POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Improving Trigger-Action Programming in Smart Buildings through Suggestions Based on Behavioral Graphs Analysis

Relatore: Prof. Donatella SCIUTO

Correlatore: Dott. Alessandro A. NACCI

Tesi di Laurea di:

Jacopo FIORENZA, matricola 814218

Andrea MARIANI, matricola 814508

Anno Accademico 2014-2015

Ringraziamenti

Ringraziamo anzitutto la nostra Relatrice Prof. Sciuto, per averci seguito in questo percorso di tesi dandoci dei consigli chiari e precisi.

Un ringraziamento speciale va al nostro correlatore Alessandro (Lo Zio) per averci guidato in questo percorso di tesi a colpi di "sarebbe bello se..". Ci hai insegnato molto ed e' stato veramente un enorme piacere lavorare con te! Grazie di cuore!

Un ringraziamento va a tutte le persone del NECSTLab, in particolare al Prof. Santambrogio (Santa) che ci ha accolti nel laboratorio credendo in noi fin dal principio e ci ha permesso di vivere delle esperienze indimenticabili.

Nella vita si continua a migliorare soprattutto grazie agli insegnamenti ricevuti e alle persone incontrate. Noi abbiamo incontrato un trio che ha letteralmente cambiato la nostra vita: Ale, Rik e TeoF (i veri Giovani di Talento). Hanno creduto in noi includendoci nei loro progetti e ora ci ritroviamo a condividere una scrivania per lavorare insieme. Grazie di Tutto!

Infine un ringraziamento va a tutti i nostri amici che ci hanno seguito, anche sem-

plicemente con una birra, durante il nostro corso di studi. Merita un ringraziamento particolare Asna (detto Asnaghiñovic, asso del Milan) per aver condiviso con noi ogni singolo giorno.

Jacopo e Andrea

Un sincero ringraziamento va a tutti coloro che mi hanno aiutato, anche con un semplice sorriso, durante questo percorso.

In primo luogo ringrazio i miei genitori per avermi dato la possibilità di compiere questo cammino e per aver creduto in me, dandomi sempre piena fiducia. Senza di voi non sarei mai riuscito a completare questo percorso impegnativo. Grazie per avermi insegnato a non mollare mai!

Desidero ringraziare la mia fidanzata Lisa, per essermi sempre stata vicina e per avermi preso per mano anche nei momenti difficili. Non so come avrei fatto senza i tuoi abbracci e i tuoi sorrisi! Grazie di tutto!

Meritano un ringraziamento anche i miei nonni, tanto fieri ed orgogliosi di suo nipote Ingegnere, Grazie Nonni!

Un particolare ringraziamento ai ragazzi con cui ho condiviso molti anni della mia vita. Sono arrivato alla fine di questo lungo percorso anche grazie a voi! Dicono sempre che chi trova un amico trova un tesoro, è proprio vero! Grazie Bucch, Ste e Benny!

Un enorme GRAZIE va al ragazzo che vedo in media 14 ore al giorno (sicuramente lo vedo più della mia ragazza e della mia famiglia). Insieme abbiamo superato qualsiasi ostacolo! Non c'è niente da fare, siamo troppo forti! Grazie Jaco!

Infine (ma non per questo meno importante) ringrazio mia sorella. Non avrei potuto sperare in una compagnia migliore per crescere e condividere tanti momenti speciali. Le

chiacchierate prima di dormire sono sempre le migliori! Grazie Vale!

"I've missed more than 9000 shots in my career. I've lost almost 300 games. 26 times, I've been trusted to take the game winning shot and missed. I've failed over and over and over again in my life. And that is why I succeed." - Michael Jordan

Andrea

Innanzitutto, il ringraziamento più grande va ai miei genitori, senza i quali non sarei stato in grado di compiere questo percorso. Grazie per avermi sempre sostenuto e aver creduto sempre in me, anche nei momenti più difficili. Grazie soprattutto per avermi sempre dato l'opportunità di crescere sotto ogni punto di vista. Non potrei desiderare genitori migliori di voi.

Un enorme grazie lo merita sicuramente la mia fidanzata Vanessa per essere sempre stata presente da 4 anni a questa parte, per aver gioito con me e avermi sempre dato man forte. Grazie di cuore.

Non potrei mai non ringraziare anche le mie nonne, in particolare la "nonna Mina", che mi ha cresciuto fin da piccolo e che tuttora ci tiene a sapere se ho "mangiato tutto". Un bacione nonna!

Un grazie va anche a Riky, Andre e Same, amici da una vita e compagni di suonate il lunedì sera!

Per ultimo, ci terrei a ringraziare Andre, compagno durante tutto il percorso di studi, compagno di tesi, socio e, cosa più importante, grande amico. Un grazie va a te per aver condiviso con me ogni momento di questi ultimi 5 anni.

Jacopo

Summary

Programming a smart building so that its behavior reflects the user's needs, is not a simple task, and often requires technical skills. As shown in the state of the art, the trigger-action paradigm is a method that has been demonstrated to be perceived as natural by the users to describe the behavior of a smart building. Unfortunately, using such paradigm, the interaction problems rise when many users insert their preferences in the system, as in case of commercial buildings. In fact, the aforementioned problems emerge, first, due to the possible conflicts between user preferences and, second, due to the difficulty of the users to understand how the building will behave. To solve these problems, we have developed a methodology that improves trigger-action programming providing suggestions about the building status. The suggestions are generated by the analyses carried out on graphs representing the behavior of the building. We have tested the system both in a real building (Joint Open Lab of Telecom Italia in Milan) and in a virtual environment. The results obtained showed that the proposed methodology improves the usability of the system, giving a clearer vision on the behavior of the building to the occupants, and an easy integration with actual Building Management Systems.

Sommario

Programmare un edificio intelligente in modo che il suo comportamento rispecchi le necessità dell'utente, non è un compito semplice, e spesso richiede competenze informatiche specifiche. Come mostrato nello stato dell'arte, il paradigma trigger-action è un metodo è stato dimostrato essere percepito come naturale dagli utenti che necessitano di descrivere il comportamento di un edificio intelligente. Sfortunatamente, utilizzando tale paradigma, i problemi di interazione sorgono quando sono presenti molti utenti che inseriscono le proprie preferenze nel sistema, come nel caso di un edificio commerciale. In questo contesto nascono problematiche dovute sia ai possibili conflitti tra le preferenze degli utenti che alla difficoltà di questi ultimi a comprendere l'effettivo comportamento che avrà l'edificio. Per risolvere questi problemi, abbiamo sviluppato una metodologia che migliora il paradigma trigger-action introducendo suggerimenti riguardanti il comportamento dell'edificio, ricavati effettuando analisi su grafi. Abbiamo installato il nostro sistema sia in un edificio reale (Joint Open Lab di Telecom Italia a Milano) che in un ambiente virtuale. I risultati ottenuti hanno evidenziato che la metodologia proposta migliora l'usabilità del sistema, dando una più chiara visione del comportamento dell'edificio, e che il sistema è di facile integrazione con un Building Management System reale.

Contents

List of Figures

List of Tables

Estratto in lingua italiana

Gli edifici, oggi, sono un insieme complesso di strutture, sistemi e tecnologie. Nel corso del tempo, ciascuno dei componenti all'interno di un edificio è stato costantemente sviluppato e migliorato, permettendo alla gente che li occupa di controllare i sistemi di illuminazione, di sicurezza, di riscaldamento e condizionamento mediante un'interazione digitale con gli oggetti fisici, utilizzando le *Tecnologie dell'informazione e della comunicazione* (TIC). Gli edifici correnti forniscono servizi utili al fine di rendere i propri occupanti più produttivi ma riducendo sia il costo energetico che l'impatto ambientale. Questo tipo di edificio prende il nome di edificio intelligente.

Gli edifici intelligenti sono, quindi, edifici abilitati alla cooperazione di oggetti (ad esempio sensori, dispositivi ed elettrodomestici) e sistemi che hanno la capacità di auto-organizzarsi dato alcune politiche [1]. Il significato di edificio intelligente, però, non è lo stesso in tutti gli ambiti in cui viene utilizzato [2]. In realtà, dal punto di vista degli utilizzatori finali, un edificio è intelligente se è possibile accedervi in remoto per accendere o spegnere dispositivi, anche se, potenzialmente, potrebbe non esserci alcun tipo di automazione effettivamente messa in atto. Dal punto di vista di ricercatori ed esperti, invece, un edificio è intelligente quando è attento ai residenti presenti al suo interno ed è

in grado di adattarsi autonomamente alle situazioni in cui si potrebbe trovare. Un esempio potrebbe essere quello di fare uso di algoritmi di apprendimento automatico per prevedere quando un utente sarà presente o meno nell'edificio per controllare in maniera efficiente il sistema di riscaldamento. Nell'industria, un edificio intelligente ha ancora un altro significato; questa parola è, infatti, generalmente utilizzata semplicemente come un termine per fare campagne di marketing in cui vengono descritte le nuove tecnologie. All'interno della nostra tesi, per edificio intelligente si intenderà un edificio che ha come obiettivo quello di aumentare il comfort, l'efficienza energetica e la sicurezza dei suoi occupanti. Il comfort verrà inteso come il livello di benessere dell'utente all'interno dell edificio, misurato con diversi parametri come il comfort termico o il comfort visivo. Per efficienza energetica, invece, intenderemo quando un dispositivo realizza un certo compito utilizzando meno energia di quella solitamente necessaria mentre per quanto riguarda la sicurezza, questa si concentra sull'eliminazione o prevenzione dei rischi per il personale. Considerando, nello specifico, il comfort dell'utente, l'uso delle TIC all'interno degli edifici intelligenti è in grado di migliorare profondamente il benessere degli occupanti, controllando automaticamente la temperatura, l'umidità e la luminosità dell'edificio in modo da soddisfare le richieste degli utenti. Per fare qualche esempio, un edificio intelligente deve essere in grado di accendere e spegnere automaticamente le luci quando una stanza è vuota oppure chiudere le finestre in caso di condizioni meteo avverse. All'interno della nostra tesi, inoltre, si farà riferimento ai soli edifici commerciali. Il motivo che ci ha spinto a scegliere gli edifici commerciali come quelli su cui concentrare il nostro lavoro di tesi, è il fatto che, prima di tutto, contribuiscono al 70% del consumo totale di energia

della rete elettrica [11] e perché di solito sono dotati di infrastrutture tali da semplificare la creazione di un ambiente intelligente.

Per edificio intelligente si intende un sistema di controllo distribuito, dove sono presenti sensori e attuatori e dove sono effettuate decine di computazioni per migliorare la sicurezza, il comfort e l'efficienza dell'edificio stesso [3]. Considerando il continuo sviluppo delle tecnologie che compongono sensori e degli attuatori durante gli ultimi anni e la crescita incessante delle loro applicazioni in quasi tutti gli aspetti della nostra vita quotidiana, sarebbe possibile creare edifici intelligenti se ne considerassimo solo l'aspetto tecnologico.

Per quanto riguarda l'hardware di cui è composto un edificio intelligente, possiamo individuare due reti principali: una rete di sensori e una di attuatori. Una rete di sensori è generalmente composta di sensori per monitorare la presenza, la temperatura, l'umidità e la luminosità. Questi sensori vengono utilizzati per raccogliere informazioni sull'ambiente come la temperatura esterna, la luminosità in una stanza, ecc. Una rete di attuatori è solitamente utilizzata, invece, per modificare lo stato dell'edificio. Esempi di attuatori sono le luci, l'impianto di climatizzazione, le finestre e gli elettrodomestici intelligenti. Sul mercato sono già presenti esempi di sensori ed attuatori: possiamo trovare termostati intelligenti come *Nest Learning Thermostat* [4], *Ecobee Wifi Thermostat* [5] e *Honeywell Lyric Thermostat* [6] con cui l'utente può monitorare la temperatura all'interno dell'edificio e agire sul sistema di climatizzazione. Un altro dispositivo che possiamo trovare sono le luci intelligenti, come le *Philips Hue Lampade*, con cui è possibile controllare la luce emessa dalla lampadina in modo da poterla regolare basandosi sul numero di persone nella stanza e la condizione di luce esterna.

In un edificio intelligente sono presenti, quindi, tanti sensori ed attuatori, ma ognuno di essi utilizza, tuttavia, un protocollo di comunicazione diverso. Per questo motivo, per rendere veramente intelligente il comportamento di un edificio sorge la necessità di avere un sistema che gestisca tutti i vari protocolli, un sistema di gestione dell'edificio chiamato Building Management System (BMS) [27, 28, 29]. In particolare, un BMS è responsabile della lettura dei dati provenienti dai sensori e del controllo degli attuatori in base allo stato dell'edificio.

Un punto chiave da tenere a mente quando si progettano edifici intelligenti è che sono strettamente collegati alle persone al loro interno e che non abbiamo bisogno di concentrarsi solo sull'edificio stesso, ma anche sul modo in cui le persone interagiscono con esso. Il vero problema, tuttavia, è che quando qualcuno cerca di programmare un BMS, questo risulta un compito molto difficile. Sono necessarie, infatti, competenze informatiche avanzate e, per questo, non tutti gli utenti sono in grado di capire come interagire con essi e hanno bisogno di una formazione specifica.Utilizzando un BMS, anche semplicemente accendere una luce in una determinata camera potrebbe risultare un compito difficile. Se consideriamo un utente di edificio intelligente, come ad esempio un responsabile della sicurezza, è probabile che egli voglia poter esprimere azioni più complesse che semplicemente quella di accendere o spegnere una luce. Egli può decidere, per esempio, per fissare l'edificio se tutti i dipendenti sono andati a casa e non vi è più nessuno negli uffici. Questo è un comportamento molto complesso che coinvolge molti sensori e attuatori, dobbiamo infatti controllare la presenza di persone in ogni camera dell'edificio, chiudendo ogni porta e finestra oltre ad accendere il sistema di allarme. Tutte queste azioni devono

essere specificate al BMS in termini di sensori e attuatori, e questo rende il compito di programmazione più difficile, non solo perché il numero di dispositivi da tenere in considerazione è notevole, ma anche perché [13, 14, 15] hanno dimostrato che gli occupanti preferiscono non interagire direttamente con sensori e attuatori. Infatti, capiscono meglio l'azione se specificata come "qualcuno è entrato nella stanza" piuttosto che "il sensore di movimento è stato attivato". Per facilitare l'interazione tra gli occupanti dell'edificio e l'edificio stesso, in particolare per quanto riguarda l'espressione di azioni complesse, abbiamo bisogno di un nuovo tipo di BMS che fornisca un'interfaccia utente intuitiva e che alzi il livello di astrazione rispetto a quello di sensore e attuatore [47].

Un problema rilevante, nella programmazione di un edificio intelligente, riguarda l'introduzione della multi-utenza nel BMS. Come mostrato in [16, 17], la maggior parte dei sistemi di programmazione per ambienti intelligenti è stata progettata per essere gestita da un utente singolo. In realtà, i BMS che possiamo trovare [18, 19, 20] sono stati progettati per essere gestiti da amministratori di condominio e personale addetto alla manutenzione. Il fatto di coinvolgere più utenti nel sistema fa sorgere una grande quantità di problemi da gestire. Come mostrato in [17], una sfida rilevante è quella di riuscire a gestire il fatto che ogni utente all'interno dell'edificio presenta diversi bisogni. Questa situazione può portare, dopo qualche tempo, ad avere conflitti nei dati inseriti nel Building Management System. Il problema dei conflitti, se non controllato, può portare a situazioni spiacevoli in cui il BMS non sa come comportarsi e, nella migliore delle ipotesi, potrebbe continuare a accendere e spegnere una luce. Nel peggiore dei casi, invece, potrebbe causare seri problemi di sicurezza come l'apertura di una porta che si trova tra te ed un incendio. Pertanto, i con-

flitti devono essere gestiti con cura dal BMS per riuscire a mantenere gli occupanti al sicuro ed evitare ogni possibile situazione di pericolo.

Un altro problema relativo alla multi-utenza nei BMS riguarda il fatto che per l'utente è difficile capire le motivazioni relative ai problemi causati dall'errata programmazione dell'edificio e, in generale, come si comporterà l'edificio data la grande quantità di preferenze immesse dagli utenti. Infatti, come mostrato in [45], anche i più recenti BMS assumono che sia l'utente a controllare ogni singola preferenza inserita nel sistema in modo che questa non presenti conflitti con le altre e in modo che l'edificio si comporti nel modo previsto. Questa assunzione reca problemi agli utenti che devono programmare un edificio in quanto non sempre riescono a individuare il motivo specifico alla base del comportamento anomalo che potrebbe venirsi a creare e, anche nel caso in cui riescano, questa resta una fase che necessità di un'ingente quantità di tempo.

La soluzione che proponiamo in questa tesi si fonda su un sistema esistente, che permette agli utenti di un edificio di personalizzare un edificio commerciale e che può essere integrato con BMS esistenti, BuildingRules 1.0. Questo sistema risolve parzialmente i problemi sopra citati. In primo luogo, per risolvere il problema di interazione con gli utenti che non sono necessariamente programmatori, BuildingRules 1.0 fornisce un'interfaccia semplice che permette agli utenti di personalizzare il loro ufficio utilizzando un sistema di programmazione che segue il paradigma trigger-action. In questo modo, gli utenti possono esprimere le politiche con cui vogliono controllare l'edificio utilizzando il pattern "SE succede qualcosa ALLORA fai qualcos'altro". Questo paradigma è stato scelto in quanto si tratta di un modo intuitivo ed espressivo per le persone che non hanno esperienze di programmazione per gestire l'edificio.

In secondo luogo, invece, per risolvere il problema dei conflitti tra le regole che si possono venire a creare nel caso in cui ci siano più utenti che hanno la necessità di personalizzare lo stesso spazio dell'edificio, BuildingRules 1.0 ha un sistema di risoluzione dei conflitti. Questo sistema fa uso della risoluzione, trasformando le regole in un linguaggio del primo ordine, utilizzando Z3, un risolutore sviluppato presso Microsoft Research [46], in combinazione con un sistema di priorità. Questa combinazione permette BuildingRules 1.0 per coprire tutti i possibili problemi di conflitto.

Tuttavia, BuildingRules 1.0 non ricerca una soluzione al problema riguardante il fatto che, per gli utenti, è difficile capire come si comporterà l'edificio in futuro. Nel caso in cui, infatti, l'utente non riesca a capire chiaramente il comportamento dell'edificio, non sarà in grado di inserire correttamente le proprie preferenze nel sistema. Per risolvere questo problema abbiamo introdotto un framework teorico che genera suggerimenti agli utenti per aiutarli a capire gli effetti che le regole che hanno inserito avranno sull'edificio. Per fornire questi suggerimenti, abbiamo effettuato analisi su dei grafi che abbiamo introdotto per per formalizzare lo stato di un edificio. I grafi sono, a loro volta, generati da a partire dall'insieme di regole presenti in ogni stanza dell'edificio mediante uno strumento di simulazione che abbiamo sviluppato.

Questa soluzione è stata convalidata in una campagna sperimentale che ha avuto luogo in un edificio intelligente reale al Joint Open Lab (JOL), i laboratori di ricerca e innovazione di Telecom Italia. Dai risultati raccolti ne abbiamo dedotto che il sistema di raccomandazione ha effettivamente aiutato gli occupanti a comprendere il comportamento dell'edificio anche quando erano presenti una grande quantità di regole nel sistema.

Chapter 1

Context Definition

This Chapter aims at giving an overview about the smart building topic, providing the essential knowledge about the state of the art and the general theoretical framework on which this thesis is based on. Starting with a definition of what a smart building is and which are its main features, the discussion will be around the issues that have still to be solved and the ones that have been faced during the development of this work.

1.1 Introduction

Buildings are, today, complex concatenations of structures, systems and technologies. Over time, each of the components inside a building has been developed and improved, allowing now people to control lighting, security, heating, ventilation and air conditioning systems with a digital interaction with the real objects thanks to the use of the Information and Communication Technology (ICT). Buildings, nowadays, deliver useful services that tries to make occupants more productive at a lower cost and environmental impact. These kind of buildings are called *smart buildings*. In this section we will present an intro-

duction about the meaning of smart building within this thesis work and how a smart environment can improve the user comfort and safety and increase the energy efficiency.

1.1.1 Smart Buildings: features and goals

Smart buildings are buildings that are enabled for the cooperation of objects (e.g., sensors, devices, appliances) and systems that have the capability to self-organize themselves given some policies [1]. The meaning of smart building is not equal for all the different stakeholders interested in this topic [2]. In fact, the potential final users call *smart* a building that can be remotely accessed to turn devices on and off, even though there is in fact no actual automation involved. On the other hand, for researchers and IT experts, a building is *smart* when it is responsive to its inhabitants and it is able to adapt autonomously in sophisticated ways, e.g., using intelligent machine learning algorithms to predict user occupancy and control the heating system. In industry there is a different meaning of *smart*, this word is generally used simply as a marketing term to describe programmable technologies. For this reason, it is essential to have the notion of what a smart building is within this thesis. A *smart building* will be intended as a building that increases the comfort, energy efficiency and safety of occupants. The comfort is intended as the level of wellness of the user, that can be measured with different metrics like thermal comfort or visual comfort. The energy efficiency, instead, is accomplishing a certain task using less energy whereas safety focuses on eliminating or preventing hazards to personnel.

From the **user comfort** point of view, the use of ICT in smart buildings can deeply improve the occupants' wellness by automatically controlling the en-

vironment temperature, humidity, brightness (and its general behavior), and fitting all the parameters on the single user needs. Just to make some examples, a smart building must be able to turn on and off automatically the lights when a room is used or empty; it has to dynamically open or close the windows with respect to the weather condition and the current users wishes.

Actually, the field that drove the research in the last two decades, is the use of ICT to increase the **energy efficiency** of the buildings: from this point of view, in fact, a smart building has to carefully manage how it uses energy to accomplish all the tasks. For instance, knowing the user needs, it is possible to tune the *Heating, Ventilation and Air Conditioning* (HVAC) system in order to heat only for the needed time, without wasting energy resources. A smart building is then thought to be part of a smart grid, the next generation electric grid: within this context, the building has to be able to sense and forecast its energy needs and to reply to energy saving commands coming from the smart grid.

When we speak about buildings we can mainly distinguish them in two categories: residential and commercial buildings. A building should be regarded as a residential building when more than half of the floor area is used for dwelling purposes, as they can be, for example, apartment blocks or houses. Whereas, commercial buildings are buildings where more than half of their floor space is used for commercial activities. One of the most common example of commercial buildings are office buildings, which contain spaces mainly designed to be used for offices. An assumption that needs to be taken into account during the reading of this thesis is that we will refer to **commercial buildings**. The reason that prompted us to choose commercial buildings as the ones on

which to focus our thesis work, is the fact that, first, they contribute to 70% of the total energy consumption of an electrical grid [11] (up to 45% of this energy consumption is due to heating, cooling, and lighting [12]); secondly, because they are usually provided with highly instrumented distributed systems and infrastructures that simplify the creation of a smart environment.

