Metrics

Release 1.2.6

Hugues Wattez, Romain Wallon, Thibault Falque

Jun 06, 2023

CONTENTS

CHAPTER

ONE

METRICS - REPRODUCIBLE SOFTWARE PERFORMANCE ANALYSIS IN PERFECT SIMPLICITY

1.1 Authors

- Thibault Falque - Exakis Nelite
- Romain Wallon - CRIL, Univ Artois & CNRS
- Hugues Wattez - CRIL, Univ Artois & CNRS

1.2 Why Metrics?

Metrics is an open-source Python library and a web-app developed at CRIL by the *WWF Team* (Hugues Wattez, Romain Wallon and Thibault Falque), designed to facilitate the conduction of experiments and their analysis.

The main objective of *Metrics* is to provide a complete toolchain from the execution of software programs to the analysis of their performance. In particular, the development of *Metrics* started with the observation that, in the SAT community, the process of experimenting solver remains mostly the same: everybody collects almost the same statistics about the solver execution. However, there are probably as many scripts as researchers in the domain for retrieving experimental data and drawing figures. There is thus clearly a need for a tool that unifies and makes easier the analysis of solver experiments.

The ambition of Metrics is thus to simplify the retrieval of experimental data from many different inputs (including the solver's output), and provide a nice interface for drawing commonly used plots, computing statistics about the execution of the solver, and effortlessly organizing them. In the end, the main purpose of Metrics is to favor the sharing and reproducibility of experimental results and their analysis.

Towards this direction, *Metrics*' web-app, a.k.a. *Metrics-Studio*, allows to draw common figures, such as cactus plots and scatter plots from CSV or JSON files so as to provide a quick overview of the conducted experiments. From this overview, one can then use locally the *Metrics*' library for a fine-grained control of the drawn figures, for instance through the use of Jupyter notebooks.

CHAPTER

TWO

INSTALLATION

To execute *Metrics* on your computer, you first need to install Python on your computer (at least version 3.8).

As the `metrics` library is available on PyPI, you install it using `pip`.

```
pip install crillab-metrics
```

Note that, depending on your Python installation, you may need to use `pip3` to install it, or to execute `pip` as a module, as follows.

```
python3 -m pip install crillab-metrics
```

CHAPTER THREE

API

3.1 Core

```
.. automodule:: metrics.core.model
    :members:
```

3.2 Builder

3.2.1 Attribute Manager

```
.. automodule:: metrics.core.builder.attribute_manager
    :members:
```

3.2.2 Builder

```
.. automodule:: metrics.core.builder.builder
    :members:
```

3.2.3 Typing Strategy

```
.. automodule:: metrics.core.builder.typing_strategy
    :members:
```

CHAPTER

FOUR

READING A CAMPAIGN INTO *METRICS*

To extract pieces of data from the campaign of experiments you ran and feed it into *Metrics*, you need to use the *Scalpel* module of *Metrics*. *Scalpel* stands for *"extraCt dAta of exPeriments from softwarE Logs"* (*sCAlPEL*).

A campaign is basically read using the following:

```
from metrics.scalpel import read_campaign
my_campaign, my_configuration = read_campaign('path/to/campaign/file', log_level='WARNING
↪')
```

Currently, two types of files can be given as input to *Scalpel*:

- a JSON file containing a serialized form of the campaign (when you have already loaded your campaign in *Metrics*, and saved it for later use), or
- a YAML file describing how to extract data from the campaign you ran.

In the first case, there is almost nothing to do, as the JSON file generated by *Metrics* already contains all the data needed by *Scalpel* (and the returned configuration will thus be `None`). In the second case, the following sections give more details on how to write a configuration file that describes your campaign (the returned configuration will be an object representation of this description).

Additionally, as you can see in the example above, a `log_level` parameter may be specified to the function `read_campaign()`. This allows to configure the minimum level of the parsing events that should be logged. Available levels are, from the lowest to the highest:

- `'TRACE'`
- `'DEBUG'`
- `'INFO'`
- `'WARNING'` (this is the default level)
- `'ERROR'`

Parsing events are regularly logged by *Scalpel* to trace the extraction it performs, mostly for debugging purposes. For instance, activating a lower logging level may allow to identify why some data is missing for a particular experiment. However, it should be noted that *Scalpel* may become particularly verbose when doing so, which may affect performance. This is why this feature should only be activated for debugging purposes.

4.1 Metadata of the Campaign

In the YAML file, you first need to give elementary information about the campaign, such as its name and the date on which it has been run.

```yaml
name: my-awesome-campaign
date: 2020-11-17
```

This information is used to identify your campaign, and is particularly interesting for the traceability of your experiments.

The YAML file must also contain the experimental setup on which the campaign took place, as in the following example:

```yaml
setup:
  os: Linux CentOS 7 (x86_64)
  cpu: Intel XEON X5550 (2.66 GHz, 8 MB cache)
  gpu: Nvidia GeForce 256 SDR
  ram: 32GB
  timeout: 1800
  memout: 1024
```

Note that, for the setup description, only `timeout` and `memout` are required. The other values may be displayed in the reports generated by *Metrics* for reproducibility purposes.

4.2 Description of the Campaign Files

Scalpel is able to parse a wide variety of files that contain the output of the experiments you ran during your campaign. All the information describing the source of your campaign must be given in the `source` section of your YAML configuration file. The main key of this section is `path`, which lists the file(s) containing the data to extract.

```yaml
source:
  path:
    - path/to/first/file
    - path/to/second/file
```

This key declares the list of the files (either regular files or directories, depending on the format of your campaign) that *Scalpel* will parse. Note that all files must have the same format.

All these files will be parsed sequentially, and their content will be merged into a single campaign. If these files represent distinct parts of your campaign (e.g., each file contains the result of a different experiment-ware), you may be interested in the extraction of metadata from the name of the file, described *here*.

If you only have one file containing all the results of your campaign, you may avoid the use of a list, and simply write the path of the file as the value for `path`:

```yaml
source:
  path: path/to/single/file
```

In the following subsections, we present what you must add to the `source` field to configure *Scalpel* for parsing your campaign, depending on its format.

4.2.1 Parsing a CSV File

The CSV (Comma-Separated Values) format is often used to store experimental data. It is mainly a tabular format, which has an (optional) *header* line giving the titles of the column. Each of the remaining lines corresponds to the data collected during an experiment.

Depending on the variant, columns may be separated by:

- a comma (`,`), giving the default `csv` format,
- a semicolon (`;`), giving the `csv2` format, or
- a tabulation (`\t`), giving the `table` format.

To specify that your campaign is in one of these formats, you need to add the following to your YAML configuration file:

```yaml
source:
  path: path/to/my/file.csv
  format: csv
```

Actually, the `format` may be omitted in this example, as the extension of the file already tells *Scalpel* that the file is in the (classical) `csv` format. Similarly, if you specify as `path` the files `path/to/my/file.csv2` or `path/to/my/file.table`, you may omit the format, as *Scalpel* will infer that such files use the `csv2` and `table` formats, respectively.

You may also have more "exotic" CSV-like files, which do not use a standard separator or quote character (by default, `"` is used as quote character). If this is the case, you may describe them by adding the following keys in the source section:

```yaml
source:
  quote-char: "%"
  separator: "|"
```

In the example above, the quote character is `%` and the columns are separated by the character `|`.

Finally, you may have a header for your CSV file, or not. By default, the first line is considered as a header line, and is used to identify the values parsed in the other lines as experimental data. If you do not have a header line, add the following key:

```yaml
source:
  has-header: false
```

In this case, values will be identified by the index of the corresponding column, as a string (starting from `"0"`). Note that, in this case, *Scalpel*'s naming convention cannot be followed. As such, do not forget to specify the mapping of the columns in the text file to fit *Scalpel*'s needs (see *below* for more details). You must also do so as long as the name of the columns in your CSV files do not fit *Scalpel*'s expectations.

4.2.2 Parsing an "Evaluation" File

If you are interested in analyzing the results of a campaign run with the so-called "Evaluation" platform (such as, for instance, the results of the XCSP'19 competition, we provide a parser to read the "results of individual jobs as text file" provided by this platform (as the one of the XCSP'19 competition, available here).

To do so, specify the following in your YAML configuration file:

```yaml
source:
  path: path/to/result/file.txt
  format: evaluation
```

As this platform does not use in general the same naming convention as that of *Scalpel*, do not forget to specify the mapping of the columns in the text file to fit *Scalpel*'s needs (see *below* for more details).

4.2.3 Parsing a "Reverse" CSV File

We call a CSV file "reverse" when each line in this file corresponds to an input, and the columns to the different statistics collected for the experiment-wares run during the campaign. Here is an example of such a file:

```
xp-ware-a,xp-ware-b,xp-ware-c
0.01,0.02,0.03
```

In this example, we consider a campaign that run three experiment-wares, namely `xp-ware-a`, `xp-ware-b` and `xp-ware-c`. Each column is by default interpreted as the CPU time of the corresponding experiment, as this is the only statistic required for an experiment. Also, note that no input is specified in this example. This is tolerated, as each line in such a format maps to exactly one input. However, we strongly recommend specifying the name of the input file, especially because it makes easier the interpretation of the experimental results and their reproducibility.

