikipedia, 2016]

The SPA is a straightforward solution because it does not require running the program for analyzing its dynamic behavior. The analysis consists in going through the source code and highlights coding errors or ensure conformance to coding guidelines. A classic example of static analysis would be a compiler which is capable of finding lexical, syntactic and even semantic mistakes. The main advantage of this method is that it allows to reason about all possible executions of a program and gives assurance about any execution, prior to deployment.

Nevertheless, according to Gosain and Sharma [2015], since the widespread use of object oriented languages, SPA is found to be ineffective. This can be explained due to the usage of runtime features such as dynamic binding, polymorphism, threads etc. To remedy this situation, developers call on DPA which can, after Marek et al. [2015], allow 0 to gain insight into the dynamics and runtime behavior of those systems during execution. Moreover, because the runtime behavior depends now on many other factors, such as program inputs, concurrency, scheduling decisions, or availability of resources, static analysis does not allow full understanding of the code. The following table, proposed by Gosain and Sharma [2015], is resuming the main differences between static and dynamic analysis.

Dynamic Analysis	Static Analysis
Requires program to be executed	Does not require program to be executed
More precise	Less precise
Holds for a particular execution	Holds for all the executions
Best suited to handle runtime programming language features like polymorphism, dynamic bind ing, threads etc.	Lacks in handling runtime programming language features.
Incurs large runtime overheads	Incurs less overheads

Table 2.1: Comparison of Dynamic analysis with Static Analysis

In the light of this comparison, it is well worth noting that Dynamic Program Analysis does not substitute the Static Analysis. Quite the reverse, both are interdependent tools and even if Static Program Analysis is not sufficient anymore, it still gives relevant information about the code to the programmer. The DPA should come in a second phase when the source code has been validated through SPA. As it can be surmised, the ability to examine the actual and exact runtime behavior of the program might be the DPA main advantage, whereas SPA prime edge could be the independence of input stimuli and the generalization for all executions. To illustrate these characteristics, some program analysis solutions are presented further in this chapter.

2.2 Program Analysis approaches

Now that a definition of Program Analysis has been established, we are going to explore some different approaches for going into the subject in depth. Yet, since the field is expansive, the purpose of this section is not to cover the entire subject. The reading of this section should, notwithstanding, give a good overview to the reader. Here we describe, first, the essential static analysis methods followed in a second time with the dynamic analysis techniques.

2.2.1 Static methods

The static methods are regrouped in four different categories proposed by Nielson et al. [2004] and briefly presented here, some information was also gathered from the Wikipedia [2016] page which is proposing a grouping based on the same criteria.

Data Flow Analysis is a technique which consist in gathering information about the values and their evolution at each point of the program. In the Data Flow Analysis the program is considered as a graph in which the nodes are the elementary blocks and the edges describe how control might pass from one elementary block to another.

Constrained Based Analysis or Control Flow Analysis, intent to know which functions can be called at various points during the execution ; what "elementary blocks" may lead to what other "elementary blocks".

Abstract Interpretation resides in proving that the program semantics satisfies its specification according to Cousot [2008]. What the program executions actually do should satisfy what the program executions are supposed to do. It can be sumarized as a partial execution of a program which gather information about its semantics without performing all the calculations.

Type and Effect Systems are two similar techniques where the second one can be seen as an extension of the first. Type systems are using types, which are a concise, formal description of the behavior of a program fragment. Rémy [2017] explains that programs must behave as prescribed by their types. Hence, types must be checked and ill-typed programs must be rejected. Effect systems are, after Nielson and Nielson [1999], an extension of annotated type system where the typing judgments take the form of a combination of a type and an effect. This combination is associated with a program relative to a type environment.

2.2.2 Dynamic methods

In the past section, some static analysis methods have been defined and therefore, the dynamic methods are depicted here. As it was heretofore specified, dynamic analysis is a quite recent research field which status could be still defined as academical. Naturally, the different techniques are not as well established as for the static analysis and can vary a lot in accordance with the author of the different papers. For this work, the following particular methods were privileged and were already proposed by Gosain and Sharma [2015] in their survey of Dynamic Program Analysis Techniques and Tools.

Instrumentation based approach needs a code instrumenter used as a pre-processor in order to inject instrumentation code into the target program. This can be done at three different stages : source code, binary code and bytecode. The first stage adds instrumentation code before the program is compiled, the second one adds it by modifying or re-writing compiled code and the last one performs tracing within the compiled code.

VM Profiling based technique uses the profiling and debugging mechanism provided by the particular virtual machine, for example the JPDA for Java SDK or the PDB for Python. These profilers give an insight into the inner operations of a program, especially the memory and heap usage. To capture this profiling information plug-ins are available and can access the profiling services of the VM. Benchmarks are then used for actual runtime analysis which acts like a black-box test for a program. This process involves executing or simulating the behavior of the program while collecting data which is reflecting the performance. Unfortunately this technique has the drawback of generating high runtime overheads.

Aspect Oriented Programming aims to increase modularity by allowing the separation of cross-cutting concerns. Because there is no need to add instrumentation code as the instrumentation facility is integrated within the programming language, the additional behavior is added to existing code without modifying the code itself. AOP adds the following constructs to a program : aspects, join-point, point-cuts and advices. These constructs can be considered like classes. Most popular languages have their aspect oriented extensions like AspectC++ and AspectJ. In Python, there are some libraries which aim to reproduce AOP behavior but there isn't any canonical one. Actually, there is a debate to what extent aspect oriented practices are useful or applicable to Python's dynamic nature.

2.3 Program Analysis tools

As the theoretical background is now settled, we want to propose in this section some static and dynamic analysis tools. The reader will discover in the next chapter that the proof-of-concept system is coded in Python and therefore additional information is given here for solutions available in that language.

2.3.1 Static Analysis tools

Following, some of the most popular tools (commercial or free) for SPA are described, picked in widespread languages : Java, C/C++ and Python. The diverse description are summarized versions of the Gomes et al. [2009] paper along with some official information gathered on the tools websites and their respective Wikipedia pages.

Starting with C/C++, **Splint** is a very well known tool, allowing to check for security vulnerabilities and coding mistakes. Splint is based on Lint and tries to minimize the efforts needed for its deployment. Additionally, with some annotation, Splint can extend its performances over Lint. Splint can among others detect : dereferencing a possibly null pointer, memory management errors including uses of dangling references and memory leaks, problematic control flow such as likely infinite loops. **Astrée** is based on abstract interpretation and can analyze safety-critical applications written or generated in C. It proves the absence of runtime errors and invalid concurrent behavior for embedded applications as found in aeronautics, earth transportation, medical instrumentation, nuclear energy, and space flight. Another worth mentioning tool is the **PolySpace Verifer** tool developed by MathWorks who also created the famous Matlab software.

Concerning Java, one recognized tool is Findbugs. With the advantage of being a Libre software, the application uses a series of ad-hoc techniques designed to balance precision, efficiency and usability. FindBugs operates on Java bytecode, rather than source code. Another Libre software is **Checkstyle** which, as his name gives a hint, allows to report any breach of standards in the source code. Finally a commercial tool, **Jtest** which is an integrated Development Testing solution, can perform Data-flow analysis Unit test-case generation and execution, static analysis, regression testing, runtime error detection, code review, and design by contract.

In the Python world, **Pylint** is a coding standard checker which follows the style recommended by the PEP 8 specification. It is also capable of detecting coding errors and is integrable in IDEs. Speaking of IDEs, **PyCharm** includes also static analysis functions like PEP8 checks, testing assistance, smart refactorings, and a host of inspections.

