UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Effective code maintenance with continuous data collection

by

David Chenaux

Thesis for the Master of Science in Computer Science
Supervised by Prof. Dr. Philippe Cudré-Mauroux

eXascale Infolab
Department of Informatics - Faculty of Science - University of Fribourg

January 27, 2017

UNIVERSITY OF FRIBOURG
Faculty of Science
Department of Informatics
eXascale Infolab

Thesis for the Master of Science in Computer Science
Supervised by Prof. Dr. Philippe Cudré-Mauroux

by David Chenaux

Abstract

This master thesis proposes a proof-to-concept system for effective code maintenance with continuous data collection.

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Integrated development environments have been around for a few decades already, yet none of the modern IDEs was able to successfully integrate their source code editors with 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*.

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 IDEs was able to successfully integrate their source code editors with the actual data stream flowing through the code. This is why the present project, which objectives are defined in the next section, is aiming to contribute to the subject.

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 all values for all variables in source code and store them somewhere, with further possibility to easily retrieve saved values. The system should also provide an API to the storage in order to make the data accessible for navigation and display in third-party applications. Also, a basic visualizing interface will also be included in order to allow an easy review of the results. Finally, an evaluation of system's performances will be established through different experiments.

1.3 Organization

The thesis is divided in four main sections :

1. **Related work**: In this first chapter of the thesis, an insight of the existing work on the field *program analysis* will be presented 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 is focusing on the development of the proof-to-concept system with a presentation of the proposed solution and detailed information about its structure.
3. **Installation guide**: Simply an installation guide of the software which describes the needed environment, the package installation and the compilation of the system.
4. **Experiments**: Finally in this section a few experiments will be conducted in order to test and check the performance and results of the software.

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 ambition, it is unavoidable 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 some technical approaches. Following, the third section introduces some trendy 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 running the software and the DPA which is obviously fulfilled during runtime. [Wikipedia, 2016]

The SPA is a really simple 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 because of the usage of run-time features like dynamic binding, polymorphism, threads etc. To remedy this situation, developers call on DPA which can, after Marek et al. [2015], gain insight into the dynamics and runtime behavior of those systems during execution. Moreover, because the run-time 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 run-time programming lan- guage features like polymorphism, dynamic bind ing, threads etc.	Lacks in handling run-time programming lan- guage features.
Incurs large run-time 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 to 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 run-time 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, some different approaches are to be exposed 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 are proposed, 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 run-time 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 run-time 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-to-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 run-time 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, run-time 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 run-time 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 to be really well 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 run-time 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 tried to summaries some 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 convoluted than that.

To complete this theoretical explanations, we presented some popular tools for both approaches and spoke about some general dynamic analysis limitations. During the redaction of the chapter, it appeared clearly that the DPA field is quite recent and therefore only a few researches were conducted on the subject.

In the next chapter, we will introduce our own contribution with development of the proof-to-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 analysis field. Introduced in the first chapter, the proof-to-concept system is staged here through the explanation of some key code parts and its exciting developed features.

3.1 Proposed solution

While working on a growing project, there is always a point where it becomes difficult to keep an eye on all the variables. In order to give the programmer an overview of the variables evolution, this work intents to propose a proof-to-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 model 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 exposed in details in the following sections.

3.2 Environment

During the development four technologies pushed them self on us in order to develop the wanted characteristics.

First, *Python* which was ordered by the project supervisor i s being used to capture the data. Python is a widely used high level programming language which has seen these last year an increasing enthusiasm around it, especially for web based applications. Thanks to the dynamic nature of Python, which includes a dynamic type system, the real-time collection of object is a pretty straightforward process and therefore it made plenty sense to use it in our project. 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, the newest version 3 of the programming language was chosen because of the better handling of encodings.

Secondly, the extracted data is stored in a *MongoDB* Database which was also wished by the supervisor. MongoDB 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 was a clever choice to store the heterogeneous extracted data.

Finally, the user interface was built with the help of *Python*, *Html/CSS* and *Javascript*. As already said, Python is now an interesting language to develop web applications and was used here, with the help of the Flask framework, for the server side process. HTML/CSS and Javascript were used for the presentation of the results.

Additionally, the chosen IDE was PyCharm academic edition version 2015 and then 2016. PyCharm is a very complete IDE which supports among others Python web frameworks, databases, code inspection. In order to optimize the development management, the GitHub online tool was chosen as version control repository. The deployment during the development of the solution was tested on virtual machine server under Ubuntu Server 14.04. The server has been provided by the Department of Informatics at the University of Fribourg and is accessible internally (on the 30 January 2017) at `http://diufpc115.unifr.ch/`. This server will also be used for the experiments in the chapter 5.

