HomePort

An extension to allow automation of smart devices on heterogeneous networks

Aalborg University
Department of Computer Science
Master Thesis

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
Selma Lagerlöfsvej 300
Telephone: +45 99 40 99 40
Telefax: +45 99 40 97 98
http://cs.aau.dk

Title:

HomePort – An extension to allow automation of smart devices on heterogeneous networks

Subject:

Home Automation

Project period:
Master Thesis,
Spring semester 2015

Project group:
DES101F15

Attendees:
Brian Holbech
Christian Mortensen
Søren Knudsen

Main advisors:
Arne Skou
Brian Nielsen

Edition: 1.0

Number of pages: 95

Appendix pages: 25

Finished: 9/6-2015

Synopsis:

In the field of Home Automation, a large number of vendor-specific protocols makes it difficult for end-users to control their smart devices in a uniform manner. The HomePort project attempts to solve this problem by providing a platform that enables end-users to uniformly access all of the connected devices. Currently, there are no means in HomePort for automating these smart devices.
In this report, a language for automating smart devices in HomePort is presented, as well as a language which can be used to specify unwanted behavior. In accordance to these languages, a prototype implementation is integrated into HomePort, which makes it possible for users to utilize the languages for automation of smart devices and for detecting unwanted behavior in the system.
Based on a set of scenarios presented throughout the report, automation and safety rules are created and their behavior is observed in order to evaluate the prototype implementation.
Lastly, a conclusion of the project is drawn and suggested improvements to the system are discussed.

Brian Holbech

Christian Mortensen

Søren Knudsen

Preface

This master thesis is written by three software engineering students during the spring semester of 2015. The overall purpose of the project is to extend HomePort with functionality to enable automation of smart devices, and to create a system that detects unwanted behavior.

References to the bibliography is indicated by the use of [ ]. For example, [1] references to the first entry in the bibliography. Moreover, "..." is used in code examples when code has been omitted.

We would like to thank Brian Nielsen and Arne Skou for supervision throughout the project period.

Contents

Introduction

More and more types of smart devices are currently being made available. Many of these devices can be controlled remotely using smart phones, and some vendors also support automation to some degree. These smart devices make it possible to automate many tasks. For example, a home automation system could automatically adjust radiators and open and close windows in order to ensure a healthy indoor climate.

The problem with these smart devices, however, is that cooperation is often limited to a range of products from the same vendor. This means that if the thermostats and windows from the previous example are made by different vendors, the devices may be incompatible, and the desired cooperation may not be possible.

In this project we will look at the home automation system HomePort, a home automation system developed at Aalborg University, which already supports integration of smart devices located on heterogeneous networks and investigate how it can be extended to allow device automation across these networks.

Initiating Problem

HomePort is a home automation platform currently used in a research context. The purpose of the system is to provide a uniform interface to smart devices on heterogeneous subnetworks and present these devices through a REST interface. This is done by allowing third-parties to implement plug-ins that are capable of translating device specific protocols into a uniform protocol understood by HomePort.

Currently, HomePort only offers limited support for interaction between devices across heterogeneous networks. This is not optimal, since the basic goal of Home Automation is to provide improved convenience and comfort by combining functionality from various devices.

How can HomePort be extended in a way that allows devices on different subnetworks to be utilized for automation? Which changes must be made to the existing HomePort architecture in order to facilitate the extension?

1 | Analysis

In this chapter, the current state of HomePort is described and related work regarding home automation systems that implement device automation in different ways are analyzed. Based on this related work a generalized architecture for home automation is proposed and four challenges that we find central in regard to extending HomePort with functionality for device automation is presented. Finally, we identify problems that should be solved in this project and present a list of requirements.

1.1 HomePort

HomePort is a system that attempts to aggregate smart devices from different vendors and present these devices through a REST interface to allow uniform control of these devices regardless of their underlying protocol. Developers can add support for smart devices located on different subnetworks by writing adapters that translate the specific subnetwork protocol into something that HomePort can understand. A subnetwork is, in this context, a network of devices with the same underlying protocol. E.g a network where all devices use the ZigBee[1] protocol.

In the latest version of HomePort, described in [6], we added support for dynamic reconfiguration of adapters through the use of plugins isolated in separate processes. Like previous versions, HomePort is developed in the C programming language and is intended to run on a Linux distribution. The resulting architecture of the project can be seen in Figure 1.1.

The HomePort Core is the main entry point of the system and is responsible for maintaining the state of the configuration and starting the Webserver and Plugin Manager.

The Configuration is a data structure containing abstractions for plugins and adapters. Figure 1.2 shows the interdependencies between data structures in the configuration. An adapter is an abstraction of a subnetwork and can contain a number of devices belonging to the specific subnetwork. Each device can have a number of services that can be used to control the device. A service must either be an actuator (read/write) or a sensor (read-only). Plugin data structures contain information about currently installed plugins.

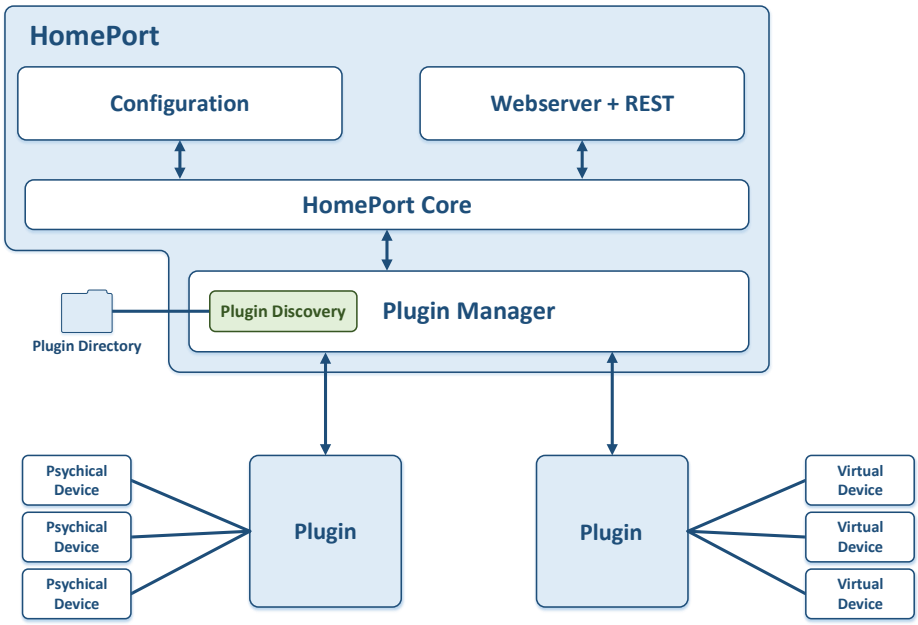


Figure 1.1: Component architecture of HomePort

The Webserver exposes a REST interface, which contains methods to get and set the state of plugins and devices. Plugins can be started and stopped and devices can be manipulated by calling one of its actuator services with appropriate parameters using a PUT request. Furthermore the state of plugins, devices and services can be retrieved through GET requests.

The Plugin Manager is responsible for discovering plugins and announcing these to the HomePort Core. When a new plugin is discovered the Plugin Manager creates a data structure instance for it, passes this data structure to the HomePort Core and attempts to start the plugin. It is also the job of the Plugin Manager to facilitate communication between the HomePort Core and individual plugins.

Plugins are used to implement code that translates subnetwork specific protocols into something HomePort can understand. Plugins can do this by using a Plugin API that can add, remove and update data structures in HomePort's configuration. Because plugins are isolated in separate processes the API uses IPC to communicate with the Plugin Manager. This communication is bi-directional and is also used to facilitate requests made through the REST interface to functions implemented in specific plugins.

The architecture in Figure 1.1 allows new adapters and devices to be added to HomePort through the use of plugins, but lacks a proper way of interconnecting them if they are located on different subnetworks. Currently, the only way of achieving interconnectivity between devices on different sub-

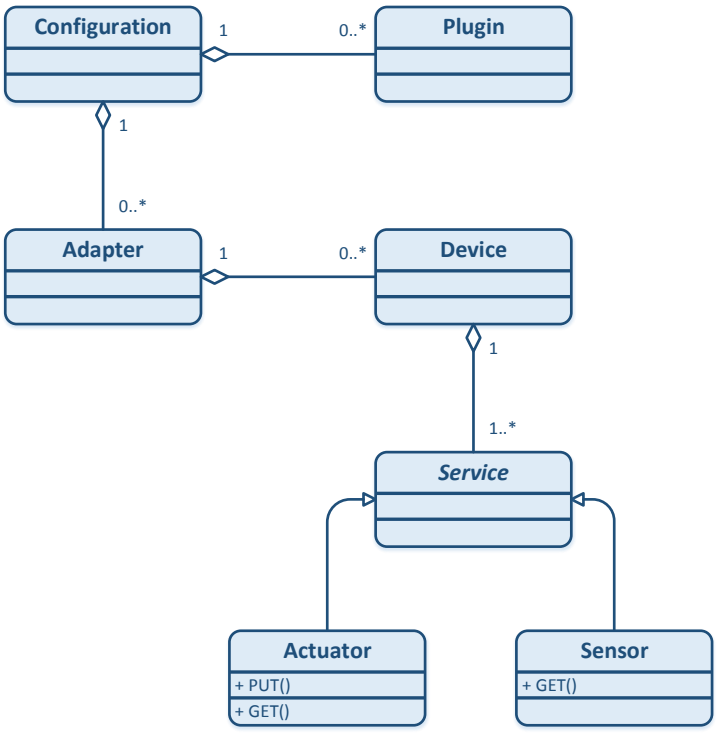


Figure 1.2: UML diagram of HomePort data structure interdependencies

networks is through the REST interface and requires that the logic of such interconnectivity is implemented across the different plugins.

1.2 Related Work

In this section, three home automation systems that showcase different approaches for solving automation tasks are described. BOSS and HomeOS rely on abstractions and application development for achieving this, while RuCAS relies on event-condition-action rules created by the end-user.

1.2.1 HomeOS

HomeOS[3] is a platform that presents users and developers with a PC-like abstraction for technology in the home. This means that network devices regardless of their underlying protocols are presented as peripherals with abstract interfaces and developers can use these interfaces to write applications that enable cross-device tasks. The platform also gives users a management interface for installing new applications, controlling access rights management, etc.

Figure 1.3 shows the different layers in the HomeOS architecture. The device connectivity layer and device functionality layer are used for discovering devices on different subnetworks and translating these devices into services that can be used by developers. Services have roles and each role contains a list of operations that enables interaction with the underlying device. A "lamp" service can for instance have the role "lightswitch" with the operations "lightOn" and "lightOff". It is also possible for a service to have more than one role.

HomeOS enables devices to interact with each other through developer written applications running in the application layer. Applications are written against standardized APIs that contain functionality for subscribing to notifications and invoking operations on existing services. Developers can leverage these APIs to create applications that perform different automation tasks. If for instance two services with a "lightswitch" role and one service with a "sensor" role existed in the system, an application could invoke "lightOn" and "lightOff" on the two "lightswitch" roles, depending on the value returned by a "motionDetected" operation on the "sensor" service.

Applications must provide a manifest file that lists mandatory and optional roles. This way the application layer can determine if an application is compatible with the home and tell the user if services are missing.

The management layer provides features for adding and removing applications as well as other management tasks. When adding a new application in HomeOS, the management layer controls that no conflicting accesses to a device exists. A conflict can occur if two different applications have access to the same device during the same time period. If a conflict is found the

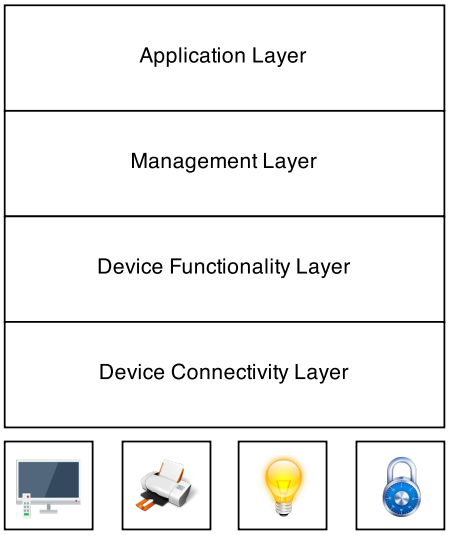


Figure 1.3: Overview of the HomeOS architecture

user is notified and must decide which application should be given the higher priority.

It is also in the management layer that new devices are configured. When a new device is registered, the user is prompted to choose a location for the device. Knowing where a group of devices is located in the home makes it possible to specify policies for a group of devices instead of individual devices. Device groups are arranged in a tree hierarchy, which allow groups to be contained in other groups. E.g. a lamp can be located in the living room and the living room can be located indoors.

HomeOS does not provide end-users with the possibility of manually creating rules for controlling automation, but relies purely on application development. If a user wants to turn on the lights in the hallway when the front door is unlocked, an application that supports this must be installed.

1.2.2 BOSS

Building Operating System Services (BOSS)[2], is a collection of services that make up a distributed operating system for controlling smart devices. By using these services, control applications can access and control devices in an abstract manner, while also having well defined fault tolerance policies. Generally, these control applications could be either automatic systems that react to the environment using a set of rules, UI applications that allow users to manually override settings in the system, or a combination of both.

Figure 1.4 shows an overview of the architecture used in BOSS. The Hardware Presentation Layer (1), the Hardware Abstraction Layer (2), and

the Time Series Service (3) all serve to create abstractions of physical devices. Devices can be accessed through the HAL using an approximate query language that allows control applications to query devices based on their relationship to other items in the building (e.g. "lights in living room"), rather than having to use the specific address or name of a device. The values of these devices can then be accessed in the TSS, which contains a database of both current and historical device readings. The HAL's query language serves to create an abstraction over devices in order to hide their physical address (e.g. an IP address). Because the control applications use this abstraction when applying actions to devices, dynamic addition and removal of devices in the system is handled seamlessly at the highest level. The categorization of devices that must happen in the HAL is, however, not trivial. According to the article[2], this problem is not solved, and the mapping of devices to types and locations is currently very manual.

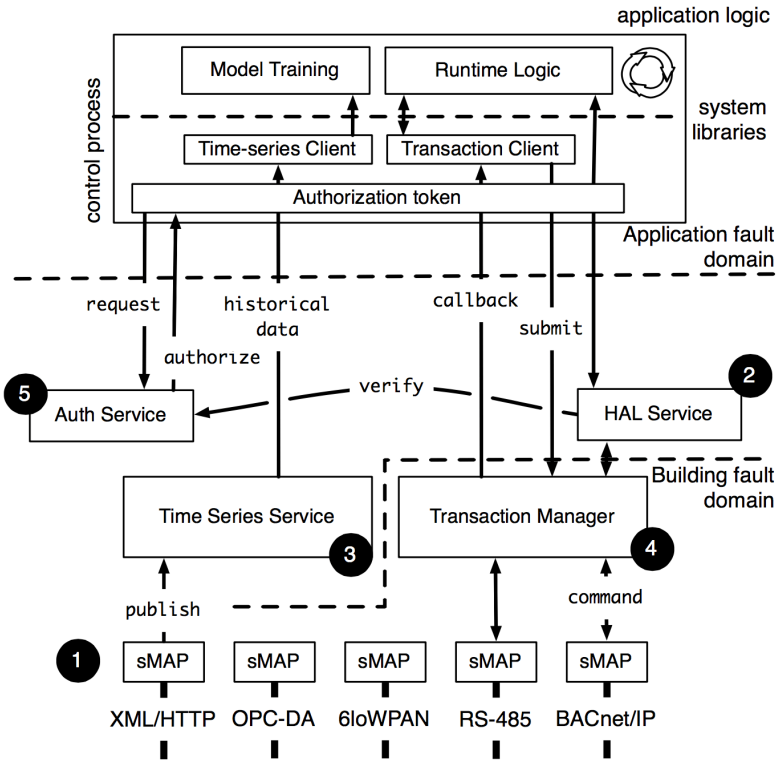


Figure 1.4: Overview of the BOSS architecture

When a control application wants to control a device, it must do so by queuing a transaction in the Transaction Manager (4). A transaction contains a number of *actions* that should be applied to a number of devices, as well as a *lease time*, a *revert sequence*, and an *error policy*. First, the trans-

action manager checks whether the devices in question are already leased out to another control application. If this is the case, the transaction is only attempted if it has a higher priority. In case a transaction fails to fully execute all of its actions, the chosen error policy will be applied.