1.1.2 The current Smart Building idea

Today a smart building is intended as a distributed control system, where dozens of distributed computation, sensing and actuation modules [3] are exploited to increase the safety, the comfort and the efficiency of the construction itself. Considering the continuous development of the sensor and actuator technologies (namely, the *Internet of things*) in the last few years, and the incessant growth of their applications in almost all the aspects of our everyday life, smart buildings are something feasible from a technological point of view.

If we have a look at the hardware that composes a smart building, we can identify two main networks: one regarding the sensors and the other regarding the actuators. A sensors network is generally composed of occupancy, temperature, humidity and luminosity monitors. These sensors are used to collect information about the environment such as room and external temperature, luminosity in a specified room etc. An actuators network is usually used to modify the state of the building. Examples of actuators are lights, HVAC, windows and smart appliances. There are some examples of this type of sensors and actuators in the market: smart thermostats like the *Nest Learning Thermostat* [4], the *Ecobee Wifi Thermostat* [5] and *Honeywell Lyric Thermostat* [6] with which the user can control the temperature in the building by acting on the

heating system and also the air conditioning system or smart lights, like *Philips Hue Lamps* [7], with which you can control the light emitted from the bulb so that the system can adjust it in a smart way, basing on the occupancy and the daylight condition.

All these smart devices have different communication protocol and every building is composed by a lot of sensors and actuators that need to cooperate to act on the building in a smart way, for this reason an orchestration system is needed: the Building Management System (BMS) [27, 28, 29]. In particular, a BMS is responsible for the gathering of the data from the sensors and the actuation to the actuators depending on the status of the building. The BMS can be centralized or distributed. In case of centralized architecture (Figure 1.1.a), there is a single BMS module that control all the sensors and actuators, instead, in the distributed solution (Figure 1.1.b) there are different gateways that are connected to a central BMS. The task of these gateways is to decentralize the decision process and, in this way, they reduce the computation to be performed by the BMS.

1.2 Problem definition

In this section we will provide details about the problems that concern building environment and that will be useful for the readers to understand the methodology presented within this thesis work. First, the problems related to the users' interaction with sensors and actuators in a smart building environment and then, some details about multi-user issues and rule conflicts will be provided. Lastly, an overview on the problems related to the difficulties in-

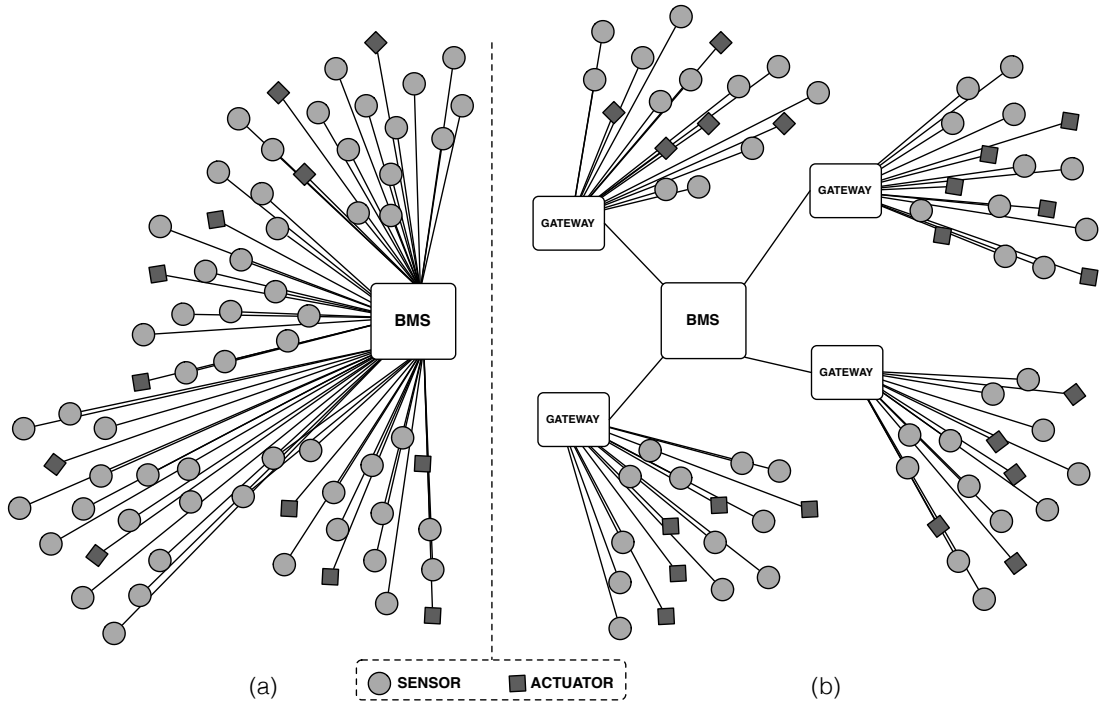

Figure 1.1: Centralized and distributed BMS architectures

troduced by the always increasing rules number will be introduced.

1.2.1 Users interaction with Smart Buildings

As said in the previous section, many BMSes have already been developed [27, 28, 29]. A key point to keep in mind when designing them is that smart buildings are interconnected with its occupants and that we don't need to focus only on the building itself but also on how the people interact with it. The real problem, however, when someone tries to program BMSes like the ones described before, is that they are very difficult to be programmed. Advanced computer skills are needed and, because of this, not every user is able to understand how to interact with them and they need, at least, a specific training. Using a BMS, even simply turning on a light in a certain room may be a hard

task. If we consider a smart building user, a safety officer for example, we can think that he may want to express more complex actions to be executed than simply turning on and off a lamp. He may want, for example, to secure the building if all the employees have gone home and there is no longer anyone in the offices. This is a very complex behavior that involves many sensors and actuators, we need to check the presence of people in each room of the building, closing every door and window and turning on the alarm system. All these actions must be specified to the BMS in terms of sensors and actuators, and this makes the programming task harder, not only because the number of devices to take into account is considerable, but also because [13, 14, 15] show that occupants prefer not to interact with sensors and actuators directly. For example, they relate better to "someone walked into a room" than "motion sensor was activated". To make easier the interaction between the occupants of the building and the building itself, especially as it regards the expression of complex tasks, we need a new kind of BMS that provides an intuitive user interface and raises the level of abstraction [47].

Another problem that rises when we try to make people program the building behavior, is that, as said in [16], people perceive tasks as activities, not as procedures. Activities are, in fact, the result of people's thinking, whereas procedures are how the buildings are actually programmed. What they found out in their experimental campaign is that occupants, who were asked to think about how they wanted to automate their home in an interview, identify their personal tasks as "do the laundry" or "make dinner". These tasks refer to several devices (washing machine, dryer, iron), and many steps are required to describe the various sub-tasks that compose it. These are activities for which

it is not trivial to understand which sensors and actuators need to be used in order to obtain the desired behavior, also because the users are not specifying any detail about the tasks, like the kind of clothes that need to be washed during the laundry or if they also need to be ironed. This is something that the users take for granted, they suppose that the system understand what they are thinking, they do not perceive all the sub-tasks that compose a complex activity as important. This is why managing the building by means of sensors and actuators risks to make the user interaction with the building a hard task for common people. There are many activities, like the ones previously described, for which a concept of interaction with sensors and actuators, with the current BMSes and technologies, is at best, inflexible, and at worst, incapable.

1.2.2 Multi-user and input conflicts

A problem that, over the years, has always been discussed and deeply analyzed is the introduction of multi-user in the BMS. As shown in [16, 17] most programming systems for smart environments have been designed for a single user to control it. In fact, BMS deployed today [18, 19, 20] are designed for building managers and maintenance personnel. Involving more users, introduces a large amount of issues that needs to be handled by the system, especially in a collaborative environment. As shown in [17], an important challenge is dealing with the diversity of building occupants, their different roles and needs. This behavior may lead, after some time, to conflicts within the inputs given to the Building Management System.

Conflicts are one of the main problems in an environment where more than one user is acting. If not controlled, they can lead to unpleasant situations in

which the BMS do not know how to behave and, at best, may just keep turning on and off a light, whereas at worst, may cause serious security problems like opening the only door that stands between you and the fire during a blaze. Therefore, conflicts must be handled with care by the BMS to keep the occupants safe and avoid every possible dangerous situation that can be produced by inputs set by unaware users. Despite this, as stated in [45], also in modern BMSes conflicts are not solved automatically, but they assume that the users will manually check each single input in order to make sure not to have conflicts between them.

Another problem concerning multi-user BMS, considering the fact that commercial buildings have many elements to be handled, is that there will be a lot of inputs that will cooperate in the system. Dealing with a large amount of inputs is difficult for the building occupants and trying to understand how the building will behave is a hard task [45].

1.2.3 Proposed Solution

Our proposed solution is based on BuildingRules 1.0 a system which allows the occupants to personalize their living environment in a commercial building and can then be integrated with existing Building Management Systems. BuildingRules 1.0 solve partially the aforementioned problems. First, to solve the problem that building occupants are not necessarily programmers, BuildingRules 1.0 provides an intuitive user interface which enable them to customize their office using trigger-action programming. This technique allows occupants to express their policies using the “IF something happens THEN do something” (IFTTT) pattern because, as shown in prior works [34, 35, 15], it is an expressive

and intuitive interface to implement building automation policies for people without programming experience. Second, to solve the problem that multiple users often customize the same building space, BuildingRules 1.0 has a conflict resolution system which makes use of a combination of first order logic resolution using Z3, a high performance theorem prover developed at Microsoft Research [46], and a priority system. This combination allowed BuildingRules 1.0 to cover all possible conflict problems.

However, BuildingRules 1.0 does not explore the problem regarding the users' difficulty in understanding how the building will behave. In this case, if the building occupant cannot clearly understand the building behavior, he will not be able to insert his preferences correctly. To solve this problem we introduced a theoretical framework that generates suggestions to the building occupants to help them understanding the effects of the inserted rules on the smart building. To provide these suggestions, we made analysis on the graphs we introduced to formalize the building state. The graphs are generated from each room ruleset making use of a simulation tool we developed.

This solution was validated in an experimental campaign that took place on a real smart building at the Joint Open Lab (JOL), the research and innovation laboratories of Telecom Italia. From the gathered results we found out that the recommendation system helped the occupants understanding the behavior of the building also when there was a large amount of rule to deal with.

1.3 Contribution and outline

Starting from the aforementioned considerations, this project aims at improving the trigger-action building programming problem, introducing a theoretical framework that generates suggestions to the building occupants to help them understanding the effects of the inserted rules on the smart building. To provide these suggestions, we made analysis on the graphs we introduced to formalize the building state.

The thesis is structured as follows: in Chapter 2 we will present the state of the art and the current solutions to the problems described in the previous section. In Chapter 3 we will explain the theoretical framework introduced during this thesis, making a detailed analysis on how we provide suggestions to the building occupants. Chapter 4, will focus on the implementation of the proposed theoretical framework in BuildingRules 2.0 and Chapter 5 will present the experimental results obtained from the experimental campaigns. In Chapter 6 the conclusion of our thesis work will be presented.

Chapter 2

State of the art analysis

This Chapter presents a survey about the state of the art regarding smart building management. We will first focus on the existing Building Management System (BMS) and the features they offer to the smart building and, secondly, we will describe the trigger-action programming technique. Finally, conflict detection and resolution in the smart building environment will be introduced.

2.1 Building Management Systems

A Building Management System (BMS) is a control system that can be used to monitor and manage the mechanical, electrical and electromechanical services in a facility. Such services can include power, heating, ventilation, air-conditioning, physical access control, elevators, lights and so on [51].

With respect to the physical architecture, the BMSes, as said in the previous Chapter, can be centralized or distributed, as shown in Figure 1.1. In case of centralized architecture, there is a single BMS module that controls all the sensors and actuators. This is the most simple solution to be implemented but,

since every request passes through that module, it will be difficult for this solution to scale in case the building grows and, therefore, it needs to have a large amount of computational power to satisfy all the requests and manage the whole building [53]. The distributed solution, on the other hand, is composed by different gateways that are connected to a central BMS. The task of these gateways is to decentralize the decision process and, thus, to reduce the computation to be performed by the BMS. This kind of architecture has a much more complex design but it also provide a higher level of scalability and can manage large buildings without the need of a central powerful machine [53].

First, we will focus on the BMSes designed to be installed in buildings using standard implementations and then we will provide details about the modern web-based BMSes.

2.1.1 Standard Building Management Systems

First examples of BMSes, like MetaSys, NiagaraAX and Desigo [18, 19, 20], still deployed today, were designed using custom solutions for a specific building focusing on building managers and maintenance personnel. Occupants interact with buildings in a limited manner, using thermostats for HVAC control, switches for lights, key cards for locks and outlets for plug loads. With these BMSes, it is not possible for the occupants to automate and personalize their environment such as setting the temperature according to outside weather or automatically brewing coffee at 8am, etc. All this kind of personalization can be made, if the BMS enables them to do so, from facility operators only.

Johnson Control [18], for example, developed a commercial product called *MetaSys* that aims at helping facility operators to solve building automation

problems of having many devices to be handled, increasing productivity and efficiency. MetaSys' interface provides facility operators with key information on building performance; its main goal is just to develop a BMS with an intuitive design, helping facility operators identifying and correcting problems related to the status of the building more quickly and making it easier for them to troubleshoot equipment. It's not designed to be flexible, it is scalable only at cost of a powerful processing unit and it needs a custom implementation for each building in which it has to be installed.

NiagaraAX [20] is another example of standard centralized BMSes and it aims at overcoming the problems of multiple, incompatible protocols prevalent in the industry, allowing devices from different manufacturers to be integrated together seamlessly within one automation system. This approach to building automation provides several benefits to building owners and operators: customized systems with components and subsystems from different vendors can be easily implemented; the configuration cost associated with building management systems can be reduced; systems can be monitored and controlled remotely; and information from multiple facilities can be centrally accessed for comparisons and identification of best practices. With respect to *MetaSys*, that provides a centralized custom solution for every building in which it is installed, *NiagaraAX* makes some steps forward in the direction of a less constrained deployment, making it compatible with the major existing protocols but still making use of a strictly centralized architecture.

2.1.2 Web Based Building Management Systems

Moving in the direction of providing occupants with the flexibility to express and implement building customizations, modern web service based BMSes, together with an advanced sensor technology, were introduce. This kind of BMSes can improve occupant comfort and productivity [30] as well as building energy efficiency [31, 32, 33].

An example is *Buildingdepot* [27], which is an extensible and distributed architecture for storage, access, and management of building sensor data. BuildingDepot provide easy access to the data generated within buildings using user management tools that let institutions in which the system is installed share their data to others and allow client applications to be built using a well defined API. It consists of four main components as shown in Figure 2.1: a core Data Server combined with a web service that exposes access through a RESTful API; a Directory Service that maps the Data Server in an institution; a User Service that provides access to outside users to the system and Data Connectors that interface with the underlying network to various data servers. The main features of BuildingDepot are:

- scalability and incremental deployment to allow institutions of different sizes to grow their installation on their own, without need of an external support;
- flexibility, so that different types of institutions with different data organization schemes could be supported by design;
- easily searchable data, so that users could find sensors and actuators easily;

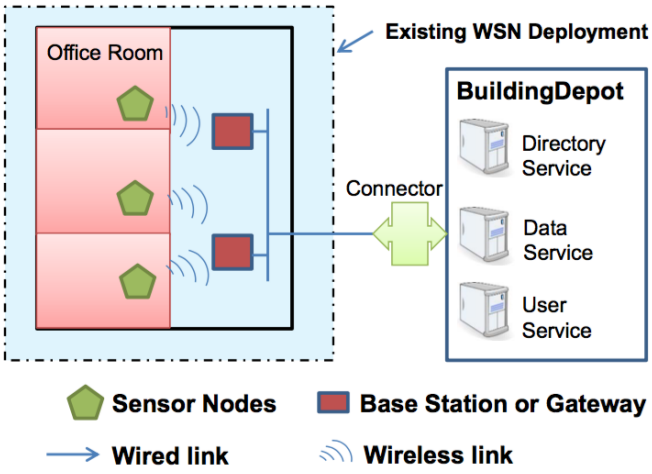

Figure 2.1: BuildingDepot architecture [1]

- extensibility with a standardized API, to provide an easy way to develop applications on top of it;

- a rich set of user management features, so that data sharing is efficient at multiple granularities for the institutions in which the system is installed.

Another example is *BOSS* [28], which aims at supporting the distributed physical resources present in large commercial buildings being fault-tolerant and scalable. The BOSS architecture is made of six main subsystems:

- hardware and access abstraction, not to make users interface directly with the hardware;

- naming and semantic modeling, that is used to describe the relationships between all the devices that compose the underlying sensor and actuator networks;

[1] Y. Agarwal, R. Gupta, D. Komaki, and T. Weng. Buildingdepot: an extensible and distributed architecture for building data storage, access and sharing.

- real-time time series processing and archiving, to be able to access the real-time and historical data coming from the network;
- control transaction system to solve the conflicts and provide security measures;
- authorization subsystem that provide an access control list;
- actual running applications.

Energy@Home Jemma [52], an Italian project developed by Telecom Italia, is the *Java Energy ManageMent Application framework*, a modular, resource-oriented middleware running in Home Gateways and in the Cloud able to support energy monitoring, management and awareness applications at home. The main features of Jemma are:

- the ability to dynamically discover devices attached to the network and instantiate the proper driver;
- power consumption monitoring to suggest better behavior to the users;
- security measures to securely transport and manage sensitive data.

In summary, all the previously shown examples are very different from the ones described in the previous section as they all present a more scalable, distributed architecture and enable the occupants of the building to customize the environment.

2.1.3 Building Management Systems integration

A slightly different example is *OpenHAB* [54], a software for integrating different building automation systems and technologies into one single solution

with uniform user interfaces. The reason for which it is different from the other BMSes we previously presented is its purpose. OpenHAB, in fact, does not try to replace the existing solutions, but its purpose is to enhance them. Therefore, it assumes that the subsystems on which it relies on have already been configured. In this way it avoids performing this task, which is often very specific and complex. Instead, what OpenHAB tries to do is to focus on the aggregation of the data coming from sensors and actuators set up in the various submodules, giving a better vision of the overall status. OpenHAB also provides the ability not to reference objects with specific information like IP addresses or IDs but to handle them as an "item", that provides an higher level of abstraction hiding all the specific device information. An important aspect of OpenHAB's architecture is its modular design. It is very easy to add new features and you can add and remove such features at runtime.

In Table 2.1 are summarized the key points of each BMS we previously described. In particular, it shows the manufacturer of the BMSes (universities or companies), its name and architecture and it provides some comments to understand better their main features.

2.2 Trigger-action Programming Technique

Technologies that enable building automation and smart homes have been around for decades but, since they have been expensive and complex, in the last few years the scientific community started to investigate a new paradigm, the *trigger-action programming*.

if **Trigger** then **Action**

Manufacturer	Name	Architecture	Installation Type	Comments
Johnson Control	MetaSys [18]	Centralized	Standard	Needs a custom implementation for each building in which it is installed
Niagara	NiagaraAX [20]	Centralized	Standard	Less constrained deployment, compatible with the major existing network protocols
UCSD	BuildingDepot [27]	Distributed	Web-based	Exposes the web services through a well defined REST-full API
Berkeley	BOSS [28]	Distributed	Web-based	Scalable, fault-tolerant and multi-layered architecture
OpenHAB	OpenHAB [54]	Integrator	Web-based	His design is modular to make it more scalable. Integrate different BMSes into one solution
Energy@Home	Jemma [52]	Distributed	Web-based	Modular architecture, supports energy monitoring and management applications

Table 2.1: BMSes Summary

Figure 2.2: IFTTT

`if` **something happens** `then` **do something**

Trigger-action programming allow the end users to specify the behavior of a system as an event, that corresponds to the trigger, and the related action, to be taken whenever the specified event occurs. This programming technique has recently emerged as a promising solution to involve users in home automation, as it provides an expressive and an intuitive interface [34, 35, 15]. In fact, it has also been chosen to carry on the most recent experimental campaigns in the smart building context like the one presented in [45].

Trigger-action programming is widely used mainly because it is human understandable and doesn't require particular programming skills to be used in practice [15]. In fact, there are also commercial examples that have been developed to connect various web services and different smart home appliances.

2.2.1 Trigger-action programming: the *If This Than That* paradigm

Trigger-action paradigm has become so popular that it has been used also in commercial product like IFTTT [36], that stands for "IF This Than That". It is a web application that allows users to create chains of simple conditional statements, called "recipes". The "recipe" makes easier the interaction between the users and the applications or product that they need to handle. Trigger-action

programming and in particular IFTTT, as shown by Dey et al., express 95% of all the behaviors that the users wished to express in smart homes, demonstrating its expressiveness across a wide set of context aware applications [34]. More recently, Ur et al. showed that 63% of smart home applications requested by occupants required programming, and all of these applications could be expressed by the IFTTT paradigm [15].

However, [48] shows that IFTTT has some deficiencies caused by its oversimplification that limit the expressivity of the programs that can be created. During the studies they carried out they identified two distinct types of triggers (event and state triggers) and three types of actions (instantaneous, extended, and sustained actions). An *event* trigger indicates the occurrence of some changes at a specific point in time, as it could be, for example, "the doorbell rings". A *state* trigger, instead, indicates that some condition is currently true and it is lasting over a period of time. Example of state trigger could be "time is between 11:00am - 03:00pm". For what concerns the actions, as said before, they can be defined as instantaneous, extended or sustained. An action is *instantaneous* when it happens at a specific moment in time and does not change the state of the system, like "sending an email". It is called *extended* if it will be completed within a certain amount of time and, when its effect is over, the state of the system returns as it previously was. Example of extended action could be "brewing coffee". An action is defined *sustained* when it involves a change in the state of the system. An example could be "turn on lights" as the effect of this action will impact on the building even when the rule stops acting on the building, permanently changing his state. The results of their study reveal inconsistencies in the user interpretation of the behavior of programs in-

volving these different types of triggers and actions. For instance, the rule "if there is someone in the bedroom, turn on the lights" for some people implies that the action will also take care of switching off the lights when nobody is in the bedroom, while for others it means that the lights will remain turned on unless directed to be turned off by another rule.

With BuildingRules, we will focus on solving the main issues related to an actual and practical usage of the trigger-action programming a within smart environment to provide personalized automation in complex commercial buildings, and address the challenges that emerge when deploying such a system in a real environment.

2.3 Conflict resolution

Deploying a system that uses trigger-action programming on a large scale, involves the rising of conflicting situations due to the fact that, when different users interact with such a system, their intentions are likely not to be the same [40]. Moreover, as shown in [45], when conflicts take place, the building occupants do not know how to handle this situation. In particular, they notice that the behavior the building is having is not the one that they wanted to express, but they do not understand the nature of the conflicting situation and it takes time to solve these issues. What is needed, is a way to solve these conflicts because they can lead to unpleasant situations in which, at worst, people can be in danger. Conflict resolution can be done in two ways, with or without making use of context information. If we consider context aware systems, conflict resolution has been studied extensively [40] and a number of conflict

resolution strategies to automatically resolve application inconsistencies have been developed. In [41], for example, Chang Xu tries to analyze conflicts in this kind of systems defining *context consistency* as the situation in which there is no contradiction in a computation task's context, and *context inconsistency* as the situation in which a contradiction occurred. To understand the meaning of context contradiction, they considered a scenario taken from the healthcare industry. A doctor is in the operating room and a surgery is being performed. From this information the system will probably conclude that the doctor is doing a surgery. Suppose that, in addition to these, there is another piece of information that makes the system believe that the doctor is looking for medical resources. Thus, the system does not know what to do anymore and it might draw the wrong conclusion. Because of the fact that multiple choices can be taken, there is a contradiction in the context. In their provided solution they detect and resolve this type of inconsistency.