A more complete example of a "reverse" CSV file is given below:

```
input,xp-ware-a.cpu_time,xp-ware-a.memory,xp-ware-b.cpu_time,xp-ware-b.memory,xp-ware-c.
↪cpu_time,xp-ware-c.memory
input-a,0.01,10,0.02,20,0.03,30
```

Here, we collect more statistics, as we consider both the `cpu_time` and `memory` needed for an experiment. Those names are used to identify the corresponding statistics in the representation of the experiment. In the example above, the experiment-ware and the statistics identifiers are separated with a dot (`.`), which is the default. If you want to specify a different separator, you can specify it in the YAML configuration as follows (make sure not to use the same separator as for the columns):

```
source:
  title-separator: "!"
```

To configure how a reverse CSV file is parsed, you can also use the same properties as those used in classical CSV file (see the previous section), and specify one of the formats `reverse-csv`, `reverse-csv2` or `reverse-table` (using the same naming convention as before).

4.2.4 Parsing Raw Data from a File Hierarchy

If you have gathered the output of your experiment-wares in a directory, *Scalpel* can explore the file hierarchy rooted at this directory and extract all relevant data for you. We support three different kinds of file hierarchies, which are described below.

Note that, by default, *Scalpel* does **not** follow symlinks when exploring a file hierarchy. For each of the configurations below, you may alter this behavior by adding the following to the `source` section of the YAML file:

```
source:
  follow-symlinks: true
```

One File per Experiment

In this case, the file hierarchy being explored is supposed to contain exactly one (regular) file per experiment. You can configure *Scalpel* to consider such a file hierarchy using the following description:

```
source:
  path: /path/to/my-experiment-directory
  format: one-file
```

Let us consider an example to illustrate how *Scalpel* extracts data based on this configuration. Suppose that the file hierarchy to explore has the following form:

```
my-experiment-directory
    + experiment-a.log
    + experiment-b.log
    ` more-experiments
        + experiment-c.log
        ` experiment-d.log
```

Here, *Scalpel* will recursively explore the whole file hierarchy, and will parse all regular files, provided that these files are specified in the `data` section of the YAML configuration file (see the dedicated documentation *here* for more details). Each file `experiment-a.log`, `experiment-b.log`, `experiment-c.log` and `experiment-d.log` will be considered as the output of a single experiment.

Note that these files may have common formats (such as JSON, XML or CSV) or may also be the raw output of the solver. More details on how to retrieve relevant information from these files are given *here*.

Multiple Files per Experiment

In this case, the file hierarchy being explored is supposed to contain a set of (regular) files per experiment. The name of the files (without their extensions) will be used to identify each experiment. You can configure *Scalpel* to consider such a file hierarchy using the following description:

```
source:
  path: /path/to/my-experiment-directory
  format: multi-files
```

Let us consider an example to illustrate how *Scalpel* extracts data based on this configuration. Suppose that the file hierarchy to explore has the following form:

```
my-experiment-directory
    + experiment-a.out
    + experiment-a.err
    + experiment-b.out
    + experiment-b.err
    ` more-experiments
        + experiment-c.out
        + experiment-c.err
        + experiment-d.out
        ` experiment-d.err
```

Here, *Scalpel* will recursively explore the whole file hierarchy, and will parse all regular files, provided that these files are specified in the `data` section of the YAML configuration file (see the dedicated documentation *here* for more details). In this case, the files `experiment-a.out` and `experiment-a.err`, for instance, will be considered as outputs of the same experiment (they are both named `experiment-a`).

Note that these files may have common formats (such as JSON, XML or CSV) or may also be the raw output of the solver. More details on how to retrieve relevant information from these files are given *here*.

One Directory per Experiment

In this case, the file hierarchy being explored is supposed to have one directory that contain the output files of each experiment. The name of the files inside this directory may be arbitrary (and even the same from one experiment to another). You can configure *Scalpel* to consider such a file hierarchy using the following description:

```yaml
source:
  path: /path/to/my-experiment-directory
  format: dir
```

Let us consider an example to illustrate how *Scalpel* extracts data based on this configuration. Suppose that the file hierarchy to explore has the following form:

```
my-experiment-directory
    + experiment-a
    |   + stdout
    |   + stderr
    + experiment-b
    |   + stdout
    |   + stderr
    ` more-experiments
        + experiment-c
        |   + stdout
        |   + stderr
        ` experiment-d
            + stdout
            + stderr
```

Here, *Scalpel* will recursively explore the whole file hierarchy, and will consider each directory containing regular files as an experiment. All the regular files contained in this directory will thus be considered as outputs of the corresponding experiments, as long as these files are specified in the `data` section of the YAML configuration file (see the dedicated documentation *here* for more details). For instance, the `stdout` and `stderr` files in the directory `experiment-a` will be considered as output files of the experiment `experiment-a`, and will thus be used together to extract relevant information for this experiment.

Note that these files may have common formats (such as JSON, XML or CSV) or may also be the raw output of the solver. More details on how to retrieve relevant information from these files are given *here*.

4.2.5 Parsing Unsupported Formats

When developing *Scalpel*, we tried to think about as many campaign formats as possible. However, it may happen that you need to parse a campaign that uses a format that is not recognized (yet) by *Scalpel*. If this is the case you may write your own parser. by extending the class `CampaignParser`. This class must define a constructor taking as argument a `CampaignParserListener` and a `ScalpelConfiguration`. To give you ideas on how to write such a parser, you may have a look to the source of our parsers.

Then, add the class of your parser to your YAML configuration file as follows:

```yaml
source:
  parser: my.completely.specified.AwesomeParser
```

Scalpel will dynamically instantiate your parser, and will then use it to parse the campaign. To make this possible, you will need to import your `AwesomeParser` before invoking `read_campaign()`, to make sure that this class will be reachable.

Remark

If you need to parse a campaign that uses an unsupported format, do not hesitate to submit an issue, with an example of what you want to parse. We will provide you some advices for writing your own parser.

We may also add a new feature to *Scalpel* by supporting this format, either by writing a parser or by integrating yours if you agree to contribute and submit a pull request.

4.2.6 Identifying Successful Experiments

When analyzing experimental results, it is often useful to identify which experiments are successful and which are not. By default, an experiment is considered as successful when it ended within the time limit. However, you may also want to perform additional checks to make sure that an experiment succeeded (for instance, by checking that the output of the experiment is correct).

To do so, you may add to your YAML configuration file an `is-success` filter that allows to make such checks, as in the following example:

```yaml
source:
  is-success:
    - ${success}
    - ${valueA} == ${valueB} or {valueC} == 0
    - ${result} in ['CORRECT', 'CORRECT TOO']
```

Let us describe the syntax of the filter in the example above. First, `is-success` defines a list of conjunctively interpreted Boolean expressions. These expressions are themselves disjunctions of predicates.

Each predicate has to contain at least one variable, delimited using `${...}`. Such a variable corresponds to the identifier of an experimental data read for a given experiment (for instance, the `cpu_time` of the experiment).

If the predicate contains only the variable (such as `${success}`), then this variable is interpreted as a Boolean value. Otherwise, the predicate can use any comparison operator (among <, <=, ==, `!=`, >=, >) to compare the variable with either a literal value (which can be a Boolean value, an integer, a float number or a string) or another variable. A predicate can also check that a variable is either contained in a list of values (either literal values or variables) or contains a value (either a literal value or a variable) using the `in` operator. Lists are delimited using `[...]`.

Remark

It is worth noting that *Scalpel* itself does **not** use `is-success` to filter data, in the sense that even failed experiments are included in the campaign it builds.

Instead, *Scalpel* passes this filter on to *Wallet*, so that the drawn figures only take into account successful experiments.

4.3 Description of the Data to Extract

In order to extract data from the files of your campaign, you need to provide a description of their content. In the following, we describe how to write such a description.

4.3.1 Extracting Data from Raw Files

If your experiment-ware produces raw output, and you want *Scalpel* to parse it, you can describe how to extract data from the corresponding files (which can be given using wildcards or relative paths) by providing regular expressions, as in the following example:

```
data:
  raw-data:
    - log-data: cpu_time
      file: "*.out"
      regex: 'overall runtime: (\d+.\d+) seconds'
      group: 1
```

In this case, when *Scalpel* reads a file with extension `.out`, it looks for a line that matches the specified regular expression, and extracts the `cpu_time` of the experiment from the group `1` (i.e. `(\d+.\d+)`) in this regular expression. In this case, the group could be omitted, as the value `1` is the default.

To make easier the description of raw data, *Scalpel* also recognizes so-called *simplified patterns*, as illustrated in the following example:

```
data:
  raw-data:
    - log-data: cpu_time
      file: "*.out"
      pattern: "overall runtime: {real} seconds"
```

Observe that, here, `pattern` is used in place of `regex`, and that the group `(\d+.\d+)` used in the previous example is replaced by `{real}`. This syntax allows to use one of the different symbols used to represent common data, and to avoid worrying about whitespaces (in a simplified pattern, any whitespace is interpreted as a sequence of whitespace characters).

