BuildingRules: A Trigger-Action Based System To Manage Complex Commercial Buildings

Alessandro Nacci[#], Bharathan Balaji[†], Paola Spoletini[*], Vincenzo Rana[#], Rajesh Gupta[†], Donatella Sciuto[†], Yuvraj Agarwal[†‡]

[#]Politecnico di Milano [*]Universita' Insubria
[†]University of California, San Diego [‡]Carnegie Mellon University
[#]{alessandro.nacci, vincenzo.rana, donatella.sciuto}@polimi.it, [†]{bbalaji, rgupta}@cs.ucsd.edu, [*]paola.spoletini@uninsubria.it, [‡]yuvraj.agarwal@cs.cmu.edu

ABSTRACT

Modern Building Management Systems (BMSs) provide limited amount of control to its occupants, and typically allow only the facility manager to set the building policies. In this context, we present BuildingRules, a system which provides an intuitive interface to the occupants of commercial buildings to customize their office spaces using trigger-action programming. BuildingRules automatically detects conflicts among the policies, expressed by the occupants, by using the Z3 SMT solver, and leverages an open source web service BMS (BuildingDepot) to provide access control and actuation services in a building. BuildingRules has been designed to scale for large commercial buildings, as it supports grouping of rooms for ease of policy expression, a scalable backend for resolving conflicts, and a simulator that shows the actuation of rules on a timeline. We tested our system with 23 users across 17 days in a virtual office building, and evaluate the effectiveness and scalability of BuildingRules.

1. INTRODUCTION

Over the years, commercial buildings have evolved to satisfy the different requirements of present day enterprises. Typically, modern buildings have centralized Heating, Ventilation, and Air Conditioning (HVAC) systems in addition to lighting, fire safety, elevators and security systems. While there are numerous Building Management Systems (BMS) [28, 40, 33] to manage and control buildings these have evolved from traditional HVAC controls and do not support emerging smart building applications such as integration with the Smart Grid [4], Microgrid [30], Demand Response [36] and Building Automation [1, 35, 31]. Recently, web service based BMSes have been proposed [2, 5, 14] to better address the challenges and requirements of scalability, maintainability, and easier application development.

BMSes deployed today [28, 40, 33] are designed for building managers and maintenance personnel. Occupants interact with buildings in a limited manner - using thermostats for HVAC control, switches for lights, keys cards for locks and power strips for plug loads. With existing 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. Modern web service based BMSes, and advanced sensor technology, will provide the flexibility to express and implement such applications which can improve occupant comfort and productivity [25] as well as building energy efficiency [21, 29, 6].

While giving occupants the ability to personalize their living environment is indeed promising, there are numerous challenges that must be addressed. First, building occupants do not understand the details of the building infrastructure, and are not necessarily programmers. As shown in prior work [41, 42, 43], 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". Therefore, it is critical that the right level of abstraction and an intuitive user interface is provided by a building automation system to enable occupants with varying levels of expertise to express their preferences [43]. Second, there needs to be the appropriate access control mechanisms when the number of users – i.e. both occupants and building managers – increase to ensure proper building operation. Finally, with multiple users often customizing the same spaces, there needs to be a scalable mechanism to detect and resolve conflicts that will occur. Existing BMSes have limited or no support for such type of access control or conflict resolution.

To address the above challenges, we present the design and the implementation of BuildingRules, a system that allows building occupants to express their automation needs while resolving possible conflicts. BuildingRules is based on the *trigger-action programming* paradigm, under which occupants can express policies using the *"IF something happens THEN do something"* (IFTTT) pattern. Prior work has shown that trigger-action programming is an expressive and intuitive interface to implement building automation policies for people without programming experience [18, 43]. BuildingRules extends the IFTTT abstraction to commercial buildings, and addresses the challenges in integrating the system with our web service BMS [2]. While similar systems have been proposed for *smart homes*, commercial buildings are significantly more complex due to their scale and their shared nature where multiple occupants with different needs inhabit the same space leading to conflicts. To study the extent of conflicts, we conducted a survey with 72 users asking for their preferred rules as applied towards shared office spaces of varying capacity. The survey revealed conflicts in 99% of the cases (details in Section 5). To resolve these conflicts in BuildingRules, we leverage techniques from context aware frameworks to check for conflicts at the moment of the policy expression [47, 34] and use rule priority to resolve conflicts that arise during actuation [37]. Furthermore, we show that BuildingRules is able to keep the latency of conflict detection low enough to ensure good user experience.

In a commercial building, typically facility managers set up au-

tomation policies using the existing BMS, such as the minimum allowable temperature or air flow. It is critical that occupants customizations don't violate these policies. Furthermore, occupants should not be able to control rooms to which they do not have access to. As automated applications such as Demand Response [3] become prevalent, BuildingRules needs to incorporate the policies expressed by them as well. In BuildingRules, we incorporate hierarchical levels of policy expression to address these challenges, and implement BR by extending an open source webservice-based BMS [2, 45]. Since BuildingRules is targeting commercial buildings, we needed it to scale to many hundreds of rooms and thousands of occupants. We achieve this scaling using several design choices. First, BuildingRules supports specifying a rule for the entire building, or a subset of rooms, using a grouping mechanism in combination with conflict resolution that may be required. Second, we have designed the conflict resolution mechanism to be performed in parallel for each room, such that the latency does not increase with the number of rooms.

We evaluate BuildingRules by creating a virtual office environment with 30 rooms, and testing it on 23 users spread across 17 days. We show that the conflict detection latency is 251 ms in the worst case, and 102 ms in the average case. 636 rules were specified during the experiment, and we detected up to 50 conflicts in a day.

2. BACKGROUND AND RELATED WORK

It has been shown that automation reduces time spent by occupants to manage their office environment, such as adjusting temparature and light levels, which in turn improves their productivity [25]. In current buildings with limited occupant control, occupants either request managers to override default settings, or implement adhoc solutions such as space heaters [21] which can lead to energy wastage and deviation from designed operating points [32]. Providing control to the occupants by design, i.e., within the boundaries specified by the building manager, will help reduce energy wastage, while improving the comfort and satisfaction to the occupants [25, 21, 29].

