WP1/D1.1
Data gathering, preparation and format

Version	Date	Description
v0	20/4/2019	Initialization
v1	01/07/2019	Deliverable

Partner Identification	Role
Grenoble INP - LIG	Main author
UBFC - Femto-ST	Contributor
USC, Australia	Contributor
SRL, Norway	Reviewer
Smartesting	Reviewer

1- Introduction

The objective of the PHILAE project is to generate and maintain automated regression tests by performing execution trace analysis and triage using machine learning techniques, combining this analysis with model inference and automated test generation. Thus, PHILAE aims at leveraging data available from development (esp. validation) and usage of software systems to automatically adapt and improve regression tests.
Figure 1 shows the main processes involved in this project, in four iterative and incremental steps:

1. Execution traces coming from the system in operation but also from manual and automated test execution are used to select trace candidates as new regression tests. Search-based algorithms and coverage metrics will be used to classify and select traces;
2. From selected traces and existing workflows, active model inference is used to infer updated workflow models, which align with the current state of the implementation;
3. Reduced regression test suites are generated from the updated workflows, and these are then executed on the current system implementation;
4. Based on the test execution results, the defects detected, and the development meta-data (such as commits in the code repository), a smart analytics and fault reporting system provides information on the quality of the system.

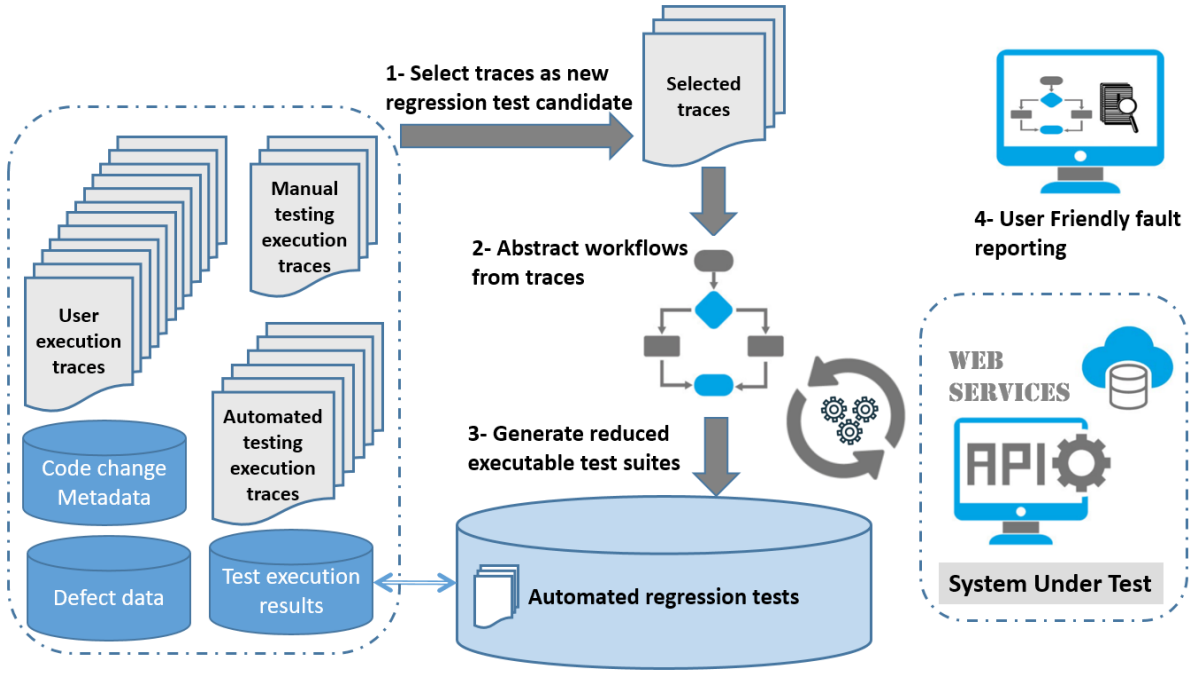


Figure 1 – PHILAE process

This deliverable D1.1 is a result of task T1.1-Define data gathering, preparation and

clustering of the work package WP1- Select traces as new regression test candidate. The goal of T1.1 is to define what type of information to extract from the trace and what requirements do exist on the trace format, to enable us organizing the data for feeding learning tools.

This deliverable D1.1 describes the types and the organization of the data used in PHILAE as inputs, so that they can feed the PHILAE processes, in particular initially as input to Machine Learning (ML) tools.
The main input for PHILAE consists of execution traces, that are collected either during validation phases (automated tests, additional tests that can be performed “manually) or while the system is running in the field. In order to develop generic tools in PHILAE that are independent of a specific software system, there are some common features for the content and description of the data collected from traces.

Scope of the document
This document summarizes the status of the studies at the beginning of the PHILAE project concerning the input data for ML techniques, and more particularly the characterization of execution traces, as well as their representation in an appropriate format.
It aims to present this format and illustrates its usage in the PHILAE case studies.

Limitation of the document
This document contains the thinking status of the PHILAE project at the date of delivery (T0+9 months). As the work progresses, the considered data will be extended (beyond the execution traces) and the formatting processes will also evolve.

Organization of the document
Deliverable D1.1 is organized into 4 main sections:

- section 2- A partial taxonomy of PHILAE data, which characterizes the input data required for any PHILAE process (see Figure 1)
- section 3- Framework for describing datasets, which introduces the proposed data format for PHILAE
- section 4- Illustration of case studies, assessing how this format can be used in PHILAE case studies
- Section 5- Conclusion and perspectives, providing a synthesis of the results and further work in PHILAE.

2- A partial taxonomy of data for PHILAE

2.1 Glossary

Event: a record of values related to the execution of a software at a single point in time; it can consist of values of program variables (including values handled by the O/S such as program counter, memory used etc), of messages, and measured external values such as current time. An event results from some observation of the execution of the program, either

internally by the program (that can write values in a log file), or externally from its environment (in particular the O/S, or some probe). An event can be a vector or a structured record of several values.

Trace: a sequence of events.

2.2 Analysis

PHILAE gets information from software testing and usage, mostly in the form of execution traces, using several sources and formats of information. At this stage of the PHILAE research, we do not focus on specific types of data, as long as they fit into the general framework of the project, as represented by Figure 1. Therefore, in this section, we attempt to cover the variety of these sources of information from software development and operation, and their content. This will help in identifying the types and formats of the data for the following phases of PHILAE.
We try to categorize them using a taxonomy based on several dimensions, that can be associated with each dataset. Each source of information (typically a file containing a trace) can come from a different phase of the lifecycle of a software,can be recorded at some level of the architecture of the system, and can contain more or less details about recorded events.

The position of a dataset along all those dimensions can be captured by metadata associated with the dataset.
We identify the following dimensions:

- **Nature** of the data: to differentiate for instance runtime call logs, interaction logs, information from development (such as code change and commits, bug tracking records etc).
- **Origin**: identifying the phase in the software lifecycle and development activities that actually produced (not just defined) the dataset ; figure 1 stresses user execution traces (from operational use), automated test traces (from regression testing) and manual testing traces (from initial test stages).
- **Observation point**: identifying where in the architecture the data is observed to be recorded (user interface, internal interfaces…)
- **Abstraction level**: data records can contain information with more or less details on the software objects and activities (e.g. just calls, or calls with parameters and sometimes stack trace)

For instance, for the shop-scanner case study (see Section 4 for a full description), an execution trace (Nature=trace) recorded in a simulated session can originate from beta-testing or demonstration (Origin), can contain events observed from the Man-Machine Interface (Observation point), may correspond to one action from a simulated user (Abstraction level= user actions) and may contain a record of information such as user-id, action type, date, etc.

Once a dataset is categorized along those dimensions, as described in some metadata associated to the dataset, its overall structure (and possibly encoding rules) can be described. Finally, the content of the dataset follows the structure.

So a dataset can be described as a triple :

(description-metadata, structure, content)

where *description-metadata* is itself a tuple (*nature*, *origin*, obs-*point*, *level*).
Of course, the content of execution traces can be huge, but the metadata would remain much smaller and easy to process
Based on the state of the art for data to be used by ML tools and the typical organization of execution logs, as will be seen in section 3, we will propose a format (GDF) for describing the datasets.
The structure part can describe the overall organization (fields of information, their types and encoding) of the data, and it can provide means for reading the content of the dataset. For instance, for CSV files with one event per line, the structure would describe the columns. For a content XML, it can be an XML schema.

2.3 Nature

PHILAE will mostly use traces, but it can also use other forms of data such as code changes, or information on defects. Based on the current case studies, we group them as follows:

- **Execution traces**: sequences of observations collected by a software system over some period of time. They typically come from some instrumentation: logging internal events (such as trace points in the code, function calls) or communications, or collecting observations when exercising the system (interaction logs).
- **Monitoring traces**: sampling characteristics of a system at runtime, that may not be directly linked to the workflow of the system. As seen in the Orange livebox case study (See Section 4 for a full description), useful information can be collected from sources that are not directly triggered automated or manual tests.
- **Non-Execution traces**: data not related to the execution of the system, such as general information on user profiles and behaviours, information on development processes etc.

2.4 Origin

Data can come from different phases of the software life cycle. In PHILAE, we consider in particular the following phases that can provide data. Note that some may produce data that would mostly be of “non-execution” nature (such as development, before the program is actually run and tested), others would produce mostly (at least in bulk) logs.

- **Development**: corresponding to the design and coding phases and associated processes, in particular
  - design: specification of new features, definition and change of architecture, components, functions, modules, etc.

    - coding: code changes, modification and enrichment of data types, code refactoring
    - debugging: if available, recorded debug sessions
    - version control: commits and associated code chunks or tests, diffs between versions
- **Validation and Verification**: All abstract and concrete traces collected from test executions, model checking, symbolic execution. Test verdicts also fall into that category.
    - Unit tests and associated architecture elements and components, expected values
    - Integration tests (e.g., values used in stubbing)
    - System test (test logs as well as internal logs)
    - Regression tests
    - Other types of automated tests (e.g., acceptance, performance testing, etc.)
- **Field and beta testing:** Traces collected from operational execution environments during beta testing phases. Interestingly, these traces are representative of the potential problems related to the operational usage of the software-system in its execution environment.
- **Exploitation:** Traces collected from operational execution environments during exploitation. These traces include:
    - Execution logs: recording events and interactions ;
    - Monitoring logs: sampling observations of the system ;
    - User traces and feedbacks ;
    - Error traces with bug report and observation.

2.5 Observation point

Recorded data can come from observations made inside the software system (e.g., traces produced by calls to loggers introduced by developers), or from external view points where the system if considered as a black box: typically user traces, or test records. Information recorded from process development such as those from bug trackers or version management are another type of viewpoint.
It is also important to relate the observation with the architecture of the software. Even for observations from inside the software system, observability can be compromised by the relation between cause of events and the point where the event is observed. The distance can induce delays (and wrong interleaving of observations w.r.t. real order), masking etc.

2.6 Abstraction level

Execution traces and monitoring records are typically structured as sequences of events. Each observed event can record information under the form of a vector of values, or with a more complex structure that may vary along the execution, for instance depending on the type of event recorded. The variation between these information records is known as abstraction level.

2.7 Requirements on PHILAE formats

The PHILAE project handles execution traces as first class citizens. Although the information collected in traces are dependent on the application, we can identify generic types of contents in each event that are particularly relevant for PHILAE.

Temporal ordering

For traces, two particular fields of information for each observation in the sequence are of particular interest:

- Index. A sequence is an indexed list of observations, Indexes can be explicit (countered indexes) or implicit (deduced from the listing)
- Timestamp. Each observation is usually timestamped. Most of the time the timestamp is explicit, although for periodic monitoring, it can be just equivalent to an index.

Location

At low level, software logging tools can provide a code line number for each recorded event, or a reference to a logging point. This is sometimes called a tracepoint (because it is the point where a call to a tracing library is inserted). At a higher level, it can be a component id or a subsystem in a distributed system, or just an observation point. Note that if the location is shared by all events of a trace, it might not be recorded (and thus repeated) in each event but in the metadata associated to the trace.

3- Framework for describing datasets

3.1 State of the art

Trace format for software execution logs

In most organized software development, traces would be produced by dedicated tracing libraries.

The organization of the information in such traces can be analyzed at three different levels:

- Conceptual level, giving the semantic structure of the information; e.g., a trace consists of sequence of events, each event starts with a location in a program, followed by a timestamp, followed by an event type, and - depending on the event type - arguments whose domain is defined
- Logical level. organizing the fields of information with precise computer data structures (e.g. record with named items, numerical value, text...) to be mapped to the physical level
- Physical level. the exact format, with syntax and encoding rules, that can be separated (as in e.g. ASN.1 and CER) or mixed (as in XML or JSON) that define both the syntax and the encoding.

Defining traces data using these 3 layers enables interoperability between trace producers and consumers (monitors, analyzers but also stores, transmission channels, etc.). For

instance, transmission channels and stores may only need to know about the Physical model, while filtering tools may ignore the Conceptual model (although the actual definition of the filter would be defined by an expert aware of the conceptual model). On the other hand, trace providers and analyzers can use the Logical model to remain independent from the Physical model (such as a wire or file format) and know the minimum about the Conceptual model required for the task at hand.

At the physical level, traces use either generic multi-purpose formats or more dedicated formats specific for event logs.
The most common generic formats used for traces are: CSV, JSON, XML that are all string based representations with no concern for compactness. Some tools may use more compact binary representations such as CBOR (http://cbor.io), or encoding rules from ASN.1 standards, such as BER, CER.
More specific formats for event logs can also be used, such as syslog (used by most operating systems for logging system events), or XES (eXtensible Event Stream http://www.xes-standard.org) used in Business Process engineering.

PHILAE should not depend on a specific physical format for traces, although this implies that specific encoders/decoders may be needed to deal with each type of trace encoding. The actual encoding/decoding can be done with existing libraries. For PHILAE tools, the adaptation will mostly lie at the logical level.

Organization of data for applying ML tools

Ideally, the representation framework used in PHILAE should be as open as possible to provide us with a wide choice of ML tools. As traces are complex data structures, composed of sequences, timestamps, application domain dependent properties, etc using available ML representations may be insufficient. The classical representations have 4 drawbacks:

1. They are poor in terms of what is possible to express since they limit themselves to only represent information directly used/usable by these tools.
2. Consequently very few information are retained concerning the semantics of the data and more generally about specificities of application domains.
3. The “encoding step”, that is to say the way we go from the initial data to the representation needed by the learning tool can be complex and mostly done by hand (or by writing a wrapper): that is inappropriate with one of PHILAE's objectives which is to automate as much as possible trace analysis.
4. These formats are mainly seen to represent “input files”, forgetting that in some cases datasets can be seen as the output of previous learning processes, thus leading to multiply the representation formats.

Classical data format in ML

In some ML models, a set of observations is expressed as a table (matrix or tensor) in which rows correspond to observations (examples) and columns correspond to features (variables). In this table each cell contains a value which is typically a number or a string (enumerated or ordinal values) but that needs to be turned into a numerical value for many

ML tools. Concerning the format to use for this table there are several possibilities. Among the most popular we find:

- Pure CSV file, in which feature semantics is mostly described with a text document independently from the dataset itself as in the UCI repository. Most of the classical ML libraries (scikit-learn, Panda, R, ...) use this kind of representation along with some tools to perform modifications or filtering while reading the data.
- ARFF files used by Weka which allows one to declare the features (type and in some cases the range) used in the data set. Amazon ML format also allows one to declare the type of the attributes using a “Data Schema”, the data themselves being classically written using a CSV file.
- Orange format that is based on header lines in the CSV file. In the oldest version only a single line is used while three lines are used in the latter.

ARFF, Amazon and Orange formats provide little information about features’ contents and their semantics. Indeed, the set of types proposed is limited to the standard ones (number, enumerate, string,...) that are close to elementary computer types. . An illustrative example is the case of “modular values” such as *hours*, *month* or *angle* : by using a raw numerical representation there is no min/max values and modular computations Consequently, some classical functions such as the geometric distances functions may compute wrong results if they ignore the modular computations.

Representation of traces, time series or sequences

Standard statistical ML is performed on unordered sets of data. But when what is learnt is sequential patterns of time-stamped values (possibly multidimensional), specific algorithms can be employed. The sequences of events that constitute the main material for PHILAE fall into the category called “time series” in ML.

