Stream Reasoning Playground

Patrik Schneider[1,2(✉)], Daniel Alvarez-Coello[3,4], Anh Le-Tuan[5], Manh Nguyen-Duc[5], and Danh Le-Phuoc[5]

[1] Vienna University of Technology, Vienna, Austria
[2] Siemens AG Österreich, Vienna, Austria
patrik@kr.tuwien.ac.at
[3] University of Oldenburg, Oldenburg, Germany
[4] BMW Technologies E/E Architecture, Wire Harness, Garching, Germany
[5] Technical University of Berlin, Berlin, Germany

Abstract. Stream Reasoning is a well established field not only in the Semantic Web, but is also adapted in the knowledge representation and reasoning and AI community in general. In the Semantic Web area, there have been valuable efforts in building data generators and benchmarks, however they are not well suited for evaluating more expressive stream reasoning approaches, since the focus is on a graph-based data model and more limited reasoning features, such as query answering. This paper aims at filling the gap, so the different communities can compare, discuss, and benchmark the various approaches for stream reasoning based on a common playground. We will present the *stream reasoning playground* that targets streaming reasoning as the first-class modelling and processing feature. Our playground includes an easy-to-extend platform for data stream generation with pluggable data formatters, whereby different data stream sources, and modelling problems for two interesting application scenarios, i.e., intelligent traffic management and vehicle stream data analytics, are provided. Furthermore, we present a more generic scenario for time-series data, where a workflow for streaming time-series data from various datasets is facilitated by using mapping functions. To illustrate a first application of the playground, we report on the usage experience of well-known stream reasoner developers in the "model and solve" Hackathon event of the annual Stream Reasoning workshop.

1 Introduction

Stream Reasoning (SR) is a well established field not only in the Semantic Web, but also in the AI community in general and focuses on inference, *i.e.,* deducing or inducing implicit facts over data streams. SR has been actively evolving for more than a decade now, and there exists a wide range of approaches to reason over streams [9,21]. Since approaches to stream reasoning can be considerably diverse, it has become desirable to have an agile and well-defined playground accompanied by the corresponding scenarios, datasets, and (output) tooling to compare and test the different approaches by fast cycles of iterative tasks. Notably, the RDF Stream Processing (RSP) community has developed several successful platforms, e.g., TripleWave [22], RSPLab [32], RSP4J [31], and LSBench [18] for benchmarking and comparing different RSP approaches. However, these platforms usually require a graph-based data model, i.e., RDF [17], that can be queried by some extension of SPARQL [25], i.e., C-SPARQL [4] or CQELS [19].

This brings us to the core of the problem, the underlying data model and query language puts already (desired) restrictions on the scope of usage regarding: (a) the expressive power of an approach, (b) the underlying data model and syntax that can be consumed, and (c) the reasoning tasks that could be solved. Hence, RSP-based tools cannot be simply adapted and used for evaluating and benchmarking logic programming- or complex event detection-based approaches. In this paper, we present the SR Playground that is an initiative and underlying *open-source framework* for providing such resources to the SR community, which should eventually lead to a better understanding of the manifolds of (formal) languages, approaches/pipelines, and reasoners. This imposes the following requirements and the derived features to the framework:

– *(F1) Stream reasoning as a first-class use case*, where the prime focus of the playground is the evaluation of a wide range of stream reasoning approaches/pipelines, i.e., RSP-, logic programming-, and complex-event-based approaches (see Sect. 4).
– *(F2) Consumer agnosticism*, where streams for several consumer modelling languages and input formats should be generated, whether the consumer is graph-, rule-, or complex event-based (see Sect. 2).
– *(F3) Extensibility*, where a simple extension with new stream players and data sources is desired, whether the data source can be a simulation tool, web stream, or collections of (preprocessed) datasets (see Sect. 2/3).
– *(F4) Availability and agility*, where the playground should be easy to deploy and fast to update, in case changes or extensions occur, e.g., a syntax change in the generated streams (see Sect. 2).
– *(F5) Base scenarios,* where already challenging scenarios from relevant domains should be given as a starting point (see Sect. 3).
– *(F6) Multiple tasks,* where for each scenario a range of reasoning tasks is given as plain text, which should go beyond query answering, and include the use of a background knowledge base (see Sect. 3).

Alas, the above features cannot be always aligned, since for instance agnosticism makes the process of extensions harder. Taking the above considerations into account and aiming at a well-balanced framework, we present the SR Playground with the following contributions:

– An easy-to-configure and extendable platform for stream generation capable of producing streams of different scenarios based on stream players and pluggable data formatters. The platform is quickly deployable using the playground's Github repository and a Docker-container-based deployment.
– Two well-defined Intelligent Transport Systems (ITS) scenarios, that consist of (a) a traffic-simulation-generated vehicle flow, and (b) a driving trace from the perspective of an ego vehicle's camera moving in a city.

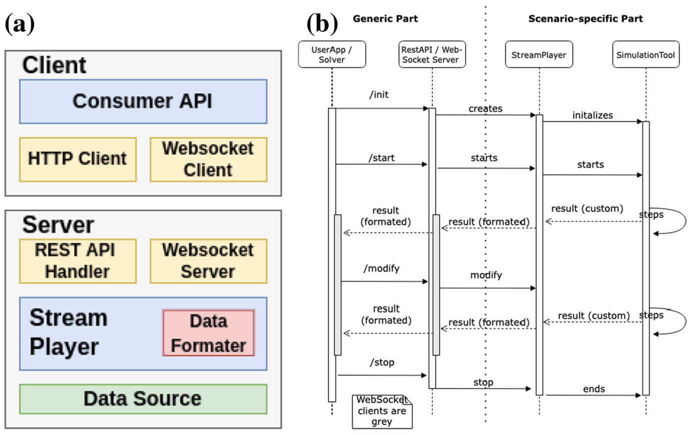


Fig. 1. (a) Overview of architecture and (b) interaction between the components, where the data source is a traffic simulation tool.

- A scenario with a workflow that indicates the steps for streaming time-series data from various datasets that is facilitated by using mapping functions.
- A case study, including the lessons learned from the first usage of the platform in the Stream Reasoning Hackathon 2021, where different *model and solve* tasks were given to participants.

The rest of this paper is organized as follows: Sect. 2 introduces the SR playground. Then, in Sect. 3, we describe two out-of-the-box scenarios that were built with the playground and suggest a workflow to reuse this work with a custom scenario. Then, a case study of a hackathon that tested the presented playground is presented in Sect. 4. Related work is covered in Sect. 5. Lastly, the final remarks are available in Sect. 6.