2.3.2 Dynamic Analysis tools

Just as for the static tools, the most popular DPA software are presented here. Following, a table proposed by Gosain and Sharma [2015] with a summary of some available DPA tools regrouped by technique. The table indicates the concerned language and also which type of dynamic Analysis is performed by the application.

Technique	Tool	Language	Type of Dynamic Analysis done								
			Cache Modelling	Heap Allocation	Buffer Overflow	Memory Leak	Deadlock Detection	Race Detection	Object LifeTime	Metric Computation	Invariant Detection
Instr.Based	Daikon	C,C++									✓
	Valgrind	C,C++				✓		✓			
	Rational Purify	C, C++, Java				✓					
	Parasoft Insure++	C,C++		✓		✓					
	Pin	C	✓								
	Javana	Java	✓						✓		
	DIDUCE	Java									✓
AOP Based	DJProf	Java		✓					✓		
	Racer	Java						✓			
VM Profiling Based	Caffeine	Java							✓		
	DynaMetrics	Java								✓	
	*J	Java								✓	
	JInsight	Java				✓	✓		✓		

TABLE 2.2: Dynamic Analysis Tools

Valgrind, Purify and Insure++ are instrumentation based, and can automatically detect memory management and threading bugs among with profiling a program in details. While Valgrind is a instrumentation framework for building dynamic analysis tools, the two others are fully-fledged analysis software. **Javana** comes with an easy-to-use instrumentation framework so that only a few lines of instrumentation code have to be programmed for building powerful profiling tools. **Daikon** and **Diduce** are trendy tools for invariant detection and are respectively an offline and online tool. Last but not least, **Pin** is a dynamic binary instrumentation framework developed by Intel. It enables the creation of dynamic program analysis tools and can be used to observe low level events like memory references, instruction execution, and control flow as well as higher level abstractions such as procedure invocations, shared library loading, thread creation and system call execution.

For AOP based applications, the two selected programs are **DjProf** and **Racer**. The first one is a profiler used for the analysis of heap usage and object life-time analysis and the second one is a data race detector tool for concurrent programs.

***J** and **DynaMetrics** are two academical research projects about Virtual Machine profiling and are proposing a solution for computing dynamic metrics for Java. The first one, proposed by Dufour et al. [2003], relies on JVMPI, while the second solution, from Singh [2013], relies on the new JVMTI. **JInsight** is for exploring visually runtime behaviour of Java programs and **Caffeine** helps to check conjectures about Java programs.

In addition to this table, some Python tools are also available even if the field seems not much developed for this programming language. This could be explainable because of the dynamic nature of the language and might be why the following tools are developed *in* Python but not *for* it. The first tool is **Angr** which is a Python framework for analyzing binaries. It focuses on both static and dynamic instrumentation analysis, making it applicable to a variety of tasks. **Triton** is another binaries analyzer framework and proposes python bindings. Its main components are Dynamic Symbolic Execution engine, a Taint Engine, AST representations of the x86 and the x86-64 instructions set semantics, SMT simplification passes, an SMT Solver Interface

2.4 Dynamic Analysis limitations

DPA is a quite new research field and as a consequence induces ineluctably some drawbacks and limitations. The following table created by Gosain and Sharma [2015] gives a good overview of the different techniques and some of their drawbacks.

	Instrumentation		VM Profiling	AOP
	Static	Dynamic		
Level of Abstraction	Instruction/Bytecode	Instruction/Bytecode	Bytecode	Programming Language
Overhead	Runtime	Runtime	Runtime	Design and deployment
Implementation Complexity	Comparatively low	High	High	Low
User Expertise	Low	High	Low	High
Re-compilation	Required	Not Required	Not Required	Required

Table 2.3: Dynamic Analysis Techniques comparison

The Table 2.3 shows straightforwardly some limitations of the different Dynamic Analysis techniques. Instrumentation and VM Profiling based techniques engender high runtime overheads whereas AOP rises heavy design and deployment efforts. While the implementation complexity is rather high for Dynamic Instrumentation and VM Profiling, a strong user expertise is also needed for the first one. Finally recompilation is required for two on four techniques.

Additionally, the programmer must be aware that the automated tools cannot guarantee the full test coverage of the source code. Moreover, however how powerful the tools can be, they might yet produce false positives and false negatives. This is why a human code understanding and reviewing is still an absolute necessity.

2.5 Concluding remarks

In this chapter, we summarized related work about program analysis. After defining what program analysis is, we briefly presented some of the static and dynamic approaches with their respective techniques. However, this is by no means an exhaustive presentation of all the approaches and the reader must be aware that the field is far more complex than that.

To complete these theoretical explanations, we presented some popular tools for both approaches and discussed some general dynamic analysis limitations. During the redaction of the chapter, it appeared clearly that the DPA field is quite recent and is at the moment being actively researched.

In the next chapter, we will introduce our own contribution with the development of the proof-of-concept system.

Chapter 3

Development

"For me, open source is a moral thing."
Matt Mullenweg

In this chapter, we present the result of our work on the dynamic program analysis. Introduced in the first chapter, the proof-of-concept system is staged here through the explanation of some key code parts and its developed features.

3.1 Proposed solution

While working on a growing project, there is always a point where it becomes difficult to have knowledge about what data is used for. Indeed variables are one of the most important pieces in a source code and in order to give the programmer a better overview of the variables evolution, this work intents to propose a proof-of-concept system which will not only monitor the data evolution, but also give the possibility to compare the gathered data between different runs.

To achieve such a system, the project has been separated in three different components which will constitute the system. First, a data capture process is monitoring all the needed variables and their evolution during the execution of the reviewed program. Then a data model has been created along with a backup procedure which stores the data in this model. Finally, a web-application processes the extracted data and shows them for reviewing the results. Each mentioned part is described in detail in the following sections.

3.2 Environment

In order to achieve the proof-of-concept system, we chose four technologies. First, we used the *Python* programming language to instrument the data. Python is a widely used high level programming language which has seen these last year an increasing enthusiasm around it and is now the most popular introductory teaching language at U.S. top universities according to Guo [2014]. Thanks to the dynamic nature of Python, which includes a dynamic type system, the real-time collection of object is a straightforward process and therefore it made plenty sense to use it for a proof-of-concept application. More over Python offers a handy integrated Debugger Framework and also good compatibility with other programming language since there are a lot of bindings available. For this project, we used the version 3 of Python because of improved encoding handling.

Secondly, the extracted data is stored in a *MongoDB* Database which is a document-oriented database entering in the new category of No-SQL database systems. MongoDB has the advantage to use JSON like documents with schema and consequently this is why we choose it to store the heterogeneous extracted data.

Finally, the user interface is built with the help of *Python*, *HTML/CSS* and *Javascript*. Python is now an interesting language to develop web applications because of his extensive support of libraries and is used here, with the help of the Flask framework, for the server side process. HTML/CSS and Javascript are used for the presentation of the results.

Additionally, the chosen IDE is PyCharm which is a very complete IDE supporting among others Python web frameworks, databases, code inspection. This IDE is essential for the development of our application because it helped us with its good library handling and its integrated version control repository manager. In order to optimize the development management, we choose the GitHub online tool as version control repository. The deployment during the development of the solution was tested on virtual machine server under Ubuntu Server 14.04.

In order to deploy regularly the newest version of the ongoing work, an automation server named Jenkins was configured. Jenkins is charged to fetch every day the latest prototype on the GitHub repository, create a package of it and install it on the server. If during this process a bug occurs, an e-mail to the interested persons is sent.

In the next section, each module of the proposed system will be exposed in details regarding their functionality and their implementations.