When a transaction's lease time is up, the state of the affected devices is rolled back using the revert sequence, and control of the devices is yielded to the transaction with the next highest priority. If no such transaction exists, a preprogrammed default behavior takes control of the devices.

Lastly, the Authorization Service (5) allows an administrator to restrict access to physical resources. These restrictions apply in terms of the approximate query language, so, for example, a user could be restricted access to a certain type of device in a specific part of the building.

1.2.3 RuCAS

RuCAS is a framework which makes it possible to create context-aware services [7]. A context-aware service is able to recognize a real-world context, and behave autonomously based on this context. All the context-aware services in RuCAS are described as ECA (Event-Condition-Action) rules. The architecture of RuCAS is shown in Figure 1.5, and is built on top of another framework called Sensor Service Framework (SSF), which presents physical devices as web services.

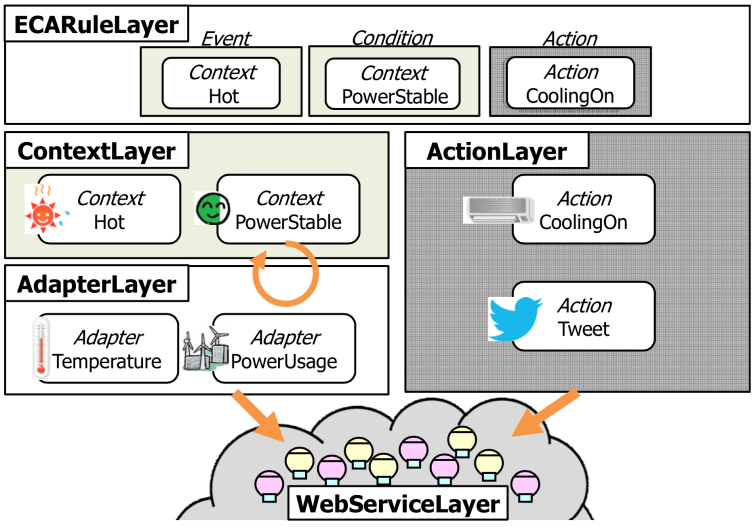


Figure 1.5: The architecture of RuCAS

The RuCAS architecture consists of several layers, which together makes it possible to create context-aware services. The first layer in RuCAS is the adapter layer, which manages all the adapters in the system. An adapter in RuCAS is a wrapper around a web service for accessing it uniformly. Similar,

the Action layer in RuCAS manages all the actions in the system. An action in RuCAS is also a wrapper around a web service e.g. the action `LightOn` could map to the web service `http://rucas/lamp/on`.

The context layer manages all the contexts created in RuCAS. A single context in RuCAS is called an atomic context, and it consists of a context expression which is a logical formula where a true or false value can be deduced e.g. `adapter1.value >= 25`. If a single atomic context lacks expressiveness, several contexts can be combined using logical operators (`AND`, `OR`) which forms a compound context. Each atomic context can be configured to be periodically evaluated, and when an evaluation is triggered the context will fetch a new value from the adapter and evaluate itself.

The contexts are the main building blocks for creating rules in RuCAS, and it is used in the event and condition part of the rule. The event part consists of a single context, which is a trigger for evaluating the rest of the rule. The condition part can consist of one or more contexts, which acts as a guard condition(s). When both the event and condition clauses are true, the chosen action(s) will be executed. RuCAS exposes a REST API which makes it possible to add new elements to the layers depicted in Figure 1.5, and uses these element to build an ECA rule.

1.3 Generalized Architecture

The three home automation systems described above, although different, have some architectural commonalities. Figure 1.6 shows a proposed multilayered architecture that encapsulates the three solutions in generalized terms.

At the lowest tier are a number of *Devices*. These devices may be connected through subnetworks and use arbitrary protocols for communication. Heterogeneously connected devices are the premise of the problem of creating a general solution for home automation. An *Adapter Layer* facilitates device discovery, as well as communication over the heterogeneous subnetworks using device specific protocols. Beyond this point, devices can be accessed in a uniform manner by the rest of the system.

Third is a *Device Abstraction Layer*. This layer serves to create meaningful abstractions over the connected devices, in an attempt to help facilitate high-level control over the system. For example, devices could be categorized by their type (e.g. lamp or television), or by location (e.g. living room or kitchen). In BOSS this layer consists of the *HPL* and *TSS* that allow users to access devices and their values using a query language. RuCAS allows the creation of *contexts* – a concept used to allow users to define and reference aggregations of several devices when creating rules. In HomeOS devices are assigned *roles* which serve as abstract definitions of the capabilities of the device.

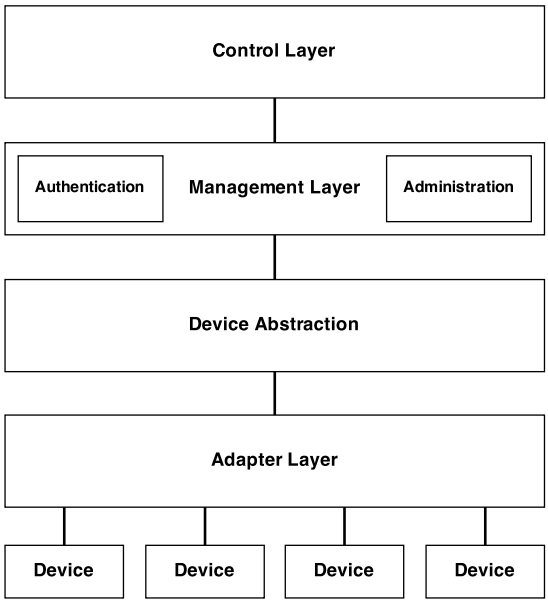


Figure 1.6: Generalized multilayered architecture for home automation systems

Above the device abstraction layer is the *Management Layer*. The purpose of this layer is to validate actions before they are sent to the physical devices. This validation can consist of several processes, such as *authentication*, *conflict detection*, and *prioritization*. In both BOSS and HomeOS, this layer contains a UI which allows an administrator to set up access rights and priorities for the various applications that control the system. In HomeOS, the Management Layer is also used for installation of applications, as well as a rudimentary view of connected devices to allow manual control.

The *Control Layer*, at the top of the architecture, is what ultimately controls the system. The Control Layer comprises components that allow either a user or an automatic system to control connected devices. In RuCAS, the control system consists of a rule engine with a REST interface, in which a user can create event-condition-action rules that are evaluated on an interval. HomeOS and BOSS both expose APIs that can be used for developing applications that control the system.

1.4 Challenges

This section describes four challenges that we find central in regard to extending HomePort with support for device automation. We compare a rule based and an application based approach for handling device automation and look at how the systems from Section 1.2 use device groups to ease manage-

ment tasks. Then we discuss how the existence of many rules or applications in a system can lead to unintended system behavior. Lastly, we compare the current HomePort architecture with the generalized architecture from Section 1.3 and identify missing components.

1.4.1 Automation Control

The systems described in Section 1.2 have different approaches on how to handle automation control. All three solutions include structured ways to introduce automation. HomeOS and BOSS rely on exposing their services through an API, and having developers make control applications for the system. These applications can then be utilized by end-users without much configuration. RuCAS relies on a central rule engine with an interface which can be used for building rules and including them in the rule engine.

Both solutions have advantages and disadvantages from the point of view of the end-user. When using an application system, it is theoretically easy for an end-user to install an application that fits the configuration of devices he needs to control. In practice, however, the user relies heavily on the specific implementation of applications, and may discover that an application has not yet been developed for his specific use-case. This restriction is not present when using a rule-engine in which all devices can be combined arbitrarily at will. This means that users can set up any form of automation they see fit, asuming the rule language has enough expressiveness, and can also change it whenever they like. On the other hand, when using the rule engine provided with RuCAS, the user is responsible for configuring the entire system, which can be a big task if there are many devices to control. In such advanced automation cases, it may be preferable to use a tried-and-true pre-configured application.

The application-based approach used by both BOSS and HomeOS rely on some form of internal typing or categorization of devices that must happen before an application can access them effectively. HomeOS tries to solve this problem using roles that can be assigned to devices. If no role that describes the functionality of a specific device exists, a new one can be added to a central database. This solution seems sound but can lead to confusion if developers create new roles that cover the same functionality. In BOSS, devices are categorized both based on the functionality that they provide, as well as their physical location. This allows applications to access them using an approximate query language designed for the task. BOSS provides no solution for mapping devices to these categories automatically. Instead, an administrator must manually perform this task each time a new device is added to the system. When using a rule-engine configured by human users this problem is diminished, since they are able to interpret arbitrary device categorization based on the context.

1.4.2 Device Groupings

Grouping of devices can make it faster and more clear to perform different managements tasks. It can for instance allow users to specify that a policy or action should be applied to a group of devices rather than be applied for each individual device. It can also make it easier for users to find the device they are looking for, if devices are grouped in a meaningful way.

In HomeOS, when a new device is registered the user must choose a location for it. This aligns with physical access, since devices are divided by rooms. It also makes it possible to allow or deny users access to devices in an entire room. The location of devices is also used by applications when they need to find a certain device type in a specific room. The type of a device is not specified by the user, but by the module that created the device.

Devices in BOSS are also grouped by location and type as in HomeOS. The intention in BOSS however, is that the grouping process should happen automatically without any user involvement. This has showed to be very difficult and a solution for this has therefore not been developed. In RuCAS device groupings do not exist. It is, however, possible to create compound contexts that aggregate information from several devices.

The challenges involved in developing a solution for device groupings is to identify what groups should be present and whether groups should be manually specified by the user or be automatically derived by the system.

1.4.3 Unintended Behavior

When controlling a home automation system, one must take care when setting up the rules that govern automatic control in order to avoid unintended behavior. Requests for control can lead to unintended behavior in several ways, some of which will be described in the following.

When several sources control the home automation system over the same period of time, some may potentially effect conflicting changes in the same device(s) concurrently. As an example of this, consider the following two rules:

if LightSensor.Value >= 50 **then** Lamp.TurnOn() **end**

if LightSensor.Value <= 60 **then** Lamp.TurnOff() **end**

If these rules are evaluated while the value of the light sensor is between 50 and 60, the conditions of both rules are fulfilled. One will request that the lamp turns on, while the other will request that it turns off. If this is not handled in the system, both actions will be executed and the ultimate result of the evaluation will depend on the order of execution in the specific case.

In HomeOS, concurrent device access is handled by assigning priorities to the applications that potentially can control the same device at the same

time. This guarantees that only one application at a time can access a specific device and since the HomeOS API allows applications to make synchronous requests to a device; applications have control over the device as long as it needs. BOSS handles concurrent device access in a similar manner to HomeOS, while RuCAS does not specify how this problem is solved.

Another kind of unintended behavior may occur when many rules or applications exist in the system. Although individual rules may work fine, rules in combination may not align with the intention of the user. As more rules are introduced in the system, it gets harder for a user to predict how it will react in all situations. In some cases this may never be a problem, or only cause minor inconveniences, but the system could potentially exhibit unintended wasteful or even dangerous behavior. In some situations problems may be easy to find since they often occur in the system, but there may also be corner cases that rarely occur and are difficult to find.

It seems that neither HomeOS, BOSS nor RuCAS have a way of specifying behavior that should not occur in the system, but trust that users are able to command the overall system behavior even though many rules or applications may be present in the system. Even in the case where a user is able to properly set up the collection of automation rules to make the system behave as intended in all situations, it seems reasonable to assume that in some cases it may be a difficult task to consider what behavior the system will exhibit in every case.

The challenge in solving these problems lies in choosing a solution that provides users with confidence in the system's ability to exhibit appropriate behavior while also being practical to use.

1.4.4 Missing Components

Comparing the HomePort architecture described in Section 1.1 with the generalized architecture from Section 1.3 reveals that some components are missing in HomePort.

HomePort's plugin system makes up the Adapter Layer, enabling third parties to develop plugins that can discover devices on any given subnetwork, and may use a provided API to add these devices to HomePort's configuration dynamically.

The configuration is an abstraction over all connected devices that can be used by the rest of the system with no regard to the specific protocol that must be used for communication. This configuration is HomePort's Device Abstraction Layer, but it is limited in that only plugins are able to control the level of abstraction. Users have no way of aggregating devices or otherwise controlling the level of abstraction with which devices can be accessed. Since HomePort currently just exposes every connected device and service in a REST interface for manual manipulation, the system does not rely on persistent device and service ID's.

The Management Layer in HomePort consists of a simple REST interface that exposes all devices for control. The interface only allows for polling device values directly through plugins (i.e. there is no event or notification system). There is no way of restricting user access to the system. If users are connected to the same network as HomePort, they have access to the entire REST interface without restriction. The system also does not handle conflicting actions in any way. If a user was to program an application for automation using the current system, he would have to guarantee that actions sent through the REST interface have already been checked for conflicts prior to this step.

Lastly, the Control Layer is completely missing. For a user to begin setting up automation in HomePort, he would have to develop this whole layer himself. The REST interface could technically be used for this, but in order to allow users to control the system or set up automation in a meaningful way, the interface must be improved, or a rule engine must be integrated in HomePort.

1.5 Problem Delimitation

To allow devices on different sub-networks to be utilized for automation, we propose that a rule engine should be integrated in HomePort. As discussed in Section 1.4.1 there are both advantages and disadvantages with choosing a rule engine over an application based approach. One advantage of choosing an application based approach is the simplicity of setting up automation as an end-user. If an application that supports a desired automation scenario exists, a user can easily install and configure the application without having to manually create advanced event-action rules.

On the other hand, if no application exists for the purpose, the user must wait until a developer finds time to built it. Using a rule engine it would be possible for the user to manually create event-condition-action rules that fulfill most scenarios. In systems using a rule engine, users can visually see devices in the system and has control over which should be included in a rule. Therefore the need for predefined device categories is not absolutely necessary.

It is, however, still useful to allow users to create groups of devices using arbitrary categories. As mentioned in Section 1.4.2, there may be some advantages to grouped device access. Namely, having the ability to reason about a group of devices instead of single devices can potentially make it easier for end-users to cope with the complexity of creating rules. We therefore propose that end-users should be able to create device groups as a form of abstraction that can ease the process of both creating and maintaining rules in HomePort.

Another focus of this project will be to handle unintended system be-

havior, described in Section 1.4.3. Solutions for handling concurrent device access are already proposed by HomeOS and BOSS and a solution for this will therefore not be considered. Instead, the focus will be on developing a language that can help users command the overall system behavior even though many rules or applications are present in the system.

As described in Section 1.4.4, service values must be retrieved using polling. By introducing automation rules, the number of requests to services will increase because the condition of a rule continuously must be checked to find out if actions in the rule should be carried out. This consumes unnecessary many resources and can be avoided by introducing an event mechanism that allow automation rules to subscribe to state changes of services. We therefore suggest that an event mechanism is integrated with HomePort.

The management layer described in Section 1.4.4 will not be considered in this project. The layer currently consists of a REST interface that allows control over selected functionality in HomePort. The REST interface will, however, be extended as appropriate in order to allow access to new functionality.

1.6 Requirements Specification

The following requirements are listed in order of importance, the first, being the most important. The reason for choosing these requirements are explained in Section 1.5.

1. **HomePort must be extended with a rule engine**

   The rule engine must support simple event-condition-action rules. To make it possible to interact with the rule engine, the existing REST interface must be extended with support for rule management operations (i.e. add, remove, and edit rules).

2. **It must be possible to subscribe to events about state changes in connected devices**

   This event mechanism should be usable across the different components in HomePort. E.g. the rule engine should be able to receive notifications when the state of a device or service changes in HomePort. Furthermore, the plugin API should be extended to allow plugins to publish updated values on services.

3. **Users must be able to define unintended behavior in the system**

   The system should be able to handle unintended behavior by allowing users to specify scenarios that should never occur in the system. We

propose that users should be able to define unintended behavior in a similar manner that intended behavior can be defined using event-condition-action rules. The mechanism for defining such unintended behavior will be named *Safety Rules*.