2 Platform - Stream Reasoning Playground

In this section, we introduce the *Stream Reasoning Playground* (SRP), an infrastructure to stream out semantically annotated data. The SRP's design is based on a client-server architecture that is illustrated in Fig. 1a. The *Server* is a publisher that generates and broadcasts data streams, whereas the *Client* is a data stream consumer that can be piped to the users' Stream Reasoning solver.

On the server-side, the *Stream Player* streams out annotated data streams based on a data source-specific implementation. For example, in the Scenario *A* (Sect. 3.1), the Stream Player embeds a microscopic traffic simulation tool [33] and forwards the simulation tool's states, e.g., vehicle positions and traffic light states. In its simplest form, the Stream Player reads preprocessed datasets and forwards their content to the Data Formatter. Since our architecture is designed to be easily extendable, new scenarios can be added with little effort.

The *Data Formatter* is embedded inside the Stream Player and allows data from the data source to be mapped into several output data formats (*i.e.,* Datalog, RDF, etc.) as the publisher desires. The data streams from the Stream Player are broadcasted via a *Websocket Server* to one or more clients. On the server-side, there is also a *REST API Handler* that allows users (clients, developers, or administrators) to operate and manipulate the behavior of the Stream Player using a given REST API. For example, via the REST API, users can request the background knowledge base (KB) used in their reasoning engine, start/stop streaming, and modify the streaming rate or stream output formats.

On the client-side, users can pipe their reasoner (*e.g.,* a SR solver) to the Playground via the *Consumer API*. This is illustrated in the sequence diagram of Fig. 1b, which describes the data flow of the Stream Playground for Scenario A (Sect. 3.1), where a user can send control commands to the Server as HTTP requests via a *HTTP client*, e.g., a web browser, and receives the data stream from the Server via a *Websocket client*. A user can set up, start, and stop the SR playground via the following commands:

– Initialise a stream player : `/init`
– Start playing a stream : `/start`
– Modify the behavior of a stream: `/modify`
– Stop streaming: `/stop`
– Get the background knowledge base: `/getkb`

At initialization a user can select the scenario and dataset using the `?streamtype` and `?streamid` arguments, as well as the output format using the `?templatetype` argument. All the possible initialization parameters are defined for a scenario in the configurations defined in the `config.yaml` file.

Example 1. HTTP requests to initialize and start of a SUMO traffic stream with JSON-LD as the output format:

– `⟨IP_ADDRESS⟩:⟨PORT⟩/init?streamtype=sumo&streamid=streamSumo1 &templatetype=traffic-json`
– `⟨IP_ADDRESS⟩:⟨PORT⟩/start`

Feature Coverage. Regarding extensibility (F3), we designed the architecture with two layers, where the generic layer is scenario-independent and allows a unified REST API and Websocket server facing the clients. The scenario-dependent part is implemented via stream players and the re-use of data formatters, where a new Stream Player inherits from an abstract player that defines the interfacing. Importantly, the Stream Player acts as a Python generator (introduced in PEP 255)[1] using the `yield` keyword to return stream messages.

Consumer agnosticism (F2) requires that different output formats, given by a modelling language, should be supported regardless of the scenario. We assume that the data sources provided are either relational (as with DB tables and log files) or tree-shaped (as with JSON files). Then, the input tuples are transformed by distinct data formatters, where we support the following types of formatters:

[1] https://www.python.org/dev/peps/pep-0255/.

– *Template-based* formatters are based on template files for stream messages given in the configuration. A set of variables is predefined and replaced on execution by the Stream Player, e.g., as shown in Scenario A (Sect. 3.1).
– With *in-line* formatters the transformations are hard-coded in the Stream Player, as shown for instance in Scenario B (Sect. 3.2).
– *Mapping-language-based* formatters are based on a standardized mapping language such as RML [10], which consists of mapping rules made of a logical source, a subject map and zero or more predicate-object maps.

The type of formatter can be chosen according to the complexity of the scenario, where for simpler scenarios template- or in-line-based and for complex scenarios mapping-language-based formatters can be used.

Regarding availability (F4), the SRP is published under the Apache 2.0 license. The source code and all the scenario data can be found in the Github repository: https://github.com/patrik999/stream-reasoning-challenge. For a fast deployment, it is dockerized and can be delivered as a Docker container.[2]

3 Scenarios

In this section, we outline a traffic management, a vehicle signal processing with object detection, and a custom time-series streaming scenario. The scenarios differ regarding the most complex reasoning tasks and background KB in Scenario A, the highest complexity of streams and novelty of the domain in Scenario B, and the easiness to extend with new sources in Scenario C.

3.1 Scenario A - Traffic Management

The first scenario is in the domain of urban traffic management and involves traffic management for Cooperative Intelligent Transportation Systems (C-ITS). Traffic is observed from a third-person, top-down perspective, and streams of vehicle movements and signal phases (states) of traffic lights in a given road network are generated. In this scenario, we have identified the following (possible) tasks to be tackled:

1. Gathering traffic statistics, *e.g.,* counting the number of vehicles passing;
2. Event detection, *e.g.,* detecting accidents or traffic jams;
3. Diagnosis, *e.g.,* finding the cause for a traffic jam;
4. Motion planning, *e.g.,* routing the vehicles optimally through the network.

Unexpected events could be triggered, *e.g.,* vehicle breakdowns, which lead to possible traffic disruptions. The data source in this scenario is a microscopic traffic simulation framework [33] called Simulation of Urban MObility (SUMO).[3]

[2] https://www.docker.com/.
[3] https://www.eclipse.org/sumo/.

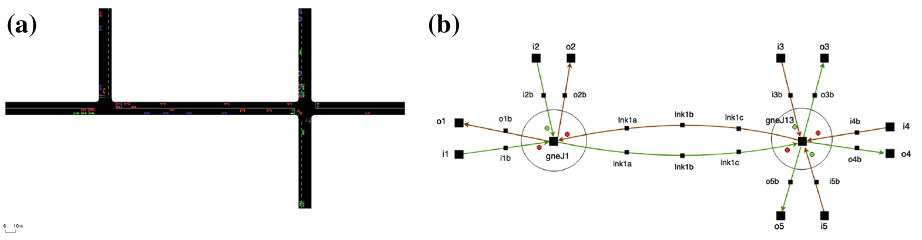


Fig. 2. (a) SUMO rendering of two intersections and (b) corresponding abstract flow model, where green/brown edges are the "we" or "ew" traffic orientation.