3.3 Data Capture Process

This section presents the crucial developed elements of the data capture process. The data capture process, also called *analyser*, is the core of the system and is based on the Python Debugger Framework (BDB). BDB handles basic debugger functions, like setting breakpoints or managing execution. Thanks to the object-oriented features (classes and functions inheritance) of Python, it is a simple task to rewrite the different functionality as needed for this project. The different features are regrouped in a global class which we called *Yoda*.

3.3.1 Setting up the trace

In order to use the analyser, some code has to be added at the beginning of the target file. The code is necessary to import the module and to set the start point of the tracing phase.

```python
import yoda.analyser
yoda.analyser.db.set_trace()
```

The `set_trace()` function is inherited from the BDB and is needed to start debugging with a Bdb instance from a caller's frame. It is also necessary to stop the trace at the end of the target code with the `yoda.analyser.db.set_quit()` function which set the quitting attribute to `True`. This raises BdbQuit exception in the next call to one of the `dispatch_*()` methods with the goal to prevent further tracing. Indeed, even if the complete file is analysed, this needs absolutely to be set to avoid unexpected further analysis.

The detailed information about the operating of the Python Debugger Framework, is provided in the official documentation [Python-Foundation, 2017].

3.3.2 Initialization

Once the analyzer module is called, the trace is automatically started. The first background step operated by the system is to setup the `Yoda` class along with some global variables needed during the tracing process. The first variable `json_results` (line 2) will be explained further but is basically where the extracted data will be stored. Then, the `instrumented_types` list (line 3) limits the amount of instrumented objects to the given set, it is possible to add further objects if needed. The next 6 variables (line 4-9) are needed for gathering and computing the line numbers, frames and files name. Finally, the `next_backup` variable (line 10) defines a limit of how many lines can be analyzed before flushing the information in the database.

```
1  class Yoda(bdb.Bdb):
2      json_results = None
3      instrumented_types = (int, float, str, list, dict)
4      prev_lineno = defaultdict(int)
5      prev_lineno['<module>'] = 0
6      cur_framename = '<module>'
7      file_name = None
8      file_id = None
9      total_linenb = 0
10     next_backup = 1000
```

As the needed variables are now set up, the script continues with the initialization of the `Yoda` class. Within the class, the connection of the database is created when the system is configured for production mode (line 4).

```
1  def __init__(self):
2      bdb.Bdb.__init__(self)
3      if settings.DEBUG is False:
4          mongoengine.connect(settings.MONGODB)
```

3.3.3 Event Catching

BDB can react to various events during the code execution which are handled by 4 functions : `user_call`, `user_line`, `user_return`, `user_exception`. Each function has been overridden in order to redirect the event to a special handling function called `interaction`.

```
def user_call(self, frame, args):
    self.interaction(frame, 'call', None)
def user_line(self, frame):
    self.interaction(frame, 'line', None)
def user_return(self, frame, value):
    self.interaction(frame, 'return', None)
def user_exception(self, frame, exception):
    self.interaction(frame, 'exception', exception)
```

Once the `interaction` function has been called, the first thing the system does is to check whenever the `file_name` variable is blank or not. If `file_name` is `None`, then a new one is retrieved and applied from the source code file, otherwise the script will continue with the handling of the events.

```
if self.file_name is None:
    self.file_name = inspect.getfile(frame)
```

3.3.4 Event Handling

The first handled event type is the `call` type. This kind of event is normally happening when the frame of the code is changing and thus there is not much to do. Indeed, it just need to capture the frame name (line 2) and catch the line number (line 3). Nothing else special is handled there.

```
1 if event == 'call':
2     self.cur_framename = str(frame.f_code.co_name)
3     self.prev_lineno[self.cur_framename] = frame.f_lineno
4     self.set_step() # continue
```

The succeeding event type is the the `line` type which occurs at each line-break. This event is vital for the data collection and therefore its operating has to be explained separately. First, the interaction function, as before, checks the type of the event and then proceeds to extract the line number which is a key information for the user interface. Then for each line, the interpreted objects have to be caught. This is handled by a external function called `_filter_locals` and called with the frame locals as an argument.

```
locals = self._filter_locals(frame.f_locals)
```

The function itself first creates an empty dictionary which will store the name and the value of each local variable (line 2). The locals starting with a double underscore are ignored and only the specified objects are fetched (line 4 to 6). The function returns the `new_locals` dictionary to the main `interaction` function (line 9).

```
1 def _filter_locals(self, local_vars):
2     new_locals = {}
3     for name, value in list(local_vars.items()):
4         if name.startswith('__'):
5             continue
6         if not isinstance(value, self.instrumented_types):
7             continue
8         new_locals[name] = [copy.deepcopy(value)]
9     return new_locals
```

Then, the locals are stored in a JSON defaultdict object along with the file name, the frame and the line number. At the end, the JSON dictionary is periodically stored in the database in order to flush the data from the memory and enhance the runtime performance. The population of the database is detailed in the next section.

```
if self.total_linenb > self.next_backup:
    self._populate_db()
    self.next_backup += self.next_backup
```

The handling of the `line` event is now finished and the interaction function continues with the two last types. The `return` event only occurs at the beginning of a file for which we just set the main frame name (line 2) and the `exception` event happens when there is an error in the code which is printed out in the console (line 6).

```
1  if event == 'return':
2      self.cur_framename = '<module>'
3      self.set_step()  # continue
4  if event == 'exception':
5      name = frame.f_code.co_name or "<unknown>"
6      print("exception in", name, exception)
7      self.set_continue()  # continue
```

3.3.5 Trace Ending

Finally, the data capture model is ended by the `set_quit()` BDB function which was remodelled for writing the last traced lines (line 7-11).

```
1  def set_quit(self):
2      self.stopframe = self.botframe
3      self.returnframe = None
4      self.quitting = True
5      sys.settrace(None)
6
7      if self.json_results:
8          if settings.DEBUG:
9              print(self.json_results)
10         else:
11             self._populate_db()
```

3.4 Data model

The data model is an in-between layer used for the Data capture model and the user interface. Both modules parts will be explained in this section along with the presentation of the data model itself.

3.4.1 Definition

The data model itself evolved a lot during the development and lead to the finale state which will be presented here. This can be observed in the chosen nomenclature, which sometimes does not exactly correspond to the reality. The best example is the use of the substantive ”file” in the code which actually describes more an analysis instance or a run than the file itself. Thanks to the MongoDB database engine, it is easy to modify the document structure without any database manipulation in contrary to the data structure of relational database systems. This was a great asset which allowed significant time savings in the development of the data model since it changed a many times.

To understand the data model it is a good reminder to enumerate what the data capture model is actually capturing. First the model is searching for objects, i.e. integers, strings and floats variables including for each type their values. These objects are linked with line numbers, which are them-self linked with frames. Finally, each frame is owned by a file (or more specifically a run as it was pointed out previously).

Keeping that in mind the different data structures can be considered as documents and defined the following way in Python. This notation is used further for reading from the database. First, the *line* which has a number and some data (objects):

```python
class Line(EmbeddedDocument):
    lineno = IntField()
    data = DictField()
```

Secondly, the *frame* which has a name and contains one or many lines :

```python
class Frame(EmbeddedDocument):
    name = StringField()
    lines = ListField(EmbeddedDocumentField(Line))
```

Finally, the *file* which has a name, a time-stamp, the content itself (source code) and additionally a revision number gathered from the git repository when available and also the user name of the person who started the analysis. The file contains logically the different frames and is defined in the following way :

```
class File(Document):
    user = StringField()
    revision = StringField()
    filename = StringField()
    timestamp = DateTimeField()
    content = StringField()
    frames = ListField(EmbeddedDocumentField(Frame))
```

3.4.2 Writing to the database

This part of the database handling is directly implemented in the data capture model along with the capture functionality. The process can be called in two different states of the analysis phase :

- The program reached the limit of lines and needs to flush the gathered data into the database. This occurs inside the `interaction()` function which has already been described in the previous section
- The system reached the end of the analysed file and the function `set_quit()` has been called.