However, enhancing human-computer interaction through the use of context is a difficult task and the context-aware models are hardly reusable [42]. Because of this, context-unaware conflicts resolution techniques have been developed. There are two main types of resolution that do not take into account context, one that do not involve the end user during the resolution, that we call *Sensor centric resolution*, and one that require the user intervention during the resolution, called *User centric resolution*.

2.3.1 Sensor centric resolution

As already mentioned, the *sensor centric resolution* does not involve the users because it is performed at sensor level using various conflict resolution strate-

gies. There are some BMSes that are already using these techniques, for example making use of BACNet (Building Automation and Control Networks) protocol, that is a widely adopted standard in industrial BMSes [37]. In this protocol, to solve conflicts, each writable sensor has a priority table and all the applications are sorted according to their importance. Web service BMSes extend this methodology adding access control and providing more metrics for conflict resolution as it will be shown in the following examples. SensorAct [38] uses a script that specifies validation conditions based on date, time, duration, location and frequency of operations. The combination of this script and a priority system allows the management of conflicts.

Another example is the BOSS [28] BMS, in which they define a database-like transactional system mapping every sensor write as a single transaction. Finally, BuildingDepot [39] proposes a solution similar to BACNet incorporating a priority array and adding a conflict-default value, used as a fallback in case there are two or more users with the same priority level. With *sensor centric resolution*, the user can not change the result of a conflicts during the conflict resolution and the system, in case of same conflicts, return always the same result. This could be a problem because, depending on situational, the user could want different result and for this reason there is the *User centric resolution* type.

2.3.2 User centric resolution

In case of conflicting situations, in particular in building environment, the user may prefer to be in control of the choice of the solution. Because of this, *User centric resolution* involves humans during the conflict resolution. Usually this type of detection finds conflicts and allows the user to solve them. For this

reason, the system needs to abstract information so that users can understand the nature of these conflicts, and, thus, try to solve them [34].

CARISMA [43], for example, is a mobile computing middleware that resolves conflicts among multiple users for a single application with pre-recorded preferences. Park et al. [44] use JESS (Java Expert System Shell) [49], a rule engine for the Java platform, which supports the development of rule-based systems that can be closely related to code written in Java. In their system, an actuator state variable can change only in two directions, namely increase or decrease. When rules trigger a change in opposite direction on the same variable simultaneously, a conflict is detected. Therefore, using this approach, a conflict is detected only when an actuation, made by a rule just activated, is in contrast with the rules already active in the system. Instead, in other solutions, when a new rule is inserted, all the rules are converted to first order logic and checked using a SMT (Satisfiability Modulo Theories) solver [46], which is the strategy followed by Zhang et al. [50]. With respect to JESS, this approach allows solving conflicts before they actually rise and it does not need to be enriched with inference rules to define what is considered as conflict. Furthermore, in JESS, every time a rule type is added, new customized inference rules need to be added as well.

2.4 Conclusion

As shown in this Chapter, many BMSes have been developed and each of them have different features. Our approach is not to develop another BMS but, to realize a system that creates an additional layer between building occupants

and the BMS: BuildingRules. The goal of our thesis work is to solve the main issues related to the actual usage of a trigger-action programming techniques in commercial buildings. For this reason BuildingRules provides an intuitive user interface which enable the user to program the BMS using trigger-action paradigm, which, as we shown in Section 2.2, is human understandable and does not require particular programming skills. BuildingRules also provides a recommendation and suggestion system to help occupants understand how the smart building will react to the inserted rules because, as shown in Section 2.3, Woo [45] stated that for building occupants is difficult to understand the building behavior in a clear way. Finally, with respect to the different conflict resolution techniques presented in Section 2.3, we decided to adopt a user centric resolution to ensure that the user is in control of choosing the solution of the conflicting situation. To archive this goal, we developed a solution which makes use of a combination of a SMT solver and a priority system.

Chapter 3

The Proposed Methodology

The aim of this Chapter is to provide an analysis of the proposed methodology. First, we will introduce BuildingRules 1.0, the previous version of BuildingRules. We will then analyze the problems that emerged in this version during the experimental campaign and we will then propose a new theoretical framework that provides suggestions to the building occupants making use of graphs.

3.1 Introduction

Our thesis work is founded on a previous version of BuildingRules, developed starting from the idea of programming a building in an easy way, through the use of the trigger-action paradigm [34, 35, 15]. This version of BuildingRules, that will be analyzed in details in the following Section, handles the main elements that compose the smart building environment. In particular:

- it manages the building occupants providing different level of building management with respect to their role;

- it provides a representation of the rooms and the groups of room that compose the building;

- it enables the occupants to insert rules in each room or group of rooms;

- it handles conflicts among rules originated from multi users interaction with the smart building providing conflict resolution techniques using a SMT Solver and assigning priority to each rule.

Even if these aspects seem to cover the fundamentals to manage a smart building, the experimental results pointed out that the behavior of the building was not clearly understandable by the building occupants, especially when the number of rules increased [45].

During our thesis, aiming at solving the aforementioned problem, we introduced a theoretical framework to make suggestions to the building occupants. To support this framework we generated graphs from the ruleset. The graphs represent the behavior of the building. The nodes represent the building state and the edges represent the transactions between them. Upon these graphs we have made some analysis, looking for undesired or unexpected behavior, to provide recommendations to the building occupants. The behavior analysis was made possible by the introduction of a building simulator, which provides the rules that will be activated during a predefined time period.

To summarize, the contributions of our thesis work are:

- **The Suggestions**: information collected from the analysis made on the graphs and provided to the building occupants.

- **The Graph Generator**: generates the graphs that represent the building behavior starting from the building simulation;
- **The Graph Analyzer**: performs analysis on the graphs generated by the Graph Generator;
- **The Simulator**: provides the rules that will be activated in a predefined time period;

In this Chapter we will present the theoretical details about BuildingRules. First, we will provide details about the previous version of the system (1.0), needed to understand the work done within this thesis, and, second, we will explain the theoretical aspects behind BuildingRules 2.0, obtained by extending BuildingRules 1.0 introducing the aforementioned contributions.

3.2 The starting point: BuildingRules 1.0

The main goal of BuildingRules 1.0 was to provide the building occupants with a simple, scalable and intuitive system to allow them expressing their preferences regarding the behavior of the building itself, thus making it possible for them to customize their working environment according to their needs. Therefore, BuildingRules was designed as an additional layer between the end users and the Building Management System (BMS) [27, 28, 29], as can be seen in Figure 3.1. This design enables the users to customize the building environment without knowing the actual physical configuration of the building itself and, therefore, simplifying the building programming phase. BuildingRules 1.0 takes charge of handling the users requests delivering them, properly pro-

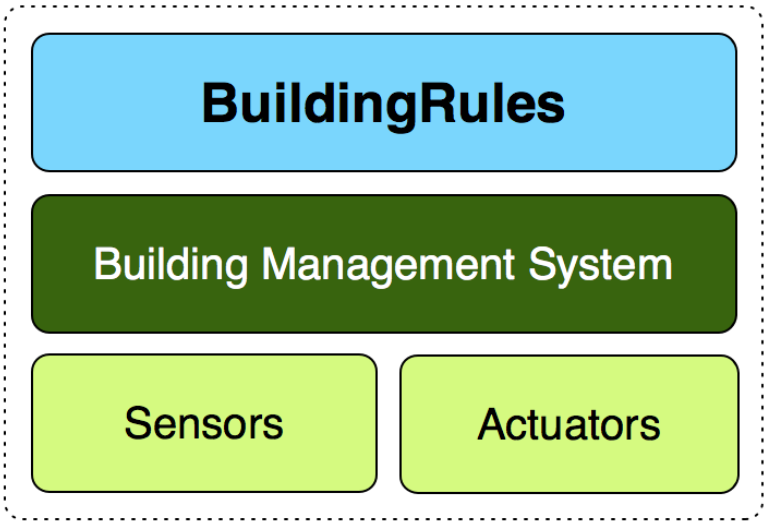

Figure 3.1: BuildingRules Architecture

cessed to match the needed format, to the underlying BMS that will manage the gathering of data from the sensors and the actual actuation on the building. For example, if the rule *"if someone is in the room then turn on the lights"* has been inserted in BuildingRules, it will periodically ask the BMS whether someone is in the room or not. If this condition is verified, BuildingRules will send a command, to the BMS that, eventually, will turn the lights on.

Since BuildingRules targets commercial buildings, it represents them as composed of rooms and groups of rooms. In each room there is one or more occupants. A room represents a physical space of the building like an office, a conference room, a lobby or a kitchen. Occupants are assigned to these rooms by the building manager, and they can customize the behavior of the room or group of rooms they are assigned to, by adding new rules.

To model all these elements and interactions, BuildingRules architecture is modular and is composed by the modules represented in Figure 3.2. What can be noticed is that, on one hand, BuildingRules models the actors we previously

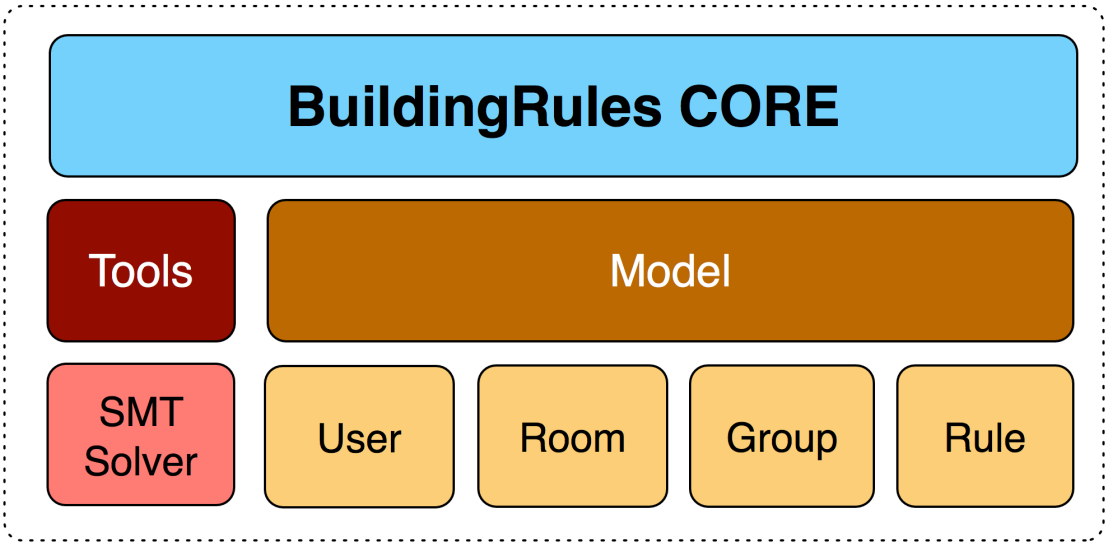

Figure 3.2: BuildingRules 1.0 features

described with the following components:

- **User**: needed to handle different types of users providing a hierarchy system to assign different privilege levels to each type of user;
- **Rule**: needed to manage the trigger-action based rules that are inserted in the system, partitioning each of them in several predefined categories;
- **Room**: needed to manage the building rooms, handling the sensors, the actuators and the rules inserted;
- **Group**: needed to handle more rooms at once, so that shared preferences between them can be managed without inserting the rules for each one individually.

On the other hand, the **SMT Solver** is the main tool that handles the conflicts between the rules, once they have been formalized as propositional logic formulae.

3.2.1 Rules

As mentioned earlier, BuildingRules 1.0 uses the trigger-action paradigm [55, 36, 15] allowing occupants to specify rules in the following format:

$$\texttt{if (something happens) then (do something)}$$

The "if" part of the rule is called *trigger* while the "then" part is called *action*. According to this paradigm, a user can specify an action to be performed when certain event conditions are met. The combination of an action with an event is defined as a *rule*. For example, in the rule

$$\texttt{if (someone is in the room) then (turn on lights)}$$

"someone is in the room" is the trigger, and "turn on lights" is the action.

BuildingRules 1.0 introduces the concept of *simple* and *complex* rules. A *simple rule* is composed of a single trigger and action, while a *complex rule* has multiple triggers or actions, each of which is connected by a logical AND. For example, "if it is raining then close the windows" is a simple rule while "if it is Sunday and it is after 10am then close the curtains" is a complex rule. In BuildingRules 1.0 complex rules are restricted to just multiple triggers; in fact, rules with multiple actions are not supported to keep the user interaction paradigm as simple as possible. Anyway, this is not a limitation for the expressivity of the system since those rules can be easily decomposed into multiple rules with the same trigger.

Two data types for triggers and actions (*boolean* and *integer*) are supported. The Boolean type is used for sensors and actuators that have only two states,

	TYPE	DATA	CATEGORY	EXAMPLE NAME	EXAMPLE HUMAN READABLE SYNTAX
1	T	BOOLEAN	OCCUPANCY	OCCUPANCY_TRUE	someone is in the room
2	T	INTEGER	EXT_TEMPERATURE	EXT_TEMPERATURE_RANGE	external temperature is between @val and @val
3	T	INTEGER	TIME	TIME_RANGE	time is between @val and @val
4	T	BOOLEAN	DATE	DATE_RANGE	the date is between @val and @val
5	T	BOOLEAN	WEATHER	SUNNY	it is sunny
6	T	INTEGER	ROOM_TEMPERATURE	ROOM_TEMPERATURE_RANGE	room temperature is between @val and @val
7	T	BOOLEAN	DEFAULT_STATUS	NO_RULE	no rule specified
8	T	INTEGER	DAY	TODAY	today is @val
9	T	BOOLEAN	EXTERNAL_APP	CALENDAR_MEETING	calendar meeting event
10	A	BOOLEAN	LIGHT	LIGHT_ON	turn on the room light
11	A	BOOLEAN	WINDOWS	WINDOWS_OPEN	open the windows
12	A	INTEGER	HVAC	SET_TEMPERATURE	set temperature between @val and @val
13	A	BOOLEAN	APPLIANCES	COFFEE_ON	turn on the coffee machine
14	A	BOOLEAN	MESSAGES	SEND_COMPLAIN	send complain to building manger
15	A	BOOLEAN	CURTAINS	CURTAINS_OPEN	open the curtains

Table 3.1: Currently supported rule triggers (T) and actions (A) categories. An example of trigger or action for each category is provided

such as the window that can be only opened or closed. Instead, the integer type is used to represent the state of sensors and actuator that cannot be represented by boolean values, such as temperature and humidity. With respect to integer values BuildingRules 1.0 forces the user to insert a range of values to keep the conflict analysis (see Section 3.2.2) simple. For instance, a user cannot insert a rule with "if it is after 8PM" as a trigger; instead he needs to specify a time interval: "if it is between 8PM and 10PM". By specifying a time range, a rule has a time validity, and it reduce the possibility to run an action forever. This restriction does not change the expressiveness of rules, but forces the user to define both the start and end points of the rule.

Table 3.1 shows the list of triggers and actions supported by BuildingRules 1.0. Each trigger and action is assigned to a *category*, representing the information it expresses. For example, the rule "if it is rainy then turn on the light" has the antecedent that belongs to the "*Weather*" category and the consequent to the "*Light*" category. *NO_RULE* is a special trigger available for the building administrators (Rule 7 in Table 3.1). This trigger is always set to *True* and

is used for setting the default conditions of the building that can be used as a fallback in case no rule is specified by the user to override them. For example, if the rule *"if no rule specified then close the windows"* has been inserted, in case the occupants forget to insert a rule that manage the windows during the night, the windows will be closed in any case.

BuildingRules also supports external applications through virtual triggers that are controlled via RESTful APIs (Rule 9 in Table 3.1).

3.2.2 Conflicts among rules

Since users can express their own rules for rooms, some of which are shared by multiple users, conflicts can arise. In BuildingRules 1.0 two rules are defined as conflicting when two or more rules act at the same time on the same actuators trying to apply different effects. To be clear, for example, considering the following rules:

if time is between 6am and 3pm then turn the light on

if nobody is in the room then turn the light off

in case nobody is in the room and time is between 6am and 3pm, BuildingRules 1.0 will try to turn the light on and off simultaneously. If these conflicts are not resolved properly, they can lead to damage of equipment, like in the previous example, or compromise user comfort and safety. BuildingRules 1.0 supports two type of conflicts: **static conflicts** and **run-time conflicts**.

3.2.3 Static conflicts

Two rules conflict statically if the triggers belong to the same category and the actions act on the same actuator. They are conflicting because the two rules may be triggered at the same time but they act in an incoherent way on the actuator. To clarify this concept we can consider two users who independently specified the two following rules:

if time is between 9am and 6pm then turn the HVAC on

if time is between 5pm and 8pm then turn the HVAC off

As can be seen, between 5pm and 6pm, the system would be in an inconsistent state since it will try to turn on and off the HVAC simultaneously during this time interval. This may cause discomfort to the occupants and could damage the HVAC damper if not actuated properly. To identify this type of conflicts among rules, a formal verification approach has been used, as will be explained in the following Section.

3.2.4 SMT Solver

In BuildingRules 1.0, in order to detect static conflicts, the rules are formalized as propositional formulae and then we the SMT Solver Z3 is used to actually solve them.

As said before, a rule is composed of two parts: a trigger or a conjunction of triggers, and an action. Before adding a rule, it is verified against the set of rules already in the room. Each rule is represented as a propositional formula composed by an implication in which the trigger implies the action. In general, the implication is satisfied if the trigger is not satisfied, or if both the trigger and

the action are satisfied. In this context, the action is considered as a proposition that is true if the action can be executed, false otherwise. The new rule, together with the existing ones, are seen as a specification and automatically verified to check their satisfiability. If the specification is satisfiable, the rules are not in conflict with each other. If not, two or more rules are in conflict and it needs to be resolved. The rules are formalized as propositional formulae compliant with the following grammar:

$$
\begin{aligned}
\texttt{rule} &::= \texttt{trigger} \Rightarrow \texttt{action} \\
\texttt{trigger} &::= \texttt{sTrig} \mid \texttt{sTrig} \wedge \texttt{trigger} \\
\texttt{action} &::= \texttt{bAct} \mid \neg\texttt{bAct} \mid \texttt{iAct} \in \texttt{[n,m]} \\
\texttt{sTrig} &::= \texttt{bTrig} \mid \neg\texttt{bTrig} \mid \texttt{iTrig} \in \texttt{[n,m]}
\end{aligned}
$$

A *rule* is an implication, where the *action* and *trigger* have a fixed structure. The *trigger* is a conjunction of conditions *sTrig*, that are built on the triggers represented in Table 3.1 (Rows 1 - 9). When the *trigger* is boolean (Rows 1, 4, 5, 7, 9), the condition is satisfied when the data is true (*bTrig*) or false ($\neg$*bTrig*). When the *trigger* is an integer (Rows 2, 3, 6, 8), the condition is satisfied when value is in the specified interval `[n, m]`, where $\texttt{n} \leqslant \texttt{m}$ and they are both specified according to the data domain. A similar method is used for both boolean (Rows 10 - 11, 13 - 15) and integer (Rows 12) *action* values. The use of disjunction in actions or triggers and the use of conjunction in actions is not allowed. The conjunction in the action is redundant and is equivalent to specifying multiple rules with the same trigger and different actions. The same reasoning can be done with the disjunction in triggers as it is equivalent to specifying multiple rules with the same action and different triggers.

Note that the disjunction in actions introduces non-deterministic rules, meaning that the system can make a choice on the actions that can be performed.

Currently, the user needs to completely specify the action for a rule using priority. For example consider a user who wants to insert a rule"if the room is dark then turn on the lights or open the blinds", with the intention that the system can choose to "turn on the light" or "open the blinds" in case of poor luminosity. The user is asked to insert two rules with different priorities to make his intentions clear without any ambiguity.

Since the rules correspond to a subset of propositional logic, they can be analyzed by encoding them in the language of the Z3 SMT Solver [46]. Since Z3 supports boolean and integer variables, translating the formalization of rules into the Z3 language is straightforward. Note that formulae must be expressed in the prefix form adopted by Z3. For example, the rule "If someone is in the room then turn on light" is represented as:

```
(assert (=> inRoom lightOn))
```

and the rule "If someone is in the room then set temperature between 68F and 72F" is represented as:

```
(assert (=> inRoom (and (<= 68 temp)
                        (<= temp 72)))).
```

The Z3 model was completed with a set of assertions that specify additional characteristics of the integer data (e.g., *time* is between 0 and 24) and the relationship among data (e.g., if it is sunny, then it cannot be rainy). The model is then verified to check for possible conflicts. It is verified multiple times by asserting the trigger of each rule in the same category. Thus, it is possible to identify the conflicts related to the same trigger or related triggers. In BuildingRules 1.0, the rule verification is performed as soon as a new rule is inserted.

	TYPE	EXAMPLE HUMAN READABLE SYNTAX	EXAMPLE Z3 SMT TRANSLATION
1	T	someone is in the room	(inRoom)
2	T	external temperature is between @val and @val	(and (>= (extTempInRoom) @val) (<= (extTempInRoom) @val))
3	T	time is between @val and @val	(and (>= (time) @val) (<= (time) @val))
4	T	the date is between @val and @val	(and (>= (day) @val) (<= (day) @val))
5	T	it is sunny	(sunny)
6	T	room temperature is between @val and @val	(and (>= (tempInRoom) @val) (<= (tempInRoom) @val))
7	T	no rule specified	(noRule)
8	T	today is @val	(= (today) @val)
9	T	calendar meeting event	(meetingEvent)
10	A	turn on the room light	(light)
11	A	open the windows	(openWindows)
12	A	set temperature between @val and @val	(and (>= (tempSetpoint) @val) (<= (tempSetpoint) @val))
13	A	turn on the coffee machine	(coffee)
14	A	send complain to building manger	(sendComplain)
15	A	open the curtains	(openCurtains)

Table 3.2: Translation to Z3 examples. Refer to table 3.1 for more details.

When a user inserts a rule that is conflicting with existing rules, a notification is raised and the user is asked to modify the rule. The rules are translated from human readable syntax to the Z3 syntax using a pre-defined look up table (see Table 3.2).

Z3 has been chosen to detect conflicts as it is efficient and reusable. Although the satisfiability problem is computationally expensive, it is possible to ensure low latency as Z3 solves this problem efficiently. An alternative would be having a custom implementation to deal with our particular variables. To ensure good performance, the algorithm we would have to be modified and evaluated every time the variable domain, the rules structure and other details in the ruleset are modified. Instead, the SMT solver requires only an addition of transformation rules to create a new model, but does not require re-evaluation of the algorithm, since it is computed efficiently by Z3. Moreover, the input language of the SMT solver is generic, and it would be easy to switch to different SMT solvers, such as Yices [56].