The data streams are generated on the fly by different simulation runs on SUMO, where the simulation design is taken from the experiments by Eiter et al. [11].

Data Generation. In Fig. 2a, we show the provided road networks of the scenario rendered in SUMO with two intersections that connect three roads (one horizontal and two vertical). Figure 2b is the graph representation of the road network of Fig. 2a including nodes for intersections, links, sources, and sinks. As shown in the figure, the street layout has two intersections and tree roads with two in/outgoing lanes each, different road segments between intersections, and each intersection with a traffic light controller generating traffic light states based on a static signal plan. We also provide different traffic scenarios to generate traffic streams of varying density, where the `streamid` parameter in the API is used for the initialization of the density:

- Light traffic with free flow (30 vehicles): `&streamid=streamSumo1`;
- Medium traffic with free flow (120 vehicles): `&streamid=streamSumo2`;
- Heavy traffic with traffic jam (180 vehicles): `&streamid=streamSumo3`.

Background KB. A static background KB adds additional immutable information to the data streams. In this scenario, it captures the SUMO graph representation, including simple traffic regulations, traffic signal plan constraints, and a simple vehicle taxonomy. The road network of the SUMO model was rendered into a graph representation encoded as Datalog facts and RDF triples. It is split into segments of the same length as shown in Fig. 2b, where the type *its:node* defines connection points between two edges, and a single edge is represented by an *its:link* property, where its subject is the source of the edge and the sink is given by the *its:linkedTo* property with the additional information on the traffic flow direction, i.e., "we" denotes west to east. Traffic regulations currently come with speed limits (in m/s) that are assigned to an edge using the *its:maxSpeed* property. Conflicts between traffic lights, i.e., lanes that are not allowed to be simultaneously red, are given for each intersection by the *its:conflictingTL* property, which are (hard) constraints regarding a signal plan. We also provide a simple vehicle taxonomy, which adds sub-types using *rdfs:subClassOf* axioms, where vehicles in the streams are relate to leaf types.

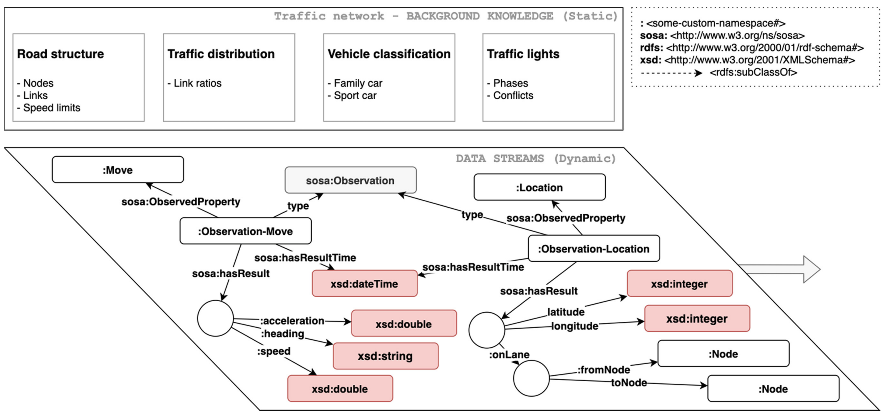


Fig. 3. Representation of the schema used for the streams of scenario A.

Streams. The given data streams are the means for processing and performing the intended evaluation, e.g., a hackathon task. The traffic streams are directly extracted from the SUMO simulation, where we distinguish between *vehicle* and traffic light *signal streams*. The generation of stream messages in this scenario is driven by each simulation step, whereby each step results in a single message for each vehicle and each traffic light signal. Figure 3 provides an overview of the model used for the traffic streams, where the background KB is static and the traffic streams capture moving objects, e.g., the vehicles, and their values as observations. Two observable properties have slightly different annotation patterns for their observations, where the first pattern describes movement-related attributes, such as speed, heading, and acceleration, and the second pattern describes position-related attributes, such as the GPS position or the active lane of the vehicle. Note that the properties *its:onLane* and the vehicle model in *rdf:type*, e.g., carA, refer to facts in the background KB. The attributes for *traffic signal* messages are not outlined here, but are simpler and include the intersection, the traffic light ID, and the signal state (green or red), as well as the message time as a time-point.

We also provide for each data stream a predefined template that allows to render the messages to RDF triples (encoded in JSON-LD) or Datalog facts, where the rendering can be set in the stream initialization by the template types of `traffic-json`, resp., `traffic-asp`. A given set of attributes can be used in the template to complete it on execution.[4] For example in vehicle streams, the variables `$VehicleID$`, `$Type$`, and `$Timestamp$` can be used in templates to add the respective SUMO-generated values.

[4] A full list of attributes is given in https://github.com/patrik999/stream-reasoning-challenge/blob/master/hackathon-2021/Hackaton_Overview.pdf.

Example 2. In the following, we give a (simplified) example message rendered in Datalog: `speed(vehicle:20, 20, 1001). vehModel(vehicle:20, carA).`, stating that *:vehicle:20* of type *carA* moves at the speed of 20 at time-point 1001. The same message rendered as RDF triples is shown below:[5]

```
<vehicle:20> a sosa:Platform;
    a its:carA;
    sosa:madeObservation <obs:20_1001>.
<obs:20_1001> a sosa:Observation;
    sosa:hasResult [ its:speed "20"^^xsd:float ];
    sosa:resultTime "1001"^^xsd:int.
```

Example Tasks. We introduce possible example tasks that could be used for a model and solve hackathon. The tasks increase in difficulty, so the first tasks could be given as a starting point.

Tasks 1. Collection of network traffic statistics updated frequently:

1. Calculating the number of vehicles (NoV) and average speed of all vehicles on each edge;
2. Separated by vehicle super-type, calculate the NoV and their average speed;
3. Based on red traffic lights, detect any vehicles that do a red-light violation.

Tasks 2. Detection of legal/illegal behavior and driving patterns of individual vehicles:

1. Detect the vehicles that perform standard maneuvers: vehicles appear/ disappear in the network, turn left/right, or make a short stop;
2. Detect the vehicles that violate traffic rules: speeding, accident, or u-turn;
3. Detect the vehicles that must stop because of another vehicle's accident.

