TECHNICAL REVIEW OF FIVE DSLs IN THE DOMAIN OF IoT

F. Morero-Peyrona, J.G. Enríquez, A. Jiménez-Ramírez
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Sevilla
2023-10-16

Introduction

In this technical review, we examine five seminal academic papers in the field of IoT declarative programming languages through the technical papers in which they were presented. Each paper explores a different facet of the role of declarative programming languages in enabling IoT applications. As a result, a systematic summary (where each summary has the same sections) is not possible.

As we delve into these academic papers, it is imperative to understand that the inherent complexity of the IoT requires innovative approaches to programming. Declarative languages, with their focus on abstraction and high-level specifications, have the potential to simplify the development process, improve code readability and promote efficient resource utilization. However, the practical application of these languages in the IoT domain presents both opportunities and challenges.

The purpose of this review is to explain the results of the five academic papers. Each paper's essential features will be outlined to give a thorough comprehension of how these declarative programming languages are shaping the IoT landscape.

These are the papers that have been the subject of analysis:

1. Soic, R., Vukovic, M., & Jezic, G. (2020). Speech controlled IoT system based on context-driven rule engine. In 2020 international conference on innovations in intelligent systems and applications (inista) (pp. 1–7).
2. Amrani, M., Gilson, F., Debieche, A., & Englebert, V. (2017). Towards user-centric DSLs to manage IoT systems. In Model-sward (pp. 569–576)
3. Petnik, J., & Vanus, J. (2018). Design of smart home implementation within IoT with natural language interface. IFAC-PapersOnLine, 51(6), 174–179.
4. Dong, W., Li, B., Guan, G., Cheng, Z., Zhang, J., & Gao, Y. (2020). Tinylink: A holistic system for rapid development of IoT applications. ACM Transactions on Sensor Networks (TOSN), 17(1), 1–29.
5. Gabbrielli, M., Giallorenzo, S., Lanese, I., & Zingaro, S. P. (2018). A language-based approach for interoperability of IoT platforms.

Speech controlled IoT system based on context-driven rule engine

Soic et al 2020

SUMMARY

SYSTEM ARCHITECTURE

The system architecture is designed with the following objectives:

1. Real-time Responsiveness: The system should respond to events from the environment in real-time or near real-time.

2. Custom Event Definition: Users should be able to define custom events based on arbitrary conditions in the environment, enabling automation of tasks.

3. User-Defined Context: Users can define custom metadata in the form of semantic tags, allowing for user-defined context information.

4. Context Sharing: System nodes should be able to share their context with other nodes.

5. Speech Interaction: The system should interact with human users using speech, specifically in the Croatian language.

Architecture Layers:
The system architecture is based on microservices, providing modularity and extensibility. It consists of three layers:

1. Application Layer: This layer serves end-users and includes features such as the user interface, configuration management, statistics, reporting, scheduling, and rule management.

2. Core Service Layer: This layer comprises fundamental system components, including node registration, device and measurement management, rule engine, event processing, etc. It is the core of the framework.

3. Communication Layer: This layer provides generic, configurable services for communication with hardware and software systems, sensors, and actuators. Currently, communication relies on the HTTP protocol, typically using a REST interface. More efficient communication mechanisms like message queues can be introduced where applicable.

Node Profiles:
The system can be deployed with different sets of services on individual devices, allowing for versatile configurations. Two basic node profiles are recognized:

1. Manager Node: The manager node orchestrates monitoring nodes, manages node information, provides data retrieval from monitoring nodes, and handles configuration management, including device and rule management.
2. Monitoring Node: Monitoring nodes communicate with sensors and other devices or systems, collect environmental data, and enable control of equipment where applicable. They are designed to be autonomous and independent, even when their manager node is unavailable.

Both manager and monitoring nodes have local data storage for events and configuration files.

CORE SERVICES
Enable the system to collect data from the environment, control connected devices, and facilitate decision-making and context sharing.

1. Node Registration Service:
- Ensures that all nodes in the network are aware of each other.
- Manages information about each node, including name, location, IP address, description, connectors, and metadata.
- Enables versatile deployment scenarios in different domains.
- Registration involves creating a node descriptor object and invoking the registration method on the manager node, updating the node repository, and notifying other nodes.