Scalpel can interpret the following symbols.

- `{integer}` for a (possibly signed) integer,
- `{real}` for a real number,
- `{boolean}` for a Boolean value (`true` or `false`, case-insensitive),
- `{word}` for a word (i.e., a sequence of letters, digits and _), and
- `{any}` for any sequence of characters (not greedy).

If the same line contains multiple relevant data, you can extract them by giving names to the groups you specified (in this case, the value for `log-data` may be omitted).

```
data:
  raw-data:
    - file: "*.out"
      pattern: "runtime: {real} seconds (cpu), {real} seconds (wallclock)"
      groups:
```

(continues on next page)

(continued from previous page)

```
        cpu_time: 1
        wall_time: 2
```

Note that it is not possible to mix regular expressions and simplified patterns.

4.3.2 Extracting Data from File Names

Depending on your setting, you may need to extract relevant information from the name of the file to parse (for instance, the name of the experiment-ware or that of the input). This can be achieved through `file-name-meta`, as in the following example:

```
data:
  file-name-meta:
    pattern: "{any}_{any}.log"
    groups:
      experiment_ware: 1
      input: 2
```

As for `log-data`, you may choose to use either regular expressions (`regex`) or a simplified `pattern`. The fields in `groups` are used to name the groups identifying relevant data.

For instance, if the file `my-xp-ware_my-input.log`, the group 1 matches with `my-xp-ware`, which is thus identified as the `experiment_ware`, while the group 2 matches with `my-input`, which is thus identified as the `input`.

As file hierarchies are explored through the file system, the paths of the files that are encountered during this exploration are system-dependent (in particular, the file separator may vary from one system to another). *Scalpel* is able to dynamically adapt file separators used in the pattern specified as `file-name-meta` to ensure cross-platform compatibility for your configuration. To make sure that this compatibility is applied, you must always use `/` as file separator (even if it is not that of your system).

4.3.3 Extracting Data from Common Formats

If your output files use a common format (as JSON, CSV or XML), you do not need to use `raw-data` to extract their value. Instead, you just need to specify the name of such files as follows (wildcards and relative paths are supported).

```
data:
  data-files:
    - "*.json"
    - "output.xml"
```

Note that *Scalpel* will be able to extract data from such files by inferring automatically identifiers for the data it extracts. In the case of CSV files, the identifiers that will be used is inferred based on the header of the file.

For JSON and XML files, a “dotted” notation will be used. For example, consider the following JSON output:

```
{
  "experiment": {
    "runtime": 123.4,
    "value": [24, 27, 42, 51, 1664]
  }
}
```

Scalpel will automatically identify the runtime as `experiment.runtime` and the list of values as `experiment.value`. The same identifiers are inferred for the following XML output:

```xml
<experiment runtime="123.4">
  <value>24</value>
  <value>27</value>
  <value>42</value>
  <value>51</value>
  <value>1664</value>
</experiment>
```

By default, all keys stored in a JSON or XML file are extracted by *Scalpel*, and stored in the internal representation of the campaign. This may be memory consuming, in particular if there are some keys that you do not need. To discard such keys, you may specify them in the field `ignored-data` in your YAML configuration (this may actually be applied to any key defined by the campaign). For instance, the snippet below allows to discard the list `experiment.value` described in the two examples above.

```yaml
data:
  ignored-data:
    - experiment.value
```

If needed, you can also configure the parser to use for reading data from data-files, as in the following example:

```yaml
data:
  data-files:
    - name: "*.json"
      format: json
      name-as-prefix: true
    - name: "*.csv"
      format: csv
      has-header: false
      separator: " "
      name-as-prefix: true
    - name: "*.txt"
      parser: my.completely.specified.AwesomeParser
```

Observe in the example above that CSV files may be configured as for CSV campaigns (the same fields are used to describe the format of the file).

For each data-file, you can also set `name-as-prefix` to `true`, so that each field in the file will be prefixed by the name of the file, using a dotted notation. This is particularly useful whe the same key appears in different files.

Moreover, it is also possible to specify a custom parser, provided you give the *completely specified* name of this class. This parser must extend `CampaignOutputParser`, and its constructor must take as input a `CampaignParserListener`, a `ScalpelConfiguration`, the path of the file to parse and its name.

Finally, you may face some cases where the wildcards you use for declaring data files (or even log-data) are too generic. To ignore some files that still match these wildcards, you may specify the files to ignore with the field `ignored-files` (wildcards and relative paths are supported). For example, the snippet below allows to ignore *some* JSON files.

```yaml
data:
  ignored-files:
    - "*ignore*.json"
```

4.3.4 Mapping Data to *Scalpel*'s Expectations

When parsing an experiment, *Scalpel* expects to find the required information to describe the result of this experiment. The identifier of such data is thus crucial to allow *Scalpel* to build consistent experiments. This is in particular true for the identifiers:

- `experiment_ware`, which is the experiment-ware run in a given experiment,
- `input`, which is the input on which the experiment-ware has been run, and
- `cpu_time`, which is the runtime of the experiment.

If these identifiers are not specified in your campaign files (for instance, you have a CSV file in which the header does not use these names), you need to tell *Scalpel* how to map your experimental data to the expected identifiers. This can be achieved by specifying a `mapping` as in the following example:

```yaml
data:
  mapping:
    experiment_ware:
      - program
      - options
    cpu_time: runtime
    input: file
    file: path
```

In this example, we have that, for each experiment, the data read as `runtime` will be interpreted as `cpu_time` and `file` as `input`.

Note that, for `experiment_ware`, two identifiers are specified. In this case, the data read as `program` and `options` will be concatenated (in this order) to build up the identifier of the experiment-ware. Moreover, if this experiment-ware does not exist yet, an object representation of this experiment-ware will be instantiated, using `program` and `options` has two additional fields.

Finally, observe that `path` is mapped to `file`, which is itself mapped to `input`. In this case, a recursive mapping is actually applied on `path`, which will be eventually interpreted as `input` while parsing the campaign. Recursive mapping is the recommended approach for mapping several identifiers to the same key.

Remark

This mapping is mainly designed to map custom identifiers to *Metrics*' naming conventions. However, you can also use this mapping to rename other data (especially when their identifiers are automatically inferred by *Scalpel*), or to group together data that are separated in your campaign files.

4.3.5 Adding Default Values

Sometimes, it may happen that some data are missing in your experiment files, either because some experiment-wares did not output them correctly, or did not have enough time to output them within the time limit. This may be a problem if this data is required by *Scalpel*. For such data, you may provide default values as follows:

```yaml
data:
  default-values:
    cpu_time: 1800
```

In this example, we have set the default `cpu_time` to the same value as the time limit (note that this is done by default by *Scalpel*). You may set default values for any key of the campaign, and even for "partial keys" (i.e., those that are part of a mapping).

4.4 Additional Information About the Campaign

When collecting data about a campaign, you may want to add relevant information that do not appear in the files produced during the execution of your experiments regarding its settings. This section presents how you can describe the experiment-wares and inputs you used for your experiments.

4.4.1 Description of the Experiment-Wares

Optionally, you may provide a description of the experiment-wares (i.e., the software programs you ran during your campaign). By default, experiment-wares are automagically instantiated when encountered during the parsing of your campaign files.

However, you may want to specify additional data w.r.t. the programs you experimented (for instance, the version of the software, the command line options passed to the program that was executed, etc.).

As such data may not appear in your campaign files, you can specify them in the YAML configuration as follows:

```yaml
experiment-wares:
  - name: my-awesome-xpware
    version: 0.1.0
    command-line: ./my-awesome-xpware -o option
  - name: my-great-xpware
    commit-sha: abcd1234
    command-line: ./my-great-xpware -v value
```

When you specify information about experiment-wares, only their names are required. The name of an experiment-ware must uniquely identify this experiment-ware in the campaign, and must match the one that *Scalpel* will extract from your campaign files. For all other information you specify, you may use any key you want to identify this information.

Also, note that you are not required to use the same keys for all experiment-wares. You may also omit experiment-wares for which you do not need more information than those mentioned in the campaign files: these experiment-wares will simply be discovered when parsing the files.

Moreover, you may simply specify the list of the experiment-wares used in the campaign:

```yaml
experiment-wares:
  - my-awesome-xpware
  - my-great-xpware
```

Doing so is rarely useful, as the name of the experiment-wares must necessarily be mentioned in the campaign files, and thus will be discovered during their parsing. However, this may be helpful to remind you that some experiments are missing, for instance, if you do not have run all experiment-wares yet.

