Enhancing IoT Interoperability: Leveraging Web of Things (WoT) Thing Descriptions and Semantic Schemas

- Prof. Badrinath Sir

- Krish Dave (IMT2022043)
- Vaibhav Mittal (IMT2022126)

Motivation

Why IOT ?

The Internet of Things (IoT) refers to the network of physical devices embedded with sensors, software, and connectivity to collect and exchange data. But why should we care?

1. Real-Time Monitoring and Control

- Enables live tracking of devices, environments, and people (think smart thermostats, wearable health trackers,).
- Improves decision-making through real-time feedback.

2. Automation and Efficiency

- Automates repetitive tasks (e.g., turning off lights when no one's in the room).
- Saves time, energy, and resources — especially in industries like manufacturing, agriculture, and logistics.

- IoT isn’t just about connecting devices — it’s about making them understand each other.

1. Heterogeneity of Devices

- Problem: IoT devices come from different vendors, use different data formats, and often don’t speak the same “language.”
- Motivation: Semantics provides a shared vocabulary so devices, platforms, and applications can understand and use each other’s data meaningfully.

2. Machine Interpretability

- Problem: Raw data (like temp: 35) is useless without context — is it hot? Is it Celsius? What does it refer to?
- Motivation: Semantics gives data meaning, enabling machines to understand, reason about, and act on it without human intervention.

3. Dynamic Environments

- Problem: IoT environments evolve — new devices, new domains, new rules.
- Motivation: Semantic models are extensible and adaptable, allowing systems to evolve gracefully without breaking everything.

4. Web of Things (WoT) Alignment

- Problem: The IoT industry is moving toward the Web of Things (WoT), where devices are exposed as RESTful resources with metadata.
- Motivation: Semantics is key to describing devices in a machine-readable way (e.g., using JSON-LD, RDF, Thing Descriptions).

Importance of IoT-Lite

IoT-Lite plays a critical role in bridging the gap between raw sensor data and meaningful, machine-understandable knowledge in resource-constrained environments.

1. Lightweight Semantic Modeling

- Full ontologies like SSN (Semantic Sensor Network) can be too heavy for IoT environments, especially with limited CPU, memory, and power.
- IoT-Lite simplifies these models, keeping only the essential concepts, making it suitable for embedded devices and edge computing.

2. Interoperability Across Devices and Systems

- IoT-Lite provides standardized vocabularies so different devices, services, and platforms can understand and interact with each other, even across vendors.
- This reduces vendor lock-in and improves system integration.

3. Foundation for Context-Aware and Intelligent IoT

- With IoT-Lite, sensor data is not just numbers — it's semantically described (e.g., "temperature of greenhouse air").
- This enables:
  - Context awareness (e.g., what a value means in a situation),
  - Reasoning using tools like Apache Jena or Drools,
  - Rule-based automation (e.g., "If temperature > 30°C → turn on fan").

4. Better Data Discovery and Reusability

- IoT-Lite provides machine-readable metadata that can be queried and reused easily.
- Devices can be self-describing, meaning other systems can automatically discover and understand how to interact with them.

Introduction to IoT Interoperability

Key Technical Points:

- **Heterogeneous IoT Devices** : Different manufacturers use different data formats, protocols, and control mechanisms, making device integration difficult.
- **Need for Standardized Metadata** : Devices lack a universal way to describe their capabilities and interfaces.
- **Current Approaches** :
  - **WoT TD (Web of Things Thing Description)** : Standardized JSON-LD format to describe IoT devices.
  - **IoT Schema (iot.schema.org)** : Defines common schemas for IoT devices.
  - **SDF (Semantic Definition Format)** : Used in oneDM for modeling device data structures.

Technical Challenges Addressed:

1. Lack of common semantics across IoT platforms.
2. Need for automatic discovery and interoperability.
3. Mapping device properties, actions, and events to standard schemas.

Overview of Web of Things (WoT) Thing Description (TD)

Technical Overview:

- **WoT TD is a W3C Standard** that provides a JSON-LD format to describe:
  - **Properties** (State information, e.g., temperature, brightness).
  - **Actions** (Operations on the device, e.g., turn on/off).
  - **Events** (Asynchronous notifications, e.g., motion detected).

Advantages:

- Uses **JSON-LD** for semantic annotations.
- Enables **cross-platform interoperability**.
- Provides **machine-readable metadata** for automated IoT device discovery and integration.