3.2 Scenario B - Vehicle Signals and Surrounding Objects

The second scenario introduces the challenge of automotive applications to SR modelers/developers, which was triggered by industry stakeholders (e.g., BMW, Bosch, and Siemens) to use SR tools and develope descriptive requirements via well-formulized/standardized data models (e.g., RDF) and query/reasoning tools. Different from the previous scenario, which focuses on a complex background-KB, this scenario will focus on facilitating the access to a large collection of open stream data sources of the automotive industry, provided along with a DNN processing pipeline. We believe that this application domain can foster interesting applications for SR community in years to come, which also distinguishes the playground from current RSP counter-parts.

[5] Additionally to the standard namespaces `rdf`, `rdfs`, and `xsd`, we have `sosa: <http://www.w3.org/ns/sosa/>`, vsso: ¡https://github.com/w3c/vsso#¿, `semkg: <http://vision.semkg.org/onto/v0.1/>`.

In this light, the second scenario consists of data streams produced from the perspective of vehicles. For each time step, data representing the driving context is generated. Although the driving context involves data from several domains (*e.g.,* traffic, weather, infrastructure, and others [16]), we focus on two specific stream sources: (1) the stream of the vehicle's location and movement *e.g.,* speed, acceleration, etc.; (2) the stream of objects that are detected from the images captured by the camera attached in front of the vehicle. In this scenario, we find the following tentative tasks to be solved:

1. Finding relevant behavioral patterns from the driving context, *e.g.,* the flow of traffic around.
2. Finding possible reasons for particular situations, *e.g.,* what was the reason for a particular maneuver?
3. Detection of complex events, *e.g.,* dangerous situations on the road.

Data Generation: The second scenario concerns with the object scene flow for autonomous vehicles. The data used in this scenario is based on a few traces from the KITTI dataset [13], a well-known dataset that has been extensively used for benchmark comparisons in tasks related to autonomous driving. The data consists of images captured from a camera attached to a car, its GPS location, and its speed and acceleration. The data is semantically annotated with SSN [14] and VisionKG[6] vocabulary [20] and is provided as a Linked Data stream using the streaming platform described in Sect. 2.

To annotate the location and movement of the car, we follow the schema as described in Scenario A (see Fig. 3). The result of each *IndividualMove* observation includes the speed (m/s), and the acceleration in X axis and Y axis of the vehicle. The *IndividualLocation* gives the GPS coordinates of the vehicle.

Figure 4 illustrates the semantic schema of the stream data of detected objects. To detect the objects from the images captured by the camera, we use an object detection algorithm (*e.g.,* FRCNN [27] or Yolo [26]) which is annotated as a *procedure* (*sosa:Procedure*). To perform object detection for our data, we use Yolo algorithm version 4 [5]. An object detection is a result of an *observation* (sosa:Observation) that is made by using Yolov4. The result of the object detection is 1:1 associated with a frame by the property *:observedFrame*. A detected object contains a box (*Box*) and a label that names the object (*e.g.,* car, van). X and Y are the coordinates of the center of the box in the image. The width and height values are the size of the box. Figure 5 illustrates an example of how a detected object from a frame is symbolically annotated (see Footnote 5). The box that is labelled with "van" is described as follows. Line 1 represents that the detected object 0 belongs to the detection 0. Line 2 links the detected object 0 with the label "van", and box 0. The coordination and the size of box 0 are annotated from lines 4 to 7.

Example Tasks.

– *Task 1:* Query (detect) other vehicles behavior in the stream of labelled objects collected by the ego-vehicle. Possible tasks are to detect:

[6] https://vision.semkg.org/.

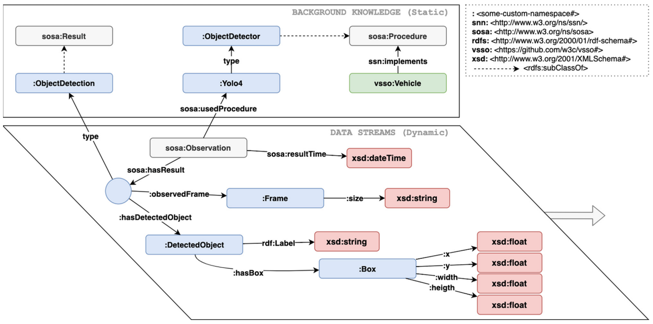
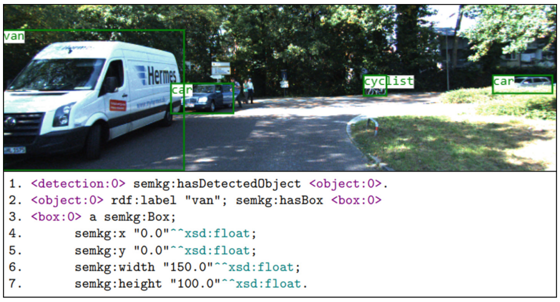


Fig. 4. Schema used for the semantic annotation of objects detected from the video frames of the KITTI dataset



```
<detection:0> semkg:hasDetectedObject <object:0>.
<object:0> rdf:label "van"; semkg:hasBox <box:0>
<box:0> a semkg:Box;
      semkg:x "0.0"^^xsd:float;
      semkg:y "0.0"^^xsd:float;
      semkg:width "150.0"^^xsd:float;
      semkg:height "100.0"^^xsd:float.
```

Fig. 5. Example of the semantic annotations of detected objects in RDF.

1. Detect all oncoming traffic or all crossing traffic.
2. Detect if one object (vehicle) is stationary or moving.

– *Task 2:* Driving scene understanding, find the explanations for certain observations, e.g., stopping because of pedestrians, traffic lights, or other cars.

3.3 Streaming a Custom Time-Series Scenario

While the previous subsections provided details about ready-to-use data streams produced from two distinct scenarios, the platform's users might be interested in streaming out a custom scenario. Hence, we present in this subsection a possible workflow to stream a custom time-series scenario, together with an example of using a public dataset.

Workflow: The workflow, illustrated in Fig. 6, shows how to reuse our platform with arbitrary time-series data sources and annotation schemes. Please note that it aims to show one way of reusing our platform. Alternatively, users might opt to design and implement their own solutions. We assume that a data source representing the scenario of interest is available. Such data source could be either a stored time-series dataset or directly a data stream (*e.g.,* readings of an actual sensor, simulation of transactions, among others). The workflow follows the general description of the so-called semantization process [29], and consists of mainly three steps: (a) *dataset preparation*, (b) *semantic annotation*, and (c) *message broadcast*.