To represent time series (TS), there is no universal format. In practice, as explained in table 1, data structure in CSV files can be done following three main approaches depending on whether data are mainly organized around the notions of features, examples or both. In the two first approaches, TS can be described in the files using either a row format or a column format.

Data Structure	File contains	Organisation	Examples
Features based	One pair example/feature	Features are folders containing one file per example, this file contains one TS that can be described either with row or column format	
	N examples per feature	Associated to each feature there is a file describing all the examples.	UCR data format (only one feature in this case), class of the TS is at the beginning of each row
Examples based	One pair based feature/example	Examples are folders containing one file per feature, this file contain one TS	
	N features per	Associated to each example there is	In Orange data format for TS, each

	example	a file describing all the features (assuming all of them are based on the same timestamp)	column (feature) has a header.
Combined	All data are organized into just one file	The file is structured by block of several lines (features or examples).	This is a classical representation in Data Analysis, in Philae Orange “use case”, use a variant with one common timeline for all the events.

Table 1 – Formats to represent time series

Regarding the way events (i.e. values) composing the TS are *indexed* in the sequences, there are two situations according to the role of the “timestamps”:

- First case: timestamps are not relevant and/or events are evenly spaced (regular sampling done by a sensor for instance). In such cases, according to the file organization, events are *implicitly indexed* by an integer that is either corresponding to the line number (column format) or the position in the row (row format).
- Second case: timestamps are meaningful and/or events are unevenly spaced (for instance when recording a human activity). In such cases, the timestamp values/scale *must be explicitly* provided and coded in the data files. According to the file organization see above, there are two possibilities:
    - *Column format*: one (or several) specific column (i.e. feature) contains the timestamp values.
    - *Row format*: a TS is composed of a list of pairs <timestamp, value>.

It is worth noticing that in the previous table, column based representations often use *explicit timestamps*, expressed into one or several columns of the file, while line based representations often use an *implicit index*, corresponding to the ranks of the values in the sequence. Finally, an important point to take into account for PHILAE is that in some use cases the traces could have distinct “clocks” or timestamps references and that merging these timestamps to create a unique temporal references is not always trivial. Thus being able to deal with multiple references coming from different sources might prove necessary for some PHILAE processing.

3.2 GDF for PHILAE: potential use and benefits

As part of a previous project (PIA IKATS) Grenoble LIG began to develop a generic format that is tool-independent and that captures most characteristics of data that may be used by tools. It is called GDF (General Data Format). This format fits well the objectives of PHILAE to enable automation of test analysis activities through generic descriptions. Actually, the GDF ambition seems to exceed the needs of PHILAE, as it is also intended to support many sorts of activities including learning, data exploration and visualization. The main advantages of GDF are:

- To gather multiple input data and put together complementary meaning and semantics into a unified data format.
- To simplify the “coding step” by providing extended representational features, including modular types, hierarchy, etc.
- To ensure data quality and traceability (a crucial challenge) first by keeping the

maximum amount of information and second by checking that the data are syntactically correct (i.e range, unit, precision, ...)
- To provide more information to the learning tools through the notion of meta-information.
- To be able to convey the learning results in a homogeneous way from one tool to another to create complex dataflows. In this respect, the goal is different but complementary of work like Predictive Model Markup Language or more recently Portable Format for Analytics allowing to describe and exchange predictive model produced by statistical or Machine Learning tools.

More related to PHILAE project, the advantages of using GDF are threefold:
- By providing a clear semantic, this representation language helps to have a clear and more complete specification of the use cases.
- The current version of GDF contains some wrappers allowing to read and export to the classical representation language used in ML. Of course, during this translation step some information is lost, but the wrapper does its best to generate an accurate encoding. In this sense GDF seems to be an interesting “pivot language” …
- The representation language is one of the project’s background, developed at LIG by a participant to PHILAE, thus it will be easier to adapt some parts of the language to the needs of the project. For instance the header part of GDF could be improved to provide a better representation of the *description-metadata* tuple (*nature*, *origin*, obs-*point*, *level*) presented in section 2.

3.3 Brief overview of main GDF constructs for describing datasets

The full description of GDF can be found in Appendix. We just introduce here the main fields of GDF that will be illustrated in section 4. In GDF, a dataset is composed of three parts:
- A **profile** describing the semantic of all features,i.e., variable, TS/sequences, patterns, etc.
- A **data table** using the classical matrix format observations/features,
- A **set of files** describing the time Series or more generally sequences that do not fit into the data table.

The **profile** provides the metadata and structure description (as in the triple identified in section 2). It consists of two sections:
- A **header**, made of doctype, date, dataset name, source and optionally authors and learning set from which this file could have been generated (as GDF can be used not only for raw data, but also for derived results of various processing)
- A so-called **dictionary**, that lists the features (variables). For a trace, it would consist of a sequence in a section called **sequences**.

Each event of a sequence in a trace would be described with:
- **name**: that should express the intended semantics
- **type**: number, boolean, string, enumerate, timestamp...
- **domain**: e.g. integer or float for a number, possibly refined with range and unit
- **index**: can be a timestamp or an integer

- **meta**: this field can be useful for providing hints to ML tools, such as weighting the importance of that type of element, or ignoring for classification.

Then comes the second and third parts of the dataset. PHILAE will mostly use execution traces, and these would come as the third part. In GDF, this would typically be coded as pointers (directory and filename) to files containing the sequences. Other types of PHILAE information (e.g. statistics on development, code metrics etc) would be represented in the second part (variables).

4- Overview of PHILAE case studies' datasets and preliminary analysis and processing of datasets

We describe the types of data that we have considered in the initial analysis of the case studies of PHILAE. We provide a description of the data and give examples of GDF descriptions when appropriate.
Each of the three case studies is given with different levels of information and traces.

- The Orange case study provides test logs from external interactions, monitoring of hardware resources, and no access to the internal events of the software, or even its structure or code (firmware).
- The Shop-scanners gives complete control on the source-code and any observation instrumentation.
- The School bus system provides records of usage traces in the field.

For each case study, we provide a description of the raw data, a preliminary analysis of its content and the preparation of data.

4.1 Orange Livebox case study

Context and overview

Orange provides equipments called "Liveboxes" that serve as networking gateways for communicating devices inside home or office. Currently, there are 3 types of liveboxes: livebox 3, livebox 4 and livebox Pro.

Orange livebox is a router (with ADSL or optical fiber connection to the telecom operator's network) that has been produced by Orange and is available to customers of Orange's Broadband services in France, United Kingdom, Kenya, Guinea, Tunisia, and Spain. The Orange livebox can be used for a number of purposes such as Wifi router, VoIP, and digital TV (IPTV). As technology evolves, the expectations of customers also rise. As a result, to ensure the quality, endurance, and customer satisfaction for these liveboxes, Orange Telecom applies a performance validation test on the liveboxes. There is a dedicated testing team called Labo Multi Service (LMS), whose main task is to validate the updates done on the firmware provided by a company affiliated to Orange.

The LMS developed a number of performance tests which are composed of specific high-level scripts of a dedicated tool (DriveYourTest), orchestrating middle level test scripts (called scenarios) that are done by RobotFramework, which itself resorts to lower level (Python)

scripts to actually create and retrieve network protocol events. The scripts trigger a number of navigation URLs or a number of other calls such as downloading and uploading files that are actually performed by several concurrent devices (PCs, smartphones, set-top boxes...) representative of a customer's environment. The scripts behave as customers when using liveboxes for different purposes. When a number of scripts are run in parallel they are called scenarios. These scenarios are executed on a daily, weekly, and monthly basis at the Lab.

Figure 2 shows the architecture of the testing platform, and figure 3 shows the flow of data among the architectural elements.

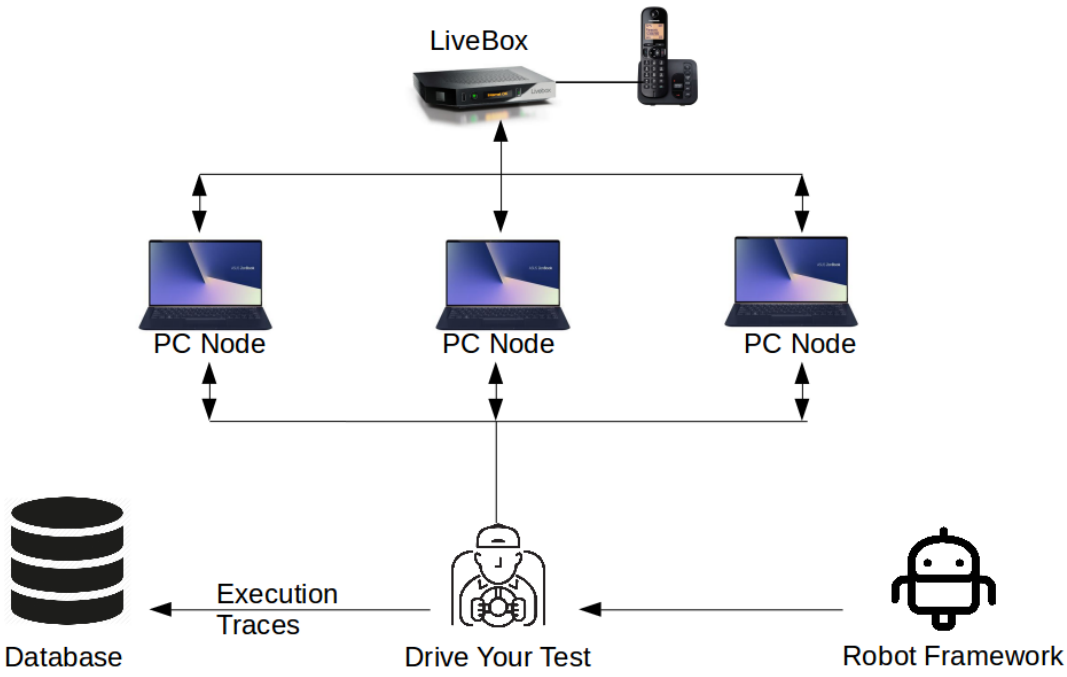


Figure 2 – Architecture of the testing platform

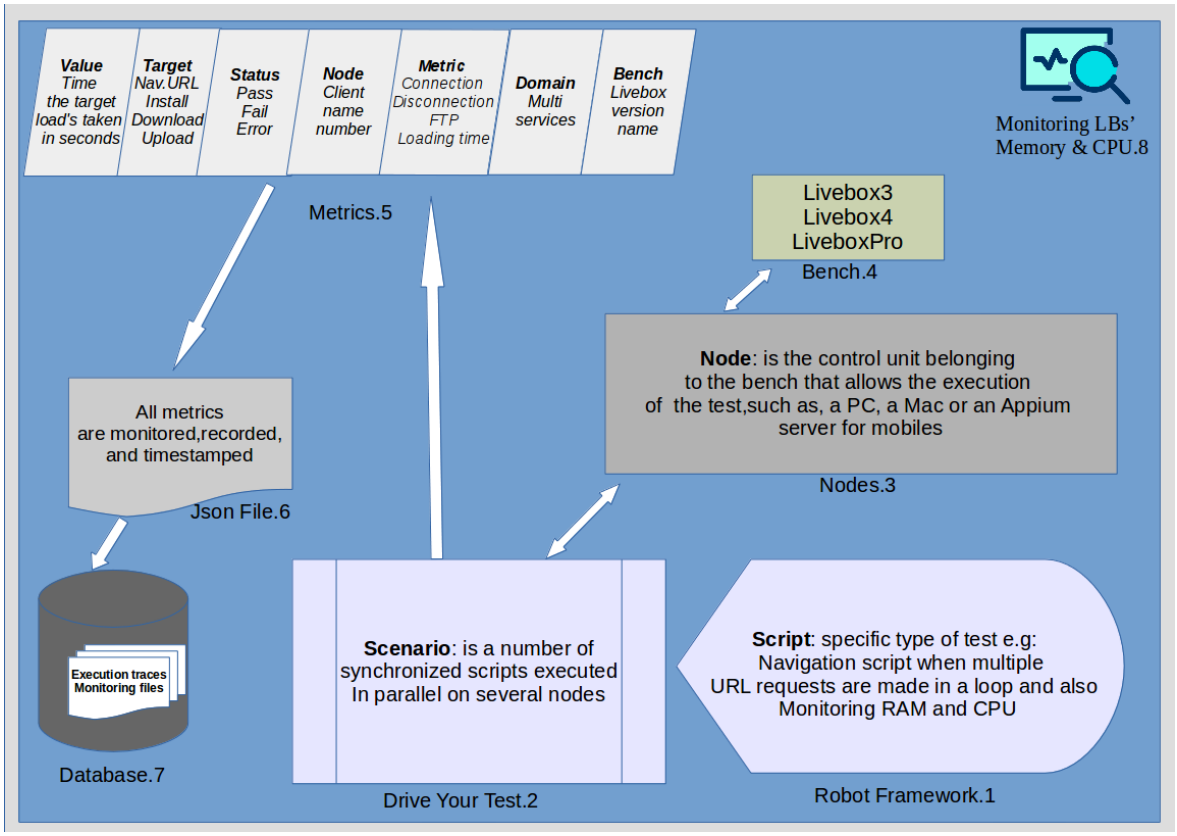


Figure 3 – Data flow diagram

There are two types of traces provided in this case study.

- Test logs, with one line for each action performed by one of the devices triggered by RobotFramework.
- Monitoring logs, that periodically sample the status of the O/S resources of the livebox, with classical measures such as memory and CPU use, number of packets etc.

There is one log file of each type per day (24 hour observation). Each log file actually collects all information from all test benches, so logs for Livebox3 are mixed with those of Livebox4 and LiveboxPro, and similarly monitoring mixes information from various liveboxes as well as set top boxes, and in fact different types of measures with different sampling rates.
Therefore, a preliminary step in processing the data from the raw log files consists in extracting the logs for one specific livebox type.

Livebox dataset and PHILAE approach

- Test logs correspond to an external observation point as it is not inside the livebox, but is actually collected inside the test harness. Adopting the PHILAE terminology, those test logs are simulated usage traces: they correspond to a high level

abstraction of global user actions (from a user's observation point) but they are still traces from regression tests. The type of test performed is not functional testing, but endurance testing.

- Monitoring logs are traces that are observed from an internal observation point, which are not triggered by the usage workflow. They correspond to "test execution results", but they do not contain the test verdicts. They are just "side effects" that need to be analyzed and interpreted. Currently, they are analyzed by test experts using data analysis tools, and one potential impact of ML techniques could be to automate the analysis.

Raw data 1: test logs

The livebox execution trace file records events that correspond to the executed scenarios on each livebox. Each trace file is composed of a number of timestamped sequences of records with over 20,000 events a day. Almost six months of trace files have been obtained from LMS for experimenting on the objectives of the PHILAE project. We have been provided with JSON files as well as a more compact binary form (BSON). Each trace file corresponds to exactly one day (from midnight to midnight) of observations, recorded in a JSON (Javascript Object Notation) file which is about eight megabytes.