Current sensing technology - occupancy sensors [7, 9], light sensors [16, 39], plug meters [27, 44] - enable automation applications for office buildings. Several context aware frameworks and web service based building management systems have been developed in anticipation of such sensors [2, 14, 17]. Greenberg et al. [24] and Bellotti et al. [8] observe that users need to be an integral part of such systems, as it is not possible to automatically infer their specific wishes with the sensing technology available today. Trigger-action programming has emerged as a promising solution to involve users in home automation, as it provides an expressive and an intuitive interface [41, 42, 43]. Dey et al. show that the IFTTT paradigm expresses 95% of all the applications envisioned by users in smart homes, demonstrating its expressiveness across a wide set of context aware applications [18]. 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 [43]. Finally, **IFTTT.com** is a popular web application that already uses this methodology for connecting various web services and different smart home appliances (e.g., Belkin Wemo) [26]. With BuildingRules, we focus on extending trigger-action programming to provide personalized automation in complex commercial buildings, and address the challenges that emerge when deploying such a system on a large scale. One of the primary concerns that needs to addressed is the resolution of conflicts that arise due to incompatible user requirements.

Existing BMSes already support conflict resolution to some extent. BACnet (Building Automation and Control Networks) protocol is a widely adopted standard by industrial BMSes [10], with support for a priority table for every writable sensor. Conflicts are resolved by assigning priority to applications by their importance. Web service BMSes extend this methodology to include access control and provide more metrics for conflict resolution. SensorAct [5] manages permissions through a combination of priority and guard-rules, a script that specifies validation conditions based on time, date, duration, location and the frequency of operation. BOSS[14] defines every sensor write as a transaction to resolve conflicts. BuildingDepot [45] proposes a combination of priority and lease times at the sensor level. Conflict resolution in these systems is at the sensor level, and does not involve the users by design. Conflicts resolution has been studied extensively in context aware systems [38]. A number of conflict resolution strategies focus on automatically resolving application inconsistencies without involving the user [46, 37]. Systems that involve humans need to abstract information such that users can understand the nature of these conflicts, and can resolve them [18]. CARISMA [11] resolves conflicts among multiple users for a single application with pre-recorded preferences. Park et al. [34] extend the conflict resolution to multiple applications, incorporating user preference and user intent in the application metadata. In BuildingRules, we focus on providing a simple interface, and hence do not collect user preferences or intent. For avoiding conflicts, BuildingRules checks if a proposed rule conflicts with the existing rules, and shows the conflicting rules back to the user so that she can modify them appropriately. The rules are converted to first order logic and checked using an SMT (Satisfiability Modulo Theories) solver [15], similar to the strategy followed by Zhang et al. [47]. Some of the conflicts cannot be detected at the time of rule specification as the conflict is evident only during actuation. Users specify a priority of rules to resolve these *run-time* conflicts, similar to the solution proposed by Gaia [37].

We use Z3 as our SMT solver for resolving conflicts [15]. Park et al. [34] use JESS to build and execute their context-aware sets of rules, and also manage conflicts among rules. Differently from Z3, JESS (Java Expert System Shell) [23] is not an SMT solver, but is a rule engine for the Java platform, which supports the development of rule-based systems that can be tightly coupled to code written entirely in Java. In their system, a context variable can change only in two directions (increase and decrease) and, when rules cause a change in opposite directions to the same variable at the same time, a conflict is detected. In this approach, the conflict is detected only when actuation by a new activated rule is in contrast with the already active rules. Instead, we analyze conflict as soon as a rule is added, even if it is not active yet. This allows us to solve potential conflicts before they actually arise. Moreover, since JESS is not an SMT solver, it needs to be enriched with inference rules to define what is considered as conflict. Thus, every time a rule type or structure is added, new customized inference rules need to be added as well.

Besides the work proposed for smart buildings, all the literature on policy definition and policy conflict management is related to our work. A policy is a rule that enables the execution of an action when some specific event takes place and if a pre-defined condition holds, and for this reason is often in the form event-condition-action and it is called ECA-rule. There are many different policy languages, such as Ponder [13] and PDL [12]. Policy languages have in general a built-in conflict detection and resolution approach and are defined for a specific context. Ponder is a declara-

tive, object-oriented language defined to specify security and management policies. The language allows the specification of both primitive and composite ECA-rules. Analogously, PDL allows the description of ECA-rules that can be translated to Datalog. Both languages are equipped with a priority and grouping mechanism to specify to which group the rules apply and with which priority, and with conflict resolution mechanism based on *meta-policies*, that describe what to do when a conflict happen. The meta-rules define the conflicts and the events that characterized the conflict are monitored. When these events are detected , the resolution actions are performed. Differently from our approach, Ponder and PDL require to define the resolution policies, defining what is considered a conflict. Notice that while we automatically detect conflict as unsatisfiable sets of rules, we can also define additional conflicts by adding rules to the specification that is given as input to the SMT-solver.

3. BUILDINGRULES DESIGN

The goal of BuildingRules is to provide an intuitive interface to the building occupants for expressing customized automation policies for their office spaces, and to use that information to control building subsystems such as HVAC more effectively. In this section we describe the overall design of BuildingRules.

3.1 Rules

As mentioned earlier, similar to prior work [18, 26, 43] we use the trigger-action paradigm to allow users to specify rules in the following format:

$$if\ (something\ happens)\ then\ (do\ something)$$

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, and the combination is a *rule*. For example, "If it is cloudy then turn on lights", where "cloudy weather" is the trigger, and "turn on lights" is the action. Rules are formed using a set of pre-defined triggers and actions.

BuildingRules represents a building with two main entities: the rooms in the building and groups of rooms. Each room is owned by one or more occupants, and the rules are specified at the room level. A room represents a physical space - an office, a conference room, a lobby or a kitchen. Occupants are assigned to these rooms by the building manager, and occupants can customize the behavior of their room by adding new rules. We chose room as the lowest granularity in our implementation, but the representation can be easily extended to be more granular, such as to cubicles or desk spaces.