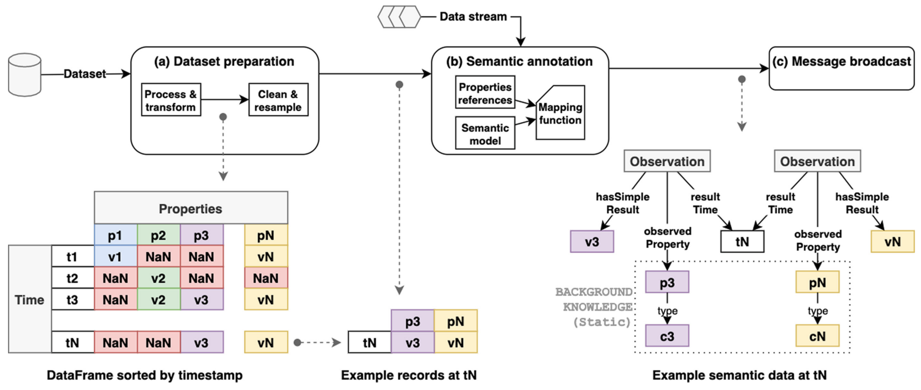


Fig. 6. Overview of the suggested workflow for streaming a custom scenario from a dataset or an actual data stream. (a) datasets must be prepared by the user. (b) a mapping function does the semantic annotation. (c) semantic data is streamed out.

The *dataset preparation* step only applies when the data source is a stored dataset. As datasets are highly diverse, they must be prepared by the user. The idea is to transform the dataset into one data frame, where columns refer to the values of the properties (*aka.*, features, variables) that will be streamed, and rows associate a set of values with the corresponding time. The data frame can then be read as a whole or iterated over its rows. Since values are sometimes unavailable for all columns at a specified time, null values can be cleaned up for simplicity. Alternatively, the user can choose to resample the series to impute or fill in the missing values to have all columns with values at each row.

The *semantic annotation* step annotates the input values with a schema defined from a semantic model. It has a mapping function that populates the schema with the source data values. The mapping itself could be performed in different ways. In the case of RDF data, we recommended using the RDF Mapping Language (RML) [10]. There are a few existing implementations of RML interpreters. For further reference see, for example, *RMLStreamer*.[7] Alternatively, one can also programmatically do the mapping.

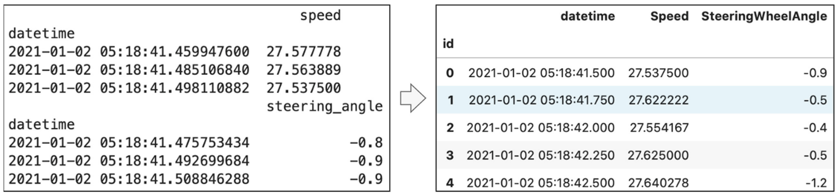
Lastly, the *message broadcast* step takes the resulting semantic data at the current time and streams it out as a message by yielding it to the stream player. An example of the workflow is presented next.

An Example with the comma2k19 Dataset: To demonstrate the suggested workflow, we applied it on the *comma2k19*[8] dataset [28]. This dataset is publicly available and consists of multiple commute journeys (*aka.*, routes) that are split

[7] https://github.com/RMLio/RMLStreamer.
[8] https://github.com/commaai/comma2k19.

into one-minute segments. Data was collected from two vehicles driven mainly on a highway in California, United States. It has data properties available in three groups: Controlled Area Network (CAN) bus, Inertial Measurement Unit (IMU) and Global Navigation Satellite System (GNSS).

We have implemented[9] the proposed workflow with one of the dataset segments and with two dynamic vehicle properties: *Speed* and *SteeringWheelAngle*. However, if needed, the principle could be replicated with more segments or properties. Figure 7 shows an excerpt from the time series data before and after the *data preparation* step.



```
                                   speed
datetime
2021-01-02 05:18:41.459947600  27.577778
2021-01-02 05:18:41.485106840  27.563889
2021-01-02 05:18:41.498110882  27.537500
                               steering_angle
datetime
2021-01-02 05:18:41.475753434            -0.8
2021-01-02 05:18:41.492699684            -0.9
2021-01-02 05:18:41.508846288            -0.9
```

id	datetime	Speed	SteeringWheelAngle
0	2021-01-02 05:18:41.500	27.537500	-0.9
1	2021-01-02 05:18:41.750	27.622222	-0.5
2	2021-01-02 05:18:42.000	27.554167	-0.4
3	2021-01-02 05:18:42.250	27.625000	-0.5
4	2021-01-02 05:18:42.500	27.640278	-1.2

Fig. 7. Sample sequences of speed and steering wheel angle before and after the dataset preparation.

Regarding the *semantic annotation*, we defined a custom schema and a mapping function. The schema, shown in Fig. 8, is based on the combination of the ontologies Sensor, Observation, Sample, and Actuator (SOSA) [15] and the Vehicle Signal Specification (VSSo) [34] (See footnote 5). It is used as a template for the mapping function, implemented with RML rules. An excerpt of the rule that maps the *Speed* is shown below:

```
rr:subjectMap [
    rr:template "http://sr-challenge/vehicle/speed/observation/{id}" ;
    rr:class sosa:Observation];
rr:predicateObjectMap [
    rr:predicate sosa:hasSimpleResult;
    rr:objectMap [
        rml:reference "Speed" ;
        rr:datatype xsd:float ] ] ;
```

Consequently, the resulting semantic data is passed as a message that the player will stream out. For instance, the semantic data (in RDF turtle format) at a particular time looks like the following:

```
<http://sr-challenge/vehicle/speed/observation/1> a sosa:Observation;
    sosa:hasSimpleResult "27.622222222222224"^^xsd:float;
    sosa:observedProperty  indv:Speed;
    sosa:resultTime "2021-01-02 05:18:41.750"^^xsd:dateTime .
```

[9] https://github.com/patrik999/stream-reasoning-challenge/blob/master/example-custom-scenario/workflow.ipynb.

