INDIVIDUAL INSTRUCTION IN COMPUTER ENGINEERING

CE 8V40.050

INDEPENDENT STUDY REPORT

Submitted by:

Sriee Gowthem Raaj Ammapet Sathiiss

Net Id: sxa156930

UTD Id: 2021259605

Table of Contents

Table of Figures

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 Aquafonics 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.

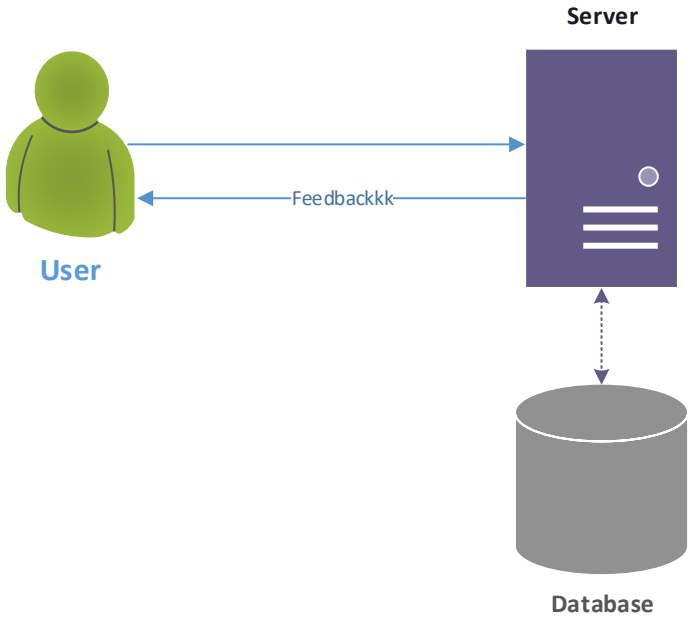

Figure 1 Overview

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 Aquafonics database schema for implementation. The interaction between the program and the Database is via JDBC / ODBC drivers and SQL queries for transactions

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.

Output:

The figure shown below raises rule conflict exception for two rules [504 & 501] which act on the same actuator with conflicting actions.