In order to deploy regularly the newest version of the ongoing work, an automation server named Jenkins was configured. Jenkins was 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 occurred, an e-mail to the interested persons was 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 Model

This section presents the crucial developed elements of the data capture model. The data capture model, or *analyser* as it was called during the development, 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 programming, the classes and the function inheritance, it is a smooth task to rewrite the different functionality as needed for this project.

The development began with study of a script provided by Roman Prokofyev which is implementing some basic data capture operations derived from the Python debugger framework. The understanding of the developed concepts was the first step to the creation of the data capture model. The analyser consists in 240 lines of code and some of the essential functions are explained here.

3.3.1 Setting up the trace

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

```
1 import yoda.analyser
2 yoda.analyser.db.set_trace()
```

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

For further information about the operating of the Python Debugger Framework, we advise the reader to refer to the official documentation [Python-Foundation, 2017].

3.3.2 Initialization

Once the analyzer module 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 instrumented objects to this list, it is possible to add further objects if needed. The next 6 variables (line 4-9) are needed for gathering and computing 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 also 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 rewritten in order to redirect the event to a self-written handling function called `interaction`.

```
1  def user_call(self, frame, args):
```

```
2         self.interaction(frame, 'call', None)
3     def user_line(self, frame):
4         self.interaction(frame, 'line', None)
5     def user_return(self, frame, value):
6         self.interaction(frame, 'return', None)
7     def user_exception(self, frame, exception):
8         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.

```
1     if self.file_name is None:
2         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 is really short. 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 in separated steps. First, the interaction function, as before, checks the type of the event and then proceed 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 in option.

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

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

```
def _filter_locals(self, local_vars):
    new_locals = {}
    for name, value in list(local_vars.items()):
        if name.startswith('__'):
            continue
        if not isinstance(value, self.instrumented_types):
            continue
        new_locals[name] = [copy.deepcopy(value)]
    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 run-time performances. The population of the database is detailed in the next point.

```
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 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).

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

3.3.5 Trace Ending

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

```
def set_quit(self):
    self.stopframe = self.botframe
    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 opposition with the data structure of relational database systems. This was a great asset which allowed tremendous saving time in the development of the data model since it changed a significant number of 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. integer, string, float variables, and their values. These objects are linked with line number, 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):

```
1   class Line(EmbeddedDocument):
2       lineno = IntField()
```

```
3       data = DictField()
```

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

```
1   class Frame(EmbeddedDocument):
2       name = StringField()
3       lines = ListField(EmbeddedDocumentField(Line))
```

Finally, the *file* which as 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 the whole is defined this way :

```
1   class File(Document):
2       user = StringField()
3       revision = StringField()
4       filename = StringField()
5       timestamp = DateTimeField()
6       content = StringField()
7       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 analyzing phase :

- The program reached the limit of lines and need to flush the gathered data into the database. This occurs inside of the `interaction()` function which has already been described in the previous section.
- The system reached the end of the targeted software 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 need 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 has been removed 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 data has been stored until now and store the data in two separate variables (`module_file` and `frames`).

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

Then, in order to display also the source code in the user interface, the content of the file retrieved (line 1-3) and as all the needed information are already there, the file document type can be created (line 4). The `user` and the `revision` variables are gathered from two function which retrieve the git repository information, but will not be explained in this report.

```
1  file = open(module_file, 'r')
2  file_content = file.read()
3  file.close()
4  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, the frame is linked to the file (line 6) and the file can be saved 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 a first backup has been created in the database for our run, the system will probably have to update the database with the following analyzed lines . With this end in mind, the `_update_file()` has been implemented and is presented the same way as the foregoing function. First, as in the previous function the JSON dictionary is looped in order to gather the needed data.