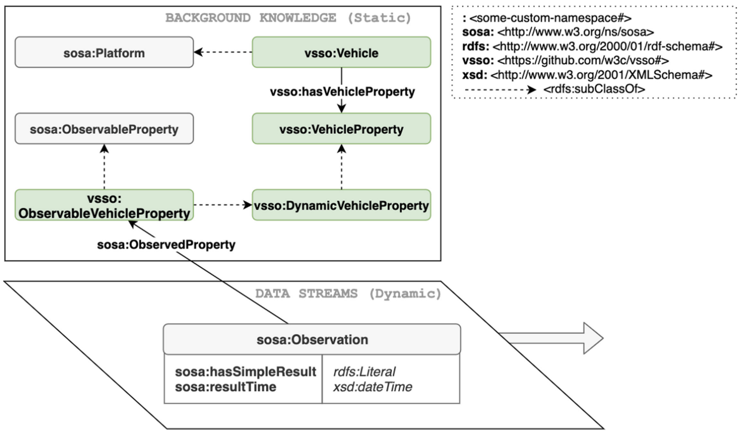

BACKGROUND KNOWLEDGE (Static)
sosa:Platform
vsso:Vehicle
vsso:hasVehicleProperty
sosa:ObservableProperty
vsso:VehicleProperty
vsso: ObservableVehicleProperty
vsso:DynamicVehicleProperty
sosa:ObservedProperty
DATA STREAMS (Dynamic)
sosa:Observation
sosa:hasSimpleResult
sosa:resultTime
rdfs:Literal
xsd:dateTime
: <some-custom-namespace#>
sosa: <http://www.w3.org/ns/sosa>
rdfs: <http://www.w3.org/2000/01/rdf-schema#>
vsso: <https://github.com/w3c/vsso#>
xsd: <http://www.w3.org/2001/XMLSchema#>
<rdfs:subClassOf>

Fig. 8. Schema used in the mapping function. It combines the SOSA and the vehicle signal specification ontology.

```
<http://sr-challenge/vehicle/steering/observation/1> a sosa:Observation;
    sosa:hasSimpleResult "-0.5"^^xsd:float;
    sosa:observedProperty indv:SteeringWheelAngle;
    sosa:resultTime "2021-01-02 05:18:41.750"^^xsd:dateTime .
```

Please refer to the platform's Github repository for further details, such as the complete RML mapping rules, the Stream Player implementation, and explanations of the workflow.

4 Case Study and Lessons Learned: SR Hackathon 2021

The stream generation platform, together with the Scenarios A and B, was for the first time applied in the Stream Reasoning (SR) Hackathon 2021,[10] which was organized as part of the SR Workshop in Milan, Italy.[11] This hackathon allowed both onsite/remote participation and was designed as a "model and solve" challenge, where participants had the freedom to choose their own SR pipelines and reasoners. The principal milestones were: (1) Hackathon announcements prior to the competition, including the description of preliminary tasks to let the participants familiarize with the platform; (2) Introduction of participants, general overview of the platform, and detailed discussion of the tasks at the beginning of the event; (3) Intermediate short sync discussions between organizers and participants to clarify problems and fix problems in the stream players and data formatters; (4) Presentation of solutions by all teams and an online voting to determine the most interesting solutions at the end.

[10] http://streamreasoning.org/stream-reasoning-hackathon-2021.
[11] http://streamreasoning.org/events/srw2021.

4.1 Solutions of the Participants

In this section, we give a short overview of the participating teams and their suggested solutions, where the solutions of Oxford University and University of Calabria collected the most votes.

FAU Erlangen-Nürnberg. The team of FAU used *Stream Containers*, which are designed to map RDF streams to a RESTful architecture by the decomposition and decentralization of stream query evaluation. The decomposition is based on a stream-to-relation (S2R) operator for creating a snapshot of tuples based on a window function, and a relation-to-stream (R2S) operator for creating streams with newly time-stamped instances. *Example solution:* `SELECT ?v ?s (AVG(?z) AS ?s) WHERE {?v :madeObservation ?y. ?y :hasResult ?x. ?x its:speed ?z } GROUP BY ?v`, where the sliding window is managed by the embedding stream container.

NCSR Demokritos. NCSR's team applied the solver Wayeb [1] for complex event forecasting using a streaming extension of symbolic automata. Symbolic automata extend deterministic finite automata with Boolean algebra that can be defined over an infinite domain. Complex (event) patterns can be defined as regular expressions with concatenation as $\cdot$, and Kleene-star as $*$. *Example solution:* `R = (speed > 13)·(speed > 13)`, detecting that the event `(speed > 13)` occurs twice on the (vehicle) speed stream.

Oxford University. The team of Oxford suggested a hybrid approach to participate at the hackathon. According to the given tasks, a decision module selects between two solvers, namely RDFox [23] and MeTeoR. In RDFox, they applied standard Datalog rules including aggregation. MeTeoR was applied, where metric temporal logic (MTL) operators such as $\boxminus_{[0,5]}$ for simulating a window operator, e.g., of 5 time steps (ts), were needed. *Example solution:* `avgSpeed[?Z,?T] :- AGGREGATE(onLane[?X, ?Z], speed[?X,?S] ON ?Z BIND AVG(?S) AS ?T).`

University of Calabria. The team of UniCal competed with an approach that extends static Answer Set Programming (ASP) with streaming features. The solver is called I-DLV-sr [7] and combines an incremental grounding solver [6] with streaming data forwarded from Apache Flink. *Example solution:* `accid(X) :- speed(X,0) always in [25].`, where the last part of the rule states that `speed(X,0)` has to occur always in a 25 ts window.

TU Wien. The TU Wien team followed the spirit of a hackathon and extended the static ASP solver Clingo [12] to handle streaming data. This was achieved by a Python-based stream handler that emulates a window operator and generates time-dependent facts. *Example solution:* `cNoV(N,X,Y,T) :- link(X,Y),` `time(T), N=#count{I:onLane(I,X,Y,T))},N` $\neq$ `0.`, where `cNoV(N,X,Y,T)` can then be chained in a new rule to sum up several time points.

4.2 Lessons Learned

The lessons learned include a user survey after the hackathon based on the introduced features, but also covers our own evaluation of organizational aspects.

Hackathon Survey. We conducted a survey after the hackathon, where six questions were asked to the participants.[12] The questions related directly to the platform features with the results shown in parenthesis: Overall suitability (4.2 out of 5 points), suitability for SR (3.4 out of 5 points), extensibility with new features (4.4 out of 5 points), preferred usability aspects (deployment, API, and fast error fixing), and difficulty of the"model and solve" tasks (well balanced). The last question regarding platform improvements is discussed below.