4. **Users must be able to manually group devices**

   To make it easier for users to create rules and get an overview of existing devices in HomePort, it must be possible to group devices based on location or type. The REST interface must also be extended to support add, edit and remove operations to facilitate management of device groupings.

2 | Automation Rules

As stated in Requirement 1, an event-condition-action rule engine must be integrated into HomePort. From now on this rule engine will be referred to as the *automation engine*.

In order to create such an automation engine, we will first design a language which can be used to express *rules* for HomePort. As the name suggests, rules in an event-condition-action format have three main components. First, one or more *event* triggers may be added to the rule. In our case these triggers will be state changes in observable devices connected to HomePort (e.g. when a switch is flipped). The *condition* component is an expression of boolean algebra. The values in this expression may be a combination of absolute values and live sensor values. The last main component of the rule is the *action*. This component defines one or more actions that must be applied in the case where an event trigger is fired, and the condition is satisfied. In our case, an action is simply an assignment of a value to an actuator connected to HomePort.

```
1 event: LightSensor
2 cond: LightSensor > 70
3 action: Lamp.TurnOn()
```

Listing 2.1: Example of event-condition-action rule

An example of a simple event-condition-action rule can be seen in Listing 2.1. In this example, when the value of the light sensor changes, the condition is evaluated. If the value of the light sensor is more than 70, the action will be applied, and the lamp will turn on.

2.1 Scenarios

In order to gather additional information about requirements for an event-condition-action type automation engine for HomePort, we will consider a number of scenarios that show potential desires of a user using such a system. The consumer persona is named Peter and is 34 years old. He likes electronic gadgets and his technical skills may be considered above average. Peter owns

a home automation system, which he uses to automate as many smart devices in his home as he can think of.

2.1.1 Event Triggers and Delayed Actions

Since installing his new smart fan in his bathroom, Peter has been thinking of different ways to utilize the fan best. He decides that he would like it to function much like an ordinary bathroom fan, which is connected to the light switch and will stay on for a while after the light has been turned off again. Therefore, he would like to create two rules. The first stating that, when the light switch is turned on, the fan should be activated. The other rule states that, when the light is turned back off, there should be a ten minute delay after which the fan should turn off.

2.1.2 Interval Triggers

During the holidays, Peter is, as usual, going to live with his family for about a week. In order to deter criminals from looting his empty home, Peter would like the lights to turn on and off on an interval in order to feign that the house is not empty. He would like to create a rule that turns on the lights in the living room, the kitchen, and the bedroom at various intervals for various amounts of time, and then turns them back off.

2.1.3 Deactivate Rules

In order to save money on energy used to light his house, Peter has installed a bunch of motion sensors around the house, and has set up appropriate rules to turn lights on and off as he moves around the house. However, his wife is a restless sleeper and has complained that if she moves even an inch during the night, the lights will turn on and wake her up. In order to prevent this from happening, Peter would like to add an activation interval to the lighting rules. This states that every day, between 24:00 and 06:30, the rule should be inactive, such that the lights will not turn on, even though the motion sensor is activated.

2.2 Requirements

From reading through the scenarios we can derive a number of requirements for the language in which rules are to be written.

1. **The user must be able to specify a number of sensors to observe for changes in their state. Upon such change, the rule must be evaluated.**

Using state changes as rule triggers in an event-condition-action automation engine is a natural choice in the context of home automation, since this allows rules to immediately react to changes in the observable environment.

2. **It must be possible to specify a time interval that describes when the rule should be evaluated.**

   Timed interval triggers enable users to create rules that are evaluated periodically without being tied to a specific event in the environment. An example of this can be seen in Scenario 2.1.2.

3. **The user must be able to specify a conditional statement that is to be evaluated when a rule is prompted for evaluation by either a state change or a time interval.**

   The conditional statement should allow users to compare absolute values and live sensor values in order to determine whether or not a set of actions should be applied. This is an integral part of the event-condition-action system.

4. **The user must be able to specify a number of actions to be applied to actuators in the environment in the event that a rule is prompted for evaluation, and its internal conditional statement evaluates to true.**

   In the context of HomePort, an action is the assignment of a value to a connected actuator.

5. **It must be possible to both specify immediate as well as delayed actions in a rule.**

   As can be seen in Scenario 2.1.1, it may be appropriate to delay an action. Defining a delay with each action will also enable users to create sequences of actions that are applied over time.

6. **It must be possible for the user to specify a time interval that describes whether or not the rule is active at a given time.**

   In some cases a rule is only appropriate at specific times of the day. An example of this can be seen in Scenario 2.1.3 It should be possible to specify this in order to allow HomePort to disable and enable rules automatically as appropriate.

2.3 Language Specification

Listing 2.2 contains an abstract context-free grammar in Backus-Naur form for expressing rules. Abstract meaning that some terminals, such as data types (eg. int, string) are not defined detail. In this language, a rule consists of four sections; event, condition, action, and within. The event section can consist of a number of intervals or service values. The condition is optional, and can consist of either an int, which describes a timed interval in milliseconds, or a string that describes a time of day. Actions consist of one or more pairs of a service value and values to be applied. The within section is optional, but may consist of a pair of intervals that describe when the rule should activate and deactivate, respectively.

The service values used in events, conditions, and actions consist of two strings which are used to identify a service. An interval may consist either of a traditional interval of days, hours, minutes, and seconds or it may be set to a specific time of day on specific days of the week.

```
rule            := 'event' eventExp 'cond' condExp 'action'
                   actionExp within
eventExp        := serviceValue eventExp
                 | interval eventExp
                 | ϵ
condExp         := cond boolOp condExp | cond
cond            := value binOp value | true | false
                 | '(' condExp ')'
                 | '!' condExp
                 | groupAggrExp
value           := literalValue | serviceValue
literalValue    := int | double | string | bool
serviceValue    := device '.' service
binOp           := '>' | '>=' | '<' | '<=' | '==' | '!='
boolOp          := '&&' | '||'
actionExp       := serviceValue literalValue delay actionExp
                 | groupActExp delay actionExp | ϵ
delay           := int | ϵ
within          := 'within' interval interval | ϵ
interval        := int | string
```

Listing 2.2: BNF for Automation Rules

The event section consists of a number of service values and intervals. Service events will trigger when the value of the service changes. In the case of an interval, a timer will ensure that the rule is evaluated according to the interval.

A rule must have a condition that can be evaluated to a true or false value to determine whether actions in the rule should be applied or not. A simple

condition could be: `service == 5`, meaning that if the value of `service` is equal to 5, then the actions in the rule should be applied. One could also imagine more complex examples where several conditions are chained together to form a single condition. Consider the more complex condition: `4 >= service_1 and service_2 == "Active"`, where two simple conditions have been combined using the `and` operator to express that both the condition on the right and left-hand side of the `and` operator must be true before any actions will be executed. The `AggrExp` component in `cond` refers to a group expression, which is described in detail in Chapter 4. In short, it allows comparison with aggregates (i.e. minimum, maximum, average, count) of a group of services.

An action consists of a service value, an absolute value, and, optionally, a delay. The service value must refer to a service which has actuator capabilities, and the value must be compatible with the service. When an action is applied, the plugin responsible for the service is notified to change the value according to the action. The delay defined in an action is relative to the action that was applied immediately before, and as such, will also affect subsequent actions. For example, if you have two actions, each with a delay of 200 ms, the first will be applied at T+200 ms, and the second at T+400 ms. In addition to applying an action to a single service, it is also possible to replace this with an `ActionExp`, which will apply a value to a group of services. Details about `ActionExp` can be found in Chapter 4. In summary, it allows the user to create a rule that applies the same action to a group of services.

The two strings comprising a service value are a device ID and a service ID, respectively. These IDs are defined by the plugin that governs the device in question.

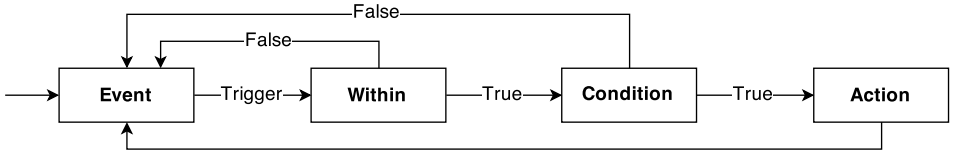


Figure 2.1: The flow of an automation rule in HomePort

The overall evaluation flow of a rule is depicted in Figure 2.1. To summarize, the event part of the rule can consist of one or more events, which will trigger the rest of the rule to be evaluated. An event can either be a timer which triggers the rule to be evaluated in a periodic fashion or it can be a service which changes its value and thereby raises an event. When an event triggers, the rule's condition will be evaluated. If the condition is satisfied, the set of actions will be executed. An overview of the full context-free grammar for an automation rule can be seen in Appendix C.

3 | Safety Rules

Requirement 3 states that users must be able to define unintended behavior in the system in a similar manner that automation rules can be used to define intended behavior. To make this possible we propose a language that can be used for this purpose.

Scenarios will be presented that try to highlight what type of unintended behavior users would like to be able to define. Based on these scenarios, a list of requirements will be given and finally a language for creating safety rules will be presented.

3.1 Scenarios

To investigate what type of unintended behavior users would like to be able to define, the following scenarios have been created. The scenarios are partly based on the discussion of unintended behavior from Section 1.4.3. The persona used in each scenario is described in Chapter 2.

3.1.1 Conditional Statement and Service Groups

Peter knows that it affects his heating bill when thermostats in his house are not switched off when he airs out the house. When windows and doors are opened, the room temperatures will drop and the radiators will consequently try to compensate by heating more.

To make sure that the thermostats in the living room cannot be turned on when he airs out the room, he would like to tell the system that if one or more windows in the living room is open, then the thermostats in that room must be switched off.

3.1.2 Time Window of Validity

Peter has created automation rules that cause the radiators in the entire house to automatically be turned down to 15 degrees Celsius, when he turns on the alarm system as he leaves for work. He knows how important this is to his heating bill and would therefore like to make sure that this actually happens as intended.

If he could tell the system that the thermostats in the entire house must never be set above 15 degrees Celsius between 8 AM and 3 PM every day except in the weekend, he would be sure that his automation rules worked.

3.1.3 Elapsed-time Monitoring

The windows in Peter's living room is equipped with motors that allow him to open and close them using his home automation system. He has created rules that will close the windows when he locks the front door after leaving his house in case he forgets to do it manually.

The windows take a few seconds to close and he wonders if it is possible to create a safety rule that monitors if the windows are properly closed 30 seconds after he turns on his alarm system.

3.1.4 Time-based Monitoring

In Peter's bathroom he has an extractor fan that should start when the bathroom light switch is turned on and continue running for 30 minutes after the light switch is turned off.

Peter is not convinced that the rule he has setup is working properly and wants to add a safety rule that monitors whether the extractor fan is actually running for the full 30 minutes.

3.1.5 Conditional Monitoring

Peter decides that it is not necessary that the extractor fan in the bathroom continues running after the light switch is turned off and deletes the old rule and creates a new one that stops the fan when the switch is turned off.

Again, Peter would like to convince himself that his new rule is working properly and wants to create a safety rule that monitors if the extractor fan is running until the light switch is turned off.

3.2 Requirements

The scenarios presented in the previous section highlighted different types of unintended system behavior. Based on these scenarios the following list of requirements for safety rules have been identified.

1. **It must be possible to specify a conditional expression that can include sensor services (e.g. light sensor, thermometer).**

   A conditional statement is used in every scenario from Section 3.1 and should always be included in a safety rule. The conditional statement can be used to define a scenario that must never occur in the system.

2. **It must be possible to use device group abstractions in a conditional expression**

   In scenario 3.1.1, the user defines a conditional statement containing the sentence: "'*if one or more windows in the living room is open..*". This suggests that an abstraction over devices would be a good idea to help abbreviate conditional statements. The design of device group abstractions can be seen in Section 4.3.

3. **It must be possible to specify a time window in which the safety rule is active**

   Users might want safety rules to be active only within a specific period of time. An example of this can be seen in scenario 3.1.2.

4. **It must be possible to specify that a condition is satisfied during a time period**

   In scenario 3.1.4 the user would like to monitor that a condition is always true during a period of time.

5. **It must be possible to specify that a condition eventually becomes true within a time period**

   In scenario 3.1.3, the user would like to check that a condition eventually becomes true before a specified period of time has elapsed.

6. **It must be possible to specify an optional, second conditional expression**

   In scenario 3.1.5, the user would like to monitor that a condition is true until another condition becomes true. To allow for this, it should be possible to specify a second conditional expression. This conditional expression should have the same characteristics as described in Requirement 1.

3.3 Language Specification

The syntax for expressing safety rules is expressed as an abstract context-free grammar in Backus-Naur Form while the semantics are described using state diagrams. Some terminals, such as data types (e.g int, string) are not specified in detail.

Based on the requirements from Section 3.2 and the scenarios from Section 3.1, an abstract grammar describing a safety rule has been created. The grammar can be seen in Listing 3.1 and specifies four types of safety rules that monitor the system in different ways. In each of the four types, an optional `within` construct can be defined if the user wants to limit the rule

to be active within a certain time window. This time window was added to each rule to fulfill Requirement 3.

The conditional expression (`condExp`), as well as the event expression (`eventExp`), are identical to the constructs defined in automation rules, described in Section 2.3. The syntax for conditional expressions support the use of sensor services and group abstractions and does therefore already fulfill Requirement 1 and 2.

```
safetyRule    := 'cond' condExp within
               | 'event' eventExp 'cond' condExp safetyType within
safetyType    := 'always' condExp 'for' uint
               | 'always' condExp 'until' condExp
               | 'eventually' condExp 'for' uint
condExp       := cond boolOp condExp | cond
cond          := value binOp value | true | false
               | '(' condExp ')'
               | '!' condExp
               | groupAggrExp
value         := literalValue | serviceValue
literalValue  := int | double | string | bool
serviceValue  := device '.' service
binOp         := '>' | '>=' | '<' | '<=' | '==' | '!='
boolOp        := '&&' | '||'
eventExp      := serviceValue '||' eventExp | serviceValue
within        := 'within' interval interval | ϵ
interval      := int | string
```

Listing 3.1: BNF for Safety Rules

The most basic safety rule that can be constructed from the grammar is `'cond' condExp within`, where `within` is optional. This rule is based on Requirement 1 and can be used to specify that the given condition should never be satisfied in the system.

Figure 3.1 shows a state diagram of this rule. The condition (*Never Condition*) is only evaluated if the current time is within the specified time window (*Within*) or if no time window is specified. If the condition evaluates to true, an unwanted state is entered to indicate that some kind of unintended behavior has occurred. As long as the condition evaluates to false, the safety rule is reevaluated.

The syntax: `'event' eventExp 'cond' condExp 'always' condExp 'for' uint within` is based on Requirement 4 and defines the safety rule for monitoring that a condition is always satisfied during a specified period of time. In this type of rule, the user must specify an event, two conditions, a time period in seconds and optionally a time window.

A state diagram of this safety rule can be seen in Figure 3.2. This type of safety rule starts the same way as an automation rule, where an event

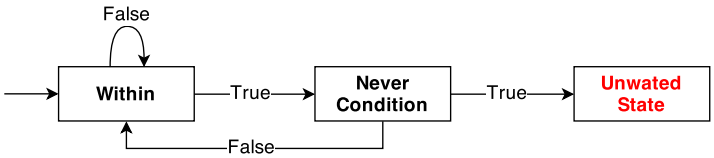


Figure 3.1: State diagram of a safety rule with a condition that must never be satisfied.

triggers the rest of the safety rule to be evaluated. When the event in the safety rule gets triggered the *Within* state is entered. If the current time is within the specified time window (*Within*) or if no time window is specified, the *Guard Condition* is evaluated in order to determine whether the rule should be evaluated further, just like an automation rule. If so, the *Always Condition* state is entered and a timer is started. Before the timer is started it is set to the time period specified by the user. If the *Always Condition* becomes false before the timer times out, the condition has not been true for the entire time period and an unwanted state is entered. If this does not happen, the timer is stopped and the rule is reevaluated.