3.2.5 Run-time conflicts

Some conflicts cannot be statically detected, namely the ones in which the actions belong to the same category but the triggers do not. Consider the following example:

if nobody is in the room then turn off the light

if time is between 5pm and 8pm then turn on the light

Using Z3 it is not possible to identify the conflicts that arise when the room is empty between 6pm and 8pm because the SMTs Solver cannot predict if nobody will be in the room between 6pm and 8pm. In this case, the system, at runtime, will try to activate both the rules causing possible damages to the light bulb as it will be continuously turned on and off. Moreover, since the users may want to express more complex policies than the one previously described, supporting this type of rule conflicts is necessary.

To understand how the resolution of run-time conflicts has been handled, it is possible to consider this example: suppose that a user want to express a policy that generally turns on the light between 6pm and 8pm, except when nobody is in the room. In this case, it is not possibile to solve the conflict statically because the antecedent "nobody is in the room" is not static by nature. As we said before, we do not know when it will be triggered *a priori*. What is clear, in this example, is the behavior that the user want to express in this room. In fact, he generally want the room light to be turned on between 6pm and 8pm but not in case nobody is in the room. As a consequence, BuildingRules 1.0 lets the user assign a priority value to each rule he inserts. If the user desires to set a policy like "usually I want this behavior but not when this event takes place",

he sets a lower priority to the general rule, and a higher priority to the rule that is more specific. Consequently, the priority value is used to order the rules by importance and to dynamically resolve conflicts during the actuation phase in an efficient way. This approach is simple to be explained to the occupants as they only need to know: "give a higher priority to the more important rules".

3.2.6 Users

In a typical commercial building, there are different types of users who may express automation policies [39]. The employees can control the offices they are assigned to, a lab manager can control the specific lab he works in while the department chair and the building manager can have overriding control over the policies expressed by all the other building occupants. BuildingRules 1.0 needs to express this hierarchy since it is needed to handle the building hierarchical structure and it affects how it resolves conflicts. This task is accomplished by assigning *privilege levels* to each type of user.

BuildingRules 1.0 supports different user categories. For example, as can be seen in Figure 3.3, the actual building occupants may be classified as building managers or standard users; while special users may be assigned to *Applications* and *Default* categories. *Applications* may be external software that interact with BuildingRules, while *Default* can represent agents controlling the default status of the building, namely when users do not specify any rule, that is represented by the DEFAULT_STATUS category in Table 3.1 line 7.

User levels are used in two different ways. First, they ensures that low level users cannot edit or delete the rules specified by users with higher level. For example, consider a room R_1 shared by two users (U_1 and U_2) (level of $U_1 >$

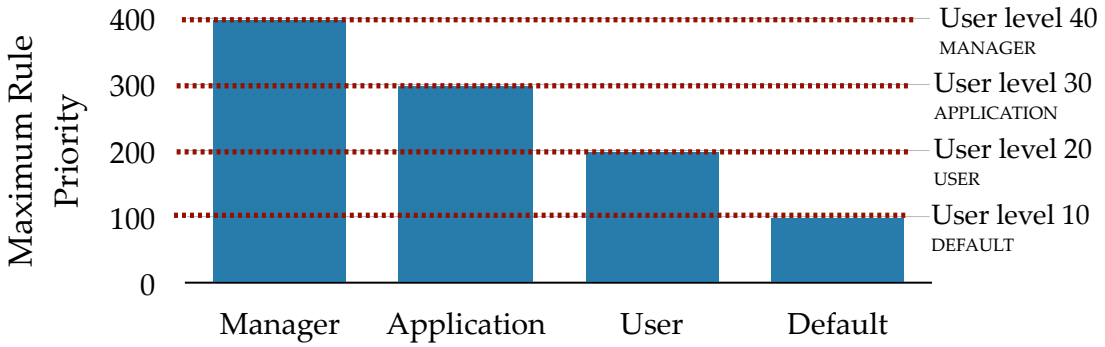

Figure 3.3: User level and rule priority relation

level of U_2), both users are allowed to enter rules into R_1; U_1 can edit or delete rules specified by U_2 but not vice-versa. Thus, a higher level user, such as the building manager, can enforce entire building policies by creating rules with a priority higher than the standard user. The levels restrict the maximum priority a user can specify for a rule. A higher level user can assign a higher priority to a rule, giving it more priority in case of runtime conflicts, namely overriding rules expressed by a lower-level user.

3.2.7 Groups

BuildingRules 1.0 enables the creation of groups of rooms to reuse rules across rooms. Suppose that the building manager wants to turn off all the building lights during the night. Without the group feature, he would have to insert the rule "If time is between 10pm and 7am then turn off the lights" in every room. Instead, creating a group that is composed of all the building rooms, he needs to create the rule once by specifying it for that specific group.

Figure 3.4.A shows an example in which the rooms are grouped on per floor basis. In this example, if R_i is a generic room belonging to a group G, the rules specified at the group level are inherited by all the R_i rooms.

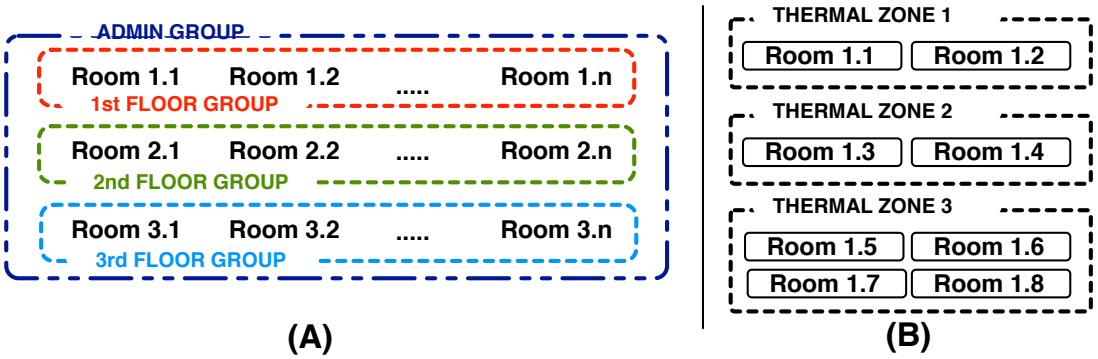

Figure 3.4: (A) Example building groups (B) Example building thermal zones distribution

BuildingRules 1.0 supports another class of groups to incorporate the physical characteristics of commercial buildings. In fact, Heating, Ventilation and Air Conditioning (HVAC) systems and lighting systems often divide the building into zones of operation, and can only be controlled at the zone level granularity [21, 32]. Figure 3.4.B shows an example of HVAC thermal zone in a building. As a result, if two rooms (R_1 and R_2) belong to the same thermal zone, the HVAC rules specified for R_1 is propagated to R_2. Such groups are called *Cross Room Validation Group* (CRVG), and they are specified for action categories that need to follow this property. The behavior of a CRVG is shown in Figure 3.5. In a CRVG, all the rules expressed (for specified actions) in one room within the group are propagated to the other rooms.

The conflict checking algorithm needs to take care of which groups a room belongs to. If a room does not belong to any group, the ruleset is composed of the rules inserted in that room. If the room belongs to one or more standard groups, the ruleset to be checked is composed of the rules saved in the considered room and the union of all the rules saved in these groups. If the room belongs to a *Cross Room Validation Group*, the ruleset of the room is the union of

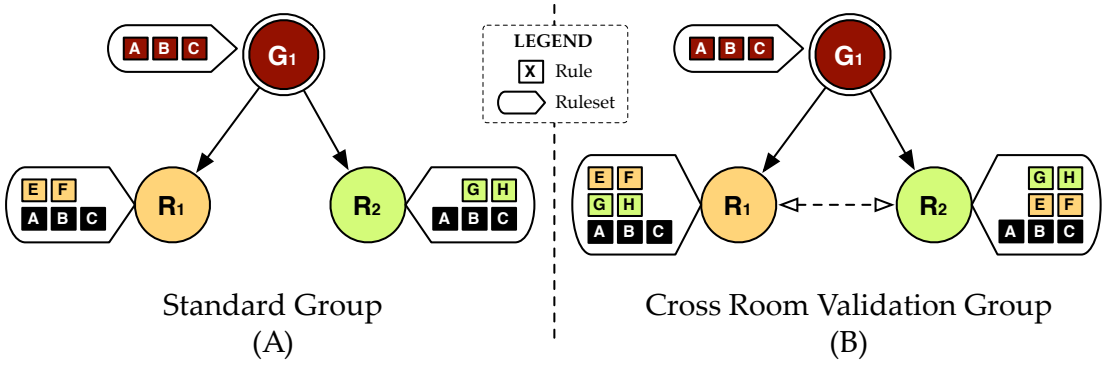

Figure 3.5: Representation of the two different kinds of supported groups

all the rulesets (for specified actions in CRVG) of the rooms belonging to that group.

3.3 BuildingRules 2.0: theoretical contribution

As we said in Section 3.1, the main problem in BuildingRules 1.0 was that the building occupants were not able to clearly understand the actual effects of the ruleset on the smart building, especially with an increasing number of rules. Let us make an example, consider these three rules:

1. if time is between 9am and 6pm then turn on the computer
2. if it is rainy then close the window
3. if someone is in the room then turn on the lights

it is relatively easy to understand the effects of the ruleset on the room. In fact, the computer will be turned on during working time, the windows will be close in case of rain and the lights will be turned on if the room is not empty.

Let us try to increase the number of rules and consider the following ruleset:

1. if time is between 9am and 6pm then turn on the lights
2. if external temperature is between 69F and 76F then open the window
3. if someone is in the room then turn on the lights
4. if time is between 9am and 6pm then turn on the computer
5. if it is rainy then close the window
6. if it is sunny then turn off the lights
7. if nobody is in the room then turn off the computer
8. if it is sunny then open the window

It is really hard to understand immediately which rules will be triggered by the system. The only way for the users to clarify how the building will behave, is to read the entire ruleset taking into account all the possible combinations of rules that can be activated. Moreover, the building occupants may forget to insert the rules to control the building in some day periods. In this example, in fact, we cannot say if the windows of the building will be opened or closed during the night because no rule has been inserted with that purpose. For this reason, the building occupants need a way to understand how the building will behave. Therefore, what is needed is a representation that models the building state and how the transactions between states occur. The representation that fits the aforementioned purpose, is the graph. Therefore, provided the ruleset of the building, we produced the graphs representing the formalization of the building state. These graphs are then used to support our framework to provide the users with suggestions about the building status, helping them to understand its behavior.

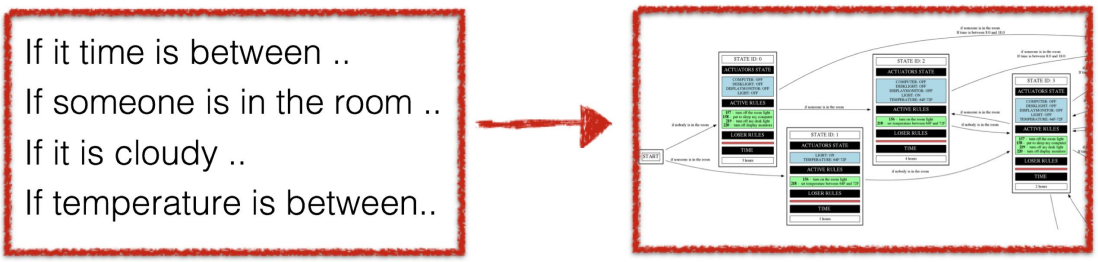

Figure 3.6: From Ruleset to Graph

In the following sections we will introduce the theoretical aspects behind our thesis work, in particular we will provide a detailed analysis about how we moved from the ruleset to the building behavioral and status graph.

3.3.1 From Ruleset to Building Behavioral and Status Graph

As said in the previous Chapters, commercial buildings have many occupants and many elements to be handled, such as lights, HVAC, windows and personal computers. The building occupants want to insert rules to customize the office the way they want. Therefore, the more users and devices the building has, the higher will be number of rules inserted. Dealing with a large amount of rules is difficult for both the building occupants and the building manager. In fact, the building occupants, before inserting a rule, need to read all the rules that are already in the room and the building manager needs to understand the building state and the impact that the rules will have on the building. What is needed, is a way to provide a clearer understanding of the effects of the inserted rules to them. Therefore, as can be seen in Figure 3.6, we leveraged the level of abstraction generating, starting from the ruleset of each room of the building, the oriented graphs that represent the behavior of the room.

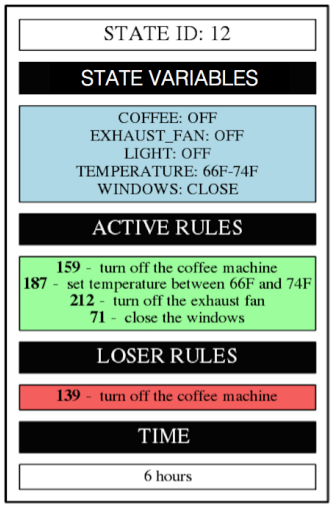

Figure 3.7: Representation of Node of the Graph

Node

As can be seen in Figure 3.7, the graphs node presents the following properties:

- **State ID**: which is used to uniquely identify the node;
- **State Variables**: that represent the values of each actuator and the environmental conditions that are currently controlled by BuildingRules;
- **Active Rules**: that represent the list of the consequent of the currently active rules, namely what rules are actually actuating in the room;
- **Loser Rules**: that represent runtime conflicting rules that are inactive because they have lower priority than the ones that are currently active;
- **Time**: that represents the number of times that a node is entered during the generation of the graph.

The ***State Variables***, as said before, represent the values of each state variable that is currently controlled by BuildingRules. A *State Variable* is one of the

set of variables that are used to describe the **Room State**. For example, the *State Variable "WINDOWS: CLOSE"* represents not only the fact that the windows are closed in the considered room but that a rule in the system is keeping them closed. All the state variables together form the *Room State*, that is the actual room status, not considering only the currently active rules but also the history of the previously activated ones. For example, in the state represented in Figure 3.7, the *State Variables* include *"LIGHT: OFF"* but there is no active rule that is currently actuating on the lights. This means that this state variable has been changed in a previous node of the graph but the effect of that action are affecting the current node state.

State Variables do not affect the room state by acting only on the state of the associated actuator, but they can have side effects also on the environmental conditions. For example, if we set a rule that turns on a light, it will not only physically switch the light on but it will also change the room luminosity. For this reason, we defined two types of state variables:

- **Direct**: the variables directly associated with the actuator
- **Indirect**: the variables that represents the environmental conditions, indirectly affected by an actuation

Table 3.3 lists some actuators that act directly on some State Variables and indirectly on others. If we consider a smart-window, it acts directly on its opening state, namely opened or closed, but, indirectly, on the temperature and humidity of the room. On the other hand, there are some actuators like smart-coffee machine, that acts only on its own state and do not affect other variables indirectly.

ACTUATOR	DIRECT STATE VARIABLES	INDIRECT STATE VARIABLES
Smart-Window	Window is opened / closed	Temperature Humidity (Luminosity)
Smart-Curtain	Curtain position	Temperature Luminosity
Smart-Light	Light is on / off	Luminosity
Smart-HVAC	HVAC is on / off	Temperature Humidity
Smart-Door	Door is opened / closed	Temperature Humidity (Luminosity)
Smart-Coffee Machine	Machine is running	-
...		

Table 3.3: Examples of direct and indirect State Variables

Introducing the distinction between direct and indirect state variables, makes possibile to adapt the graph analysis (see Section 3.3.2) to the different environment models. For example, if the rule "is someone is in the room then open the window" is triggered, we do not simulate only the actuation on the window (direct state variable) itself but we make also the humidity and the temperature (indirect state variables) of the room change.

The ***Active Rules*** represent the list of the actions of the currently active rules, namely what rules are actually actuating in the room. The *Active Rules* are stateless, this means that the *Active Rules* in one node are not affected by the ones in

the previous nodes. The effect that the active rules have on the room state are shown in *State Variables*. In fact, if there is a node in the graph in which there are some active rules, those rules will affect the *State Variables* directly or indirectly associated with its activation until some other rules modify them. For example, in Figure 3.7, the rule "turn off the coffee machine" affects the State Variable "COFFEE: OFF" that will be left unchanged unless the rule "turn on the coffee machine" will be present in the *Active Rules*. The *Active Rules* also provide the id of the currently active rules because it makes possibile to easily edit or delete a rule for the building manager if it is not needed anymore or if some anomalies are detected.

The ***Loser Rules*** represent runtime conflicting rules that are inactive because they have lower priority than the ones that are currently active. This node component is important to make detection of the possible conflicting rules at runtime. If a rule is found in Loser Rules it means that it will not affect the State Variables and, thus, will be ignored by the system in the current node.

Edge

The edge of the graph is oriented and annotated. The annotation represents the conditions that trigger a node change following the direction specified by the head of the arrow. In our case, the conditions are the antecedents of the rules that need to be triggered to change node. The most trivial case which may occur, is that there is only one condition associated with the edge. In this case, to pass through the aforementioned edge, the only condition that needs to be verified is the one associated to it.

It is also possible to find more than one condition on a single edge. In fact, as

shown in Figure 3.8, two different conditions can be annotated to an edge. The room goes from node with ID: 0 to the node with ID: 1 only if the antecedents are both verified, namely if both *nobody is in the room* and *time is between 6:00 PM and 6:00 AM* are verified. On the other hand, for example, if the only condition that is verified in the predecessor node is that *nobody is in the room*, the transaction does not occur because the conditions on the edge are partially triggered, namely they are not all true at the same time. Therefore, when two different conditions are annotated to an edge, it represents a logic AND.

The last possible situation that can take place is the one in which there will be two or more edges that start from the predecessor node and end up in the successor. In this case the transaction from one node to another occur when the conditions on one of the edges are verified. Therefore, it represents a logic OR.

Building Behavioral Graph

The Building Behavioral Graph (BBG) is the formal representation of the building state, made from building simulation (see Section **??**), that aims at exploring all possible states in which the building rooms can be set by the inserted rules. It makes the building manager easily understand how the building will behave during a specified time interval and how the active rules will impact on the current and future states of the building. Moreover, as soon as a rule is added or deleted from the system, it immediately shows the effect of the change on the building state.

As can be seen in Figure 3.9, the BBG starts from the node that we defined ***Node 0*** in which there are no rules and the *State Variables* field is empty. In this way, we can explore every possibile condition of the building rooms. However,

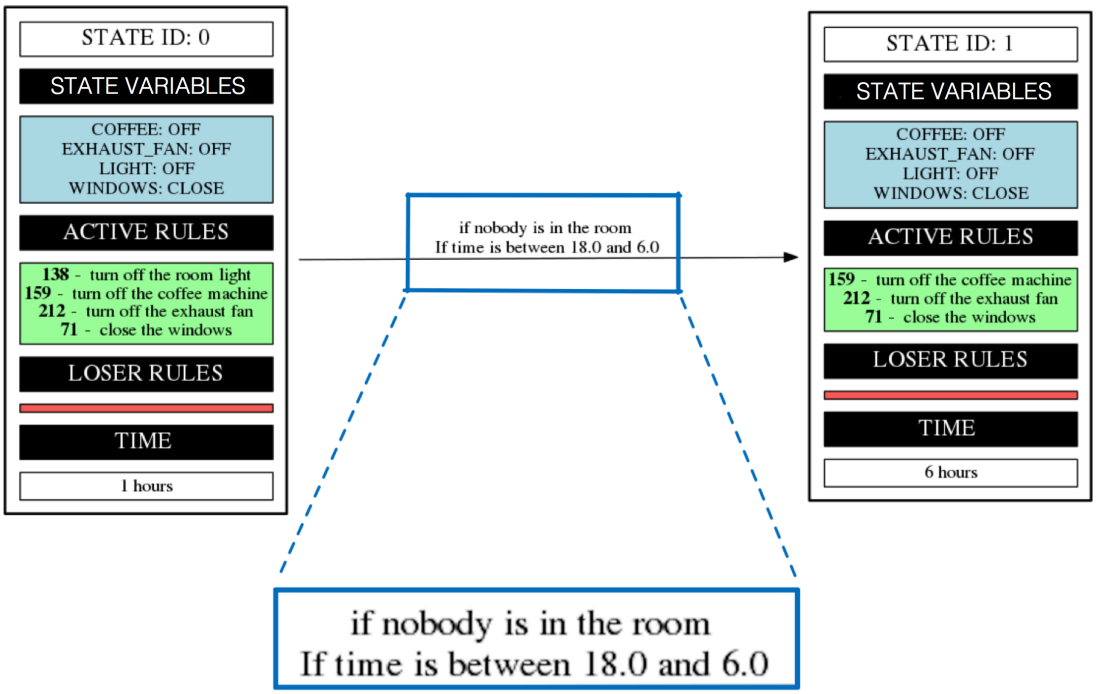

Figure 3.8: Representation of the Edge of the Graph

if needed, the BBG can start analyzing the behavior of the rooms from a specific room status. In this case, the BBG will explore all the possibile ways the rooms can evolve starting from that conditions.

First, a node in the BBG is univocally identified by the *State Variables*, the *Active Rules* and the *Loser Rules*. This way, we can have a more detailed understanding of the building behavior. We can distinguish situations in which, for example, we have two nodes in the graph that have the same *State Variables* field but with different currently active rules or that have the same active rules but one of them has a loser rule.

Second, in the graph, self loops will not be displayed. Self loops would be found when the only conditions triggered in the current node are the ones annotated on the incoming edges. The decision to not to show self loops was made because the loops were distracting and not useful to understand the evolution of the building. In fact, the assumption that we made is that it will not be possibile to leave a node until the conditions on one of the outgoing edges are verified. Since we did not want to lose the information about the fact that the room can stay in a certain node in presence of a self loop, we introduced the concept of ***Time***.

The *Time* field provides information about how many times an edge enters the node, considering also loops. Essentially, *Time* provides an estimation of the permanence time in a node and it can be exploited, for example, to make an estimate of the consumed energy of each room of the building considering the energy consumption of every device which is active in each node.

To have a clearer understanding of how a BBG works, we can analyze the snippet provided in figure 3.9. What can be seen is that, from the starting node,

we can reach two different nodes, one with ID: 0 and one with ID: 1, depending on whether somebody is currently in the room or not. One thing to notice, in this example, is the fact that these nodes have the same active rules but they are not the same node. In fact, as said before, the node of the BBG is univocally identified by all the components that compose it.

To generate this graph, we have to explore all the possible combinations of antecedents that can be verified with respect to the rules that have been inserted. The naive way consists in simulating all possible values in the domain of the antecedent, analyzing for each one the triggered actuators. Being it too expensive in terms of time, we decided to simulate only significant values in the domain that triggers all possible antecedents.

