Dynamic Constraint Satisfaction with Space Reduction in Smart Environments

VIKTORIYA DEGELER*
Distributed Systems Group, Johann Bernoulli Institute, University of Groningen, The Netherlands
vdegeler@gmail.com

ALEXANDER LAZOVIK
Distributed Systems Group, Johann Bernoulli Institute, University of Groningen, The Netherlands
a.lazovik@rug.nl

A scalable, reactive and easy to evolve reasoning mechanism is essential for the success of automated smart environments, augmented with a large number of sensors and actuators. While constraint satisfaction problem (CSP) model is applicable for modelling decision making in such environments, the straightforward representation of the model as a CSP leads to a great number of excessive calculations. In this paper, we propose a method of modelling the task as a Dynamic CSP in a way that avoids unnecessary recalculations with new events in the environment. We present a Dependency Graph data structure, which not only allows to reduce CSP search space for every consecutive sensor event by detecting only affected parts of the environment, but also allows to give enough information to users of the system to specify the exact reasons of system's decisions, even with a large number of constraints. We formally prove that partial recalculation of affected parts still keeps the full environment globally satisfied and globally optimal. The evaluation of the system in the living lab showed real-time responses for all events. Additional simulated performance experiments showed that the Dependency Graph approach consistently outperforms the straightforward CSP representation. The experiments also showed that the clusterization of the environment has a noticeable effect on the performance, with highly clusterized environments requiring less computations.

1. Introduction

A smart office is a building that integrates and controls different appliances (PCs, lights, thermostats) to improve the productivity and comfort of its occupants. What even several years ago seemed to be a distant future, now quickly becomes a reality: modern office buildings contain thousands of smart devices that can be remotely controlled and potentially adjusted in accordance with dynamic context information received from the installed sensors. While one of the goals of such systems is to improve the quality of occupants' working conditions, the main driving factor for the building facility managers is to make the building more sustainable. Unfortunately, current building management systems often fail to reduce and optimize the unnec-

*Current affiliation: Airbus Group Innovation Works, Newport, UK

essary energy consumption while maintaining (and improving) the user comfort in the same time. This is caused by the difficulty of coping with the dynamic changes caused by the user's interaction with the environment, changing user preferences (often depending on the current work-related goals), and the dynamic environment itself (with new devices and sensors being installed and/or replaced). At the same time, in any smart environment the autonomy and reasoning power should be counterbalanced by the ability of users to fully understand the reasons of the system's automated operations. Users should have the ability to fully control the system's decisions, and be able to adapt the system's reasoning to fit their goals and desires at any moment of time. Therefore, flexible and adaptable reasoning mechanisms are essential for environments automation.

Modern building automation systems typically represent the decision logic via hard-coded rules, where each concrete sensor input corresponds to the output that is hard-wired with a set of actuators. For example, a signal from the presence sensor may result in turning on the light in the room, while no signal for a certain period of time will result in turning the light off. However, this approach has a number of drawbacks. Firstly, with an increasing amount of devices it is very hard to maintain the integrity of the rules. Secondly, as the rules are often hard-coded and depend on concrete sensor and device installations, it is difficult to maintain and upgrade the system, not mentioning aiming at energy saving and sustainability.

As a logical step, instead of hard-coded if-then rules, it is desirable to model the system behavior via a set of generic rules (or constraints) that represent possible global system states and then allow the system to pick one of the possible solutions satisfying given constraints. The selected solution may be then chosen based on some optimization criteria, thus forming an optimization problem that can be informally defined as follows: *given sensor inputs (representing the current state of the system), a set of constraints (representing user comfort preferences), and a desired optimization criteria (e.g., total energy consumption), modify the state of the controllable actuators to satisfy the given constraints with the best value of the cost function associated with the given optimization criteria.*

Our approach, implemented in the GreenerBuildings[a] European FP7 project,[1] is to specify scenarios of the building's operations via sets of logical rules. The predefined sets of rules for standard behavior may always be modified or fully overridden on global or local levels by facility managers or particular users. The rules combine context information about the environment with the desired behavior of actuators, and they must at all times be satisfied whenever it is possible or be able to communicate failure to relevant users when impossible.

This behavior can be represented as a constraint satisfaction problem (CSP) model. However, the resulting optimization problem is computationally complex given the expected number of variables (up to several thousands for modern buildings), which makes it unfeasible to apply as-is for a real-time building optimization.

[a]http://www.greenerbuildings.eu/

The model though can be represented as Dynamic CSP (DCSP) due to the necessity to solve the problem over and over again, every time with small changes (due to changing environmental context) from the previous task.[2] If costs are involved, e.g. the desire to find the most energy efficient way to satisfy current set of rules, the usual CSP task may need to be solved as an optimization CSP.

In this paper we present an approach based on the dynamic constraint satisfaction that has been implemented within the GreenerBuildings project. The contributions of this paper are several. First of all, we explain why the straightforward encoding of the problem to the (D)CSP task does not bring the maximum efficiency, and how the specific structure of the smart environments domain can be exploited in order to make CSP models smaller and every subsequent CSP solution recheck only parts of the environment. In particular, the existence of context variables (information from sensors) and controllable actuators as well as the uneven dependency of variables are exploited. By uneven dependency we mean the existence of highly dependent subsets of variables (for example, devices that are part of a common area within a single room) with many interconnecting rules, which have very loose or no dependency on another subset of variables (e.g. devices from a different room).

The main contribution involves the formulation of the dependency graph data structure, which makes it possible to split CSP into dynamically independent subtasks, and to find only the affected parts of the problem every time a new event arrives to the system, which severely reduces the size and complexity of the CSP to be solved at every subsequent step. We also present specific transformation of rules into a form which makes the dependency graph easy to calculate. It is important to note though that splitting the original CSP into independent subtasks during the modelling process is not possible to the full extent, as any part of the CSP may potentially depend on any variable or rule thus preventing the possibility to decompose the problem in advance. However, we demonstrate that given a CSP problem with a structure typical for office automation, it is often possible to decompose it into a set of dynamically forming independent subtasks. Additionally, we show that by doing this it is possible to give specific feedback to people that manage the environment about the exact causes of certain actions, or parts of the rules that are inconsistent.

Following our initial publications on this topic,[3,4] in this paper we formally define a context-aware rule maintenance problem as DCSP, prove the correctness of the solution, and present the implementation of the system and its evaluation in the living lab and with performance experiments.

The remainder of the paper is organized as follows. Section 2 presents the related work. Section 3 provides an initial description of the system and defines the structure of rules. Section 4 formalizes the environment. Section 5 presents the transformation of rules. Section 6 presents the Dependency Graph data structure that is used to reduce the constraint problem space. The feedback mechanism of the Dependency Graph is explained in Section 7. Section 8 evaluates the solution. Finally, Section 9

provides conclusions.

2. Related work

2.1. *Constraint Satisfaction in Smart Environments*

In the field of smart environments several studies propose to use constraint satisfaction techniques to solve reasoning problems. For example, multi-agent coordination in smart homes is modelled as distributed constraint optimization problem by Pecora and Cesta.[5] The coordination is fully distributed, i.e. every agent relies only on communication with other agents and manages one or more variables. In this scenario constraints model the desired minimum-cost concurrent behavior of agents.

On the other hand, Petersen et al. propose a solution with centralized command post for mission-critical environments, such as search and rescue operations with robot teams.[6] They present a method for efficient task-assignment, where constraints that are added by humans via a dedicated interface are combined with physical constraints of the environment, and are solved by a modern Mixed Integer Linear Programming (MILP) solver.

Koes et al. present a first order logic constraint language for such search and rescue domains for robots.[7] They also introduce a goal-oriented Constraint Optimization Coordination Architecture (COCOA), which aims to transform the original problem by formulating it as a constraint optimization problem. The solution to this problem will generate a schedule that can be executed by a robot with some level of abstraction.

Cesta et al. describe a problem solving environment that deals with complex scheduling problems, which are represented as constraints.[8] They present O-OSCAR, a CSP-based object-oriented scheduling framework.

CSP-based AI planner is used by Kaldeli et al. to compose services for smart home scenarios.[9,10] The planner allows the expression of extended goals and utilizes the latest advancements in the CSP field to make the search faster using enhanced inference techniques.

2.2. *Dynamic Constraint Satisfaction*

The formulation of the dynamic constraint satisfaction problem (DCSP) as a set of successive static CSPs with addition or removal of constraints was first proposed by Dechter and Dechter,[11] and subsequently elaborated in many other works.

Bessiere investigated the application of arc-consistency algorithms for Dynamic CSPs.[12] Bessiere considers binary constraints, i.e. those that involve only two variables. The original arc-consistency algorithms do not solve static CSP completely, but eliminate all values that are mutually inconsistent and are definitely not part of the solution, but that otherwise would be discovered by backtracking procedures over and over.[13,14] For the Dynamic CSP the original arc-consistency cannot be reused if a constraint is relaxed, because the reasons for marking values inconsistent are not being tracked, and inconsistency marking cannot be removed without

fully rerunning the algorithm. Bessiere therefore proposed an extension to the algorithm to track the original reasons for marking inconsistent values, which allows to make incremental changes to arc-consistency values when constraints are changed. An improved algorithm with lower space complexity is proposed in [15].

Algorithms based on *nogood* recording that may be used in both static and dynamic CSPs are proposed by Schiex and Verfaillie.[16,2] Similarly to arc-consistency, a *nogood* is a pair of value assignments that cannot be contained in any solution of the CSP. A set of nogoods is built during a backtrack search.