Ur et al. [43] introduced the concept of *simple* and *complex* rules for smart homes. A simple rule is composed of a single trigger and action, and a complex rule may have 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, we support both simple and complex rules, but the complex rules are restricted to just multiple triggers. Currently, we do not support multiple actions 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. Table 1 summarizes the list of the currently available triggers and actions in BuildingRules.

We support two datatypes for triggers and actions - *boolean* and *integer*. We force a range of values for integer valued triggers instead of comparison against a single value, as it expresses the rule clearly. For instance, a user cannot make a rule: "if it is after 8PM", instead she specifies a time interval: "if it is between 8PM and 10PM". By specifying a time range, a rule has a time validity, and actions which run into perpetuity are avoided. This restriction does not change the expressivness of rules, but forces the user to define both the start and end points of the rule.

Table 1 shows the list of representatives rules supported by BuildingRules. Each rule is assigned to one of several predefined *categories*, based on what they want to control. For example, the two rules "if it is rainy then turn on the light" and "if it is a holiday then turn off the light" are in the same "*Light*" category, while the rule "if it is rainy then close the windows" is in the "*Window*" category. *NO_RULE* is a special trigger available for the building administrators (Rule 7 in Table 1). This trigger is always set to *True* and is used for setting the default conditions of the building. Occupants can override these default rules with more specific rules. BuildingRules also supports external applications through virtual triggers that is controlled via RESTful APIs (Rule 9 in Table 1). We are aware that there are many policy languages, such as PDL [12] and Ponder [13], that can formalize the same information that is provided by the chosen format. However, in our case the format is simple and close to the natural language to be easily understood by the users, and the conflict resolution approached introduced in the next section can be applied to any language that can be encoded in Z3 input language, using a suitable formalization.

3.2 Conflict Resolution

Since users can express their own rules for rooms, some of which are shared by multiple users, conflicts can arise. We define two rules as conflicting when the rules can be in effect at the same time, but the action specified by the rules cannot be satisfied at the same time. If these conflicts are not resolved properly, it can lead to damage of equipment or compromise user comfort. To clarify, consider two users who independently specify the rules: "if time is between 9am and 6pm then turn the HVAC on" and "if time is between 5pm and 8am then turn the HVAC off". Between 5pm and 6pm, the system would be in an inconsistent state. This may cause discomfort to the occupants and could damage HVAC damper if not actuated properly.

To identify the conflicts among rules, we formalize them as propositional formulae and analyze the formalization using the SMT Solver Z3 [15]. A rule is composed of two parts: a (conjunction of) trigger(s), and an action. Before adding a rule, it is verified against the set of rules already active in the room. We represent each rule as a propositional formula composed by an implication (the trigger implies the action) that is satisfied if the trigger is not satisfied, or if both the condition 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 need to be resolved. If a user tries to insert a conflicting rule, the list of the conflicting rules is displayed to the user.

We formalize the rules as propositional formulae compliant with the following grammar:

A `rule` is an implication, where the `action` and `trigger` have

	TYPE	DATA	CATEGORY	EXAMPLE NAME	EXAMPLE HUMAN READABLE SYNTAX	EXAMPLE Z3 SMT TRANSLATION
1	T	BOOLEAN	OCCUPANCY	OCCUPANCY_TRUE	someone is in the room	(inRoom)
2	T	INTEGER	EXT_TEMPERATURE	EXT_TEMPERATURE_RANGE	external temperature is between @val and @val	(and (>= (extTempInRoom) @val) (<= (extTempInRoom) @val))
3	T	INTEGER	TIME	TIME_RANGE	time is between @val and @val	(and (>= (time) @val) (<= (time) @val))
4	T	BOOLEAN	DATE	DATE_RANGE	the date is between @val and @val	(and (>= (day) @val) (<= (day) @val))
5	T	BOOLEAN	WEATHER	SUNNY	it is sunny	(sunny)
6	T	INTEGER	ROOM_TEMPERATURE	ROOM_TEMPERATURE_RANGE	room temperature is between @val and @val	(and (>= (tempInRoom) @val) (<= (tempInRoom) @val))
7	T	BOOLEAN	DEFAULT_STATUS	NO_RULE	no rule specified	(noRule)
8	T	INTEGER	DAY	TODAY	today is @val	(= (today) @val)
9	T	BOOLEAN	EXTERNAL_APP	CALENDAR_MEETING	calendar meeting event	(meetingEvent)
10	A	BOOLEAN	LIGHT	LIGHT_ON	turn on the room light	(light)
11	A	BOOLEAN	WINDOWS	WINDOWS_OPEN	open the windows	(openWindows)
12	A	INTEGER	HVAC	SET_TEMPERATURE	set temperature between @val and @val	(and (>= (tempSetpoint) @val) (<= (tempSetpoint) @val))
13	A	BOOLEAN	APPLIANCES	COFFEE_ON	turn on the coffee machine	(coffee)
14	A	BOOLEAN	MESSAGES	SEND_COMPLAIN	send complain to building manger	(sendComplain)
15	A	BOOLEAN	CURTAINS	CURTAINS_OPEN	open the curtains	(openCurtains)

Table 1: Currently supported rule triggers (T) and actions (A) categories. An example of trigger or action for each category is provided

```
    rule  ::=  trigger ⇒ action
 trigger  ::=  sTrig | sTrig ∧ trigger
  action  ::=  bAct | ¬bAct | iAct∈[n,m]
   sTrig  ::=  bTrig | ¬bTrig | iTrig∈[n,m]
```

a fixed structure. The `trigger` is a conjunction of conditions `sTrig`, that are built on the triggers represented in Table 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 (¬`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 `n`≤`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. We do not allow the use of disjunction in action or trigger, or use of conjunction in the actions. The conjunction in the action (and the disjunction in triggers) is redundant and is equivalent to specifying multiple rules with the same trigger (action) and different actions (triggers).

