Querying Flying Robots and Other Things:

Ontology-Supported Stream Reasoning

A discussion on the role of ontologies and stream reasoning in Internet of Things applications.

By Daniel de Leng

Imagine a world in which the Internet has become so ubiquitous that it extends itself even to everyday *things*, such as our cars, fridges, and milk cartons. How far are we then removed from being able to query *things* for the location of a delivery van, or as some companies foresee it, flying delivery robots? After all, sensor platforms could themselves be regarded as complex things. Could I perhaps ask them more complex questions that involve many (complex) *things*?

Unmanned Aerial Vehicles (UAVs) have increasingly become a topic of discussion in today's society, in part because of having become more and more affordable and easy to use. Many are equipped with cameras for taking pictures or recording video, and commercial UAVs marketed for the general public can sometimes be controlled through simple means such as tablets. The wide-spread deployment of UAVs has not come without consequence. UAVs have been blamed for privacy infringement, airspace intrusions and near-misses with commercial airliners, and military applications. But they also serve as useful tools for artists and farmers, help to monitor the health of levees protecting increasingly many people, and have the immense potential to save lives in hard-to-reach disaster areas when things do go wrong - by assisting rescue workers in acquiring information they themselves cannot readily obtain. They too play a role in the Internet of Things as complex entities that have the potential to provide a wider array of services compared to the more traditional *things*, as part of a diverse ecology.

In this text we focus on a type of *stream reasoning* involving a wide variety of *things* at the application layer. So what is stream reasoning, and what does it mean in this context? Many have pointed out that the amount of raw unfiltered data that is produced per day is so enormous that any attempts at storing this data for processing is very difficult, if not impossible. Some of this data is produced by humans: Social media communication platform Twitter reported in 2013 that it saw a spike of 143,199 tweets (messages) in a single second. Other data is produced by sensors or programs at potentially high intervals. In both cases, the data becomes incrementally available in what we call *streams*. Stream reasoning is the reasoning over these streams, bearing in mind that any received data sample is likely to be forgotten shortly after arrival. By stream reasoning we mean the answering of questions or drawing conclusions over streams, in part through aggregation of raw data into more abstract information using various means of processing. In a way, stream reasoning attempts to make sense of a large volume of incrementally-available data as it arrives.

Figure: A Yamaha RMAX used by Linköping University.

A Perimeter Monitoring Example

Let us consider a situation where we want to know whether someone is trespassing during a certain time period. Maybe the area is a construction site deemed unsafe. We are therefore not interested in 'catching' a possible trespasser. Rather, we wish to warn the trespasser to make him or her leave at their own accord first. Sending someone over to deal with the situation is a secondary course of action in case this is unsuccessful.

The restricted area is surrounded by a security fence, which has two gates with sensors detecting whether they are closed or not. We also have a few cameras on-site, and we have a private security guard in case human intervention is needed. We also have several small UAVs with cameras to cover the area, and a third-party virtual server provider for computational power at a price. Finally, all of these *things* are able to interact over the Internet and are aware of each other's identities as part of a local security network.

Suppose we wish to query this security network in order to determine for the next 24 hours if and when a trespasser stays within the restricted area for more than 15 minutes. There are many possible languages to pose such a query in. We focus on temporal logics to leverage their expressivity with regards to temporal concepts. In particular, using Metric Temporal Logic (MTL), this query can be posed by evaluating the following formula:

$$\Box_{[0,1440]}(\mathsf{Person}(x) \wedge \mathsf{Inside}(x, \mathsf{restricted}) \rightarrow \Diamond_{[0,15]} \neg \mathsf{Inside}(x, \mathsf{restricted}))$$

Here $\Box_I$ stands for 'it must always be the case within interval *I*', and $\Diamond_I$ stands for 'eventually it will be the case within interval *I*'. The intervals are assumed to be measured in minutes. Concretely, this formula returns *False* if there is an instance x that is a person and inside of the restricted area, and does not leave the restricted area within 15 minutes. Otherwise it returns *True*.