We used two different approaches depending on the type of antecedents, namely Boolean and Interval. In case of Boolean triggers, being just two values to be simulated, we made the simulation with both of them. For example, if we consider the rule *"if it is sunny then turn off the room light"* we will take into account both the cases, namely that it is sunny or not. On the other hand, in case of Interval trigger, we cannot simulate all the values in the domain of the antecedent because it is too expensive in terms of time. To make the understanding of our approach easier, let us make an example. Suppose that the only rule inserted in BuildingRules with respect to the room temperature is *"if room temperature is between 60F and 67F then close the windows"*. In this case, the only temperature values that need to be simulated to explore all the possibile cases in which the antecedents can be triggered with respect to temperature are: one value outside the interval and one value inside it, for example 59F and 63F. In case there is more than one rule with room temperatures in the antecedent, we

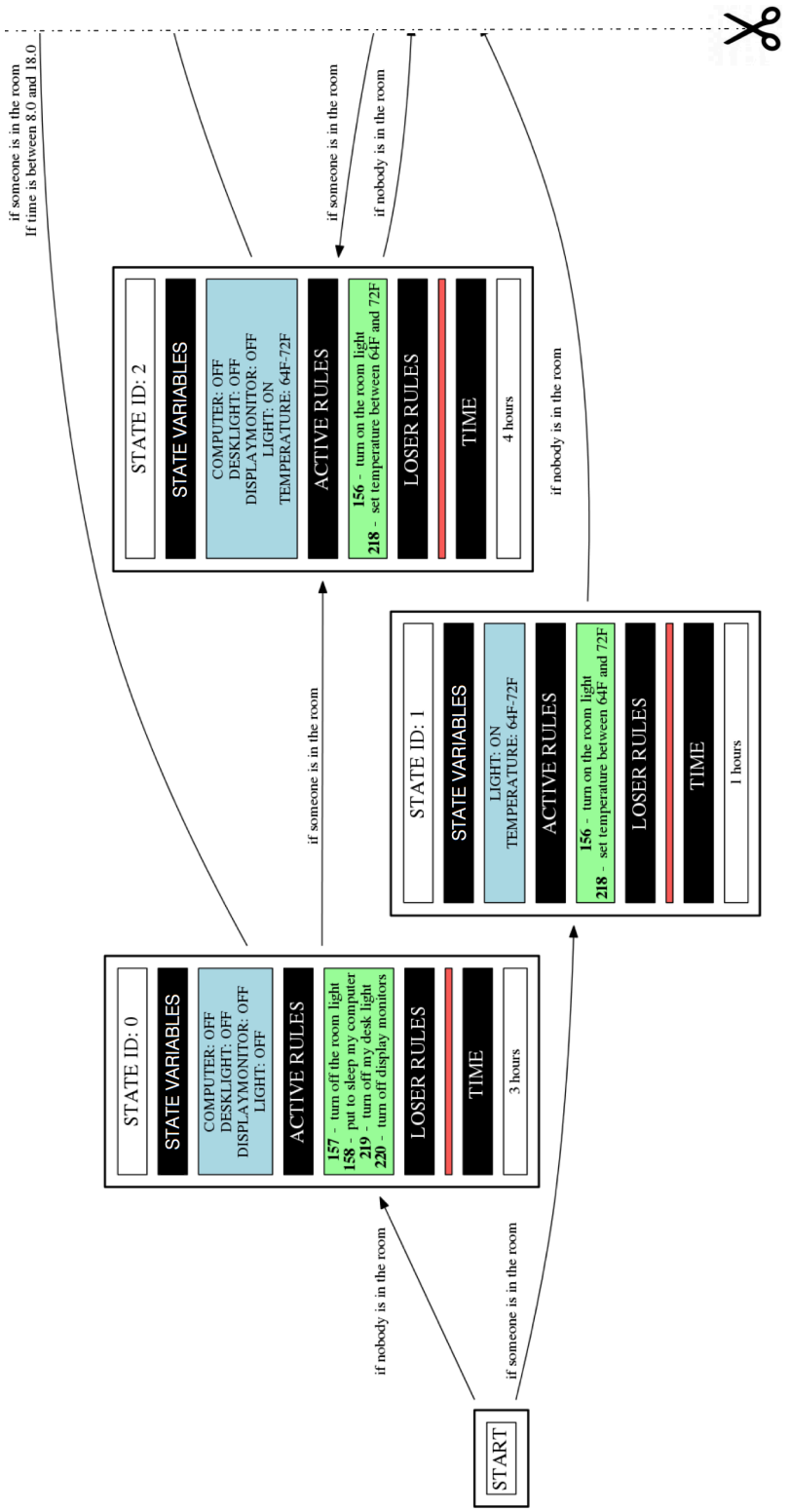

Figure 3.9: Snippet of a Building Behavioral Graph

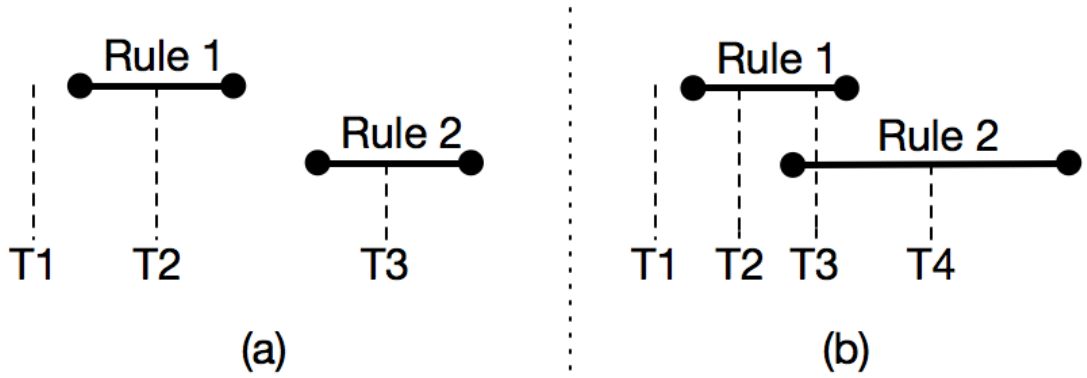

Figure 3.10: Interval Trigger

need to test all the values that assure us to simulate all the combination of rules that can be triggered. For instance, if we have two overlapping rules: *"if room temperature is between 60F and 67F then close the windows"* and *"if room temperature is between 65F and 70F then turn on HVAC"*, we are in the situation that can be seen in Figure 3.10.a. In this situation, one possible solution is to simulate the room behavior with the following values: 59F, 63F, 66F and 69F. Instead, if the intervals do not overlap, applying the same reasoning, we need to run the simulation with only three temperature values, as can be seen in Figure 3.10.a.

For sake of simplicity, here we presented only the case in which we have temperature values in the antecedent, but it is trivial to understand that this approach can be generalized to all the other integer antecedents, like humidity.

Building Status Graph

Increasing the number of rules in a room, we found out that the BBG becomes hardly readable and difficult to be managed because the number of nodes grows. For this reason, we need to introduce a new formalism and,

Algorithm 1 BBG Generation Pseudocode

```
1: function BBGGENERATOR(simulatedData)
2:     nodesList ← []
3:     nodeInfos ← getInfoFromData(simulatedData)
4:     if firstHour then
5:         for all node ∈ getNodesOfHour(hour) do
6:             if node ∉ nodesList then
7:                 addNode(nodeInfos)
8:                 nodesList.append(node)
9:             end if
10:        end for
11:    else
12:        for all oldNode ∈ getNodesOfHour(hour − 1) do
13:            for all node ∈ getNodesOfHour(hour) do
14:                updateNodeStateIfNeeded(node)
15:                if node ∉ nodesList then
16:                    addNode(nodeInfos)
17:                    nodesList.append(node)
18:                    addEdge(node, oldNode)
19:                else
20:                    addEdge(getSameOldNode(node), oldNode)
21:                end if
22:            end for
23:        end for
24:    end if
25: end function
```

thus, we generated another type of graph, that we called Building Status Graph (BSG).

The node in a BSG is uniquely identified only by the State Variables. This way, the BSG considers only the changes in the building behavior that actually affect the state of the building. The BSG is generated starting from the BBG structure, joining the nodes with the same State Variables. Both the active rules and the loser rules are obtained from the union of the previous nodes active rules and loser rules. This means that we do not need to simulate the building behavior again. As shown in Algorithm 2, starting from the BBG (Line 3), if the nodes have the same State Variables (Line 6) we will join them merging the Active Rules and Loser Rules. The arches are then redrawn (Line 14) considering the ones that connected the nodes of the BBG and connecting the BSG nodes according to the fact that now some of them have been merged.

As expected, the number of nodes decreases with respect to the BBG and also if the number of rules increases, the graph is readable and understandable. In the BSG the Time field has been removed because it is not needed in this representation. In fact, this graph aims at giving an event based evolution of the building which is not time based anymore.

As we can see in Figure 3.11, that represents a snippet of the BSG generated from the previously mentioned BBG (Figure 3.9), the BSG node with ID: 2,7 is the merge of the BBG nodes with ID: 2 and ID: 7. Making a comparison, this node has the same State Variables and Active Rules with respect to the one that can be seen in the BBG, but it has one more loser rule coming from the node with ID: 7.

Algorithm 2 BSG Generation Pseudocode

```
1: function BSGGENERATOR(buildingName, roomName)
2:     nodesList ← []
3:     BBG ← getBBG(buildingName, roomName)
4:     for all node ∈ getNodes(BBG) do
5:         for all node2 ∈ getNodes(BBG) do
6:             if getStateVariables(node) = getStateVariables(node2) then
7:                 unionActiveRules(node, node2)
8:                 unionLoserRules(node, node2)
9:             end if
10:        end for
11:        BSGNode ← addNode(nodeInfos)
12:        nodeList.append(BSGNode)
13:    end for
14:    addEdges(BBG, nodeList)    ▷ Takes the edges from the BBG and connect the
    one in the BSG
15: end function
```

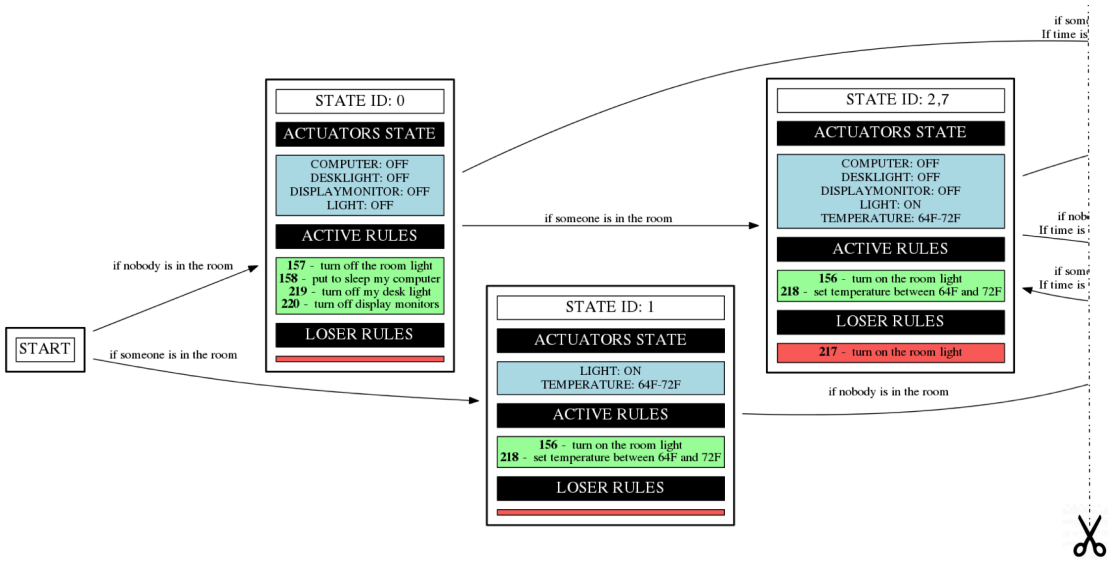

Figure 3.11: Snippet of a Building Status Graph

3.3.2 Analysis

Given the BBG and the BSG we can perform different analysis on the graphs to make suggestions to the building occupants. The main analysis we made are the following:

- Unmanaged State
- Uncontrolled State Variables
- Run-time Conflicting Rule
- Useless Rules

With the **Unmanaged State** analysis we can find out if the room is out of control in certain time periods. The **Uncontrolled State Variables** analysis, instead, check the presence of actuator installed in the room not used and, consequently, not controlled. The **Run-time Conflicting Rule** analysis finds run-time

conflicting rules before the conflict actually arises. The last analysis is the **Useless Rules** one that discover the unnecessary rules, namely the rules that will never be active.

Unmanaged State

To analyze the *Unmanaged State* analysis in detail and to understand the reasons that brought us to perform this analysis, we can consider a room with these rules inserted:

if time is between 0am and 8am then turn off the light

if time is between 10am and 1pm then open the window

The BBG that will be generated from these two rules, considering one day of simulation time, is the one that as shown in Figure 3.12. We can notice that in the BBG there are two edges that have no conditions associated that enter two nodes that have no active rules. This is an undesired behavior in a smart building because it means that we are losing control of the building behavior for a certain period of time, which means that we are probably wasting energy and, therefore, money. In fact, if we analyze these simple rules, we find out that from 8am to 10am and from 1pm to the end of the day we are not specifying any rule that can be triggered.

BuildingRules Graph Manager makes the analysis looking for empty edges that lead to nodes without active rules and will then provide a suggestion to the building occupants and to the building manager so that they can solve the possible problems.

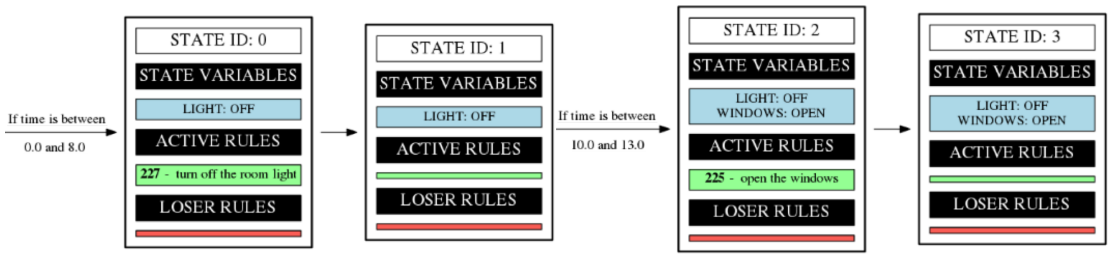

Figure 3.12: Toy Example that shows an unmanaged room with uncontrolled states

Uncontrolled State Variables

Regarding the *Uncontrolled State Variables* analysis we will consider the aforementioned example:

if time is between 0am and 8am then turn off the light

if time is between 10am and 1pm then open the window

Inspecting these rules we find out that there are are two actuators that we want to control in the room: the light and the window. What we can see analyzing the BBG in Figure 3.12 is that in the nodes with ID: 0 and ID: 1 we are not controlling the windows, we are not saying anything about how we want them to behave. This situation may be undesired, because when other rules will be added to the system it will be easier to lose control on its behavior.

BuildingRules Graph Manager makes the analysis collecting all the actuators placed in the room and checking that every node of the BBG contains in the State Variables all the actuators, which means they are controlled by BuildingRules. All the uncontrolled actuators will be shown to the building occupants.

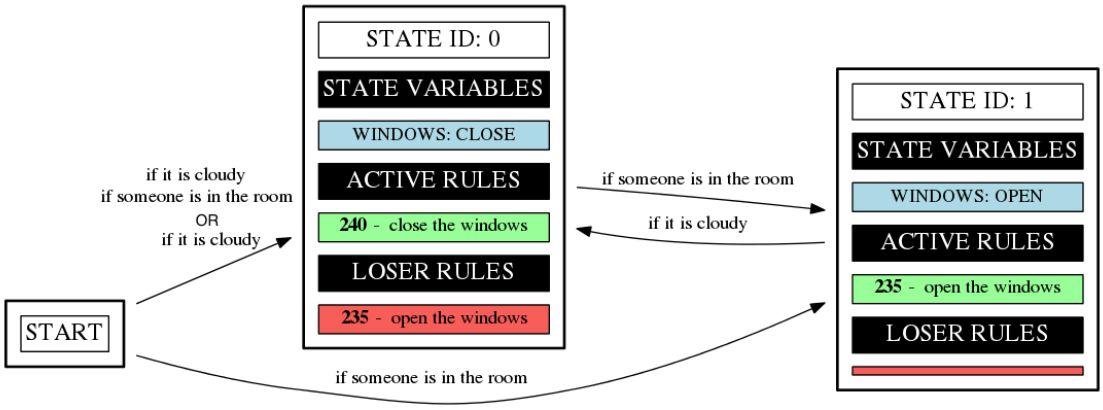

Figure 3.13: Toy Example that shows a run-time conflict

Run-time Conflicting Rules

With respect to the *Run-time Conflicting Rules* analysis, we will consider the following example. Consider a room with the following rules, ordered by descending priority:

$$\texttt{if it is cloudy then close the window} - \textbf{Priority: 100}$$

$$\texttt{if someone is in the room then open the window} - \textbf{Priority: 50}$$

The BSG that will be generated from these three rules, considering one day of simulation time, is the one that can be seen in Figure 3.13. From the ruleset, we can see that if someone is in the room when the weather is cloudy, BuildingRules needs to solve a run-time conflict activating only the rule with the higher priority, namely "*if it is cloudy then close the window*". Analyzing the BSG, we can see that in the Node with ID:0 the rule "*if someone is in the room then open the window*" is in the *Loser Rules* field.

In general, when the BuildingRules Graph Manager finds a loser rule in a node of the graph, it means that a run-time conflict is detected. BuildingRules will provide suggestions to the occupants and building manager notifying the

fact that a run-time conflict has been detected.

Useless Rules

To provide a better understanding of the *Useless Rules* analysis we need to introduce another example. Consider a room with the following rules, ordered by descending priority:

if time is between 0am and 10am then close the window – **Priority: 100**

if time is between 10am and 12pm then close the window – **Priority: 100**

if someone is in the room then open the window – **Priority: 50**

The BSG that will be generated from these three rules, considering one day of simulation time, is the one that can be seen in Figure 3.14. It is trivial to see, from the provided rules, that the rule "*if someone is in the room then open the window*" will never be activated, since it has a lower priority then the other two rules, that cover the entire day. In Figure 3.14 we can see that the BSG is composed by one node only. This is because there is no way we can move from this node with these rules in the system. Since the rule "*if someone is in the room then open the window* is in the *Loser Rules* field in every node of the BSG it means it is a useless rule and therefore, it will never activate.

In general, when BuildingRules Graph Manager finds a loser rule in a node of the graph, it checks if it is present in the Active Rules field. If it is not, it means that the above-mentioned rule is never active in the system because every time it is triggered it has lower priority than the active ones. As for the other analysis BuildingRules provide suggestions to the occupants and building manager

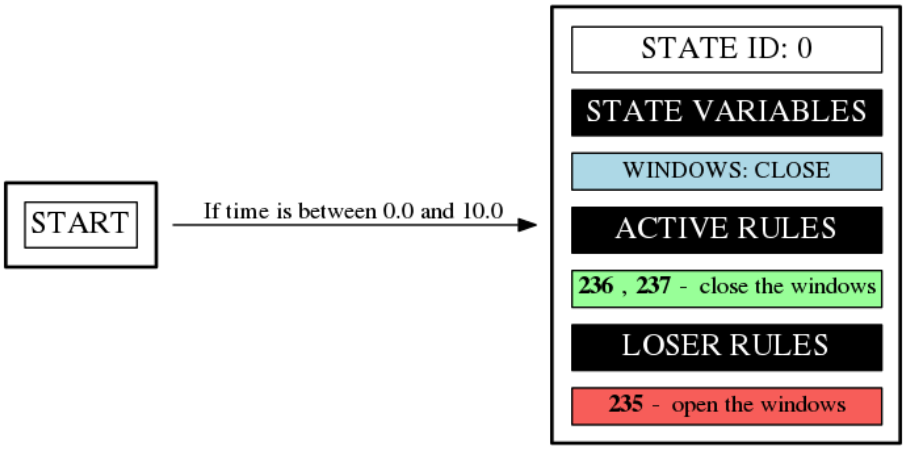

Figure 3.14: Toy Example that shows a useless rule

notifying the fact that the useless rule can be removed if its presence is not wanted.

Chapter 4

Implementation

In the previous Chapter the proposed methodology has been described. This Chapter presents BuildingRules. We will first provide an overview of the system and then a more detailed analysis about the software design patterns and the main technologies that has been exploited. Finally, we will show the web interface that we used in our experimental campaign.

4.1 General Overview

BuildingRules has been developed to be scalable, flexible and intuitive with respect to existing solutions. It has been designed as a RESTful HTTP/JSON web service in Python 2.7 using the Flask framework [57]. Flask is a **micro** web development framework for Python. We choose to use Flask because its "micro" feature keeps its core simple but extensible. In this way, the modules can be changed or added without critical problems.

As can be seen in Figure 4.1, BuildingRules is composed by a **frontend** that provides the user interface and a **backend** that communicates with the BMS,

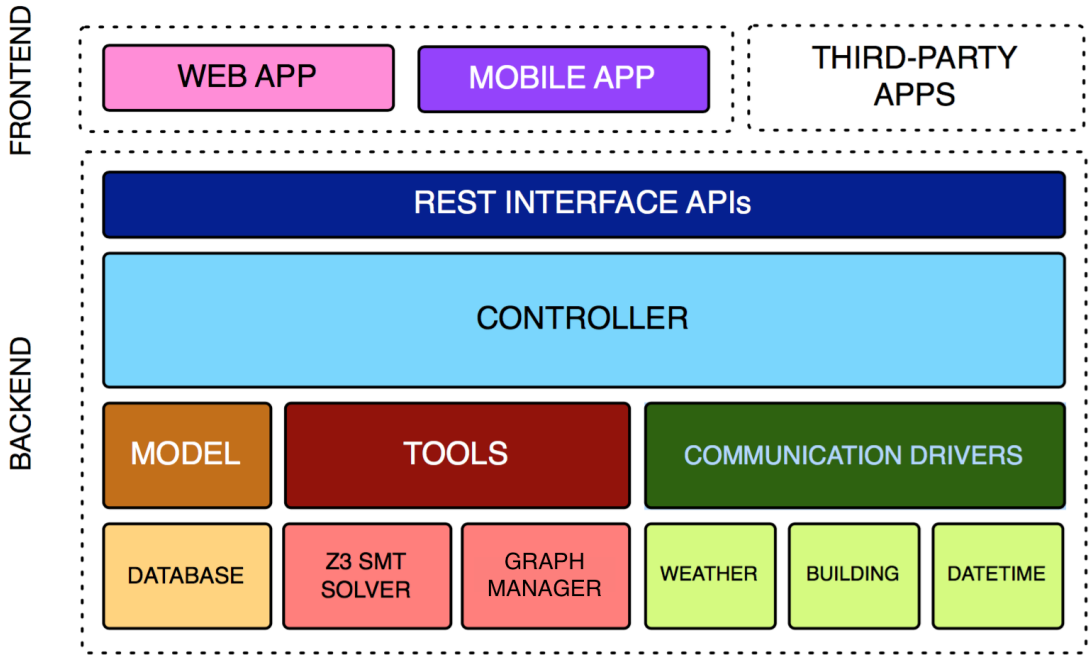

Figure 4.1: BuildingRules System Architecture

stores information about the rules, runs the conflict resolution algorithm, analyzes the graphs and gives suggestions besides providing RESTful APIs for native mobile applications or building management applications.