4.4.2 Description of the Inputs

As for experiment-wares, you may want to add data about the inputs of your experiments. This is achieved by defining an `input-set` in your YAML configuration file, and giving it a proper name, as in the following example:

```yaml
input-set:
  name: my-awesome-input-set
  type: list
  files:
    - path: path/to/instanceA.cnf
```

(continues on next page)

(continued from previous page)

```yaml
      family: F1
    - path: path/to/instanceB.cnf
      family: F2
```

In this example, `files` allows to list all the inputs you used in your experiments. As for experiment-wares, you may specify as many data as you want for your inputs. You may also use different keys for these data, and omit input files when you do not need to add more information than that provided in the campaign files. The only required key is `path`.

In the example above, observe that the type `list` is declared, to specify that all relevant information are specified in `files`.

Another possible type is `file-list`, if you only list the path of the files (in which case, you do not need to specify the `path` key). You may also use `file` if this list is written in a separate file (one path per line), in which case the `files` must give the list of the files to read.

Finally, you may also specify a `hierarchy` type, in which case *Scalpel* will explore a file hierarchy to find all input files from the file hierarchy rooted at the directory specified in `files`, as in the following example:

```yaml
input-set:
  name: my-awesome-input-set
  type: hierarchy
  extensions: ".cnf"
  files: /path/to/my/benchmarks
  file-name-meta:
    pattern: /path/to/my/benchmarks/{any}/{any}.cnf
    groups:
      family: 1
      name: 2
```

Note that, in this example, the input files that are considered are those stored in the file hierarchy rooted at the directory `/path/to/my/benchmarks`, and having a `.cnf` extension (you may also specify a list of extensions if you have more than one).

Also, observe that a `file-name-meta` section is specified, with the same syntax as that described *here*. It allows extracting relevant information from the name of each input.

Both `extensions` and `file-name-meta` are taken into account for any type of input-sets.

CHAPTER

FIVE

ANALYZE A CAMPAIGN IN *METRICS*

Once the YAML file is correctly configured (*Reading a Campaign into* Metrics), the analysis of data can started. To analyze the campaign of experiments thanks to *Metrics*, you need to use the *Wallet* module of *Metrics*. *Wallet* stands for *"Automated tooL for expLoiting Experimental resulTs"* (*wALLET*).

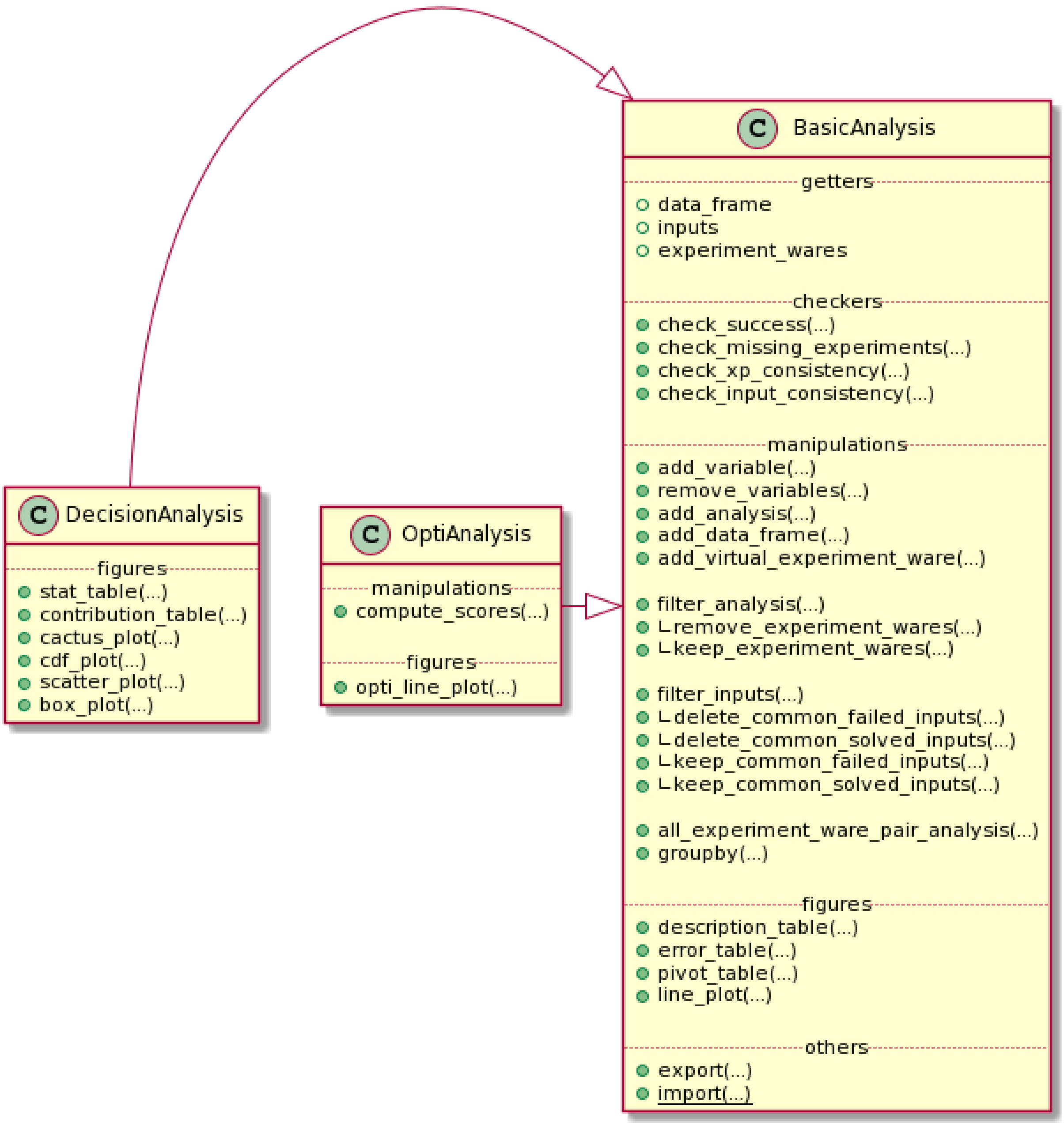
To manipulate data, *Wallet* uses a *pandas Dataframe*. A dataframe is a table composed of rows corresponding to experimentations (also denoted as observations) and columns corresponding to the variables/metrics of an experimentation.

It is not necessary to have wide knowledge about this library to manipulate *Wallet* data but in order to have a better idea on how data are manipulated, an example of a classical analysis dataframe is given:

	input	experiment_ware	cpu_time
0	XCSP17/AllInterval/AllInterval-m1-s1/AllInterval-035.xml	BTD 19.07.01	0.031203
1	XCSP17/AllInterval/AllInterval-m1-s1/AllInterval-035.xml	choco-solver 2019-09-16	1.51053
2	XCSP17/AllInterval/AllInterval-m1-s1/AllInterval-035.xml	choco-solver 2019-09-20	1.52427
3	XCSP17/AllInterval/AllInterval-m1-s1/AllInterval-035.xml	choco-solver 2019-06-14	1.60424
4	XCSP17/AllInterval/AllInterval-m1-s1/AllInterval-035.xml	AbsCon 2019-07-23	3.65329

For the next, the documentation focuses on the analysis of a CSP solver competition (XCSP'19).

5.1 A Preview of What is Able to Do an `Analysis`



Globally, an *Analysis object is composed of five parts:

- `getters` to get basical objects from the analysis
- `checkers` that permit to check many important information about the analysis
- `manipulations` that permit to manipulate the state of the analysis
- `figures` that permit to draw some tables and plots representing the data
- `others` that correspond to operations like exporting.

Here, an analysis is divided in three different objects:

- `BasicAnalysis` is an analysis with the only constraint of a complete cartesian product between inputs and experiment-wares
- `DecisionAnalysis` is an analysis taking into account the time and the success, or not, of each experiment
- `OptiAnalysis` is an analysis of an optimality problem taking into account the list of bound and timestamps, and the success, or not, of each experiment

A `BasicAnalysis` could be used openly without the constraint of success, time, or bound list information.

In this tutorial, we firstly focus on a `DecisionAnalysis` with the inherited methods from a `BasicAnalysis`.

A final part focuses on the optimality analysis of the object `OptiAnalysis`.

5.2 Create/Import/Export a DecisionAnalysis

5.2.1 The DecisionAnalysis Object

To create a new analysis, you only need to import the `DecisionAnalysis` class from *Wallet* module and instantiate a new `DecisionAnalysis` object with the path to the YAML configuration file:

```
from metrics.wallet import DecisionAnalysis

analysis = DecisionAnalysis(input_file='path/to/xcsp19/YAML/file')
```

In the constructor above, it is possible to specify a `log_level` that will be passed to *Scalpel* to log parsing events.

The analysis is composed of many variables describing the experiments:

- necessary ones: `input`, `experiment_ware`, `cpu_time`, `timeout`
- optional ones (given by the current competition file): `Category`, `Checked answer`, `Objective function`, `Wallclock time`, `Memory`, `Solver name`, `Solver version`.