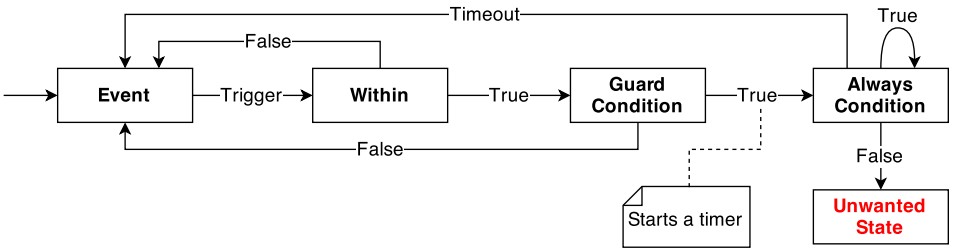


Figure 3.2: State diagram of a safety rule with a condition that must always be satisfied during a specified time period.

The rule for monitoring that a condition is eventually satisfied before a defined time period has elapsed can be constructed using the syntax: `'event' eventExp 'cond' condExp 'eventually' condExp 'for' uint within`. This rule is based on Requirement 5 and like in the latter, the user must specify one or more events, a guard condition, a time period in seconds and optionally a time window. The difference between the rules is the second keyword, `'always'` and `'eventually'`, respectively. This keyword makes it possible to distinguish between the two and give them semantically different meanings. The semantics of the different rules will be explained in the following section.

The state diagram in Figure 3.3 shows how this safety rule is evaluated. This diagram is similar to the one in Figure 3.2 but differs in how the true, false and timeout transitions are taken from the condition state (*Eventually*

Condition). In this safety rule, an unwanted state is entered if the condition has not evaluated to true before the timer has timed out.

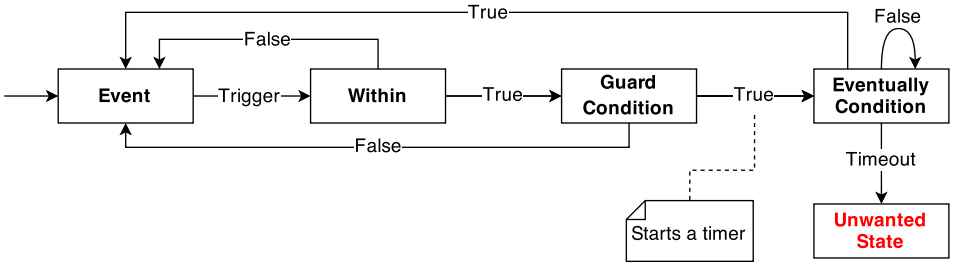


Figure 3.3: State diagram of a safety rule with a condition that must eventually be satisfied during a specified time period.

The last type of safety rule has two conditions and can be used to verify that one condition is true until another condition becomes true. The rule has the syntax: `'event' eventExp 'cond' condExp 'always' condExp 'until' condExp within` and is based on Requirement 6. The user must specify an event, three conditions, and an optional window of time.

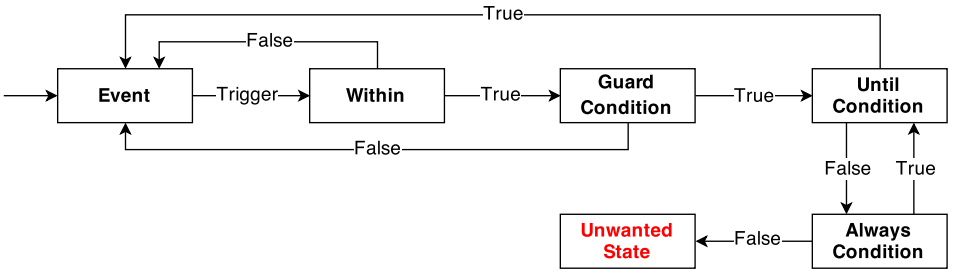


Figure 3.4: State digram of a safety rule with two conditions.

A state diagram describing this rule can be seen in Figure 3.4. As with the other rules, an *Event* triggers the rule to be evaluated, and the *Within* state is entered, and if it evaluates to true the *Guard Condition* state is entered, and similarly, if the *Guard Condition* is satisfied the *Until Condition* is entered. If this condition is satisfied, the entire rule is reevaluated, since there is no reason to check the *Always Condition*. If the *Until Condition* evaluates to false, the *Always Condition* is checked and an unwanted state is entered if it also evaluates to false. This is because the *Always Condition* has not been true until the *Until Condition* was satisfied. Otherwise, the *Always* and *Until* conditions will be evaluated until either *Until* evaluates to true, or *Always* evaluates to false.

An overview of the full context-free grammar for a safety rule can be seen in Appendix D.

4 | Grouping

In this chapter a solution for grouping devices in HomePort will be given. Based on Requirement 4, it should be possible to group multiple devices. Grouping of devices will act as an abstraction, that can help ease creation and maintenance of rules in HomePort. Scenarios will be presented, that demonstrates different use cases for where groups could be useful, and how they are supposed to work. Requirements from these scenarios will be identified, and used when designing the concept of groups. Finally, a language will be designed, to make it possible for users to utilize groups in their rules.

4.1 Scenarios

To get an overview of possible use cases where device groupings could be useful, a number of scenarios have been developed. The persona used in the scenarios is the same as in Chapter 2.

4.1.1 Grouping of Devices

Peter does not want to spend time, going through the entire house to switch off all the lights before he leaves his house. He therefore decides to create an automation rule that will turn off all lights in the house when he turns on the alarm system. He discovers that it takes a lot of time to specify an action for each individual light unit and wonders if it is possible to perform an action on a group of devices instead.

4.1.2 Group Subsets

One day when Peter is going to bed, he discovers another problem which annoys him. Peter has to go through the entire house to check if all the lights are turned off. Peter already has a button installed in his bedroom, which is currently not in use, and he figures that he could automate it via HomePort. Peter has already created device groups based on location in the house, and wants to use these groups when using the button via HomePort. When someone presses the button, Peter wants it to turn all the lights off in the house except for the bedroom.

4.1.3 Group Filtering

Peter has a tendency to forget to close the windows in his home when he leaves. So he figures, one thing he always remembers to do is to turn on the alarm system. Peter has already created a group for the windows in his home. He creates a rule that states, if one or more windows are open when the alarm system is turned on, a lamp next to the door should start flashing.

4.2 Requirements

The scenarios presented in the previous section, display different use cases for how groups could be utilized in HomePort. Based on these scenarios the following requirements has been identified.

1. **It must be possible to group multiple devices**

   As already stated in Requirement 4, groups should act as an abstraction mechanism for users, which should make it possible to group multiple devices. An example of this can be seen in Scenario 4.1.1.

2. **It must be possible to control a group of devices uniformly** Devices in HomePort can consist of multiple services, which each can have a different data type, and might also use a different naming convention. This has to be taken into consideration when designing the groups to allow devices to be uniformly accessed.

3. **It must be possible to combine multiple groups**

   To avoid that redundant groups are created for special purposes, it should be possible to combine existing groups.

4. **It must be possible to exclude one or more groups from an existing group**

   Using similar logic as the requirement above it should be possible to exclude one or more groups from a set of groups. An example of why group exclusion can be useful is shown in Scenario 4.1.2.

5. **It must be possible to use aggregate functions on groups**

   It must be possible to use the aggregate functions count, average, minimum and maximum on groups. This could make it possible to query how many services that fulfill a certain condition or get the minimum value among a set of services. An example of how the count function can be used is shown in Scenario 4.1.3.

6. **It must be possible to apply an action to a group**

It must be possible to apply an action uniformly on a group, like the scenario in Section 4.1.2, where Peter turns off all the light in his house.

4.3 Grouping of Devices and Services

All of the above scenarios display different use cases for where groups can help simplify the process of creating and maintaining rules in HomePort. We will introduce the concept of a *device group*, which will make it possible to group an arbitrary number of both devices and other device groups. A device group in HomePort will act as an abstraction layer, which, in theory, should make it easier for users to manage rules. To target a specific subset of devices, operations from set theory can be borrowed. Simple operations, like joining multiple device groups, or excluding one or several device groups from a set of device groups would make it easier for users to target a specific set of devices. Each device in HomePort can only belong to a single device group, which forces users to create device groups which reuses existing device groups. Forcing users to decide which device group a given device should belong to should prevent redundant device groups. This way of structuring devices has been exemplified in Figure 4.1, which creates a tree-like structure for all of the grouped devices in the system.

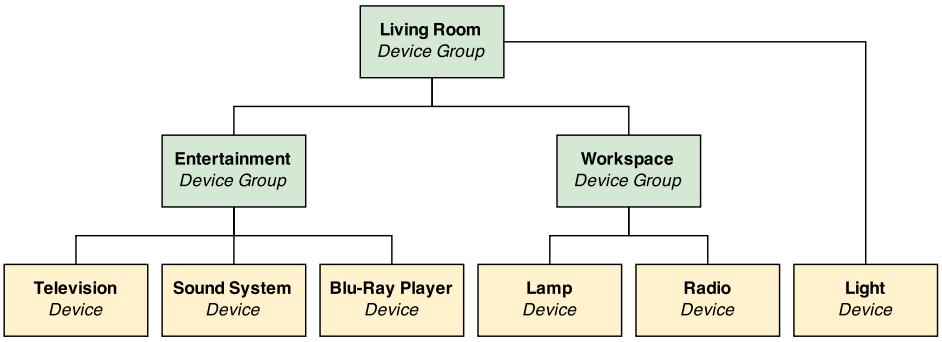


Figure 4.1: An example of device groups

Although device groups makes it possible to group multiple devices, it is still not possible to access their services uniformly. Devices in HomePort can consist of an arbitrary number of services, which each can have a different data type, but also use a different naming convention.

The main problem with uniformly accessing services on a set of devices is that services are not bound to a predefined type in the system and it can therefore not be automatically inferred what the service actually does. Although services have a name and a data type, different naming conventions and data types may be used on different services that have the exact same functionality.

An example could be a lamp with an ON/OFF service. The "ON/OFF" service name can be expressed in many different forms. Furthermore, the data type of the "ON/OFF" service could possibly be implemented as a boolean value, but could as well be an integer value.

In a naive solution you could try to access services based on name and data type. This way work in some situations, but may also result in accessing services with completely different functionality.

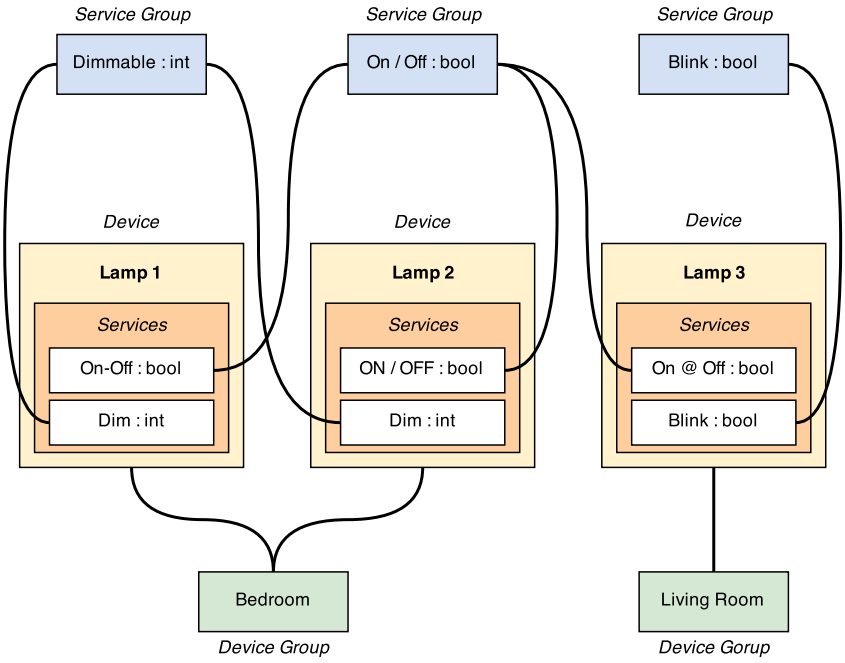


Figure 4.2: An example of device groups and service groups

Since uniform service access on device groups is not possible in the current design, we propose a solution where users must manually group services based on functionality and data type. This way it is guaranteed that a group of services can be accessed in the same way.

We will from now on refer to this type of grouping as a *service group*. A service group will consist of a name, a data type, service type and one or several services. The name can be used to describe the functionality of the contained services and the data and service types are used as constraints that services must adhere to. The service type can either be sensor, actuator or both.

A service group could for instance be named "Lamps On/Off", specify boolean as data type and actuator as service type. In this example, all services in the group must be actuators and have the data type boolean.

Using these device and service groups, queries that target services on multiple devices in a uniform manner can be formulated. Figure 4.2 dis-

plays three different lamps, each of which has some functionality for turning the lamp on and off. The lamps could have been produced by different manufacturers and, therefore, have a different naming convention for similar functionality. In this example, a service group named "On / Off" has been created. The data type of the service group is boolean, which matches all of the on/off services of the lamps. Combining a service group and one or several device groups makes it possible to either perform an action or query a group of related services uniformly.

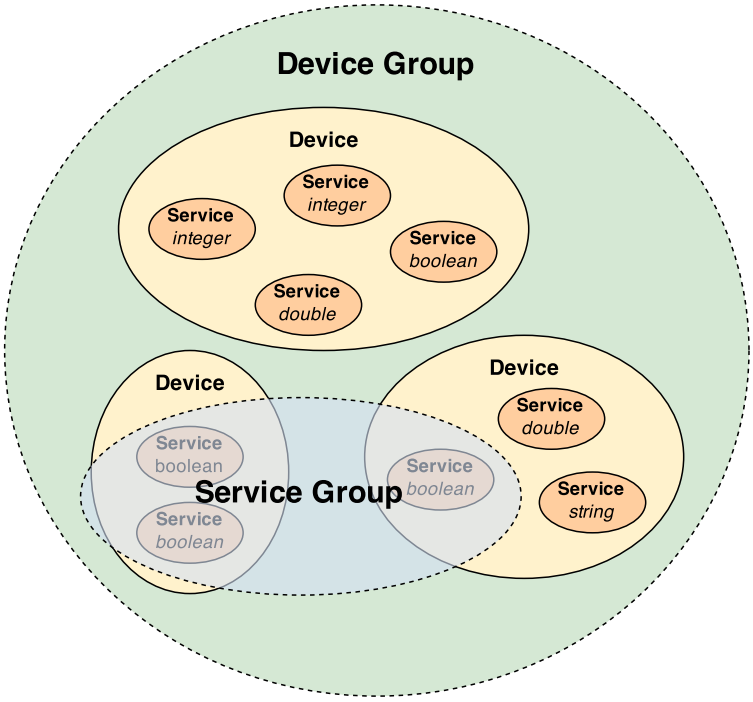


Figure 4.3: Intersection of a device group and a service group

The intersection of a device group and a service group can be described via a Venn diagram as seen in Figure 4.3. The diagram displays a device group containing three devices, and a service group containing three services from two of the devices. The intersection of these two groups is a collection of related services that can be accessed and manipulated uniformly.

4.4 Language Specification

An abstract grammar for querying and performing actions on a group of devices has been developed, and can be seen in Listing 4.1.

```
groupActExp   := '[' groupExp 'do' literal ']'
groupAggrExp  := '[' groupExp aggrFunc binOp literal ']'
groupExp      := 'group' groupType excludeExp 'type' type whereExp
groupType     := '*' | groupCom
groupCom      := group ',' groupCom | group
excludeExp    := 'exclude' groupCom | ϵ
whereExp      := 'where' whereCondExp | ϵ
whereCondExp  := whereCond boolOp whereCondExp | whereCond
whereCond     := whereValue binOp whereValue | true | false
               | '(' whereCondExp ')'
               | '!' whereCondExp
               | groupAggrExp
whereValue    := literal | device '.' service | #
literal       := int | double | string | bool
aggrFunc      := 'count' | 'min' | 'max' | 'avg'
boolOp        := '&&' | '||'
binOp         := '>' | '>=' | '<' | '<=' | '==' | '!='
```

Listing 4.1: BNF for Grouping

Two types of group expressions are available depending on where it is used in a given rule. `groupActExp` should be used in the action part of a rule in order to perform a uniform action across several services. `groupAggrExp`, on the other hand, is used in the condition part of a rule. It computes an aggregated value from the set of selected services and compares it to a literal value.