The records' data types are either categorical or numerical. The JSON format structures key-value pairs as arrays of objects. One object in an array represents an event in the trace file. Most events in the traces correspond to client actions, typically navigating popular URLS such as Facebook, Ebay, Wikipedia or downloading files from the internet.
As seen below in the json example (Figure 4), an event is timestamped and contains the measured results of a test. For example, here is a test of livebox 4 such as: metric used, bench name, target name, (local) test status, (lab) domain type, node name (that created the request), and the computed values.


```
{
    "timestamp": "2018-08-02T00:04:34+00:00",
    "metric": "loading time",
    "bench": "Livebox4",
     "target": "NavWeb http://wikipedia.fr",
     "status": "PASS",
     "domain": "Multi-services",
      "value": 532.0,
      "node": "mac01"
},
```


Figure 4 – JSON example of an event from the execution trace

Raw data 2: monitoring files

The monitoring files contain the records of the status of Memory (RAM), CPU, and WIFI channel of Liveboxes while being tested. As for test logs, there is one file per day. Each monitoring file is composed of a number of time-stamped records with over 10,000 events in a single day file. Actually, the status of resources for various devices (liveboxes and set-top box) are sampled at a period of around 5 minutes, but for each device, several resources are sampled at the same time, each one corresponding to an event in the file. The record's data types are either categorical or numerical. On a daily basis one trace file is recorded in a JSON file which is about five to six megabytes. Figure 5 represents an event example which was recorded on 02/08/2018 at 21:54:25 on Livebox3.
The meaning of the fields differ from the execution traces.
More precisely, the node does not relate to an external node from the environment, and the target is usually the intended livebox version to be monitored as shown in the example event in the figure. The metric variable on the other hand is recording the stats of the RAM, CPU, or WIFI Channel.
Actually, at a given sampling time, around a dozen different metrics and corresponding values are recorded, all of them with the same time-stamps. The period for sampling is typically 5 minutes for a given livebox. Note also, that a single monitoring file record mixes monitoring information for several liveboxes and other devices (such as set-top box), with different metrics depending on the device.
Almost six months of monitoring files have been obtained from LMS for experimenting on the objectives of the Philae project.


```
{
    "value": 1.3,
    "node": "monitoring",
    "timestamp": "2018-08-02T21:54:25+00:00",
    "domain": "Multi-services",
    "target": "livebox->process->",
    "metric": "cpu",
    "bench": "Livebox3"
  },
```


Figure 5 – JSON example of a monitoring record

Analysis

The first task on execution traces consisted in performing statistical analysis in order to observe the data at hand. Thus, some simple analytics have been done on Livebox execution traces in order to acquire some understanding of the data, and comparing it with the “ideal” perception conveyed by the presentation of this data by Orange LMS.
The analysis raised a number of issues, some of them appeared linked to specific cases known to LMS team members, and others were not really expected.

The naive analysis confirmed the overall architecture described above. A first observation is that on a single day, three test scenarios are run from (01 to 05-07am, 02 to 09 - 10am, and

18 to 21-22pm). Each test session continuously runs for almost six hours. The second observation is that three livebox versions have been tested (livebox3, livebox4, and liveboxPro). In addition, the three tested liveboxes have a number of clients connected to them, from client1 to client10. Besides, the three liveboxes' events are all recorded in a single execution trace file.
Unexpectedly, test execution traces of one day we considered (26/08/2018) have a high rate of Fail verdict, significantly on Livebox3 (see Figure 6 below).

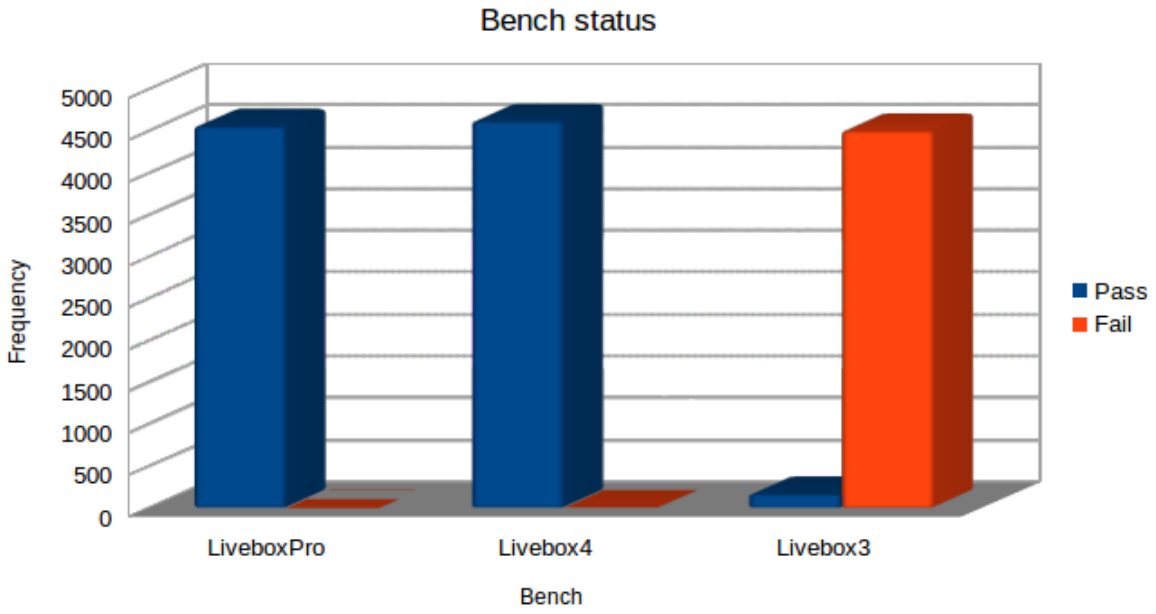


Figure 6 – Number of pass and fail verdicts on 26/08/2018

Also, execution traces of the same day also have a significant ratio of Error verdict for Appium01 client. Finally, it could be also observed that the metric, target, and value are strongly correlated.

Generally speaking though, any day would contain a high number of FAIL tests, an observation that did not match the feedback from Orange LMS that only few (new) bugs were discovered. FAIL status seems to be associated to local failure of a test script (e.g. by being unable to connect to some website at a given time), not to a failure of the livebox.
After doing a number of such analyses, we considered that identifying (real) failures from execution traces was not a task that an expert could easily do on such traces, and even less a machine learning tool.
Therefore, we based our second step in analysing the raw data on the other raw input: the monitoring files. Experts from the LMS confirmed that this was their starting point to investigate failures, as well.

To observe the data in the monitoring files, Grafana (https://grafana.com) has been highly used by the LMS testing team. Grafana is an open platform for monitoring data analytic and visualization. The LMS uses Grafana to observe, note and store the acquired Memory and

CPU statistics of the three liveboxes. Through Grafana (see figure), it can be noticed that the RAM and CPU status is monitored continuously over one week from 14-05-2019 to 20-05-2019 for Livebox3. One important measure that is seen in Grafana statistics is the Uptime values which show the time elapsed since last reboot. Figure 7 shows that two reboots happened during the week.

Figure 7 – Grafana screenshots showing uptime values, RAM and CPU status during a week

GDF description

Since there are actually two datasets of traces in the Orange Livebox case studies, we provide two GDF descriptions.

We first present the GDF description of a test log (here a single day, but several files could be concatenated).

```
{
    "header": {
        "doctype": ["GDF", 0.42],
        "date": "19-03-27 13:41",
        "dataset": "2018-10-14",
        "source": "philae-data",
        "authors": "Orange",
        "learningSet": []
},
    "dictionary": {
        "sequences": [
          { "name": "timestamp",
            "type": "timestamp",
            "domain": "iso_8601",
```

```
            "units": "seconds",
            "meta": [["ignore"]],
            "sampling": "uneven"
          },
          { "name": "Unnamed: 0",
            "type": "number",
            "domain": "integer",
            "comment": "the event number",
            "meta":[["id"]],
            "index":"timestamp"
           },
           { "name": "bench",
             "type": "enumerate",
             "domain": "String",
             "comment": "name of the livebox",
             "range": ["LiveboxPro","Livebox3","Livebox4"],
             "index":"timestamp"
            },
           { "name": "domain",
             "type": "string",
             "comment": "One type of livebox services",
             "meta":[["ignore"]],
             "index":"timestamp"
           },
           { "name": "metric",
             "type": "string",
             "comment": "connection/disconnection/loadingtime of
target(URL&Calls)",
              "index":"timestamp"
            },
           { "name": "node",
             "type": "enumerate",
             "domain": "String",
             "range": ["client01", "client02",
                       "client03", "mac01", "appium01",
                       "client04", "stb01", "client05",
                       "client06", "client07"],
             "comment": "clients on running test scenario on livebox",
             "index":"timestamp"
           },
           { "name": "status",
             "type": "enumerate",
             "domain": "String",
             "range":["PASS","FAIL","ERROR"],
             "comment": "pass/fail/error of events",
             "meta":[["class"]],
             "index":"timestamp"
           },
           { "name": "target",
             "type": "string",
             "comment": "URLs+Calls test scripts",
```

```
              "index":"timestamp"
           },
           { "name": "value",
             "type": "number",
             "domain": "float",
             "comment": "time taken in seconds to load,connect, or
disconnect the target",
             "index":"timestamp"
           }
    ]
}
```

Then the GDF description of the monitoring files.

```
{
    "header": {
        "doctype": ["GDF", 0.42],
        "date": "06-03-19 13:41",
        "dataset": "2018-07-15/2019-01-22",
        "source": "philae-data-monitoring",
        "authors": "Orange",
        "learningSet": []
},
    "dictionary": {
        "sequences": [
          { "name": "timestamp",
            "type": "timestamp",
            "domain": "iso_8601",
            "units": "seconds",
            "meta": [["ignore"]],
            "sampling": "uneven"
          },
          { "name": "id",
            "type": "number",
            "domain": "integer",
            "comment": "the event number",
            "meta":[["id"]],
            "index":"timestamp"
          },
           { "name": "bench",
             "type": "enumerate",
             "domain": "String",
             "range": ["LiveboxPro","Livebox3", "Livebox4",
                       "Lb5-MF1","Banc1-103","Banc2-103",
                       "Livebox_Dev_253","dev"],
             "comment": "name of the livebox",
             "index":"timestamp"

           },
           { "name": "domain",
             "type": "enumerate",
```

```
              "domain": "String",
              "range": ["Multi-services","Fonctionnel"],
              "comment": "The type of livebox services",
              "meta":[["ignore"]],
              "index":"timestamp"
            },
            { "name": "metric",
              "type": "string",
              "comment": "WiFi,download_rate,cpu_total,mem_free ect.. of
target(Livebox,STBPlay&P2P)",
              "index":"timestamp"
            },
            { "name": "node",
              "type": "enumerate",
              "domain": "String",
              "range": ["client01", "client02", "client03",
                        "mac01", "monitoring", "Monitoring",
                        "client06","client07","client08",
                        "modem01","p2p01"],
              "comment": "clients running test scenario on liveboxes",
              "index":"timestamp"
             },
             { "name": "target",
               "type": "string",
               "comment": "The Livebox,P2P, and STBPLAY being monitored",
               "index": "timestamp"
             },
             { "name": "value",
               "type": "number",
               "domain": "float",
               "comment": "time taken in seconds to load,connect, or
disconnect the target",
               "index":"timestamp"
             }
    ]
}
```

4.2 Shop-scanners (FEMTO-ST)

Shop-scanners is a simplified but realistic application implementing a system of shopping scanners. These devices allow customers to shop autonomously, from product-price recording, until reaching the checkout to proceed to the full payment.

The workflow is as follows: the customer unlocks the device as he/she enters the supermarket. He then takes products from the shelves and scans their barcode while adding them to his basket. He also has the possibility to remove products from his basket. It is possible that, for some products, the barcode is not recognized by the device. When the shopping is complete, he reaches the checkout and the data stored on the scanner is transferred to the checkout. At this step, it is possible that the cashier randomly proceeds to

a control check which consists in re-scanning a significant subset of the products in the basket, to check that no product has been "missed" during the shopping.

When interacting with the cashier, it is possible to request the assistance of a cashier:

- either manually, to add or remove products from the shopping list, and/or
- automatically if some products have not been recognized during the shopping.

In both cases the cashier has to authenticate on the checkout.

Implementations

We have two implementations of the system:

- a Java implementation made of 4 classes that describe the scanner, the products, a product database, and the cashier;
- a web-based simulator, developed as a (sort of) multi-agent system which can be used to produce the usage traces that are described hereafter.

In both cases, implementation traces are sequences of operation calls on specific objects. The two main classes which represent the externally-visible objects are the scanner and the cashier for which we provide a short specification below.

The scanner class

The `Scanette` class is composed of the following operations:

- `int debloquer()` unlocks the scanner. If the scanner was not blocked it returns status code -1, otherwise it unlocks the scanner and returns 0.
- `int scanner(long)` scans a product for which the EAN13 code is given in parameter. This operation is used either to add a product to the client's cart, or to check the products in the cart during the verification phase. It can return -1 if the operation is called in a wrong state, -2 if the code is not recognized during the shopping phase, -3 if the product is not recognized during the verification phase. If the product is recognized, whatever the state, 0 is returned.
- `int supprimer(long)` removes one occurrence of the article during the shopping phase. If the state is incorrect, it returns -1. If the product did not exist in the cart, it returns -2. Otherwise, the product is removed and the value 0 is returned.
- `int quantite(long)` makes it possible to retrieve the number of products that exists for a given EAN13 code. If the product does not exist, the quantity is 0.
- `void abandon()` cancels the current transaction, empties the cart and re-locks the scanner.
- `Set<Article> getArticles()` makes it possible to retrieve the set of product EAN13 codes that are present on the cart.
- `Set<Long> getReferencesInconnues()` retrieves the set of unknown references that have been scanned during the shopping phase.
- `int transmission(Caisse)` performs a connexion of the scan with the checkout. At this step, two results are possible, as the checkout may ask for a verification of the products inside the cart. The scanner enters a verification mode, in which the cashier

can re-scan a subset of products that are supposed to be in the cart. Otherwise the cart is transferred to the checkout with which the session continues. This operation returns -1 if it is called in the wrong state, 0 if the data were transferred to the checkout and the use of the scanner is over, and 1 if a verification is requested.

The following diagram (Figure 8) represents the different states of the scanner.

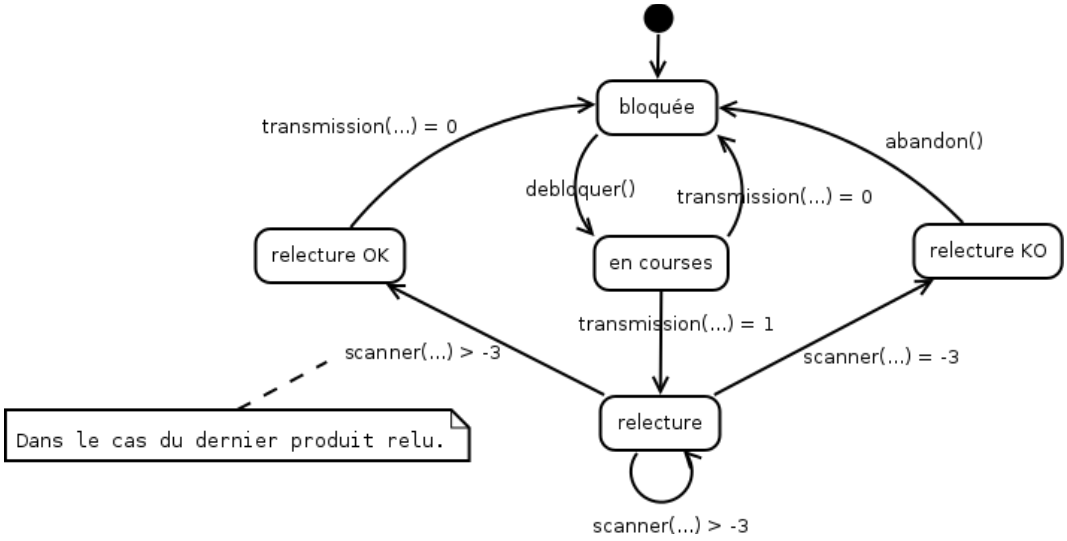


Figure 8 – State diagram for the scanner

The checkout class

The `Caisse` class contains the operations that make it possible to handle the connection of the scanner, and manage the shopping list before proceeding to payment:

- `int connexion(Scanette)` handles the connection for the scanner. It can be invoked to signal the end of the shopping phase. In this case, it randomly returns a verification request (10% rate) and returns 1. If no verification is requested, it returns 0. If it is invoked to signal the end of the verification, then it returns 0 and expects a payment (except if the shopping list is empty, in which case, the checkout waits for a cashier). If invoked in a wrong state, it returns -1.
- `double payer(double)` represents the payment. The parameter corresponds to the amount paid by the client. The return value indicates the possible change (>= 0). It returns a negative value if the checkout is in the wrong state when the operation is invoked.
- `void abandon()` cancels the current transaction, and returns to the state in which the checkout is waiting for a connection.
- `int ouvrirSession()` opens a session during which the cashier is authenticated, and can perform manual additions to (resp. removals from) the shopping list. If invoked from the wrong state, it returns -1. Otherwise if the operation succeeds, it returns 0.
- `int fermerSession()` closes an opened session. If invoked from the wrong state, it returns -1. Otherwise if the operation succeeds, it returns 0.
- `int scanner(long)` makes it possible to manually add a product to the shopping

list, by scanning its bar code (in parameter) directly on the checkout. If the product is successfully scanned, it returns 0. Otherwise it returns -1 if the checkout is in the wrong state, or -2 if the product is not recognized by the checkout.