Both states induce the call of the `_populate_db()` which is constituted of an `if...else` condition. This condition checks whenever it is the first time the system tries to backup the data or not and calls respectively the `_create_new_file()` or the `_update_file()` functions (line 3 and 6).

```
1 def _populate_db(self):
2     if self.file_id is None:
3         self._create_new_file()
4         self._clear_cache()
5     else:
6         self._update_file()
7         self._clear_cache()
```

If the system needs to create a new document in the database, as already stated it will call the `_create_new_file()` which is explained here in a simplified and step-by-step version. The complete version of the function includes also a compatibility layer for Python 2 but we removed it here for readability reasons. The first step of the creation of a new entry is to fetch each row of the JSON type dictionary (line 2), where the system stored the data until now, and split the data in two separate variables (`module_file` and `frames`).

```
def _create_new_file(self):
  for module_file, frames in self.json_results.items():
```

Then, in order to display also the source code in the user interface, we retrieve the content of the file (line 1-3). Finally, we gather the `user` and the `revision` variables with the git command line tool and create the file object in the database (line 4).

```
1 file = open(module_file, 'r')
2 file_content = file.read()
3 file.close()
4
5 item = File(user=self._get_git_username(), revision=self.
      _get_git_revision_short_hash(), filename=module_file,
      timestamp=datetime.now(), content=file_content)
```

The next step is to create the `frame` document (line 2) along side with each `line` document belonging to this frame (line 4). Finally, we link the frame to the file (line 6) and save the file into the database (line 7). Additionally the variable `file_id`, which was previously defined, is set.

```
1 for name, lines in sorted(frames.items()):
2     frame = Frame(name=name)
3     for lineno, data in sorted(lines.items()):
4         line = Line(lineno = lineno, data = data)
5         frame.lines.append(line)
6     item.frames.append(frame)
7     item.save()
8 self.file_id = item.id
```

Now that we created a first backup in the database for our run, the system will eventually have to update the database with the following analysed lines and therefore it will call again the `_populate_db()` function. This time, because the system set the *file_id* variable just before, the `_update_file()` function is called. We implemented this function in a similar way as the `_create_new_file()` one. First, as in the earlier function the JSON dictionary is looped in order to gather the needed data.

```
def _update_file(self):
  for module_file, frames in self.json_results.items():
```

Then for each frame, the system first checks if it is a new frame or not (line 2) and hence creates it in the database (line 3-4). Finally, we create the new analysed lines and save them in the database (line 5-7).

```
for name, lines in sorted(frames.items()):
    if not File.objects(id=self.file_id, frames__name=name):
        frame = Frame(name=name)
        File.objects(id=self.file_id).update(push__frames=frame)
    for lineno, data in sorted(lines.items()):
        line = Line(lineno = lineno, data = data)
        File.objects(id=self.file_id, frames__name=name).update(push__frames__S__lines=line)
```

3.4.3 Reading from the database

Reading from the database exclusively arises in the user interface module. In standardise the procedure, we chose to use the *MongoEngine* library. The MongoEngine is a Document-Object Mapper for working with MongoDB in Python. Thanks to the use of this library, it is possible to gather the data of a run in the database only with one line of code.

```
file_object = File.objects(id=file_id)
```

This line of code gathers the complete data of a run, but thanks to the API of MongoEngine it is also possible to retrieve specifically the needed data. If these special options are in the interest of the reader, we suggest to refer directly to the MongoEngine documentation.

3.5 User interface

The user interface is a web application which helps the programmer to review the result of the data capture model. In this section, we decided to focus on the features rather than on the code for different reasons. First, we do not consider the user interface code here as the core knowledge of the developed system. Then the complete web application represents almost 1000 lines of code and would stretch considerably out this report with few added value. Finally a non negligible part of the code fulfils a visual and presentation purpose rather than actual features.

3.5.1 Technologies

The web application is based on the Python web framework *Flask* which is intended to be as lightweight as possible. The flask micro-framework comes with some handy features such as built-in development server and debugger, integrated unit testing support, RESTful request dispatching, or *Jinja2* templating. Flask is normally designed to connect with standard SQL databases but with the help of the *flask-mongoengine* extension the process is pretty straightforward.

The shaping of the web-application is indeed done with the help of *HTML/CSS* and *Javascript*. For purposes of standardization, the *Bootstrap* framework came in help which includes also the convenient *JQuery* javascript framework. Bootstrap is a popular HTML, CSS, and JS open-source framework for developing front-end projects on the web. As for Jquery it is a fast, small, and feature-rich JavaScript library which makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers.

Some other libraries came also during the development to format the data out of the database. With this aim in mind, we first used the DataTables library which makes formatting data in table a child's play. Additionally we are using the *Highcharts* Javascript library to generate every graphs and *pygments* to highlighting the Python syntax.

3.5.2 The cockpit

The user interface is composed of three main pages. The first one, represented by the Figure 3.1, is the index page which shows an overview of the different runs stored in the database. This page also acts as a cockpit which allows the user to view runs in detail, select runs for comparison and to delete unwanted runs from the database. The runs are searchable and can be sorted by dates, file name, Git revision or user name. It is also possible for the user to configure how many runs should be shown per page.

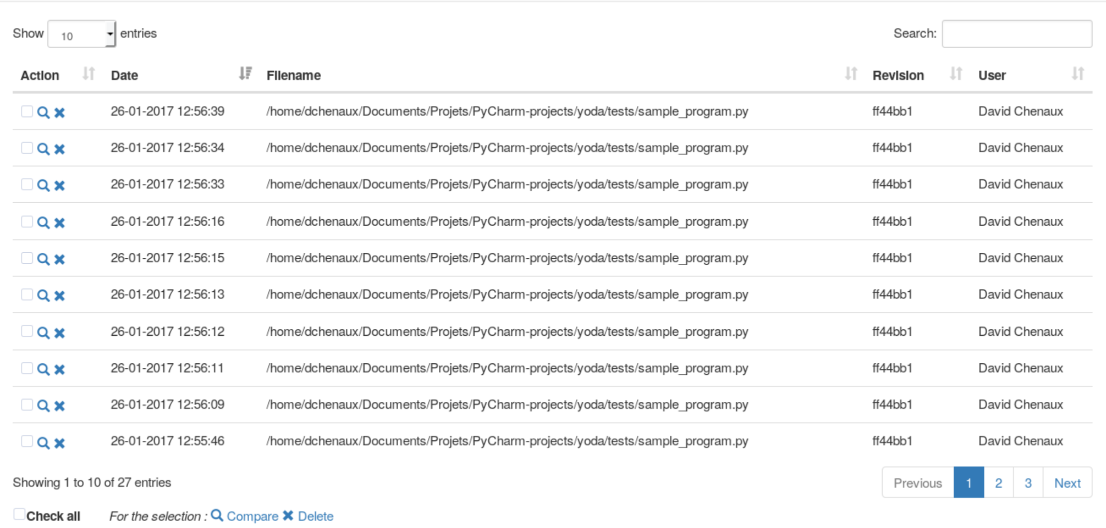

FIGURE 3.1: The cockpit

3.5.3 The file reviewer

Once the user selected a run in the cockpit, he will be directly redirected to a the corresponding results page as illustrated by the Figure 3.2.

