



CERN - EUROPEAN ORGANIZATION FOR
NUCLEAR RESEARCH

SUMMER STUDENT PROJECT

VyPR Analysis Library

MARTA HAN

University of Zagreb, Croatia
CERN, Geneva, Switzerland

supervised by

JOSHUA HENEAGE DAWES

University of Manchester, UK
CERN, Geneva, Switzerland

August 21, 2019

Contents

1	VyPR	2
1.1	Introduction to VyPR	2
1.2	Performance analysis by specification	2
2	VyPR Analysis Library	3
2.1	Object-relational mapping	3
2.2	SQL vs Analysis Library	4
2.3	Verdict severity and state comparison	6
2.4	Path reconstruction functions	6
2.5	Analysis Library Tutorial	7
3	Conclusion	8
	Acknowledgements	9
	References	9

1 VyPR

1.1 Introduction to VyPR

VyPR [1, 2] is a framework for automated performance analysis of Python programs developed at the CMS Experiment at CERN. It works so that a user describes the desired behaviour of a program using a specification. VyPR then monitors the performance of the operations of interest at runtime to check for the given conditions, storing the performance data and verdicts that say whether the given conditions were satisfied in a database. The observed data can then be analysed offline.

The point of the analysis is to find an explanation for violating the specified conditions, which is why not only the observed values of specified properties, but also the different valuations of a variable and the paths taken up to an observation, are being recorded.

1.2 Performance analysis by specification

The condition that a program run should satisfy can be expressed using a logical formula. This formula should specify the property that should be checked and the points in the program at which it should be checked. For example, the following two formulae define a constraint on the duration of a function call, with a slight difference being that the first one checks *every* call, while the other one checks only the calls that happen after the value of the specified variable a changes.

$$\forall t \in \text{calls}(f) : \text{duration}(t) \in [0, 1] \quad (1)$$

$$\forall s \in \text{changes}(a) : \text{duration}(\text{next}(s, \text{calls}(f))) \in [0, 1] \quad (2)$$

However, the language in which the user will write the specification is not exactly what we have seen in the example above. The specifications for VyPR are written in PyCFTL, as shown below. The following specification also shows that, apart from monitoring the duration of the function f , the value that the variable is assigned can also be recorded.

```
verification_conf = {
    "app.routes" : {
        "paths_branching_test" : [
            Forall(
                s = changes('a')
            ).Check(
                lambda s : (
                    s.next_call('f', record=['a']).duration()._in([0, 1])
                )
            )
        ]
    }
}
```

The idea behind having the option to record the values that the variables are being assigned is to be able to determine if there is a significant difference in verdicts generated by different valuations. This is done later, in the offline analysis phase.

To sum up, given a specification written in PyCFTL, VyPR automatically selects the point of interests in the code and monitors the behaviour of a program run at those instrumentation points. The observed data are stored in the database and require further analysis. Extracting the information from the database means querying it, which makes the analysis a bit difficult, as the user needs to be familiar with the database schema. This is where the idea to create an analysis library occurs naturally - in order to access the data, it should be simpler to work with classes and built-in functions in Python that connect to and query the database without the user having to do it manually.

2 VyPR Analysis Library

2.1 Object-relational mapping

Having stored the performance data into the database at runtime, we now have a database that contains tables which describe the program run. This includes the functions of interest, the calls of those functions and the HTTP requests during which they occurred, along with the observed values of the constrained quantities given by the specification and verdicts on those values satisfying or violating the logical formula in the specification. For example, as formula (1) defines a constraint over the duration of all function calls of f , the duration of every call is observed and stored as an observation. A verdict is made based on the observed duration and the interval given by the specification.