Both `groupActExp` and `groupAggrExp` are encapsulated by `[ ]` to make it clear where a group expression starts and ends. Both of them can consist of one or several device groups which are expressed by `group` followed by the device group names which should be separated by a comma if there is more than one. If it is the objective to target all of the devices in the system, the `*` operator can be used. This will result in either all of the devices in the system if the `*` operator is used, or the union of all of the declared device groups.

Moreover, it is possible to exclude one or several device groups. This is expressed in an `exclude` clause containing the names of the device groups that should be excluded. Similar to how `group` works, it will result in the union of all of the declared device groups. The combination of the `exclude` and `group` clauses is the set difference of the two groups of devices.

Next, a service group has to be defined in the `type` clause. The intersection between the device group clause and the service group clause will result

in a unique set of related services of the same data and service type.

Subsequently, it is possible to filter the set of services by defining a `where` clause containing a `whereCondExp`. `whereCondExp` is similar to the `condExp` used in rules and safety rules. It is possible to either use literal values, service values or the `#` operator that refers to each individual service in the group of services being filtered. An overview of the evaluation flow of both `groupActExp` and `groupAggrExp` can be seen in Figure 4.4.

When using `groupAggrExp`, `aggrFunc` has to be defined which can either be `count`, `min`, `max`, and `avg`. `count` returns the number of selected services, `min` returns the minimum value of all of the selected services, `max` returns the maximum value of all of the selected services, and `avg` returns the average value of all of the selected services. `aggrFunc` has to be combined with a binary operator and a literal value. Similarly, `groupActExp` has to be followed by a `do` clause containing a literal value that should be applied to all of the specified services.

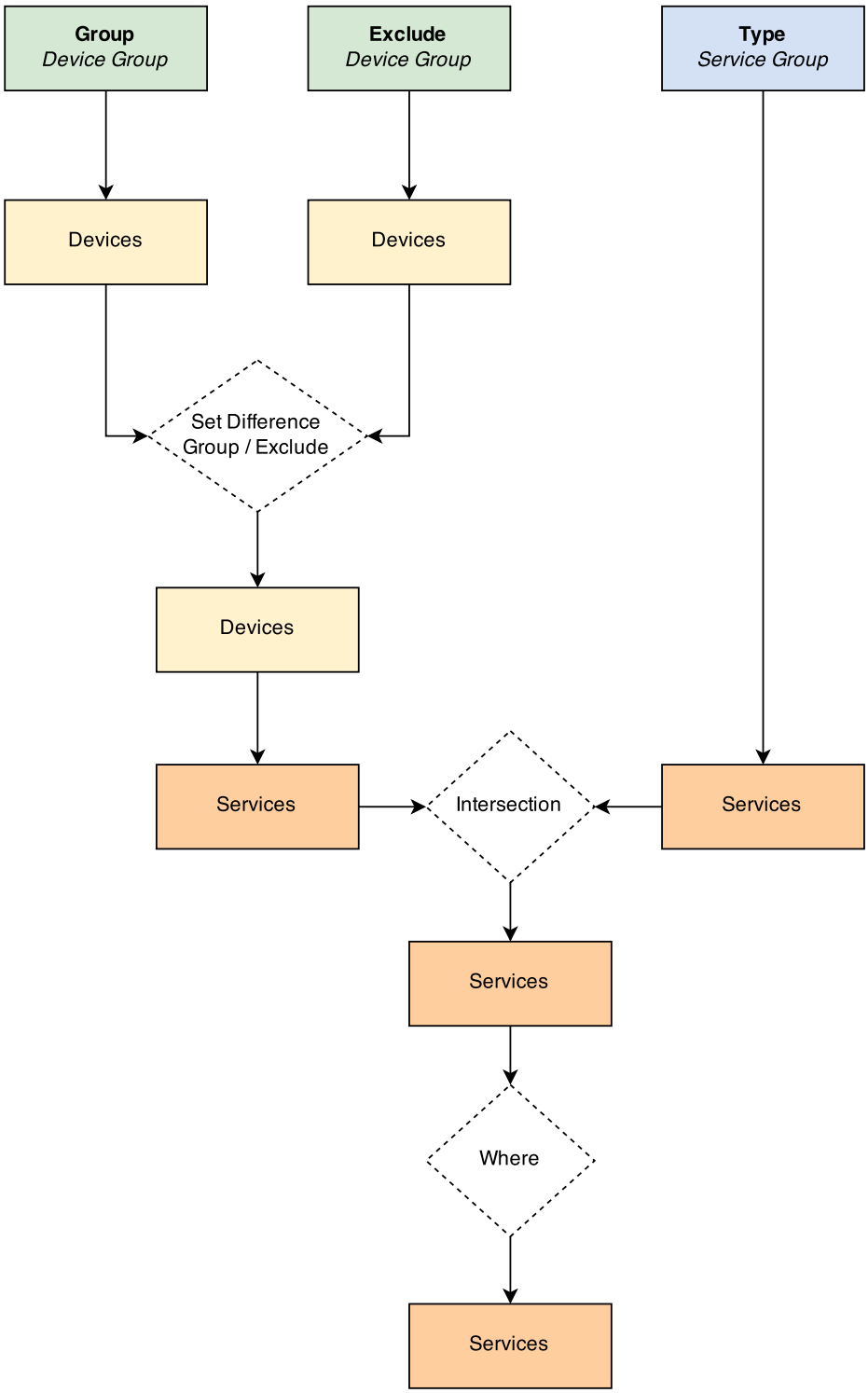


Figure 4.4: Evaluation flow of `groupActExp` and `groupAggrExp`

5 | System Design

In this chapter, a system design for HomePort will be presented. The design is based on the Requirements Specification, Section 1.6, as well as Chapters 2, 3, and 4, in which languages for automation and safety rules are described.

5.1 Component Architecture

The different components which HomePort consist of are depicted in Figure 5.1. The components are grouped using a three-tiered architecture. This architecture was chosen to separate the responsibilities of the different components in the system, but also to create a more modular architecture, where components can easily be exchanged by others.

The Presentation Tier is made up by components that expose functionality to the 'outside world'. The components in this tier are acting as an interface between HomePort and inputs from foreign stimuli. In our case, the presentation tier consists of a REST Interface, which third parties can use to interact with HomePort.

The components in the Logic Tier are responsible for coordinating and managing the main functionality in HomePort. More specifically, the Automation Engine, the Safety Engine, the Plugin Manager and the Configuration.

The last tier in the architecture is the Data Tier, which contains all the components which contains the components responsible for managing persistent data. Below is a more detailed description of each of the components in the HomePort architecture.

REST Interface The REST Interface exposes an API which third parties can utilize for interacting with HomePort. Its responsibilities is to expose the necessary functionality for the end-users and pass the retrieved information forward to the logic tier, but also retrieve data from the logic tier when a request is made.

Automation Engine The Automation Engine should maintain a list of automation rules in the system, as well as expose functionality for

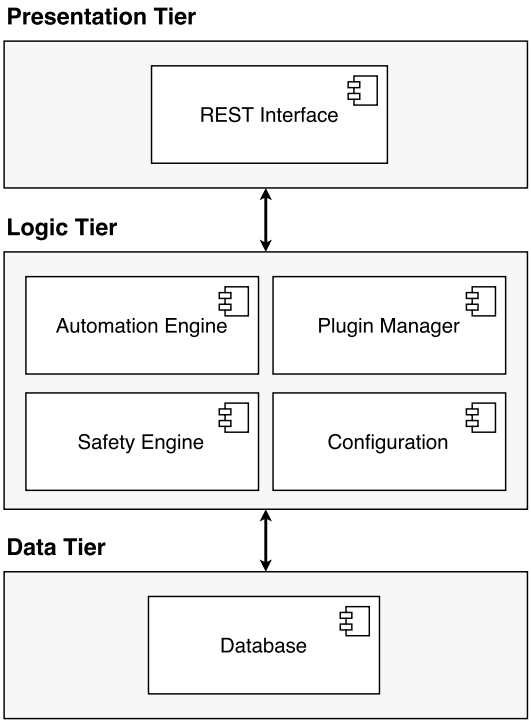


Figure 5.1: Component Architecture of HomePort

managing the rules (e.g. add, remove, edit).

Safety Engine The Safety Engine is responsible for managing all the safety rules in HomePort, and should also expose appropriate management operations, similarly to the Automation Engine.

Plugin Manager The Plugin Manager's responsibility is to manage all the plugins. It should automatically discover when new plugins are available in the file system, and try to load them into HomePort.

Configuration The configuration contains an internal representation of plugins, devices, and services that are currently connected to HomePort.

Database The database component should facilitate persistent storage of automation rules, safety rules, device groups, and service groups.

5.2 Event System

According to Requirement 2, it must be possible to subscribe to state-change events of connected devices. The event system will be designed and implemented with the use of the publish-subscribe pattern. Using this design

pattern will allow other components to subscribe to specific events and be notified when the event is triggered.

The plugin API will be extended with functionality to allow plugins to publish service values to HomePort. It is the responsibility of the plugin to publish updated service values. This design choice is made on the basis that many embedded devices do not have an event mechanism which triggers when its value updates. In these cases the plugin would be able to update the value as appropriate for a given device.

5.3 Grouping

The concept of device groups and service groups has been presented in Section 4.3, and will be further elaborated with design details. Device and service groups will act as an abstraction mechanism in HomePort. A device group can contain an arbitrary number of devices, and there are no restrictions on how many nor which devices can be grouped together.

A device group will have a name to uniquely identify it when users, for instance, utilize them in rules. Figure 5.2 displays a class diagram of a device group in HomePort. The device group will be equipped with functionality for controlling and manipulating the device group itself.

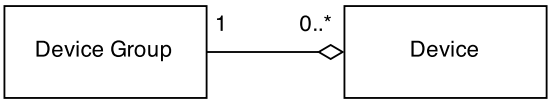


Figure 5.2: Class diagram for a device group

Service groups, on the other hand, can contain multiple services. There is no restriction on the number of services which a service group can contain, only that they need to be of the same data type and service type. Furthermore, similarly to a device group, a service group will also have a name to uniquely identify it. Figure 5.3 displays a class diagram for a service group.

Figure 5.3: Class diagram for a service group

Both device and service groups will be created by the user, and their data has to be saved persistently to avoid data loss.

5.4 Automation Engine

Since the event-condition-action automation rules in HomePort are based on event triggers, each automation rule can exist as a self-contained entity. As such, there is no need for an actual rule engine, but rather, it will act as a relatively simple container which is responsible for maintaining the collection of automation rules. The rule engine will expose functionality for adding and removing rules, and also for activating and deactivating them manually, overriding the automatic activation functionality. The automation rules maintained by the automation engine are designed in accordance with the specification in Chapter 2.

Firstly, an *automation rule* contains a collection of *Events*, each of which can either be a *Sensor* or an *Interval*. These are the services and intervals specified in the 'event' clause when a rule is created. An interval maintains a timer which triggers an event as appropriate, and a sensor triggers events when a new value is pushed from its governing plugin. While the automation rule is active, it subscribes to all events in the collection. When an event is triggered, the rule will evaluate its conditional statement. If the condition is satisfied, the collection of *Actions* will be applied in the sequence that they were specified when the rule was created. Since the application of actions may happen over time because of delays specified by the user, the rule will be deactivated for the duration to prevent unexpected behavior that may occur if the rule is prompted multiple times for evaluation faster than its actions can be applied. Each application of an action simply constitutes the assignment of a value to an actuator. Lastly, an automation rule also contains a *Within* object which maintains the times at which the rule should automatically activate and deactivate as specified by the user. The within object publishes an *activate* and a *deactivate* event that the rule subscribes and responds to as appropriate.

5.5 Safety Engine

As with automation rules, safety rules rely on event triggers for their evaluation, and as such do not require an engine to periodically monitor them. Therefore, the safety engine will not have any responsibilities apart from maintaining a collection of safety rules. As with the automation engine, the safety engine should expose functionality for adding and removing safety rules, as well as for manual activation and deactivation of rules.

Each safety rule maintained by the safety engine can be one of four types of safety rules, each of which is described in Chapter 3. Each safety rule contains a combination of service event triggers, conditions, and timed triggers that allows the rule to evaluate itself in accordance with the specifiction. These constructs have all been explained in the previous section detailing

the automation engine. While active, a safety rule may discover that the system is in an unwanted state. When this happens, the rule will notify the user of the violation, but will not take any action to correct it.

5.6 REST Interface

In the previous version of HomePort, a REST interface was designed to allow external sources to control HomePort. This interface will be extended with additional functionality, for managing both groups of devices, groups of services, automation rules, and safety rules. An overview of the functionality which the REST interface exposes can be seen in Table 5.1. It depicts all of the resources that are available for the client, and the available actions which can be applied on a given resource. A placeholder in the form of {id} is used to indicate a unique id for either devices, services, plugins, device groups, service groups, automation rules, or safety rules.

Clients should be able to retrieve information on both services and devices. Moreover, client should be able to change the value of a service via the REST interface. Devices and services in HomePort are created by plugins that other parties have developed, which entails that certain constraints exist on what clients can do via the REST interface to these types. It is therefore not possible to either add or delete any of them. Clients will only be able to change their state and retrieve information about them. Changing the state of a device refers to changing its meta-information, and for services clients can change both the meta-information and its value.

URI	GET	POST	PUT	DELETE
/devices	×			
/devices/{id}	×		×	
/devices/{id}/services	×			
/devices/{id}/services/{id}	×		×	
/plugins	×			
/plugins/{id}	×		×	
/devicegroup	×	×		
/devicegroup/{id}	×		×	×
/servicegroup	×	×		
/servicegroup/{id}	×		×	×
/rules	×	×		
/rules/{id}	×		×	×
/safetyrules	×	×		
/safetyrules/{id}	×		×	×

Table 5.1: Overview of REST Interface

The REST interface will be extended in order to allow creation and

management of device and service groups in HomePort. Moreover, HomePort has been extended with automation and safety rules, both of which can also be managed via the REST interface. A more detailed description of each of the resources can be found in Appendix A, which elaborates how each of the resources functions.

6 | Evaluation

In order to determine the viability of the proposed solution of automation and safety rules, the system will be tested and a qualitative evaluation will be conducted. The tests will be based on the scenarios presented throughout the report, since these scenarios are the basis for the requirements for the system.

For each scenario, the necessary automation and safety rules will be produced using their respective syntax, which can be found in Appendix C and D. The rules will be loaded into HomePort, and the system will be connected to a virtual home which contains all the devices described in the scenarios. Then, the behavior of the virtual home will be recorded and compared to the desired outcome in each scenario.

6.1 HomePort Implementation

During the last extension of HomePort, described in our previous work [6], we found that working with C was often unnecessarily low-level, taking into account the system we were developing. It was therefore decided that the core architecture of HomePort should be re-implemented in the C# programming language using Mono [5]. By doing so, the system runs on all platforms that support Mono or .NET.

The resulting architecture after the re-implementation can be seen in Figure 6.1, and is based on the three-tiered architecture described in Section 5.1. Compared with the previous architecture, described in Section 1.1, three new components have been added. That is, the `Automation Engine`, `Safety Engine` and the `Database` module. In the previous version of HomePort, plugins were run in separate processes in order to ensure the stability of the system. Instead of this approach, plugins will be isolated using AppDomains [4].