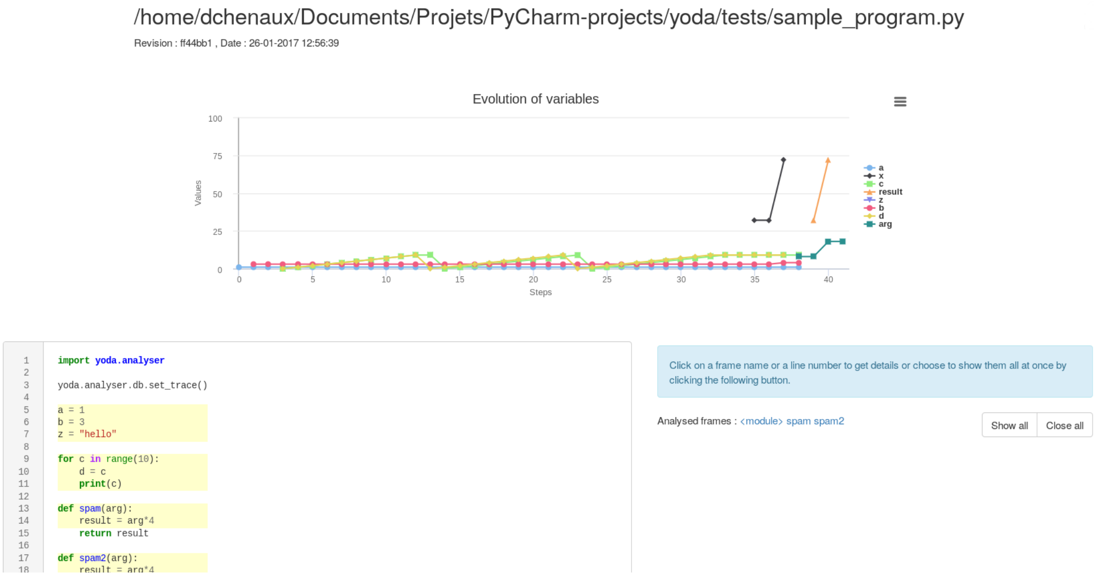

FIGURE 3.2: Reviewing a file

This page is separated in 4 main sections. First, at the head of the page there is some basic information about the run such as the file name, the date of the run or the git revision. Directly underneath, when possible, a plot graph is computed and shows by default all the available objects. The user can choose to hide some variable and the scope of the graph is automatically adapted to the remaining values. Additionally, the chart can also be printed and exported in several different formats.

Next, in two vertical columns, the user will find in the left part the complete source code of the analyzed file. The pieces of code which were genuinely analyzed are highlighted in a light yellow color and moreover every line has been numbered and syntactically colored. The line numbers are clickable and open a panel in the right column with some further information about the analyzed objects. Each panel contains a header with the frame name and the line number, and a body with the objects names, their values and a small inline graph representing the evolution of the value. The Figure 3.3 shows in detail these described features. Additionally on the top of the right column some buttons allow to show all the panels, close them or selectively open all panels linked to a frame.

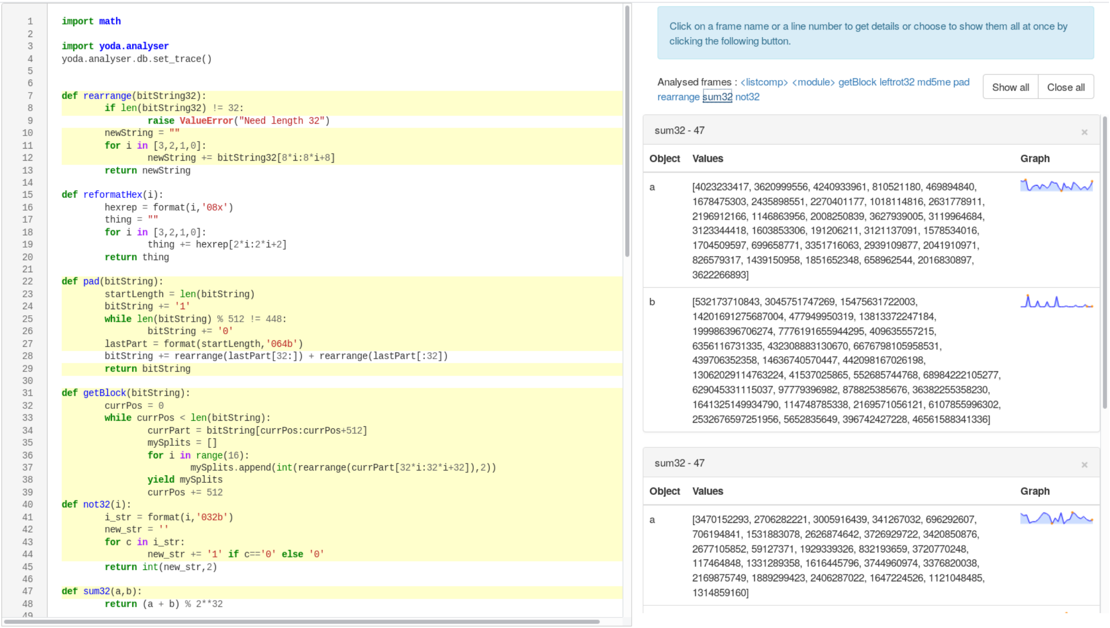

FIGURE 3.3: Sources code and detail panels

3.5.4 File comparison

If needed the user has also the possibility to select several files in the cockpit in order to compare them. This is done by checking the needed runs and clicking the "compare" link at the bottom of the page. Doing so will redirect the user to the start page of the comparison. From this page, each file can be inspected and the user is given the choice of which object he wants to select for the comparison as shown on the Figure 3.4.

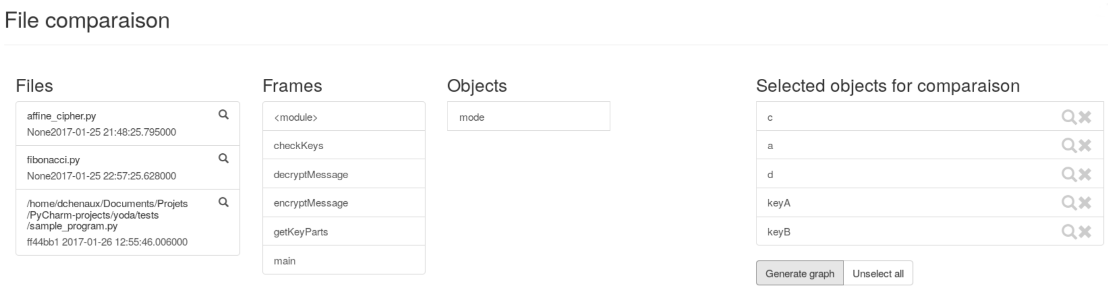

FIGURE 3.4: Runs comparison

For each selected object, the user has the possibility to see the source file or eventually to deselect it. When he is happy with his selection, graphs can be generated with the triggering of the "Generate graphs" button. For each run a graph will be created and displayed in a way which facilitates comparison as shown on Figure 3.5.