Each verdict is made during a single function call and hence refers to a unique function paired with a property. This property, along with the collapsing atom index which is stored as an attribute of the verdict, determines a unique atom which represents the part of the formula that affected the verdict value. The concept of the collapsing atom comes from the fact that some specifications are more complex than those shown by formulae (1) and (2) and have multiple predicates. If that is the case, we want to know which of the predicates caused the violation. In total, all of this performance information adds up to eighteen tables in the database, which is why we decided to create the analysis library.

Through the analysis library, the database is mapped to a virtual one by ORM (object-relational mapping). This means that each table from the database is represented as a class in the library with corresponding columns from the table as attributes and relationships between entities as object functions. The names of each class and its attributes match the names of the corresponding table and its columns in the database. In order to initialise a class, it is required to pass one or more arguments to the initialising function. In all cases, passing just the ID of the object is enough, as this attribute always determines a row of the table in the database uniquely. However, in some of the tables there are other attributes that belong to a unique row in the table, i.e. the name of the function, the time of HTTP request etc. In these cases, it is also possible to initialise the class by passing just one argument.

The initialising function then queries the database to get the corresponding row in the table and creates an object with all attributes' values. The querying process involves communicating with the server side, where the database is stored, using HTTP requests. The requests are delivered via an API written using the Flask library [3]. For example, in order to get a variable of class type `function`, one of the options shown below can be used.

initializing function by ID	initializing function by name
<code>f1=analysis.function(id=1)</code>	<code>f1=analysis.function(fully_qualified_name = "app.routes.paths_branching_test")</code>

Apart from the attributes that match the columns in the tables, most of the classes also have methods that realise the relationships between two or more tables in the database. For instance, if we want to find all the function calls of a certain function, we use the relationship between the two tables to list the required calls. In order to do this by querying the database directly, one would use a query similar to the following one:

```
select * from function_call where function=id;
```

whereas in the analysis library, there is a built-in function which queries the database and is called as

```
f=function(id)
calls_list=f.get_calls();
```

2.2 SQL vs Analysis Library

The main idea behind creating the analysis library is to simplify the querying process. The more complex the relationship between entities, the more obvious it is that having built-in functions that automate the querying is useful. Let's take a look at the method of getting all rows in the table that represent function calls during which a failure occurred. The query to find this in the database is the following:

```
select function_call.id, function_call.function,
       function_call.time_of_call, function_call.http_request
from function_call inner join verdict
on verdict.function_call=function_call.id
inner join function on function_call.function=function.id
where function.id=1 and verdict.verdict=0;
```

And the analysis library's equivalent of this query is:

```
f=function(1)
calls=f.get_calls_with_verdict(0)
```

Another example of this is getting the deserialised structure of the logical formula that represents the above mentioned collapsing atom of a given verdict. To find this by querying means doing the following:

```
select atom.serialised_structure from atom
inner join verdict on atom.index_in_atoms=verdict.collapsing_atom
inner join binding on verdict.binding=binding.id
inner join function on binding.function=function.id
where atom.property_hash=function.property;
```

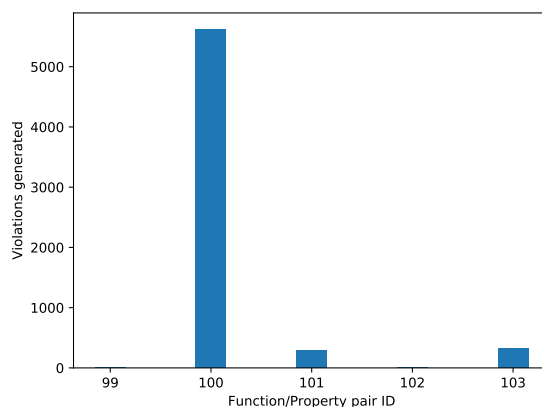
On the other hand, using the analysis library, the formula can be obtained as

```
v=verdicts_list[0] #finds a violation
a=v.get_collapsing_atom()
print(a.get_structure())

#output
d(t = StaticTransition(operates_on=f)) in [0, 1]
```