These variables permit to check the consistency and the validity of information. Some methods, called checkers, permit to operate some basic operations:

- `<analysis>.check_success(<lambda>)`: given a lambda, this method permits to check if an experiment is a success or not (this method is automatically executed when the user has informed it in the Scalpel file);
- `<analysis>.check_missing_experiments()`: this method is automatically called by the `*Analysis` constructor to replace missing experiments by unsuccessful experiments;
- `<analysis>.check_xp_consistency(<lambda>)`: given a lambda, this method permits to check the consistency for each experiment;
- `<analysis>.check_input_consistency(<lambda>)`: given a lambda, this method permits to check the consistency for each input (composed of many experiments); it asks some basic knowledge on DataFrame manipulation (an example is given by the next).

`check_success` and `check_missing_experiments` are automatically called during the `*Analysis` constructor call. After, the user could (re-)check these success and consistency methods as follow:

```
inconsistent_returns = {
    'ERR WRONGCERT', 'ERR UNSAT'
}

```

(continues on next page)

(continued from previous page)

```
successful_returns = {'SAT', 'UNSAT'}

is_consistent_by_xp = (lambda x: not x['Checked answer'] in inconsistent_returns)
is_consistent_by_input = (lambda df: len(set(df['Checked answer'].unique()) & successful_
↪returns) < 2)
is_success = (lambda x: x['Checked answer'] in successful_returns)

analysis.check_success(is_success)
analysis.check_input_consistency(is_consistent_by_input)
analysis.check_xp_consistency(is_consistent_by_xp)
```

The `*Analysis` construction warns the user when inconsistencies are found, missing data, . . . :

```
1 experiment is missing and has been added as unsuccessful.
4 experiments are inconsistent and are declared as unsuccessful.
1 input is inconsistent and linked experiments are now declared as unsuccessful.
```

The analysis creates also its own variables corresponding to the previous checkings: `error`, `success`, `missing` `consistent_xp` and `consistent_input`.

It exists another way to build an analysis that is presented in the `Advanced Usage` section.

5.2.2 Export and Import an Analysis

At any moment, the analysis could be exported to save its state into a file:

```
analysis.export('analysis.csv')
```

An analysis could be exported as a csv (as a `DataFrame` representation) if the `.csv` extension is used, else the analysis is exported as a binary object.

To import an analysis from a file, the function `import_analysis_from_file` may be used:

```
imported_analysis = DecisionAnalysis.import_from_file(filepath)
```

> You can observe an example of these functions in this notebook.

5.3 Manipulate the Data from *Analysis

Before producing the first figures, *Wallet* proposes to manipulate the different experiments composing the dataframe. It allows to analyze more finely the campaign.

5.3.1 Generate a New Information/Variable for Each Experiment

Wallet can add new information to the underlying dataframe by giving a function/lambda to a mapping method of `BasicAnalysis`. For the next example, the input name corresponds to the path of the input (e.g., `/XCSPxx/family/.../input-parameters.xcsp`). It could be interesting to extract the family name to use it in the rest of the analysis. For this, the method `add_variable()` from `BasicAnalysis`:

```
import re
family_re = re.compile(r'^XCSP\d\d/(.*?)/')

new_analysis = analysis.add_variable(
    new_var='family',
    function=lambda x: family_re.match(x['input']).group(1)
)
```

`add_variable()` takes as first parameter the name of the future created column, and as second parameter the lambda that applies the regular expression `family_re` to the variable `input` of the row `x` (the regular expression returns an object corresponding to the matching strings: `.group(1)` permits to retrieve the family name of the input).

The result (as a sample of 5 experiments with the only 2 interesting columns shown) is:

	input	family
3641	XCSP17/Primes/Primes-m1-p25/Primes-25-80-2-7.xml	Primes
2992	XCSP17/MaxCSP/MaxCSP-maxclique-s1/MaxCSP-brock-800-2.xml	MaxCSP
2956	XCSP17/MagicSquare/MagicSquare-sum-s1/MagicSquare-13-sum.xml	MagicSquare
7106	XCSP18/GracefulGraph/GracefulGraph-K05-P02_c18.xml	GracefulGraph
4423	XCSP17/QRandom/QRandom-mdd-7-25-5/mdd-7-25-5-56-09.xml	QRandom

Thanks to this method, the user is also able to update existing columns (e.g., renaming the experiment-wares to simplify their names).

You can observe an example of this command in this notebook.

5.3.2 Remove Variables from the Analysis

Sometimes, some analysis information are not useful: it could be interesting to simplify and lighten the dataframe (e.g., when we need to export the analysis in a lighter format). To do this:

```
analysis.remove_variables(
    vars=['Category', 'Objective function']
)
```

where `vars` parameter take the list of variables to remove.

5.3.3 Add an Analysis or a DataFrame to the current Analysis

When many campaigns needs to be compared and two analysis `a1` and `a2` have been created, it is possible de merge them:

```
a3 = a1.add_analysis(a2)
```

The user has to be careful to merge consistent data: the new analysis needs to contain the Cartesian product of the available inputs in its dataframe with the experiment-wares. To ensure this and the consistency of its analysis, the user can also apply the lambda as described for the Analysis construction.

In the same way, it is possible to append the analysis with a consistent dataframe:

```
a3 = a1.add_data_frame(a2.data_frame)
```

5.3.4 Add a Virtual Experiment-Ware

Sometimes, it may be interesting to introduce what we call a *Virtual Experiment-Ware* (VEW), which generalizes the well-known *Virtual Best Solver* (VBS). It allows to compare our current experiment-wares to the virtual (best) one. A VBEW (*Virtual Best Experiment-Ware*) selects the best experiment for each input from a selection of real experiment-ware thanks to the function `find_best_cpu_time_input`:

```
from metrics.wallet import find_best_cpu_time_input

analysis_plus_vbs = analysis.add_virtual_experiment_ware(
    function=find_best_cpu_time_input,
    xp_ware_set=None, # None corresponds to the all available experiment-wares of the␣
↪analysis
    name='my_best_solver'
)
```

Here, we create a VBEW named `my_best_solver` and based on the best performances of the overall set of experiment-wares. `my_best_solver` will receive the result of one of these experiment-wares minimizing the `cpu_time` column.

`find_best_cpu_time_input` is a function using some basic knownledge about dataframe. As an example, `find_best_cpu_time_input` representation is shown:

```
def find_best_cpu_time_input(df):
    s = df['cpu_time']
    return df[s == s.min()]
```

`find_best_cpu_time_input` receives a dataframe composed of the experiments composing a given input. It finds the minimal `cpu_time` value and returns the row corresponding to this best time.

You can observe an example of this method in this notebook.

5.3.5 Subset of *Analysis Rows

*Analysis is also able to make a subset of its experiments.

By Filtering Inputs

By default, it exists some useful subset methods in *Analysis object to filter inputs (and linked experiments):

- keep_common_failed_inputs(): returns a new *Analysis with only the common failed experiments. It corresponds to inputs for which no experiment-ware has succeeded;
- keep_common_solved_inputs(): returns a new *Analysis with only the common successful experiments. It corresponds to inputs for which no experiment-ware has failed;
- delete_common_failed_inputs(): returns a new *Analysis where commonly failed inputs are removed;
- delete_common_solved_inputs(): returns a new *Analysis where commonly succeeded inputs are removed.

Finally, we present a last and generic method to make a subset of inputs:

```
analysis.filter_inputs(
    function=<lambda>,
    how=<"all"|"any">
)
```

The filter_inputs method takes two parameters:

- function corresponds to a True/False lambda that says if an experiment (from input experiments) is acceptable or not
- how corresponds to the need to have at least one or all the experiments from input acceptables.

As examples, we show how the four default methods are set with this generic one:

Default method	Implementation
delete_common_failed_inputs	analysis.filter_inputs(function=lambda x: x['success'], how='any')
delete_common_solved_inputs	analysis.filter_inputs(function=lambda x: not x['success'], how='any')
keep_common_failed_inputs	analysis.filter_inputs(function=lambda x: not x['success'], how='all')
keep_common_solved_inputs	analysis.filter_inputs(function=lambda x: x['success'], how='all')

You can observe an example of this subset in this notebook.

By Filtering Experiments

Analysis permits also to precise what are the experiments that the user wants to filter:

```
analysis_no_para = analysis.filter_analysis(
    function=lambda x: 'parallel' not in x['experiment_ware']
)
```

The previous example permits to remove all the solvers containing the term *parallel* in its title.

Derived from this previous generic method, some default actions are also existing:

Default method	Implementation
`remove_experiment_wares(<set>)`	`analysis.filter_analysis(lambda x: x[EXPERIMENT_XP_WARE] not in experiment_wares)`
`keep_experiment_wares(<set>)`	`analysis.filter_analysis(lambda x: x[EXPERIMENT_XP_WARE] in experiment_wares)`

5.3.6 Grouping the Analysis

To group the analysis into specific analysis, two more methods are presented: the classical `groupby` method and another one to group experiment-wares by pairs.

`groupby` Operator

The `groupby` operator allows to create a list of new `*Analysis` instances grouped by a column value. For example, if we have the family name `family` of inputs in the dataframe, it could be interesting to make separated analysis of each of them:

```
for sub_analysis in analysis.groupby('family'):
        print(sub_analysis.description_table())
```

These previous lines will describe the analysis of each family of `my_analysis`.