BuildingRules design follows the Model-View-Controller (MVC) architecture. The *model* is an abstraction layer over BuildingRules MySQL [61] database, the *view* is composed of the applications that can be developed as BuildingRules frontend and the *controller* implements the application logic. In particular, the controller validates the inserted rules using the Z3 SMT Solver, suggest the building occupants making use of the Graph Manager and stores the data to manage buildings, rooms, groups, users and rules using model. Moreover, the controller gathers the needed data about weather, building systems and date-time status through a specific model component which is a standardized driver interface. The frontend helps the users performing tasks like registering,

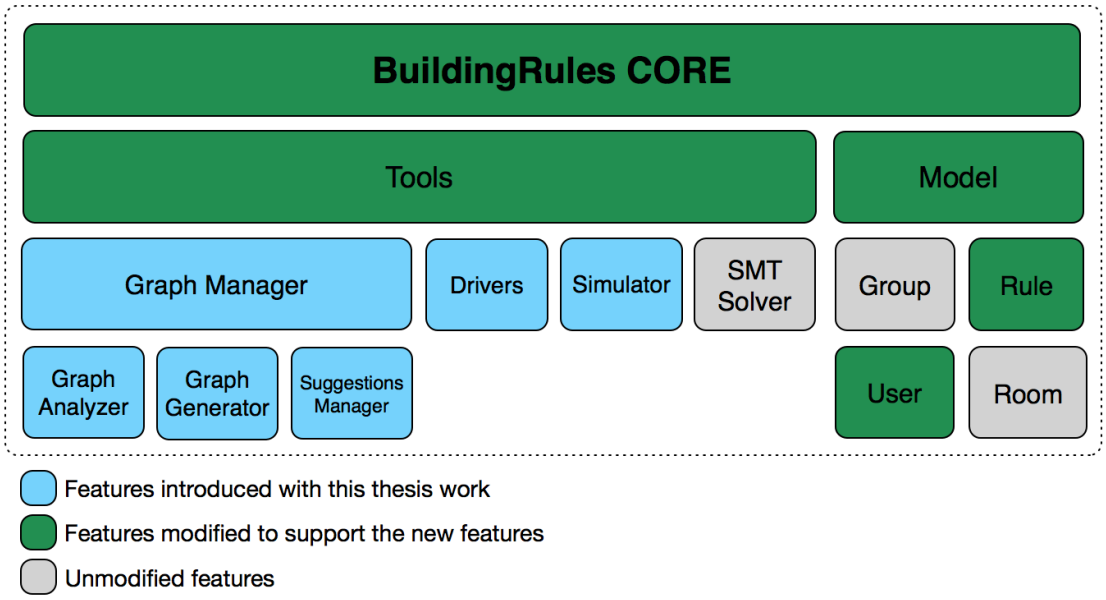

Figure 4.2: Differences between BuildingRules 1.0 and BuildingRules 2.0 features

adding triggers and actions or specifying rules for individuals rooms or groups of rooms. In addition, on top of the backend REST APIs, applications can be implemented. For example, we can think about applications that need to interact with a building, like a Demand Response application [58] that can inject rules to reduce energy use across all rooms when a pre-registered trigger condition is met or a Calendar Manager that can insert rules to turn ON and OFF the projector or modify temperature set points using room schedules.

BuildingRules 2.0 extends the previous version of BuildingRules (1.0) implementing the theoretical framework described in Section 3.3. For this reason we introduced new features and we modified others, adapting them to our needs. In Figure 4.2, we can see the details about the features used by BuildingRules 2.0. In particular, we introduced the *Simulator*, essential for the *Graph Manager* to generate the suggestions to the users exploiting its submodules, namely the *Graph Generator*, the *Graph Analyzer* and the *Suggestion Manager*. We

modified all the *Model* components we needed to support the new features, namely the *Rule*, the *User* and the *Room*.

4.2 Backend

We will describe the functionality of the main modules that compose the BuildingRules backend. We will first analyze the database logical schema, providing details about the main entities. Then, we will provide details about the controller and the main modules that compose it. Finally, we will explain how the RESTful APIs work.

4.2.1 Model

Commercial buildings are composed of many rooms, such as offices, conference rooms, laboratory and kitchens; these rooms can be grouped based on their purpose. The building occupants are usually assigned only to some specific building rooms, for example they can be assigned to their own office, but they can also share common spaces like the kitchen or the meeting room. To personalize the behavior of the building, the occupants can insert rules, that are composed of a trigger and an action, in the system. To store all the information needed to manage BuildingRules we have to create the room and user entities, that permit to describe the rooms and the occupants in the building. We have also to model the information about the rooms groups and all the details about the rules that the occupants insert. The model in BuildingRules provides an abstraction layer which is used from the core system to communicate with both the database and the physical devices protocols. The communica-

tion between the core system and the database is performed by an Application Programming Interface (API) allowing to manage the MySQL database from the controller. As can be seen in Figure 4.3, the main entities that compose the structure of our **database** are:

- **Building**: this entity represents the building in which BuildingRules is installed and it can have one or more *Rooms*;
- **Room**: this entity represents the room located in the *Building*;
- **User**: this entity represents all the information of the building occupants among which we can find the level of the user;
- **Group**: this entity represents the information about the rooms that are grouped;
- **Rule**: this entity represents all the rule information. We also added the *number_of_edits* attribute to handle maximum number of rule changes during the experimental campaign;
- **Trigger, Action**: these entities represents the antecedent and the consequent of a rule.

Trigger and **Action** are connected directly to the room because each single room need to be aware of which sensors and actuators are installed. This way, we can show only the rules that are compliant with the room capabilities. For example, the rule with the antecedent "if the temperature is between 69F and 75F" can be added only if there is a temperature sensor in the room.

In BuildingRules 2.0 we have made some changes to the database structure, in particular we added the *number_of_edits* field. We also added methods to the

APIs interfacing with the database changing the format of the *User* data and adding the methods to manage the *number_of_edits* field during the experimental campaign. These improvements made feasible the implementation of the *Graph Manager* and its submodules.

4.2.2 Drivers

The communication between the core system and the physical devices protocols is managed by the **Drivers** module. This module acts as an abstraction to different sensor protocols providing a standard naming convention across different vendors, allowing us to deploy BuildingRules across different buildings with minimum effort. There are two type of Drivers: *genericTriggerDriver* and *genericActionDriver*. A *genericTriggerDriver* takes as input a sensor source (e.g., a humidity sensor in a room) and a condition to verify (e.g., the humidity is between 40% and 50%). It provides a notification through the eventTriggered method when the antecedent is true. Instead, a *genericActionDriver* takes a target actuator (e.g. a HVAC control system) and the actual value (e.g. set humidity to 35%) as input, and uses an actuate method to execute a rule action. The drivers are implemented exploiting the class inheritance, which makes the structure more extensible. A driver inherits attributes and generic behavior from *genericTriggerDriver* or *genericActionDriver*, depending on whether it is a trigger or an action. Moreover, the driver implements its methods according to the task it needs to accomplish. For example, if we want to control a HVAC in a specific building room, we will need to use *roomHvacActivationDriver* that extends *genericActionDriver* and will be in charge of handling the HVAC. Others examples of drivers implemented in BuildingRules are: the *roomLightActi-*

vationDriver to control room lights, the *roomWindowActivationDriver* to open or close windows or the *weatherTriggerDriver* to gather information about weather conditions.

In BuildingRules 2.0 we have implemented the drivers needed to run the experimental campaign in a real environment (see Section 5.4) interfacing with OpenHAB. In particular, we implemented the drivers to communicate with the available devices, like lights (*roomLightActivationDriver*) and occupancy detection sensors (*roomTriggerDriver*).

4.2.3 Controller

The core module of BuildingRules is the **Controller**. As said before, it is in charge of processing all the request made by the users, like the registration or the insertion or the deletion of a rule in a specific room, to perform all the tasks that are needed to resolve conflicts statically and dynamically and to provide suggestions to the building occupants and to the building manager. The controller communicates with both the view, to collect the users inputs, and the model, to gather data from the devices and to access the database. The main tools that the Controller exploits to provide the aforementioned functionalities are the *SMT Solver*, the *ActionExecutor*, the *Building Simulator* and the *Graph Manager*.

The **SMT Solver** is in charge of solving static conflicts among the rules inserted by the users. First, it translates all the rules triggers and actions inserted in the room to match the form that is allowed by Z3 (see Section 3.2), then it adds them to the list of rules to be validated and, finally, the list items are iteratively checked against the solver that performs the logic resolution to check the

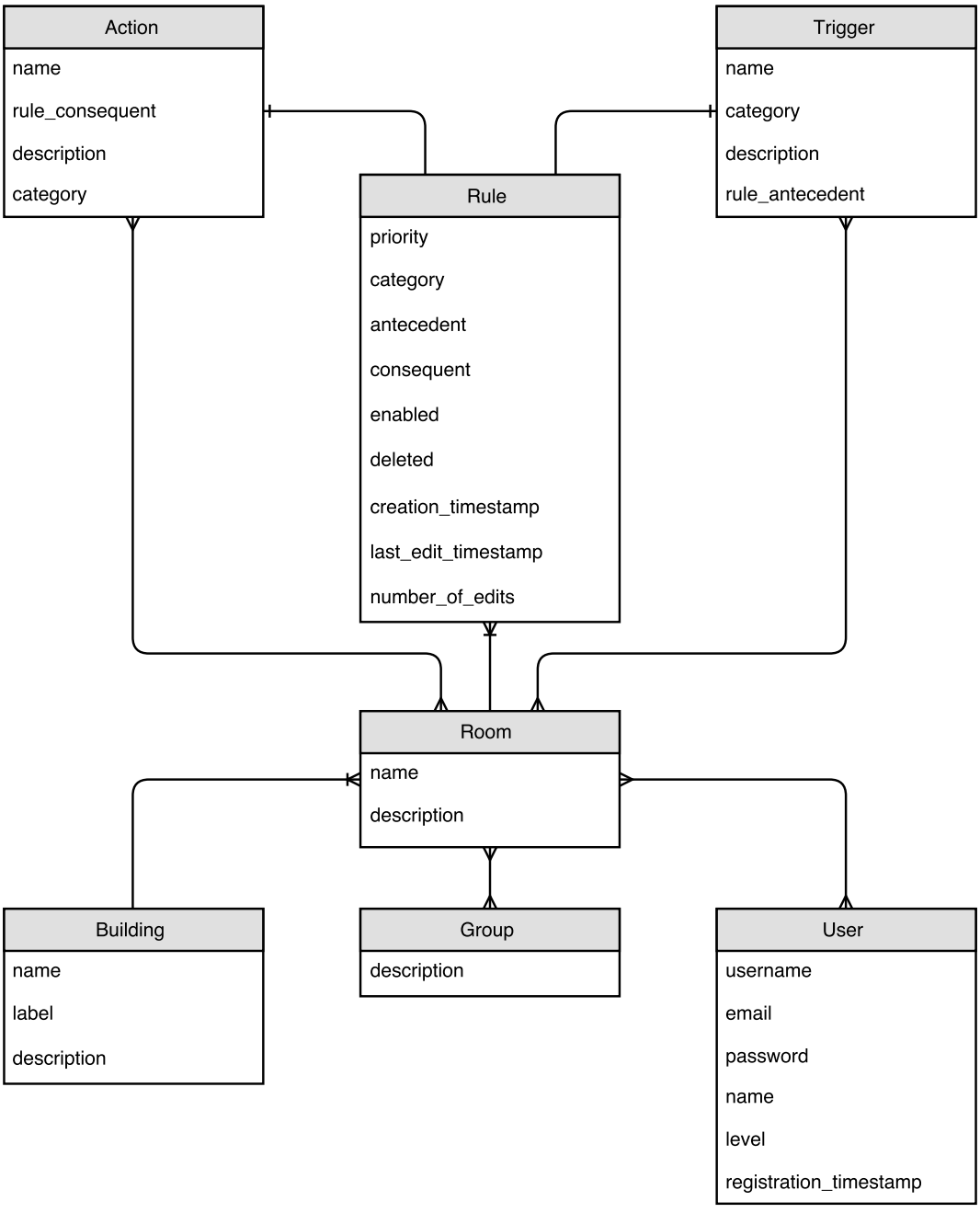

Figure 4.3: BuildingRules Database

satisfiability of the ruleset. The satisfiability problem consists in determining if exists an interpretation that satisfies a given logic formula. In other words, it establishes if the variables of a given logic formula can be assigned in a way that make the formula evaluate to TRUE. If no such assignments exist, the function expressed by the formula is FALSE for all possible variable assignments and the problem is unsatisfiable, otherwise it is satisfiable. Since the solver can only verify the satisfiability of a problem and not the contrary, given a ruleset with cardinality n, we need to generate n SMT problems, one for each rule. If only one of the generated SMT problems is unsatisfiable it means we found a conflict.

The **Building Simulator** is a fundamental part of BuildingRules 2.0 as it was needed to generate the *Building Behavioral Graph* and to test our work in a simulated environment. For this reason, we focused on two different simulator solutions, one for the experimental campaign and an extended one to make analysis and give suggestions. The *simulator* is a time-driven tool that, for each step, reads the specified environment conditions (the room temperature, the weather condition, the occupancy status, etc.) and checks which rules are triggered under the provided conditions. We develop two version of the simulator. In the first version, used in the experimental campaign, all the environment conditions are specified as parameters, as shown in Algorithm 3, and the behavior of each rooms in the building is simulated under the given conditions. The algorithm is composed by two phases and perform the simulation hour by hour. In the first phase, after having fetched all the room rules (Line 4), the algorithm verifies which rules are triggered with the provided parameters (Line 5) and it adds them to the triggeredRules (Line 6). In the second one, for each

rule category, the rule with the higher priority that do not generate conflicts is chosen to be activated (Line 14). We set up the core of the algorithm in this way because it is extensible and support different environment models. For example, we can model the room temperature changes as linear with respect to time. If air conditioning is ON it will keep changing linearly until it reaches its set-point, whereas, if the air conditioning is OFF and windows are OPEN the temperature would change linearly as a function of difference between indoor and outdoor temperature. The second version of the algorithm, namely the one that enabled us to make the analysis we described in the Section 3.3.2, simulates all the possible combination of environment conditions that need to be simulated by specifying them as parameters, as can be seen in Algorithm 4. We simulate all the combination of room temperature (Line 2), external temperature (Line 3), occupancy status (Line 4) and weather condition (Line 5), but it is easily extensible to support any kind of environment variable.

The **ActionExecutor** is the module that is in charge of handling the real and simulated rules actuation. It automatically detects if BuildingRules is acting in a real or a simulated environment and, therefore, it choses if the action needs to be actually executed on the building or if it just needs to be logged to a file. The *ActionExecutor* is a daemon that runs in background and constantly monitor the rules inserted in the building. During the actuation phase, the *ActionExecutor* checks which rules needs to be triggered and it solves the runtime conflicts by actuating only the rule with the higher priority in a specific category, as we have already explained in Chapter 3.

The task of providing suggestions to the users, instead, is delegated to the **Graph Manager**. It generates the Building Behavioral Graph (BBG) and the

Building Status Graph (BSG) starting from the simulation of the building behavior and, starting from the graphs, it performs all the analysis that were explained in the previous Chapter. The graphs were created using NetworkX [59], a Python software package to create and manipulate complex networks. The graphs structure is then saved to a file in the Backend and, afterwards, analyzed making use of NetworkX graph exploration tools. The graphs structure was kept separated from the actual visualization, which is made only when the user request one of the graph to be shown from the frontend. The visualization of the graph is made with GraphViz [60], which is an open source graph visualization software, to which we provided the graph structure as inputs. The output is an HTML based graph that is finally shown to the user.

4.2.4 REST Interface APIs

To enable the communication with the *Frontend* we developed a RESTful interface. As can be seen in Figure 4.1, the **Rest Interface** is the layer between the *Controller* and the *Frontend*. The APIs provide the methods needed by the applications built on top of it to exploit BuildingRules functionalities in a real environment. They enable the users to register to the system, giving them the chance to add triggers and actions, to specify rules for each single room or for a group of rooms. Table 4.1 shows the available APIs. In particular, we can find the *description* of each endpoint and, in case of an API that allows multiple action to be performed, we can find the <*Action*> column that lists all the available possibility.

To expose these APIs to the aforementioned applications, as previously said, we used Flask. The methods that we implemented on Flask communicate with

Algorithm 3 Simulator Pseudocode

```
1: function STARTACTIONEXECUTOR(parameters)
2:     for all room ∈ getBuildingRooms(parameters.building) do
3:         triggeredRules ← []
4:         for all rule ∈ getAllRoomRules(room) do
5:             if isTriggered(rule, parameters) then    ▷ True if rule is triggered with
   the provided parameters
6:                 triggeredRules.append(rule)
7:             end if
8:         end for
9:         triggeredRules ← orderByPriority(triggeredRules)
10:        alreadyAppliedCategories ← []
11:        for all rule ∈ triggeredRules do
12:            ruleCategory ← getRuleCategory(rule)
13:            if ruleCategory ∉ alreadyAppliedCategories then
14:                actuate(rule)
15:                alreadyAppliedCategories.append(ruleCategory)
16:            end if
17:        end for
18:    end for
19: end function
```

Algorithm 4 Simulator Extension Pseudocode

```
1: function SIMULATOR(parameters)
2:     for all roomTemperature ∈ parameters.roomTemperatures do
3:         for all externalTemperature ∈ parameters.externalTemperatures do
4:             for all occupancy ∈ parameters.occupancies do
5:                 for all weather ∈ parameters.weathers do
6:                     startActionExecutor(filteredParameters)
7:                 end for
8:             end for
9:         end for
10:    end for
11: end function
```

the frontend using POST. This choice was made because, this way, the HTTP request does not have restrictions on data type and on data length. The responses to the frontend requests are provided in JSON (JavaScript Object Notation), that is a lightweight data-interchange format. In this way, for example, the implementation of the "/rooms/<roomName>/graph" API was straightforward and it immediately enabled the user to get the BBG or BSG.

4.3 Frontend

The frontend, in BuildingRules, is designed to be modular and composed by different applications that need to interact with the underlying building. In fact, our backend, through the RESTful APIs enables the integration of multiple frontend applications to match the need of the commercial building. In this Section we will present the web view, the main frontend application, used by

Web API	Request Method	Description	<Action>
/buildings	POST	get buildings list	
/buildings/<buildingName>	POST	get building info	
/buildings/<buildingName>/rooms	POST	get rooms	
/users/login	POST	user login	
/users/register	POST	user registration	
/users/logout	GET, POST	user logout	
/groups/<Action>	POST	add/delete group	add, delete
/groups/<groupId>/rules	POST	get group rules	
/groups/<groupId>/rooms	POST	get group rooms	
/groups/<groupId>/<Action>	POST	get group triggers and actions	triggers, actions
/rooms/<roomName>	POST	get room info	
/rooms/<roomName>/<Action>	POST	add/delete room	add, delete
/rooms/<roomName>/simulation	POST	get room simulation results	
/rooms/<roomName>/activeRules	POST	get room active rules	
/rooms/<roomName>/<Action>	POST	get room triggers and actions	triggers, actions
/rooms/<roomName>/<Action>	POST	get room users/groups	users, groups
/rooms/<roomName>/conflictingRules	POST	get room conflicting rules	
/rooms/<roomName>/rules	POST	get room rules	
/rooms/<roomName>/rules/<Action>	POST	manage room rules	add, delete, enable, disable, setpriority, edit
/rooms/<roomName>/graph	POST	get BBG/BSG graph	
/rooms/<roomName>/graphAnalyzer	POST	get analysis	

Table 4.1: REST interface APIs

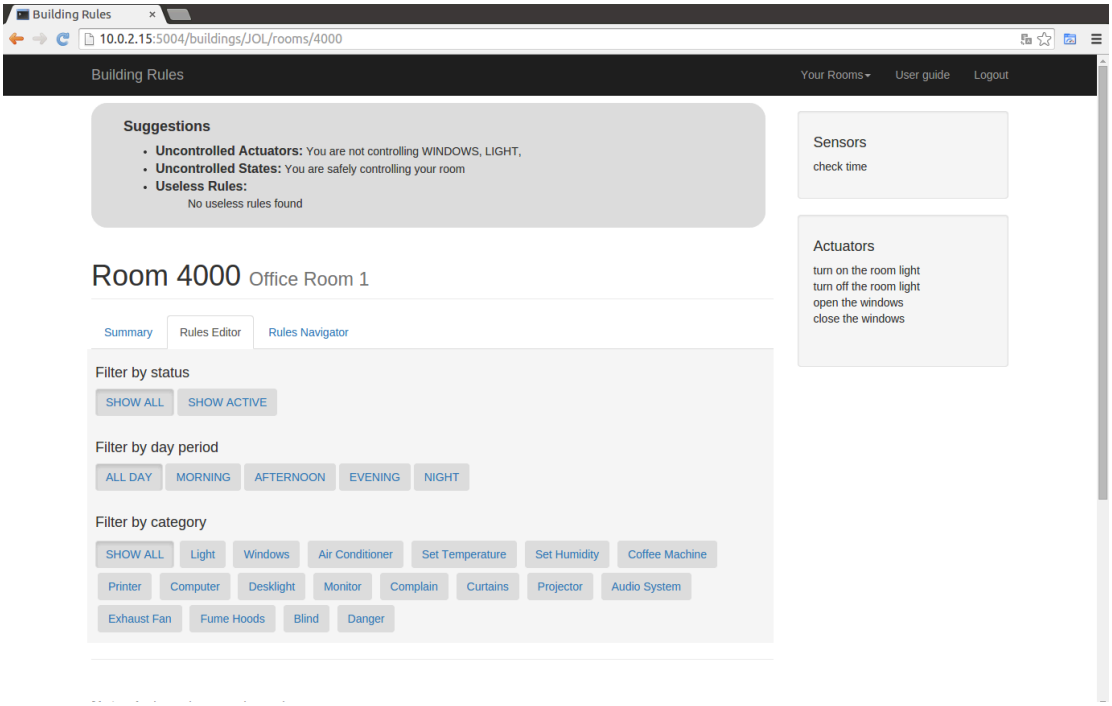

Figure 4.4: Room Home Page

the occupants to express their preferences about the building.