2. Measurement Provider Service:
- Provides a mechanism to efficiently retrieve measurement data and registered events from every node.
- Allows data retrieval based on specific user-defined filters, which are processed by monitoring nodes.
- Important for high-level features like statistics and reporting that rely on measurement data.

3. Rule Processing Service:
- Enables the definition of custom rules that can be applied to any node in the system.
- Central component providing automation and context sharing based on user-defined criteria.
- Requires working memory (collection of facts) and a set of rules.
- Rules are evaluated when changes in the working memory occur, enabling advanced decision-making and alarming mechanisms.
- Performs rule validation using a domain-specific language.

4. Measurement Collector Service:
- Manages the registration process of Connector services and receives measurements from them.
- Responsible for updating the working memory in the Rule Processing service, ensuring immediate updates to measurement changes.
- Handles the persistence of received measurements by storing them in the database.

5. Event Processing Service:
- Manages the event flow in the system.
- Receives events created by the Rule Processing service and shares them with the entire system.
- Enables efficient sharing of user-defined context across the distributed system, allowing for system-wide automation.
- Broadcasts user-defined events to all nodes, which may be suitable for systems with a limited number of nodes.

6. Connector Services:
- Independent services that communicate with connected sensors, actuators, and external systems.
- Deliver measurements and status information to the Measurement Collector service.
- Perform measurements according to their configuration.
- Important sources of context information, providing measurements and semantic metadata to the system.
- Must register with the Measurement Collector service to be available to the entire system.

EXAMPLE

DSL Definition
```
#conditions
```

```
[condition]there is a command like “{message}”
    =Event(message==”{message}”)
[condition]there is a message like “{message}”
    =Event(message==”{message}”)
[condition]there is a location like “{location}”
    =Event(location==”{location}”)


#consequences
[consequence]turn on “{connector}“
    =Connectors.set(“{connector}”), “ON”);
[consequence]activate “{connector}“
    =Connectors.set(“{connector}”), “ON”);
[consequence]deactivate “{connector}“
    =Connectors.set(“{connector}”), “OFF”);
[consequence]notify value “{meassurement}“
    =EventFactory.createEvent(“{meassurement}”);
[consequence]report “{message} to “{location}”
    =EventFactory.createEvent(“{message}”, “{location}”);
```

Rules

rule “Humidity reading”
    when
        there is a command like “read humidity”
    then
        notify value “humidity”
end

rule “Activation”
    when
        there is a command like “activate ventilator”
    then
        turn on “ventilator”
end

rule “Missing resource”
    when
        there is a command like “missing steel beams”
        and location like “construction hall”
    then
        report “steel beams to construction hall” to “storage 4”
end

Towards user-centric DSLs to manage IoT systems

Amrani, et al (2017)

SUMMARY

The paper examines the development of a User-Centric Domain-Specific Language (DSL) called IoTDSL for managing IoT (Internet of Things) systems. It addresses the challenges of creating well-calibrated DSLs and the importance of clear separation of concerns and event-driven frameworks in IoT systems.

The paper illustrates the usage of IoTDSL with a smart home scenario, where devices like door detectors, presence detectors, and temperature monitors are controlled based on various rules triggered by events such as door openings, temperature changes, or the presence of smoke.

The key components of IoTDSL are:

1. Type Definition: IoT device types are defined, specifying their capabilities for environment sensing and actuating. These types can be defined by advanced users or come from preexisting device databases. Nodes, including gateways and devices, have capabilities that can capture data from the environment or act on it. Types can be imported and combined for IoT specifications.

2. Network Configuration: This part involves specifying the communication paths between IoT devices. Nodes are instances of defined types and may communicate through predefined communication paths, which define protocols used for interaction. Existing platforms handle protocol details, and network configurations can be automatically established following discovery protocols.

3. Business Rules: The heart of IoT system manipulation, business rules are event-based and implemented using rules and reactions. Rules are triggered when specific expressions become true based on environmental conditions. Reactions define actions, such as switching on lights or maintaining temperature within a range. IoTDSL allows for the use of local or global variables, sequential or parallel composition of reactions, time management for events, and handling interesting non-events.