Note that the disjunction in actions introduces non-deterministic rules, meaning that there is a choice in how the actions can be performed. Currently, we require the user to completely specify the action for a rule using priority. Consider a user who wants to insert a rule "if the room is dark then please 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. We ask the user to insert two rules with different priorities to make her 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 [15]. Translating the formalization of rules into the Z3 language is straightforward since Z3 supports boolean and integer variables. 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)))).
```

We complete the Z3 model with a set of assertions that specify the 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). We then verify the model to check for possible conflicts. The model is verified multiple times by asserting the trigger of each rule in the same category. Thus, we can identify the conflicts related to the same trigger or related triggers. In BuildingRules, the rule verification is performed as soon as a new rule is inserted. 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. We translate the rules from human readable syntax to the Z3 syntax using a pre-defined look up table (see Table 1).

We chose Z3 to detect conflicts as it is efficient and reusable. Although the satisfiability problem is computationally expensive, we ensure low latency as Z3 solves this problem efficiently. An alternative is to have a customized implementation to deal with our particular variables. To assure good performance, we would have to modify and evaluate the algorithm every time we modify the variable domain, the rules structure and other details in the rule set. Instead, the SMT solver just requires 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. Further, the input language of our SMT solver is generic, and it would be easy to switch to different SMT solvers, such as Yices [19].

3.2.1 Run-time Conflicts

Some conflicts cannot be detected using the SMT solver. Consider the following example: 1) If nobody is in the room then turn off the light; 2)If it is between 6pm and 8pm then turn on the light.

Using Z3, we would run the verification twice: once by asserting nobody is in the room and again with the time interval 6pm to 8pm. Both the runs are satisfiable, as it cannot be known apriori if the triggers will conflict in time (see Table 1). We cannot identify the conflict that arises when the room is empty between 6pm and 8pm. In this case, the light will be both on and off at the same time. Note that we need to support similar rules, as the user may want to express a complex policy by combining rules. For example, the user may want a rule that generally turns on the light between 6pm and 8pm, but not when the room is unoccupied.

To resolve these conflicts, we let the user to assign a priority to each rule. If the desire of the user is to set a policy like "generally I want this behavior but not when this event happens", the user sets a lower priority to the general rule, and a higher priority to specific rule. The priority number is used to order the rules by importance and dynamically resolve conflicts during the actuation phase in an efficient way. Moreover, it is simple to explain this concept to the

occupants as they only need to know: "give the more important rules a higher priority".

Listing 1: Rule to actuate extraction pseudo-code

```
triggeredRules  = []
for room in getBuildingRooms(building):
    for rule in getAllRoomRules(room):
        if isTriggered(rule):
            triggeredRules.append(rule)
triggeredRules = orderByPriority(triggeredRules)

alreadyAppliedCategories = []
for rule in triggeredRules:
    ruleCategory = getRuleCategory(rule)
    if ruleCategory not in alreadyAppliedCategories:
        actuate(rule)
        alreadyAppliedCategories.append(ruleCategory)
```

Once a valid ruleset is saved and the desired priority is assigned to each rule, it is possible to apply those rules to the building. Obviously, in a specific instant, only some rules will be applied (the one that are triggered and with the higher priority). Listing 1 shows pseudo-code of the algorithm we designed for rule selection. For each rule in a room, we test if the rule is currently triggered. If the rule is triggered, it is copied into a temporary list(line 8). For every rule category, the rule with the maximum priority for that category is chosen from this list for actuation (line 10).

3.2.2 User Feedback

The output of the Z3 verification phase only specifies that a new rule is conflicting with one of the rules already saved. Next, we need to notify the users the rules that are conflicting, so that they can either edit the existing rule or revise the new rule. We cannot provide the list of rules it is conflicting with easily, as Z3 only detects whether there is a conflict, and does not return which rules are conflicting. One solution is to create a new SMT problem for all possible pairs of rules. However, this approach does not scale - if n is the cardinality of the ruleset, we have to solve n^2 SMT problems. We opted for a less precise but more efficient and feasible solution: when a conflicting rule is detected we inform the user that the possible conflicting rules are all the rules than may be logically conflicting, i.e. all the rules with the same trigger category and same action category. Suppose we have a conflicting rule "If it is cloudy then set temperature between 70F and 75F", we report that rules in the weather category and the HVAC category can be statically conflicting. This strategy works in practice as there are generally few rules which have both the same action and rule category, and the user can easily pick the specific one which is conflicting.

3.3 Users

In a typical commercial building, there are different types of users who may express automation policies. A department chair and the building manager can have over riding control, a lab manager can control her specific lab, while individual users can control their own spaces. In BuildingRules, we need to express this hierarchy since it affects how we resolve conflicts, by assigning *privilege levels* to each user.

Figure 1 provides an example with four categories of users - building managers and standard users, that are suited to represent the actual occupants of the building; and some special users - *applications* and *default* status. *Applications* are external softwares that interact with BuildingRules using RESTful APIs, while *default* represents agents controlling the default status of the building (when users do not specify any rule).

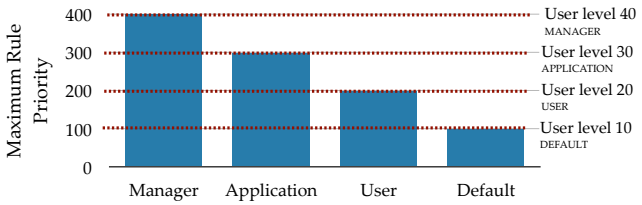

Figure 1: User level and rule priority relation

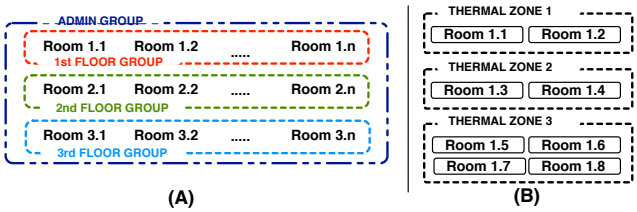

Figure 2: (A) Example building groups (B) Example building thermal zones distribution

User levels are used in two ways. First, it ensures that low level users cannot edit or delete the rules specified by users with higher level. For example, for a room R_1 shared by two users (U_1 and U_2) (level of $U_1 >$ 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. Second, levels also 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 preference in case of runtime conflicts, thereby over-riding rules expressed by a lower-level user.

3.4 Groups and Group Conflicts

In BuildingRules, we provide the facility to create groups of rooms to reuse rules across rooms. Let us suppose an administrator wants to turn off all the building lights at night. Without the group feature, she would have to insert the rule "If time is between 10pm and 7am then turn off the lights" in every room. Using groups, she needs to create the rule just once by specifying it for a group with all the rooms in the building. Figure 2.A shows an example grouping of rooms 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.

We support another class of groups in BuildingRules to incorporate the physical characteristics of commercial buildings. 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 [7, 29]. Figure 2.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 must also be propagated to R_2. We call such groups as *Cross Room Validation Group* (CRVG), and they are specified for action categories which need to follow this property. The behavior of a CRVG is depicted in Figure 3. In a CRVG, all the rules expressed (for specified actions) in one room within the group are propagated to the other rooms.

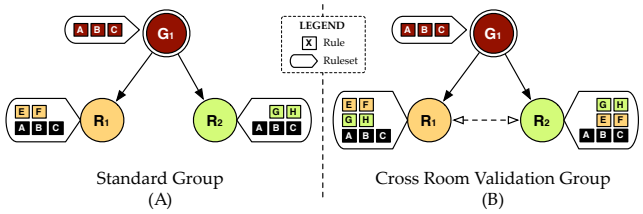

Figure 3: Representation of the two different kinds of supported groups

The conflict checking algorithm needs to take care of which groups a room belongs to. For a room not belonging to **any group**, the rule set is composed of the rules saved for the room. If the room belongs to one or more **standard groups**, the rule set to be checked is composed of the rules saved in the considered room plus the union of all the rules saved in these groups. If the room belongs to a *Cross Room Validation Group*, the rule set of the room is the union of all the rule set (for specified actions in CRVG) of the rooms belonging to that group. Listing 2 presents the pseudo-code that illustrates all the possible cases to generate rule sets for performing static rule verification.

Listing 2: Ruleset generation pseudo-code