Roos, Ran and van den Herik propose an algorithm to solve each CSP in a sequence of consecutive CSPs by using previous solutions.[17] To avoid big differences in successive solutions, which are often undesirable in practice, they propose a repair-based algorithm RB-AC, which performs a local search in the neighborhood of an infringed solution to find a new nearby solution which is the most similar to the old one. The results however show that repairing a solution may be much harder than creating a new one from scratch if several constraints are changed simultaneously. Therefore authors later proposed an approximate algorithm that reduces the time complexity of a repair by relaxing the optimality requirement with respect to number of changes made.[18]

An alternative definition of Dynamic Constraint Satisfaction Problem was formulated by Mittal and Falkenhainer, where DCSP defines a single CSP with different additional sets of variables and constraints depending on variable values.[19] In this paper we use a more commonly accepted Dechter and Dechter's definition, when referring to the DCSP.[11] Verfaillie and Jussien present a comprehensive survey of the DCSP related research.[20]

3. Rule Satisfaction in Smart Environments

Smart homes, and in general other types of smart environments, can be defined by several important characteristics. The most important is undoubtedly the ability to be context-aware, to sense the physical surroundings and to understand the context of the current situation. Also, smart environments should be able to reason using this information and to deduce valuable knowledge. And finally, they should have the ability to act intelligently in response to changing situations, according to certain goal criteria. Smart environments are often ubiquitous, which means their sensing and acting capabilities come from devices that are embedded in the physical world.

There are several criteria, according to which the intelligence of smart environments can be judged. Most smart environments are designed to increase the comfort and quality of life of their users, e.g. inhabitants of a building. The automation of surrounding devices usually goes towards this goal, for example by understanding current user goals and problems and performing actions directed towards solving them. In most of the cases, however, this should not lead to situations where users are unable to override the system's decisions, as this not only severely decreases their comfort levels, but also can be dangerous in some unaccounted for situations.

It is important as well that users can anticipate to a certain extent the actions of the smart environment and understand why certain actions were performed. Seemingly erratic and illogical actions lead to lesser trust of users to the system and its decreased usage. Therefore the ability of users to control the smart environment and to influence its reasoning in a particular way is also an important criterion. Many smart environments are designed particularly to help elderly or disabled people, thus supporting a healthy ageing process. And, of course, increasing energy prices and adoption of renewable energy sources bring forth the topic of energy awareness and energy savings in smart environments.

The GreenerBuildings project aims to increase the overall users' comfort by adapting to their needs. Usually there are different ways to satisfy user requirements, so the additional goal of the project is to assure the minimum energy consumption of the building without sacrificing user comfort. The reasoning is handled by the Rule Maintenance Engine (RME) component. Main RME goals can be defined as follows:

Given a set of user-defined rules of the building's behavior and information about the current environment state, the Rule Maintenance Engine must ensure that:

(1) The rules are satisfied and adhered to, whenever it is possible.
(2) If there are rules which cannot be satisfied at a given moment, users must be presented with sufficient information to identify the cause.
(3) For any actuator at any given moment users must be given information on exact reasons of performed actuations. Users must be able to override the state of the actuator.
(4) While satisfying all rules, the energy consumption of the building should be minimal.
(5) Decisions should be made in real-time and be scalable with respect to the environment size.

In general, rules are entered to the system by its users. However, there are certain "ready-made" presets of rules that users may use. The system gives the ability to modify sets of rules or switch between different sets.

Rules describe the expected and desired behavior of the smart building. In general there are two different types of rules. The RME system itself handles those types equivalently, but for the users of the system they represent a difference between what is *necessary* and what is *desirable*.

The first type represents a *dependency between variables*. For example, a rule $desk1.monitor = active \Rightarrow desk1.pc = on$ tells the system that it is not possible to have a monitor in an active state if the PC, to which the monitor is connected, is off. The second example is $\neg(room1.blinds1 = down \wedge room1.window1 = open)$, which represents a physical constraint that blinds can only be put into down position if the window is closed.

The rules of the second type are in essence *user preferences*. They describe the desired behavior of the system. For example, a rule $room1.presence > 0 \Rightarrow$

$room1.ceilinglamp = on \vee room1.desklamp = on$ represents a desire to have a light on in the room, if there are people inside.

The rules are defined as formulas in a predicate logic over finite domains. Every atomic predicate represents a certain condition over a variable, and should result in $true$ or $false$. There are several available operations in predicates. The equality represents that a variable should be equal to a given value for a predicate to be true. For example: $room313.dimmer1 = 0$. Opposite to it, the inequation is used to forbid a variable to be equal to a certain value, e.g. $room313.dimmer1 \neq 0$. It is also possible to use a set of values instead of a single value in both cases, e.g. $room313.dimmer1 \in \{0; 10; 20\}$ or $room313.dimmer1 \notin \{0; 10; 20\}$. These operations are available for all types of variables. For ranged variables, i.e. integer or real ones, it is also possible to use inequalities, i.e. greater (or equal) / less (or equal) than. For example: $room313.dimmer1 > 50$; $room313.dimmer2 \leq 200$. To summarize, the rule with only a single atomic predicate is represented as:

$$P ::= (v_i = d) \mid (v_i \neq d) \mid (v_i \in \{d_i\}) \mid (v_i \notin \{d_i\})$$
$$P ::= (v_i < d) \mid (v_i > d) \mid (v_i \leq d) \mid (v_i \geq d),\ v_i \in \mathbb{R}$$

Of course, atomic predicates can be combined together to form logical formulas of any additional complexity, using the standard logical operators:

$$R ::= P \mid \neg R \mid R \wedge R \mid R \vee R \mid R \Rightarrow R \mid R \Leftrightarrow R$$

4. Environment Definition as CSP

The environment $\langle V, D \rangle$ is defined by a set of context variables $V = S \bigcup A$; $S \bigcap A = \emptyset$, where $S = \{s_1, s_2, ..., s_n\}$ is a set of uncontrollable variables, and $A = \{a_1, a_2, ..., a_m\}$ is a set of controllable variables. Uncontrollable variables S represent sensors, they provide information about the environment, and cannot be directly influenced by the system. They do not necessarily represent a physical sensor. A variable can represent a combined value of several sensors, or a result of a certain activity recognition task.

On the other hand, controllable variables A can be seen as actuators that can act in the environment in an automated way, i.e. by receiving appropriate commands from the system. We assume that it is possible to change the state of every actuator independently from other actuators, and that it is possible to transform an actuator from any state of its domain to any other state of its domain.

Every variable $v \in V$ varies over a finite *states domain* $d(v)$ with size k_v, which can be either a range of integer or real values, a boolean, or a set $d(v) = \{d_{v1}, d_{v2}, ..., d_{vk_v}\}$. Each variable v has a *cost function* $c_v(d_i)$, associated with its state domain $d(v)$ that shows the cost of keeping the variable in this state. For the GreenerBuildings project the cost is associated with the energy consumption of corresponding devices.

The original set of rules R_o contains a set of logical formulas over variables in V. Every rule $r \in R_o$ can be represented as a constraint to the classical CSP

model, which corresponds to a subset of variables $V_r = \{v_{r1}, v_{r2}, \ldots\}$, and represents a subset X_r of a Cartesian product over their respective domain values $d(v_{r1}) \times d(v_{r2}) \times \ldots$, which specifies the sets of values of those variables that are compatible with each other. This subset can be trivially constructed by constructing the full truth table for a set of variables V_r, and retaining only those values from a table, for which the rule evaluates to *true*.

It is possible to use the original set of rules R_o as a set of constraints to the CSP task, though we also need to add the knowledge about the current sensor values to the problem definition, since we know their values from the context environment information, and we cannot influence them directly. For every sensor $s \in S$, if its current value is d_s, one more rule $s = d_s$ is added to restrict the sensor. In this case, the natural constraint satisfaction problem for the smart environment will be defined as follows:

Find a valuation for a set of variables $V = S \bigcup A$ which satisfies all constraints $C = R_o \bigcup R_s$, where R_o is the original set of predefined rules, and R_s is a set of sensor constraints for every sensor: $\forall s \in S : s = d_s$, where d_s is the current sensor value of the sensor s, obtained from the context information. We will refer to this CSP definition as $CSP(V, R_o \bigcup R_s)$.

Such CSP representation, however, is very inefficient in practice with respect to the amount of required computations. The reasons for this are the following:

- In order to keep the solution valid and up to date, the CSP task should be solved for every new sensor change event. For the smart buildings with hundreds of sensors several of such events arrive every second. Solving the CSP for the full environment is a computationally heavy task, and doing it for every new sensor change event can represent a big strain on resources. Such solution has a very low scalability potential.
- Every sensor change affects only a small part of the environment, therefore solving from scratch every time produces a large amount of duplicate work. Using dynamic constraint satisfaction techniques is more computationally efficient.
- In practice, many rules (constraints) for intelligent environments are only applicable for a particular situation, which may occur only a small percentage of the time. For most of the time the constraints will not be applicable, however they will still need to be added as a part of the CSP over and over again.

The classic definition of Dynamic Constraint Satisfaction Problem [11,12] defines it as a set of successive CSPs, where every next CSP is created from the previous one by adding or removing a variable or a constraint. Though in our case most of the changes to the environment do not involve direct addition or removal neither of a variable, nor of a constraint, we can still represent a problem in such a way, by representing a change of a sensor value $s \in S$ from d^s_{old} to d^s_{new} as removal of a constraint $s = d^s_{old}$ and addition of a constraint $s = d^s_{new}$.