The analysis library also enables writing simple scripts that generate plots helpful for performance data analysis. Here is the example of such a script.

```
def plot_failed_verdicts_vs_function():
    verdicts=[]
    f_list=analysis.list_functions()
    function_ids=[]
    for f in f_list:
        function_ids.append(f.id)
        verdicts.append(len(f.get_verdicts(0)))
    plt.bar(function_ids,verdicts,width=0.3)
    plt.xticks(function_ids)
    plt.xlabel('Function/Property pair ID')
    plt.ylabel('Violations generated')
    plt.savefig('plot_verdicts.pdf')
    plt.close()
    return
```

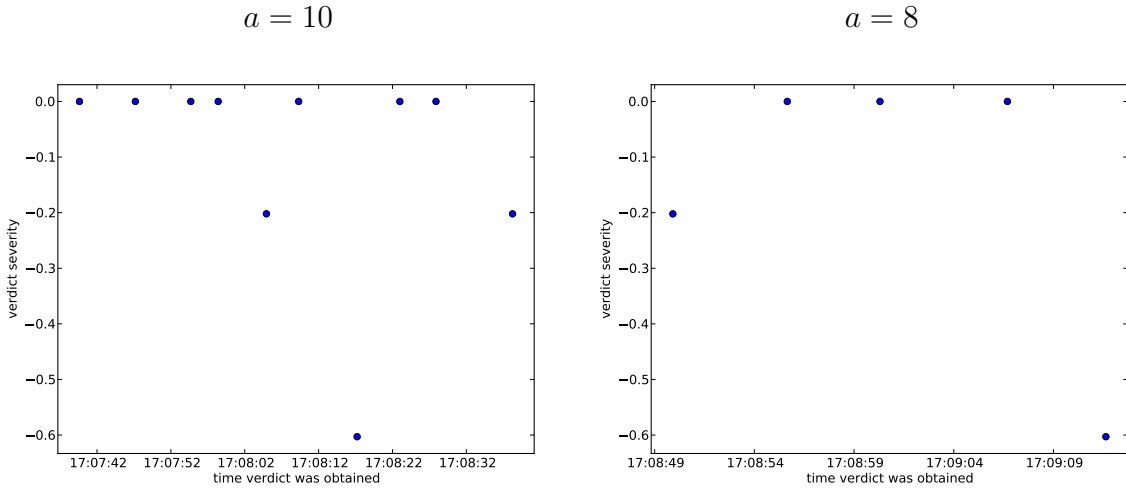


2.3 Verdict severity and state comparison

Beside standard verdict analysis which tells us whether a given condition was satisfied or violated, it might be of interest to know by how much it was violated, or if it does satisfy the constraint, how close it is to violation. The function which measures this is called verdict severity and one possible way to realise it is using the distance between the observed value and the interval boundaries, assigned a negative sign if the value is not in the interval.

$$\text{Sev}(x) = \begin{cases} -\inf_{y \in I} |x - y|, & x \notin I \\ +\inf_{y \notin I} |x - y|, & x \in I \end{cases}$$

An example of how this can be used is finding explanation for differences in verdicts via state comparison. In order to determine if there are values assigned to the variables at an instrumentation point which are more likely to cause a failure, we group the observations by valuations and create a separate plot of verdict severity through time for each 'group'.



We see that these valuations generate both positive and negative severity values. Hence, it is not likely that the assignment values are causing the failure. Another approach to find the explanation behind verdict violations is to consider the paths that a program run took before making an observation.

2.4 Path reconstruction functions

To determine whether the runs that take various paths differ significantly in the final verdicts, we want to be able to reconstruct the path that the run took before the observed function call [4]. Therefore, the analysis library should provide the methods necessary for path reconstruction and comparison.

The functions that are currently available in the library are able to reconstruct the paths taken before one or more observations made at an instrumentation point (if there are more, the point in the code at which they are made must be the same) and find their intersection. Having found the paths, it is also possible to find branching points in the intersection and highlight the lines of code at which the branching occurs.