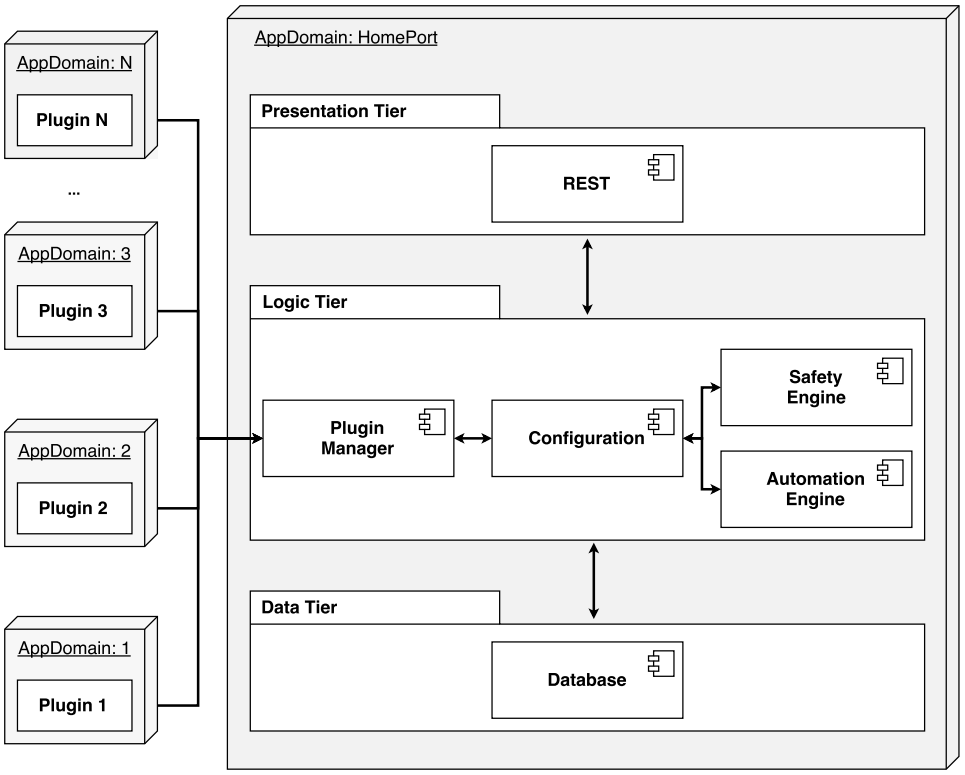


Figure 6.1: HomePort's Implemented Architecture

6.2 Test Setup

The physical model contains in total 17 devices, each of which is one of the following: window, lamp, thermostat, motion sensor, switch, or fan. Each of these devices can be utilized by HomePort, and can be used in either an automation or safety rule. An overview of the home can be seen in Figure 6.2. Each device has its own symbol with a color code to distinguish them from each other. The description of each of the devices can also be seen the figure.

The physical model is build out of a cardboard box where a drawing of the floor plan is attached to the front. To emulate the devices previously described, electronic components that mimic their behavior have been used. The windows are implemented with yellow LEDs to indicate whether they are opened or closed and lamps are implemented with green LEDs. To be able to adjust the temperature on a thermostat we used potentiometers. Motion sensors are implemented with a light-dependent resistor. Lastly, switches are implemented with on-off buttons, and the fan is implemented with an old PC cabinet cooler. The front and rear view of the model can be seen in Appendix B.

The devices described above are connected to a Raspberry Pi via its GPIO pins. A small program has been implemented on the Raspberry Pi to make it possible to control the electronic components remotely. To make the devices available in HomePort a plugin that communicates with the Raspberry Pi program has also been written. An overview of the test setup can be seen in Figure 6.3.

6.3 Experiment

To evaluate the automation and safety rule systems, a number of rules have been created. These rules are based on scenarios that have been used as a foundation for the specification of the rule languages. For brevity, the rules are described according to the grammars that can be found in Appendix C and D instead of their JSON equivalents.

6.3.1 Automation Rules

Multiple automation rules will be created based on the scenarios described in Chapter 2 in order to test if the desired behavior can be expressed with the proposed language, and whether or not the behavior is in accordance with the intent of a given rule. The automation rules will be described in details and linked with our model home depicted in Figure 6.2.

Peter wants both the bathroom fan (F_1) and the light (L_5) to turn on when he turns the bathroom switch (S_2) on. Furthermore, when someone leaves the bathroom and turns the light (L_5) off again, the bathroom fan

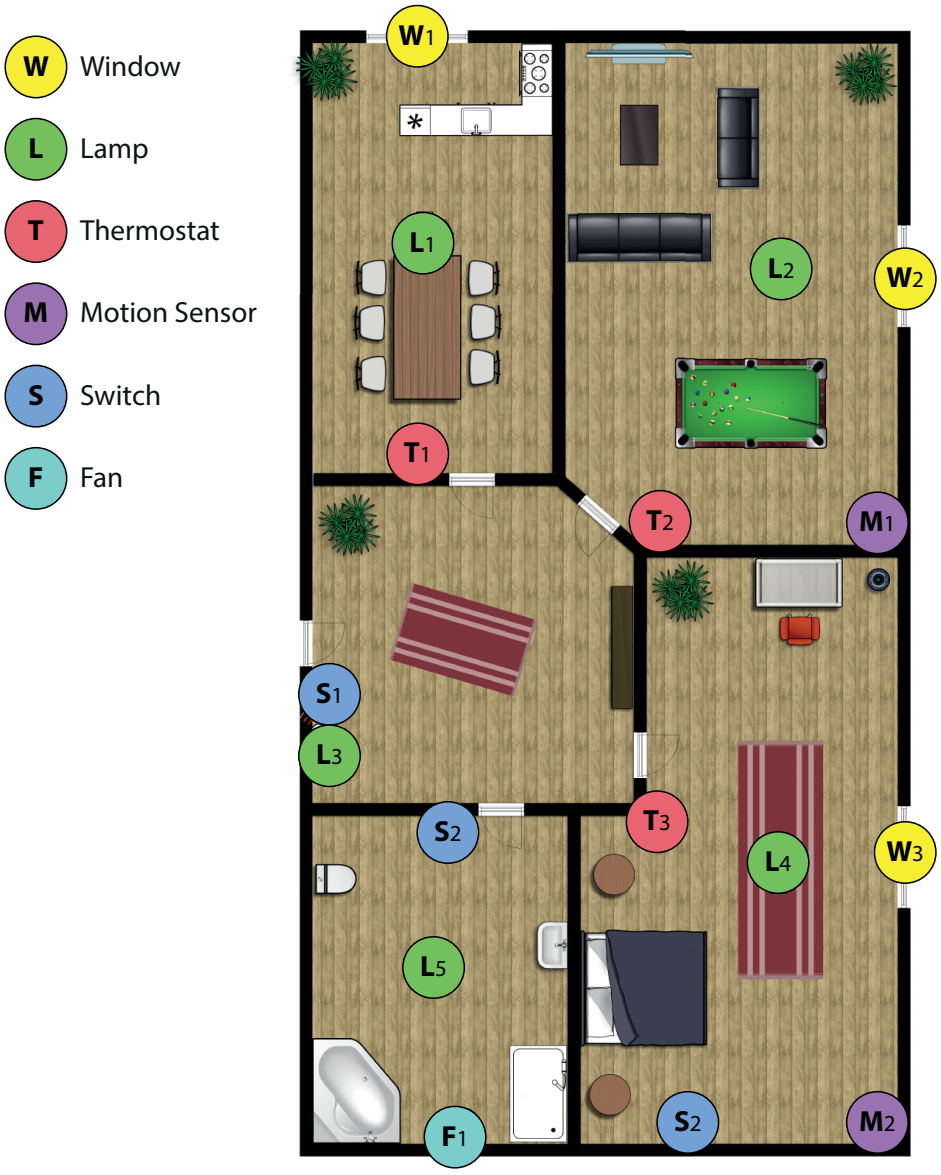


Figure 6.2: Overview of the Floor Plan

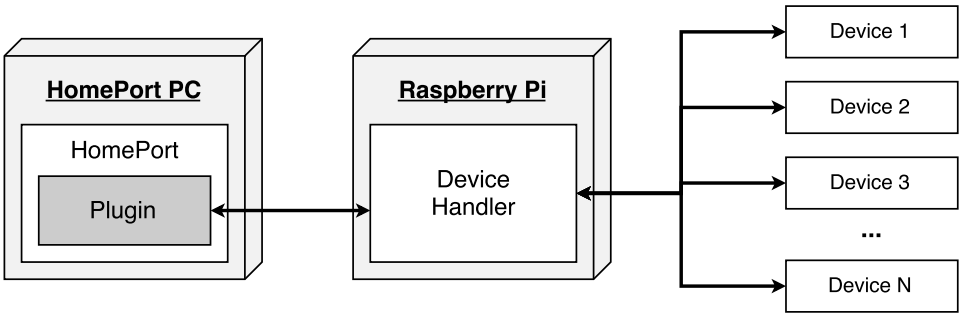


Figure 6.3: Overview of the Test Setup

(F_1) should keep going for another 15 minutes. This scenario has been transformed into two automation rules which can be seen in Listing 6.1.

```
event BathroomLightSwitch.OnOff cond BathroomLightSwitch.OnOff == ↩
  true action BathroomFan.OnOff true

event BathroomLightSwitch.OnOff cond BathroomLightSwitch.OnOff == ↩
  false action BathroomFan.OnOff false 600
```

Listing 6.1: Automation rule for bathroom fan

When Peter leaves his home for a vacation. He wants his lamps (L_1, L_2, L_4, and L_5) to turn on and off on a specified interval to mimic the behavior of him being at home. This scenario has been transformed into an automation rule which can be seen in Listing 6.2.

```
event 3600 cond true action KitchenLamp.OnOff true, BathroomLamp.↩
  OnOff true 10, BedroomLamp.OnOff true 20, KitchenLamp.OnOff ↩
  false 600, BathroomLamp.OnOff false, BedroomLamp.OnOff false
```

Listing 6.2: Automation rule for vacation

Furthermore, Peter also wants to save money on his energy bill and, therefore, wants the motion sensors (M_1 and M_2) in his home to turn the light (L_2 or L_4) on in a given room when motion is detected, and vice versa if no motion is detected turn the light off again. Because Peters wife is a restless sleeper these rules will only be active between 24:00 and 06:30. To express the desired behavior, four automation rules has been created which can be seen in Listing 6.3.

```
event LivingroomMotionSensor.Sensor cond LivingroomMotionSensor.↩
  Sensor == true action LivingroomLamp.OnOff == true within 06:30 ↩
  24:00
```

```
event LivingroomMotionSensor.Sensor cond LivingroomMotionSensor.↩
  Sensor == false action LivingroomLamp.OnOff == false

event BedroomMotionSensor.Sensor cond BedroomMotionSensor.Sensor ↩
  == true action BedroomLamp.OnOff == true within 06:30 24:00

event BedroomMotionSensor.Sensor cond BedroomMotionSensor.Sensor ↩
  == false action BedroomLamp.OnOff == false
```

Listing 6.3: Automation rule for motion sensors

All of the listed automation rules have been tested in our model home, and all of them work as expected. Each rule performed the task which was expressed by the automation rule language successfully. The only apparent problem is that if two or more automation rules control the same service concurrently, the outcome is unpredictable. As described in the analysis, this problem is well known, and solutions for it have been developed in other contexts.

6.3.2 Safety Rules

The premise for testing the Safety Rule Engine and the specified Safety Rule language is the same as with Automation Rules. The scenarios described in Chapter 3 will be implemented in the test setup in order to determine whether or not the language specification and engine implementation is adequate. Because of limitations of the physical devices in the test setup, some of the automation rules defined in the safety system scenarios will not work, and will therefore not be implemented in the system. Because of this, the affected safety rules will be tested with manual inputs.

Scenario 1

Peter wants to make sure that when a window is open in a room, the thermostat in that room is also turned off. In other words, if W_1 is open, T_1 must be turned off. If W_2 is open, T_2 must be turned off, and if W_3 is open, T_3 must be turned off. A safety rule expressing this behavior can be seen in Listing 6.4.

```
1 cond (LivingroomWindow.OpenClose == true && LivingroomThermostat.↩
    Temp > 0) || (KitchenWindows.OpenClose == true && ↩
    KitchenThermostat.Temp > 0) || (BedroomWindows.OpenClose == true↩
     && BedroomThermostat.Temp > 0)
```

Listing 6.4: Thermostats must be turned off when windows are open

This safety rule works as expected. Each logical pair of window and thermostat reacts identically. The safety rule issues warnings when a thermostat is on and its respective window open. It does not issue warnings if neither or only one or the other is on.

Scenario 2

When Peter is about to leave his home he always turns the alarm system (S_1) on. To save money on his heating bill, he wants all the thermostats (T_1, T_2, and T_3) to adjust their temperature to 15 degrees when the alarm system is turned on. A safety rule that monitors this behavior can be seen in Listing 6.5.

```
1 event AlarmSystem.OnOff cond AlarmSystem.OnOff == true always ↩
    LivingroomThermostat.Temp <= 15 && BedroomThermostat.Temp <= 15 ↩
    && KitchenThermostat.Temp <= 15 until AlarmSystem.OnOff == false
```

Listing 6.5: Thermostats must be turned down while no one is home

This safety rule works as expected. If one of the thermostats is set to more than 15 degrees as the alarm system turns on, a warning is issued. If one of the thermostats is turned to more than 15 degrees while the alarm system is on, a warning is issued as well.

Scenario 3

Peter would like to make sure that the windows are closed when the alarm system is turned on in accordance with the rule that he has set up. Since it takes a few seconds to close the windows, Peter would like to incorporate a period of 30 seconds in the rule. In short, when S_1 is turned on, W_1, W_2, and W_3 must be closed within a period of 30 seconds. The safety rule that expresses this behavior can be seen in Listing 6.6

```
1 event AlarmSystem.OnOff cond AlarmSystem.OnOff == true eventually ↩
    [group Windows type WindowOpenClose where # == true count == 0] ↩
    period 30
```

Listing 6.6: Windows must be closed when no one is home

This safety rule works as expected. If all windows are closed before the timer times out, the rule does not issue a warning. There is, however, an inherent problem with the way "eventually" safety rules work. If the alarm system is turned on, and the windows are closed within the time limit, the rule is deactivated, and nothing happens if the windows are once again opened. It seems reasonable that the intent of the safety rule in this

scenario is that the windows should be closed, and then remain closed for as long as the alarm system is on.

Scenario 4

Peter is not convinced that the extractor fan in the bathroom is on for as long as he would like it to be after the lights are turned off. He creates a safety rule to monitor the situation. The safety rule states that when S_2 is switched off, F_1 must be on for 30 minutes. The safety rule used for this scenario can be seen in Listing 6.7.

```
1 event BathroomSwitch.OnOff cond BathroomSwitch.OnOff == false ↩
    always BathroomFan.OnOff == true period 1800
```

Listing 6.7: The extractor fan must stay on after leaving the bathroom

This safety rule works as expected. If the extractor fan is turned off before the timer times out, a warning is issued.

Scenario 5

Later, Peter decides that he wants the extractor fan to only be on while the lights in the bathroom are on. He creates a safety rule to make sure that the setup is working as intended. The rule states that when S_1 is turned on, F_1 must also turn on and must stay on until S_1 is turned off. A safety rule expressing this behavior can be seen in Listing 6.8.

```
1 event BathroomLightSwitch.OnOff cond BathroomLightSwitch.OnOff == ↩
    true always BathroomFan.OnOff == true until BathroomLightSwitch.↩
    OnOff == false
```

Listing 6.8: The extractor fan must be on while the lights are on

In this scenario, an issue arrises as the logic for turning the bathroom fan on and off is defined using automation rules. One rule states that the fan must turn on when the switch is flipped on, and another states the opposite. When these automation rules and the above safety rule are active in the system at the same time, the outcome depends on the order of evaluation of the rules and the speed of the overall system. When the switch is flipped on, the automation rule will send a signal that the fan should turn on, and at the same time the safety rule will instantly check whether the fan is on or not. The safety rule reacts either before the automation rule or before the actions of the automation rule has had any effect on the physical devices. The result is that the safety rule will issue a warning because the fan was not on when the rule was evaluated, but it is obvious that in a lax context this behavior does not seem appropriate.

6.3.3 Grouping

Rules that implement the behaviors of the scenarios listed in Section 4.1 can be seen in Listings 6.9, 6.10, and 6.11. this shows that the grouping syntax is expressive enough to cover the proposed scenarios. However, support for the proposed queries has not yet been implemented, and as such, cannot be tested in the current system.

```
1 event AlarmSystem.OnOff cond AlarmSystem.OnOff == true action [↩
    group LightingUnits type LightOnOff do false]
```

Listing 6.9: Apply action to a group of services

```
1 event BedroomButton.OnOff cond BedroomButton.OnOff == false action↩
    [group LightingUnits exclude Bedroom type LightOnOff do false]
```

Listing 6.10: Apply action to subset of a group of services

```
1 event AlarmSystem.OnOff cond AlarmSystem.OnOff == true && [group ↩
    Windows type WindowOpenClose where # == true count > 0] action ↩
    WarningLamp.OnOff == true
```

Listing 6.11: Count the number of windows that are currently open

6.3.4 Findings

Since the automation rules work as expected and the grouping functionality cannot be tested due to not being implemented, this section will focus on the problems discovered during evaluation of the safety rule system.