Pairs of Experiment-wares

To compare more precisely the overall pairs of experiment-wares, a method is implemented to return the corresponding analysis:

```
for sub_analysis in analysis.all_experiment_ware_pair_analysis():
        print(sub_analysis.description_table())
```

5.4 Draw Figures

After having built the analysis and manipulated the data we want to highlight, we can start drawing figures. Thanks to *Wallet*, we are able to build two kinds of plots: static and dynamic.

Wallet permits to draw static plots and computing tables showing different statistic measures. These figures can easily be exported in a format specified by the user, such as LaTeX for tables and PNG or vectorial graphics (such as SVG or EPS) for plots. Static plots are highly configurable in order to fit in their final destination (e.g., in slides or articles).

5.4.1 Static Tables

Each table that will be described hereafter are exportable into the LaTeX format. In addition to this transformation, it is possible to personnalize the the number pattern:

- `dollars_for_number` puts numbers in math mode (for LaTeX outputs);
- `commas_for_number` splits numbers with commas in math mode (for LaTeX outputs).

  Each table generated are observable in this notebook.

Describe the Current Analysis

Before manipulating the analysis, it could be interesting to describe it:

```
analysis.description_table()
```

which yields the following:

	analysis
n_experiment_wares	13
n_inputs	300
n_experiments	3900
n_missing_xp	0
n_inconsistent_xp	2
n_inconsistent_xp_due_to_input	0
more_info_about_variables	.data_frame.describe(include='all')

This first method allows to fastly understand how is composed the campaign. Here, simple statistics are shown, as the number of experiment-wares, inputs, experiments or missing ones, but one can also show exhaustively the different variable descriptions by applying `<analysis>.data_frame.describe(include='all')`.

Describe the Errors

If it exists missing data, the *Wallet* analysis can print a table showing what are these missing experiments by calling:

```
analysis.error_table()
```

which yields the following:

	input	experiment_ware	cpu_time	Checked answer	Wall-clock time	Memory	Solver name	Solver version	timeout	success	user_success	missing	consistent_xp	consistent_input	error	family
3576	XCSP19/hcp/graph45.xml	cosoco 2	0.045158	ERR UNSAT	0.04514	4021	cosoco	2	2400	False	False	False	False	True	True	hcp
3596	XCSP19/hcp/graph48.xml	choco-solver 2019-09-16	2306.48	ERR WRONGCERT	583.697	7.55305e+07	choco-solver	2019-09-16	2400	False	False	False	False	True	True	hcp

The Statistic Table

The table allows to show a global overview of the results through the following statistics:

- `count` is the number of solved inputs for a given experiment-ware;
- `sum` is the time taken by the experiment-ware to solve (or not) inputs (including timeout inputs);
- `PARx` is equivalent to `sum` but adds a penalty of `x` times the timeout to failed experiments (*PAR* stands for *Penalised Average Runtime*);
- `common count` is the number of inputs commonly solved by all the experiment-wares;
- `common sum` is the time taken to solve the commonly solved inputs;
- `uncommon count` corresponds to the number of inputs solved by an experiment-ware less the common ones (the common ones could be considered as easy inputs);
- `total` the total number of experiments for a given experiment-ware.

```
analysis.stat_table(
    output='output/stat_table.tex',
    commas_for_number=True,
    dollars_for_number=True,
)
```

This table is given by calling the previous method with different parameters:

- `par` corresponds to the different values we want to give to the PARx column(s);
- `output` is the path to the output we want to produce (e.g., a LaTeX table).

experiment_ware	count	sum	PAR1	PAR2	PAR10	common count	common sum	uncommon count	total
VBS	270	90388	90388	162388	738388	65	405	205	300
PicatSAT 2019-09-12	246	192377	192377	321977	1.35878e+06	65	11093	181	300
Fun-sCOP hybrid+CryptoMiniSat (2019-06-15)	209	274323	274323	492723	2.23992e+06	65	16472	144	300
Fun-sCOP order+GlueMiniSat (2019-06-15)	190	320070	320070	584070	2.69607e+06	65	14632	125	300
AbsCon 2019-07-23	168	341387	341387	658187	3.19259e+06	65	2805	103	300
choco-solver 2019-06-14	168	369846	369846	686646	3.22105e+06	65	7875	103	300
Concrete 3.10	165	369615	369615	693615	3.28562e+06	65	5182	100	300
choco-solver 2019-09-16	165	372266	372266	696266	3.28827e+06	65	7790	100	300
choco-solver 2019-09-20	165	372316	372316	696316	3.28832e+06	65	7754	100	300
Concrete 3.12.3	156	386276	386276	731876	3.49668e+06	65	7198	91	300
choco-solver 2019-09-24	149	390634	390634	753034	3.65223e+06	65	2570	84	300
BTD 19.07.01	135	421087	421087	817087	3.98509e+06	65	6718	70	300
cosoco 2	127	448425	448425	863625	4.18522e+06	65	6810	62	300

The Pivot Table

The pivot table allows to show exhaustively a precise variable between the set of experiment-wares (rows) and inputs (cols).

```
analysis.pivot_table(
    index='input',
    columns='experiment_ware',
    values='cpu_time',
    output='output/pivot_table.tex',
    commas_for_number=True,
    dollars_for_number=True,
)#.head()
```

- `index` permits to precise what we want in the rows;
- `columns` permits to precise what we want in the cols;
- `values` permits to precise what we want to show in the cells as information crossing `index` and `columns`.

input	AbsCon 2019-07-23	BTD 19.07.01	Concrete 3.10	Concrete 3.12.3	Fun-sCOP hybrid+CryptoMiniSat (2019-06-15)	Fun-sCOP order+GlueMiniSat (2019-06-15)	Picat-SAT 2019-09-12	VBS	choco-solver 2019-06-14	choco-solver 2019-09-16	choco-solver 2019-09-20	choco-solver 2019-09-24	cosoco 2
XCSP17/AllInterval/AllInterval-m1-s1/AllInterval-035.xml	3.65329	0.011283	2.146	6.80993	1.6944	14.1492	228.644	0.031203	0.604	241.5105	31.5242	769.121	914.919
XCSP17/AllInterval/AllInterval-m1-s1/AllInterval-040.xml	3.77132	0.045357	1.2	4189.84	14.8833	14.6022	290.328	0.045317	5.6885	61.7533	91.5793	846.7505	0.347661
XCSP17/Bibd/Bibd-sc-open/Bibd-sc-85-085-36-36-15.xml	2520.04	2519.91	2520.16	2520.2	2520.44	2520.28	2520.07	140.442	2520.42	2520.66	2520.74	2520.05	140.442
XCSP17/Bibd/Bibd-sc-stab1/Bibd-sc-25-05-01.xml	2520.1	2519.89	2520.11	2520.14	45.027	43.2998	21.1751	21.1751	2520.63	1666.8	1680.89	379.265	2519.75
XCSP17/Bibd/Bibd-sc-stab1/Bibd-sc-25-09-03.xml	2520.1	2519.72	2520.1	2520.07	1689.17	2520.14	515.664	137.506	137.506	260.369	211.421	2520.11	2520.02

The output is truncated.

The Contribution Table

This last table proposed by *Wallet* allowing to show the **contribution** of each experiment-ware:

- `vbew simple` corresponds to the number of times an experiment-ware has been selected in the VBEW;
- `vbew d` corresponds to the number of times an experiment-ware solves an instance `d` second(s) faster than all other solvers;
- `contribution` corresponds to the case that an experiment-ware is the only one that has been able to solve an input (a.k.a. state-of-the-art contribution).

As for the previous table, one just needs to call the following method:

```
analysis.remove_experiment_wares(['VBS']).contribution_table(
    output='output/contribution_table.tex',
    commas_for_number=True,
    dollars_for_number=True,
)
```

NB: the previously created virtual experiment-ware *VBS* is removed to avoid errors in the computations.

`deltas` correspond to the list of `vbew d` we want to show in the table.

experiment_ware	vbew simple	vbew 1s	vbew 10s	vbew 100s	contribution
BTD 19.07.01	76	28	11	1	0
cosoco 2	59	35	17	9	5
PicatSAT 2019-09-12	40	35	30	9	0
Fun-sCOP hybrid+CryptoMiniSat (2019-06-15)	38	38	35	18	0
AbsCon 2019-07-23	19	19	15	1	0
Fun-sCOP order+GlueMiniSat (2019-06-15)	16	16	11	5	3
choco-solver 2019-09-24	14	14	6	1	0
choco-solver 2019-06-14	7	6	5	3	0
Concrete 3.10	6	6	5	1	0
Concrete 3.12.3	4	4	4	0	0
choco-solver 2019-09-16	3	3	2	1	0
choco-solver 2019-09-20	1	1	1	0	0

5.4.2 Static Plots

Wallet proposed many plots to show data. Static plots have some common parameters:

- `figure_size`: size of the figure to output (inches);
- `title`: the figure title;
- `x_axis_name`: the x-label title;
- `y_axis_name`: the y-label title;
- `output`: output path to save the figure or `None`;
- `color_map`: a map to force the color of each experiment-ware line;
- `style_map`: a map to force the line style of each experiment-ware line;
- `title_font_*`: the title font properties;
- `label_font_*`: the label font properties;
- `latex_writing`: if `True`, allows to write in LaTeX mode;
- `logx`: log scale for the x-axis;
- `logy`: log scale for the y-axis;
- `[x|y]_[min|max]`: set the limit of an axis, or `-1` to take the default value of `matplotlib`;
- `legend_location`: the four legend positions (Position.RIGHT, Position.LEFT, Position.TOP, Position.BOTTOM);
- `legend_offset`: a couple `x` and `y` as offsets for the current legend location;
- `ncol_legend`: number of columns for the legend (default: `1`).