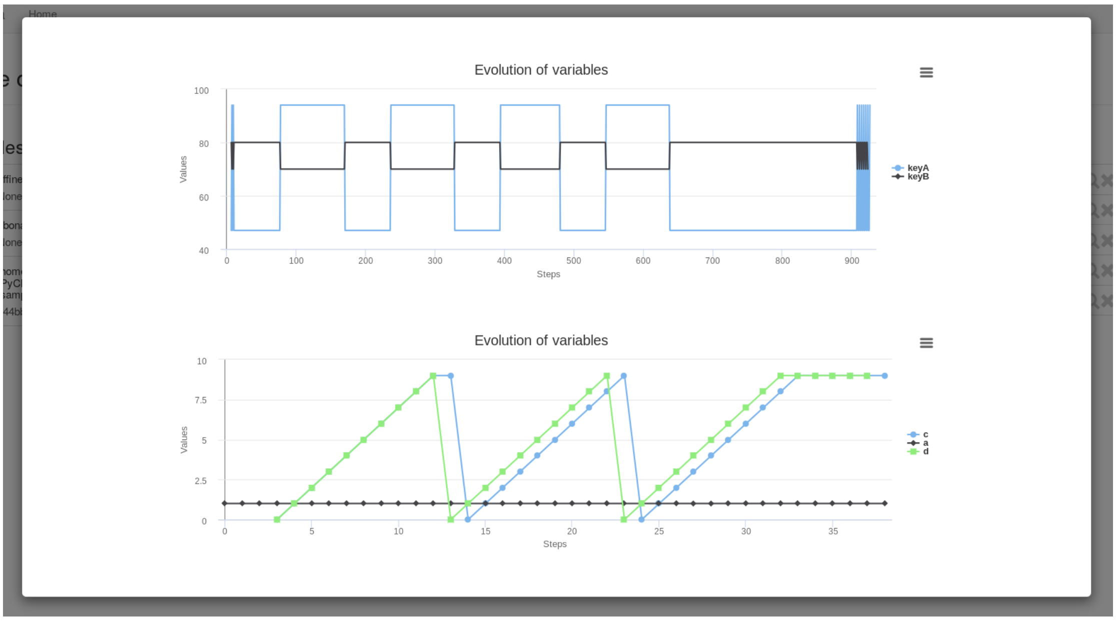

FIGURE 3.5: Generated graphs for comparison

3.6 Concluding remarks

In this chapter, we gave a brief but complete insight of the implementation of our solution. It was quite a challenge to summary 6 months of development, over 2000 lines of code in a short and comprehensive chapter. We hope we were able to give the reader a good insight of operating method of our developed system. In the next chapter, we are giving a complete guide to install and start the system.

Chapter 4

Installation guide

"If Microsoft ever does applications for Linux it means I've won."
Linus Torvalds

Because the installation guide is often left out in academic works, we want to dedicate a special chapter to the process. Here the reader will be able to learn how to deploy the system on his own machine by first configuring his environment and then deploy the application by installing of the provided packages or by compiling from the source code.

4.1 Setting up the environment

In order to use the developed tool, it is highly recommended to install it on a system providing a GNU/Linux distribution. The tool should work under Windows or MacOS as the used libraries are all cross-platform, but the software has not been tested under these platforms. For those who might not want to switch to a native GNU/Linux system it is needless to say that it will also work in a virtual machine. As it was used during the development and the testing phases we strongly recommend a Fedora distribution and therefore this guide is based on this distribution's commands.

4.1.1 Python installation

As the main used language is Python and more specifically Python 3, the first step is to verify its installation and in case it would not be present install the needed packages by using the following command :

```
sudo dnf install python3 python3-pip
```

4.1.2 MongoDB installation

Next step is to install the second dependency : the MongoDB database engine. First thing first, the repository has to be added to the install sources and can be done by creating a `/etc/yum.repos.d/mongodb-org-3.4.repo` file containing :

```
[mongodb-org-3.4]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/redhat/7/mongodb-org/3.4/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-3.4.asc
```

Then the latest stable package (currently version 3.2.11) can be installed with the dnf package manager and then directly launched with the following command :

```
sudo dnf install mongodb-org
sudo service mongod start
```

In case of installation problems we ask the reader to refer to the official documentation.

4.1.3 Setting up a virtual environment

The required environment is now set up. Additionally, the user will certainly want to create a python virtual environment in order to keep the original installation clear. First, the required package has to be installed :

```
sudo pip3 install virtualenv
```

Then in a new directory, the virtual environment is created :

```
virtualenv -p /usr/bin/python3.5 venv
```

Finally to start using the new created environment :

```
source venv/bin/activate
```

More information about using a virtual environment is available online. [Reitz, 2016]

4.2 Installation

4.2.1 Package installation

In order to simplify the installation process, we build a packaged version of our system and it is ready to be downloaded from the projects GitHub page. The installation process is straightforward since it is a pip package. Assuming the reader followed the instruction in the past section, the following command will install the package and its required dependencies.

```
pip install yoda-1.0.tar.gz
```

The installed files are now located in `venv/lib/python3.5/site-packages/yoda/`.

4.2.2 Package creation

In case the reader wishes to do some modification of the system and wants to create a new packaged version, we created a `setup.py` file which allows easy package creation

```
python3 setup.py sdist
```

This package can be installed the same way as described a bit earlier.

4.3 Usage

Now that the complete system is installed, the user interface can be launched with the following command :

```
python venv/lib/python3.5/site-packages/yoda/web_exec.py
```

The user interface should be now locally accessible in any browser at the `http://127.0.0.1:80` address.

To try a first analysis we recommend the user to download the sample script which is also available on the GitHub repository and launch it from the command line.

4.4 Concluding remarks

To conclude the installation guide, we wanted to point out that only the basic setup was explained here. Indeed, thanks to the use of the flask framework a lot of power user configuration is possible, such as running the user interface through a different port, making the interface accessible from other machines, deploying it on many popular web-servers. In the next chapter, we evaluate the performance of the whole system.

Chapter 5

Experiments

In this section different type of experiments are conducted in order to test and check the performance of the developed software.

5.1 Test script and machine

In order to conduct the different experiments, we chose a test script available online on *TheAlgorithms* GitHub page. We selected this test script because MD5 checking is a really common task and generates many numerical values for which our solution is the most intended.

Concerning the testing machine itself, we chose to run the tests directly on our personal laptop because it correspond to the usage the system is intended for : a personal debugging tool. The specifications of the machine are the following ones :

- Lenovo Thinkpad T460p
- CPU : Intel Core i7-6700HQ @ 2.60GHz x 8
- OS : Fedora 25 64bits
- GPU : Intel HD Graphics 530
- RAM : 15.1Gio

5.2 Objectives

For the experiments, we want here to test two aspects of our system : the memory usage and the runtime. To be able to measure these two parameters, we use a Python tool called *Memory profiler*. This tool is for monitoring memory consumption of a process as well as line-by-line analysis of memory consumption for python programs. It allows also to plot memory consumption as a function of time and measure the execution time of the target script. With these extracted parameters, the tool is also capable to plot graphs. Our aim is to verify if dynamic analysis induces overhead with the help of the memory profiler and our dynamic analysis solution.

5.3 Performances

5.3.1 Test script

The first necessary step in order to conduct our experiments correctly, is to create a reference run from which we will be able to compare our results. In this idea, we profiled the memory consumption and the runtime of the test script without plugging in our system. The Figure 5.1 illustrates the results of the Memory profiler: the test script uses a total of 13.45MiB of memory for a runtime of 0.1s.