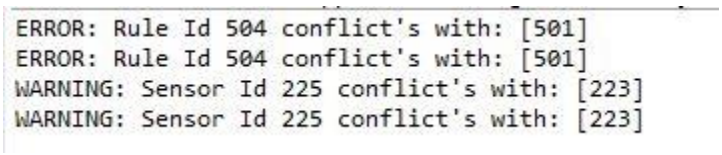

Figure 2 Demo 1 output

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)
```

Disadvantages:

The above implementation works very well for a rule with one condition and one action. It can catch rule conflicts with dependencies but doesn't support complex rules with m condition and n action. To support the Aquafonics complex rule engine a new framework is required.

Implementation based on Research Paper

The reader is requested to refer Yan Sun et al. [1] for implementation of Conflict Detection Algorithm. This implementation mimics their algorithm with some modification. The database schema is rewritten to support the new framework. We won't be using Aquafonics database schema. The overview of this implementation is shown below

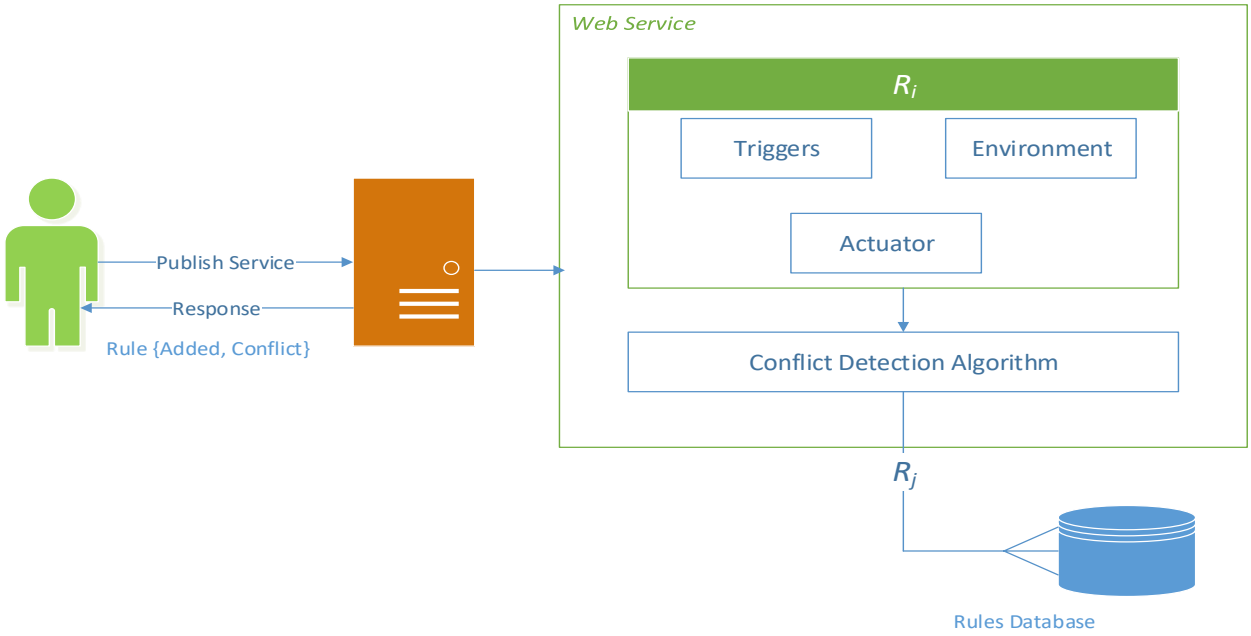

Figure 3 Implementation 2 Overview

Modification

- The implementation contains a major adoption of the research paper with few important changes. We have neglected the User entity in UTEA rule model. We are assuming that a single user will be using the system. This eliminates the construction of authority tree and some of the condition checks in conflict detection algorithms
- Also, this implementation has removed the priority logic for triggers and environments but the framework has the required parameters for the logic to be implemented in the future. At present the default priority for triggers and environments are *-1*.
- The format used for communicating between the subscriber and publisher (in other words user interface and web services) are in XML format. We are using JSON format and have modified its contents.

- The paper considers *'location'* as one of the parameter while checking the conflicts. We haven't specified a **location tree** where the rules can be checked against only the rules which are associated with a location node.

This implementation has removed few of the conflict classification to synchronize with the modification discussed above. The figure below shows the supported type of rule relation and classification of conflicts

Relation	Abbreviation	Condition
Similar trigger	$\text{SimlTr}(R_A, R_B)$	$\text{Trigger\_ID}_{A=B} \wedge \text{Event}_{A \subseteq B} \wedge \text{Priority}_{A=B}$
Similar action	$\text{SimlAc}(R_A, R_B)$	$\text{Actuator\_ID}_{A=B} \wedge \text{Action}_{A \subseteq B}$
Similar prestate	$\text{SimlPr}(R_A, R_B)$	$\text{E\_ID}_{A=B} \wedge \text{Pre\_s}_{A \subseteq B}$
Contrary poststate	$\text{ContPs}(R_A, R_B)$	$\text{E\_ID}_{A=B} \wedge \text{Next\_s}_{A \neq B}$
Explicitly dependence	$\text{ExplDepe}(R_A, R_B)$	$(\text{Actuator\_ID}_A = \text{Trigger\_ID}_B) \wedge (\text{Action}_A \supseteq \text{Event}_B)$
Implicitly dependence	$\text{ImplDepe}(R_A, R_B)$	$(\text{E\_ID}_A \supseteq \text{Trigger\_ID}_B) \wedge (\text{Next\_s}_A \supseteq \text{Event}_B)$
Negative action	$\text{NegiAc}(R_A, R_B)$	$\text{Actuator\_ID}_{A=B} \wedge \text{Action}_{A \neq B}$
Trigger event	$\text{TrigEve}(R_A, R_B)$	$\text{E\_ID}_{A=B} \wedge (\text{Pre\_s}_A \subseteq \text{Next\_s}_B)$

Figure 4 Rule Relation

Conflict	Condition
Execution Conflict	$(\text{SimlTr}(R_A, R_B) \vee \text{TrigEve}(R_A, R_B)) \wedge \text{NegiAc}(R_A, R_B)$
Environment Mutual Conflict	$\text{SimlTr}(R_A, R_B) \wedge \text{ContPs}(R_A, R_B)$
Direct Dependence Conflict	$(\text{ExplDepe}(R_A, R_B) \vee \text{ImplDepe}(R_A, R_B)) \wedge (\text{ExplDepe}(R_B, R_A) \vee \text{ImplDepe}(R_B, R_A))$
Indirect Dependence Conflict	$(\text{ExplDepe}(R_A, R_B) \vee \text{ImplDepe}(R_A, R_B)) \wedge (\text{ExplDepe}(R_B, R_C) \vee \text{ImplDepe}(R_B, R_C)) \cdots \wedge (\text{ExplDepe}(R_N, R_A) \vee \text{ImplDepe}(R_N, R_A))$

Figure 5 Conflict Classification

Input format:

The input for the algorithm will be in JSON format. JSON format is chosen because of it was the format used in Aquafonics to transfer data between IOT device and the REST controller.

- *condition_expression* - Holds the string format of the Boolean expression for condition. The expressions may be complex
- *action_expression* - Holds the string format of the Boolean expression for actions. The expression may be complex
- *condition_count* - Total number of condition in the expression

- *action_count* - Total number of action in the expression

Followed by the parameters required for each condition and action. Please note that the JSON files are located at resource folder for demo purpose. The JSON data should be sent by the IoT device to the web server.