In the classic definition the domain remains of the same size. On the other hand,

our solution for dynamic constraint satisfaction of smart environments allows to make the problem domain smaller for every subsequent CSP, by reusing dynamically independent parts of the previous problem.

5. Rule Transformations

Users may enter rules in any form they like, but to make the automated processing easier, the rules are transformed into a special uniform shape. Transformations are done only once at the time of addition of a new rule by users (or after a rule was modified), and should ensure that the least amount of processing is kept for the real-time system's operation. There are two reasons for transformations.

First of all, we split the rule into as many independent sub-rules as possible. For example, a rule $chair = occupied \Rightarrow pc = on \wedge lamp = on$ should be split into two different rules: $chair = occupied \Rightarrow pc = on$ and $chair = occupied \Rightarrow lamp = on$. This will not change the overall rule satisfaction logic, as all the rules should be satisfied, however, such splitting ensures that we do not register a false dependency between two variables "*pc*" and "*lamp*", as it can be seen that, at least if using only this rule, they may be satisfied or not satisfied independently.

The second reason is that at the end we want all resulting rules to have a form $F_s(S) \Rightarrow F_a(A)$, i.e. some function of sensors implies a function of actuators. The benefits we achieve with this are twofold. First of all, the sensors S cannot be influenced by the system, thus they represent the situation that is given to us. There is no possibility to directly influence the antecedent of the equation $F_s(S)$; with the given context in the current situation it is either satisfied or not. If it is not satisfied, or let us rather say "the situation described in $F_s(S)$ does not occur", then we do not need to do anything about the consequent of the equation, the $F_a(A)$, which contains actuators, as the full equation is already satisfied. The rule is then in the "inactive" state, i.e. it is possible to skip it in the constraint satisfaction problem, which can help us to severely reduce the search space and decrease dependencies. If, on the other hand, the $F_s(S)$ is met, i.e. results to *true*, then we must ensure that the consequent, which contains actuators $F_a(A)$, is satisfied. Thus the second benefit. Since we can only control actuators, only actuator variables are meaningful for the CSP search space. When we use such a form, we can only put $F_a(A)$ part of the formula to the CSP description, and only when we actually need it to be satisfied.

Finally, to ensure the fastest processing the functions $F_s(S)$ and $F_a(A)$ are transformed into the form $\bigwedge_s(P(S)) \Rightarrow \bigvee_a(P(A))$. Here $P(S)$ and $P(A)$ are atomic predicates with respective variables. The form $\bigwedge_s(P(S))$ ensures that with every new sensor reading $s = d_s$ it is possible to recheck only a single atomic predicate $P(s)$. The form $\bigvee_a(P(A))$ is the easiest for CSP solvers to work with.

It is always possible to transform any human-defined rule into such a form. The actual transformation is done in the following steps. First of all, the original rule is transformed into the CNF form. Every conjuncted clause (the disjunction) in the

CNF form is connected by $\wedge$-clause and, since all rules must be satisfied, may be regarded individually. Therefore every such clause will represent a single separate rule in the resulting set, so often an original rule will result in several final rules. Every resulting rule is a disjunction of atomic predicates (possibly negated). On the second step it is transformed into an implication by taking those atomic predicates that contain only sensors, and putting them (in negated form) into the antecedent of the implication. This is possible due to a fact that for all predicate logic formulas the equivalence $a \Rightarrow b \equiv \neg a \vee b$ holds. The next step is not mandatory, and is done only for convenience, in order to unify further representation and processing of transformed rules: negation is removed from all negated atomic predicates by flipping the operation. For example, the $\neg(room1.dimmer1 > 100)$ becomes $room1.dimmer1 \leq 100$, and $\neg(desk1.pc = on)$ becomes $desk1.pc \neq on$.

The overview algorithm of the transformation is presented in Algorithm 1. The $parseToCNF$ transformation is a standard one. For completeness, we present it in Appendix of this paper. Note, that we do add the operation flipping for atomic predicates, which is not the part of the standard CNF transformation.

Algorithm 1 Rule Transformation

```
1: function transformRules (R_orig)
2: R ← ∅
3: for all r_orig ← R_orig do
4:   r_cnf ← parseToCNF(r_orig)
5:   while r ← getNextDisjunction(r_cnf) do
6:     P_s ← ∅; P_a ← ∅
7:     while p ← getNextAtomicPredicate(r) do
8:       if p.v ∈ S then
9:         P_s ← {P_s; flipOperation(p)}
10:      else
11:        P_a ← {P_a; p}
12:      end if
13:    end while
14:    R ← {R; ⋀_s(P_s) ⇒ ⋁_a(P_a)}
15:  end while
16: end for
17: return R
```

For example, let us assume we have a rule that requires to have light in the room if there are people inside. Light can be achieved either by turning on the lamp, or by opening the blinds, but only in case there is enough light outside:

$$room1.presence > 0 \;\Rightarrow\; room1.lamp = on \vee outsidelux > 1000 \;\wedge\; room1.blinds = open \tag{1}$$

Sensors here are $room1.presence$ and $outsidelux$. So, by putting it into CNF, splitting it into two distinct rules, putting the sensors to the antecedent, and removing the negation from atomic predicates we obtain the following two rules:

$$room1.presence > 0 \;\Rightarrow\; room1.lamp = on \;\vee\; room1.blinds = open \tag{2}$$

$$room1.presence > 0 \;\wedge\; outsidelux \leq 1000 \Rightarrow\; room1.lamp = on \tag{3}$$

If someone is present in the room, the first rule will be "active" and the system will need to either turn on the lamp or open the blinds. In practice, since optimization CSP is used, if both choices are not restricted the system will choose to open the blinds as the most energy efficient choice. But if the outside light level is insufficient, the second rule will also become active, which means the only choice left will be to turn on the lamp, as it will satisfy both rules.

6. Dynamic Dependency Graph

The environment size for pervasive smart buildings may become considerably large, easily reaching hundreds of variables. One of the goals of the RME component is to ensure that such environments can be handled in real-time, thus rechecking all variables after every event registered by one of the sensors is definitely a non-practical solution.

It is better to recheck only parts of the environment, which are actually affected by a change. This is, however, not always a straightforward task. Dependencies are introduced via rules, but it is not enough to recheck all the rules that contain the changed sensor to find a new optimal state of the environment, as easily shown by the earlier example that requires either the lamp to be on or the blinds to be open during the day, if people are inside the room. Let us assume that rules (2) and (3) compose our ruleset. When it is still dark, someone enters the room, so sensor values are $room1.presence > 0$ and $outsidelux = 500$. The only way to satisfy both rules is to turn on the lamp, so the system does it, while keeping the blinds shut. Now the outside light gradually increases, and at some point becomes bigger than our threshold: $outsidelux = 1100$. At this moment only the sensor from the rule (3) is changed, and the rule is not active anymore. But because of this change the first rule (2) can now be satisfied in a different way, by opening the blinds, which is more energy efficient, so it also needs to be rechecked.

Another option is to transitively consider all variables affected, if they are a part of the affected rules. However, this will largely overestimate the amount of rules and variables to be rechecked. For example, assume we have a rule $(desk1.chair = occupied \wedge desk1.paperwork = true) \Rightarrow desk1.lamp = on$, and the chair becomes occupied, while the paperwork does not change and remains false. In this case the total antecedent of the rule has not changed, it is still not satisfied, thus we should not even trigger the rechecking of the lamp and all other variables, which may be dependent on it.

The dynamic dependency mechanism, which is realised via the use of the Dependency Graph, is specifically the mechanism designed to keep track of the actual dependencies between the variables, based on the context information, and only invoke re-optimization tasks for the smallest subsets of the variables which are actually affected.

As shown in Section 5, after performing rule transformations, we obtain an internal set of rules R, where every rule $r \in R$ is in a form $F_s(S) \Rightarrow F_a(A)$, specifically $\bigwedge_{s \in S}(P(s)) \Rightarrow \bigvee_{a \in A}(P(a))$.

First of all, sensor variables should be removed from the CSP model. At every moment in time sensor variables have a particular valuation, based on the context environment information, and represented by a set of rules R_s: $s = d_s$, $\forall s \in S$. Therefore, while the sensor values influence the valuation of actuators, the sensors themselves are not *decision variables*, as only a single value is applicable to them, and we know this value in advance.

The rule form $F_s(S) \Rightarrow F_a(A)$ helps to construct an equivalent CSP model that does not contain sensor variables. For this, we define an *active* property of rules:

Definition 6.1. Active/inactive rule. A rule $r = (F_s^r(S) \Rightarrow F_a^r(A))$ is *active* in the current state of the environment, i.e. with a given valuation of sensors R_s, if the antecedent part of the rule $F_a^r(A)$ valuates to *true*, and *inactive* otherwise.

Let $R^* \subseteq R$ represent an active subset of rules R.

If the rule is inactive, it poses no constraint for the actuator values, as the full rule is already satisfied regardless of them. So the rule may be removed from the CSP model at this moment in time. The activeness of a rule changes with time and different sensor values.

Using the notion of rule activeness, we change the previous CSP definition:

$$CSP(V, R_o \bigcup R_s) \equiv CSP(A, F_A^R),$$

$$\text{where } F_A^R = \{F_a^r(A)\},\ \forall F_a^r(A) \text{ of } r \in R^*$$

Not only such definition removes all sensor variables from every consecutive CSP task, but also many original rules are removed, leaving only those that are actually relevant to the current situation and state of the environment. Given the nature of smart environment rules, it is usually a small subset of the original rules at any moment in time.