EXAMPLE

Network Configuration for Alice’s House:

```
1 configuration MyHome {
2    node gw: Central
3    node livingTemp: TemperaturSensor
4    node kitchenTemp: TemperaturSensor
5    node livingHeater: Heater
6    node frontdoor: DoorLock
7    node corridorLight: LightBulb
8    node livingLight: LightBulb
9    node timer: Timer
10   node alarm: Alarm
11   node corridorDetector: PresenceDetector
12   node livingDetector : PresenceDetector
13   node smokeDetector: SmokeDetector
14
```

```
15   from livingTemperature to gw via MQTT
16   from HomeTemperature to gw via DDS
17   from livingHeater to gw via DDS
18   from frontdoor to gw via MQTT
19   from light to gw via MQTT
20   from timer to gw via DDS
21   from alarm to gw via DDS
22   from bodydetector to gw via MQTT
23   from smokeDetector to gw via MQTT
24 }
```

When Alice gets home (and thus opens the front door), she wants the lights in the hallway and the living room to be automatically switched on.

```
1 rule SwitchLightsWhenEnter:
2  when (frontdoor.opened() and after
3      corridorDetector.detected()) do {
4         corridorLight.on() || livingLight.on()
5 }
```

When Alice is in the living room, the temperature inside the room should be maintained between comfortable temperatures; whereas when she is not, the temperature should not drop below 16°C.

```
1 rule PresentInLiving:
2      when (timer.timeout()) do {
3         livingDetector.isPresent() || presenceTimer.set(TIMEDETECT)
4      }
5 rule MonitorLivingTempInStop:
6      when (livingDetector.detected() and livingTemp.getTemp > IN_MAX) do {
7         livingHeater.stop()
8      }
9 rule MonitorLivingTempInStart:
10     when (livingDetector.detected() and livingTemp.getTemp < IN_MIN) do {
11        livingHeater.warm(TIMEHEAT)
12     }
13 rule MonitorLivingTempOutStart:
14     when (timer.timeout() and before livingTemp.getTemp < OUT_MIN) do {
15        livingHeater.warm(TIMEHEAT)
16     }
```

There is a critical fire situation when smoke is detected in the kitchen while temperature is upper 40°C for at least five minutes.

```
1 rule AlarmWhenSmokeAndHighTemp:
2      when (kitchenTemp.getTemp() > 45
3         within 5 min from smokeDetector.smoke()) do {
4            alarm.sound()
5      }
```

Design of smart home implementation within IoT with natural language interface

Petnik y Vanus - 2018

SUMMARY

Authors introduce (design and functional implementation) the Smart Home, which is based on KNX and its integration with several IBM Cloud services (PaaS) and where to deploy applications.

KNX TECHNOLOGY

The text discusses KNX technology, a decentralized system for smart building automation. It supports communication among various building devices like lights, sockets, and sensors. KNX can connect to IoT platforms using IP protocols. Two methods, KNX Web Services and KNX/IP Router, enable external access to KNX devices. Libraries are available to build applications for accessing and controlling KNX devices. These applications run on edge devices connected to the KNX/IP Router via LAN or WLAN.

CLOUD BACKEND SERVICES

The text discusses the use of IBM Cloud and its Platform as a Service (PaaS) model for deploying IoT applications. IBM Cloud is based on open-source technology called Cloud Foundry and offers multiple data centers globally for flexible deployment. It supports various programming languages and offers services from IBM, third-party companies, and communities. IBM Cloud also supports DevOps and CI principles for end-to-end application development and management.

The Internet of Things (IoT) platform is described as an integration layer connecting physical devices, sensors, actuators, and applications. It often involves gateways for devices that can't connect directly. IoT platforms provide APIs for communication, and security is a priority with authentication and encryption. MQTT broker is used for message exchange in the IBM Cloud IoT platform, enabling publish-subscribe communication.

The Message Hub, built on Apache Kafka, is introduced as a solution for handling large volumes of messages and data in a scalable way. It provides low-latency messaging and supports queuing and publish-subscribe delivery models. Apache Kafka also offers stream processing capabilities and connectors to store data in various storage systems for data analytics.