```
{
    "condition_expression": "c1 & (c2 | (c3 & c4))",
    "action_expression": "a1 & a2",
    "condition_count": 4,
    "action_count": 2,
    "c1": {
        "name": "H",
        "id": "10",
        "operator": "<",
        "value": 40
    },
    "c2": {...},
    "c3": {...},
    "c4": {...},
    "a1": {
        "name": "Window",
        "id": "2",
        "operator": "=",
        "value": 0
    },
    "a2": {...}
}
```

Database schema

Hibernate Object Relational Mapping (ORM) is used for handling SQL Transactions. Entities were configured using hibernate annotations. Helper shell scripts for initializing the database, loading the contents and purging the database is present in the *resources* folder in the project. Store the database configuration in *".database.cnf"* file in the same folder where you run the shell scripts

- Run *./init.sh* – To initialize the database, create the tables and load the values into the database
- Run *./purge.sh* – To delete the database and its contents
- To add contents to database while initializing populate the data in *"loal.sql"*

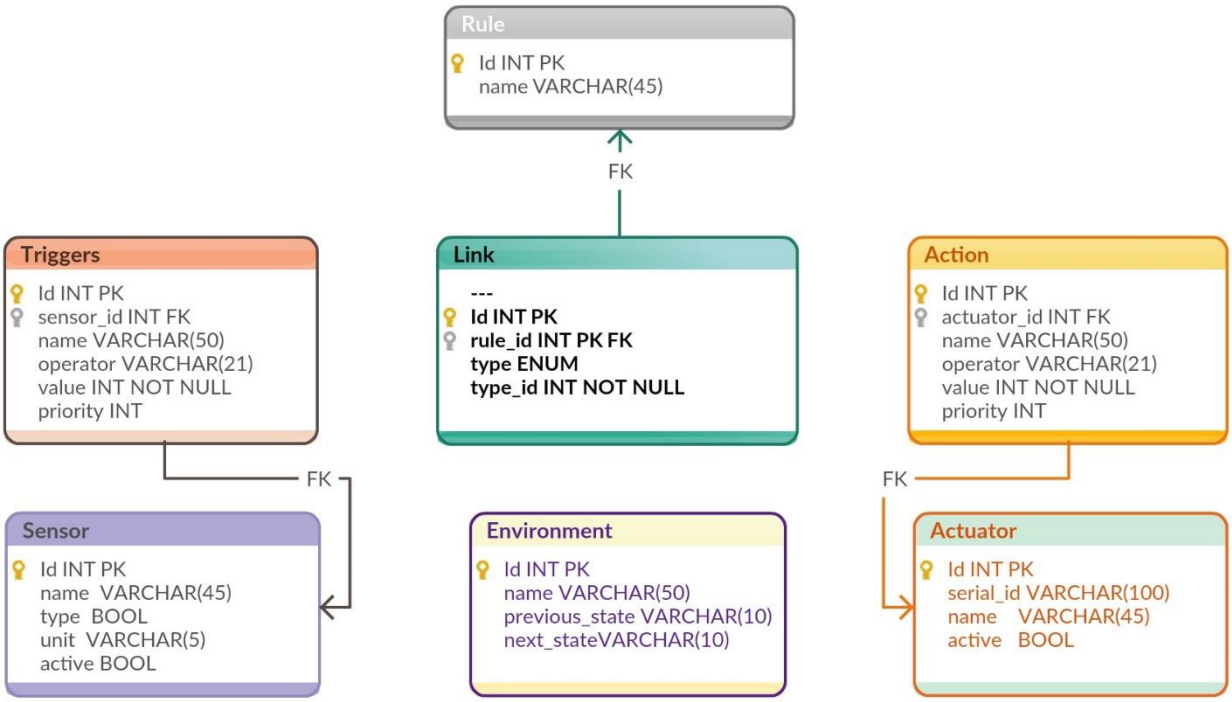

Figure 6 Implementation 2 Database Schema

Pseudo code