The next step in transforming the task definition is to find sets of dependent variables. For this, we formally define dependency of variables and rules:

Let $X(V_x)$ represent the set of *full Cartesian product of values* for a variable set V_x: $d(v_{x1}) \times d(v_{x2}) \times \ldots$.

Let $r(x)$ for $r \in R$ and $x \in X(A)$ identify the result of evaluation ($true$ or $false$) of the consequent actuator part $F_a(A)$ of a rule r with actuator values in valuation x.

If $B_r = \{a_{r1}, a_{r2}, \ldots\}$ is a subset of actuators $B_r \subseteq A$, then let NB_r be a complement subset: $NB_r = A \backslash B_r$.

Definition 6.2. Dependency. The rule $r \in R$ is said to introduce a *dependency* over a subset of actuator variables $B_r = \{a_{r1}, a_{r2}, \ldots\}$ (or, alternatively, a rule r depends on variables $a_{r1}, a_{r2}, \ldots$), iff:

(1) $\forall x \in X(NB_r) : \exists w_1, w_2 \in X(B_r)$ s.t.: $r(x \times w_1) \neq r(x \times w_2)$
(2) $\nexists a_{nb} \in NB_r$ s.t. $\exists d_1, d_2 \in d(a_{nb})$, $\forall x \in X(NB_r \backslash a_{nb})$, $\forall w \in X(B_r) :\ r(d_1 \times w \times x) \neq r(d_2 \times w \times x)$
(3) $\nexists a_b \in B_r$ s.t. $\forall d_1, d_2 \in d(a_b)$, $\forall w \in X(B_r \backslash a_b)$, $\forall x \in X(NB_r)$: $r(d_1 \times w \times x) = r(d_2 \times w \times x)$

The first part ensures that the result of a rule evaluation will indeed change with different valuations of variables in B_r.

The second part ensures that the set B_r is *complete*, i.e. there is no variable outside of this set, s.t. changing a value of this variable will still result in a change of a rule evaluation result.

The third part ensures that the set B_r is *minimal*, i.e. there is no variable in this set, which does not influence the evaluation result irrespectively of its value.

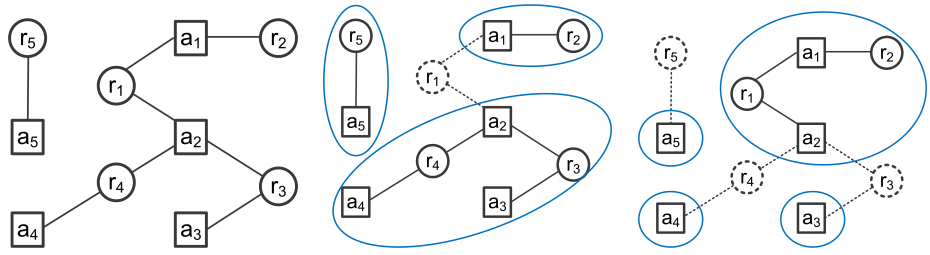
We use the dependency relation to find subsets of dependent variables and rules. To do it, we introduce a *dependency graph*:

Definition 6.3. Dependency graph. The dependency graph for a set of actuators A and a ruleset R is a bipartite graph $G = \langle A, R, E \rangle$, where A and R are two sets of vertices, and $E \subseteq A \times R$ is a set of edges, $(a, r) \in E$ iff the consequent part $F_a(A)$ of the rule r depends on a.

Figure 1a shows a dependency graph example. Two disconnected subgraphs in the figure represent a *static independency*, i.e. there is no rule that may potentially make the variables from different connected subgraphs dependent on each other. Every rule and every variable are a part of only a single subgraph, and it is clear (see Lemma 6.1 for proof) that instead of having a single big CSP with all variables and rules combined, it is possible to "divide and conquer" by creating several smaller CSPs for every independent subgraph.

But most of the division benefits are gained not from static, but from *dynamic independency*, which exists when there are no *active* rules that make the variables mutually dependent. This dependency changes over time and with different sensor values, so two variables may be dynamically dependent at one moment, and independent at the next one.

Definition 6.4. Active subgraph. At a certain moment in time, an *active subgraph* of the dependency graph G is a connected subgraph of G that consists only of active vertices.



(a) All rules are active (b) r_1 is inactive (c) r_3–r_5 are inactive

Fig. 1. Dependency graphs

Examples of active subgraphs are shown in Figures 1b and 1c. By using this notion, we show that our solution to the DCSP of smart environments is globally optimal even with partial environment rechecking. First, we prove a lemma that smaller-sized CSPs for connected active subgraphs can be solved independently. Then, we prove the main theorem, stating that at every subsequent step it is possible to only recheck those subgraphs that changed their structure.

Lemma 6.1. *For any set of solutions $x_i \in X(A_i)$ for all connected active subgraphs $G_i \subset G$, $G_i = \langle A_i, R_i^*, E\rangle$ s.t. $\bigcup_i(G_i) = G$, $\bigcup_i(A_i) = A$, $\bigcup_i(R_i) = R^*$, their combination $x = \bigcup_i(x_i)$ is a solution of the full $CSP(A, F_A^R)$, $\forall r \in R^*$. And vice versa, if $x \in X(A)$ is a solution to the full CSP, when split into subsets of variables per active subgraph, these values will be a solution to smaller CSPs for connected active subgraphs: $CSP(A, F_A^R) \equiv \bigcup_i CSP(A_i, F_{A_i}^{R_i})$*

Proof. We split the proof into two parts. First we prove that if $x \in X(A)$ is a solution to $CSP(A, F_A^R)$, then all x_i which are parts of the x that contain variables from active subgraphs G_i, are solutions to respective $CSP(A_i, F_{A_i}^{R_i})$. Then we prove that if $\forall i$: x_i is a solution to the $CSP(A_i, F_{A_i}^{R_i})$ of the subgraph G_i, then $x = \bigcup_i(x_i)$ is a solution to the full $CSP(A, F_A^R)$.

1. Assume $x \in X(A)$ is a solution to $CSP(A, F_A^R)$. Then x must also be a solution for a $CSP(A, F_A^{R_i})$, $\forall i$, since these CSPs contain the same set of variables A, but only a subset of original constraints $R_i \subseteq R^*$, therefore are less restrictive. From Definition 6.2 and Definition 6.4 it follows that the satisfaction of constraints R_i from active subgraph G_i depends only on variables from subset A_i, irrespectively of values of variables $A \backslash A_i$, thus the rules R_i are satisfied by valuation of $x_i \in X(A_i)$, $x_i \subseteq x$, therefore the smaller $CSP(A_i, F_{A_i}^{R_i})$ must also be satisfied $\forall i$.

2. Assume that $\forall i$: $x_i \in X(A_i)$ is a solution to $CSP(A_i, F_{A_i}^{R_i})$. If we add new variables $A \backslash A_i$ (to the total set of A) for every such CSP to obtain $CSP(A, F_A^{R_i})$, it will be satisfied for any valuation of new variables, since by Definitions 6.2 and 6.4 no constraint out of R_i changes its satisfaction status no matter the values of $a \in A \backslash A_i$. Therefore we can use valuation $x = x_1 \times x_2 \times \ldots$ to satisfy all $CSP(A, F_A^{R_i})$. So, the valuation x satisfies all rules in every set R_i. Therefore it must satisfy all rules in a combined set $R^* = \bigcup_i R_i$, $\forall i$, which means the valuation x must be a solution

to the $CSP(A, F_A^R)$. □

Since every cost function only depends on a single variable, if x is optimal for $CSP(A, F_A^R)$, all $x_i \subseteq x$ must also be optimal for the respective smaller CSPs. Otherwise, if a valuation x_i' is better for $CSP(A_i, F_{A_i}^{R_i})$, following the chain of reasoning from part 2 of the proof, we arrive to conclusion that valuation $x' = x \backslash x_i \cup x_i'$ must also be a solution to $CSP(A, F_A^R)$, and it must be better than x, which contradicts the premise. And vice versa, if all independent subsets x_i are optimal, the full set x must also be optimal.

During the operation of the smart environment system, new sensor readings arrive as events. The change in a sensor value may potentially cause some rules to change their activeness status. The check takes constant time for every rule, as the form $\bigwedge_{s \in S}(P(s)) \Rightarrow \bigvee_{a \in A}(P(a))$ ensures that only a single atomic predicate $P(s)$ for a sensor s may change, and needs rechecking. Only the change in activeness status affects the actuators, and only a small percentage of new sensor readings actually change the activeness of a rule, which saves the system from many unnecessary CSP solution invocations.

The change of activeness status changes the structure of active subgraphs around the rule. Either a single subgraph has one more (one less) constraint, or two or more subgraphs may join into one (one subgraph split into two or more).

We now prove the main theorem for DCSP in smart environments:

Theorem 6.1. For every event in the system, only active subgraphs that changed their structure must be rechecked for the whole valuation of actuators to remain satisfied and optimal.

Proof. Let x^t be the optimal solution found for the $CSP(A, F_A^{R^t})$ at time t, with active rules R^t. Let G^t represent a set of active subgraphs G_i^t at time t. Let x^{t+1}, $CSP(A, F_A^{R^{t+1}})$, R^{t+1}, G^{t+1} represent same notions for the time $t+1$.

We split x^t to a set of valuations $\{x_i^t\}$ that correspond to active subgraphs G_i^t. As proven in Lemma 6.1, every x_i^t is a solution to a corresponding $CSP(A_i, F_{A_i}^{R^t})$.