Web Interface

Our web interface starts from two simple pages that allow the registration and the login on BuildingRules. After the login process, the user interface shows the list of the room that the user can manage. For each room BuildingRules provides three different visualization tools (views can be found in Appendix A) : a *Summary* tab, a *Rules Editor* tab and a *Rules Navigator* tab. The *Summary* tab presents a Gantt diagram that shows the behavior of the room given some fixed parameters such as occupancy, room temperature, external temperature, etc. The *Rules Editor* tab presents the list of the inserted rules ordered by priority. Through this interfaces the user can add, delete, modify, enable and disable rules. The currently active rules will be shown in green to give a visual

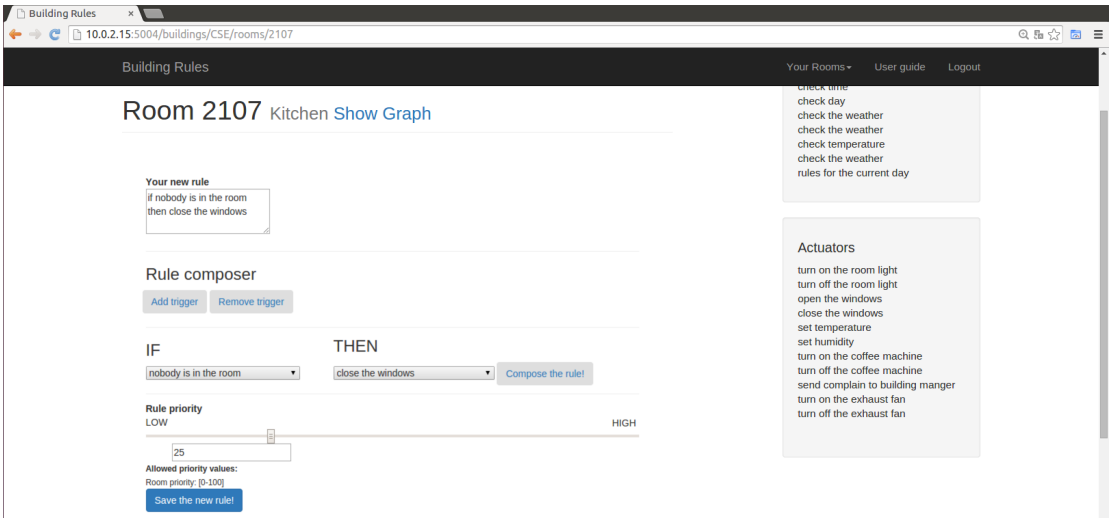

Figure 4.5: New Rule Page

feedback to the users on the current building status. Despite this, as the number of rules in a room increases, it becomes harder to read and understand all the rules. For this reason, we implemented three filters whereby the user can filter the rules by status, day period and category. Finally, the *Rules Navigator* tab visualizes rules in a table in which each row represent an action category. The addition of a new rule, as shown in Figure 4.5, is simple and intuitive, the user can choose the trigger and the action that compose the rule and, then, the associated priority. When the user inserts a rule, if conflicts are detected, the interface shows the rules that are in conflict with the new one, as shown in Figure 4.6.

The room page shows also the suggestions (Figure 4.4) about the *useless rule*, *unmanaged state* and *uncontrolled state variables*. From this page the building manager can move to the view that enable interaction with the *Graph Manager*. As shown in Figure 4.7, we implemented a simple web terminal in javascript that allows the building manager to generate BBG, BSG and to run all the build-

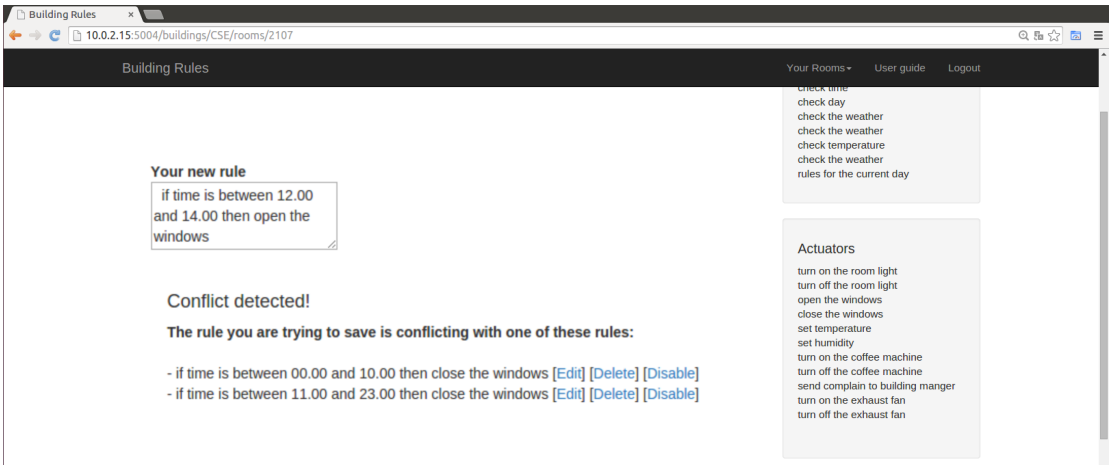

Figure 4.6: Conflict Detection Page

ing analysis. Moreover, the terminal let the building manager enable or disable the rules making use of the ids provided by the graphs. This interface makes the changes of the building state immediately clear to the building manager because the graphs will be immediately redrawn.

In BuildingRules 2.0 we changed the room page adding the suggestions, the *Summary Tab* and the filters in the *Rules Editor* tab. We added the page that enable the graphs management as described above.

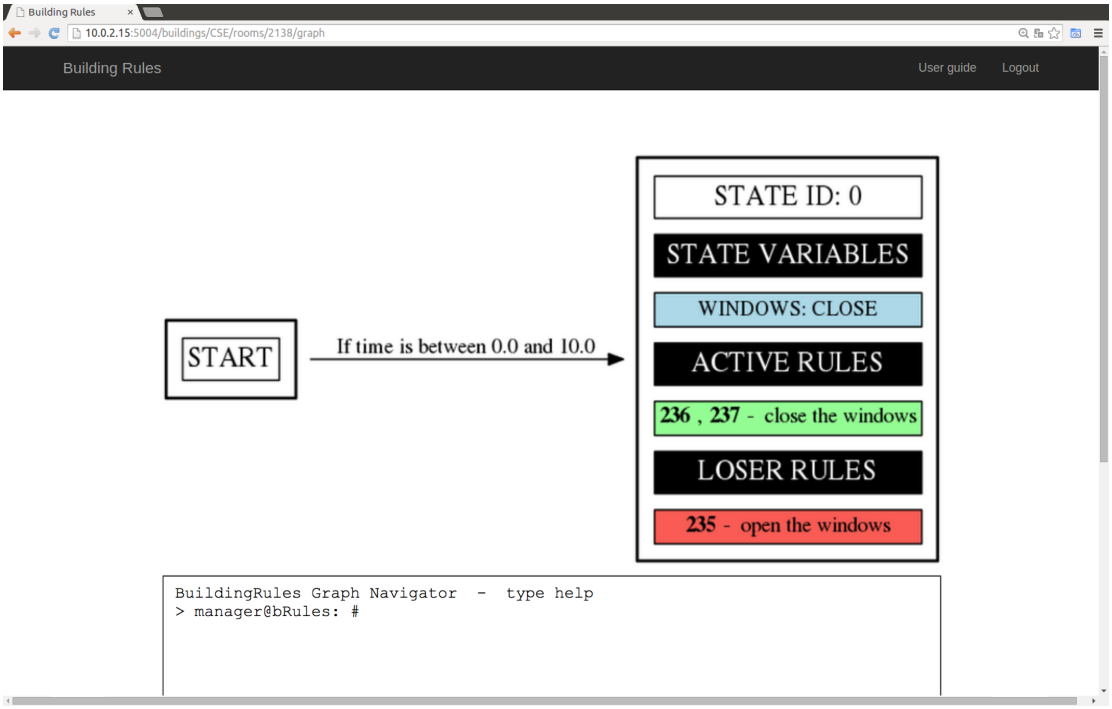

Figure 4.7: GraphGenerator Page

Chapter 5

Experimental results

This Chapter provides details about the experimental campaigns that we conducted to validate our thesis work. We will first present the baseline on which we will compare our results, then we will focus on the two experimental campaigns made to validate both the usability of the proposed system and the ease of integration.

5.1 Overview

The main problems that affect smart buildings, emerged from the state of the art presented in Chapter 2, are the following:

- the interaction between the occupants of the building and the building itself is hard, especially as it regards the expression of complex tasks, because buildings are managed by means of sensors and actuators;
- smart buildings are not designed to be controlled by multiple users, which lead to conflicting situations among the inputs when we allow the occupants to personalize the environment in a commercial building scenario.

Therefore, BuildingRules was designed to provide an intuitive interface to the occupants of commercial buildings, enabling them to customize their office spaces using trigger-action programming. In addition to this, BuildingRules automatically detects conflicts among the policies expressed by the occupants, being able to give users suggestions to help them in the building management.

Before providing the details on the experimental campaigns we conducted, we will introduce the methodology we used to test our results. As we can see in [62], nowadays, one key approach towards the vision of a smart building that can support the goals of the occupants is putting more effort on studying technologies "in the wild". This means that in smart building research field, the studies of new proposed technologies need to be tested on the users and on the buildings to collect information directly from them. Therefore, the approach that we followed to test our system, was the implementation of the minimum set of features that allowed us to test a reasonable solution collecting baseline results. Once the data had been processed and analyzed, we made other experimental campaigns to understand if we were following the right path, testing both the usability and the integration of the system.

5.2 Baseline Experiments

As said before, to collect the baseline results of our thesis work, an experimental campaign has been conducted using the first version of BuildingRules (see Section 3.2). Evaluating BuildingRules in an actual environment is not trivial. On the one hand, it is possible to create a testing environment, creating from scratch a typical smart commercial building and installing an appropri-

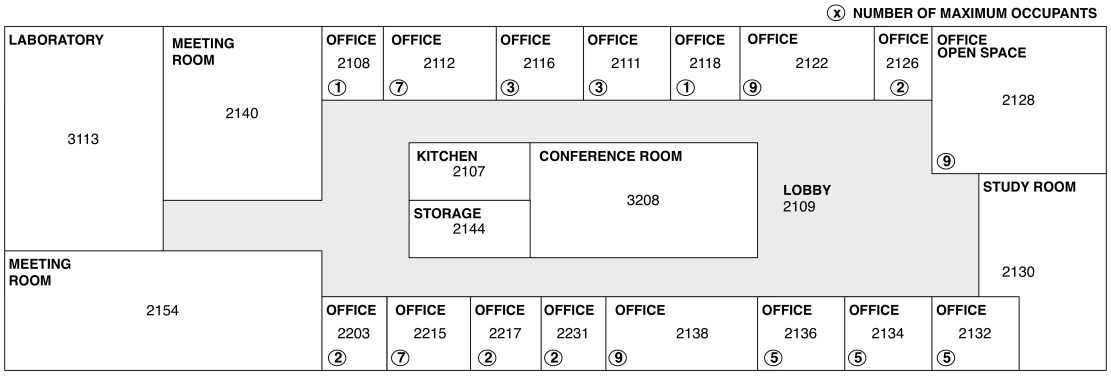

Figure 5.1: Virtual Office Environment

ate BMS. In this case, to make the tests relevant, the dimension of the building, the number of occupants and the amount of smart devices placed in the testing environment need to be appropriate. For these reasons, reproducing a smart environment, would be prohibitive in terms of cost. On the other hand, it is possible to install BuildingRules in an existing real smart building that already provides its own BMS. Since BuildingRules has never been tested in a real commercial environment, it risks to endanger the building occupants and to compromise the security of the building itself. For example, if BuildingRules runs into a technical malfunctioning during a fire, it could lock the occupants inside the building.

Since the aforementioned problems were not possible to solve, BuildingRules was evaluated in a virtual office environment, designed to recreate the structure of a real commercial building. Usually, a commercial building is composed of many rooms and each of them has a specific purpose. In addiction to the offices, usually assigned to a limited amount of people, shared spaces, like meeting rooms, kitchen or conference rooms, can be found. Therefore, as can be seen in Figure 5.1, the building plant has been designed to be representative of a typical office, and incorporated different types of rooms such as conference

rooms, research laboratory, kitchen, storage and offices. A unique id has been assigned to each room, and, with respect to the offices, a number that identifies the maximum amount of users allowed to work in there has been introduced. In the virtual environment all the components that we can usually find in a real one have been inserted. For example, computers, lights, desk lights, printers and other office related devices have been placed in the offices, whereas the projector and the audio system can be found in the meeting room.

Each participant was assigned to a random set of rooms, for example, an office space, kitchen, and meeting room. The participants were told to use BuildingRules for at least 10 minutes and complete a set of actions, i.e., add, remove, edit rules, each day (10 actions the first day, then decreasing each day. Average of 5 actions per day). A final survey was taken at the end of the week to understand the usability of the system. Each user was required to have at least one month of office experience for participation. Not all the participants started at the same time, and we had a total of 23 users spread over 17 days. Notice that there is no bias in the choice of the population; the requirements in the selection were imposed to test the system with a population conformance to the target of BuildingRules, i.e., offices.

A total of 636 rules has been obtained from this study, with an average of 15 rules per room, and 16 rules per user. Figure 5.2 shows the distribution of these rules across the various triggers and actions. The default status rules were inserted by us for scheduling of lights and HVAC system when no other rules were specified for a room. The most popular trigger was the day of week, followed by occupancy. The most popular action was lights, followed by plug loads (computer, desk light, monitor and printer). The study was designed in

	Date	Day	Default Statatus	External Temperature	Occupancy	Room Temperature	Time	Weather
Audio	0	9	0	0	6	0	3	1
Coffee Macchine	2	2	0	0	2	0	3	0
Computer	1	43	0	0	5	0	7	1
Desk Light	1	41	0	0	7	0	5	3
Display Monitor	2	39	0	0	9	0	1	1
Printer	1	40	0	0	5	0	5	0
Projector	0	10	0	0	10	0	6	0
Blind	3	8	0	0	6	0	7	24
Exhaust Fan	1	2	0	0	1	1	2	0
Fume Hoods	0	0	0	0	2	1	0	0
HVAC	12	18	0	4	23	13	3	6
Room Humidity	0	1	24	2	7	3	2	4
Room Temp.	3	4	24	6	9	6	1	1
LIGHT	5	47	5	0	38	0	18	24
Send Complain	0	1	0	2	8	19	5	0
Windows	4	12	0	26	22	9	4	29

Figure 5.2: Composition of rules created during the baseline campaign

a way that forces participants to create or modify rules, so that we can analyze effect of the combination of these rules in the virtual office. As a result, some of the rules inserted in the rooms were very similar, but not in conflict. For example, in one of the rooms there were three rules to open windows with three different temperature ranges. In a real deployment, users would probably use one rule to cover all the three temperature ranges.

To evaluate the usability of BuildingRules the participants was asked to fill a survey. Table 5.1 shows the results of our survey. In this survey it was noticed that the system was a good idea (7.0) and BuildingRules can be useful in offices (7.6). On the other hand, what was found out is that it was very difficult to understand the behavior of the room when the number of rule increase (5.0) and also insert (5.4) and edit (5.8).

Survey Questions	Baseline	Experimental Campaign
Overall impression score	7,0	7,8
System usability	6,0	7,6
Would BuildingRules be useful in your office?	7,6	8,2
How easy was it to insert new rules?	5,4	8,3
How easy was it to edit existing rules?	5,8	5,9
Do you like the philosophy behind the system?	7,0	8,8
How easy was it to resolve conflicts within the rules?	5,8	5,0
Was it easy to understand how the combination of rules will affect the office?	5,0	7,2
How useful was the *rule editor* to identify individual rules?	x	7,2
Was it possible to grasp all rules using *rule navigator*?	x	7,1

Table 5.1: Surveys results

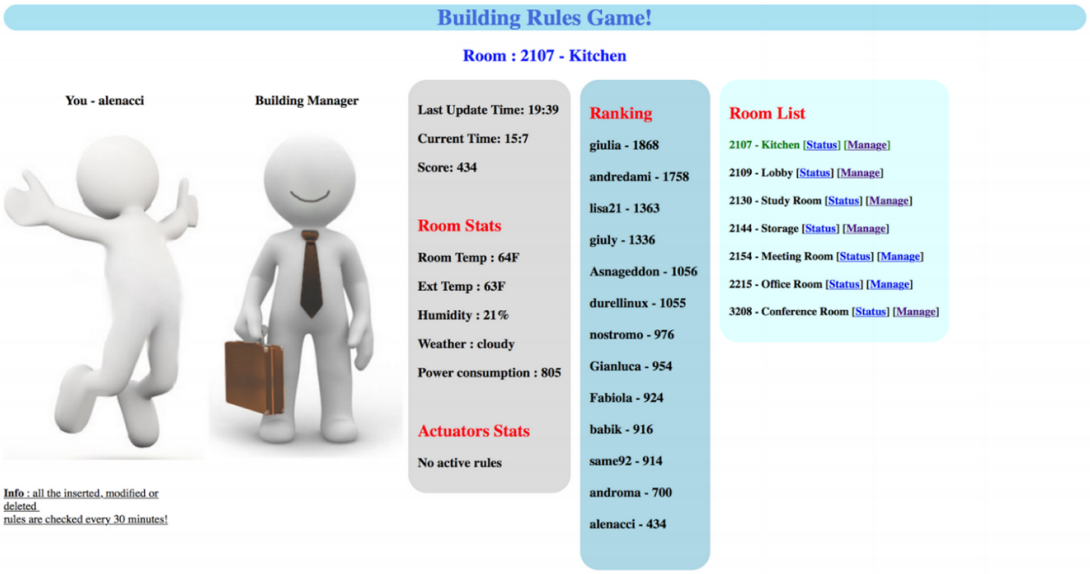

Figure 5.3: BuildingRules Game Interface

5.3 Usability Experimental Campaign

Based on the feedback from the participants of the *baseline* user study, we created a second version of the study to improve the user experience and to help the users in the creation of realistic rules. In fact, in the previous campaign, no feedback about the room changes was provided to the users and, moreover, they had no reason to insert rules that could increase the comfort of the users in the rooms. Therefore, we improved our building simulator to incorporate the effects of actuation. For example, room temperature changes with respect to time, if HVAC is ON the temperature would change until it reaches its set-point, if the HVAC is OFF it would change linearly as a function of difference between indoor and outdoor temperature. We provided suggestions to the users using the Graph Manager to help them understanding the future behavior of the building. We also added power values to different appliances, with fixed values when they are ON. These effects were a representation of what the

user might expect in a real setting.

The users were provided with a *Tamagotchi-like* interface that showed the information about the virtual environment, as can be seen in Figure 5.3. This choice has been made to develop a higher level of empathy between the real users and the virtual ones. In fact, exploiting a psychological phenomenon called *Tamagotchi Effect* [65], which is defined as the development of emotional attachment with machines, robots or software agents, we want to make the users feel personally involved in this experimental campaign. Therefore, each user was assigned an avatar that represented himself in the virtual building. The avatar, in turn, was assigned a happiness index based on comfort, i.e., an occupied room is within temperature and humidity bounds, and lights are ON when it was dark outdoors. In addiction to the users, also the building manager was represented by an avatar, which was assigned a happiness index based on the power consumption of the building. The goal of the participants was to increase the happiness index of both the occupant and building manager avatars. In fact, if the happiness index of the avatars was above a predefined threshold, the users gained points. The user with the highest score at the end of the experimental campaign was awarded $20 Amazon Gift Card.

The experiment was conducted over three days, with 13 users. The users were given a short video tutorial and restricted to create 6 rules on the first day, 4 rules on the second and 2 rules on the final day. Each rule could be edited only twice a day. We assigned two building managers to support the users, as well as monitoring rules being created. The users could see which rooms they were assigned to, the current ranking and the statistics of each room (i.e. room temperature, humidity, etc.). During this experiment 179 rules were created. Their

composition is shown in Figure 5.4. The general behavior of the users was similar to the baseline experiment: the majority of the rules were about occupancy, room temperature, time and weather with respect to room temperature and humidity, lights and windows.

Table 5.1 shows the result of this survey, users assigned higher scores than the baseline case study. Thus, improving the user interface, providing a short preliminary tutorial and giving a runtime support to the users improved our system usability. Moreover, what emerged from this experimental campaign is that the improvement we made, namely giving suggestions, improving the simulator and introducing an interface to give users feedback about the current building status, helped the users to have a clearer understanding of the rooms behavior. In fact, in Table 5.1, the scores that have improved most of all are those that concern the overall system usability (7.2), the insertion of the rule (8.3), and the ease to understand how the combination of rules affected the office (7.2). On the other hand, an unexpected behavior that emerged, is that, even if we focused on trying to make people behave in a way that could increase the overall room comfort without impacting too much on the energy consumption, some users inserted rules with a higher priority than the ones inserted by the other occupants to try to overcome them. To solve this problem, the building managers needed to directly disable the rules that were inserted with this purpose.

In conclusion, from the results that emerged from our campaign, we can state that, on one hand, the philosophy behind BuildingRules 2.0 was appreciated by the users and the system usability had a positive response. On the other hand, there is still need of a building manager to resolve situations like the one

	Date	External Temperature	Occupancy	Room Temperature	Time	Weather
Audio	0	0	2	0	2	0
Coffee Macchine	0	0	3	0	4	0
Computer	1	0	3	0	6	0
Desk Light	0	0	5	0	2	1
Display Monitor	0	0	5	0	2	0
Printer	0	0	4	0	2	0
Projector	0	0	4	0	2	0
Blind	0	0	1	0	0	1
Exhaust Fan	0	0	0	0	0	0
Fume Hoods	0	0	1	0	2	1
HVAC	0	0	2	1	1	0
Room Humidity	0	1	10	0	4	8
Room Temp.	1	3	4	19	2	3
LIGHT	0	0	23	0	14	24
Send Complain	0	0	1	1	0	0
Windows	0	9	4	4	2	13

Figure 5.4: Composition of rules created during the usability experimental campaign

we previously described.

5.4 Integration Experimental Campaign

Since with the previously described experimental campaign we couldn't test how easy was the integration of BuildingRules with the existing BMSes, we installed BuildingRules at the **Joint Open Lab** (JOL) of Telecom Italia in Milan. The JOL did not have enough smart offices and smart spaces to make a usability campaign on a real environment, but it gave us the chance to understand how easy was the integration of the proposed system with an existing BMS. In fact, it took us 3 hours to install the overall system over OpenHAB,

the BMS integrator we described in Chapter 2. OpenHAB is one of the most popular open source BMS and, therefore, represents a valid sample for our experimental campaign.

The software architecture of BuildingRules made the integration straightforward because we needed only to implement the drivers to communicate with OpenHAB. In particular, we locally deployed BuildingRules and we set up 2 rooms, namely the *meeting room* and the *kitchen*. In the *meeting room*, we implemented the drivers to communicate with the smart lights, the Netatmo Weather Station [63] that provided room temperature and humidity, and the occupancy detection made by an Estimote Beacon [64] that communicated with the smartphones of the building occupants, sending distance information to them. Afterwards, we inserted some rules, like *"if someone is in the room then turn on the light"* or *"if time is between 6pm and 8am then turn off the light"*, to verify it was properly working. In the *kitchen*, we implemented the drivers to communicate with the smart lights, the coffee machine that could be only turned ON and OFF and an Estimote Beacon to make occupancy detection. In this case, we inserted rules like *"if nobody is in the room then turn off the coffee machine"*. In Algorithm 5, we can see the pseudocode that communicates to OpenHAB the intention to turn on and off the room light. The only operation that needs to be performed is the dispatch of a HTTP POST request to the light controller endpoint of OpenHAB, setting the body of the message with the action to be executed.

What emerged from the installation of BuildingRules in a real environment is that we managed to fully integrate the system easily without running into particular problems related to communication with the underlying BMS.