Example WoT TD JSON Representation:

Copy Edit

```
{
  "@context": "https://www.w3.org/2019/wot/td/v1",
  "id": "urn:dev:ops:32473-WoTLamp-1234",
  "title": "Smart Lamp",
  "properties": {
    "brightness": {
      "type": "integer",
      "minimum": 0,
      "maximum": 100,
      "forms": [{"href": "https://mylamp.example.com/properties/brightness"}]
    }
  },
  "actions": {
    "toggle": {
      "forms": [{"href": "https://mylamp.example.com/actions/toggle"}]
    }
  },
  "events": {
    "overheated": {
      "data": {"type": "boolean"},
      "forms": [{"href": "https://mylamp.example.com/events/overheated"}]
    }
  }
}
```

Explaining WOT-TD JSON

Json-LD format for a thing such as a lamp. Here the properties has a status attribute which is linked to a site which describes what it means. The “actions” key has attribute called “toggle” which provides a link to what it means. Similarly we have events such as overheating.

1. **Security** :
   - Defines **Basic Authentication** ("basic_sc") where credentials are sent in the header.
2. **Properties** :
   - "status": A readable property (e.g., "on" or "off"), accessed via https://mylamp.example.com/status.
3. **Actions** :
   - "toggle": An action that changes the lamp's state (like turning it on/off), accessible at https://mylamp.example.com/toggle.
4. **Events** :
   - "overheating": An event that provides alerts when the lamp overheats. It sends data in **string format** and uses **long polling** (subprotocol: "longpoll") for real-time updates.

EXAMPLE 1: Thing Description sample

```
{
    "@context": "https://www.w3.org/2022/wot/td/v1.1",
    "id": "urn:uuid:0804d572-cce8-422a-bb7c-4412fcd56f06",
    "title": "MyLampThing",
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in": "header"}
    },
    "security": "basic_sc",
    "properties": {
        "status": {
            "type": "string",
            "forms": [{"href": "https://mylamp.example.com/status"}]
        }
    },
    "actions": {
        "toggle": {
            "forms": [{"href": "https://mylamp.example.com/toggle"}]
        }
    },
    "events": {
        "overheating": {
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

Deep Dive into Semantic Annotations in TD

JSON-LD and Ontologies

- JSON-LD allows linking data to external vocabularies.
- **Semantic enrichment** : Each TD can reference terms from ontologies such as **Schema.org, SOSA/SSN, SAREF, IoT Schema, etc.**

```
{
  "@context": {
    "iot": "https://iot.schema.org/",
    "td": "https://www.w3.org/2019/wot/td#"
  },
  "actions": {
    "toggle": {
      "@type": "iot:ToggleAction",
      "forms": [{"href": "https://mylamp.example.com/toggle"}]
    }
  }
}
```

Here in the figure,

- **Iot: ToggleAction** maps to a standard IoT schema concept.
- Enables interoperability with smart home assistants, industrial automation, etc.

Integrating with IoT Schema and Other Ontologies

What is IoT Schema?

- **iot.schema.org** is a structured vocabulary for IoT devices.
- **Used for** :
  - **Device discovery**
  - **Data exchange**
  - **Capability modeling**

Connecting WoT TD to IoT Schema

- Define IoT device actions using standard schema terms.
- Example: A thermostat integrating with iot.schema.org:

```json
{
  "@context": {
    "iot": "https://iot.schema.org/",
    "td": "https://www.w3.org/2019/wot/td#"
  },
  "properties": {
    "temperature": {
      "@type": "iot:TemperatureProperty",
      "forms": [{"href": "https://thermostat.example.com/properties/temperature"}]
    }
  }
}
```

Other Ontologies:

- **SAREF (Smart Appliances REFerence ontology)** → For smart home interoperability.
- **SOSA/SSN (Sensor, Observation, Sample, and Actuator)** → For sensor-based IoT systems.

Comparative Analysis: WoT TD, IoT Schema, and SDF

Feature	WoT TD	IoT Schema	SDF
Data Format	JSON-LD	JSON-LD	JSON
Semantics Support	Yes	Yes	Limited
Action Definitions	Yes	Yes	Yes
Ontology Linking	Yes	Yes	Limited
Industry Adoption	High	Medium	Low

Best Choice?

- **For Interoperability:** WoT TD + IoT Schema.
- **For Lightweight Applications:** SDF.

Mapping IoT Semantics to Capabilities Using Apache Jena

Objective

- We want to connect IoT devices to their functionalities using semantic ontologies.
- Store device capabilities in Apache Jena, enabling structured queries.
- We can use SPARQL and reasoning to infer new relationships between devices.

Key Components of Semantic IoT Modeling

- Ontologies (SSN/SOSA, IoT-Lite, WoT-TD) – Define how devices relate to their capabilities.
- Thing Descriptions (TDs) – JSON-LD metadata describing IoT device properties.
- Apache Jena – Stores IoT semantics in RDF format, enabling reasoning.
- SPARQL Queries – Extract information on available device capabilities.

Step-by-Step Approach to Semantic Mapping

Step 1: Define IoT Semantics Using Ontologies

- We can use SSN/SOSA or IoT-Lite to model devices as sensors, actuators, and properties.
- Example:
  - Temperature Sensor → Detects temperature.
  - Smart AC → Controls and adjusts temperature based on sensor input.

Step 2: Store and Query Data Using SPARQL

- We will store IoT device descriptions in RDF format in the Apache Jena Triple Store.
- We will run SPARQL queries to retrieve devices based on capabilities (e.g., "Find all devices that measure temperature").

Step 3: Apply Reasoning to Infer Relationships

- Use Apache Jena's inference engine to automatically link sensor data to actuators.
- Example:
  - If a temperature sensor exists in a room, the Smart AC can adjust based on its readings.
  - Using ontologies, we can infer which devices should interact even if not explicitly linked.

Step-by-Step Approach to Semantic Mapping

Step 4: Implement Rule-Based Automation with DROOLS

- Use DROOLS rules engine to automate actions based on data.
- Example Rule: If a room temperature > 28°C, turn on the AC.
- Steps Followed to do this:
  - Fetch temperature data from the semantic store.
  - Apply DROOLS rules to determine actions.
  - Execute control actions via IoT APIs.

```
rule "Turn on AC when hot"
when
    $temp : TemperatureReading(value > 28)
then
    System.out.println("Turning on AC...");
    acDevice.turnOn();
end
```

Example Rule which we can write

Conclusion

Key Takeaways

- **WoT TD enables standardized device descriptions.**
- **JSON-LD facilitates semantic linking with ontologies.**
- **IoT Schema and SDF provide structured vocabularies for IoT.**
- **Inference engines (e.g., Drools) can automate decision-making.**
- **Semantic interoperability is the future of IoT automation.**

NOTE :
After this , we did some coding and implemented our approach , which is all documented in our Latex coding report that we have posted on our github repository.

Future Work

Recap of our Current State of our Work :

- Right now, as one can see in the detailed code Report, we have created a 2 way pipeline :
  - where our python server publishes the temperature ,
  - then HiveMQ cloud cluster stores this temperature
  - which is then received by our java client code
  - which then the java code uses to fire Drools Rules, and then store it in Triple store using the turtle format.
  - Once this is done, according to the rules by Drools, heater status of our thermostat is published to our cluster on HiveMQ
  - Which is then again received by our python server and based on the heater status, python server generates the next subsequent temperature.

Challenges and Flaws in our current Approach :

- Currently, our first flaw is that we are only dealing with thermostat temperature, and hence our only action is heater status.
- Therefore, #1 challenge is scalability , where it is difficult to scale the application with multiple properties and multiple actions.

- Therefore , when we need to add any property, we need to change the entire structure of our thermostat class as well as change all the drool rules .

Challenge #2

- Now, even if we have only 1 property we are technically hard coding the rules in our drools file. This is a problem as if we want to keep custom boundary temperatures , that is a current flaw in our system.
- One very practical use-case is when we use AC in the day, we might want to keep our lower threshold temperature at a low value.
- However , at night as it is cool we might want to keep lower threshold temperature a bit higher.

- So, there should a way of communicating this rules between the Drools engine and the user. So , one can set this temperatures without actually changing the configuration default thresholds.

Final Challenge #3 (IMP)

- Now even if we achieve the above system , it still lacks a major functionality. This configuration is very far from the end user. So , our final aim should be to create a mid way system which connects the user and system configuration details.

- Lets understand this in depth with some indepth scenario and its solution :

PROBLEMS :

1. **Static Configuration in Rule Engines (e.g., DROOLS)**
   - Most IoT devices follow fixed rules defined by developers (e.g., heater turns on below 24°C).
   - These rules are hardcoded in logic files, often in rule engines like DROOLS.
   - End-users have no way to modify these settings themselves.
2. **No Dynamic Rule Updates**
   - If a user wants to change the temperature threshold or behavior:
   - They must request a developer to update the rule file.
   - The rule engine must be reloaded or redeployed.
   - This defeats the purpose of smart and adaptive IoT.
3. **No Natural Language Understanding**
   - Systems don't understand phrases like:
     - "Change the minimum temperature to 22."
     - "Turn on the heater only at night."
   - There is no interface to translate user intent into updated behavior.

4. Rule Addition Without Code? Not Really

- Users cannot create new rules through a simple interface.
- Adding a new rule (e.g., “Turn off heater if window is open”) still requires:
  - Writing a new rule manually
  - Understanding syntax and logic
  - Developer-level access

Solution: Mid-Layer for Dynamic, User-Driven Configuration

1. Introduce a Configurable Abstraction Layer

- Store rules and thresholds in a dynamic configuration store (e.g., database, JSON file).
- Rule engine reads these values at runtime instead of using hardcoded ones.

2. Use Natural Language Interfaces

- Let users express preferences in plain language (voice/text).
- Use NLP to parse user input and update configuration automatically.

3. Generate or Modify Rules Dynamically

- Instead of editing rule files manually, use templates or parameterized rules.
- System updates logic based on user-defined config values.

4. Empower the User, Not Just the Developer

- Users can:
  - Change temperature thresholds
  - Toggle behavior settings (eco mode, night mode)
  - Enable/disable specific rules — all without code

Proposed Enhancements to Overcome Identified Challenges

In order to address the two key challenges in our current thermostat simulation workflow—**lack of scalability for multiple properties and actions** , and **rigid hardcoded rules in Drools** —we propose the following architectural and design enhancements.

Challenge #1: Limited Scalability for Multiple Properties and Actions

Problem Recap:
Currently, our system handles only a single property (temperature) and a single actuator response (heater status). Adding a new property (e.g., humidity, CO_2 level) or actuator (e.g., fan, humidifier) would require significant changes in:

- The Thermostat Java class.
- The Drools .drl file structure.
- MQTT topics and payload structure.
- The overall logic and synchronization across the pipeline.

High-Level Solution Overview:

1. **Adopt a Generic Property-Action Model:**
   - Refactor the Java class to use a dynamic data model (Map<String, Object> or similar) to hold multiple properties and their values, rather than hardcoded fields.
   - This allows for runtime extensibility without needing to modify the class structure.

2. **Modular Drools Rules:**
   - Organize Drools rules by domain or device module (e.g., TemperatureRules.drl, HumidityRules.drl, etc.).
   - Use rule attributes or metadata to apply logic selectively based on property type.
3. **Semantic Topic Naming + JSON Payloads:**
   - Follow a consistent topic naming scheme like /iot/property/{name}.
   - Structure MQTT payloads as JSON objects to allow multi-property messages.
4.