```
1  def _update_file(self):
2    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 create it in the database (line 3-4). Finally the new analyzed lines are created and saved in the database (line 5-7).

```
1  for name, lines in sorted(frames.items()):
2      if not File.objects(id=self.file_id, frames__name=name):
3          frame = Frame(name=name)
4          File.objects(id=self.file_id).update(push__frames=frame)
5      for lineno, data in sorted(lines.items()):
6          line = Line(lineno = lineno, data = data)
7          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 order to cut down the procedure, the *MongoEngine* library has been chosen. The MongoEngine is a Document-Object Mapper for working with MongoDB from Python. Hence the use of this library allows to gather the data of a run in the database only with one line of code.

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

This line of code gather the complete data of a run, but thanks to the API of MongoEngine it is also possible to retrieve specifically the needed data. If this 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, the focus will be made on the features rather

than on the code for different reasons. First, the user interface code is not considered here as the core knowledge of the developed system. Then the complete web application represent almost 1000 lines of code and would stretch considerably out this report with few added value. Finally a non negligible part of the code fulfill 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, the first used library was DataTables which makes formatting data in table a child's play. Additionally the *Highcharts* Javascript library is used to generate every graphs and *pygments* helps for 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 simply propose an overview of the different runs stored in the database. This page also act 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.

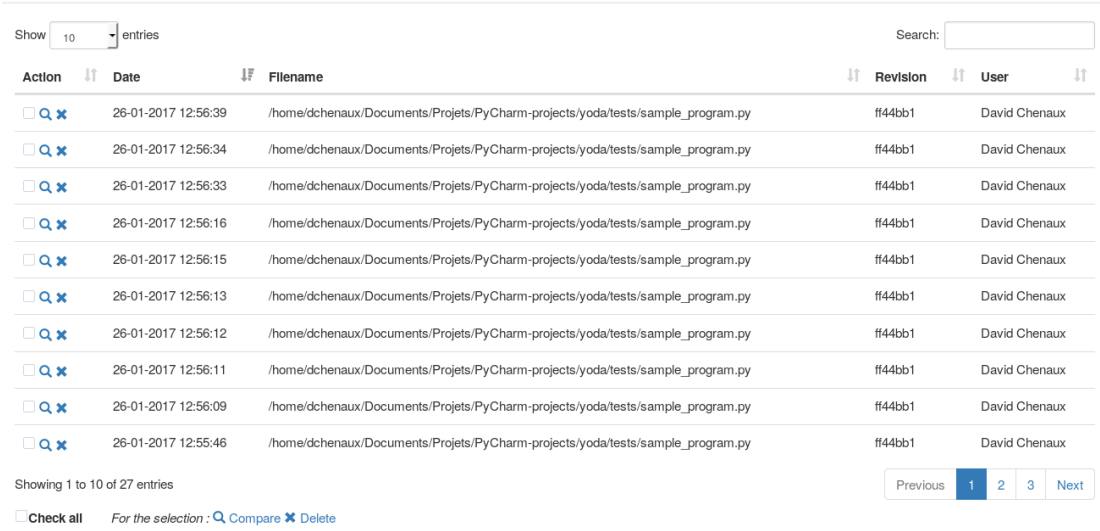

FIGURE 3.1: The cockpit

3.5.3 The file reviewer

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

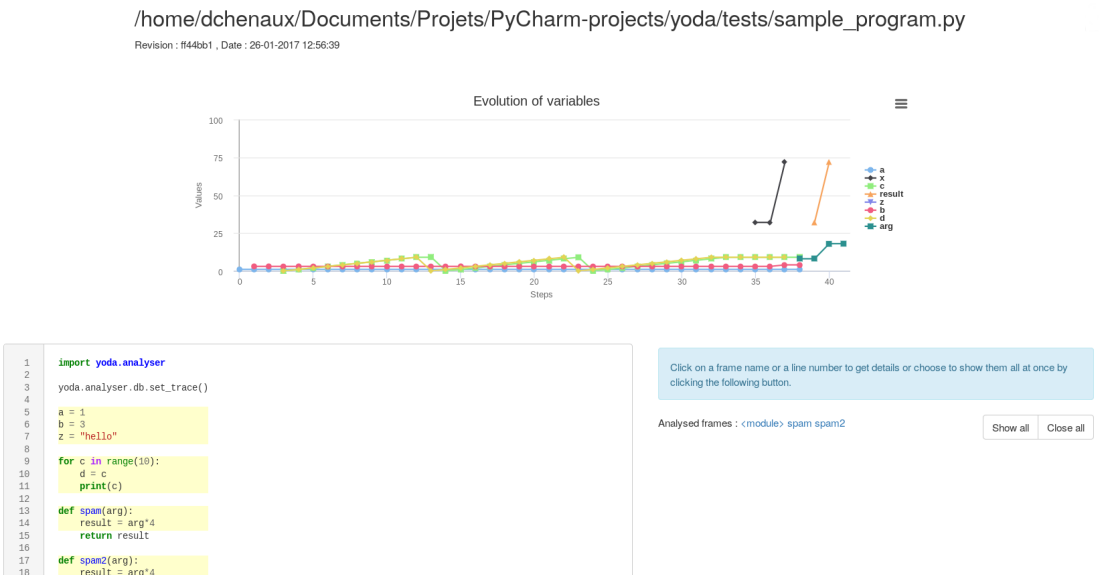

FIGURE 3.2: Reviewing a file