- `int supprimer(long)` makes it possible to manually remove a product from the shopping list. It returns 0 if the product has been successfully removed, -1 if the checkout was not in the appropriate state, or -2 if the product did not exist in the shopping list.

The following state machine (Figure 9) summarizes the behavior of the checkout.

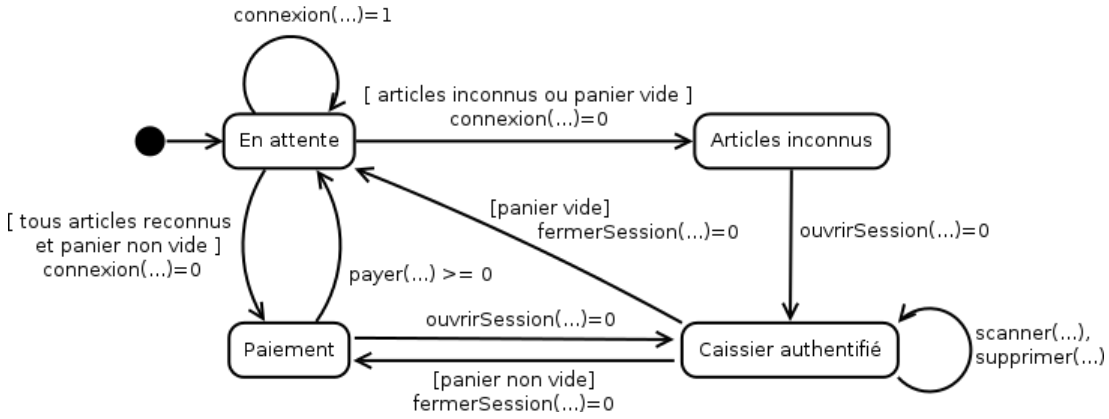


Figure 9 – State diagram representing the behavior of the checkout

Usage sessions

A usage session usually starts with the unlocking of a scanner by a client, followed by a shopping sequence in which products are scanned one by one. Then, the client reaches the checkout to transmit the cart. If there is a verification, a cashier scans a sample of the client cart to check the existence of the products before proceeding to the payment on the checkout. Once the cart has been transmitted, the scanner is no longer necessary and the payment can be processed on the checkout. At this step it is possible that the cashier is requested by the client to modify the shopped products by "manually" adding or removing references in the shopping list. The session ends with a payment.

The following state machine (Figure 10) represents the different usages of the system. In this diagram, the diamond state represents the initial state. A usage session is considered to be the use of the scanner, followed by the use of the checkout, after a transmission between them and a possible control check by the cashier.

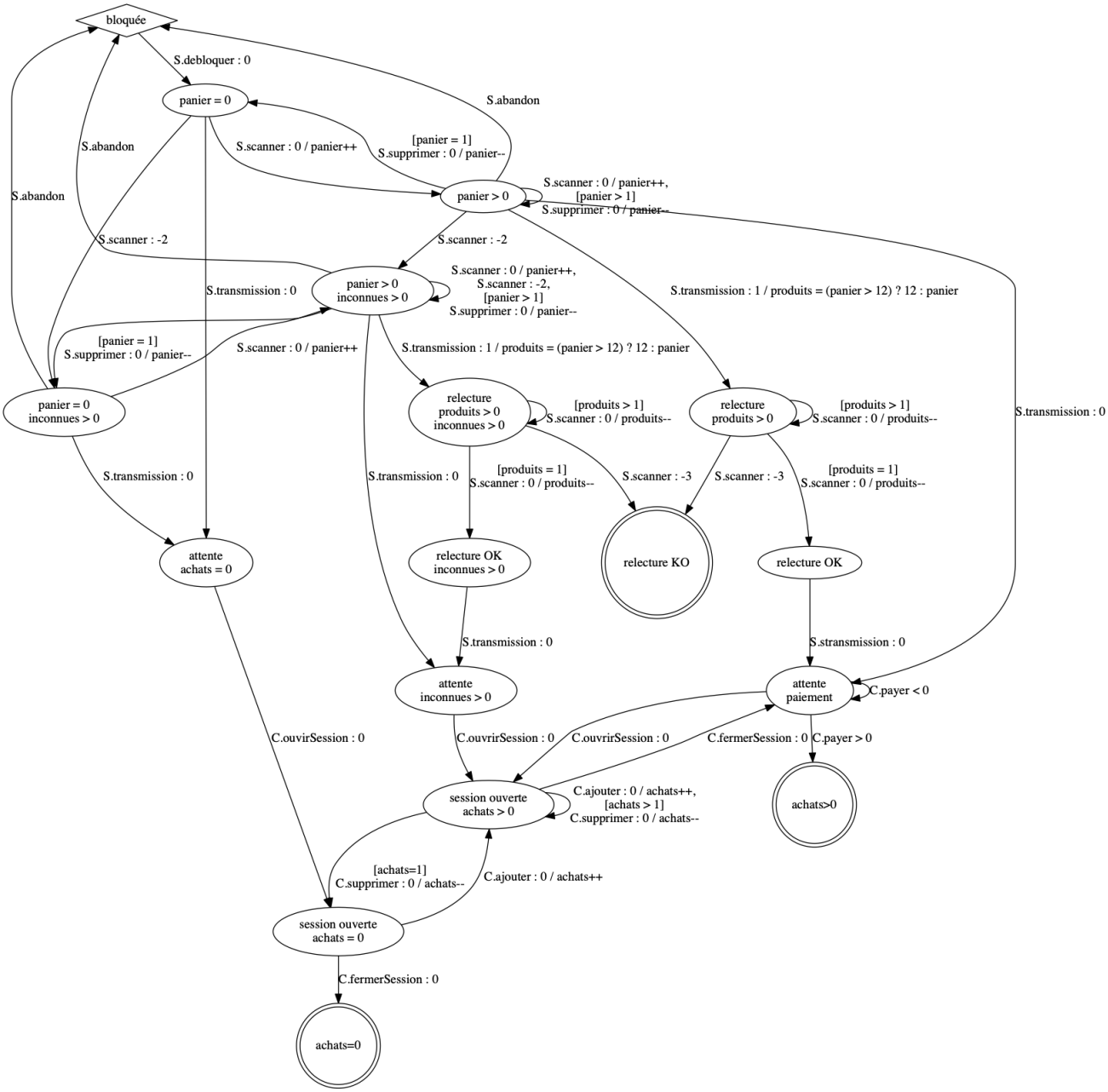


Figure 10 – State diagram representing the different uses of the system

Experimental data

For the experimental data, we used a set of 20 student implementations of the specification plus one reference implementation, along with different sets of developer tests.

The simulator is available at the following address: http://fdadeau.github.io/scanette?simu

It simulates the usage of the system by a set of customers that manipulate the device as in a real-world deployment of the system. The inner classes that are called to perform business actions are a faithful implementation of the Java version described above. The simulator is

used to produce the raw data that are described below.

Raw data

As the system is our own development, the content of the raw data can be easily adapted or extended. We present below the set of data that are available.

id	The ID of the entry (incremented each time an operation is called)
timestamp	The timestamp of the action that is logged
object	The object that is considered of type scanner (Scanette) or checkout (Caisse)
method	The operation/method that is invoked, which depends of the object that is considered
parameters	A list of parameters, possibly empty
return value	The return value of the operation which indicates the behavior of the operation that was activated

As the notion of session is essential to delimit one usage of the system, we add a complementary parameter which identifies the session. This preprocessing step is mandatory.

session	The ID of the session for the considered data set

The following table represents a simple usage session in which the client scans some products, including one unknown reference, and then reaches the checkout. Although there is no verification, the checkout requests a control check by the cashier due to the presence of unknown products. The cashier opens the session and manually adds the missing reference to the shopping list. The client then proceeds to payment.

```
1 ; session1 ; 1559635938725 ; scan1 ; debloquer ; [] ;  0
2 ; session1 ; 1559635947081 ; scan1 ; scanner ; [8718309259938] ; 0
3 ; session1 ; 1559635969806 ; scan1 ; scanner ; [3570590109324] ; -2
4 ; session1 ; 1559635974089 ; scan1 ; scanner ; [3520115810259] ; 0
5 ; session1 ; 1559635987261 ; scan1 ; transmission(caisse3) ; 0
6 ; session1 ; 1559635987262 ; caisse3 ; connexion ; [scan1] ;  0
7 ; session1 ; 1559635987263 ; scan1 ; abandon ; [] ; undefined
8 ; session1 ; 1559635995456 ; caisse3 ; ouvrirSession ; [] ; 0
9 ; session1 ; 1559636011475 ; caisse3 ; ajouter ; [3570590109324] ; 0
10 ; session1 ; 1559636015547 ; caisse3 ; fermerSession ; [] ; 0
11 ; session1 ; 1559636024337 ; caisse3 ; payer ; [25] ; 4.3799999999999
```

GDF profile of the dataset

We provide hereafter the profile of the dataset described as a GDF file named `profile.json`

```
{
  "header": {
      "doctype": ["GDF", 0.42],
      "date": "19-06-04 10:49",
      "dataset": "scanette0",
      "source": "PHILAE project - scanette",
      "authors": "F. Dadeau",
},
"dictionary": {
  "variables": [
     { "name": "id",
       "type": "number",
       "domain": "integer",
       "comment": "entry identifier",
       "meta": [ ["id"] ]
     },
     { "name": "sessionID",
       "type": "number",
       "domain": "integer",
       "comment": "usage session identifier",
     },
     { "name": "timestamp",
       "type": "date",
       "comment": "timestamp of the operation call",
     },
     { "name": "object",
       "type": "string",
       "comment": "the object instance on which the operation is called",
     },
     { "name": "operation",
       "type": "string",
       "comment": "the name of the operation that is invoked on the object",
     },
     { "name": "parameters",
       "type": "string",
       "comment": "a JSON list of parameters (usually integers)",
      },
      { "name": "returnValue",
        "type": "string",
        "comment": "the JSON value returned by the operation invocation",
        "meta": [ ["class"] ]
      }],
   "sequences": [],
   "patterns": []
  }
}
```

4.3 USC: Bus System

Goals

The goal of this case study is to use AI and Machine Learning techniques to fully automate the model-based testing of an existing web service. Starting from the API documentation of the web service, plus example traces of interactions with the web service, we will automatically build a simple model of the 'normal' behaviour of the web service, as well as running machine learning algorithms on them to identify higher-level abstract patterns in the test sequences. Then we will use that normal-behaviour model to generate several kinds of tests:

1. **regression tests**, which just replay the given traces as executable tests. This is useful for testing that existing behaviour still works after system updates;
2. **robustness tests**, which systematically insert various common types of errors (such as missing values, or out-of-range values) into traces in order to test the robustness of the web service;
3. **model-based tests**, these use more abstract behaviour models learned automatically from the example traces, in order to generate tests that fill in the missing gaps identified by the model. This can help to expose missing cases in the logic of the web service.

A key goal is that all these kinds of tests should be able to be generated and executed automatically, at the push of a button. This gives a gentle learning curve, which means that users can get the benefits of basic testing techniques without having to learn any new skills, and of more advanced model-based testing techniques simply by learning how to read models of their own system, rather than the more difficult skill of writing models.

Context and Overview

We want to test an existing web service for recording bus runs, where clients swipe an ID card as they enter or leave the bus, so the server can track progress of the bus in real time, as well as the entry and exit of each client. For bus routes where the clients are school students or disabled people, the system also has the capability to notify the parents/carers of those clients via text messages or email, etc.

The scenario is that each bus driver has an iPad in the bus, with GPS facilities, a card reader for reading client bus-cards, and mobile connectivity to the web server. The following diagram shows the general architecture of the existing system (in black), and how our proposed Automated Testing approach (in red) fits into that architecture.

The central web server maintains a database of companies, bus runs, clients, parents/carers, etc. The details of this database are out of scope of this testing project and should not need to be known in order to generate tests. But note that interactions with the server just ADD records to the database, so for testing purposes it is possible to reset the database back to a known state simply by deleting all new records after a certain date.

Figure 11 shows the architecture of the system.

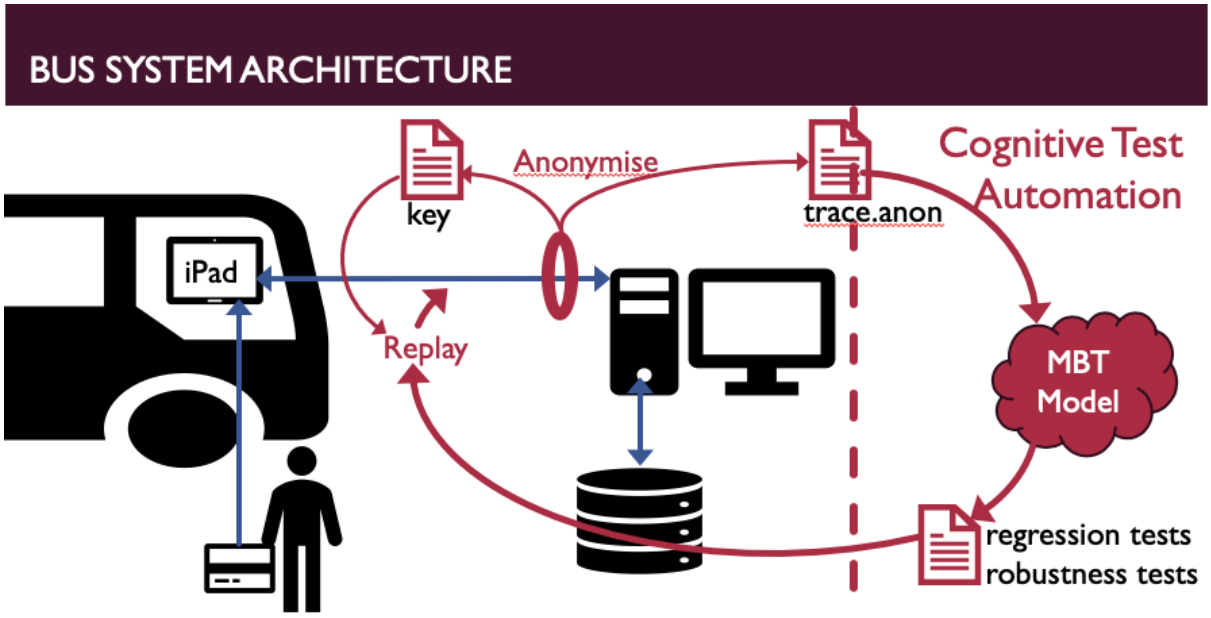


Figure 11 – Bus system architecture

Typical Scenarios

For this project, we will focus on testing two bus scenarios: pick-up runs and drop-off runs. In both these scenarios, the set of clients is known in advance, and all clients must be dropped off or picked up at a central location.
The typical operation of a bus pickup run is as follows:

- The driver logs on to the iPad, which records his username and password for the duration of the run. The iPad sends a **Login** message to the server, which responds with a set of safety questions that the driver must answer correctly, plus a list of bus runs that the driver can choose from.
- the driver answers the safety questions. These are sent to the server for validation in a **ConfirmPreCheck** message, which replies with a yes=0 or no=1 (plus error message) status. (This step can be skipped if there are no safety questions).
- the driver then chooses a bus run number, which is sent to the server in a **GetManifest** message. The reply to this message contains the list of client information for that run (client IDs, addresses, carer contact details, etc.);
- the driver then drives the bus along the run, and the iPad sends a **SaveGPS** message to the server each minute with the current GPS location, speed and time information, so that the server can track the bus location.
- the driver stops and picks up clients at each location. Each client taps their ID card as they enter the bus. This sends a **CheckIn** message to the server, including the time of the pickup and the GPS coordinates of the location.
- alternatively, if a client fails to appear for pickup, the bus driver can send a **MarkAbsent** message to the server to indicate that the client was absent.
- optionally, if an incident occurs, such as the bus breaking down or being delayed in

heavy traffic, the driver can send a **Notify** message to the server to inform a contact person about the delay, and possibly send a notification to all the carers of the clients who are waiting for pickup.

- or the driver can send a **ReportIncident** message about a client discipline issue.
- when the bus reaches the central drop-off location, all clients exit the bus and tap out. (Or alternatively the driver can press a button to send a single **BulkCheckOut** message, which sends the IDs of ALL the clients on the bus, but we ignore bulk checkins/outs in this simplified description).
- finally the driver logs off the iPad and exits the bus. This does not send any message to the server, as the web service is stateless (there is no 'connection' that needs to be shut down).

