Chapter 1

Heaven

In this chapter we present $\mathcal{H}$eaven, an open source framework for Systematic Comparative Research on RSP Engine. It consists in four baselines and two main components: the Test Stand and the Analyser. The first section, Section 1.1 introduces the Test Stand, which satisfies requirements from R.1 to R.8 and from R.10 to R.12, by executing experiments on an RSP Engine. Section 1.2 describes the Baselines: four RSP Engines that are included in $\mathcal{H}$eaven They fulfil requirements from R.13 to R.14, so are exploitable as naive terms of comparison. In Chapter **??** we use those Baselines to show $\mathcal{H}$eaven potential). Finally Section 1.3 presents the Analyser, which addresses requirements R.9 and R.10 allowing the user to visualise, investigate and compare experiment results.

1.1 Test Stand

Aerospace engineering defines an Engine Test Stand as a facility used to develop, study and characterise engines. It allows to test operating regimes and offers measurement of several variables associated with engine workflow. A Test Stand may uses actuators for attaining a specific engine state, which is a unique combination of the engine properties. The information collected through the sensors depends on the engine manufacturer, which usually provides his own stand or the facilities to test the engine with commercial solutions. The Test Stand executes black box analysis, because usually the engines does not allow easily to interact with its internal mechanisms.

A Test Stand can be described similarly in the SR context. The definition above still holds its relevance with the difference that the tested engines, RSP Engine, are IO-Systems. An RSP Engine consumes an RDF Stream and produces a new one, by applying queries under some entailment regime and w.r.t. some static data (detailed in Section **??**). Describe an RSP Engine means understanding the relation between input stream, registered queries and what we call "operational semantics". Investigate this last element, which is strictly implementation dependent, requires a kind of analysis that cares about the RSP Engine internal process. But, in general, a Test Stand provides only black box testing, even having access to the entire RSP Engine code. Anyhow $\mathcal{H}$eaven Test Stand allows the user (the RSP Engine developer) to add its own sensor to the execution, respecting requirement R.7 and R.10. In this way is possible to develop a specific testing procedure, which consider also the know-how about this particular engine.

1.1.1 Architecture & Workflow

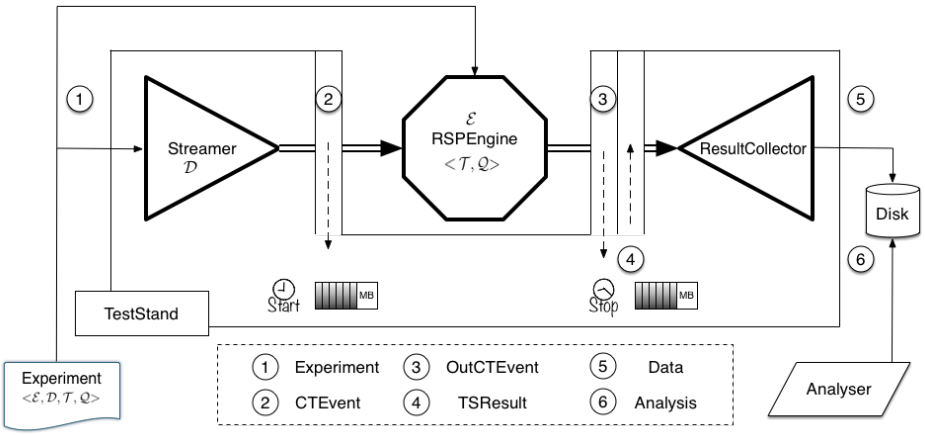

Figure 1.1: $\mathcal{H}$eaven modules and workflow

A test stand exploits different modules to simulate the operating regime for the engine in use (i.e a module for fuel distribution, one for the engine mechanic support or to enable users interaction during the execution). $\mathcal{H}$eaven Test Stand is modular, it consists in stand-alone components that can be replaced with ones with the same interfaces [R.10]. At this moment, three modules compose the Test Stand:

- the Streamer, a source for the input RDF Stream

- the RSP Engine we want to test;
- the Result Collector, a Data acquisition system for both the query results and the gathered measurements.