```
for expression : Json {
    List<> ContainerList
    for lhs : ContainerList:
        for rhs : Database:
            ruleRelation = lhs, rhs
            throw RuleConflictException
    add(container)
}
Build(Json){
    List<> ruleTokens
    List<> container

    DeserializeJsonContent
    if expression == simple:
        format string
        ruleTokens.add(formatted string)
    else:
        Build Abstract Syntax Tree
        Build expression to a format supported by jboolExpression library
        dnf = expression.toDNF()
        ruleTokens.add(dnf)

    for token : ruleTokens:
        wrappedToken = wrap(token)
        container.add(wrappedToken)
}
```

Future Work

- This implementation is stand alone. It should be integrated with Aquafonics web controller module. Although majority of the work has been done, changes with respect to integration has to be done
- The implementation could be stress tested for multiple complex rules. Test suite could be created to test specific modules in the implementation

Conclusion

The project started with studying and understanding the Aquafonics framework. Complex rule engine feature which handles the functionality of converting user inputs into Boolean expression was analyzed. The notion of rule conflict was introduced and scenarios where users could cause these conflicts has been studied. To support IoT heterogenous system and context driven rule configuration an efficient framework is required to detect conflicts and provide feedback to the users are required. Two methods were implemented and tested for scenarios pertaining to Aquafonics.

References:

- Yan Sun, et al., “Conflict Detection Scheme Based on Formal Rule Model for Smart Building Systems” *IEEE Transactions on Human-machine Systems*, vol. 45, no. 2, Apr. 2015.

Appendix

Dependent libraries

The project requires the following dependent libraries. Libraries are found in the projects ***lib*** folder

Library name	*Comments*
▪ *antlr-runtime-3.3* ▪ *cdi-api-1.1* ▪ *classmate-1.3.0* ▪ *dom4j-1.6.1* ▪ *el-api-2.2* ▪ *geronimo-jta_1.1_spec-1.1.1* ▪ *hibernate-commons-annotations-5.0.1* ▪ *hibernate-core-5.2.4* ▪ *hibernate-jpa-2.1-api-1.0.0* ▪ *jandex-2.0.0* ▪ *javassist-3.20.0-GA* ▪ *javax.inject-1* ▪ *jboss-interceptors-api_1.1_spec-1.0.0.Beta1* ▪ *jboss-logging-3.3.0.Final* ▪ *jsr250-api-1.0*	Hibernate
▪ *jbool_expressions-1.4* ▪ *antlr-2.7.7* ▪ *commons-lang-2.5* ▪ *guava-14.0-rc3*	Disjunctive Normal Form
▪ *gson-2.8.2*	MySQL Database
▪ *mysql-connector-java-5.1.33-bin*	JSON Processing

Project Structure

Package Name	*Functionality*
com.entity	Entity POJO's for Hibernate mapping
com.exception	Rule Conflict Exception
com.json	Deserialize JSON
com.logger	Logging module
com.parser	Parse expression to Objects
com.rule	Rule conflict algorithm