Own Evaluation. Using Github and providing an easy-to-follow installation to replicate the environment via Docker were positive decisions and allowed participants to quickly deploy after we made changes. The release of the initial hackathon's tasks a few weeks before the competition helped participants get familiar with the platform and set up their working environments on time, where the use of the messaging platform Slack was essential for the rapid communication between (remote) teams and organizers. The increasing difficulty of the tasks was an appropriate way to keep the engagement of participants. Tasks that were not fully solved are a clear indicator of possible future work. Giving the participants the freedom to "model and solve" the solutions resulted (as expected) in different ways of solving the tasks, which is an excellent way to cope with the diverse techniques and approaches existent in the SR area.

What Could Be Improved? From the participants perspective, the following suggestions for improvements were given: (1) For Scenario A, provide a larger set of streams, (2) give a clearer definition of the tasks and some example solutions, (3) provide all formats for all scenarios including plain JSON, (4) add benchmarking features for automated measurements, (5) extend the protocols so other communication methods such as Apache Kafka Producer API could be used. From our perspective, we released the platform/documentation only a few weeks in advance, hence the timeline was tight to spot, and issues needed to be fixed on-the-fly. Therefore, the earlier the competition details and tools can be released, the better.

Based on the evaluation, we conclude that the reuse of the proposed platform is recommended as it constitutes a strong foundation for the preparation of future competitions, where the focus could be on the other scenarios presented or on entirely new scenarios taken from different domains such as robotics.

[12] The questions and results are provided in https://github.com/patrik999/stream-reasoning-challenge/blob/master/hackathon-2021/Survey.pdf.

5 Related Work

There have been many benchmarks and data generators for RDF stream data processors. The earliest ones are SRBench [35], LSBench [18], and CityBench [2], which focus on the query features of C-SPARQL and CQELS-QL. SRBench uses data from three sources, i.e., LinkedSensorData, Geonames and DBpedia to create data streams. LSBench provides social network stream data via its simulated data generator. CityBench provides stream data from a Smart City application for the city of Aarhus, Denmark. Recently, integrated tools and benchmarks, such as TripleWave [22] and RSPLab [32] aim to reduce the effort required to design and execute reproducible experiments as well as share their results. RSPLab integrates two existing RSP benchmarks (LSBench and CityBench) and two RSP engines (C-SPARQL engine and CQELS). It provides a programmatic environment to deploy in the cloud RDF Streams and RSP engines. While LSBench's and CityBench proposed two test-drivers that push RDF Stream to the RSP engines subject of the evaluation, RSP Lab is developed to be benchmark-independent. The most common processing features of this line of work are based on SPARQL such as C-SPARQL and CQELS-QL. In some cases, complex event query patterns such as [3] and [8] are also introduced.

There have been some extensions of the above RSP-based data generators to accommodate reasoning features. However, such extensions only cover some small set of reasoning features such as RDFS or a fragment of OWL-DL. In SRP, reasoning is the first-class feature by design which is motivated the emergent application scenarios around V2X and autonomous driving. Our playground has been encouraged by the developers of stream reasoners participating at the annual stream reasoning workshops (see Footnote 11). For instances, the ones listed in Sect. 4 are among the well-known reasoners. Moreover, with the extensibility of the playground presented above, the mentioned benchmarking systems can be reused in our players for their systems.

6 Conclusion

This work is sparked by the diversity of approaches in the field of SR, making it challenging to compare formal languages, approaches/pipelines, and reasoners. Existing works from the RSP community have constituted a good starting point, but they are too restrictive regarding the data model and the reasoning tasks (*i.e.,* RDF and query answering). To overcome these limitations, we presented the Stream Reasoning Playground (SRP), which is an open source platform and treats SR as a first-class use case. As indicated by a user survey, it provides a satisfying level of extensibility (F3), consumer agnosticism (F2), availability/agility (F4), and a base scenarios with reasoning tasks that were considered as "well balanced" (F5/F6). The SRP comes with an easy-to-configure and extensible platform to generate streams for different scenarios based on stream players and pluggable data formatters. Notably, besides two well-defined ready-to-use ITS scenarios, we have described a workflow for streaming custom time-series data.

The features were evaluated in a case study based on the SR Hackathon 2021, where we reported on developed solutions, a user survey, and lessons learned.

We consider that the following aspects deserve the attention of the future development of the SRP: (i) defining and including benchmarking features and the corresponding metrics to validate and compare the performance of different solutions to a standard set of tasks; (ii) extending SRP to support streams that include probabilities and addition of probabilistic reasoning features to integrate deep learning models and common-sense reasoning, such as [24,30]; (iii) organizing a repository for community-contributed scenarios and data sets; and (iv) improving the semantization step by adding new formatters covering other languages and a tighter integration of RML.

Acknowledgements. This work was funded by the German Research Foundation under grant nr. 453130567 (COSMO), the German Ministry for Education and Research BIFOLD grant nr. 01IS18025A and 01IS18037A, the German Academic Exchange Service grant nr. 57440921, and the EU Horizon 2020 Research and Innovation program under grant nr. 779852 (IoTCrawler).

References

1. Alevizos, E., Artikis, A., Paliouras, G.: Wayeb: a tool for complex event forecasting. CoRR abs/1901.01826 arXiv:1901.01826 (2019)
2. Ali, M.I., Gao, F., Mileo, A.: Citybench: a configurable benchmark to evaluate RSP engines using smart city datasets. In: Arenas, M., et al. (eds.) The Semantic Web - ISWC 2015. Lecture Notes in Computer Science, vol. 9367, pp. 374–389. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_25
3. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: Srinivasan, S., Ramamritham, K., Kumar, A., Ravindra, M.P., Bertino, E., Kumar, R. (eds.) Proceedings of the 20th International Conference on World Wide Web, WWW 2011, pp. 635–644. ACM (2011). https://doi.org/10.1145/1963405.1963495
4. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. In: Proceedings of the 18th International Conference on World Wide Web, pp. 1061–1062 (2009)
5. Bochkovskiy, A., Wang, C.Y., Liao, H.Y.M.: Yolov4: optimal speed and accuracy of object detection. arXiv preprint arXiv:2004.10934 (2020)
6. Calimeri, F., Ianni, G., Pacenza, F., Perri, S., Zangari, J.: Incremental answer set programming with overgrounding. Theory Pract. Log. Program. **19**(5–6), 957–973 (2019). https://doi.org/10.1017/S1471068419000292
7. Calimeri, F., Manna, M., Mastria, E., Morelli, M.C., Perri, S., Zangari, J.: I-DLV-sr: a stream reasoning system based on I-DLV. Theory Pract. Log. Program. **21**(5), 610–628 (2021). https://doi.org/10.1017/S147106842100034X
8. Dell'Aglio, D., Dao-Tran, M., Calbimonte, J., Phuoc, D.L., Valle, E.D.: A query model to capture event pattern matching in RDF stream processing query languages. In: Blomqvist, E., Ciancarini, P., Poggi, F., Vitali, F. (eds.) Knowledge Engineering and Knowledge Management, vol. 10024, pp. 145–162 (2016). https://doi.org/10.1007/978-3-319-49004-5_10