Figure 1.1 shows both the architecture of $\mathcal{H}$eaven TS, how the modules are configured, and its workflow, how the modules interact.

We start presenting the TS architecture, where the components above are arranged into a pipeline and communicates exchanging events [R.11]. The communication starts from the Streamer, which hides the data generation logic in order to obtain data independence [R.1]. Moreover, it pushes an RDF Stream directly to any mounted RSP Engine, respecting [R.5] by not influence the memory footprint with heavy data loading tasks.

An interface adapts the event flow to the RSP Engine in use, fulfilling [R.2] (Engine Independence) and hiding the query registration process [R.3] (Query independence), which happens at engine level and is up to the RSP Engine provider.

The Result Collector is the tail of the pipeline. It is part of the Test Stand because the performance measurements are processed and gathered during the execution, together with the queries results data. The evaluation usually happens a-posteriori trough the Analyser (Section 1.3). However, real time analysis of the Key Performance Indicator (KPI) are possible, but they may violate some requirements like [R.4 and 5].

Last but not least, the Test Stand has an external structure that sustains other modules. It allows the user to control the process through accessible APIs. This structure can be considered as a module itself, it adds sensor data to query results controlling the execution flow as required by [R.4]. The Test Stand orchestrates the communication between the upstanding models, forcing the Streamer to push events to the RSP Engine and the Result Collector to listen the output an collect the results.

To explain the Test Stand workflow we split the process at the points when the modules exchange events, since each event represent a different logic step in the experiment execution cycle. In this way we distinguish six steps.

In step (1) the Test Stand accepts as input an EXPERIMENT in the form of a tuple $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q}, >$ where:

- $\mathcal{E}$ is the RSP Engine subject of the evaluation (satisfying requirement [R.3]);
- $\mathcal{D}$ is the input dataset [R.1];
- $\mathcal{T}$ is the ontology [R.1];
- $\mathcal{Q}$ is the query to be continuously answered by $\mathcal{E}$ [R.2].

The TEST STAND executes the experiment $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q}, >$ stressing $\mathcal{E}$ for a certain period of time looping through the steps from (2) to (5) illustrated in Figure 1.1.

In step (2), the STREAMER pushes to $\mathcal{E}$ an event CTEVENT. This event is a portion of an RDF Stream picked from the data $\mathcal{D}$ and it consists of a set of RDF triples with the same timestamp. In order to satisfy [R.12], it sends triple in N-Triple[1], which is the easiest RDF serialisation to parse.

In step (3) $\mathcal{E}$ pushes to the RESULT COLLECTOR an event OUTCTEVENT. It contains the current answer to the query $\mathcal{Q}$ registered in $\mathcal{E}$ given the ontology $\mathcal{T}$. The TEST STAND expects $\mathcal{E}$ to output result in N-Triple format.

Notably, to place any RSP engine on the TEST STAND (requirement [R.3]) $\mathcal{H}$eaven provides a simple software wrapper that, when it receives a CTEVENT, adapts it to the RSP engine specific format, pushes it in the RSP engine, and listens to the RSP engine output so to transform such an output in a OUTCTEVENT.

To measure performances (requirement [R.6]) the TEST STAND performs several actions both before step (2) and after step (3). To measure latency, it starts a timer before (2) and stops it after (3). To measure memory load, it asks for the free memory of the system after step (3). In step (4), those observations are added to the outputs of $\mathcal{E}$ as annotations and are pushed to the RESULT COLLECTOR. We name TSRESULT the event that contains the sensor data plus the query results produced by the engine. The TEST STAND works in a single thread mode, blocking the execution of its components when it perform the performance measurements in (2) and (3) [R.4].

[1] `http://www.w3.org/2001/sw/RDFCore/ntriples/`

In step (5) the Result Collector saves TSResult for post processing by the Analyzer satisfying [R.9]. It does so appending TSResults to an open file to fulfil [R.8].