Such a formula would of course be easy to evaluate if only we had data for the truth values of the individual predicates at all time-points. It is more likely that we do not have this data in its completeness, nor does this data come from a single source. This thus entails some form of stream reasoning using various heterogeneous data resources. Could we reasonably require the UAVs to evaluate this formula? What is reasonable depends on e.g. whether the UAV has access to the required data resources, whether it has the necessary processing capacity, and whether the data resources are reliable enough. This is a multifaceted and interesting problem, for which we do not yet have a general solution.

Setting Up Stream Reasoning

Given a complex query such as a temporal logic formula, we want to have the information that is referenced by the symbols used in such a query. From a streaming perspective, this information is represented as streams, and so the challenge becomes to generate a stream with the desired information. We call this an *applicable stream*, to indicate that it is applicable in relation to the query.

There exist a number of approaches to stream processing. Complex event processing (CEP) considers event streams where every stream sample corresponds to an event. We can then consider complex events, i.e. temporal combinations of (non-)events that are represented as events themselves. An example of CEP would go as follows. First, there is an event that car A is closely behind car B. This is followed by the event that car A is beside car B. Finally we detect the event that car A is in front of car B. This could then be seen as the 'overtake' complex event. Data Stream Management Systems (DSMS) take a different approach. Here streams are sequences of values on which we apply various variants of window operations, such as sliding or tumbling windows. This results in manageable tables unto which the usual database aggregation methods can be applied. Take for example a stream of speeds produced by a smartphone, as is not uncommon for runners. These values may individually deviate a lot due to hardware quality, so we could then take the average over a sliding window. Clearly there are many similarities between DSMS and CEP. Both CEP and DSMS are applicable for Internet of Things applications.

The focus areas for stream processing are very diverse. Some research focuses on query expressivity and the ability to make use of the absence rather than exclusively the presence of data. Another focus area is that of performance; how can we maximize the throughput of these systems? Yet another area considers streams of RDF triples in the context of the Semantic Web. Most of these areas consider a single stream processing engine. We take a different approach that focuses on the problem of integrating many (specialized) stream processing engines within the robotics domain. This of course comes at a cost; we concern ourselves less with throughput optimization, and we focus primarily on fast and quantitative sensor data.

These specialized stream processing engines effectively offer stream-based services, taking streams of data as input and producing output streams in accordance with their specifications. A middleware architecture can be used to manage this multitude of services. Suppose that such a stream reasoning framework distinguishes between streams, transformations, and "computational units" (CUs). Streams are named and can carry a vector of values in every sample. Every sample also contains the timestamp at which the sample became available, and the timestamp for which the data is valid. These two

timestamps can be different, for example when a stream contains predictions about a future time. Transformations are stream-generating functions that take other streams as input. They can, for example, be used to generate a stream of speeds from a stream of coordinates, or serve as data sources by relaying sensor information or importing streams from outside of the system. Transformation specifications describe how to instantiate a transformation and what parameters to use, and so multiple specifications can describe the same implementation with different parameters. CUs represent instantiations of transformations. They have unique names and can have subscriptions to existing streams. CUs can be created and destroyed as desired. DyKnow [1] represents an instance of such flexible middleware architectures. We can use systems such as DyKnow to evaluate temporal logic formulas in MTL. To do so, a configuration specification describing which transformations and CUs to use and how to connect them needs to be executed. However, writing such a configuration specification is tedious and error-prone. It significantly complicates the writing of queries, and it is not very scalable. After all, we want to use information resources provided by the *things* we have access to. Another complication is that a particular information resource may not always be available.

Managing Semantics

Misunderstandings happen. As a Dutch person, I am convinced that my office is on the ground floor, whereas my American partner claims it is on the first floor, and the university considers it to be on the second floor due to the presence of a basement. This works for humans, but can be disastrous for machines. Do we use metric or Imperial units of measurement? What frame of reference is used for coordinates? Misunderstandings with regard to these examples can crash your UAV, literally. As a real-world example, the 'Mars Climate Orbiter Mishap' was the result of unintentionally using different units of measurement, resulting in the unfortunate loss of an interplanetary probe.

There is a need for *things* to have some degree of semantic understanding. Granted, specifications can help harmonize semantics, but how well does this scale when we add further systems into the mix, some of which were written by other people? Ideally we would like to simply write a query describing the information we are interested in without having to worry about how the information is acquired, unless we explicitly put constraints on that - a concept similar to declarative languages like Prolog or SQL.