Let a sensor change at time $t+1$ make ruleset R_-^{t+1} inactive and ruleset R_+^{t+1} active. The total active ruleset at time $t+1$ is thus $R^{t+1} = R^t \backslash R_-^{t+1} \cup R_+^{t+1}$, and the rules $R_{const} = R^t \backslash R_-^{t+1}$ are active at both times t and $t+1$.

Since variable vertices are always active, active subgraphs that consist only of rules in R_{const} are defined by G_{const} and are the same for both times: $\forall G_i$ s.t. $R_i \subseteq R_{const}$: $G_i^t = \langle A_i, R_i, E \rangle = G_i^{t+1}$. Since we know that x_i^t is a solution for G_i^t, it must also be a solution for G_i^{t+1}. Let us denote the set of valuations for G_{const} as x_{const}.

Let x_{nc}^{t+1} represent (newly found) solutions for all active subgraphs $G^{t+1} \backslash G_{const}$. As proven in Lemma 6.1, the combined $x^{t+1} = x_{const} \times x_{nc}^{t+1}$ must be a solution for the $CSP(A, F_A^{R^{t+1}})$. Therefore it is proven that it is possible to reuse solutions x_{const} from G_{const} in a global solution. □

In the case of a usual CSP instead of an optimization CSP, i.e. if the cost of the solution is not relevant and any solution that satisfies the rules is equally good, the dynamic rechecking can be made even smaller, as the rechecking will be required only if the constraint is added (i.e. the rule becomes active), but not if the rule becomes inactive, as in this case the previous solution is still valid.

Algorithm 2 Event processing

```
1: function processChange (s,d_s)
2: for all rule ← R s.t. rule.F_s contains s do
3:    Update rule status
4:    Mark rule as changed if status is changed
5: end for
6: while ∃rule ∈ R s.t. rule is changed do
7:    subGraphs ← findActiveSubGraphs(rule)
8:    for all sg ← subGraphs do
9:       newstate ← OptimizeCSP(sg)
10:      for all v_c ∈ sg.vars s.t. v_c ≠ newstate.v_c do
11:         createAction(v_c,newstate.v_c)
12:      end for
13:      Unmark changed status from all r ∈ sg.rules
14:   end for
15:   Unmark changed status from rule, if still marked
16: end while
```

Algorithm 2 presents the reaction of the system to a new sensor reading $s = d_s$. For all rules from a ruleset R that depend on s, the system checks the status of the rule (*active* vs. *inactive*), and if the status is changed, the rule is marked accordingly.

While a *changed* rule r exists, the system finds a set of adjacent active subgraphs for this rule. If rule changed to inactive there may be more than one. The optimization CSP is invoked for every such subgraph. For all actuators that changed their state an action is created and is sent further to be executed. Finally, the system removes *changed* status from all rules in checked subgraphs and the original rule.

7. Explaining the Actuation

It is essential for general acceptance of any smart environment system that people remain in full control of it. It is important for them to be able to override the decisions of the system, but it is equally important to be able to understand why the system decided to perform the actuations that it had performed. Such understanding not only gives insights to the reasoning of the system, but also allows to tweak its behavior with minimum changes, if the behavior is not the expected one or the desired one.

A typical smart reasoning system as the one described in this paper may have a large amount of rules added over time. Moreover, every particular actuator may be referenced by many different rules that are often entered to the system by different people. With increasing amount of rules, it becomes increasingly hard to keep them fully consistent. However, while it is desirable to keep the system consistent at all times, strictly rejecting the rules that may *potentially* cause an inconsistency is very often impractical if the rules are unlikely to collide. For example, assume that two people, Alice and Bob, share a room at work, but they work in shifts, so that their working schedules never overlap. Alice works during the day, and she prefers not to use artificial lighting, therefore she adds the rule $Alice.location = room1 \Rightarrow room1.lamp = off$. Note that the rule is very simple, and does not include exceptions for many intricate situations, yet it works perfectly fine for Alice's needs. On the other hand, Bob works in this room in the evening, and he always needs an artificial light, so he adds the rule $Bob.location = room1 \Rightarrow room1.lamp = on$. As long as these two rules are part of the system, there is a potential unsatisfiability for the room lamp status. Indeed, on a very rare occasion Alice and Bob may happen to be in the room at the same time, which will cause the system to issue a warning about the unsatisfiable status of the lamp. Even if Alice and Bob never work at the same time, the system should have this fact explicitly stated as additional constraint, in order not to detect potential inconsistency. One of possible solutions would be to change the second rule to $Bob.location = room1 \wedge \neg Alice.location = room1 \Rightarrow room1.lamp = on$, yet it makes the rule more complex, and requires Bob to think in advance about potential situations and implications on other rules without his direct involvement, which is an unreasonable requirement for an average user of the system.

For the reference implementation of the Rule Maintenance Engine we opted to allow potential inconsistencies in the specified rules. As long as all rules can be satisfied given the current situation of sensors, the system works as planned. However, if there is a conflict, which cannot be resolved given the current state of sensors, it is logged and a corresponding warning is issued to a facility manager or some other person in charge to resolve the conflict. During the conflicting state, no actuations are performed on the actuators that are part of the inconsistency. When asking a person to resolve an inconsistency, it is very important to pinpoint exact rules or parts of the rules that caused it. Due to potentially huge amount of rules, checking manually all rules that contain the affected actuator (in our example these are all rules that contain $room1.lamp$), as well as those that transitively contain other dependent actuators, will require to check many irrelevant rules.

Unsatisfiability resolution is not the only case when people may need to understand the reasoning behind the provided system decisions. For example, for the rule in Equation 1, a person who added the rule may reasonably expect that during daytime the blinds will remain open (instead of the lamp being turned on) as it is the most energy efficient solution. If for some reason the system decides to turn on the lamp, the person may want to understand why such decision has been made.

Table 1. Highlighting inconsistency in original rules.

ID	Original rules
1	$room.presence > 0 \Rightarrow$ **room.lamp = on** $\vee\ outsidelux > 1000 \wedge$ **room.blinds = open**
2	$chair.pressure = true \Rightarrow room.lamp = on \vee desk.lamp = on$
3	$desk.activity = workWithPC \Rightarrow$ **room.blinds = closed** $\wedge$ $desk.PC = on \wedge desk.LCD = on$
4	$\neg(desk.activity = workWithPC \wedge desk.lamp = on \vee$ $desk.activity = paperwork \wedge desk.lamp = off)$
5	$room.activity = presentation \Rightarrow room.beamer = on \wedge$ **room.lamp = off** $\wedge\ desk.PC = on$

ID	Transformed rules
1.1	**room.presence > 0 $\Rightarrow$ room.lamp = on $\vee$ room.blinds = open**
1.2	$room.presence > 0 \wedge outsidelux \leq 1000 \Rightarrow room.lamp = on$
2.1	$chair.pressure = true \Rightarrow room.lamp = on \vee desk.lamp = on$
3.1	**desk.activity = workWithPC $\Rightarrow$ room.blinds = closed**
3.2	$desk.activity = workWithPC \Rightarrow desk.PC = on$
3.3	$desk.activity = workWithPC \Rightarrow desk.LCD = on$
4.1	$desk.activity = workWithPC \Rightarrow desk.lamp \neq on$
4.2	$desk.activity = paperwork \Rightarrow desk.lamp \neq off$
5.1	**room.activity = presentation $\Rightarrow$ room.lamp = off**
5.2	$room.activity = presentation \Rightarrow room.beamer = on$
5.3	$room.activity = presentation \Rightarrow desk.PC = on$

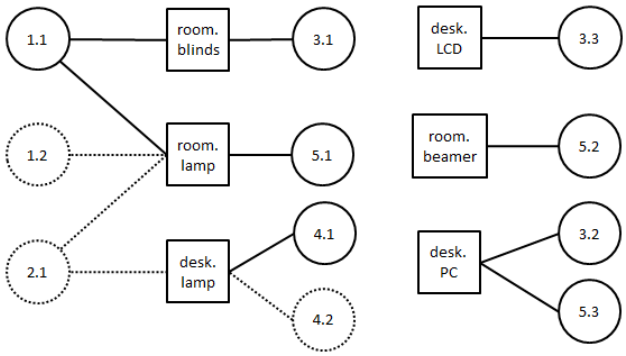


Fig. 2. Dependency Graph for rules in Table 1 and sensor values in Table 2

Table 2. Sensor values.

Sensor	State
chair.pressure	false
desk.activity	workWithPC
outsidelux	1500
room.activity	presentation
room.presence	5

Very often the reason is that some other rule added constraints preventing blinds to be open, but manually finding such rules may be a hard and tedious process. As with the case of unsatisfiability, the system should be able to show exactly, which

rules or parts of rules participate in the current solution of turning on the lamp instead of opening the blinds.

Interestingly, the Dependency Graph allows to address these issues easily. During the rule transformation process that is described in Section 5 and in Appendix, every original atomic predicate can be traced from the original rule to the transformed rules. This feature allows the system to mark those atomic predicates that participate in the unsatisfiable CSP task. It also allows to mark all atomic predicates that are in the same subgraph as a certain actuator (e.g. *room.lamp* in our example). Only those parts of rules should be shown to the person and examined, which largely reduces the amount of time for manual rules examination and correction.

For example, Table 1 presents original rules, as entered by people, and transformed rules, as they are used to create a dependency graph, which is itself shown in the Figure 2. The activeness status corresponds to the state of sensors as shown in Table 2. The rules that are marked as bold produce an unsatisfiable active subgraph. Note that only relevant parts of the original rules are highlighted.