  A full example of a static plots is given in this notebook.

Static Cactus-Plot

A first kind of plots that allows to consider an overview of all the experiment-wares is the *cactus plot*. A cactus plot considers all solved inputs of each experiment-ware. Each line in the plot represents an experiment-ware. Inputs are ordered by solving time for each experiment-ware to build this figure: the x-axis corresponds to the rank of the solved input and the y-axis to the time taken to solve the input, so that the righter the line, the better the solver. Note that we can also cumulate the runtime of each solved inputs to get a smoother plot.

```
analysis.cactus_plot(
    # Cactus plot specificities
    cumulated=False,
    cactus_col='cpu_time',
    show_marker=False,

    # Figure size
    figure_size=(7, 3.5),

    # Titles
    title='Cactus-plot',
    x_axis_name='Number of solved inputs',
    y_axis_name='Time',

    # Axis limits
    x_min=50,
    x_max=None,
    y_min=None,
    y_max=None,

    # Axis scaling
    logx=False,
    logy=False,

    # Legend parameters
    legend_location=Position.RIGHT,
    legend_offset=(0, 0),
    ncol_legend=1,

    # Style mapping
    color_map={
        'VBS': '#000000'
    },
    style_map={
        'VBS': LineType.DASH_DOT,
    },

    # Title font styles
    title_font_name='Helvetica',
    title_font_color='#000000',
    title_font_size=11,
    title_font_weight=FontWeight.BOLD,

    # Label font styles
    label_font_name='Helvetica',
```

(continues on next page)

(continued from previous page)

```
    label_font_color='#000000',
    label_font_size=11,
    label_font_weight=FontWeight.BOLD,

    # Others
    latex_writing=True,
    output="output/cactus.svg",
    dynamic=False
)
```

By default, the cactus plot draws its graphic by using the `cpu_time` of the results: you are free to change this behaviour by replacing the `cactus_col` parameter. You can ask this plot to cumulate the runtime by giving `cumulated=True`. We can show and hide markers thanks to `show_marker` parameter. The legend ordering corresponds to the decreasing order of the number of solved inputs for each experiment-ware.

Static CDF-Plot

Equivalently to cactus plot, one may instead use the so-called *Cumulative Distribution Function* (CDF), which is well-known when considering statistics. In this plot x-axis corresponds to the y-axis of the cactus-plot (time), and the y-axis corresponds to the normalized number of solved inputs. A point on the line of the CDF may be interpreted as the probability to solve an input given a time limit.

```
analysis.cdf_plot(
    # Cactus plot specificities
    cumulated=False,
    cdf_col='cpu_time',
    show_marker=False,

    # Figure size
    figure_size=(7, 3.5),

    # Titles
    title='CDF-plot',
    x_axis_name='Time',
    y_axis_name='Number of solved inputs',

    # Axis limits
    x_min=None,
    x_max=None,
    y_min=None,
    y_max=None,

    # Axis scaling
    logx=False,
    logy=False,

    # Legend parameters
    legend_location=Position.RIGHT,
    legend_offset=(0, 0),
    ncol_legend=1,
```

(continues on next page)

(continued from previous page)

```
    # Style mapping
    color_map={
        'VBS': '#000000'
    },
    style_map={
        'VBS': LineType.DASH_DOT,
    },

    # Title font styles
    title_font_name='Helvetica',
    title_font_color='#000000',
    title_font_size=11,
    title_font_weight=FontWeight.BOLD,

    # Label font styles
    label_font_name='Helvetica',
    label_font_color='#000000',
    label_font_size=11,
    label_font_weight=FontWeight.BOLD,

    # Others
    latex_writing=True,
    output="output/cdf.svg",
    dynamic=False
)
```

By default, the CDF plot draws its graphic by using the `cpu_time` of results: you are free to change this behaviour by replacing the `cdf_col` parameter.

Static Box-Plot

In addition to cactus and CDF plots, one may consider *box plots* to get more detailed results about the runtime of each solver. A box in such a plot represents the distribution of each experiment time of a given experiment-ware. In particular, such plots allow to easily locate medians, quartiles and means for all experiment-wares in a single figure. We can find a practical application of this plot in the case of randomized algorithms: it permits to visualize the variance and to simply compare the effect of changing the random function seed for a given fixed solver configuration using it.

```
analysis.box_plot(
    # Box plot specificities
    box_by='experiment_ware',
    box_col='cpu_time',

    # Figure size
    figure_size=(7, 7),

    # Titles
    title='Box-plots',
    x_axis_name=None,
    y_axis_name=None,
```

(continues on next page)

(continued from previous page)

```
    # Axis limits
    x_min=None,
    x_max=None,

    # Axis scaling
    logx=True,

    # Title font styles
    title_font_name='Helvetica',
    title_font_color='#000000',
    title_font_size=11,
    title_font_weight=FontWeight.BOLD,

    # Label font styles
    label_font_name='Helvetica',
    label_font_color='#000000',
    label_font_size=11,
    label_font_weight=FontWeight.BOLD,

    # Others
    latex_writing=True,
    output="output/box.svg",
    dynamic=False
)
```

By default, the box plot draw its graphic by using the `cpu_time` of results: the user is free to change this behaviour by replacing the `box_col` parameter. Also, by default, the `box_by` parameter is set to `experiment_ware` meaning that each box represents an experiment_ware. The user may like to replace this by another column, for example the `family` col, and explore family data distributions.

Static Scatter-Plot

Finally, to get a more detailed comparison of two experiment-wares, one can use scatter plots. Each axis in this plot corresponds to an experiment-ware and displays its runtime (between $0$ and the timeout). We can place each input in the plot as a point corresponding to the time taken by both experiment-wares to solve this input. We can quickly observe if there exists a trend for one experiment-ware or the other in terms of efficiency.

```
rename = {
    "PicatSAT 2019-09-12": '$PicatSAT^{2019-09-12}$',
    "Fun-sCOP+CryptoMiniSat": '$^{Fun-sCOP}/_{CryptoMiniSat}$'
}

a2 = analysis.add_variable(
    'experiment_ware',
    lambda x: x['experiment_ware'] if x['experiment_ware'] not in rename else rename[x[
↪'experiment_ware']]
)

a2.scatter_plot(
```

(continues on next page)

(continued from previous page)

```
            "$PicatSAT^{2019-09-12}$",
            "$^{Fun-sCOP}/_{CryptoMiniSat}$",
            scatter_col="cpu_time",
            title=None,

            color_col="Checked answer",
            x_min=1,
            x_max=None,
            y_min=1,
            y_max=None,
            logx=True,
            logy=True,

            figure_size=(7, 3.5),

            legend_location=Position.TOP,
            legend_offset=(0, -.1),
            ncol_legend=2,

            title_font_name='Helvetica',
            title_font_color='#000000',
            latex_writing=True,
            output="output/scatter.svg",
            dynamic=False
    )
```

To draw a scatter-plot, we need to specify the experiment-wares on the x-axis and tge y-axis: `xp_ware_x` and `xp_ware_y`. By default, the scatter plot draw its graphic by using the `cpu_time` of results: you are free to change this behaviour by replacing the `scatter_col` parameter.

5.4.3 Dynamic Plots

Dynamic plots can be called by simply setting the `dynamic` parameter to `True`.

For example:

```
my_analysis.get_scatter_plot(dynamic=True)
```

5.5 Advanced Usage

For a more advanced usage, it is possible to get the original *pandas Dataframe* and to manipulate it thanks to this instruction:

```
df = analysis.data_frame
```

Then simply follow *pandas documentation* or more concisely this *pandas cheat sheet*.

If the user keeps the minimal necessary information in the modified dataframe, a new `Analysis` could be instanciated (with the optional success and consistency lambda checkers):

```
analysis = Analysis(data_frame=modified_df)
```

Every previous static tables correspond to pandas DataFrame and are thus manipulable.