This page is separated in 4 main sections. First, at the head of the page the user gets 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 directly from the graph 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.

A bit further, separated 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 left column with some further information about the analyzed objects. Each panel contains a header with the frames name and the line number, and a body with the objects names, their values and a small inline graph resuming 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 panel, close them or selectively open all panels linked to a frame.

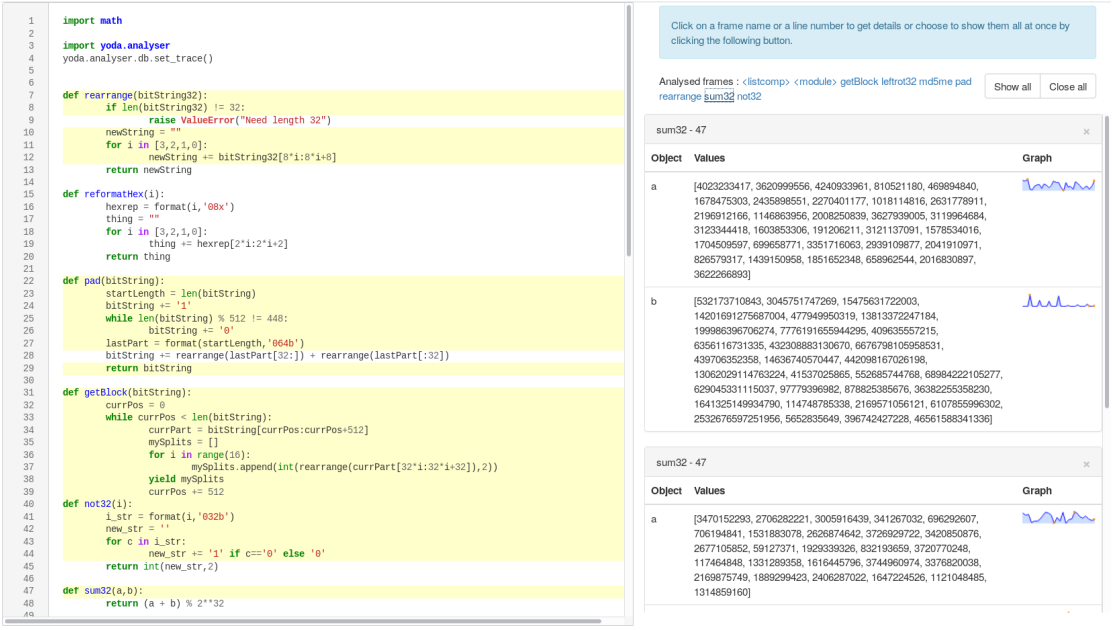

FIGURE 3.3: Sources code and detail panels

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 on 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.

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 an disposed in a way which facilitates comparison as shown on Figure 3.5.

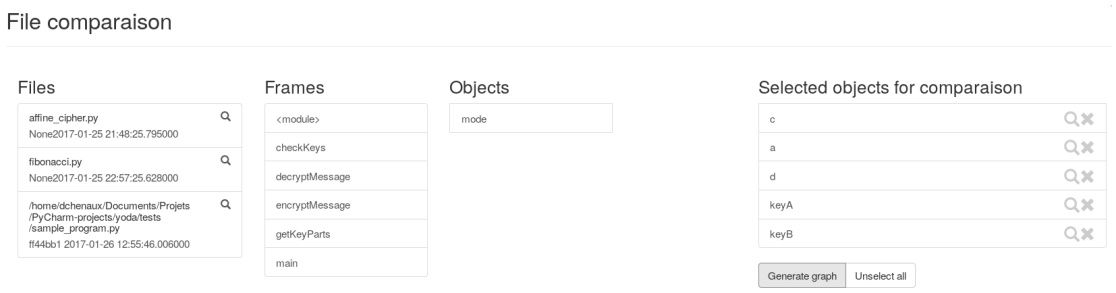

FIGURE 3.4: Runs comparison

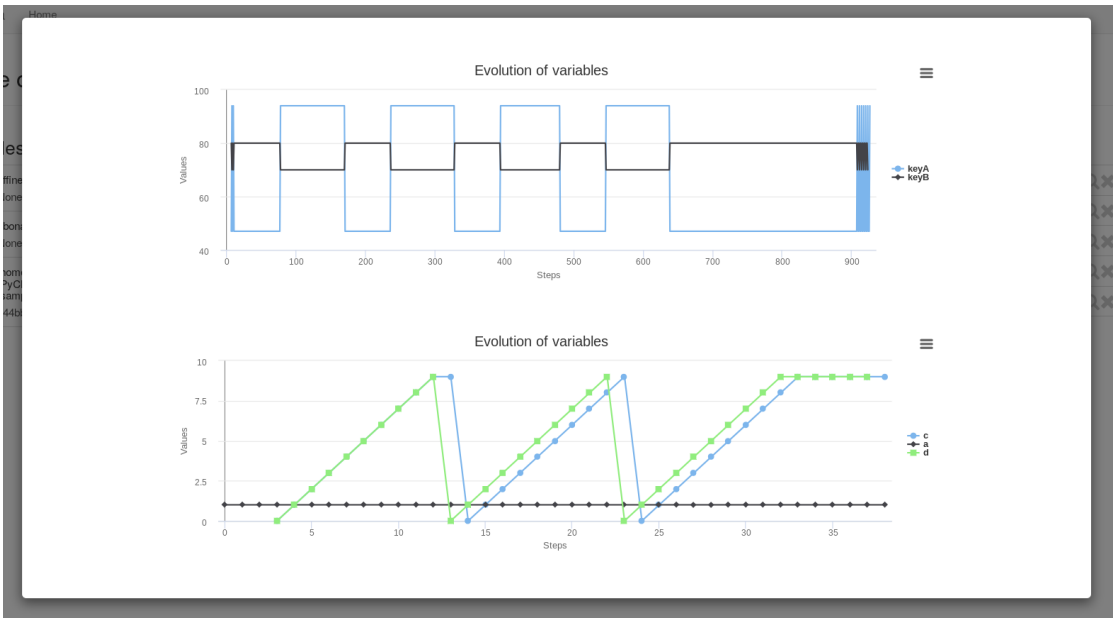

FIGURE 3.5: Generated graphs for comparison

3.6 Concluding remarks

In this chapter, we tried to give an brief but complete insight of the implementation of our solution. It was quite a challenge to summaries 6 months of development, over 2000 lines of code (around 32'000 with all libraries included !) in a short and comprehensive chapter. The use of Python was a new challenge for us as we were more used to develop in PHP for this kind of application and therefore we has to learn to know and use all the different libraries. The choice of MongoDB was also a discovery as we were more used to work with relation databases. 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 the installation of the provided package or by doing a compilation 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 might work under Windows or MacOS as the used libraries should all be cross-platform, but the software has never 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 distributions commands.

4.1.1 Python installation

As the main used language is Python and more specifically the third version of it, 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 launch by the following command. In case of installation problems we ask the reader to refer to the official documentation.

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