A bus drop-off run is similar, except that the role of **CheckIn** and **CheckOut** messages is reversed.

Trace Analysis and Test Generation

We have obtained one day of sample bus run traces from the bus software company, and have anonymized them to remove client details. This gives a single 2.5Mb file containing a single trace of all the events for the whole day, where each event is represented by an XML structure whose content is dependent upon the type of event.

Stage 1: The first stage of analysis was to break this single trace up into many different traces, based on the IP address of the bus where each request originates from. This gave several hundred traces. We then experimented with several clustering algorithms and approaches, to identify the most common kinds of traces. One useful clustering algorithm was to use a 'bag-of-words' approach to code each trace, and then K-means to perform clustering. This clearly identified the most common groups of traces, and the differences between the groups generally had clear real-world importance. By choosing one trace from each group we were able to obtain a small set of traces that were suitable for regression testing. By varying the K parameter to the K-means algorithm, we could easily control the size of this regression suite.

Stage 2: After discussing the above clustering results with the bus software company, it became clear that identifying traces based on IP addresses was undesirable (because the IP address of a bus changes as it moves between different cellular environments). Our second approach was to identify traces based on the bus driver ID. This gave a smaller number of more complete traces, all showing similar sequences of behavior. The clustering approach (based on 'bag-of-words' plus K-means) was still able to identify useful groupings of the traces, which reflected important real-world distinctions.

Stage 3: Our current approach (on-going) is to investigate the sequencing of operations in the traces in more detail, to learn a probabilistic finite-state model that can be used for test generation of sequences of abstract web service operations. In parallel with this, we are also investigating various machine learning algorithms to learn how to predict the input values for each web service operation. We expect that combining both these models will

allow us to generate a richer variety of model-based tests that can start to move beyond simple regression testing, into robustness testing and model-based testing.

GDF Description

The analyses described above were done directly on the raw data trace from the bus system. The format of this raw data is a single chronological trace per day (including all events for all buses), with each event represented as an XML dump of a web service query and the resulting response. The structure of each response is different for each kind of event, as different data is returned to the bus. This raw trace format does not fit easily into the current GDF structure. There are several different ways in which the raw traces can be transformed into GDF format, such as a separate GDF file for each bus, or a single GDF file with each row of that file being a complete trace for one bus. We are currently investigating the tradeoffs of the various approaches, to determine which of these views of the raw data will provide the best interface to the various machine learning algorithms.

Bus System Conclusions

We have successfully analysed a set of real-world traces from customer interactions with a web service. We have developed a simple algorithm that can choose a given number of representative traces for regression testing. The algorithm uses very little domain knowledge (just the choice of using 'username' to split the traces), so is likely to be applicable to other web services. The next stage will be to infer richer models of the traces, to move beyond regression testing into robustness and model-based testing.

5- Conclusion

Since the initial phase of PHILAE, the partners have worked on the three case studies described in this document in order to identify issues for data curation and preparation. This operation is typical in ML, where one goal is to prepare data to train ML models. This document details the dataset available from those case studies, relating them to the PHILAE approach of leveraging execution traces and other data associated to development for extracting relevant traces, enabling workflow retrieval related to regression testing issues.
The case studies provided by the PHILAE partners are diverse, and so are the types of data available. This resulted into two proposals for analyzing and presenting the datasets available:

1. A taxonomy to characterize the types of data formats to be used for execution traces (section 2)
2. A generic framework (GDF) for the description of datasets for ML tools (section 3)

The taxonomy associates key characteristics such as metadata about the available datasets, to trigger various processings in PHILAE tools. GDF (General Data Format) provides a generic format for datasets that is

- independent of specific ML tools, with converters to popular formats;
- usable at various stages of data processing, both for input and output formats;
- able to associate metadata (e.g., from the taxonomy) and some semantics with the dataset, and to describe the data organization.

Being able to provide a common framework for various case studies is essential for developing PHILAE tools.. It is known that data preparation is an important step (and often time consuming) before applying ML tools to perform clustering or classification tasks. The first attempt presented in section 4 indicates that we will need further generalization of the preparation tasks to come up with a global methodology for this step of the PHILAE approach.

The finalization of deliverable D1.1 concludes the first phase of the PHILAE project, In this document, we focused on 3 main points:

- Data acquisition exploration, analysis and preparation for machine learning, for clustering execution traces in particular;
- Study a common representation format (GDF) for PHILAE tools;
- Case studies examination and experiment.

Our conclusions are:

1. The PHILAE initial vision which was to extract information from test and operational execution traces is relevant for the industrial partners of the project.

2. By analyzing the various formats required for training ML models, it appears that software execution traces are difficult to handle without too many simplifications. Further research activities are thus necessary to develop a viable approach for learning from traces.

3. The three case studies provided in the PHILAE project are adapted to the PHILAE vision and foster interesting discussions among the partners. The generic GDF format offers us an appropriate initial tool to capture relevant information from the traces and will serve to feed the training process of ML tools employed in the project.

Appendix

GDF: General Data Format

I. General Data Format (GDF)
1. 1. Context and objectives
2. 2. State of the art
    α. 2.1 Classical data format
    β. 2.2 Representation of Time Series
    γ. 2.3 Representation of sequences
3. 3. Definition and organisation of a dataset
4. 4. Description of the dataset : profile part
    α. 4.1 Variables section
    β. 4.2 Sequences section
    γ. 4.3 Pattern section
5. 5. Description of the data table
    - 5.1 GDF table
    - 5.2 DAF table
    - 5.3 The case of special values (missing, unrelevant, …)
6. 6. Links to the external Sequences
    - 6.1 The specific case of inlined sequences
    - 6.2 Automatic file format detection
7. 7. Reading and exporting data format
    - Canonical data format for GDF
8. 8. Merging data files
9. 9. Example of GDF representation
10. Annex A : Composite types.
11. Annex B : Relational information.
12. Annex C : Representation of hierarchical clustering results.
13. Annex D : Flat representation of the profile.
14. Annex E : Change log.

General Data Format (GDF)

Date: 18th december 2018
Revision: V 0.5
Author: G. Bisson, J. Marinescu, S. Alkhoury
Project: IKATS
Document status: draft

Changes done with respect to version 0.46	
Description	**Sections concerned**
Description of differences between variables and patterns are better explained.	4.3
Introduce a new property *type* in the pattern to make more explicit the meaning of a pattern.	4.3
Introduce some examples, based on HARP, to explain how patterns are coded.	4.3
Introduce the new property *confidence* for the variables and sequences.	4.1, 4.2
Correction of many typos as usual.	

1. Context and objectives

This document describes a General Data Format (GDF) allowing to represent the data sets (i.e learning, validation, test, …) used in the learning & visualization tools developped by the LIG for IKATS project. Along with the classical ordinal and nominal variables, this format allows to represent datasets containing Time Series (TS) and more generally ordered sequences of values. Furthermore, it allows to store the discriminant patterns (i.e. small kind of regular expressions) that can be extracted from the sequences. Here these patterns are produced by HARP algorithm (Heuristical Approach to Research Patterns). Other possibilities allowing to improve the representational capacities of GDF would be introduced in future releases, preliminary versions of some new options are discussed in the different annexes of this document:

- Annex A : Definition of composite types (i.e combinations of basic types),
- Annex B : Notion of *object* allowing to express relational information between observations,
- Annex C : Storage of hierarchical clustering results using extensions proposed in Annexe B.

Ultimately, the goal of this format would to be to express a large spectrum of the data used in ML and in some cases to represent the results produced by the learning systems when they correspond to an “enrichment” of the initial data. When defining such kind of general format there are several requirements to fullfilled as explained in this page data format should be:

- Rich enough to represent and defined data types and application domain dependant information.
- Compact enough to not waste storage for large data sets.
- Be readable by various classical learning tools and by human being.
- Parts of the data can be transferred independently and easily to another format.
- Conversely, the other classical format must be simple to import or transform.
- Be textual as to enable munging using system/shell tools.

At first glance, defining a new representation language seems a little bit useless since, as we will see in section 2, many “general” data formats already exist. However, these formats were mainly defined to represent the “learning set” of the current (statistical) learning tools and are generally 1) poor in terms of what is expressed since they limit themself to only represent information directly used/usable by these tools, 2) consequently very few information are

retained concerning the *semantics of the data* and more generally about the specificities of the application domain and 3) these formats are mainly seen to represent “input files”, forgetting that in some cases datasets can be seen as the output of previous learning processes. Thus, in the context we are pursuing several goals allowing to:

- *Gather multiple input data* : put together complementary meaning and semantic into an unified data format.
- *Simplify the coding step* : by providing extended representational features.
- *Ensure data quality and tracability* : (a crucial challenge) first by keeping the maximum amount of information and second by allowing to check that the data are syntaxically correct (i.e range, unit, precision, …)
- *Provide more information to the learning tools* : that could lead to explore new kind of learning approaches, for instance to deal with dimensional analysis, new sort of data, …
- *Convey the learning results* : in an homogeneous way from one tool to another to create complex dataflows.

We need to emphasize that our goal is not to represent any kind of results/models produced by a learning process, but just those that can be seen an enrichment of the input data. A typical example of that use is are clustering algorithms, such K-means, whose output can be seen as an addition of a new attribute(s) in the learning set expressing observations’ category. In this respect, our goal is different but complementary of work like Predictive Model Markup Language or more recently Portable Format for Analytics allowing to describe and exchange predictive model produced by statistical or Machine Learning tools.

In this document, in parallel of describing semantical information managed with GDF, we are focusing on the notion of “text file” representation since they is the most portable and general way to store data. Nevertheless it is would be very easy to have a “database” version of the format in which all of pieces information will be managed by different tables.

2. State of the art

2.1 Classical data format

In Machine Learning (or in Data Analysis) a set of observations, classically named *Learning Set* (LS), is generally expressed as a table (matrix) in which rows represent observations (examples) and columns features (variables). Concerning the file format to use to represent this table into a file there are several possibilities, among the most popular we find:

- Pure CSV file in which the semantic of the features are mostly described with a text document independently from the dataset itself as in the UCI repository. Most of the classical ML libraries (scikitlearn, Panda, R, …) are able to use this kind of representation along with some tools to perform (complex) modifications or filtering while reading the data.
- ARFF files used by Weka which allows to declare the features (type and in some case the range) used in the data set.
- Amazon ML format also allows to declare the type of the attributes using a “Data Schema”, the data themself being classically written using a CSV file.
- Orange format that is based either on three (old version) or one (new version) header lines in the CSV file.

ARFF, Amazon and Orange formats are interesting but they provide little information about features’ contents and their semantics. Indeed, the set of types proposed is limited to the standard ones (number, enumerate, string,…) that are closer to computer language types than to a way to represent semantical information. In ARRF format is it possible to declare the range for nominal values.
In GDF we propose to manage a larger set of possibilities (cf. 4.1) allowing to express:

- Raw values that appear in the observations, that classically belong to three families:
  - *Quantitative values* allowing to measure or to count some quantities.
  - *Ordinal values allowing* allowing to rank some information or to express a preference.
  - *Qualitative values* allowing to express some properties or to provide data.
- Complex values that correspond either to (still missing in this version):
  - *Composite set of data*, for instance the 3D coordinates which are a mixture of three raw values X, Y, and Z, or geographical position expressed with latitude and longitude.
  - *Generalized values* that are produced by agregating/summarizing different values. These data can be for instance the

result of a clustering method. For each type of data there are many ways to represent such agregations: for instance a set of numbers can be represented such as an interval, the parameters of a statistical model, an histogram, etc.

2.2 Representation of Time Series

To represent TS there is no universal format. In practice, as explained in the table below, data structure in CSV files can be done following three main approaches depending on whether data are mainly organized around the notions of **features** or **examples** or both of them. In the two first approaches, TS can be described in the files using either a **row format** or a **column format**. In GDF, in order to help the user to reuse existing datasets, we are able to deal with all of these representations (see section 6).

Data structure	Each File contains	Organization of the different files	Observation/Exemple
Features based	One pair example/feature	Features are folders containing one file per example, this file contain one TS that can be described either with row or column format	Initial proposition of CS ?
	N examples per feature	Associated to each feature there is a file describing all the examples, in this file TS are oftenly described using row format	UCR data format (only one feature in this case) moreover, class of the TS (supervised learning) is provided at the beginning of each row
Examples based	One pair feature/example	Examples are folders containing one file per feature, this file contain one TS that can be described either with row or column format	-
	N features per example	Associated to each example there is a file describing all the features (assuming all of them are based on the same timestamp), in this file TS are oftenly described using column format	In Orange data format for TS, each column (feature) has an header. The same with Airbus 2 with 4 columns: 2 timestamps, ground speed and heading
Combined	All data are organized into just one file	The file is structured by block of several lines (features or examples). A classical representation used in statistics is to store from line 0 to N–1 the first example, N to P–1 the second one, etc. In this representation columns are the features.	This is a classical representation in Data Analysis, here we call this format “DAF” (Data Analysis Format)

Regarding the way events (i.e. values) composing the TS are *indexed* in the sequences, there are two situations according to the role of the “timestamps” associatd to these events:

- **First case**: timestamps are not relevant and/or events are evenly spaced (regular sampling done by a sensor for instance). In such cases, according to the file organization, events are *implicitly indexed* by an integer that is either corresponding to the line number (column format) or the position in the row (row format).
- **Second case**: timestamps are meaningfull and/or events are unevenly spaced (for instance when recording an human activity). In such cases, the timestamp values/scale *must be explicitly* provided and coded in the data files. According to the file organization see above, there are two possibilities:
  - *Column format*: one (or several) specific column (i.e. feature) contains the timestamp values.
  - *Row format*: a TS or sequence is composed of a list of pairs <timestamp, value>. This special format will not be used in GDF to store the temporal data. However it is possible to store the sequences and their associated timestamps into different files using the row format (see section 6 for more details).

Finally, it worth notice that in the previous table, column based representations often use an *explicit timestamps*, expressed into one or several columns of the file, while line based representations often use an *implicit index*, corresponding to the ranks of the values in the sequence.

2.3 Representation of sequences

The notion of TS can be easily generalized to the notion of *sequences* corresponding to series of ordered values. In such case, the values of the sequence can have also either an *implicit index* based on the position of the values in the sequence or an *explicit index* based on any ordered scale (distance, temperature, …).

As for the TS, this kind of data is very classical. For instance, that is the case of a DNA sequence in which each “base” (among A, T, G, C) can be indexed by its codon number. Moreover, in some cases, each value of a sequence can be indexed by multiples correlated scales: for instances, in geology, to describe a *core sample* (une “carotte” in french) each values can be indexed both by the distance to the ground and by its corresponding geological time.

3. Definition and organisation of a dataset

In the current proposal a dataset is composed of three parts:

- A **profile** describing the semantic of all features (i.e variable, TS/sequences, patterns, etc),
- A **data table** using the classical matrix format observations/features,
- A **set of files** describing TS or more generally sequences that doesn’t fit into the data table.

Of course, all these information could also be organized in a database but we focus here on a file based organization allowing to have a portable external format for the dataset.

In this document, the characteristics of a the profile are based on JSON format. For the other files we can use a classical csv file. Others, more human friendly possibilities, exist such as YAML. However, as the hereunder format is flat and homogeneous enough, another solution to ease writing/editing for a human being, is to allow the user to code the full profile using a classical spreadsheet. Such approach , base on using XSLS or ODF formats is explored in Annex D of this document.

4. Description of the dataset : profile part

The profile is composed of two parts:

- A *header* part providing some general (meta)-information about the source of the dataset. Let’s notice that we use here a very simplified/modified version of the Dublin Core. It would be interesting to discuss if more compatible version would make sense or if the current version is enough.
- A *dictionary* part describing the set of features used in the Learning Set, this dictionary being divided into three families of componants: variables, sequences and patterns. Of course, a dataset can contain just one or all of these families (see also Annex B introducing an extension to deal with multiple dictionaries).