8. Evaluation

8.1. *Architecture*

The internal architecture of the Rule Maintenance Engine implementation is shown in Figure 3.

The Web User Interface presents all information about the RME system and its decisions to users. The RME itself runs as a back-end server, and provides a REST interface to show and modify the data.[21] The REST interface is used by the front-end of the system, which consists of a client web interface which runs on the Play Framework[b] and an HTML5-based interface for building context information and immediate manual control via mobile devices, such as smartphones and tablets running Android. The REST interface can also be used by other applications to make modifications to the system programmatically.

The initial configuration of the RME system is loaded at startup from the Repository. The Repository contains all required information about the devices, services, or virtual variables, which together form an environment description for the RME. The Repository also contains the latest values of the sensors, so it is immediately possible for the RME to make an initial check of the environment and issue any state goals. This also makes the system tolerant to failures and crashes, as it automatically returns to its latest state after restart. Users can override any part of the environment configuration via the dedicated Web UI. The reconfiguration, as well as addition of new devices or modification of existing ones can be done dynamically, without the need of restarting the system.

Another part of the RME system is the Rule Manager, which manages the set of rules of the building behavior. The initial set of rules is loaded at the startup of

[b] `http://www.playframework.com/`

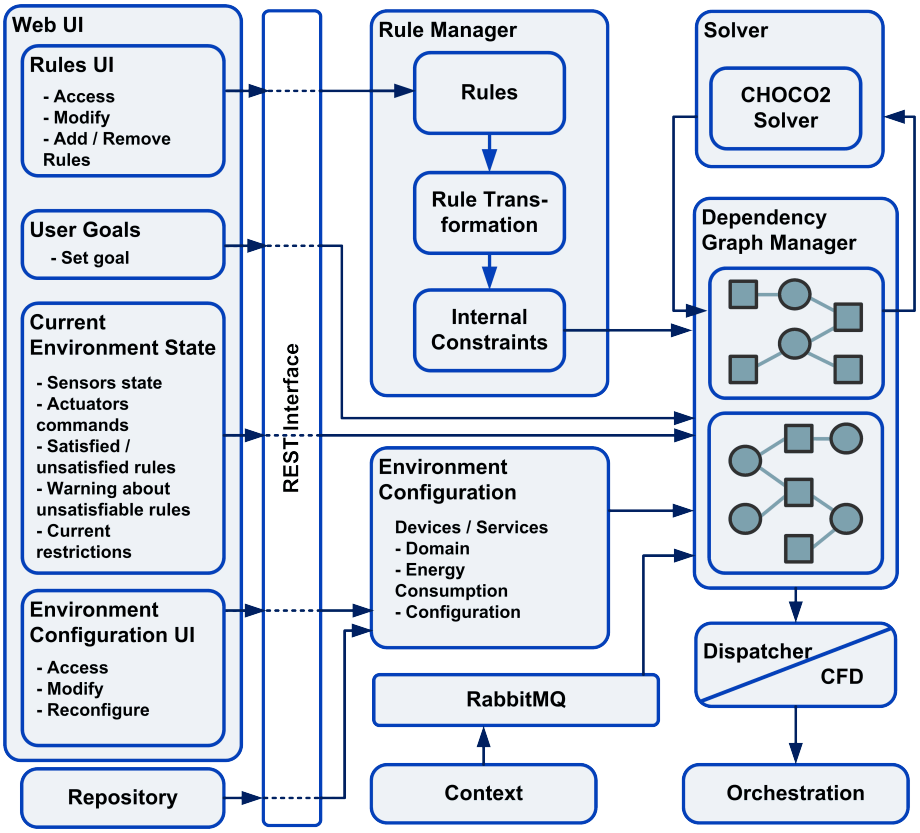


Fig. 3. Rule Maintenance Engine Architecture

the system. It is also possible to switch between sets, and a user can modify any rule, add new ones, or remove obsolete ones through the dedicated Web UI. Every rule is a formula in predicate logic, it can be added in any form by the users, then it will be checked for correctness and consistency, and then it will be transformed into the internal constraint form, which is used by the Rule Maintenance Engine. The transformation is done once every time the rule is added or changed, and it may result in several internal constraints from a single initial rule.

Devices and rules are combined in the Dependency Graph Manager (DG). It contains the current environment state; the commands, issued to the actuators, and their execution status; warnings about currently unsatisfiable rules; which manual goals were set by system's users previously, etc. All this information is shown on a "Current Environment State" dashboard of the Web UI. This is one of the main dashboards available to users, through which they can control the system and keep track of its status.

Sensors are the main source of events. The Context component collects raw sensor data, processes it, performs activity recognition,[22,23] and sends results to

the RME. For the RME effectively both low-level sensors and high-level activities are represented through environment variables. Since there can be many events per second, the scalable and highly reliable messaging system is used to transfer this information. For the GreenerBuildings project the RabbitMQ[c] messaging framework is used.[24] The RME subscribes to the updates it is interested in, and receives them as soon as they are published by the Context. When the event arrives, the Dependency Graph Manager checks, which parts of the environment may be affected by this change, and whether any actuators' states should be rechecked. If this is the case, the Solver is invoked, which finds the new optimal states of affected actuators. The search problem for the Solver is represented as an optimization constraint satisfaction problem (CSP). We use the CHOCO2 Solver[d] library for the task.[25]

The second way of obtaining events is from the User Control component (or UI), where users may set their goals manually. For example, the current rules may say that a temperature in a certain room may be as low as 19 degrees Celsius, but if a user specifies that she wants the temperature to be 22 degrees, the event will be generated and sent to the Dependency Graph Manager.

Finally, when it is calculated that some actuator should perform a certain action or change its state, the goal is generated by the Dependency Graph Manager and is sent further to the GreenerBuildings system for execution.

8.2. *Living Lab*

The system was evaluated in the living lab constructed on the premises of the Technical University of Eindhoven, the Netherlands. In this section we will describe the implementation details of the living lab.

The living lab features two large spaces: a working room with four working desks, and a meeting room, with a meeting table and a presentation area. The sensors include Plugwise power meters[e], CO_2 and humidity, passive infrared (PIR) motion, temperature, light, ultrasound (USR), acoustic. The actuators include Plugwise switches for devices such as projectors and lamps, dimmers for fine-grained control of ceiling lamps' light levels, motor controllers for blinds heights and angles, HVAC system.

As variables, the Rule Maintenance Engine contains both raw sensor data and activity recognition results. Tables 3 and 4 show the description of the variables within the RME system. In total there are 135 variables. Among them 82 represent physical sensors and contain raw sensor data, 26 represent virtual sensors with higher level recognized activity, and 27 represent actuators, 3 of which belong to the thermo-fluid dynamics (TFD) system.

Rules were changing over time, with the original preset having 39 rules that are transformed as described in Section 5 into 62 internal constraints. The rules are de-

[c] `http://www.rabbitmq.com/`
[d] `http://www.emn.fr/z-info/choco-solver/`
[e] `http://www.plugwise.com/`

Table 3. Living Lab actuators details

Type	Name	Datatype	States/Range	Number of variables
Misc	Blinds angle	Float	-90.0..90.0	3
Misc	Blinds height	Float	0..265	3
TFD	PMV comfort	Float	-2.0..2.0	1
Misc	Table lamp	Boolean	true/false	4
Misc	Light dimmer	Integer	0..1000	10
TFD	Policy	Set	comfort/economy	1
Misc	Screen	Boolean	true/false	4
TFD	Temperature	Float	16.0..27.0	1
			Total number of actuators:	27

signed for different adaptation scenarios, which include the adaptation for natural and artificial lighting, different activity types in a meeting room, rules for working space personalization, heating system, etc. Control UI allows users to override system's decisions, and set any actuator manually. Here we present the main ideas of several rule adaptation scenarios.

The first scenario concerns adaptation of natural lighting, and contains rules for blinds control (except those that are defined for the meeting and presentation activities, see below). It is usually beneficial for a room to get natural light and warmth from the outside, but when the natural light outside is too bright, it causes glares inside, so the system must ensure that the blinds angle is enough to give sufficient light inside, but not that small to enable sun glare, which decreases people's comfort. The total number of human-defined rules for this case was 5, and these rules translated into 15 internal rules. Examples of these rules are:

$$room313.presence1 = false \Rightarrow room313.blinds.height1 = 0$$

$$room313.presence1 = true \Rightarrow room313.blinds.height1 = 100$$

$$light.luxlevelout1 > 5000 \Rightarrow room313.blinds.angle1 > 70$$

The next scenario is the activities in the meeting room and adaptation to them. It is possible to have a normal brainstorming meeting inside, or to give a presentation, or even none of the above, as the room is open for occasional presence. If people are present inside, but there is no presentation, the total lighting level should be sufficient (at least 500 lux). If the brainstorming meeting is taking place, the light levels should be even higher. There are several different dimmers on the ceiling, so there are many different ways to achieve such conditions. If people use manual control to set certain dimmers to their preferred level, it is possible to use dimmers in other areas to satisfy the rule.