```
def getAllGroupRules(g):
    groupRuleSet = getGroupRules(g)
    if isCrossRoomValidation(g):
        for r in getGroupRooms(g):
            groupRuleSet.extend(getRoomRules(r))
    return groupRuleSet

def getAllRoomRules(r):
    ruleSet = getRoomRules(r)
    groups = getRoomGroups(g)
    for g in groups:
        ruleSet.extend(getAllGroupRules(g))
    return ruleSet
```

Conflict detection for a new rule inserted into a group requires more analysis. Let us consider the example of a group G_1 composed of three rooms R_1, R_2, R_3. All the rooms have already been programmed by occupants, so they have their own rule set. At this point, if the building manager inserts a new rule at the group level, and if this rule is conflicting with one of the rules already present in R_1, R_2 or R_3, two problems arise - (1) the rule sets of the rooms are no longer consistent since there are conflicts, and (2) the occupants of the rooms will be never be able to add or modify their conflicting rule set since the administrative rule is at a higher priority.

Thus, when a new rule is inserted or modified in a group, a temporary rule set S_T composed of the union of the rule set of the rooms in the group is created. S_T is checked for conflicts using Z3. If a conflict is found, the building administrator has to remove the conflicting rules for each room before inserting the new one. To perform this operation automatically, we currently use the RESTful APIs exposed by BuildingRules. In our real deployment, we plan to develop a dashboard for the building manager to perform such powerful tasks easily.

4. IMPLEMENTATION

We have designed BuildingRules as a RESTful HTTP/JSON web service, with a ***frontend*** for the user interface, and a ***backend*** which communicates with the BMS, stores necessary information about

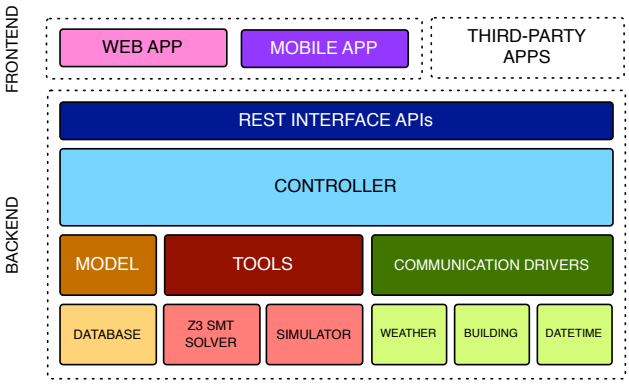

Figure 4: BuildingRules System Architecture

rules, runs the conflict resolution algorithm and provides RESTful APIs for native mobile applications or building management applications. Figure 4 shows the software architecture of the system. We have implemented BuildingRules in Python 2.7 using the Flask framework [22].

The *backend* design follows the Model-View-Controller (MVC) architecture. The REST interface enables communication with the *frontend*, and is managed by a **controller** which implements the application logic. The controller stores the data to manage buildings, rooms, groups, users and rules using the *model* which works as a *database abstraction layer*. The controller also validates building rules using the Z3 SMT solver. Finally, the controller gathers the needed data about weather, building systems and date-time status through a standardized **driver** interface. The drivers allow the controller to read building sensor values and send actuation signals to the building to apply the triggered rules. The driver interface allows BuildingRules to support a variety of web service BMS like BuildingDepot [2], and conform to standards such as oBIX [20]. A **simulator** is also available in order to predict the behavior of the rooms with the specified rules. It is a time-driven framework where for each time step, the simulator reads the specified the environment conditions (the room temperature, the weather condition, the occupancy status, etc.), checks for rules that are triggered, and decides the actions based on priority. The actions are not executed, but are written to a log file. The log file is the automatically converted to a timeline that represents the behavior of the room.

The frontend is a lightweight user interface and interacts with the backend using a well-defined API. Using these APIs, tasks such as user registration, adding triggers and actions, specifying rules for individuals rooms or groups of rooms can all be performed. On top of this API, applications can be implemented that can automatically insert rules. For example, a *Demand Response* application [4] can be implemented using a special user, who injects rules to reduce energy use across all rooms when a pre-registered trigger condition is met. Another example is a *Calendar Manager* that can insert rules automatically, like turn ON/OFF the projector or modify temperature set points based on room schedules.

Drivers in BuildingRules provide necessary abstractions between the core system and low level sensors (reusable across deployments), and are of two types: *TriggerDrivers* and *ActionDrivers*. A *TriggerDriver* takes as input a sensor source (e.g., a temperature sensor in a room) and a condition to verify (e.g., the temperature is set between 70F and 75F). When the above condition is met, it provides a notification through the *eventTriggered* method. An *ActionDriver*

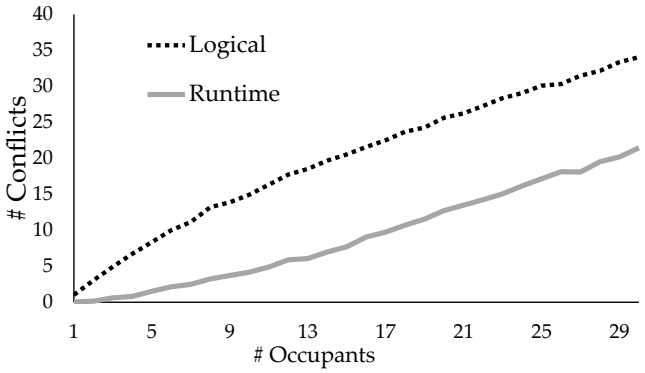

Figure 5: Preliminary survey: number of logical and runtime conflicts detected as participants share office spaces.

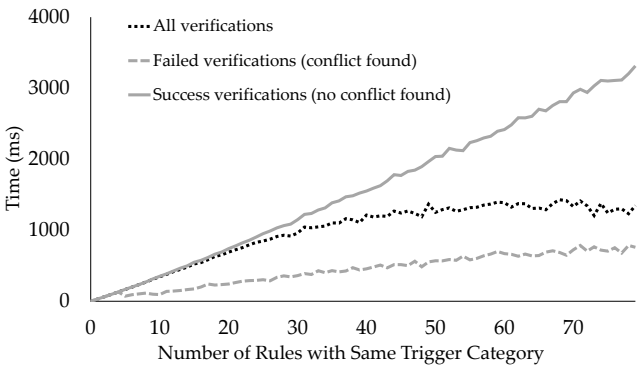

Figure 6: Conflict checking time when the inserted or modified rule. Successful inserts are slower as the rule has been verified against all the rules with the same trigger category. Average detection time is 102ms for a room with 100 rules.

takes a target actuator (e.g. an HVAC control system) and the actual value (e.g. set temperature to 70F) as input, and uses an *actuate* method to execute a rule action. The translation from the human readable form of the rule trigger/action to the input data for the drivers is performed by the controller. This allows to keep the drivers structure simple for developing interfaces to support new environment quickly and with minimum effort.

