Introduction

About Rule conflict

Need for Rule conflict

Demo 1 – Rule Conflict using Interval tree

Implementation details

Database schema

Output

Shortcomings

Demo 2 – Rule Conflict implementing algorithm in referred paper

Modification

Classification

Input format

Database schema

Explanation of each conflict

Output

Conclusion

References

Appendix

Project structure

Dependent libraries

Introduction

Internet connected and interconnected devices, collectively referred to as Internet of Things (IoT) are becoming reliable means to automate daily activities for people and organizations. This interconnection among devices and web services requires a need for representing and managing interactions between them.

In traditional systems, policies are typically used to govern these interactions. However, most of these systems are static in nature when compared with IoT systems. In IoT, devices act with respect to context and how they have been configured. Thus, efficient tooling and framework are required for governing such heterogenous systems.

Rule Conflicts

A key way in which the programming of IoT systems can become unsafe is through conflicts which we refer it as Rule Conflict.

1. Conflicts can emerge when two or more instructions given to IoT devices cannot be satisfied simultaneously. A simple but practical example of this is when two instructions are provided to a single device simultaneously, both of which cannot be executed. For instance, a single light-bulb may have two simple rules provided to it – one that requires it to be turned on during evening hours, and other that requires it to be turned off when no one is in the room. Conflicting IoT programs can occur with a single user who perhaps does not realize instructions can conflict. Or through multiple users who encode opposing preferences.
2. Conflicts can arise between an app rule and a predefined policy. For example, turning on a light based on time can violate an energy-conserving policy that turns off light based on room occupancy. In these two cases, what is required are automated methods to highlight to users when such situations arise before they become a problem.

Programming Goal:

Key source of complexity for IoT applications include a significant amount of event-based (for example, context driven) logic that is well known to be error prone. This problem does not disappear even if the building blocks of programming model are simple as the logic that needs to be encoded does not change. The present implementation of Aquaponics does not have a conflict checking mechanism for detecting rule conflicts and providing feedback to the user. The challenging part is to accurately detect conflicts and provide feedback to the users when the size and complexity of IoT systems increase. The Overview of rule detection mechanism is shown below.



Figure 1 Overview

Interval Tree Implementation

Brief Overview of Interval Trees

An interval tree is a tree data structure to hold intervals. Specifically, it allows one to efficiently find all intervals that overlap with any given interval or point. The key to maintaining an interval search tree is to store each interval in a balanced binary search tree, sorted by the left endpoint. In addition, we maintain some auxiliary information in each node x, namely the maximum value of any (right) endpoint stored in the subtree rooted at x. If two intervals with identical endpoints are inserted, we only maintain one copy. The reader is requested to refer the internet for further description of interval trees.

Data Model

The data model used here is a combination of action and actuator. The action represents the condition that must be executed on an actuator. It captures the result of execution of a Rule. The idea here is to map interval tree for each rule. We use the existing Aquaponics database schema for implementation.

Implementation Details

The algorithm started by iterating through the rules in the database. Each rule record contains the necessary information about trigger condition and action. Based on the action and actuator each rule will be mapped to an Interval Tree. The rule conflict detected starts by inserting the present rule interval in the Interval Tree. A Rule Conflict exception is raised if the intervals intersect. If none of the intervals in the Interval Tree intersect then a new Interval Tree node is created and additional information for the nodes are updated.

The above algorithm repeats for opposite actions. As the next state of one rule may conflict with the current state of some other rule. The above process continues similarly for all other rules in the table. Please note that the algorithm is invoked whenever the user adds a new rule. The result of the execution is notified back to the user.

Pseudo code

Rule**:**

String action

String actuator

HashMap**<**Rule**,** Interval Tree**>** map

**for** Rule **:** Database

Evaluate expression using Expression Evaluator

**if** expression is not valid

**throw** Exception

Action **->** addRule**(**Rule**)**

Opposite Action **->** addRule**(**Rule**)**

addRule**(**Rule**):**

**if** map**.**get**(**Rule**)** is Empty**:**

Create **new** Interval Tree

add Interval **for** Rule

**else**

Interval Tree **=** map**.**get**(**Rule**)**

**for** node **:** Tree**:**

Check **for** Interval intersection

**if** intersect

**throw** Rule Conflict Exception

**while** Iterating back**:**

update Interval Node auxiliary information

map**.**add**(**Rule**)**