If a presentation is in progress, special lighting conditions should follow. First of all, the dimmable light spot directly above the presentation screen should be fully off, otherwise the visibility of the screen severely decreases. Other dimmers should keep certain level of light, which, however, should stay low, so not to decrease the

Table 4. Living Lab sensors detailed

Type	Name	Datatype	States/Range	Number of variables
Raw	Projector	Boolean	true/false	1
Raw	CO^2 level	Float	100..3000	1
Raw	Outdoor CO^2 level	Float	100..3000	1
Raw	Computer status	Boolean	true/false	4
AR	Computer work	Boolean	true/false	4
AR	Desk work	Boolean	true/false	4
Raw	Door	Boolean	true/false	2
AR	Energy balance	Float	-10000..10000	1
AR	Heat losses	Float	-10000..10000	1
AR	PMV Status	Float	-2.0..2.0	1
AR	Policy Status	Set	comfort/economy	1
Raw	Humidity	Float	0..100	4
Raw	Outdoor Humidity	Float	0..100	1
Raw	HVAC heat production	Float	-10000..10000	1
Raw	HVAC status	Boolean	true/false	2
Raw	Lamp status	Boolean	true/false	4
Raw	Light power consumption	Integer	0..400	4
Raw	Lux level	Float	0..30000	5
Raw	Outdoor lux level	Float	0..30000	3
Raw	Lights status	Boolean	true/false	4
Raw	Lights switch	Set	0, 1, 2	4
AR	Meeting brainstorming	Boolean	true/false	1
AR	Presentation	Boolean	true/false	1
Raw	Power consumption	Float	0.0..1320.0	15
Raw	Distance	Integer	0..100	9
Raw	Motion	Boolean	true/false	4
AR	Number of people	Integer	0..50	2
AR	Area presence	Boolean	true/false	9
Raw	Status screen	Boolean	true/false	4
Raw	Outdoor temperature	Float	-10.0..80.0	1
AR	Indoor temperature mean	Float	16.0..27.0	1
Raw	Indoor temperature	Float	-10.0..80.0	4
Raw	Window	Boolean	true/false	4
			Total number of sensors:	108

screen visibility. The 3 human-defined rules are translated into 10 internal rules. Examples are:

$$room313.presence1 = true \wedge room313.meeting.presentation \neq true \Rightarrow room313.light.dimmer1 > 200 \wedge room313.light.dimmer2 > 200 \wedge room313.light.dimmer4 > 100$$

$$room313.meeting.brainstorming = true \wedge room313.meeting.presentation \neq true \Rightarrow room313.light.dimmer2 > 600 \wedge room313.light.dimmer3 > 600 \wedge room313.light.dimmer4 > 600$$

$$room313.meeting.presentation = true \Rightarrow room313.blinds.angle1 > 80 \wedge room313.light.dimmer2 = 0 \wedge room313.light.dimmer1 < 300 \wedge room313.light.dimmer4 < 300$$

The heating mechanism of GreenerBuildings is based on the Computational Fluid Dynamics (CFD) module[f], which calculates the air quality, temperature, humidity, and other climate conditions within the room, and uses available actuators for fine-grained control of the climate comfort levels. As there is a dedicated module which does the required complex computations, the RME does not invoke heating actuators (such as HVAC module) directly, and instead has abstracted actuators, namely mean temperature, policy (economy or comfort) and PMV (predicted mean vote) comfort level. When the best value of abstracted actuators is calculated, it is sent to the CFD component, which performs the necessary fine-grained control of physical devices. The RME rules include having a comfort policy only when there are people inside (otherwise it is automatically set to economy), and stopping all fine-grained control if windows are open. Most of the time, people add their own rules for the temperature levels, which they deem comfortable to them, so the temperature rules are not included in the preset. The 2 rules that are included correspond to 5 internal rules:

$$room313.window1 = true \vee room313.window2 = true \Rightarrow room313.pmv = unmanaged \wedge room313.temperature\_mean = unmanaged$$

$$room313.presence1 = true \Rightarrow room313.policy = comfort$$

Working areas are the areas where most of the "personalized" rules appear, as naturally the area is occupied by one person. The preset rules use standardized approach, by turning off the screen when the person is not around, distinguishing the lighting conditions for desk work and computer work, etc. The human-defined preset rules include 17 rules, which translate to 20 internal rules. Examples are:

$$room326.presence1 = true \Rightarrow room326.screen1 = true$$

$$room326.desk1.deskwork = true \Rightarrow room326.light.dimmer1 > 700$$

$$room326.presence1 = true \wedge room326.desk1.computerwork = false \Rightarrow room326.light.dimmer1 > 500$$

$$room326.desk1.computerwork = true \Rightarrow room326.light.dimmer1 < 500$$

There is a special type of rules, introduced for smooth continuous operation of the system: trigger rules. The initial reason for them is the capability of the people inside the building to manually control some of the actuators. For example, lamps or the temperature setting is generally controlled by the system via the defined

[f]Fluid Solutions - @lternative, `http://fluidsolutions-a.com/`

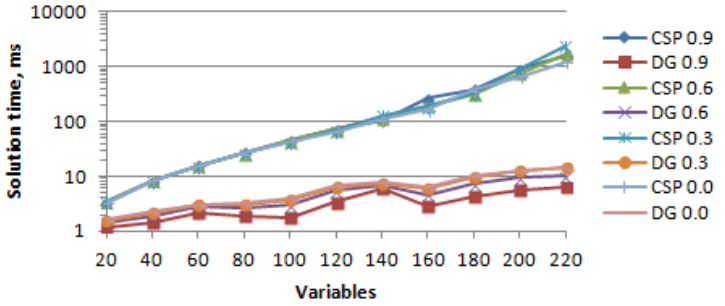
rules. However, a person has the ability at any moment to press the lamp switch or to set a certain temperature level on the thermostat. Such ability to manually set the state of the building even in contradiction with previously defined rules is very important to retain even in highly automated buildings, as it keeps people in full control increasing their overall satisfaction level. So, once a person sets some actuator manually, this actuator should be forbidden to be changed by the system. It may still be possible to satisfy existing rules in a different manner. For example, for the rule (1), if a person prefers to keep the blinds closed, it is still possible to turn on the lamp, which will satisfy the rule. But then another problem arises. Once a certain actuator is activated manually, when is it possible for the system to "take it back", i.e. to remove the restriction on its state modification? For example, once a person turned on a lamp, if she goes away from the room, we may assume that the restriction is no longer valid. Trigger rules are designed specifically for such case. The rule may be specified, as usual, but it must be specifically marked as a trigger rule, and two additional things must be provided: (1) the actuator, for which the restriction is removed once the rule is satisfied, and (2) the type of the restriction that is removed. The current implementation of the system supports two types of restriction: the manual user input, and the error in the device (which marks the device as broken, and stops the control of it until fixed). In the preset, all devices are released from their user restriction if the room becomes unoccupied. There are 12 preset trigger rules, an example of such a rule is:

$$\begin{aligned} &actuator: \ room326.light.dimmer3 \\ &type: \ user\_feedback \\ &rule: \ room326.presence1 = false \end{aligned}$$

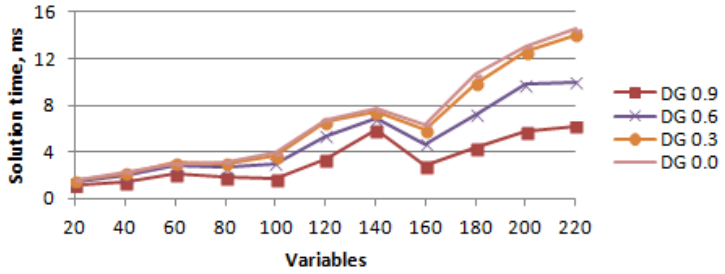
The operation of the living lab showed that our module solves all resulting CSPs in a matter of milliseconds, returning real-time commands to actuators. The next step of the project is to extend the system to more rooms, and the whole building, so the next section discusses the performance and scalability potential of our solution in depth.

8.3. *Performance*

To evaluate the effectiveness of our solution with greater flexibility, we also made performance experiments that were running on Windows 7, Intel Core2Duo E7400 @2.8GHz, 4 Gb RAM, Java7 machine. As a baseline, we used random instances with boolean variables. Note that any instance with arbitrary sizes of domains can be converted into an equivalent instance with boolean variables, one per each domain value. Every instance has half of its variables as sensors, and half as actuators. For every set of parameters we generate 50 different instances. Every instance ran for 100 sensor change events. For every event the time to find a solution is recorded, and the average time across these runs is presented in the figures. Every rule is a random constraint between two sensors and two actuators, and the number of rules equals to the 120% of the number of variables.



(a) CSP vs. DG, log scale, clusterization values of 0.9, 0.6, 0.3, 0.0



(b) DG-only close-up

Fig. 4. Average solution times of CSP and DG representations

We also analyzed the impact of clusterization on the performance of the DG solution. In smart environments most variables are naturally split into clusters of highly-dependent variables, e.g. by location, with loose dependency between clusters. Thus we introduce clusters of variables in our instances, with varying degrees of clusterization. For example, for a degree of 0.6, 60% of rules will connect variables within a cluster, and remaining rules connect any variables, also across clusters. We used clusterization values of 0.9 (very distinctly defined clusters), 0.6, 0.3 and 0.0 (no clusters, every rule connects variables fully randomly). The number of clusters is $\sqrt{|V|}$, so an instance with 40 variables has 6 clusters with 6-7 variables each, while an instance with 400 variables has 20 clusters with 20 variables each.

Figure 4 compares solution times using a natural CSP definition (as given in Section 4), and using the Dependency Graph data structure. The time of rule activeness rechecking and graph traversals is *included* into the resulting time for the DG, i.e. results include all overhead, associated with using the DG data structure. It can be seen that for all cases DG severely outperforms the natural CSP definition, staying at around 10 milliseconds time for over 200 variables, while CSP already goes to over 1000 milliseconds solution time for such cases. The clusterization parameter has no influence on CSP solution time, which is expected, since CSP takes the full environment into account. However, for DG it is shown, that the bigger the clusterization is, the lower the solution time will be, which also means much bigger

scalability potential for implementing the solution in smart buildings.