In the rest of the description, the property with a leading “*” are always **mandatory**.

```
header : {
        *doctype: ['GDF', '0.46'],                    // representation language and version
        *date: 'yy-mm-dd hh:mm:ss',                    // creation date of the current file
        *dataset: 'official name of the dataset',      // id of the dataset
        *source: 'references/origin of the dataset",   // references to the data
        authors: 'name of the persons who collected data", // references of the authors
        learningSet: ['learningset1', ...]      // learning set(s) possibly used to generate this one
        }

dictionary: {
        variables:      [ ... ]         // List of the "classical" variables (i.e. mono-valued)
        sequences:      [ ... ]         // List of the TS or sequences
        patterns:       [ ... ]         // List of the patterns and their associated sequences
}
```

- **doctype**: identify the representation language (here GDF) and the version used.
- **date**: date of creation of the file. This is useful to keep track of the initial creation date when the data are copied on different support. We follow a subset of norm ISO 8601.
- **dataset**: official name (ID) of the dataset.
- **source**: information about the dataset and identifier (URL, ISBN, …) allowing to get more information
- **authors**: names of people who collected and/or prepared the dataset
- **learningSet**: List of the learning set (in the current format) that have been used to generate this file. Of course this information only makes sense when this data set is the results of a workflow. For instance in IKATS, HARP tool (pattern generator) reads one or several LS and produces a new LS containing the patterns that have been generated. Thus, the reading procedure of the learning tool could be (optionally) able to read recursively these learning sets while reading the current one, the "full" information being providing by the merge of all the files (see section 7).

4.1 Variables section

Variables are corresponding to the *classical features* that are used in Machine Learning to describe a simple quantitative, ordinal or quantitative value (i.e. not Time Series, sequences or patterns).

```
variables: [
        {*name:         [var1, var2, ...],
         *type:         'number|boolean|timestamp|string|enumerate|...',
         *domain:       'integer|float|...'
         range:         [ ...],
         units:         'm',
         relevance:     2.0             // the relevance is expressed as a float, default value is 1.
0
         confidence:    0.8             // perceived or real quality of the information
         comment:       'other information about these variables',
         meta:          [['class'] ['id'] ['ignore'] ...]
        }, ...]
```

- **name**: variables names. Here we can define the characteristics of several variables at the same time when all of these variables share the exactly same definition. This is just a convenient way tog gather similar information.
- **type**: allows to declare the GDF type of the values stored in the variable (see hereunder table)
- **domain**: allows to declare the way values are represented in the computer. For instance, with GDF type *number*, values can be represented by an *integer* or a *float* (other possibilities could exist such as: complex, probability, …). This property is mandatory with **with three exceptions** for GDF types : Boolean, String and Picture whose domain is "self defined".
- **range**: allows optionally to express the possible/relevant values of the type according to its domain. When the range is missing any value compatible with the domain are acceptable. The range is mandatory only for the following GDF types: modular, ordered, hierarchy and enumerate. Beyond the semantical aspect, knowing the range a value is relevant to control de input data and in some ML processes for instance to normalize a similarity.

Properties *type*, *domain* and *range* are, in most of the cases, correlated and allow GDF to represent the three main families of values whose semantic is classical in Data Analysis, that is to say:

- **Quantitative values**: these values allow to measure or to count some quantities. In GTD, types *number* and *modular* belong to this family. Clearly many other types (or domain) could be defined such as interval or normal distribution to represent data expressing some variability but these representation are more used to express set of data rather individuals.
  - **Number** type can be either *float* or *integer* or a restricted interval of these sets. With floating number it is moreover possible to declare the *precision* (i.e number of decimal) of the value.
  - **Modular** type is a reference to the *modular arithmetic* in which values are represented modulo a given constant. These kind of data are very classical, the most classical examples being the way time is handled modulo 24 (the value following 23h being again 0h) or the measure of an angle in degree which is done module 360. This type of data must be treated in a specific way: for instance, with a learning tool using a distance measure on an angle, distance (360, 0) is smaller that distance (360, 350). Let's notice that *Circular statistics* is a relatively new domain of statistics (~1980) dealing with this kind of data.
- **Ordinal values**: these values allow to rank some information or to express a preference. In GTD, types *date*, *timestamp*, *mark* and *ordered* belong to this family.
  - **Date** type allows to represent a date. However, this type is not intended to represent a timestamp of a time series, this is the role of the next type. We accept two formats widely used : Unix time and iso 8601

    - **Timestamp** type allows to represent any “time mark” of a temporal sequence. The format can be either a the one of a real date or a numerical value indicating any period of time.
    - **Mark** type is a generalization of the notion of “scale” that we already have in the timestamps. It can accept any kind of units (time, distance, temperature, …).
    - **Ordered** type allows to describe a list of numbers or symbols. When the representation use a list of N symbols, their underlying semantic is similar to the one of a list of integer from 0 to N–1. Thus it is for instance possible to compute a numerical distance between two any symbols.
    - **Hierarchy** type allows to represent a hierarchy (tree) of symbol. Thus operation of generalization and specialization are possible and a learning tool can compute a distance, based on the length of the path, between two any symbols.
- **Qualitative values**: these values allow to express some properties or to provide data. In GTD, types *boolean*, *string*, *enumerate* and *picture* belong to this family.
    - **Boolean** type expresses a boolean value, thus domain and range are totally defined .
    - **String** type expresses a string with a possible control on the maximum length.
    - **Enumerate** type allows to describe a set of symbols.
    - **Picture** allows to express a filename containing a picture or photo.

The following table sums up the basic types and their associated domain and range along with some examples.

GDF Type	Possible domain	Example of range	Example of value
number	• integer • float	• *none* • [–3, 16] • [3.14, 18.0]	• 512, 2396.425, 3.45e+12 • –2 • 5.67
modular	• integer • float	• [0, 23] • [–180.0, 180.0]	• 10 • –45.76
date	• unix_time • iso_8601	• *none* • [0, 250000] • [“1 jan 2012”, “31 dec 2012”]	• 125000 • “23 jan 2012 17:56:00”
timestamp	• integer • float • unix_time • iso_8601	• *none* • [0, 4000] • [0.0, 60.0] • [1200000000, 2400000000] • [“1 jan 2010”, “31 dec 2017”]	• 2308 • 23.614 • 1270990000 • “23 jan 2012 17:56:00”
mark	• integer • float	• *none* • [0, 10000] • [–273, 10e9]	• 10, 34.43 • 3 • 4.56e6
ordered	• integer • string	• [5, 6, 7, 8] • [“small”, “medium”, “large”]	• 8 • “medium”
hierarchy	• string	[“size” [“small” [“xs”,“s”,“m”]][large]]	“m”
boolean	N/A	-	true
string	N/A	• *none* • 10 (*string max length*)	• “Once upon a time …” • “Very short”
enumerate	• string	[“tv”, “radio”, “web”, “paper”]	“web”
picture	N/A	-	“face45.jpg”

- **precision** allows to express the precision of the float number (when used) for types: number, modular, timescale and mark.
- **units** allows to express the unit of the variable when this information makes sense. When possible, this information should be coded in order to be compliant with International System of Units notation. For a timestamp, unit is by default the second “s” but can be redefined by the user.

- **relevance**: semantical relevance or preferences (expressed as a numerical weigth) associated to the current variables. This information can be used for visualization purpose (display order of the variable) or during the learning process, for instance: in a distance based learning tool (for instance, in k-NN algorithm) to weight the importance of a variable, or to solve a “draw” between some variables in a decision tree. When missing, *Default value* for relevance is 1.0 (floating number).
- **confidence**: express the “quality” of the variable, that is to say in which extent the information provided by this variable are trustable. The value is a probability in the interval [0.0, 1.0]; when missing, *Default value* for relevance is 1.0 (full trustability).
- **comment**: description of the role of the (set of) variable(s) that can be used in the interface to explain some information about the meaning of a result for instance.
- **meta**: List of meta information indicating the role of the current variables mainly according to the learning tasks. Each meta information is a list [‘keyword’, ‘value1’, ’value2,], however values are optional in many cases. The list of meta information for the variables is given in the hereunder table but this list can be extended in the future. The most generic solution is to consider this field as a list of “open” pairs [keyword values] whose information are used (or not) by the learning tool.

Keyword	Semantic of the meta-information
[CLASS]	Indicates that all the variable(s) in the block express the class to be predicted (supervised learning). Classically, there is only one variable having this metadata but it is possible to declare several times in the LS that a variable is a class when we want to do a multi-target classification. *Let’s notice this keyword is equivalent to “c: class attribute” in Orange format*.
[ID]	This indicates that the content of the variable(s) are used to “name” the current example when reading a data table. If several variables have the ID metatag the name of the example is the concatenation of all these values separated by an underscore. Of course, learning/viz tools are free to internally use or not this information to name the examples. Let notice that declaring a variables as an ID imply that it is not used during the learning (imply IGNORE meta). When no variables are declared with an ID metatag, the loader classically uses the line number of the examples in the dataset has its ID.
[IGNORE <value1>, …]	All variable(s) of the block or just those in the list are not explicitly used to learn. Nevertheless, they are loaded into memory. *Let’s notice this keyword is equivalent to “m:meta attribute” in Orange format*.
[SKIP <value1>, …]	All variable(s) of the block or just those in the list are not loaded into memory. *Let’s notice this keyword is equivalent to “i: ignore attribute” in Orange format*
[STRENGTH]	Indicates that this variable is used to set a weight to the observations (examples) in the data tables. Classically the weight is coded as number but in GDF we make no assomption about its type. This meta information *must appear only once in the profile*. This information can be used by the learning tool to focus the learning process on some observations.*Let’s notice this keyword is equivalent to “w: instance weight” in Orange format*
[PROTOTYPE]	Indicates that this variable is used to indicate that an observation in the data tables is a *prototype*. This notion is relevant if GDF file is the result of a clustering algorithm: a *prototype* is an observation which is a good representative of a cluster (typically the *centroid* of K-means). Classically the prototype property is coded as a boolean but in GDF all possible type can be used. As for the weight this meta information *must appear only once in the profile*.

4.2 Sequences section

This section allows to describe the sequences or TS that appear in the data table. Most of the properties are identical to the ones found in the previous section to describe the variable, but there are two kinds of information indicating the way values of the sequence are *indexed* (timestamps or marks) and *sampled*.

```
sequences: [
        {*name:         [seq1, seq2, ...],
         *type:         'number|boolean|timestamp|string|enumerate|...',
         *domain:       'integer|float|...'
         range:         [ ...],
         units:         'kg',
         *index:        [timestamp_sequence_name|mark_sequence_name]|'integer',
         *sampling:     either a number, 'uneven' or 'unknown',
         relevance:     2.0             // the relevance is expressed as a float, default value is 1.
0
         confidence:    0.5             // perceived or real quality of the information
         comment:       'other information about these sequence',
         meta:          [['ignore'] ['skip'] [inline]]
```

```
}, ...]
```

- **name**: list of sequences names.
- **type**, **domain** and **range**: type of the values composing the sequences values and their range. The list of possible types is the same as for the variables (see section 4.1).
- **units** allows to express the unit of the sequence values (see section 4.1). There is no semantical verification of the correctness of the unit. Thus when the role of the sequence is “derivative” (see below) there is no checking that the unit is in coherence with the derivation done.
- **index** allows to declare the scale associated to the sequences of the current block. **Caution**: a sequence whoe type is declared as *timestamp* or *mark* cannot have an index since THEY ARE the indexes. There are two possibilities for an index:
  1. either the time series use an **implicit numerical** index (row number or column number according to the representation format, see sections 2.2 and 6) to ordered the events: in this case the value is the string “integer” which is a keyword.
  2. or the time series use one or several **explicit index** represented by others sequences (i.e. a list of timestamps or marks) and we need to provide *the name of these sequence* expressing this scale. Let’s notice that these sequences must **have been declared previously** in the profile and the type of their values must be equal to either “timestamp” or “mark”. In the data part all the sequences associated to a sequence of indexes must have a number of values equal or smaller: in other term an index can be longer that the sequences using it but not shorter!
- **sampling** allows to indicates the sampling rate (or more generally the precision) of the sequence when the data are coming from a sensor or repeated measures. **Caution**: this property only concerns sequence of type *timestamp* and *mark* (i.e. sequence that will be used as an index). There are three possibilities of sampling:
  1. The sampling is regular (synchroneous) and we can provide the sampling rate. Let’s notice that the **unit** used to express the sampling rate is the same as for the values.
  2. When the sampling is asynchroneous (for instance if we have recorded user events on a time scale) this field will contains the string ‘uneven’.
  3. Finally, it is possible to indicate that the sampling rate is just ‘unknown’.
- **relevance**: semantical relevance/preference (i.e. weigth) associated to the current variables (see section 4.1 for more details). *Default value* for relevance is 1.0 (floating number).
- **confidence**: express the “quality” of the sequence, that is to say in which extent the information provided by this feature are trustable. (see section 4.1 for more details)
- **comment**: description of the role of the sequence or the « long name » that can be use in the interface to comment a result.
- **meta**: List of meta information indicating the role of the sequences for the learning task. Generally when the option “weight” is used (see above) for a given sequence SEQ, it would be interesting to add automatical the meta information [‘ignore’, ‘SEQ.weight’] to indicate to the learning tool that this information is not to used as a “real” feature during the learning step.

Keyword	Semantic of the meta-information
[IGNORE <value1>, …]	All sequence(s) of the block or just those in the list are not explicitly used to learn. Nevertheless, they are loaded into memory.
[SKIP <value1>, …]	All sequence(s) of the block or just those in the list are not loaded into memory.
[INLINED <value1>, …]	Indicates that when writing a datatable, all sequence(s) of the block (or just those in the list) must be “inlined” into the table (see section 6.1) rather than written in an external file. This option is interesting when the sequence are very short and that the want to be able read it directly. This option is taken into account only when a data file is written (section 7) during the reading process the rules expressed in section 6 are used.
[WEIGHT <sequence>]	Indicates that this sequence is a vector of weights associated to a another *sequence* S allowing to asset an “importance” to each value of S. The lenght of current sequence must be the same than S otherwise a reading error is raised. If a vector of weights is missing (see section 5.1) we assume that this vector is uniform and composed of 1.0.
[DERIVATIVE <Nth> <sequence>]	Indicates that this sequence is a *Nth* derivative of a another *sequence* S. This information can be useful for some learning tools that are dealing with a value and its derivative at the same time (typically the case of HARP in IKATS). There are also two constraints: 1) the *index* (timestamps or marks) and *sampling* information must be the same as S and 2) the lenght of the current sequence must be the same than |S|-*Nth*. If one of these two conditions is false a reading error is raised.

4.3 Pattern section

This section allows to describe the patterns (i.e a small part of discretized TS or sequence) that have been generated with other discovery tools (for instance, HARP in IKATS project) or even generated by hand by the user. From a learning point of view, patterns are very similar to the variables, but *the way they are processed are quite different*:

1) While the value of a variable is directly provided in the data table, a pattern must be "pre-computed" at the runtime to retrieve in the new set of sequences *if* and *where* this pattern occurs and to compute all the information needed to fulfill the *indicators* (see below). Thus, we need to provide all the information allowing this dynamic exploration.
2) From the point of view of GDF, there are two possibilities for a given pattern: either the variables (i.e generated from the indicators) corresponding to this patterns already exist in the data table (i.e the value has been already computed) or the variables are missing but the patterns definition allows to compute these variables when needed.

```
patterns: [
        {*names:        [pat1, pat2, ...],
         *generator:    ['HARP' ['version', 1]],
         type   :       string | [string, [arg1 val1], [arg2, val2] ...]
         *definition:   ['[AB].$C[DE]', '[AB][CD]$.[BC]', ...],
         *sequences:    ['speed', 'angle_dr'],
         slice:         [start_point, end_point],
         indicators:    [exist, count, fst_pos,...],
         breakpoints:   [[-5.6, -4, 0, 2], [-2, 5, 12, 19]],
         vocabulary:    [[], ['freeze','cold','fresh','mild', 'warm']],
         relevance:     2.0             // the relevance is expressed as a float, default value is 1.
0
         comment:       'other information about these patterns',
         meta:          [['class'] ['id'] ['ignore'] ...]
        }, ...]
```

- **name**: pattern names. Let's notice that theses names must correspond (as for the variables and sequences names) to a readible description of these patterns. In other terms, these names must be as explicit as possible about their meaning. As usual we can several patterns at the same time when all of these variables share the exactly characteristics but the definition (i.e the expression describing the pattern).
- **generator**: name of the tool that is able to parse and to execute the patterns with: its version and the (optional) list of parameters used to learn the patterns each argument of this list is a pair [name, value].
- **type**: the *type* of a pattern is an *optional property* helping to interprete the *definition* part. Indeed a generator can be able to generated different kinds of patterns. Of course all these information could be embedded in the *definition* part, but we make them explicit in order to have a less opaque definition of the patterns and this ease to improve the output of the generator with new kind of patterns without having to change the coding of the *definition* part.
- **definition**: this is the "black box" part containing the full description of the patterns using the language used by the generator. Each pattern is described by a string.
- **sequences**: sequences involved in the patterns; a pattern can correspond to one sequence or to code a correlation between two or more sequences. Theses sequences must **have been declared previously** in the profile. This is important since they allows to know the type, domain and range on which the patterns are generated. There are two possibilities:
  - When type equals to 'number', 'modular', 'timestamp' or 'mark', the values can be discretized and then the properties **breakpoints** (and optionally **vocabulary** allowing to rewrite the pattern in a more readable way) are necessary to the discretization processes.
  - When type equals to 'boolean', 'string', 'enumerate', 'ordered' or 'hierarchical', the values are already discretized (by definition) and properties **breakpoints** is useless, the same for **vocabulary** assuming the initial coding make sense. A warning must be generated by the reading procedure if they are provided in the description.