In our current implementation, four *TriggerDrivers* and one *ActionDriver* are available. The four *TriggerDrivers* are for weather, date/time, external applications and room level sensors. We have implemented a single generic room level *ActionDriver* that can be used for actuation such as changing the temperature in the room. The room level drivers interact with the building using RESTful APIs exposed by the BMS.

5. RESULTS

BuildingRules has been designed for office building occupants to express their preferences using custom automation policies. To evaluate BuildingRules, we first examine the rules expressed by users in a preliminary survey, looking for conflicts and measuring the latency of our conflict detection. Next we ran a larger user study with 23 users using BuildingRules for a week. Using this dataset we analyze the rules expressed, conflicts detected, and finally, how our system can affect the office environment.

5.1 Preliminary Survey

We conducted a preliminary survey to get an understanding of the type of rules that will be generated in an office setting, and how these rules may conflict with each other. The participants were asked to create trigger action rules on a web interface using specified triggers and actions. Only simple rules (no multiple triggers or actions) were requested for this study. The trigger set included *{occupancy, temperature, time, weather}*, and the action set included *{lights, heating, cooling, window, curtains, coffee machine, microwave}*.

We received a total of 72 valid (and 2 invalid) replies with a minimum of 2 rules per participant and a total of 284 rules. The most popular trigger was *occupancy* with 141 rules and the most popular action was *lights* with 100 rules. To analyze the potential conflicts between the rules, we assigned the participants randomly to shared offices (total of 3069 office instances), and ran our conflict detection algorithm. The participants were assigned to offices with capacity of 1 to 30 occupants. We detected conflicts 99% of the time, and there were duplicate rules in 96% of these virtual offices. The conflicts are higher than what would be observed in a BuildingRules installation as the occupants cannot view the rules expressed by others in the same room. However, a significant number of rules will still conflict both statically (at rule specification time) and at run time due to the varying preferences of the occupants as we show in the next section. Figure 5 shows the trends of increase in static and runtime conflicts that occur as more occupants share a single office space.

5.2 Conflict Resolution Latency

Our conflict resolution module checks for conflicts on a per room basis. Each of these checks can be run in parallel, thus making BuildingRules scalable to a large number of rooms in a building. A transaction mechanism is required for handling race conditions when users simultaneously insert rules in the same room or in rooms which belong to the same *cross room validation group*. We have not implemented the transaction mechanism yet as there was a low probability to have simultaneous conflict checking for rooms in the same group for our scale of deployment.

As BuildingRules is an interactive web application, the conflict resolution latency needs to be tolerable to the users. We collected the rules obtained from the preliminary survey, and measured the latency for resolving conflicts as the number of rules in a room increases. Figure 6 shows the latency of conflict resolution when the inserted rules does not conflict with any of the existing rules. Note that, if the rule were conflicting, the latency would decrease as Z3 would return as soon as a conflict is found. Further, the conflict checking only occurs for rules which are in the same rule category, i.e. related actions and triggers. The maximum ratio of the number of rules in the same category to the total number of rules in a room is 6% from the rules collected in our virtual building user study (next section). Thus, we estimate that for a room with 100 rules in place, the average time for conflict checking is 102 ms, and the worst case time is 251 ms.

5.3 Virtual Building Study

To evaluate BuildingRules in a more realistic setting, we created a virtual office environment as depicted in Figure 7. We chose a virtual setting because it is easier to study the rules made using different kinds of sensors that we cannot deploy in a real build-

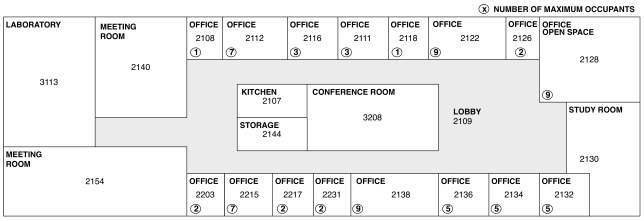

Figure 7: Virtual office plan

	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 8: Rule usage frequency

ing. Actuation of a real building system needs to be done carefully without causing discomfort to occupants or damage to equipment, and we wanted to fully understand the effects of BuildingRules in this pilot experiment before a real deployment. We focus on the analysis of our virtual builing study in this paper, and will address real deployment challenges in future work. Note that, during this experiment the *rule navigator* and *room behaviour* tabs were not available. We added these two UI elements after feedback from participants in this experiment.

We have designed the building plan to be representative of a typical office, and incorporated different types of rooms such as conference rooms, research laboratory, kitchen, storage and offices with varying capacity. 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.