1.2 Baselines

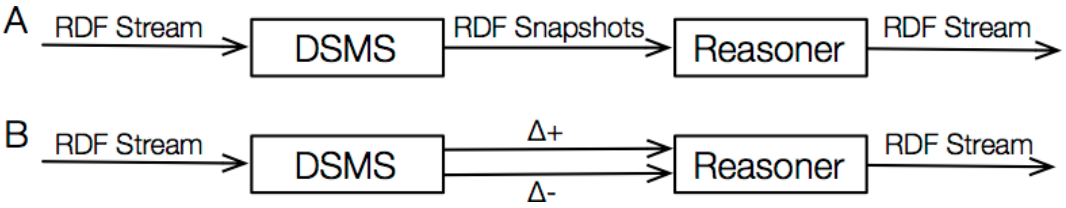

Figure 1.2: A: the architecture of the Naive baselines. B: the one of the Incremental ones.

In Chapter **??** we state that a Systematic Comparative Research Approach needs initial terms of comparison to lead the investigation. $\mathcal{H}$eaven contains a set simple and easy-to-use RSP Engines called "baselines", developed to fulfil this lack . We exploit them to define some qualitative methods of investigation and to prove the usability of the Test Stand. In section **??** we identifies four characteristics that classify a case-study as a baseline inside a research field. Indeed, in this section we present how $\mathcal{H}$eaven Baselines are *Elementary*, *Relevant*, *Simple* and *Eligible*.

As explained in Section **??**, the Stream Reasoning research area has three main building blocks: 1) RDF streams, 2) an extensions of SPARQL to manage continuos data and 3) reasoning algorithms. These there elements are summarised into RSP Engines, systems that can apply reasoning techniques upon rapidly changing information (Section **??**). Early works on SR [**?**, **?**] describe the most simple approach to create a stream reasoning system: pipelining a DSMS with a reasoner.

The first one, the DSMS, is responsible to handle the data stream, moving from infinite sequences to finite (and processable) sets of events. The second one, the reasoner, applies SPARQL queries on this set of events, exploiting its reasoning capabilities over a context that can be considered as static, but remains continuous.

Follow this approach does not require any new design effort. We just need to link two existing technologies and develop the communication between them.

The proposed baselines are *Elementary* because they pipeline Esper[2], a mature and open source DSMS, with the Jena general purpose rule engine[3] that is a flexible reasoning engine. Esper and Jena are two excellences in their context, Baseline *Eligibility* comes form their value. We can consider their couple as a simple but valid term of comparison.

Baselines *Relevance* means to cover all the most important theoretical variants the approach above conveys. In terms of reasoning and with reference to the data stream processing model, four baseline implementations cover two main design decisions about the RDF Stream Model and the Reasoning architecture. The first one, the RDF Stream model, describes how the input RDF Stream is processed, different systems accept data in different models, which depends on how RDF Stream is considered in terms of events contemporaneity. The two most relevant ones are:

- Triple-based model, where the events pushed in the DSMS are timestamped triples. The timestamps are non decreasing, different triples could have the same timestamp to denote that they are contemporary.
- RDF Graphs-based: the event pushed in esper are timestamped RDF graph. The timestamps are increasing and the graph is used as a form of punctuation [**?**] to separate consequent portions of the RDF stream.

The Reasoning Architecture, depends on the techniques to make inference, which are strictly related to the way data can flow from the DSMS to the reasoner. Again the possibility are two:

- Snapshot, where all the window contents is sent to the reasoner at each window slide
- IRStream, where the DSMS outputs the differences between the current window and the previous one

Thus, two possible solutions for reasoning on the triples in the window are possible:

- Naive solution: as shown in Figure 1.2-A, the DSMS determines the triples in the current window, produces an RDF snapshot of it and the

[2] `http://esper.codehaus.org/`

[3] `http://jena.apache.org/documentation/inference/#rules`

reasoner materialise all the implied triples at each cycle. This approach is implemented in the C-SPARQL Engine [?] and in Sparkwave [?].

- Incremental solution: as shown in Figure 1.2-B, the DSMS outputs the IRStream. The Δ^+ snapshot contains the new triples that have just entered in the current window, while the Δ^- snapshot contains the triples that have just exited from the current window. The reasoner, using Δ^+ and Δ^-, incrementally maintains the materialisation over time. This approach is taken as term of comparison in [?] and it is inspired from [?].