Algorithm 5 roomLightActivationDriver Pseudocode

```
1: function ACTUATION(operation)
2:     if operation = LIGHT_ON then
3:         sendPost(endpoint, ON)
4:     else
5:         sendPost(endpoint, OFF)
6:     end if
7: end function
```

5.5 Results Discussion and Limitations

The goal of our thesis work was trying to solve the problems of interaction between the building occupants and the smart buildings. The main problem we observed, with respect to the building programming phase, was that the building occupants did not understand how the building would behave. From Table 5.1, we can see that the goal was reached. In fact, the results we were interested in improving the most, namely *"how easy was to understand how the combination of rules will affect the office"* and *"how easy was it to insert new rules"*, are the ones that actually increased their score the most with respect to the baseline. As a consequence, also the overall *"system usability"* score increased. The *"philosophy behind the system"* and the *"overall impression score"* did not significantly change from the baseline results and, therefore, prove the fact that BuildingRules idea is a good way of programming commercial smart buildings. The previous statement is validated also by the fact that the users found that *"BuildingRules would be useful in their office"*. Instead, with respect to *"how easy was to edit the existing rules"* and *"how easy was to resolve conflicts"*, the score did not improve and confirmed a low score. These scores are justified by the

fact that we did not focus on improving these phases of the user interaction with BuildingRules that remained unchanged from BuildingRules 1.0. Despite the results we obtained achieved our objectives, during our experimental campaign we observed that BuildingRules 2.0 is not enough to manage building occupants that try to overcome the others to assert their preference. For this reason we think that BuildingRules cannot be used only by the building occupants and the presence of a building manager is still needed to orchestrate them. The conclusions and the future works are presented in the next Chapter.

Chapter 6

Conclusions and Future Works

This Chapter presents the conclusion about this thesis work. First, we will synthesize what has been discussed in the previous Chapters, highlighting the results we collected during the experimental campaigns. Finally, analyzing the current limitations, we will provide direction for future works in this research field.

6.1 Conclusions

Technologies that enable building automation have been around for decades but, since they have been expensive and complex, in the last few years the scientific community started to investigate a new paradigm, the *trigger-action programming*. Within this context, we analyzed BuildingRules 1.0, a system that enables the expression of personalized automation rules in commercial buildings using trigger-action programming paradigm. The problem we aimed to solve, as observed in the state of the art, was that the building occupants could not understand clearly the behavior of the building, in particular when the number of trigger-action based rules inserted in the system increased. This thesis

presented a methodology to provide suggestions to the building occupants in a trigger-action based programmable smart building environment. We presented a theoretical framework that generates suggestions to the building occupants to help them understanding the effects of the inserted rules on the smart building. To provide these suggestions, we made analysis on the graphs we introduced to formalize the building state, namely the Building Behavioral Graph and the Building Status Graph. The graphs are generated from each room ruleset making use of a simulation tool we developed. The building simulator is a time-driven tool that, given the specified environment conditions (the room temperature, the weather condition, the occupancy status, etc.), checks which rules are triggered by the system.

Therefore, we improved BuildingRules 1.0 by implementing the aforementioned framework that was then evaluated in two different experimental campaigns:

- in a *virtual environment*, that represents an actual smart commercial building, to evaluate the system usability;
- in an *actual smart building environment*, Joint Open Lab (JOL) of Telecom Italia in Milan, to evaluate the system integration on existing BMSes.

From the *usability experimental campaign*, conducted over three days with 13 users, what emerged is that the improvement we made helped the users to have a clearer understanding of the rooms behavior. With respect to the *integration experimental campaign*, we installed BuildingRules 2.0 over the BMS installed in JOL (OpenHAB) to manage the available smart rooms. The integration of BuildingRules 2.0 with OpenHAB took us only three hours because we needed

to implement the communication drivers.

6.2 Limitations Analysis and Future Works

Although the results presented in Chapter 5 have demonstrated the effectiveness of our approach, there are a number of problems that still remains unresolved.

First, the problem we identified during our experimental campaign (see Chapter 5), namely when the building occupants try to overcome the other asserting their rules in the system, needs to be solved. Therefore, we can think of a rule validation system in which every single rule is evaluated from a building manager before it is actually inserted in the system. An alternative solution could be introducing a Quora-like system [66]. Every user is provided with a predefined number of points that is deducted when a rule is inserted or modified. Each rule can be scored by the other building occupants; if the rule score is positive, the user that inserted that specific rule will gain points, otherwise it will lose them. In this way, if an occupant tries to insert rules that aim at decreasing the comfort of the other occupants, it will lose his points and will not be able to insert rules anymore. However, both the previously mentioned solutions have deficiencies. The first one implies that the building manager is a neutral and honest person, so that he do not filter rules supporting only few building occupants. The second one needs the occupants not to form a coalition against a specific user, making him unfairly losing points.

Second, another problem occurs when we set rules in the system that, once reached a room state, do not allow the room state to change anymore. In fact,

this situation may take place in case these rules are never triggered or in case they always have lower priority than the ones that are currently active. For example, considering the following two rules: *"if time is between 0am and 12pm then close the windows"* with maximum priority and *"if someone is in the room then open the windows"* with a lower priority, the system will always keep the windows closed. This situation may be unwanted by the building occupants and can be detected by making an analysis on the Building Status Graph looking for sinks.

Third, some methods to analyze the stability of the room need to be investigated. It might be useful to analyze what are the conditions in which a room can be considered to be stable. A possible condition would be to have a room which rules are triggered cyclically every day. In this case the building behavior will remain inside a specific *routine*. To detect this *routine* we can look for cycles in the Building Status Graph, so to ensure that the room remains in those states and that cannot reach any other state except in case of need, like a situation in which the occupants' safety may be in danger. To perform this analysis, however, we need to make considerations about the *routine* size. In fact, for example, a cycle composed of a small number of nodes could be an undesirable situation, in which the building state changes too often.

Appendix A

Frontend Views

In this Appendix we will present the images concerning the view tabs that the users can us

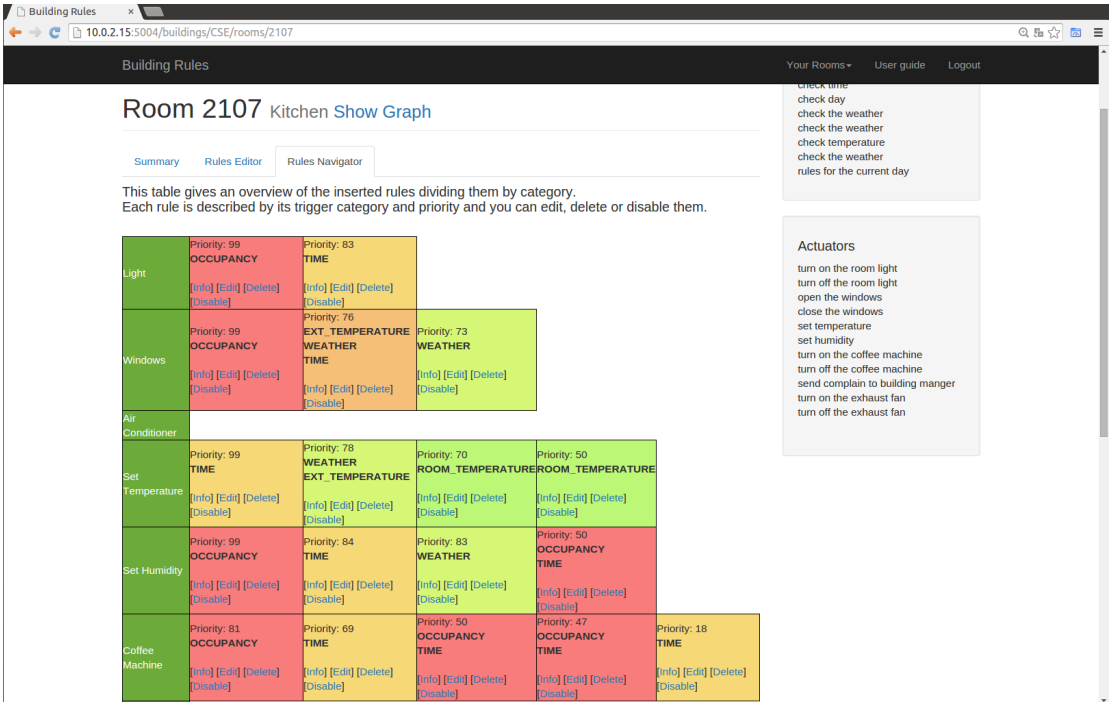

Figure A.1: The Rule Navigator Tab represents the view tab in which the user can see the rules grouped by category.

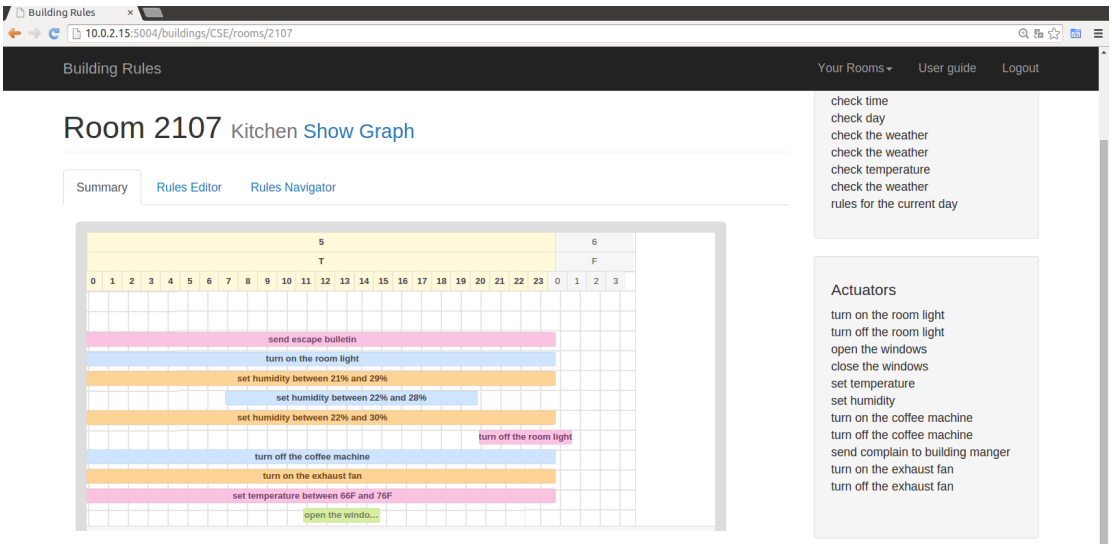

Figure A.2: The Summary Tab represents the view tab in which the user can see a summary of how the building will behave during the current day.

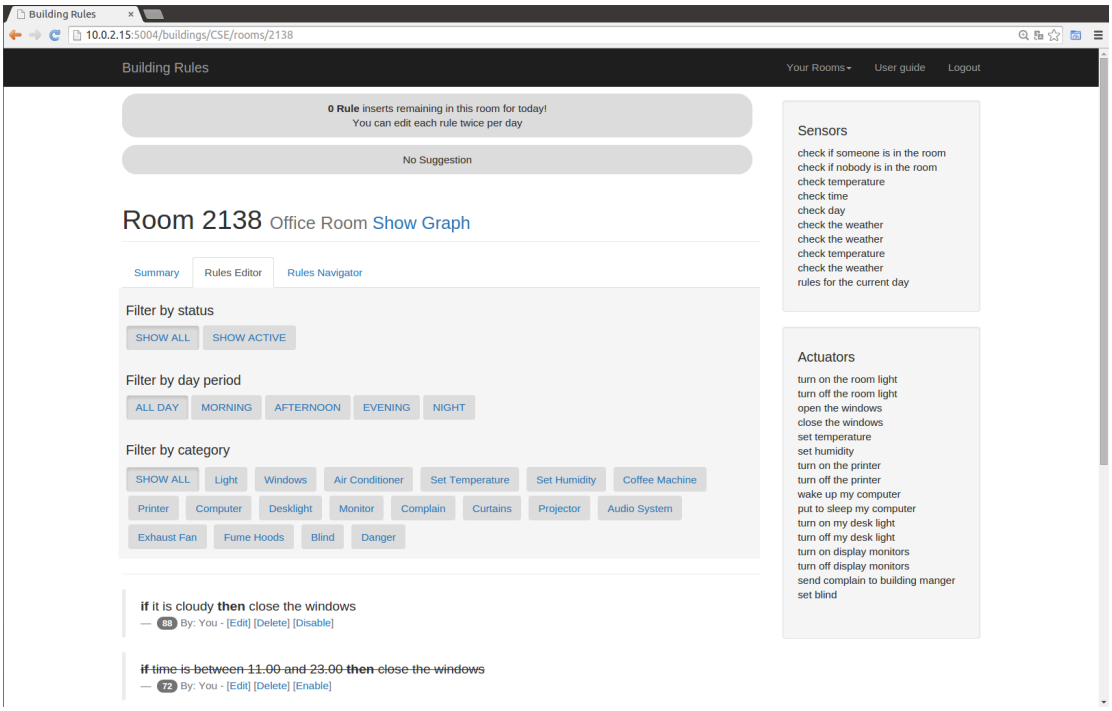

Figure A.3: The Rule Editor Tab represents the view tab in which the user can manage the room rules.

Bibliography

[1] ICT Labs. European Institute for Innovation and Technology ICT Labs. 2014.

[2] Mennicken, Sarah and Vermeulen, Jo and Huang, Elaine M. Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing. 2014.

[3] European Institute for Innovation and Technology ICT Labs.

[4] Nest Learning Thermostat. 2015.

[5] Ecobee. Ecobee Thermostat. 2015.

[6] Honeywell. Lyric Thermostat. 2015.

[7] Philips. Hue Lamp. 2015.

[8] Ninja Blocks. 2015.

[9] Nacci, Alessandro Antonio and Rana, Vijay and Sciuto, Donatella. A Perspective Vision on Complex Residential Building Management Systems. 2014.

[10] Dawson-Haggerty, Stephen and Krioukov, Andrew and Taneja, Jay and Karandikar, Sagar and Fierro, Gabe and Kitaev, Nikita and Culler, David. BOSS: building operating system services. 2013.

[11] US Department of Energy. Buildings Energy Data Book. 2012.

[12] Harle, Robert K and Hopper, Andy. The potential for location-aware power management. 2008.

[13] Timothy Sohn and Anind Dey. icap: an informal tool for interactive prototyping of context-aware applications. In CHI 2003 extended abstracts on Human factors in computing systems, pages 974–975. ACM, 2003.

[14] Khai N Truong, Elaine M Huang, and Gregory D Abowd. Camp: A magnetic poetry interface for end-user programming of capture applications for the home. In UbiComp 2004: Ubiquitous Computing, pages 143-160. Springer, 2004.

[15] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. Practical trigger-action programming in the smart home. Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, 2014.

[16] Scott Davidoff, Min Kyung Lee, John Zimmerman, Anind Dey. Socially-Aware Requirements for a Smart Home. Proceedings of the International Symposium on Intelligent Environments, pages 41–44. 2006.

[17] Jo Vermeulen, Russell Beale. Challenges and Opportunities for Intelligibility and Control in Smart Homes. In CHI 2015 Workshop Smart for Life:

Designing Smart Home Technologies that Evolve with Users', Seoul, Republic of Korea, 2015.

[18] Johnson Controls. http://www.johnsoncontrols.com/content/us/en/products/building efficiency/building management.html.

[19] Siemens Building Technologies. http://www.buildingtechnologies.siemens.com.

[20] SAMAD, Tariq; FRANK, Brian. Leveraging the Web: A universal framework for building automation. In: American Control Conference, 2007. ACC'07. IEEE, 2007. p. 4382-4387.

[21] B. Balaji, J. Xu, A. Nwokafor, R. Gupta, and Y. Agarwal. Sentinel: occupancy based hvac actuation using existing wifi infrastructure within commercial buildings. In Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems. ACM, 2013, p. 17.

[22] A. Beltran, V. L. Erickson, and A. E. Cerpa. Thermosense: Occupancy thermal based sensing for hvac control. In Proceedings of the 5th ACM WorkshoponEmbeddedSystemsForEnergy-EfficientBuildings. ACM, 2013, pp. 1-8.

[23] S. DeBruin, B. Campbell, and P. Dutta. Monjolo: an energy-harvesting energy meter architecture. In Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems. ACM, 2013, p. 18.

[24] B. Roisin, M. Bodart, A. Deneyer, and P. Dherdt. Lighting energy savings in offices using different control systems and their real con- sumption. Energy and Buildings, vol. 40, no. 4, pp. 514-523, 2008.

[25] X. Jiang, S. Dawson-Haggerty, P. Dutta, and D. Culler. Design and implementation of a high-fidelity ac metering network. In Information Processing in Sensor Networks, 2009. IPSN 2009. International Con- ference on. IEEE, 2009, pp. 253-264.

[26] T. Weng, B. Balaji, S. Dutta, R. Gupta, and Y. Agarwal. Managing plug-loads for demand response within buildings. In Proceedings of the Third ACM Workshop on Embedded Sensing Systems for Energy- Efficiency in Buildings. ACM, 2011, pp. 13-18.

[27] Y. Agarwal, R. Gupta, D. Komaki, and T. Weng. Buildingdepot: an extensible and distributed architecture for building data storage, access and sharing. In Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings. ACM, 2012, pp. 64-71.

[28] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. Culler. Boss: building operating system services. In Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2013.

[29] A. K. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-computer interaction, vol. 16, no. 2, pp. 97-166, 2001.

[30] B. P. Haynes. The impact of office comfort on productivity. Journal of Facilities Management, vol. 6, no. 1, pp. 37-51, 2008.

[31] V. L. Erickson and A. E. Cerpa. Thermovote: participatory sensing for efficient building hvac conditioning. In Proceedings of the Fourth ACM

Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings. ACM, 2012, pp. 9-16.

[32] A. Krioukov and D. Culler. Personal building controls. In Proceedings of the 11th international conference on Information Processing in Sensor Networks. ACM, 2012, pp. 157-158.

[33] B. Balaji, H. Teraoka, R. Gupta, and Y. Agarwal. Zonepac: Zonal power estimation and control via hvac metering and occupant feedback. In Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings. ACM, 2013, pp. 1-8.

[34] T. Sohn and A. Dey. iCAP: an informal tool for interactive prototyping of context-aware applications. In CHI'03 extended abstracts on Human factors in computing systems. ACM, 2003, pp. 974-975.

[35] K. N. Truong, E. M. Huang, and G. D. Abowd. CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. In UbiComp 2004: Ubiquitous Computing. Springer, 2004, pp. 143-160.

[36] IFTTT. https://ifttt.com/.

[37] S. T. Bushby. Bacnet: a standard communication infrastructure for intelligent buildings. Automation in Construction, vol. 6, no. 5, pp. 529-540, 1997.

[38] P. Arjunan, N. Batra, H. Choi, A. Singh, P. Singh, and M. B. Srivastava. Sensoract: a privacy and security aware federated middleware for building management. In Proceedings of the Fourth ACM Workshop on Em-

bedded Sensing Systems for Energy-Efficiency in Buildings. ACM, 2012, pp. 80-87.

[39] T. Weng, A. Nwokafor, and Y. Agarwal. Buildingdepot 2.0: An integrated management system for building analysis and control. In Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings. ACM, 2013, pp. 1-8.

[40] S. Resendes, P. Carreira, and A. C. Santos. Conflict detection and resolution in home and building automation systems: a literature review. Journal of Ambient Intelligence and Humanized Computing, pp. 1-17, 2013.

[41] C. Xu and S.-C. Cheung. Inconsistency detection and resolution for context-aware middleware support. ACM SIGSOFT Software Engineering Notes, vol. 30, no. 5, pp. 336-345, 2005.

[42] A. Ranganathan and R. H. Campbell. An infrastructure for context-awareness based on first order logic. Personal and Ubiquitous Computing, vol. 7, no. 6, pp. 353-364, 2003.

[43] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. Software Engineering, IEEE Transactions on, vol. 29, no. 10, pp. 929-945, 2003.

[44] I. Park, D. Lee, and S. J. Hyun. A dynamic context-conflict management scheme for group-aware ubiquitous computing environments. In Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International, vol. 1. IEEE, 2005, pp. 359-364.

[45] Jong-bum Woo, Youn-kyung Lim. User experience in do-it-yourself-style smart homes. In Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing. ACM, New York, NY, USA, 2015, pp. 779-790.

[46] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, volume 4963 of Lecture Notes in Computer Science, pages 337-340. Springer, 2008.

[47] Jeff Kramer and Orit Hazzan. The role of abstraction in software engineering. In Proceedings of the 28th international conference on Software engineering (ICSE '06). ACM, New York, NY, USA, 1017-1018. 2006.

[48] Justin Huang and Maya Cakmak. Supporting mental model accuracy in trigger-action programming. In Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15). ACM, New York, NY, USA, 215-225. 2015

[49] Ernest Friedman-Hill. JESS in Action. Manning Greenwich, CT. 2003

[50] T. Zhang and B. Brugge, Empowering the user to build smart home applications, in International Conference on Smart Home and Health Telematics (ICOST-04, Singapur, 2004, pp. 170-176.

[51] Knibbe, E.J. Building management system, https://www.google.com/patents/US5565855, 1996.

[52] Energy@Home Java Energy ManageMent Application (JEMMA). http://www.energy-home.it/SitePages/Activities/JEMMA.aspx, 2014

[53] Cao, Xianghui, et al. Building-environment control with wireless sensor and actuator networks: Centralized versus distributed. Industrial Electronics, IEEE Transactions on, 2010, 57.11: 3596-3605.

[54] Hosek, Jaromir, et al. Universal smart energy communication platform. In: Intelligent Green Building and Smart Grid (IGBSG), 2014 International Conference on. IEEE, 2014. p. 1-4.

[55] Dey, Anind K., et al. iCAP: Interactive prototyping of context-aware applications. In: Pervasive Computing. Springer Berlin Heidelberg, 2006. p. 254-271.

[56] Dutertre, Bruno; DE MOURA, Leonardo. The yices smt solver. Tool paper at http://yices. csl. sri. com/tool-paper. pdf, 2006, 2.2.

[57] Flask (a python microframework) http://flask.pocoo.org/ [Online; accessed 02/11/2015].

[58] S Massoud Amin and Bruce F Wollenberg. Toward a smart grid: power delivery for the 21st century. Power and Energy Magazine, IEEE, 3(5):34-41, 2005.

[59] https://networkx.github.io/ [Online; accessed 02/11/2015].

[60] http://www.graphviz.org/ [Online; accessed 02/11/2015].

[61] https://www.mysql.it/ [Online; accessed 02/11/2015].

[62] Sarah Mennicken, Jo Vermeulen, and Elaine M. Huang. 2014. From today's augmented houses to tomorrow's smart homes: new directions for home automation research. In Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '14). ACM, New York, NY, USA, 105-115.

[63] https://www.netatmo.com/en-US/product/weather-station [Online; accessed 09/11/2015].

[64] http://estimote.com [Online; accessed 09/11/2015].

[65] Holzinger, Andreas, et al. TRIANGLE: A Multi-Media test-bed for examining incidental learning, motivation and the Tamagotchi-Effect within a Game-Show like Computer Based Learning Module. In: World Conference on Educational Multimedia, Hypermedia and Telecommunications. 2001. p. 766-771.

[66] https://www.quora.com [Online; accessed 15/11/2015].