The first problem is with the "eventually" safety rule construct, in which a condition that must eventually become true is expressed. In some cases, the actual intent of the user may be that the condition should eventually be true, and then stay true either for a specific duration, or until some other condition is satisfied.

The second problem happens when automation rules and safety rules depend on some of the same parameters, in which case the behaviour of the system depends on the order of evaluation of the rules, as well as the overall response speed in the system.

Even though these two problems are very different, a single solution may solve both of them by allowing the user to specify a *tolerance* in the safety rules that contain "always" conditions. This tolerance should be a time period that is allowed until the condition of the rule must be satisfied, which

effectively permits a lag in the evaluation of the safety rule. A safety rule that works as intended for the first problem can be seen in Listing 6.12

```
1 event AlarmSystem.OnOff cond AlarmSystem.OnOff == true always [↩
    group Windows type WindowOpenClose where # == true count == 0] ↩
    until AlarmSystem.OnOff == false tolerance 30
```

Listing 6.12: Safety rule with tolerance

Without the tolerance, this rule would state that for as long as the alarm system is turned on, all windows must remain closed. The tolerance gives the system 30 seconds of slack from the moment that the alarm system is turned on, until the windows must be closed. After this time, the windows must then remain closed until the alarm system is turned back off.

For the second problem, a tolerance would help by giving the environment time to react to the actions applied by automation rules. In this case, a small tolerance would allow the system to enter an unwanted state for a short amount of time before the given safety rule should be enforced. In strict, safety critical systems this may not be appropriate, but in many applications it will not be a problem to allow some slack in the safety system. Listing 6.13 contains a safety rule with a short delay that would solve this problem.

```
1 event BathroomLightSwitch.OnOff cond BathroomLightSwitch.OnOff == ↩
    true always BathroomFan.OnOff == true until BathroomLightSwitch.↩
    OnOff == false tolerance 0.5
```

Listing 6.13: Safety rule with tolerance

Because of the 500 ms tolerance that was added to the rule, the negative effect of the order of evaluation is negated, and the environment is allowed some time to react to the actions of automation rules. For casual applications it might even be appropriate to incorporate a default tolerance in all safety rules unless the user specifies otherwise, in order to make the overall system more lax.

6.4 Conclusion

In this project we have investigated how the home automation system HomePort can be extended to support automation of devices located on heterogeneous networks. We have analyzed three existing home automation solutions with different approaches on how to handle automation and identified missing components in HomePort that must be added to make it possible to support device automation.

HomePort was extended with an automation engine that supports event-condition-actions rules, because it makes the user independent of application development and does not depend on predefined device types in the system. Based on user scenarios that highlight possible use cases of device automation, a language that supports the creation of these rules was designed and implemented.

In the analyzed home automation systems, solutions for handling conflicts between rules and actions are given, but none of the systems provide means that can help users convince themselves that the system behaves correctly when the complexity of automation increases. To combat this, we have designed and implemented a system where users are able to define unintended behavior in the system, by the means of safety rules, in the same way as intended behavior is defined using automation rules.

HomePort has also been extended to support device and service groups, with the purpose of providing users with a better overview of available devices in the system and easing the creation of automation and safety rules. A language that can be used to query and perform actions on these groups has been designed but not implemented.

In the evaluation of HomePort, scenarios developed throughout the report have been used as a basis to investigate whether the final design of automation and safety rules are sufficient. The evaluation shows that safety rules in most situations can help users convince themselves that the system is always in a correct state, but also reveals that the safety rule language lacks expressiveness in some scenarios.

This project contributes a prototype implementation of an event-condition-action automation engine for HomePort, which makes it possible to automate devices located on heterogeneous networks. Contrary to the existing home automation solutions that have been analyzed in this report, HomePort makes it possible to define unintended system behavior through the use of safety rules, which has the purpose of ensuring intended system behavior.

6.5 Future Work

In this section we will discuss ideas for leveraging the solution that has been developed in this report by using the knowledge about the state of the system that safety rules provide.

6.5.1 Handling Unwanted States

The system of safety rules that has been developed throughout this report is able to detect when the system enters unwanted states for various reasons. In order to take advantage of such a system, we must consider whether the system should take action when an unwanted state is entered, and if so, what that action should be.u

Taking inspiration from the concept of rollback actions used in transactions in BOSS [2], one solution could be to specify a set of actions for each safety rule. When the safety rule discovers that the system has entered an unwanted state, its set of actions could be performed in order to recover to a safe state.

One could also imagine that a default state known to be safe could be defined for the system, or parts of the system. This would allow the system to fall back to this default state or behavior in case a safety rule detects unwanted behavior.

In other cases, the most appropriate course of action may be to simply warn the user that a violation has happened, and then allow them to correct the state of the system manually.

Ultimately, the best solution depends on the specific context in which the violation happens. Under strict circumstances, automatic corrections would usually be preferable, since they can be applied immediately in order to correct the system state as quickly as possible. Under less strict circumstances, such as the scenarios presented in this report, the most appropriate course of action may be to present the user with different suggestions as to how the unwanted behavior can be corrected, and then letting them make the decision in each case.

6.5.2 Cooperation Between Automation and Safety Rules

In the forms that they have been described so far, automation and safety rules are completely separate entities. A safety rule will only act when a state change event is published, and its condition is not satisfied (i.e. when something has already gone wrong). The current flow of the indirect interaction between automation and safety rules can be seen in Figure 6.4. In this example, a rule is triggered and its action is applied with no regard to active safety rules. After the value changes, the safety rules will be able to

pick up on the change because of the event system, and will discover that the system is in an unwanted state.

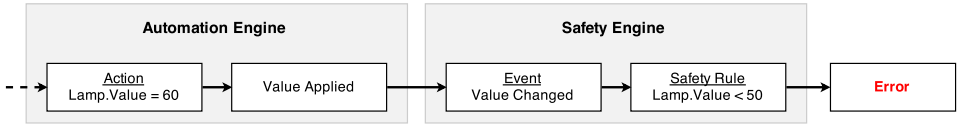


Figure 6.4: Automation and safety rule with no cooperation.

In order to better utilize the functionality of the safety rules, we can change their functionality to allow them to intercept actions that would create an unwanted state if they were applied. In order to achieve this, a system that intercepts the actions of all automation rules and validates them according to the safety rules could be implemented.

This can be done by simulating each action and evaluating the conditions of the safety rules. While evaluating a condition, when we encounter the service that would be affected by the action, the action's intended value should be used instead of a live one from the service. Each type of safety rule (as described in Chapter 3) should be handled differently when simulating actions.

The simplest of the safety rules that simply consist of a single condition should always be tested when actions are applied, since they contain no temporal constraints.

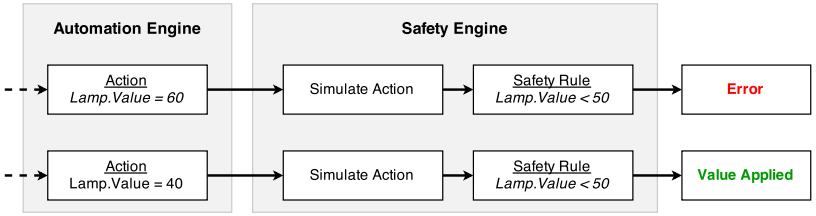


Figure 6.5: Automation and safety rule with simple simulation.

Figure 6.5 illustrates the proposed flow of interaction between automation and safety rules with no temporal constraint. In the examples, an automation rule is triggered, but before its action is applied, the safety rule's condition is tested with a simulated value. If the simulation violates the safety rule's condition, applying the action would cause the system to enter an unwanted state.

The three types of safety rules that do contain temporal constraints should only be simulated when the specific rule is actively monitoring its condition(s).

- **Always safety rule**

In this case, the safety rule's condition must always be true while its timer is active. Therefore, when simulating actions, the conditions of these safety rules must be tested only while its timer is active. If the condition evaluates to `false`, the action will violate the rule.

- **Eventually safety rule**

  Since these rules reason about something that must eventually be true, we cannot say anything about whether or not a single action would cause a violation with the rule, and as such there is no benefit from simulation. If the action happens to satisfy the condition, the rule will terminate its timer when the action is applied, and if not, the rule will simply keep waiting to see if its condition will eventually be satisfied.

- **Always Until safety rule**

  This type of safety rule must be simulated when it has been triggered, but the *Until* condition still has not been satisfied. In this case, the action must be simulated against the *Always* condition. If this condition evaluates to `false`, the application of the action will cause the system to enter an unwanted state.

At first glance the original functionality of the safety rules (i.e. the ability to passively monitor changes) may seem redundant if actions can be simulated, but this is indeed still useful in cases where outside forces change the values of devices connected to HomePort (e.g. manual user interaction).

When a simulation determines that an action would cause the system to enter an unwanted state the best course of action depends on the context. The action may be part of a collection of actions that a rule is currently applying, and in this case some actions may already have been applied, and some may be queued for application after the violating action. In the case of actions that are yet to be applied, a decision must be made as to whether or not they should be applied or not after an action in the chain has been blocked as a result of simulation. In the case of actions that have already been applied, a decision must be made as to whether they should be rolled back or not, and if so, how this can be done in an appropriate manner. Ultimately, the most appropriate course of action depends entirely on the context, and as such it should be left up to the user to define this behavior for each rule.

Bibliography

[1] ZigBee Alliance. Zigbee. `http://zigbee.org/`. [Online; accessed 17-02-2015]. 13

[2] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David E Culler. Boss: Building operating system services. 17, 18, 66

[3] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A. J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An operating system for the home. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 25–25, Berkeley, CA, USA, 2012. USENIX Association. 16

[4] Microsoft. Microsoft – AppDomain Class. `https://msdn.microsoft.com/en-us/library/system.appdomain`. [Online; acccessed 03-06-2015]. 55

[5] Mono Project. Mono Project – Cross platform, Open source .NET framework. `http://www.mono-project.com/`. [Online; acccessed 23-02-2015]. 55

[6] Christian Mortensen Søren Knudsen, Brian Holbech. Homeport - Dynamic Loading and Isolation of Device Adapters in Homeport. Technical report, Department of Computer Science, Aalborg University, Jan 2015. 13, 55

[7] H. Takatsuka, S. Saiki, S. Matsumoto, and M. Nakamura. A rule-based framework for managing context-aware services based on heterogeneous and distributed web services. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014 15th IEEE/ACIS International Conference on*, pages 1–6, June 2014. 19

A | REST API Specification

This gives an overview of how the REST API should be used. In each table a request URI as well as the request type (GET, PUT, POST, DELETE) is specified. The request and response format is JSON and if a request returns data, an example JSON object is shown. In requests that requires a body, optional and required object fields as well as query parameters are described.

GET: /devices/{id} Returns a specific device based on a the device id.	

Example Result	{
	"identifier": "RPIPlugin$23",
	"name": "Bedroom Lamp",
	"description": "Bedroom",
	"pluginId": "RPIPlugin",
	"services": [
	{
	"identifier": "1",
	"name": "On/Off",
	"description": "Turn lamp on or off",
	"type": "System.Boolean",
	"parameter": [
	true,
	false
	],
	"value": false,
	"deviceId": "RPIPlugin$23"
	}
	]
	}

GET: /devices
Returns a list of the installed devices in HomePort.

Example Result

```
[
    {
        "identifier": "RPIPlugin$11",
        "name": "Livingroom Window",
        "description": "Livingroom",
        "pluginId": "RPIPlugin",
        "services": [
            {
                "identifier": "1",
                "name": "Open/Close",
                "description": "Open or close window",
                "type": "System.Boolean",
                "parameter": [
                    true,
                    false
                ],
                "value": false,
                "deviceId": "RPIPlugin$11"
            }
        ]
    },
    {
        "identifier": "RPIPlugin$12",
        "name": "Livingroom Lamp",
        "description": "Livingroom",
        "pluginId": "RPIPlugin",
        "services": [
            ...
        ]
    }
]
```

PUT: /devices/{id}
Modifies an existing device based on the device id.

Fields	`name`	Name of the device (Optional).
	`description`	Description of the device (Optional).

GET: /devices/{id}/services/{id}
Returns a specific service based on the device and service id. The first id is a device id, and the second is a service id.

Example Result

```
{
    "identifier": "1",
    "name": "On/Off",
    "description": "Turn lamp on or off",
    "type": "System.Boolean",
    "parameter": [
        true,
        false
    ],
    "value": false,
    "deviceId": "RPIPlugin$23"
}
```

GET: /devices/{id}/services
Return a list of all of the services belonging to the specific device in question.

Example Result

```
[
    {
        "identifier": "1",
        "name": "On/Off",
        "description": "Turn lamp on or off",
        "type": "System.Boolean",
        "parameter": [
            true,
            false
        ],
        "value": false,
        "deviceId": "RPIPlugin$23"
    }
]
```

PUT: /devices/{id}/services/{id}
Modifies an existing service based on the provided device and service id. It is only possible to change the meta-information and the value of the service.

Fields	`name`	Name of the service (Optional).
	`description`	Description of the service (Optional).
	`value`	Value of the service(Only on actuator).

GET: /plugins/{id} Returns a specific plugin based on the plugin id.	

Example Result	{
	"identifier": "RPIPlugin",
	"name": "RPI Plugin",
	"description": "Plugin to control GPIO on Raspberry PI",
	"isActive": true
	}

GET: /plugins Returns a list of all of the installed plugins in HomePort.	

Example Result	[
	{
	"identifier": "RPIPlugin",
	"name": "RPI Plugin",
	"description": "Plugin to control GPIO on Raspberry PI",
	"isActive": true
	}
	]

PUT: /plugins/{id} Modifies the state of a plugin to either active or deactivate.		
Fields	`name`	Name of the plugin (Optional).
	`description`	Description of the plugin (Optional).
	`isActive`	Activate/deactivate the plugin (Optional).

GET: /devicegroups/{id}
Returns a specific device group based on the device Group id.

Example Result

```
{
    "identifier": 1,
    "name": "Kitchen",
    "description": "Devices located
        in the kitchen",
    "devices": [
        {
            "device": "RPIPlugin$16"
        },
        {
            "device": "RPIPlugin$18"
        },
        {
            "device": "RPIPlugin$22"
        }
    ]
}
```

GET: /devicegroups
Returns a list of all of the device groups in HomePort.

Example Result

```
[
    {
        "identifier": 1,
        "name": "Kitchen",
        "description": "Devices located in
            the kitchen",
        "devices": [
            {
                "device": "RPIPlugin$16"
            },
            {
                "device": "RPIPlugin$18"
            },
            {
                "device": "RPIPlugin$22"
            }
        ]
    },
    {
        "identifier": 2,
        "name": "Indoor",
        "description": "Devices located indoors",
        "devices": [
            {
                "device": "RPIPlugin$3"
            }
            {
                "identifier": 1,
                "name": "Kitchen",
                "description": "Devices located
                    in the kitchen",
                "devices": [
                    {
                        "device": "RPIPlugin$16"
                    },
                    {
                        "device": "RPIPlugin$18"
                    },
                    {
                        "device": "RPIPlugin$22"
                    }
                ]
            }
        ]
    }
]
```

POST: /devicegroups
Create a new device group in HomePort. The POST data should contain a name, and an array of device groups id or service ids.

Fields	`name`	Name of the device group (Required).
	`description`	Description of the device group (Required).
	`devices`	An array of ids of devices or existing device groups (Required).

Example Result

```
{
    "identifier": 1,
    "name": "Kitchen",
    "description": "Devices located
        in the kitchen",
    "devices": [
        {
            "device": "RPIPlugin$16"
        },
        {
            "device": "RPIPlugin$18"
        },
        {
            "device": "RPIPlugin$22"
        }
    ]
}
```

PUT: /devicegroups/{id}
Modifies an existing device group in HomePort.