9. Dell'Aglio, D., Della Valle, E., van Harmelen, F., Bernstein, A.: Stream reasoning: a survey and outlook: a summary of ten years of research and a vision for the next decade. Data Sci. **1**(1–2), 59–83 (2017). https://doi.org/10.3233/DS-170006
10. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: a generic language for integrated RDF mappings of heterogeneous data. In: Proceedings of the 7th Workshop on Linked Data on the Web, April 2014
11. Eiter, T., Falkner, A.A., Schneider, P., Schüller, P.: ASP-based signal plan adjustments for traffic flow optimization. In: Giacomo, G.D., et al. (eds.) ECAI 2020–24th European Conference on Artificial Intelligence, Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020). Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 3026–3033. IOS Press (2020)
12. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.T.: Potassco: the potsdam answer set solving collection. AI Commun. **24**(2), 107–124 (2011)
13. Geiger, A., Lenz, P., Stiller, C., Urtasun, R.: Vision meets robotics: the kitti dataset. Int. J. Robot. Res. **32**(11), 1231–1237 (2013)
14. Haller, A., et al.: The modular SSN ontology: a joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation. Semant. Web **10**(1), 9–32 (2019). https://doi.org/10.3233/SW-180320
15. Janowicz, K., Haller, A., Cox, S.J., Le Phuoc, D., LefrançSois, M.: SOSA: a lightweight ontology for sensors, observations, samples, and actuators. J. Web Semant. **56**, 1–10 (2019). https://doi.org/10.1016/j.websem.2018.06.003
16. Klotz, B., Troncy, R., Wilms, D., Bonnet, C.: A driving context ontology for making sense of cross-domain driving data (2018). https://www.researchgate.net/publication/331991645_A_driving_context_ontology_for_making_sense_of_cross-domain_driving_data
17. Klyne, G., Carroll, J.J.: Resource description framework (RDF): concepts and abstract syntax. W3C Recommendation (2004). http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/
18. Le-Phuoc, D., Dao-Tran, M., Pham, M.-D., Boncz, P., Eiter, T., Fink, M.: Linked stream data processing engines: facts and figures. In: Cudré-Mauroux, P., et al. (eds.) The Semantic Web – ISWC 2012. LNCS, vol. 7650, pp. 300–312. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35173-0_20
19. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Aroyo, L., et al. (eds.) The Semantic Web – ISWC 2011. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25073-6_24
20. Le-Tuan, A., Kien-Tran, T., Nguyen-Duc, M., Yuan, J., Hauswirth, M., Yuan, J.: VisionKG: towards a unified vision knowledge graph. In: Proceedings of the ISWC 2021 Posters and Demonstrations Track. CEUR Workshop Proceedings (2021)
21. Margara, A., Urbani, J., van Harmelen, F., Bal, H.: Streaming the web: reasoning over dynamic data. J. Web Seman. **25**, 24–44 (2014). https://doi.org/10.1016/j.websem.2014.02.001
22. Mauri, A., et al.: TripleWave: spreading RDF streams on the web. In: Groth, P., et al. (eds.) The Semantic Web – ISWC 2016. LNCS, vol. 9982, pp. 140–149. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46547-0_15
23. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: a highly-scalable RDF store. In: Arenas, M., et al. (eds.) The Semantic Web - ISWC 2015. Lecture Notes in Computer Science, vol. 9367, pp. 3–20. Springer (2015). https://doi.org/10.1007/978-3-319-25010-6_1

24. Phuoc, D.L., Eiter, T., Lê Tuán, A.: A scalable reasoning and learning approach for neural-symbolic stream fusion. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, pp. 4996–5005. AAAI Press (2021). https://ojs.aaai.org/index.php/AAAI/article/view/16633
25. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation, January 2008. http://www.w3.org/TR/rdf-sparql-query/
26. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: unified, real-time object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 779–788 (2016)
27. Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: towards real-time object detection with region proposal networks. Adv. Neural Inf. Process. Syst. **28**, 91–99 (2015)
28. Schafer, H., Santana, E., Haden, A., Biasini, R.: A commute in data: the comma2k19 dataset. arXiv:1812.05752 (2018)
29. Shi, F., Li, Q., Zhu, T., Ning, H.: A survey of data semantization in Internet of Things. Sensors **18**(2), 313 (2018). https://doi.org/10.3390/s18010313
30. Suchan, J., Bhatt, M., Varadarajan, S.: Commonsense visual sensemaking for autonomous driving - on generalised neurosymbolic online abduction integrating vision and semantics. Artif. Intell. **299**, 103522 (2021). https://doi.org/10.1016/j.artint.2021.103522
31. Tommasini, R., Bonte, P., Ongenae, F., Della Valle, E.: RSP4J: an API for RDF stream processing. In: Verborgh, R., et al. (eds.) The Semantic Web, ESWC 2021. LNCS, vol. 12731, pp. 565–581. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77385-4_34
32. Tommasini, R., Della Valle, E., Mauri, A., Brambilla, M.: RSPLab: RDF stream processing benchmarking made easy. In: d'Amato, C., et al. (eds.) The Semantic Web – ISWC 2017. LNCS, vol. 10588, pp. 202–209. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_21
33. Treiber, M., Kesting, A.: Traffic Flow Dynamics: Data, Models and Simulation. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-32460-4
34. Wilms, D., Alvarez-Coello, D., Bekan, A.: An evolving ontology for vehicle signals. In: 2021 IEEE 93rd Vehicular Technology Conference (VTC2021-Spring), pp. 1–5. IEEE, Helsinki (2021). https://ieeexplore.ieee.org/document/9448884/, https://doi.org/10.1109/VTC2021-Spring51267.2021.9448884
35. Zhang, Y., Duc, P.M., Corcho, O., Calbimonte, J.-P.: SRBench: a streaming RDF/SPARQL benchmark. In: Cudré-Mauroux, P., et al. (eds.) The Semantic Web – ISWC 2012. LNCS, vol. 7649, pp. 641–657. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35176-1_40