```
{
  "temperature": 22.5,
  "humidity": 45,
  "co2": 400
}
```

1. **Dynamic Property-Rule Binding:**
    - Include metadata in the WOT-TD description that defines which rules to apply for which property types.
    - This way, the workflow remains loosely coupled and extensible.

Challenge #2: Hardcoded Rule Thresholds and No User Configuration Support

Problem Recap:
The Drools rules currently use hardcoded temperature thresholds to determine heater status. This means users cannot dynamically set their preferred temperature boundaries for different contexts (e.g., day vs night) without modifying the .drl file and redeploying the Java server.

High-Level Solution Overview:

1. **Parameterize Rules with External Facts:**
    - Introduce a ThresholdConfig fact (Java class) that holds the user-defined boundaries.
    - Inject this fact into the Drools session at runtime, making the thresholds data-driven rather than hardcoded.
2.

```
ThresholdConfig config = new ThresholdConfig();
config.setMinTemp(userMin);
config.setMaxTemp(userMax);
ksession.insert(config);
```

```
rule "Turn on heater"
when
    $t : Temperature(value < config.minTemp)
    $config : ThresholdConfig()
then
    ...
end
```

1. **Add a Configuration API Layer:**
    - Create a REST or MQTT-based API for users to send new configurations dynamically.
    - Store these in a small config store (e.g., Redis or a local file) and load into Drools at runtime

1. **Enable Context-Aware Rule Activation:**
    - Extend the configuration to include time-of-day or user-defined context tags.
    - Drools rules can be activated conditionally based on time ranges or active context labels (e.g., night, day, away).

2. **Use a DSL (Domain-Specific Language) or GUI to Define Rules (Optional Advanced Feature):**
    - Consider integrating a web interface where users define their thresholds and preferences.
    - Translate this into Drools-compatible data (or a DSL that compiles to DRL).

This could be one of the ways to solve both the challenges. It is just a brief overview of what could be done. We haven't thought of it in extreme detail on how to solve it.

This is the conclusion of all our work and research we did for our work in IOT.

Thank you Sir. We enjoyed the R.E and had a great time , working under you sir. We conclude the R.E with this document.