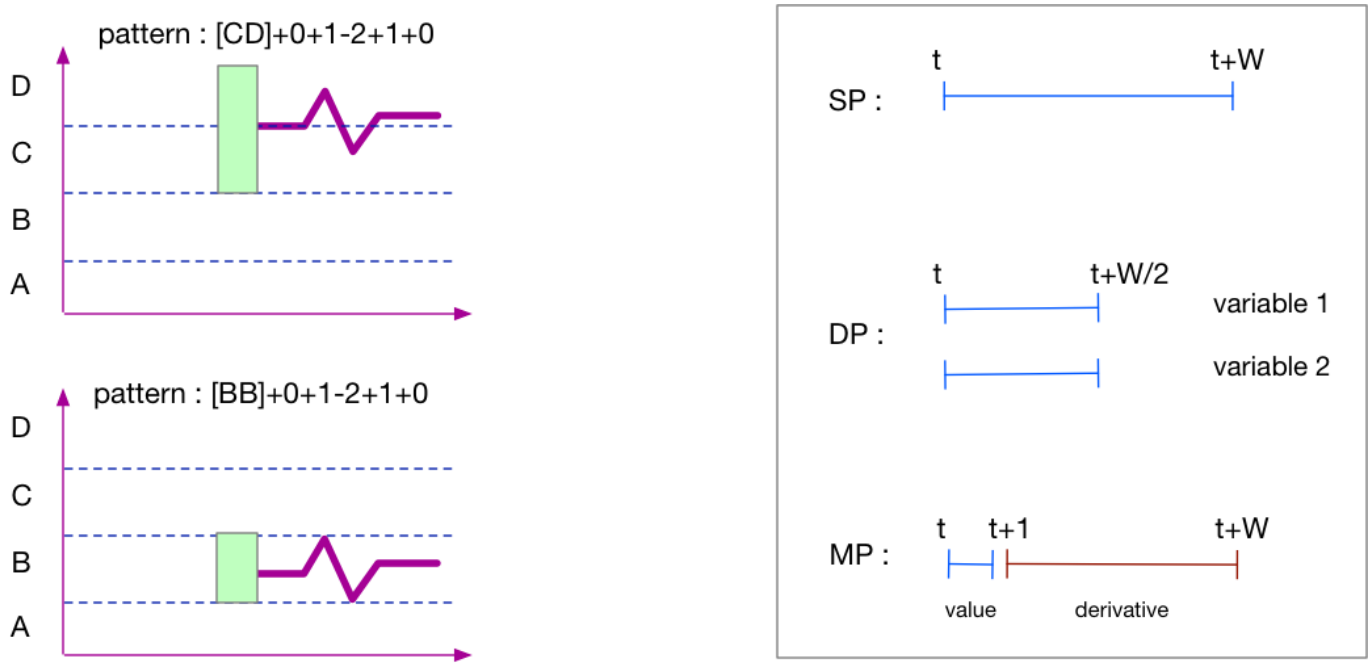
In order to illustrate how patterns are coded in GDF, here are some examples of patterns produced with HARP generator. In this system pattern definition are close (but not identical) to a simplified version of Regex. A pattern can concern one sequence (SP), 2 sequences (DP) or a sequence and its derivative (MP) ; additional arguments can also exist as a "phase shift" between 2 sequences.

- *simple pattern on one sequence and vocabulary*
  - *names: [pat1],*
  - *generator: ['HARP' ['version', 1]],*

- *type : 'SP'*
- *definition: ['[AB].D[EF]]'],*
- *sequences: ['temperature'],*
- *breakpoints: [[–5.6, –4, 0, 10]],*
- *vocabulary: [['freeze','cold','fresh','mild', 'warm']],*

- *pattern on two sequences with a shift*
  - *names: [pat1, pat2, …],*
  - *generator: ['HARP' ['version', 1]],*
  - *type : 'SP'*
  - *definition: ['[AB].#C[DE]', '[AB][CD]# .[BC]', …],*
  - *sequences: ['speed', 'angle_dr'],*
  - *breakpoints: [[–5.6, –4, 0, 2], [–2, 5, 12, 19]],*

- *sequence with a nominal value (already discretized)*
  - *names: [pat1, pat2, …],*
  - *generator: ['HARP' ['version', 1]],*
  - *type : 'SP'*
  - *definition: ['[AB].D[EF]]', '[AB][CD]# .[BC]', …],*
  - *sequences: ['speed', 'angle_dr'],*
  - *breakpoints: [[–5.6, –4, 0, 2], [–2, 5, 12, 19]],*
  - *vocabulary: [[], ['freeze','cold','fresh','mild', 'warm']],*



Examples of patterns according to the type parameter

```
patterns: [
        {*names:        [pat1, pat2, ...],
         *generator:    ['HARP' ['version', 1]],
         type   :       string | [string, [arg1 val1], [arg2, val2] ...]
         *definition:   ['[AB].$C[DE]', '[AB][CD]$.[BC]', ...],
         *sequences:    ['speed', 'angle_dr'],
         slice:         [start_point, end_point],
         indicators:    [exist, count, fst_pos,...],
         breakpoints:   [[-5.6, -4, 0, 2], [-2, 5, 12, 19]],
         vocabulary:    [[], ['freeze','cold','fresh','mild', 'warm']],
         relevance:     2.0             // the relevance is expressed as a float, default value is 1.
0
         comment:       'other information about these patterns',
         meta:          [['class'] ['id'] ['ignore'] ...]
        }, ...]
```

- **slice**: indicates that the patterns concern only a subpart of the sequences. The starting and ending point are expressed in the units of the first sequence (assuming all the sequences are aligned). If this property is missing the all values of the sequence are taken into account and explored to find the patterns.
- **indicators**: when HARP generates a pattern it is also able to create a set of new variables about it: for instance the number of occurrences of the pattern, its average position, the gap between two occurrences, etc. To avoid to have to declare in the profile part all of these descriptors, we just provide here a list of these "indicators". For instance, if in a block containing the patterns ['P1, 'P2] we declare the indicators ['exist', 'min_gap'], this means that two descriptors will exists in the data table for all the patterns: 'P1.exist' 'P1.min_gap', 'P2.exist' and 'P2.min_gap'.
  There is also a very special "indicators" named "pos" coding for a sequence and not a value. It contains all the occurrences (i.e. positions) of the current patterns in the observations with two possible format:
  - index1, index2, … when the pattern have a constant size (default case). When the sequence has more than one index only the first one is used in this list.
  - [index1, length1], [index2, length2], … where length is the length of the pattern. This information makes sense when the size of the pattern is not a constant, for instance with a regex as "B[A]+C" whose length depends on the number of letter "A" found in the sequence.

Important: to avoid to have a separate file for each observation it seem logical to use the format "row based representation" (see section 3.2) to store this information !

Here is the list of current indicators for HARP and their types (this list will be possibly extended/modified !). When property *indicators* is missing of the pattern definition, column 'exist' is automatically generated in the data table.

Indicators	domain	Semantic
exist	boolean	indicates if the pattern exist in a sequence.
count	integer	number of occurrences.
fst_pos	index	position of the first occurrence.
lst_pos	index	position of the last occurrence.
avg_pos	index	average position.
min_gap	index	smaller interval between two consecutive occurrences.
max_gap	index	larger interval between two consecutive occurrences.
avg_gap	index	average interval between two consecutive occurrences.
pos	sequence of positions	list of the occurrences of the patterns.

- **breakpoint**: List of breakpoints associated to each sequence allowing to discretized the numerical values (not needed when 1) a sequence is made of nominal values and/or 2) the generator is not based on a discretization of the values). There are 2 important things to notice. Firstly, in GDF each set of patterns has its own list of breakpoints, meaning that for a same seqeunces several discretizations are possible. Secondly, the first and last values of a list of interval in the breakpoints are "open" and only bounded by the **range** of the sequence. For instance the list of breakpoints [–2, 0, 2]:
  - **means** : there are four intervals [min_range, –2[; [–2, 0[; [0, 2[; [2, max_range,]
  - **doesn't mean** : there are two intervals [–2, 0[; [0, 2]
- **vocabulary** is used to express the discretized values in a more readable format using all possibilities of Unicode. The number of items in the vocabulary must equal to the *number of breakpoints plus one* since the breakpoints are correspond to the treshold and the vocabulary to the intervals! By default, if this field is missing the generator uses its default representation (e.g. alphabet letters {A, B, C, …} for HARP), or the real values in the case of discrete values (ordered, string, …).
- **relevance**: semantical relevance (i.e. weigth) associated to the current pattern. This value is inherited by all the indicators derivate from the pattern. *Default value* for relevance is 1.0 (floating number).
- **comment**: description of the role of the pattern or the « long name » that can be use in the interface to comment a result.
- **meta**: List of meta information indicating the role of the patterns in the learning task. Generally when the indicator "pos" is used (see above) for a given pattern P1, it would be interesting to add automatical the meta information ['SKIP', 'P1_pos'] to indicate to the learning tool that this information is (in most of the cases) not to used during the learning step.

Keyword	Semantic of the meta-information
[IGNORE, <value1>, …]	All pattern(s) of the block or just those in the list are not explicitly used to learn. Nevertheless, they are loaded into memory. *Let's notice this keyword is equivalent to "META" in Orange format*.
[SKIP, <value1>, …]	All pattern(s) of the block or just those in the list are not loaded into memory.

5. Description of the data table

The data table contains a set of pairs feature/value allowing to describe the examples. In GDF we accept two different formats for this data table: GDF table and DAF table (see section 2.2). Both of them are based on a classical CSV coding allowing to use the classical "delimiters" (e.g: comma, tab, …).

In both table formats the **header line** of the file contains the name of the columns to improve readability of the table. Moreover, in this way, the order of the features/column can be different from the declaration order in the profile but of course feature names must be identical and an error must be raised by the parser if not.

5.1 GDF table

This format, which is the default data format for GDF, is classicaly used by most the machine learning tool to describe the dataset. In this kind of table:

- each line is an example/observation
- each column is the value of a feature (variable/sequence/pattern).

In the case of the sequences, this value is a "pointer" on the file containing the set of values in the case of a TS or sequence (see section 6 for more details). When the format is used along with a database (as in IKATS project for instance), this pointer is an identifier to the corresponding item in the database.

5.2 DAF table

As we saw in section 2.2 another format is widely used in Data Analysis in which all data, including sequences are stored in the same file. In this kind of table:

- each example/observation are stored using several lines, the number of lines for each observation being equal to the length of its longuest sequence.
- as in GDF format each column contains the values of a feature (variable/sequence/pattern). When the features is a variable or a pattern the first ligne of the example contains its value, the other lines can either repeat the same value or being just empty. If a sequence is shorter than the other the lines after the end of this sequence are empty for the corresponding column.

5.3 The case of special values (missing, unrelevant, …)

Independently from the way the tables are stored (GDF or DAF) there are often some values in the dataset that are either:

- missing or too noisy (named *unknown values*, generally coded « ? ») or
- without any meaning (named *don't care values*, generally coded « * »), in this case it means that the corresponding value is unrelevant for the current observation.

These "special values" can occur in any kind of variable (standard values or part of sequences) and GDF **must be able** to represent them in the files and within memory. A good example of how these values can "appear" is given by the *patterns* part of the profile. For instance, let's imagine that a pattern "A[B]BC" is relevant for a class C1 but that it is not (or rarely) occurring in the other classes. This means that the values of the position/interval based *indicators* of

this pattern (e.g. “fst_pos” or “min_gap”) can be “undefined”, for the observations not belonging to class C1. By coding the information with any other values than *don’t care* (for instance fst_pos=0, min_gap=–1) we would just transmit false information to the learning tools and thus could leading them to take a bad decision.

From the learning point of view, the way to deal with them is very dependent of the learning algorithm (but they **don’t concern** the reading process). There are some classical strategies, for instance:

- **Unknown values**: to approximate the value by the average value in the database or by the average value of the class to which belong the example in supervised learning, or by the average value of the current node in a decision tree method.
- **Don’t care values**: to “neutralize” the value by considering all the possible values (but this method is only possible when the feature is a nominal type) or to consider that the corresponding pair observation/value doesn’t provide any information.

6. Links to the external Sequences

As we saw in the previous section, when the file describing data table use the GDF format, values corresponding to sequences are generally (see exception herunder) “pointers”, namely a directory and a filename, to the external CSV files containing these data. As we saw (details in section 2.2), there are two “classical” ways to describe the TS or sequences, to summarize:

- **Row format**: each line describes the TS/sequence associated to a given feature, each file describing one or several observations.
- **Column format**: each column describes the sequence associated to one or several features (thus, number of lines of a file = number of timestamps of the current observation). In this format, when there are several columns, the first line is an header indicating the name of the feature associated to each column.

6.1 The specific case of inlined sequences

Sometime a sequence can be very short containing just 2, 3 items. In this case it is interesting to allows the possibility to “inline” this sequence directly in the data table thus avoiding to create some files. This feature is only possible when the sequence use an *implicit index* (integer). The format for this data is a string beginning with the two characters “@[”. For instance: “@[item 1, item 2]”, in such case case the different values are separated with a comma.

6.2 Automatic file format detection

As the files describing sequences can come from different “sources” and to avoid the user to have to do a lot of preprocessing, it would be interesting to implement an **intelligent reading process** that would be able to automatically “detect” the different kind of formats (row/column, with/without header, etc) and to process the files containing sequences in a relevant way. We are going to consider all classical situations and look at the way to deal with them.

First, in **row format**, for a given feature (i.e. sequence), a file often described more than one observation, meaning that in the data table, the name of this file will be repeated several times in a same column (feature). That is a problem since we need to assume that the line number describing an example « n » is coherent in both files (i.e. data table and external file). However, as soon as the user slightly modify the data table (for instance by sorting observations), this “order based” link will be silently broken. Thus, to have a more robust way to associated the right sequence to the right observation it is important to create an explicit link between the two pieces of information. We propose two different methods:

1. to associate a line number to the name of each file to identify the right sequence. This method has the advantage to need no extra-information.
2. to associate a name (i.e string) to the of each file , assuming that in this file each line is indexed by a name placed on the first column. This second method is more robust.

Second, in **column format**, things are simpler if we assume that the first line always contains an header (i.e the list of feature(s)) as soon as there is more than one column. In the later case we could accept that the header is present or missing.

The next table summarize all possible cases (according to the content of the file in terms of number of lines, columns and presence of a header) when one read an external sequence. It describe how the file "X" must be described in the data table and the action to do with the possible failures. Let's notice that the reading procedure should be able to remove the empty lines or empty columns in the files to be more robust and to avoid raising useless errors.

#Row, #Col, Header	External file content	Data table syntax	Action …
1, N, no	one sequence using row format	"X"	Read the sequence of length N
N, 1, no	one sequence using column format without any header	"X"	Read the sequence of length N
N, 1, yes	one sequence using column format with an header	"X"	Read the sequence of length N–1 and verify that the header name corresponds to the current column of the data table
N, P, no	N sequences with row format	"X:<nnn>" or "X: <name>"	Read the sequence <nnn> or <name>, an error is raised if it doesn't exist
N, P, yes	N sequences with column format	"X"	Read the sequence in the column name corresponding to current column of the data table, an error is raised if it doesn't exist

Finally, once the file format has been determined for a given sequence, the reader process interprets the sequence according to **index** property (see section 4.2) indicating the way to associate properly the index(es) with the values of the sequence. An error is raised when:

- A sequence use an *explicit index* and that the length of the two sequences (index and values) are not equal.
- A feature corresponding to the same index appears into several files and that the different occurrencies are different.