Fields	`name`	Name of the device group (Optional).
	`description`	Description of the device group (Optional).
	`devices`	An array of ids of devices or existing device groups (Optional).

DELETE: /devicegroups/{id}
Deletes a specific group based on the device group id.

GET: /servicegroups/{id}
Returns a specific service group based on the service group id.

Example Result

```
{
    "identifier": 1,
    "name": "On/Off",
    "description": "Turn lamps On/Off",
    "serviceType": "HomePort.Data.IController",
    "dataType": "System.Boolean",
    "services": [
        {
            "device": "RPIPlugin$18",
            "service": "1"
        },
        {
            "device": "RPIPlugin$8",
            "service": "1"
        }
    ]
}
```

GET: /servicegroups
Returns a list of all of the service groups in HomePort.

Example Result

```
[
    {
        "identifier": 1,
        "name": "On/Off",
        "description": "Turn lamps On/Off",
        "serviceType": "HomePort.Data.IController",
        "dataType": "System.Boolean",
        "services": [
            {
                "device": "RPIPlugin$18",
                "service": "1"
            },
            {
                "device": "RPIPlugin$8",
                "service": "1"
            }
        ]
    }
]
```

<table>
<tr><td colspan="3">POST: /servicegroups
Create a new service group in HomePort. The POST data should contain a name, data type, and an array of service ids.</td></tr>
<tr><td rowspan="2">Params</td><td><code>servicetype</code></td><td>The type of services in the service group. Can either be sensor, actuator or controller (Required).</td></tr>
<tr><td><code>datatype</code></td><td>The data type of services in the service group. Can either be int, double, bool or string (Required).</td></tr>
<tr><td>Example Request</td><td colspan="2">POST /servicegroups?servicetype=controller&datatype=bool</td></tr>
<tr><td rowspan="3">Fields</td><td><code>name</code></td><td>Name of the service group (Required).</td></tr>
<tr><td><code>description</code></td><td>Description of the service group (Required).</td></tr>
<tr><td><code>devices</code></td><td>An array of service ids (Required).</td></tr>
<tr><td>Example Result</td><td colspan="2"><pre>{
    "identifier": 1,
    "name": "On/Off",
    "description": "Turn lamps On/Off",
    "serviceType": "HomePort.Data.IController",
    "dataType": "System.Boolean",
    "services": [
        {
            "device": "RPIPlugin$18",
            "service": "1"
        },
        {
            "device": "RPIPlugin$8",
            "service": "1"
        }
    ]
}</pre></td></tr>
</table>

<table>
<tr><td colspan="3">PUT: /servicegroups/{id}
Modifies an existing service group in HomePort.</td></tr>
<tr><td rowspan="3">Fields</td><td><code>name</code></td><td>Name of the service group (Optional).</td></tr>
<tr><td><code>description</code></td><td>Description of the service group (Optional).</td></tr>
<tr><td><code>devices</code></td><td>An array of service ids (Optional).</td></tr>
</table>

<table>
<tr><td>DELETE: /servicegroups/{id}
Deletes a specific service groups based on the id provided in the URI.</td></tr>
</table>

GET: /rules/{id}
Returns a specific rule based on the id provided in the URI.

Example Result

```
{
    "identifier": 1,
    "events": [
        {
            "device": "RPIPlugin$5",
            "service": "1"
        }
    ],
    "cond": {
        "cond": {
            "left": {
                "device": "RPIPlugin$5",
                "service": "1"
            },
            "operator": "==",
            "right": true
        }
    },
    "actions": [
        {
            "device": "RPIPlugin$10",
            "service": "1",
            "value": true,
            "delay": 0
        }
    ]
}
```

GET: /rules
Returns a list of all the rules in HomePort.

Example Result

```
[
    {
        "identifier": 1,
        "events": [
            {
                "device": "RPIPlugin$5",
                "service": "1"
            }
        ],
        "cond": {
            "cond": {
                "left": {
                    "device": "RPIPlugin$5",
                    "service": "1"
                },
                "operator": "==",
                "right": true
            }
        },
        "actions": [
            {
                "device": "RPIPlugin$10",
                "service": "1",
                "value": true,
                "delay": 0
            }
        ]
    },
    {
        "identifier": 2,
        "events": [
            {
                "interval": {
                    "cron": "0 0/1 * 1/1 * ? *"
                }
            }
        ],
        "actions": [
            {
                "device": "RPIPlugin$18",
                "service": "1",
                "value": true,
                "delay": 0
            }
        ]
    }
]
```

<table>
<tr><td colspan="3">POST: /rules
Creates a new rule in HomePort. The POST data should contain a rule in a valid format for it to be accepted by HomePort.</td></tr>
<tr><td rowspan="4">Fields</td><td><code>event</code></td><td>Event for triggering evaluation of the rule (Required).</td></tr>
<tr><td><code>cond</code></td><td>Condition that must be satisfied for actions to be triggered(Required).</td></tr>
<tr><td><code>action</code></td><td>Actions to be performed (Required).</td></tr>
<tr><td><code>within</code></td><td>Time period where the rule should be active (Optional).</td></tr>
<tr><td>Example Result</td><td colspan="2"><pre>
{
    "identifier": 1,
    "events": [
        {
            "device": "RPIPlugin$5",
            "service": "1"
        }
    ],
    "cond": {
        "cond": {
            "left": {
                "device": "RPIPlugin$5",
                "service": "1"
            },
            "operator": "==",
            "right": true
        }
    },
    "actions": [
        {
            "device": "RPIPlugin$10",
            "service": "1",
            "value": true,
            "delay": 0
        }
    ]
}
</pre></td></tr>
</table>

PUT: /rules/{id} Modify an existing rule in HomePort. The PUT data should contain a rule in a valid format for it to be accepted by HomePort.		
Fields	`events`	Event for triggering evaluation of the rule (Optional).
	`cond`	Condition that must be satisfied for actions to be triggered (Optional).
	`actions`	Actions to be performed (Optional).
	`within`	Time period where the rule should be active (Optional).

DELETE: /rules/{id} Delete a specific rule from HomePort, which is selected by the id from the URI.

GET: /safetyrules/{id} Returns a specific safety rule based on the id provided in the URI.	
Example Result	`{ "identifier": 4, "cond": { "cond": { "left": { "device": "RPIPlugin$11", "service": "1" }, "operator": "==", "right": true }, "boolOp": "&&", "expr": { "cond": { "left": { "device": "RPIPlugin$13", "service": "1" }, "operator": ">", "right": 0 } } } }`

GET: /safetyrules
Returns a list of all the safety rules in HomePort.

Example Result


```
[
    {
        "identifier": 3,
        "cond": {
            "cond": {
                "value": true
            }
        },
        "always": {
            "cond": {
                "left": {
                    "device": "RPIPlugin$10",
                    "service": "1"
                },
                "operator": "==",
                "right": true
            }
        },
        "period": 10,
        "events": [
            {
                "device": "RPIPlugin$5",
                "service": "1"
            }
        ]
    },
    {
        "identifier": 4,
        "never": {
            "cond": {
                "left": {
                    "device": "RPIPlugin$11",
                    "service": "1"
                },
                "operator": "==",
                "right": true
            }
        }
    }
]
```

<table>
<tr><td colspan="3">POST: /safetyrules
Creates a new safety rule in HomePort. This schema shows four different safety rules that can be created.</td></tr>
<tr><td colspan="3">Safety rule 1</td></tr>
<tr><td rowspan="2">Fields</td><td><code>cond</code></td><td>Condition that if satisfied, will result in an unwanted state (Required).</td></tr>
<tr><td><code>within</code></td><td>Time period where the rule should be active (Optional).</td></tr>
<tr><td>Example Result</td><td colspan="2"><pre>
{
    "identifier": 4,
    "cond": {
        "cond": {
            "left": {
                "device": "RPIPlugin$11",
                "service": "1"
            },
            "operator": "==",
            "right": true
        },
        "boolOp": "&&",
        "expr": {
            "cond": {
                "left": {
                    "device": "RPIPlugin$13",
                    "service": "1"
                },
                "operator": ">",
                "right": 0
            }
        }
    }
}
</pre></td></tr>
</table>

<table>
<tr><th colspan="3">Safety rule 2</th></tr>
<tr><td rowspan="5">Fields</td><td><code>events</code></td><td>Event for triggering evaluation of the condition, <code>cond</code> (Required).</td></tr>
<tr><td><code>cond</code></td><td>Condition that if satisfied, will cause the entire safety rule to be evaluated (Required).</td></tr>
<tr><td><code>always</code></td><td>Condition that should always be satisfied (Required).</td></tr>
<tr><td><code>for</code></td><td>Time unit where the <code>always</code> condition must be true (Required).</td></tr>
<tr><td><code>within</code></td><td>Time period where the rule should be active (Optional).</td></tr>
<tr><td>Example Result</td><td colspan="2"><pre>
{
    "identifier": 3,
    "cond": {
        "cond": {
            "left": {
                "device": "RPIPlugin$5",
                "service": "1"
            },
            "operator": "==",
            "right": false
        }
    },
    "always": {
        "cond": {
            "left": {
                "device": "RPIPlugin$10",
                "service": "1"
            },
            "operator": "==",
            "right": true
        }
    },
    "period": 10,
    "events": [
        {
            "device": "RPIPlugin$5",
            "service": "1"
        }
    ]
}
</pre></td></tr>
</table>

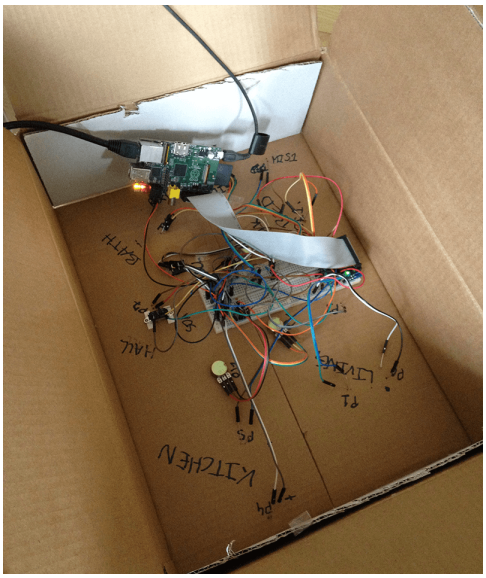
<table>
<tr><th colspan="3">Safety rule 3</th></tr>
<tr><td rowspan="5">Fields</td><td><code>events</code></td><td>Event for triggering evaluation of the condition, <code>cond</code> (Required).</td></tr>
<tr><td><code>cond</code></td><td>Condition that if satisfied, will cause the entire safety rule to be evaluated (Required).</td></tr>
<tr><td><code>eventually</code></td><td>Condition that must eventually be satisfied (Required).</td></tr>
<tr><td><code>for</code></td><td>Time unit that specifies the latest point in time where the <code>eventually</code> condition must be satisfied (Required).</td></tr>
<tr><td><code>within</code></td><td>Time period where the rule should be active (Optional).</td></tr>
<tr><td>Example Result</td><td colspan="2"><pre>
{
    "identifier": 2,
    "cond": {
        "cond": {
            "left": {
                "device": "RPIPlugin$3",
                "service": "1"
            },
            "operator": "==",
            "right": true
        }
    },
    "eventually": {
        "cond": {
            "left": {
                "device": "RPIPlugin$11",
                "service": "1"
            },
            "operator": "==",
            "right": false
        }
    },
    "period": 5,
    "events": [
        {
            "device": "RPIPlugin$3",
            "service": "1"
        }
    ]
}
</pre></td></tr>
</table>

<table>
<tr><th colspan="3">Safety rule 4</th></tr>
<tr><td rowspan="5">Fields</td><td><code>events</code></td><td>Event for triggering evaluation of the condition, <code>cond</code> (Required).</td></tr>
<tr><td><code>cond</code></td><td>Condition that if satisfied, will cause the entire safety rule to be evaluated (Required).</td></tr>
<tr><td><code>always</code></td><td>Condition that should be satisfied until the <code>until</code> condition is satisfied (Required).</td></tr>
<tr><td><code>until</code></td><td>Condition that defines when the <code>always</code> condition is allowed to be false (Required).</td></tr>
<tr><td><code>within</code></td><td>Time period where the rule should be active (Optional).</td></tr>
<tr><td>Example Result</td><td colspan="2"><pre>
{
    "identifier": 1,
    "events": [
        {
            "device": "RPIPlugin$3",
            "service": "1"
        }
    ],
    "cond": {
        "cond": {
            "value": true
        }
    },
    "always": {
        "cond": {
            "left": {
                "device": "RPIPlugin$22",
                "service": "1"
            },
            "operator": "<=",
            "right": 15
        }
    },
    "until": {
        "cond": {
            "left": {
                "device": "RPIPlugin$3",
                "service": "1"
            },
            "operator": "==",
            "right": false
        }
    }
}
</pre></td></tr>
</table>

PUT: /safetyrules/{id}
Modify an existing safety rule in HomePort. The object fields depend on the type of safety rule that should be edited. The four types of rules and their specific fields can be seen in the specification for a POST request. All fields are optional.

DELETE: /safetyrules/{id}
Deletes a specific safety rule from HomePort, which is selected by the id from the URI.

B | Test Setup



C | Automation Rules

```
rule          := 'event' eventExp 'cond' condExp 'action'
                 actionExp within
eventExp      := serviceValue eventExp
               | interval eventExp
               | ϵ
condExp       := cond boolOp condExp | cond
cond          := value binOp value | true | false
               | '(' condExp ')'
               | '!' condExp
               | groupAggrExp
value         := literalValue | serviceValue
literalValue  := int | double | string | bool
serviceValue  := device '.' service
binOp         := '>' | '>=' | '<' | '<=' | '==' | '!='
boolOp        := '&&' | '||'
actionExp     := serviceValue literalValue delay actionExp
               | groupActExp delay actionExp | ϵ
delay         := int | ϵ
within        := 'within' interval interval | ϵ
interval      := int | string
groupActExp   := '[' groupExp 'do' literal ']'
groupAggrExp  := '[' groupExp aggrFunc binOp literal ']'
groupExp      := 'group' groupType excludeExp 'type' type whereExp
groupType     := '*' | groupCom
groupCom      := group ',' groupCom | group
excludeExp    := 'exclude' groupCom | ϵ
whereExp      := 'where' whereCondExp | ϵ
whereCondExp  := whereCond boolOp whereCondExp | whereCond
whereCond     := whereValue binOp whereValue | true | false
               | '(' whereCondExp ')'
               | '!' whereCondExp
               | groupAggrExp
whereValue    := literal | device '.' service | #
literal       := int | double | string | bool
aggrFunc      := 'count' | 'min' | 'max' | 'avg'
```

D | Safety Rules

```
safetyRule    := 'cond' condExp within
               | 'event' eventExp 'cond' condExp safetyType within
safetyType    := 'always' condExp 'for' uint
               | 'always' condExp 'until' condExp
               | 'eventually' condExp 'for' uint
condExp       := cond boolOp condExp | cond
cond          := value binOp value | true | false
               | '(' condExp ')'
               | '!' condExp
               | groupAggrExp
value         := literalValue | serviceValue
literalValue  := int | double | string | bool
serviceValue  := device '.' service
binOp         := '>' | '>=' | '<' | '<=' | '==' | '!='
boolOp        := '&&' | '||'
eventExp      := serviceValue '||' eventExp | serviceValue
within        := 'within' interval interval | ϵ
interval      := int | string
groupActExp   := '[' groupExp 'do' literal ']'
groupAggrExp  := '[' groupExp aggrFunc binOp literal ']'
groupExp      := 'group' groupType excludeExp 'type' type whereExp
groupType     := '*' | groupCom
groupCom      := group ',' groupCom | group
excludeExp    := 'exclude' groupCom | ϵ
whereExp      := 'where' whereCondExp | ϵ
whereCondExp  := whereCond boolOp whereCondExp | whereCond
whereCond     := whereValue binOp whereValue | true | false
               | '(' whereCondExp ')'
               | '!' whereCondExp
               | groupAggrExp
whereValue    := literal | device '.' service | #
literal       := int | double | string | bool
aggrFunc      := 'count' | 'min' | 'max' | 'avg'
```