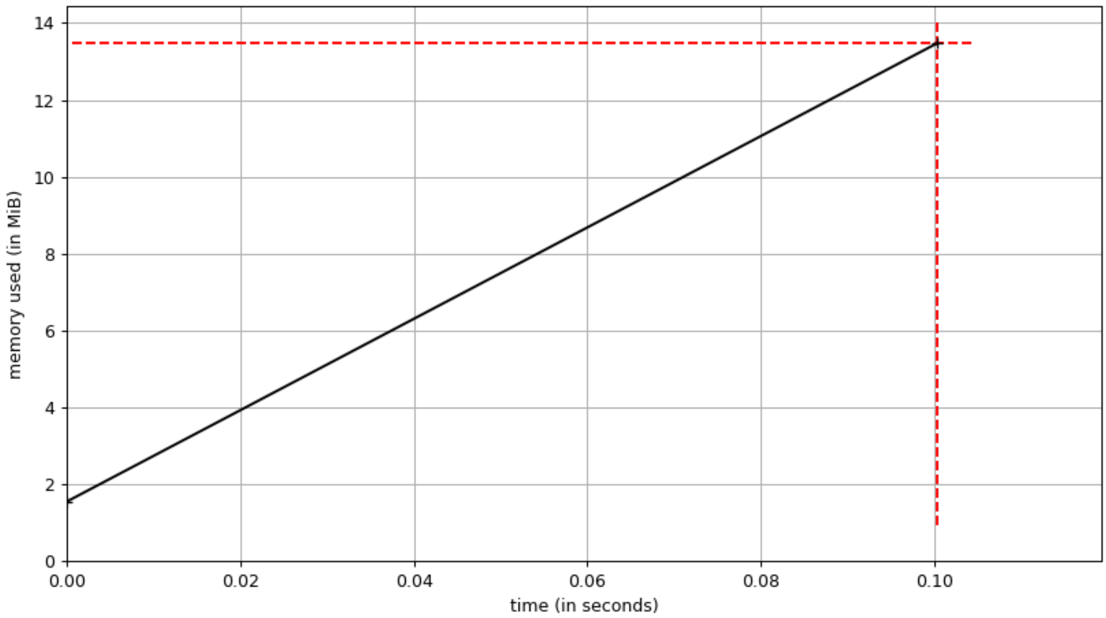

FIGURE 5.1: Memory usage and runtime of the test script

5.3.2 Data capture

The first process in our solution which could induce overhead is the data capture process and therefore, we want here to test its performance. In order to exclusively determine the memory usage and runtime of the data capture process, we activated the debug mode of the system to avoid the database writing process.

The Figure 5.2 presents the results of the memory profiler analysis. As we supposed, the process is inducing overhead. The execution time is now of 0.6s which represents a multiplication by 6 compared to the reference test. Concerning the memory the script now needs 27.6MiB of RAM which is more than the double of the original run.

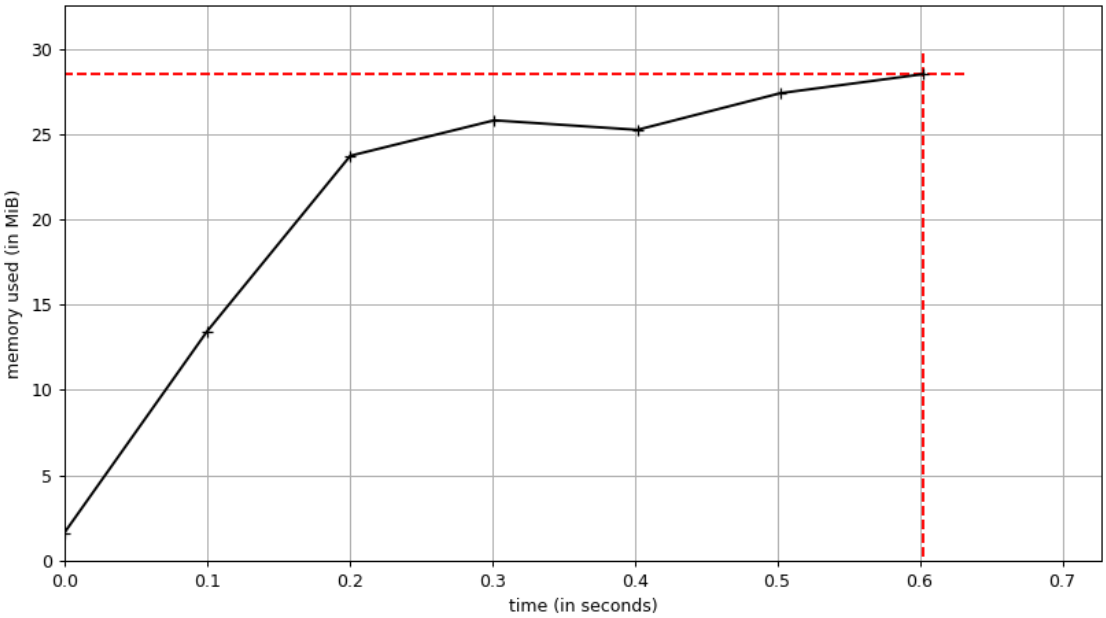

FIGURE 5.2: Memory usage and runtime with the Yoda system activated

These results could be seen as deceiving but in fact are inevitable because of the nature of our solution. Indeed, because we want to track the value of each variable at each line, the values are not overridden as in the normal run of the script. Instead, for each value, we store in the memory a copy of the variable and its data.

5.3.3 Database writing

The second process of interest which we want to test here is the database writing process. By activating this phase in the tool, we want to see if it also induces an overhead.

As it is shown in the Figure 5.3, the introduction of the data writing process in our test does not induce any significant extra memory usage which is now at the maximum around 29MiB. Nonetheless, the writing process induce a runtime overhead of 0.5s to now reach a total of 1.21s.

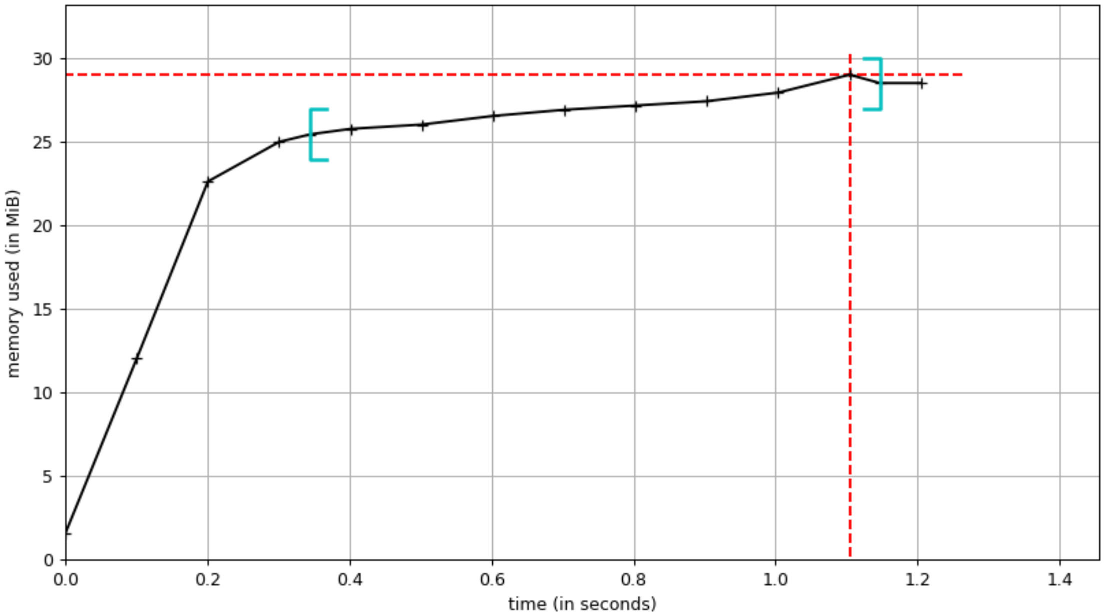

FIGURE 5.3: Memory usage and runtime with the database writing

5.4 Concluding remarks

In the Chapter 2, we already introduced some limitation of dynamic analysis tools, which is often heavy runtime overhead. We showed in the experiments that our solution is not an exception and induces even on a small script a heavier memory consumption and a significant runtime overhead. An idea to optimize the process could be asynchronously writing via a message queue.