Section 9 provides an exemple of some of the different ways to represent the sequences and their associated indexes.

7. Reading and exporting data format

To ease diffusion of GDF, it is very important to provide users with a "translation" tool able to **read** and **write** several classical data format to simplify the convertion into the format describe in this document. The most interesting targets are:

- UCR format (TS data bases)
- Orange format
- Weka (ARFF) format
- Amazon ML format
- DAF representation (see section 2.2) is widely used in Data Analysis. In this format all data are stored in one file: columns being the features and examples being organize into block of N consecutive lines. This format leads to some odd characteristics complicating a little bit the parsing. For instance as all features have the same number of lines, this leads to have some cells with a repeated or null value in the case of classical variables.
- A **spreadsheet based representation** to help edition of the profile (see annex D). Such format is very interesting since in many scientifical or industrial domain, users are storing/modifying their data using this format (even if the adequacy of this approach is very questionable).

Of course when exporting GDF file into another format, some features are impossible to express. Translation process must be able **to keep as much information as possible**, for instance missing types will be translated into their closest equivalent. Another problem is that in some data format there are "implicit" information that we want to becomes explicit (i.e described as a feature) in our format. For instance, if observations are named in the source file but that this name is not associated to a variable we need to generate a specific variable with two contraints to respect:

- we need to avoid any naming conflict with the other features (one naming space)
- this features must be easily identified when we export to another format to avoid to generate a feature that doesn't previously exist (e.g when we import then export to GDF a data format, the exported file must be semantically identical to the initial one !)

So we use the following convention: features that has been generated due to the conversion process begin with the character "@". For instance "@ID" is a name automatically generated, "@class" the feature containing the class of the observation, etc.

Canonical data format for GDF

As we saw in section 6 the sequences stored in different files can have different formats (row/column, with/without header, etc). If the reader process is able to deal with all of them, it is also important to be able, when we export a learning set to GDF to "normalize" the representation of the sequences and to provide the most interesting structure according to the number of observations/features. Thus, two main options must be implemented (corresponding 4 possibilities):

- selection between observations based or features based organization of the files.
- Selection between mono or multi-sequences files.

Here is a short description of files organization for each case:

-	Observations based	Features based
One sequence/file	There is only one folder per observation, each folder containing as many files than the number of features (excluding indexes); we use column format to store the values and the n indexes sequences (generally n=1) are provided inside the same file leading to a n-columns representation (with n–1 indexes).	There is one folder per feature (excluding indexes), each folder contains as many files than the number of observations, we use column format to store the values and the indexes are provided inside the files leading to a n-columns representation (with n–1 indexes).
Many sequences/file	The sequences associated to an observation are described into one or several files according to the number of different index sequences that exist (timestamp or mark). Thus, each file contains values of one index and as many columns as the number of features using this index.	There is one file per variables, each file contains as many lines than the number of observations. These observations are named using their current ID (i.e. and thus we use of "X:<name>" format in data table) and explicit indexes are stored as standard variables in other files. This representation is mainly interesting when index are implicit (position based).

8. Merging data files

When we describe a Learning Set, it is sometime useful (or even necessary) to split it into several parts, that typically the case when a tool as HARP "complete" an initial LS by adding some patterns. Thus the LS readers must be able to deal with these situations. In practice we have two cases:

- The LS files shared the same profile (i.e. sames features) and the collection of data are just corresponding to different set of examples. In this case, the resulting data set is just the a concatenation of the data part (i.e merging lines in the data table).

- The examples are described using several profile, each profile corresponding to a subset of the descriptors. In such case the different definition of descriptors can be “concatenated” (i.e merging columns in the data table) but with some cautions to be able to export a correct format:
  - Descriptors of two different profiles must have different names to avoid naing conflict.
  - The resulting header is the header of the first profile (assuming that the list of LS is ordered) and the other part of the profiles are merged together.

9. Example of GDF representation

Here we provide a small example illustrating some of the possibilities of sequence descriptions. The profile (see section 4) contains the following declarations:

- OBS is the name of the observation
- V1 is an ordered variable with range [a, b, c, d]
- T1 and T2 are correspond to explicit timestamps
- S1, S2, S3 are sequences of integer, S1 is indexed by T1 and S2, S3 by T2
- P1 is a pattern defined as “A[BC]” with variants *count* and *min_gap*

This will correspond to the following JSON code:


```
{
  "header": {
    "doctype": ["GDF", 0.32],
    "date": "18-02-14 22:10",
    "dataset": "Data example",
    "source": "IKATS project",
    "authors": "G. Bisson",
    "learningSet": []
  },
  "dictionary": {
    "variables": [
      {"name": "OBS",
        "type": "string",
        "comment": "name of the observations",
        "meta": [["id"]]
      },
      {"name": "v1",
        "type": "ordered",
        "domain": "string",
        "range": ["a", "b", "c", "d"]
      }
    ],
    "sequences": [
      {"name": ["T1", "T2"],
        "type": "timestamp",
        "domain": "integer",
        "units": "year",
        "sampling": "uneven"
      },
      {"name": "S1",
        "type": "number",
        "domain": "integer",
        "range": [0, 100],
        "units": "m",
        "index": "T1"
      },
      {"name": ["S2", "S3"],
        "type": "number",
        "domain": "float",
```

```
        "precision": 2,
        "units": "cm",
        "index": "T2"
      }
    ],
    "patterns": [
      { "name": "S[ML]",
        "generator": ["HARP", ["version", 1]],
        "definition": "S[ML]",
        "sequences": "S1",
        "indicators": [ "count", "min_gap" ],
        "breakpoints": [10, 25, 65],
        "vocabulary": ["S", "M", "L"]
      }
    ]
  }
}
```

Here is an example of the data table:

OBS	V1	S[ML]_count	S[ML]_min_gap	T1	T2	S1	S2	S3
obs1	b	1	0	T1-ob1.csv	data1.csv	S1.csv:00	data1.csv	data1.csv
obs2	c	4	3	T1-ob2.csv	data2.csv	S1.csv:01	data2.csv	data2.csv
obs3	c	2	6	T1-ob3.csv	data3.csv	S1.csv:02	data3.csv	data3.csv
obs4	a	2	1	T1-ob4.csv	data4.csv	S1.csv:03	data4.csv	data4.csv

As we can see features T1 and S1 are stored into two separate using a row format, while T2, S2 and S3 are stored in the same files using the column format. Here is an example of the content of this files:

For the four files corresponding to T1:

File	Values
T1-ob1.csv	23624, 23627, 23700, 23745, 23802
T1-ob2.csv	4567, 4678, 5000, 5005,
T1-ob3.csv	45890, 45895, 46700, 48904, 48950, 48960
T1-ob4.csv	0, 8, 14, 25, 102, 206, 424, 425

For sequence S1 there is only one file using row format. The number of values on each line must be coherent with content of the files describing the sequence T1.

S1.csv file
45, 67, 12, 0, 14
100, 98, 94, 80
56, 58, 78, 96, 100, 88
67, 56, 45, 54, 78, 100, 98, 86

For data1.csv using the column format. Files data2, data3 and data4 have a similar structure.

Data1.csv

T2	S2	S3
678	45.40	−12.64
702	56.67	−10.00
804	57.43	−9.24
856	65.08	−11.56

Annex A : Composite types.

In section 4.1 we described the GDF basic types. This list could be extended, for instance to include complex numbers, probabilities, … However, in many domains some values are composed of several basic arguments. For instance, to store a geographical position we can use 2D or 3D coordinates (corresponding to 2 and 3-uplets) ; we need the same thing to store some statistical information (mean and standard deviation, intervals, …); etc. Of course, each value could be stored independently in several variables of the data table but by doing that we lose the meaning of the data and a program that will analyze them have to “rebuilt” the complete information.

Thus the idea would be to add a new block in the dictionary allowing to declare *composite types* that is to say a type of data which is a n-uplet composed of the GDF basic types. Here, we make no assumption on the way the ML tool will use these data and it will be probably interesting to introduce some semantical information to express how these data must be processed (for instance: providing a similary function).

Here is an example of declaration in the Dictionnary block. In the other part (variable, etc) the user will just use the new type name without any possibility to specialized the definition.

```
type: [
   {name: "coordinate",                 // new composite type
    type: ["modular", "modular", "number"],
    domain: ["float", "float", "float"]
    range: [[-180.0, +180.0], [-180.0, +180.0], []],
    units: ["degree", "degree", "m"]
    comment: "Longitude, latitude and altitude"
    },
    { ... }
    ]
variables: [
      {name: "object-pos",
       type: "coordinate",              // we use the composite type as previously defined
       comment: "X, Y, Z observations",
```

Annex B : Relational information.

The tabular representation widely used in ML (see section 2.1) is simple but it offers no way to easily express relational information (i.e: datasets in which parts of the observations contain some references to one or several other observations). Here the idea is to propose a simple extension of GDF to deal with this problem.

- Benefit: allow to represent relational information in a dataset
- Basic idea: the profile can contain several dictionaries with some link between them
  - Each dictionary has a name (default name is “data”)
  - Each dictionary **has its own table**
  - A dictionnary can be seen as a new domain of the type “relation”.
  - The type of a variable or a sequence could be the name of one (or several) dictionary. In this case that means that we have a pointer on an element (observation) of another table. In the next example, a person can have pointers on a *company* table, a *town* table and even on his own table.

Example of declaration of a relational structure

```
dictionary:[
   person:{                              // name of the file containing the table
          variables:[
                { name: "firstname",
                  type: "string"},
                { name: "employer",
                  type: "relation",     // relational field
                  domain: "company"},   // value is the ID on an observation in table "company.csv"
                { name: "adress",
                  type: "relation",     // relational field
                  domain: "town"}       // value is the ID on an observation in table "town.csv"

                ]
          sequences:[
                { name: "children"
                  type: "relation",     // relational field
                  domain: "person"}     // values of this sequence are ID of observations in "person.
csv"                "
                ]
          patterns: [ ... ]
        },
   company {...},                        // dictionnary of company
   town {...},                           // dictionnary of town
   ]
```

Annex C : Representation of hierarchical clustering results.

The output of hierarchical clustering is a binary tree whose nodes are corresponding to a cluster. Each cluster gathering either two others clusters, or one cluster and one observation, or two observations. Thus the output contain some relationnal information. Here, the idea is allow GDF to natively represent such kind of relational structure by using two dictionaries (see Annexe B):

- observations: the initial observations used to built the hierarchy
- clusters: the hierarchy, each nodes being define with either an observation or another cluster.

Example of declaration of such dataset.

```
dictionary:[
   instances:{                          // name of the file containing the instances
          variables:[
                { name: "firstname",
                  type: "string"},
                { name: "employer",
                  type: "relation",
                  domain: "company"},
                ...
                ],
        }
   clusters: {                          // name of the file containing the clusters
          variables:[
                { name: "left-node"
                  type: "relation"
                  domain: ["instances", "clusters"]},   // left node: ID of an instance or a cluster
                { name: "right-node"
                  type: "relation"
                  domain: ["instances", "clusters"]}    // right node: ID of an instance or a cluster
                ...
                ]
        },
   company {...},
   ]
```

Annex D : Flat representation of the profile.

Using JSON format (or even a more simple one as YAML) it is clearly quite impossible to write the “profile” of a dataset with a text editor even if it is doted with a JSON mode. Thus we need to provide to the user a more simpler way to write it. There are two possibilities:

- the user can write a very first version using simple format (orange, weka) and then import the file. The main drawback of this approach being that most of GDF features will be ignored and a lot of work is required to complete the description of the data.
- providing another more user friendly external format for GDF. In practice, GDF format can be easily turn into a tabular format that can be edited with a very classical spreadsheet tool. The advantages of this approach are multiple: spreadsheet is a widespread tool, it is well-know by ton of users and, all in all, it is well-adapted to write semi-structured data.

The target format to use for this file can be either :

- xlsx: native format of Microsoft Excel. Some libraries exists (e.g. pyexcel-xls) in python to read/write this format.
- odf: native format of Apache Libre Office. Here too libraries exists (e.g. pyexcel-ezodf) in python to read/write this format.

The structure of the flat external format is this one:

- The profile file is composed of several sheets:
  - a first sheet describing the header of the profile
  - a sheet per dictionnary and category of data among: variables, sequences and patterns. For instance, if the dictionary is named “data” we can have the three following sheets. Of course a sheet can de removed if there is no feature belonging to this category.
    - data.var : to define the variables (one line per variables)
    - data.seq : to define the sequences (one line per sequences)
    - data.pat : to define the patterns (one line per patterns)
  - optionally the file could also contains the table associated to each dictionary (for instance data.csv), in this way profile definition and main data are at the same place. Here there is two cases (see section 5):
    - GDF table: the files containing the sequence values must not appear here.
    - DAF table: all data (including sequences) are stored in the table.

Furthermore with this organization it would be relativelly easy to provide some “template” containing macros allowing to simplify creation of the profile and to avoid some basic errors (wrong type name, …)

Annex E : Change log.

Changes done with respect to version 0.42 Description	Sections concerned
Update and complete introduction part and the State of the art of Time Series representations.	1, 2.2, 7
Introduce a new type “*date*” in order to avoid confusion with the type “*timestamp*” whose use is reserved for sequences only.	4.1
The type “*timestamp*” now accept to have some *range* restriction.	4.1
The property “*inlined*” for the sequences has been transformed into a meta information since it doesn’t concern the semantic of the sequences.	4.2
The property “*weighted*” for the sequences has been transformed into a *meta information* since it mainly concern the learning process.	4.2
A new meta information “*derivative*” for the sequences has been added. It allows to indicate that a sequence is the n-th derivative of another one. This information mainly concern the learning process.	4.2
Improve the description of the need to handle correctly “unknown” and “don’t care” values in GDF format.	5.3
Modification to deal with the multi-lines data format (DAF) often used in Data analysis.	5, 7
Modification in section 6 and 7 in order to solve some ambiguities in the descriptions.	6, 7
Correction of many typos as usual.	

Changes done with respect to version 0.40 Description	Sections concerned
A new meta-keyword *prototype* has been added to help coding of the clustering results.	4.1
The definition of variables and sequences has been modified. Now we have 3 properties: *type*, *domain* and *range* instead of 2. *Domain* expresses the way the type in represented and *range* the possible values or a domain restriction.	4.1, 4.2
The meta-keyword *weight* has been renamed *strength*.	4.1
The meta-keyword *relevance* has been turned into a *property* in the dictionnary since it is more related to the semantic of the feature (is this information important or not) than just a meta-information.	4.1, 4.2, 4.3
More precisions have been added concerning the meta-data for the sequences and the patterns.	4.2, 4.3
The property *weight* has been added to the sequences to indicate when a sequence has an associated vector of weights.	4.2
The property *inlined* has been added to the sequence allowing to directly store a sequence within a data table.	4.2
Content of the annexes has been extended to provide further details about the possible improvements.	A, B, C, D
Correction of many typos as usual.	

Changes done with respect to version 0.32	
Description	**Sections concerned**
Section 1 has been rewritten in order to provide a better outlook of the goals and interests of this work.	1
New section 2.3 discussing how to generalize the time series representation to deal with any kind of sequences.	2.3
Property “LSformatVersion” has been replaced by a more general “doctype” in the header.	4
Property “Authors” has been added to the header.	4
Major changes has been done in the list of types handled by the representation language. These modifications have two goal: first becoming closer to the classical types used in data analysis and second, to extend and generalize the representational possibilities.	4.1, 4.2, 4.3
Notion of sequence has been generalized and now values can be indexed by any kind of ordered values and not just timestamps.	4.2
Correction of the definition of “vocabulary” in patterns description section which was not coherent with the way breakpoints are defined.	4.3
Property “variants” in patterns description has been renamed “indicators” which is more appropriate.	4.3
Property “length” in patterns description has been removed, this information is very dependant of the description language used to represent the patterns. If a tool is able to interpret this language it is also able to compute the size.	4.3
Several annexes has been added to describe some possible improvements to GDF.	A, …
Correction of many typos as usual.	