The baseline answer the most simple query that involves a big reasoning effort: the identity query, which requires to materialise all the implicit information entailed by the content of the window, given the ontology chosen by the user. We chose ρDF as entailment regime, because it is the minimal meaningful task for a Stream Reasoner. It is the minimal RDF-S fragment which reduce complexity but preserves the normative semantics and the core functionalities [?], several works in the field exploit it [?] and we do it as well to provide Simplicity.

Previous works on correctness of RDF stream processing [?] explain the importance of external control on time to assure that the RSP Engine always outputs the correct answers (even when overloaded). The proposed baselines take advantage of the ability of Esper to be temporally controlled by an external agent[4] by sending time-keeping events to synchronise the internal time flow. One time-keeping event is sent before injecting the triples in a TCEvent and another one after all triples in TCEvent were sent. In this way all the triples in the TCEvent are consider contemporary by the baselines. Prove baselines soundness and completeness is important to sustain their Eligibility, and to allow the user to use them as an oracle for its own engine evaluation.

More consideration on Baseline Eligibility are presented in Chapter of **??**, which deeply answer the questions about baselines performance with experimental results.

[4] `http://esper.sourceforge.net/esper-0.7.5/doc/reference/en/html_single/index.html#api-controlling-time`

1.3 Analyser

The Analyser is conceptually a set of statistical tools that allows visualisation and investigation of the results data. First of all we need to describe the Test Stand outputs. In the related section we states that the TS attaches to the query results the values gathered by the sensors during the execution, which are about memory and latency (or other sensor if any). Then it sends to the Result Collector a complex event with the entire content. We define there main kinds of experiment, distinguishing on the data we want to sample and save. This distinction is relevant from an experimental point of view: indeed, ask the system for the memory usage may influence the latency calculous or saving on disk the query results may influence the memory footprint. From the Analyser point of view we have:

- Latency Experiment, where only the latency is calculated and no query result is saved on file
- Memory Experiment, where only the memory is gathered and no query result is saved on file
- Query Experiment, where query results are saved on file.
- Any combination of the previous one experiment.

In general an experiment produces two output: a CSV file, which contains a tuple with the sensor data and metadata for each event passed during the experiment execution; a set of TriG files that represents the window materialisation at each cycle, even in case of incremental reasoning. (It is also possible to save the non-materialised window, in order to verify the Completeness and Soundness of the reasoning procedure for the baselines). In practice the Test Stand outputs results in time series format and the Analyser toolset has to handle this kind of information. Time series analysis is a very wide matter and design a general architecture for the Analyser is difficult at this point of our research. The main motivation regards the impossibility of a complete decoupling of the analysis form the tool that sustain it, because they depends on the hypothesis of the research.

The relation between the analysis and the experiment is deep, but is possible to identify some common characteristics, which have a general meaning.

The comparative approach we follow (Chapter **??**) can be applied at different levels of analysis. Usually, time series analysis starts with data visualisation: the most common and relevant plotting tools should represents the series in temporal form and show their behaviour in frequency domain. This analysis sustain the comparative approach: a) we can identify patterns by comparing different experiments; b) is possible to understand the relation between variables, by over-plotting many of them in the context of a single experiment; c) we may compare different experiments on the same variable, arguing about the similarity or the difference upon a certain characteristic.

Moreover, we can set the analysis at a deeper level, analysing the data with some statistical investigations. Almost any research needs tools for steady state identification and to calculate the statistical data for the execution: average, standard deviation, maximum and minimum, both at steady state or over all the execution data. Upon this processed data is possible to build simple but meaningful comparisons.

In order to reduce the Byzantine behaviours we need an automatic procedure to repeat the experiment and average the data.

Finally, the time series describe how a dynamic system evolves over time, so is meaningful to formulate hypothesis on behaviour changes. The Analyser has to supports this kind of hypothesis verification trough tools that allow statistical data mining. However, data mining procedures are very system-dependent, to this extent we decide to include an example of the possible analysis in Chapter **??** about the experimental evaluation of the Baselines.