It has to be pointed out that the memory profiler is not an exact tool. Indeed, according to Pedregosa [2016], this module gets the memory consumption by querying the operating system kernel about the amount of memory the current process has allocated, which might be slightly different from the amount of memory that is actually used by the Python interpreter. Also, because of how the garbage collector works in Python the result might be different between platforms and even between runs.

Chapter 6

Conclusion

In this final chapter, we summarize the work done in this master thesis and give complementary remarks along with potential improvements that came into mind during the development and then the writing of the report.

6.1 Final remarks

The main goal of the thesis was to develop a system which could pursue effective code maintenance with continuous data collection. This report is the summary of 6 months of development and in here we additionally presented related works in order to build a theoretical background. Our proposed solution is explained in a dedicated chapter along with a handy installation guide. The explanations were focused on the core features of our system because we did not want to overwhelm the reader with too much technical aspects. Indeed, even if the coding of the user interface was around half of the coding time, we only presented the final features without spending time about its programming implementation. Additionally, we conducted some experiments on the developed applications to find out if it engendered some drawbacks. During the development, we thought also about some future work which could not be included in the scope of a master thesis. These points are exposed in the next section.

6.2 Personal remarks

On a personal point of view, this master thesis was the second experience in the development of a complete high technical subject. But even though, I had already the chance to develop a bin packing system for my Bachelor Thesis, this was a completely new challenge in a absolutely unfamiliar field. As the development was cut in two half, because I was in Austria for an internship during the summer, I also lost some time to step back into the code after this long pause. On the other hand, this brought a fresh view on the code and some parts that seemed really good before were rewritten in a more clever way.

Also, the use of Python was a new challenge for me as I was more used to develop in PHP for this kind of application and therefore I had to learn to know and use all the different libraries. The choice of MongoDB was also a discovery as I was more used to work with relational databases.

Finally, to conclude this work, I learned the hard way that programmers may plan as well as they want the available time, but there will always be unexpected events. Particularly on this kind of research field, it was quite difficult for me to estimate how much time could be needed to develop an idea. I can clearly remember that after 2 months of development I already had the felling that I was close to the end. This is something I want to take with me for my future projects.

6.3 Future work

As the developed application is only a proof-to-concept system, there is plenty of room for future improvements. In this final section, we want to propose the reader some topics and features that came into mind during the development and the final experiments. These improvements could lead to a software which could be used by final users. Following, in order of importance, some of the further work we could think about :

- At the time, the data capture model has some issues with complex data retrieval. Indeed, it is more than likely that the system will throw an error when the analyzed script opens large text files in order to work with them. Our insight is that there is a conflict between the JSON way of the MongoDB database and the pythonic way of storing dictionaries ;
- Currently, the system was focused on numeric variables, even if it is capable of retrieving strings. We would suggest some enhancement in that way by first removing the string variables from the plot charts and then perhaps find an another way to represent them. One idea could be to create a kind of tree mapping of the different values ;
- The plot graphs definitely need a zoom functionality. In deed, we observed that with scripts which can generate a big amount of values it is increasingly hard to use the graphs ;

- The following point is something we already discussed at the beginning of the coding phase and it is still an open point. In the actual proposed solution, all the values are stored in the database. In the example of a looping variable which could range from 1 to 1 million, is it really useful to store every value in between ? We think that there could be a simpler way to store such data ;

- A last point concerning the user interface itself, thanks to some new framework like *Electron*, it would be pretty easy to adapt it to a standalone application.

Appendix A

Glossary

AOP Aspect Oriented Programming

AST Abstract Syntax Tree

BDB Python Debugger Framework

CPU Central processing unit

DPA Dynamic Program Analysis

GPU Graphical processing unit

JPDA Java Platform Debugger Architecture

JSON JavaScript Object Notation

JVMPI Java Virtual Machine Profiling Interface

JVMTI Java Virtual Machine Tools Interface

Libre or Free software, is distributed under terms that allow users to run the software for any purpose as well as to study, change, and distribute the software and any adapted versions.

MD5 The MD5 algorithm is a widely used hash function producing a 128-bit hash value

OS Operating system

PDB The Python Debugger

pip Pip Installs Packages is a package management system used to install and manage software packages written in Python

RAM Random-access memory

SDK Software Development Kit

SMT Satisfiability Modulo Theories

SPA Static Program Analysis

VM Virtual Machine

Appendix B

License of the software

Copyright (c) 2016 DAVID CHENAUX

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Bibliography

Patrick Cousot. Abstract interpretation, August 2008. URL https://www.di.ens.fr/~cousot/AI/. [Online; accessed 9-January-2017].

Mireille Ducassé and Jacques Noyé. Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming*, 19:351–384, 1994.

Bruno Dufour, Laurie Hendren, and Clark Verbrugge. * j: a tool for dynamic analysis of java programs. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–307. ACM, 2003.

Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira. An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal*, 2009.

Anjana Gosain and Ganga Sharma. *A Survey of Dynamic Program Analysis Techniques and Tools*, pages 113–122. Springer International Publishing, Cham, 2015. ISBN 978-3-319-11933-5. doi: 10.1007/978-3-319-11933-5_13. URL http://dx.doi.org/10.1007/978-3-319-11933-5_13.

Philip Guo. Python is now the most popular introductory teaching language at top u.s. universities, 2014. URL http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext. [Online; Accessed 7-march-2017].

Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Lubomír Bulej, Aibek Sarimbekov, Walter Binder, and Petr Tůma. Introduction to dynamic program analysis with disl. *Science of Computer Programming*, 98, Part 1:100 – 115, 2015. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2014.01.003. URL http://www.sciencedirect.com/science/article/pii/S0167642314000070. Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).

F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 2004. ISBN 9783540654100.

Flemming Nielson and Hanne Riis Nielson. *Correct System Design: Recent Insight and Advances*, chapter Type and Effect Systems, page 114–136. Springer International Publishing, 1999.

Fabian Pedregosa. Memory profiler github page, 2016. URL https://github.com/fabianp/memory_profiler. [Online ; accessed 16-march-2017].

Python-Foundation. Python documentation, pdb — the python debugger¶, 2017. URL https://docs.python.org/3.6/library/pdb.html. [Online; accessed 17-January-2017].

Kenneth Reitz. Virtual environements. http://docs.python-guide.org/en/latest/dev/virtualenvs/, 2016. [Online; accessed 25-January-2017].

Didier Rémy. Type systems for programming languages, January 2017.

Paramvir Singh. Design and validation of dynamic metrics for object-oriented software systems. 2013.

Wikipedia. Program analysis — wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/w/index.php?title=Program_analysis&oldid=732080552. [Online; accessed 7-January-2017].

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Faculté des sciences économiques et sociales
Wirtschafts- und sozialwissenschaftliche Fakultät
Boulevard de Pérolles 90
CH-1700 Fribourg

DECLARATION

Par ma signature, j'atteste avoir rédigé personnellement ce travail écrit et n'avoir utilisé que les sources et moyens autorisés, et mentionné comme telles les citations et paraphrases.

J'ai pris connaissance de la décision du Conseil de Faculté du 09.11.2004 l'autorisant à me retirer le titre conféré sur la base du présent travail dans le cas où ma déclaration ne correspondrait pas à la vérité.

De plus, je déclare que ce travail ou des parties qui le composent, n'ont encore jamais été soumis sous cette forme comme épreuve à valider, conformément à la décision du Conseil de Faculté du 18.11.2013.

……………………………..……, le .... 20..........

..……..................................…
(signature)