We obtained a total of 636 rules from this study, and there is an average of 15 rules per room, and 16 rules per user. Figure 8 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). Users also created some unique rules we did not anticipate, like the rule "if someone is in room and time is between 10pm and 6am then send a complaint", using the "send complaint" as an alarm.

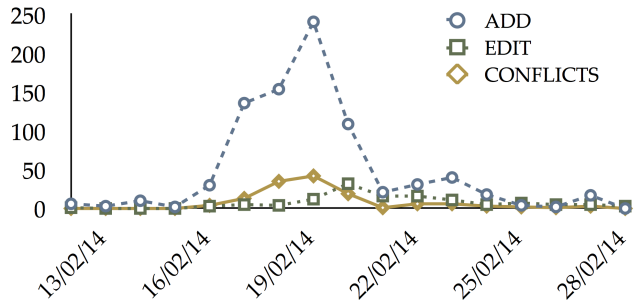

Figure 9: User requests

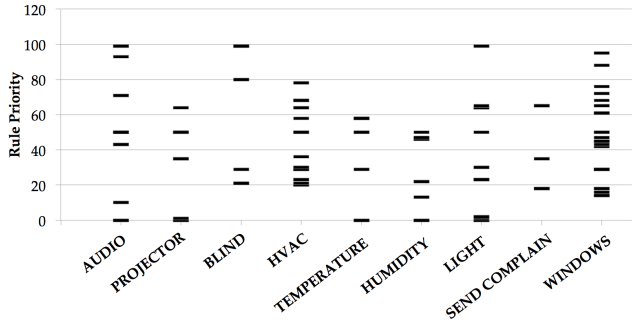

Figure 10: Priority of rules in the room with the highest number of rules (64). Every action has multiple rules associated with it, and triggered rule with highest priority is actuated.

Figure 9 shows the addition of rules across the period of the study. Understandably, the number of logical conflicts have decreased compared to the preliminary survey as the users consider which rules to add after looking at the current rules in effect. However, there is still a significant number of conflicts as some of these logical conflicts are not intuitive.

BuildingRules associates a priority with each rule to maintain the privilege level of the users and allow them to express complex policies using an interplay of rules that would conflict at runtime (Conflict Resolution Section). In order to verify if the users exploited this feature, we present the rules inserted in a room with the highest number of rules in Figure 10.

We show the effect of the combination of rules on a room using BuildingRules simulator. The simulator results are based on real weather data and temperature readings we obtain from the BMS, and we simulate occupancy using data collected in prior work [7].

The study was designed in such a way that participants were forced 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.

6. CONCLUSION

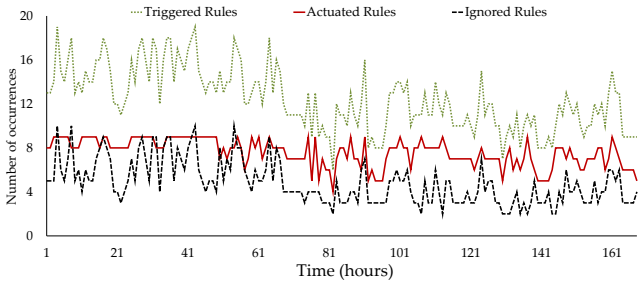

Figure 11: Runtime conflicts on a room across one week based on BuildingRules simulator. Significant amount of run-time conflicts occur due to multiple rules present per action as shown in Figure 10.

We have presented the design and the implementation of BuildingRules, a system that enables expression of personalized automation rules in commercial buildings using the well known trigger-action programming paradigm, which can then be integrated with existing Building Management Systems (BMSes) to actuate buildings. We show that when multiple users express different policies for the same physical space conflicts can occur. To resolve these conflicts, we have implemented two mechanisms in BuildingRules. First, we avoid logical conficts by detecting them as rules are inserted using the Z3 SMT solver. Second, BuildingRules resolves run time conflicts using a priority assigned to individual rules. We show that our conflict detection algorithm is parallelizable and scales to large commercial buildings, such that the latency is low enough to support the interactive web application UI of BuildingRules. To ease rule expression and expose the physical constraints imposed by building systems, BuildingRules provides a grouping mechanism. To incorporate the hierarchy commonly seen in commercial buildings, BuildingRules provide mechanisms for access control and different levels of privileges for rule expression. The final component of BuildingRules is an intuitive web interface for building occupants to express their rules. Using this UI, we evaluated the use of BuildingRules in a virtual office building with 23 users across 17 days, and found that BuildingRules allows expression of a wide set of rules, resolves conflicts effectively.

There are however several aspects of the system that can be improved. A redesigned UI will be needed to provide a good overview, especially for building managers who have to potentially manage all the rules expressed in a building (for them, for instance, a special UI is needed in order to perform the rule editing over multiple rooms at the same time). Furthermore, the UI can be improved further to show existing rules and possible conflicts as users are typing their rules. We leave this limitation to future work although one of the ideas we are pursuing is to use a building level simulator to display the final effect of all the rules in one view. We also plan to increase the number of actions (e.g. email notification) and triggers (e.g. type of room, occupant identity) supported by BuildingRules assuming the underlying sensors are available, as well as allow nesting of rules (may require incorporating temporal logic in our conflict detection algorithm). We also plan to explore how rules can be migrated across buildings. Most importantly, we plan to evaluate BuildingRules in a real building. Unfortunately, this can only be done at a small constrained environment, where occupants can express their desires easily. It would be interesting to study the rules expressed, especially their evolution as occupants get direct feedback about the effect of those rules on their surroundings. Finally, we plan to study the group dynamics in shared spaces when multiple occupants are expressing rules that conflict with other occupants rules.

7. REFERENCES

[1] Y. Agarwal, B. Balaji, S. Dutta, R. K. Gupta, and T. Weng. Duty-cycling buildings aggressively: The next frontier in hvac control. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 246–257. IEEE, 2011.