DyKnow does this by representing its state in terms of streams, transformations and CUs, with the help of an ontology. Ontologies formally describe concepts and the relations between those concepts, and are based on Description Logics. They can be used as a common vocabulary or as a data model, among other things. We use an ontology to query whenever we need information about the state of a DyKnow instance. Facts are then represented using RDF triples, consisting of a subject, a predicate, and an object, in accordance with the ontology. We can then also describe properties of individuals (objects), such as specific streams, CUs or transformations. This allows us to assign properties to transformations, for example to describe the semantics of their inputs and outputs. We call these properties *semantic annotations*. Since transformation instances are CUs, and CUs produce streams, these semantics indirectly describe the information contained within streams. This approach is similar to OWL-S and its service profiles, or the Semantic Sensor Network (SSN) ontology.

One obvious weakness to this approach is that someone or something needs to provide these semantic annotations, although the same is true of transformation specifications themselves. Another is that it presumes concepts already exist for annotating transformations with. Usually the desired semantic annotations of transformations are application or domain specific. Ultimately it is the expressivity of the annotation language that determines whether queries using that language have the intended effects. This is by no means a trivial problem. But what benefits do we gain, assuming we have a suitable, developer-provided domain-specific annotation language?

Recall that in order to set up stream reasoning, we require a configuration. Assume we have a description of some kind of desired information in the form of a query using the vocabulary provided by the ontology. The semantic annotations make generating a configuration something that can be done automatically by recursively searching for appropriate transformations. This is similar to configuration planning, and is based on introspective capabilities. The matching of desired semantics with semantic annotations is a procedure we call *semantic matching*. It returns a set of applicable transformation trees - trees of transformations where the leafs take no inputs and the root produces the desired information. Some of these transformations may use other *things*. If only a single solution is found, it can immediately be instantiated. If no solutions are found, the query cannot be answered, but some approximations may be possible by relaxing constraints. If multiple solutions are found, there are many ways of choosing between them. One can associate costs with instantiating transformations, favoring pre-existing CUs. Alternatively one may desire smaller trees as a heuristic for CPU load. One could also take into account some preferential ordering over the possible information providers, such as preferring on-board sensors or external image processing.

Going back to the example, note that both the cameras and the UAVs are able to provide camera data of various areas. If we have some person detectors and trackers, we could determine whether the predicate 'Inside' holds. If the UAVs do not have the processing capacity, one might support using external computation facilities to offload the heavy work. A progressor can be used to evaluate MTL formulas. If the formula evaluates to *False*, we can regard this as a violation event itself, which can be sent to the security guard. This way multiple *things* offer services that are combined based on their semantic annotations.

But even here things can go wrong. CUs may crash or stall, streams may be of low quality or stop altogether. Perhaps the wireless signal went bad, or the server providing CCTV streams went down. Some processing has to change to accommodate failures. This would not be possible if the subscriptions were syntactic, but since they are semantic we can attempt to repair the broken pipeline and clean up any CUs that are now inactive due to the broken inputs. None of this *requires* human input (but does not exclude it), which makes for a very adaptive system.

Conclusion

The Internet of Things opens up many exciting opportunities for acquiring information resources and for the sharing of information between *things*. A lot of information comes in the form of streams. Being able to use this information has the potential to greatly enhance Internet of Things applications. This is not limited to static household objects, but can be extended to autonomous robots. However, one precondition is that the information available is understood, and for this the semantics of the streams,

transformations and CUs need to be made clear. We need to understand who provides what information with which semantics under which constraints at what price within which time period and with what quality. Ontologies and semantic web technologies can help with this.

When *things* come to agree on the semantics of streaming data, many interesting applications are possible.

References

[1] de Leng, D. and Heintz, F. 2015. Ontology-Based Introspection in Support of Stream Reasoning. In Proceedings of the Thirteenth Scandinavian Conference on Artificial Intelligence (SCAI).

Biography

Daniel de Leng is a Ph.D. student at the Department of Computer and Information Science at Linköping University in Sweden. His work focuses on on-demand knowledge acquisition for grounded spatio-temporal stream reasoning through collaboration and the semantic web.