Table 5. Random instance run, 100 variables, 0.0 clusterization

Event		1	2	3	4	5	6	7	8
CSP Time		45.68	41.38	71.06	36.05	25.24	32.93	34.02	66.47
DG	**Time**	8.71	12.11	8.47	10.62	7.86	6.71	5.69	4.19
	Size(s)	1;21	22;1;1;8	1;1;20	26;2	3;25	25;5	1;26	1;1;3

Event		9	10	11	12	13	14	15
CSP Time		29.42	31.18	31.04	56.98	29.20	29.10	66.16
DG	**Time**	8.33	7.83	4.88	0.08	18.04	2.80	4.46
	Size(s)	1;1;23;1	22;1;2;1	2;20	-	2;1;2;1;1;20	21	3;5;17

For better insight we included the detailed data from one of the runs of the system on an instance of 100 variables with 0.0 clusterization in Table 5. Every event corresponds to a single sensor change. The size of the CSP definition is always the same (100 variables, among which 50 are decision variables, i.e. actuators), while the DG size varies, depending on the current size of active subgraphs. As every sensor can be a part of several rules, it is customary that a single sensor change triggers re-optimization of several subgraphs. E.g. event 6 triggers two DG tasks, one with 25 variables, and the other with 5 variables. Event 12 has no impact on active subgraphs, so no re-optimization occurs.

9. Conclusions

In this paper we investigated the problem of finding the best environment state that conforms to all constraints that appear due to physical environment configuration and due to different preferences of users. We showed that such a problem can be easily modelled as a constraint satisfaction problem (CSP). However, the straightforward CSP task does not fulfill the requirements for scalability and computational efficiency, because of multiple unnecessary computations that need to be performed after every new sensor change. Since in a dynamic environment there may be many sensor changes per second, the system may not be able to respond to changes in real-time.

The dependency graph data structure that is introduced in this paper is able to dynamically keep track of interdependent parts of the environment. After every sensor change the dependency graph provides information on which parts of the environment and which rules are affected by it. We formally proved that it is possible to only recheck the affected part while still keeping the full environment globally satisfied and optimal.

The experiments performed in a real living lab environment and in simulations proved that the usage of the dependency graph consistently outperforms the straightforward CSP task. The solution for real environments is able to compute the optimal answer in real-time, therefore it complies to the requirements of scalability and computational efficiency.

The description of real smart environments shows that such environments contain *clusters* of variables, with high level of dependency among variables within a cluster, and loose dependency of those variables on variables outside of the cluster. Clusters usually contain variables within a single physical space, such as a room or a single working desk, and sometimes clusters can be split by nature of variables, for example variables that affect lighting system can form a cluster. The experiments performed with the solution based on the dependency graph showed that the performance increases with more distinctly defined clusters.

Acknowledgment

The research is supported by the EU project GreenerBuildings, contract FP7-258888, and by the Dutch NWO Smart Energy Systems program, contract 647.000.004.

We would like to thank Marco Aiello and Krzysztof Apt for useful comments about this work.

Appendix

In this appendix we present Algorithm 3, which describes the process of rules transformation to the CNF form. Note that we do not allow negated atomic predicates, therefore a step is added that flips the operation of an atomic predicate if it is negated.

References

1. V. Degeler, L. I. L. Gonzalez, M. Leva, P. Shrubsole, S. Bonomi, O. Amft, and A. Lazovik, "Service-oriented architecture for smart environments (short paper)," in *Service-Oriented Computing and Applications (SOCA), 2013 IEEE 6th International Conference on.* IEEE, 2013, pp. 99–104.
2. G. Verfaillie and T. Schiex, "Solution reuse in dynamic constraint satisfaction problems," in *Proc. of the National Conference on Artificial Intelligence*, 1994, pp. 307–312.
3. V. Degeler and A. Lazovik, "Cost-efficient context-aware rule maintenance," in *IEEE Int. Conf. Pervasive Computing and Communications (PERCOM) Workshops*, 2012, pp. 608–612.
4. ——, "Dynamic constraint reasoning in smart environments," in *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, Nov 2013, pp. 167–174.
5. F. Pecora and A. Cesta, "Dcop for smart homes: A case study," *Computational Intelligence*, vol. 23, no. 4, pp. 395–419, 2007.
6. K. Petersen, A. Kleiner, and O. von Stryk, "Fast task-sequence allocation for heterogeneous robot teams with a human in the loop," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on.* IEEE, 2013, pp. 1648–1655.
7. M. Koes, I. Nourbakhsh, and K. Sycara, "Constraint optimization coordination architecture for search and rescue robotics," in *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on.* IEEE, 2006, pp. 3977–3982.

Algorithm 3 Parsing to CNF

```
1: function parseToCNF (r)
2: switch r
3:    case a ∧ b[∧...]: return parseToCNF(a) ∧ parseToCNF(b)[∧...]
4:    case a ∨ b[∨...]: return combineCNF(parseToCNF(a),
5:          [parseToCNF(b) | combineCNF(parseToCNF(b),...])
6:    case ¬a: return parseNot(a)
7:    case a ⇒ b: return parseToCNF(¬a ∨ b)
8:    case a ⇔ b: return parseToCNF((¬a ∨ b) ∧ (a ∨ ¬b))
9:    case a ∈ P(v): return a
10: end switch
11:
12: function parseNot (r)
13: switch r
14:    case a ∧ b[∧...]: return parseNot(a) ∨ parseNot(b)[∨...]
15:    case a ∨ b[∨...]: return parseNot(a) ∧ parseNot(b)[∧...]
16:    case ¬a: return parseToCNF(a)
17:    case a ⇒ b: return parseToCNF(a ∧ ¬b)
18:    case a ⇔ b: return parseToCNF((a ∨ b) ∧ (¬a ∨ ¬b))
19:    case a ∈ P(v): return flipOperation(a)
20: end switch
21:
22: function combineCNF (a, b)
23: required a = a1[∧a2 ∧ ...]; b = b1[∧b2 ∧ ...]
24: return (a1 ∨ b1)[∧(a1 ∨ b2) ∧ ... ∧ (a2 ∨ b1) ∧ ...]
```

8. A. Cesta, G. Cortellessa, A. Oddi, N. Policella, and A. Susi, "A constraint-based architecture for flexible support to activity scheduling," in *AI* IA 2001: Advances in Artificial Intelligence*. Springer, 2001, pp. 369–381.
9. E. Kaldeli, E. U. Warriach, J. Bresser, A. Lazovik, and M. Aiello, "Integrating, composing and simulating services at home," in *International Conference on Service Oriented Computing (ICSOC)*, 2010.
10. E. Kaldeli, E. U. Warriach, A. Lazovik, and M. Aiello, "Coordinating the web of services for a smart home," *ACM Transactions on the Web*, 2012.
11. R. Dechter and A. Dechter, *Belief maintenance in dynamic constraint networks*. University of California, Computer Science Department, 1988.
12. C. Bessiere, "Arc-consistency in dynamic constraint satisfaction problems," in *Proceedings AAAI'91*, 1991.
13. A. K. Mackworth, "Consistency in networks of relations," *Artificial intelligence*, vol. 8, no. 1, pp. 99–118, 1977.
14. R. Dechter and J. Pearl, "Network-based heuristics for constraint-satisfaction problems," *Artificial Intelligence*, vol. 34, no. 1, pp. 1–38, 1987.
15. R. Debruyne, "Arc-consistency in dynamic CSPs is no more prohibitive," in *IEEE Int. Conf. Tools with Artificial Intelligence (ICTAI)*, 1996, pp. 299–306.
16. T. Schiex and G. Verfaillie, "Nogood recording for static and dynamic constraint

satisfaction problems," *Int. Journal of Artificial Intelligence Tools*, vol. 3-2, pp. 187–207, 1994.
17. N. Roos, Y. Ran, and J. Van Den Herik, "Combining local search and constraint propagation to find a minimal change solution for a dynamic csp," in *Artificial Intelligence: Methodology, Systems, and Applications*. Springer, 2000, pp. 272–282.
18. Y. Ran, N. Roos, and J. van den Herik, "Approaches to find a near-minimal change solution for dynamic CSPs," in *Fourth international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimisation problems*, 2002, pp. 373–387.
19. S. Mittal and B. Falkenhainer, "Dynamic constraint satisfaction," in *Nat. Conf. on Artificial Intelligence*, 1990, pp. 25–32.
20. G. Verfaillie and N. Jussien, "Constraint solving in uncertain and dynamic environments: A survey," *Constraints*, vol. 10, no. 3, pp. 253–281, 2005.
21. R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.
22. O. Amft and C. Lombriser, "Modelling of distributed activity recognition in the home environment," in *Int. Conf. Engineering in Medicine and Biology Society (EMBC)*. IEEE, 2011, pp. 1781–1784.
23. F. Wahl, M. Milenkovic, and O. Amft, "A distributed PIR-based approach for estimating people count in office environments," in *Int. Conf. Computational Science and Engineering (CSE)*. IEEE, 2012, pp. 640–647.
24. A. Videla and J. J. Williams, *RabbitMQ in action*. Manning, 2012.
25. N. Jussien, G. Rochart, X. Lorca *et al.*, "Choco: an open source java constraint programming library," in *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, 2008, pp. 1–10.