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, a packaged version has been built and is ready to be downloaded from the projects GitHub page. The installation process is

really straightforward and 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 installated 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 also available on the GitHub repository and simply launch it from the command line.

4.4 Concluding remarks

To conclude the installation guide, we wanted to point out that only the most 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, make the interface accessible from other machines, deploy it on many popular web-servers. In the next chapter, some experiments are done to measure the performance of the whole system.

Chapter 5

Experiments

In this section different type of experients will be conducted in order to test and check the performance of the developped software. In order to test the following variables the same script was used for all experiments.

5.1 Test script and machine

In order to conduct the different experiments, a test script has been chosen.

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 Data extraction analysis

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

5.2.1 Memory usage

5.2.2 Run-time overheads

5.3 Database performances

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

5.3.1 Some numbers

Speak about db size, memory usage, etc.

5.4 Concluding remarks

Chapter 6

Conclusion

6.1 Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

6.2 Future work

String data –¿ complex data is not well handled String data –¿ remove it from graph listing Graphs –¿ zoom functionality Long data is stored completely Electron app ? caching for accelareted rendering

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Appendix A

Glossary

AOP Aspect Oriented Programming

AST Abstract Syntax Tree

DPA Dynamic Program Analysis

IDE Integrated development environments

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.

PDB The Python Debugger

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

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.

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.

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].