5.6 Make an Optimality Analysis with `OptiAnalysis`

To make an optimality analysis, the user needs to parse and get back some needed information:

- the usual `input`, `experiment_ware`, `cpu_time`, `timeout` columns
- the additional columns:
  - `bound_list` is the list of all found bounds during an experiment
  - `timestamp_list` is the corresponding timestamp of each bound of bound_list
  - `objective` is equal to `min` for minimization problem else `max`
  - `status` informs the final status of the experiment (`COMPLETE` or `INCOMPLETE`)
  - `best_bound` is the final found bound before the end of the resolution

5.6.1 Create an `OptiAnalysis`

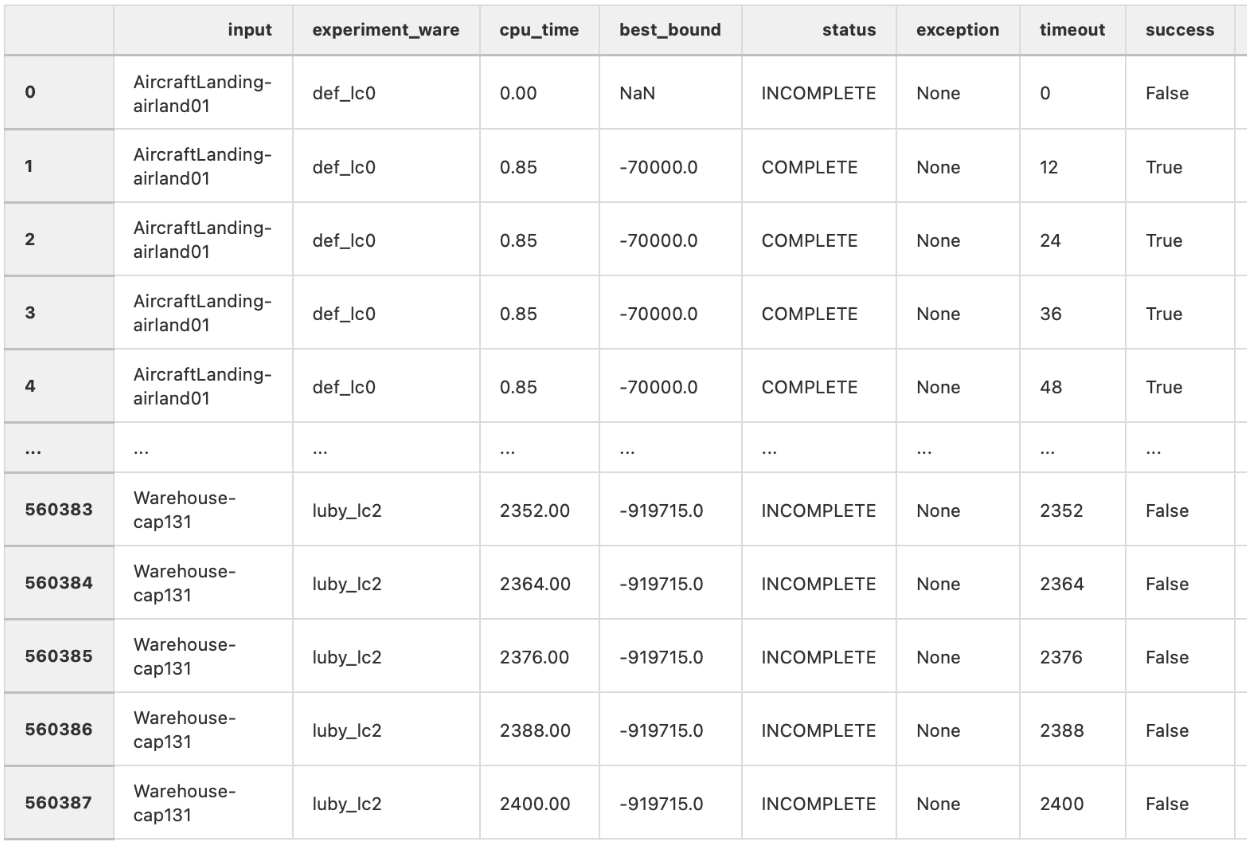
Once the previous needed data are well filled out in the yaml file (an example here), we can build a first optimality campaign as follows:

```
samp = [1,10,100,1000]
analysis = OptiAnalysis(input_file=SCALPEL_INPUT_FILE, samp=samp)
```

The parameter `samp` permits to explode the experiments in many timestamps that will permit to compute a score for each of them. In the example we focus on four timestamps (1s, 10s, 100s, and 1000s): this is an exponential way of observing results but a linear view is also interesting.

A default function `default_explode` is given by default to explode these data, but the advanced user could give another one to well-matching with its own extracted data.

Once constructed, the `analysis` object has this next data-frame in memory:



	input	experiment_ware	cpu_time	best_bound	status	exception	timeout	success
0	AircraftLanding-airland01	def_lc0	0.00	NaN	INCOMPLETE	None	0	False
1	AircraftLanding-airland01	def_lc0	0.85	-70000.0	COMPLETE	None	12	True
2	AircraftLanding-airland01	def_lc0	0.85	-70000.0	COMPLETE	None	24	True
3	AircraftLanding-airland01	def_lc0	0.85	-70000.0	COMPLETE	None	36	True
4	AircraftLanding-airland01	def_lc0	0.85	-70000.0	COMPLETE	None	48	True
...	...	...	...	...	...	...	...	...
560383	Warehouse-cap131	luby_lc2	2352.00	-919715.0	INCOMPLETE	None	2352	False
560384	Warehouse-cap131	luby_lc2	2364.00	-919715.0	INCOMPLETE	None	2364	False
560385	Warehouse-cap131	luby_lc2	2376.00	-919715.0	INCOMPLETE	None	2376	False
560386	Warehouse-cap131	luby_lc2	2388.00	-919715.0	INCOMPLETE	None	2388	False
560387	Warehouse-cap131	luby_lc2	2400.00	-919715.0	INCOMPLETE	None	2400	False

We can observe that the same couple (input, experiment-ware) appears many times – for each sampling asked by the user, visible through the timeout column. Each tuple composed of a specific (input, experiment-ware, timeout) is composed of the best_bound at this time, the current status and the success column that inform about the actual performances.

A full example here.

5.6.2 Compute scores

Now we have a well constructed `analysis` we can apply scoring methods thanks to the `compute_scores` method:

```
analysis.compute_scores(
  score_map=DEFAULT_SCORE_METHODS,
  default_solver=None
)
```

where:

- `score_map` is a dictionary of scoring methods with their names and the function to apply :

```
DEFAULT_SCORE_METHODS = {
    'optimality': optimality_score,
    'dominance': dominance_score,
    'norm_bound': norm_bound_score,
    'borda': borda_score
}
```

- `default_solver` is a default solver permitting to apply an additional operation (with additional data variables in the final data-frame) permitting to compare scores to a default solver score.

By default the different methods inside `DEFAULT_SCORE_METHODS` will be applied on each observation:

- `optimality` is equal to 1 if the optimality is found/proved or 0
- `dominance` is equal to 1 if the current bound is equal to the best one for this input
- `norm_bound` is the normalization of the current bound, based on min and max values found for this input at the current time
- `borda` is based on the Borda voting method by rating each solver for a given input; "Complete Scoring Procedure" in this page

The advanced user could give its function following this schema:

```
def dominance_score(x, min_b, max_b, df):
    return 1 if x['best_bound'] == max_b else 0
```

where:

- `x` is the current experiment/observation
- `min_b` is the worst found bound at the current time of the analysis for a given input
- `max_b` is the best found bound at the current time of the analysis for a given input
- `df` is the dataframe of the current analyzed input experiments (from which `min_b` and `max_b` are computed)

  A full example here.

  with a preview of created score columns:

opti	dominance	norm_bound	borda	opti_less_def	dominance_less_def	norm_bound_less_def	borda_less_def
0	0	0.000000	0.000000	0	0	0.000000	0.000000
1	1	1.000000	1.631439	0	0	0.000000	0.120805
1	1	1.000000	1.631439	0	0	0.000000	0.120805
1	1	1.000000	1.631439	0	0	0.000000	0.120805
1	1	1.000000	1.631439	0	0	0.000000	0.120805

5.6.3 Make figures

Finally, the user is now able to draw figures with the previously computed scores by giving, for example, `col='borda'`:

```
analysis.opti_line_plot(
    col='borda',
    show_marker=False,

    # Figure size
    figure_size=(5, 3),

    # Titles
```

(continues on next page)

(continued from previous page)

```
    title='',
    x_axis_name='Temps $t$',
    y_axis_name='Borda score',

    # Axis limits
    x_min=None,
    x_max=None,
    y_min=None,
    y_max=None,

    # Axis scaling
    logx=False,
    logy=False,

    # Legend parameters
    legend_location=Position.RIGHT,
    legend_offset=None,
    ncol_legend=1,

    # Style mapping
    color_map=None,
    style_map=None,

    # Title font styles
    title_font_name='Helvetica',
    title_font_color='#000000',
    title_font_size=11,
    title_font_weight=FontWeight.BOLD,

    # Label font styles
    label_font_name='Helvetica',
    label_font_color='#000000',
    label_font_size=11,
    label_font_weight=FontWeight.BOLD,

    # Others
    latex_writing=True,
    output=f'fig/borda_score.pdf',
    dynamic=False
)
```

A full example is given here

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search