NATURAL LANGUAGE INTERFACE SERVICES

1. Watson Assistant: This service is used to create a chatbot or virtual agent for controlling Smart Homes. It automates interactions with users, answering frequently asked questions, providing information about home states, and controlling actuators in the Smart Home. It requires configuration of Intents (goals expressed in user input), Entities (classes of objects or data types), and the Dialog (decision tree based on intents and entities). The service processes incoming text messages, classifies them into the right category (intent), retrieves entities, selects the appropriate node in the dialog tree, and returns an output text with context.

2. Text to Speech and Speech to Text: These cloud services complement Watson Assistant by enabling natural language interaction in the form of voice. Text to Speech converts text to audio streams, while Speech to Text converts spoken words to text. Both services offer APIs for building applications or streaming interactions. The sequence of calls involves recording voice (1), converting it to text (2), processing it with Watson Assistant (3), generating an output message (4), converting it back to audio (5), and playing it to interact with the user (6).

HUMAN INTERFACE DEVICES

User interaction occurs through various Human Interface Devices (HID). These devices include built-in control panels with custom firmware and UI for direct communication with cloud services or the Cloud Control Center. Tablets and cell phones can also be used, either through hardware-dependent native applications or web browsers displaying the responsive Cloud Control Center.

Regardless of the HID, Smart Home control can be achieved through control panels, chat interfaces similar to human communication, or voice commands. Voice interactions utilize Speech to Text (STT) and Text to Speech (TTS) services, with the main logic implemented through a dialog service.

EXAMPLE

The authors does not provide an example (not even a basic one) and it was impossible to find the software associated with this paper.

Tinylink: A holistic system for rapid development of IoT applications

Dong et al. - 2020

SUMMARY

TinyLink is a comprehensive system designed for the rapid development of IoT (Internet of Things) applications. It allows developers to specify application logic using a C-like language without needing to deal with the intricacies of underlying hardware, drivers, or compilers. Here are the key features of TinyLink:

1. **Automatic Hardware Configuration**: TinyLink takes application code as input and automatically generates hardware configurations tailored to the target hardware platform. It includes a hardware database with information about off-the-shelf hardware components, enabling it to make informed hardware selections.
2. **Consideration of Constraints**: It considers both hardware constraints (based on the hardware database) and user constraints derived from the code. For instance, if the code invokes a GPS sensor, TinyLink understands that a GPS sensor is required.
3. **Optimization**: TinyLink adopts an optimization criterion (e.g., cost minimization) and explores the design space to find the best solution for hardware configurations. It outputs the selected hardware components and their connections.
4. **Arduino-Like Environment**: It provides a familiar Arduino-like environment for application programming and offers unified APIs for interfacing with underlying hardware components. These APIs are implemented on popular mainboards.
5. **Cloud-Based Compilation**: TinyLink uses a cloud-based compilation architecture, making it easier to compile code for various platforms.
6. **Web-Based Interface**: It offers a web-based programming interface for ease of use.
7. **Validation and Benchmarking**: TinyLink includes commonly used benchmarks for system validation.

EXAMPLE

Hardware configuration header file.

```
#include “TL_DeviceID.h”

#define MAINBOARD ARDUINO_UNO
#define WIFI ESP8266_ESP01
#define LIGHT GROVE_LIGHT
#define SOIL_MOISTURE SOIL_MOISTURE_ANALOG

#define WIFI_UART_RX 1
#define WIFI_UART_TX 0
#define LIGHT_ANALOG_OUTPUT 0
#define SOIL_MOISTURE_ANALOG_OUTPUT A2
```

Application code example:

```
void upload();

void setup() {
    TL_WiFi.init();
    TL_WiFi.join("SSID", "PASSWORD");
    TL.Light.setMeassuringRange(1,30000,"LUX");
    TL.Light.setADCResolution(10);
}

void loop() {
    TL.Light.read();
    TL_Soil_Moisture.read();
    upload();
    TL_Time.delayMillis(60000);
}
void upload() {
    double light = TL_Light.data();
    double sm = TL_Soil_Moisture.data();
    String url = "http://hostname/ul.php?";
    url += String("light=") + String(light);
    url += String("&sm=") + String(sm);
    TL_WiFi.get(url);
}
```