[2] 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*, pages 64–71. ACM, 2012.

[3] O. Alliance. Openadr 2.0 profile specification, a profile.

[4] S. M. Amin and B. F. Wollenberg. Toward a smart grid: power delivery for the 21st century. *Power and Energy Magazine, IEEE*, 3(5):34–41, 2005.

[5] 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 Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 80–87. ACM, 2012.

[6] 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*, pages 1–8. ACM, 2013.

[7] 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*, page 17. ACM, 2013.

[8] V. Bellotti and K. Edwards. Intelligibility and accountability: human considerations in context-aware systems. *Human–Computer Interaction*, 16(2-4):193–212, 2001.

[9] A. Beltran, V. L. Erickson, and A. E. Cerpa. Thermosense: Occupancy thermal based sensing for hvac control. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, pages 1–8. ACM, 2013.

[10] S. T. Bushby. Bacnet< sup> tm</sup>: a standard communication infrastructure for intelligent buildings. *Automation in Construction*, 6(5):529–540, 1997.

[11] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *Software Engineering, IEEE Transactions on*, 29(10):929–945, 2003.

[12] J. Chomicki, J. Lobo, and S. A. Naqvi. A logic programming approach to conflict resolution in policy management. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *KR*, pages 121–132. Morgan Kaufmann, 2000.

[13] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, POLICY '01, pages 18–38. Springer-Verlag, 2001.

[14] 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.

[15] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[16] 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*, page 18. ACM, 2013.

[17] 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*, 16(2):97–166, 2001.

[18] A. K. Dey, T. Sohn, S. Streng, and J. Kodama. icap: Interactive prototyping of context-aware applications. In *Pervasive Computing*, pages 254–271. Springer, 2006.

[19] B. Dutertre and L. De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006.

[20] P. Ehrlich and T. Considine. Open building information exchange (obix) version 1.0. oasis committee specification, december 2006.

[21] 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*, pages 9–16. ACM, 2012.

[22] Flask Web Microframework. `http://flask.pocoo.org/`.

[23] E. Friedman-Hill. *JESS in Action*. Manning Greenwich, CT, 2003.

[24] S. Greenberg. Context as a dynamic construct. *Human-Computer Interaction*, 16(2):257–268, 2001.

[25] B. P. Haynes. The impact of office comfort on productivity. *Journal of Facilities Management*, 6(1):37–51, 2008.

[26] IFTTT. `https://ifttt.com/`.

[27] 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 Conference on*, pages 253–264. IEEE, 2009.

[28] Johnson Controls. `http://www.johnsoncontrols.com/content/us/en/products/building_efficiency/building_management.html`.

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

[30] R. H. Lasseter and P. Paigi. Microgrid: a conceptual solution. In *Power Electronics Specialists Conference, 2004. PESC 04. 2004 IEEE 35th Annual*, volume 6, pages 4285–4290. IEEE, 2004.

[31] A. Majumdar, D. H. Albonesi, and P. Bose. Energy-aware meeting scheduling algorithms for smart buildings. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 161–168. ACM, 2012.

[32] E. Mills. Building commissioning: a golden opportunity for reducing energy costs and greenhouse gas emissions in the united states. *Energy Efficiency*, 4(2):145–173, 2011.

[33] Niagara AX. `http://www.www.niagaraax.com`.

[34] 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*, volume 1, pages 359–364. IEEE, 2005.

[35] M. Pichler, A. Dröscher, H. Schranzhofer, G. Kontes, G. Giannakis, E. Kosmatopoulos, and D. Rovas. Simulation-assisted building energy performance improvement using sensible control decisions. In *Proceedings of the Third ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 61–66. ACM, 2011.

[36] F. Rahimi and A. Ipakchi. Demand response as a market resource under the smart grid paradigm. *Smart Grid, IEEE Transactions on*, 1(1):82–88, 2010.

[37] A. Ranganathan and R. H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing*, 7(6):353–364, 2003.

[38] 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*, pages 1–17, 2013.

[39] B. Roisin, M. Bodart, A. Deneyer, and P. DŠherdt. Lighting energy savings in offices using different control systems and their real consumption. *Energy and Buildings*, 40(4):514–523, 2008.

[40] Siemens Building Technologies. `http://www.buildingtechnologies.siemens.com,`.

[41] 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*, pages 974–975. ACM, 2003.

[42] 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*, pages 143–160. Springer, 2004.

[43] B. Ur, E. McManus, M. P. Y. Ho, and M. L. Littman. Practical trigger-action programming in the smart home. *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, 2014.

[44] 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*, pages 13–18. ACM, 2011.

[45] 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*, pages 1–8. ACM, 2013.

[46] C. Xu and S.-C. Cheung. Inconsistency detection and resolution for context-aware middleware support. *ACM SIGSOFT Software Engineering Notes*, 30(5):336–345, 2005.

[47] T. Zhang and B. Brügge. Empowering the user to build smart home applications. In *International Conference on Smart Home and Health Telematics (ICOST'04, Singapur*, pages 170–176, 2004.