The following piece of code shows how, given a function name, a list of observation IDs and the ID of the instrumentation point at which the observations are made, the intersection of paths is found and used to highlight the branching point in the code.

```
name='app.routes.paths_branching_test'
path=analysis.get_intersection_from_observations(name,[1,2,3],1)
analysis.edit_code(path,name)
```

As a result, the `edit_code()` function can either display the code inline or create a copy of the file containing the code of interest and highlight the branching there. The resulting output is shown below.

```
def paths_branching_test(n,value_a):
    a = value_a
*   if n > 10:
        l = []
        for i in range(n):
            l.append(i**2)
    else:
        l = []
    f(l)
    return "function for experiments"
```

The specification (2) places a constraint over the duration of the call of f , so the path that the program run takes before that point is recorded. Various runs take different paths and by finding the intersection of those, we also find the part that the paths differ in - this tells us that they followed different branches and comparing that to the verdicts would show if the path that a run takes affects the observed duration of the corresponding function call of f .

2.5 Analysis Library Tutorial

As a way of documenting the analysis library, we opted for a Jupyter [5] notebook format which resulted in creating an interactive tutorial [6] that can be a helpful tool, for both users and new contributors to the project, to become familiar with the functionalities that are available in the VYPR Analysis Library. The tutorial is set up locally by creating a self-contained server-client environment, having cloned VYPR server side and analysis library, along with the VYPR core framework, into an empty directory.

Initialising the database and connecting to it in order to be able to query the database is a somewhat complex process that requires multiple terminals. To simplify the process of cloning the repositories, initialising the database and starting the verdict server in a separate terminal, a shell script that contains the commands which will set up the necessary directories is available. To start up the terminals from the script, we use `tmux` [7] (terminal multiplexer).

Hopefully, apart from being a demonstration of the methods in the analysis library, this tutorial can be helpful as an example of a formal theory implemented as an industrially used software which is invaluable as an educational tool for students learning formal methods and their usage.

3 Conclusion

This summer student project was mainly oriented towards creating the VYPR analysis library, the aim of which is to simplify offline analysis of the performance data stored at runtime. Using the built-in methods that have been added to the library should be more intuitive and effortless for the user than querying the database with a somewhat complex schema. Although the main functionalities are available, the analysis library can always be expanded for the purposes of further analysis. Another thing that has yet to be finished is the automated testing procedure, which is currently under development.

Acknowledgments

Apart from hopefully contributing to the VyPR framework, working on this project was also highly beneficial for me. I have learned a lot and had the privilege of collaborating with the most inspiring, talented and supportive group of people. I would like to express my special thanks to my supervisor Joshua Dawes for his guidance and endless patience - it has been a pleasure working with you. Also, special thanks to team members Andreas Pfeiffer and Giovanni Franzoni for their invaluable suggestions and the effort to create a friendly and welcoming environment. I am also grateful to the entire Summer Student Team who have done an amazing job coordinating the programme that has been an outstanding experience for so many students.

References

- [1] Vypr official website. <http://vypr.web.cern.ch/vypr/>.
- [2] Joshua Heneage Dawes, Giles Reger, Giovanni Franzoni, Andreas Pfeiffer, and Giacomo Govi. Vypr2: A framework for runtime verification of python web services. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 98–114, Cham, 2019. Springer International Publishing.
- [3] Python Flask. <https://flask.palletsprojects.com/en/1.0.x/>.
- [4] Joshua Heneage Dawes and Giles Reger. Explaining Violations of Properties in Control-Flow Temporal Logic.
- [5] Jupyter. <https://jupyter.org/>.
- [6] A Tutorial for VyPRs Analysis Library. <http://jdawes.web.cern.ch/jdawes/students/marta-han-2019/tutorial-talk.pdf>.
- [7] tmux. <https://github.com/tmux/tmux/wiki>.