To make the source code usable follow these steps:

1. Write Application Code: Create the application code in a hardware-independent manner. Use functions from TinyLink's built-in APIs (prefixed with "TL_") to interact with modules and components.
2. Upload to the Cloud: Upload your code to the TinyLink cloud platform. TinyLink will process the code and perform multiple tasks.
3. Retrieve Outputs: After processing, you receive two outputs:
    - Hardware Configuration: This includes the selected hardware components and their connections. TinyLink provides a visualization of this configuration, making it easy to assemble the hardware platform.
    - Hardware-Dependent Code: TinyLink transforms your hardware-independent code into hardware-dependent binary code, suitable for the target hardware platform.
4. Assemble IoT Platform: Using the visualization of the hardware configuration, assemble the selected components into your IoT platform.
5. Burn Code to Platform: Finally, burn the hardware-dependent code onto the hardware platform to execute the desired functionalities.

A language-based approach for interoperability of IoT platforms

Gabrielli et al. - 2018

SUMMARY

This proposal aims to develop a programming language for the Internet of Things (IoT) built upon the Jolie programming language.

Key points in the proposal:

1. **Leveraging Existing Work**: The proposal begins by acknowledging that it builds upon the groundwork laid in Service-Oriented Architectures (SOAs) and specifically leverages the Jolie programming language. The syntax and abstraction mechanisms in the IoT application example provided are based on Jolie.
2. **Jolie's Strengths**: Jolie enforces a separation between behavior and deployment, which is advantageous for IoT development. It offers communication primitives, structured programming constructs, and concurrency support.
3. **Communication Modalities**: Jolie supports various communication media and data protocols in a uniform manner. It originally supports common communication media like TCP/IP sockets and data protocols like HTTP and JSON-RPC.
4. **IoT-Specific Challenges**: However, IoT technologies like CoAP and MQTT pose unique challenges. CoAP, based on UDP, can be unreliable, and MQTT follows a publish-subscribe paradigm, different from Jolie's point-to-point communication. The proposal addresses these challenges by integrating CoAP and MQTT into Jolie.
5. **Flexibility for IoT**: Jolie's flexibility enables support for and interconnection of multiple IoT islands. It can adapt to different communication protocols based on internal or environmental conditions.
6. **Edge Device Considerations**: The proposal omits modeling IoT edge devices like Arduino, typically programmed in low-level languages due to hardware constraints. While it suggests that Jolie-like languages could be extended for edge devices, it acknowledges the challenges and engineering effort involved.

There is a web page associated with the language presented in this paper:

http://www.cs.unibo.it/projects/jolie/jiot.html

Extensive documentation and examples of source code can be found there.

EXAMPLE

Scenario: A controller, programmed in Jolie, communicates with one of many thermostats in a home automation context. In the automation system, thermostats are accessible at the generic address `"coap://thermostat/##"` where `"##"` is a two-digit number representing the identifier of a specific device. Thermostats accept two interactions: a GET request on URI `"coap://thermostat/##/getTemperature"`, that returns the current temperature sensed by the thermostat, and a POST request on URI `"coap://thermostat/##/setTemperature"`, that sets the temperature of the HVAC system controlled by the thermostat.

```
type TmpType: void { .id: string } | int { .id: string }

interface ThermostatInterface {
  RequestResponse: getTmp( TmpType )( int )
  OneWay: setTmp( TmpType )
}

outputPort Thermostat {
  Location: "datagram://thermostat:5683"
  Protocol: coap {
    .osc.getTmp << {
      .messageType = "CON",
      .messageCode = "GET",
      .contentFormat = "text/plain",
      .alias = "/%!{id}/getTemperature"
    };

    .osc.setTmp << {
      .messageType = "CON",
      .messageCode = "POST",
      .alias = "/%!{id}/setTemperature"
    }
  }
  Interfaces: ThermostatInterface
}

main {
  getTmp@Thermostat( { .id = "42" } )( temp );
  if ( temp > 27 ){
    setTmp@Device( 24 { .id = "42" } )
  } else if ( temp < 15 ){
    setTmp@Device( 22 { .id = "42" } )
  }
}
```