Dynamic Constraint Reasoning in Smart Environments

Viktoriya Degeler and Alexander Lazovik
Distributed Systems Group, Johann Bernoulli Institute,
University of Groningen, The Netherlands,
Email: {v.degeler, a.lazovik}@rug.nl

***Abstract*—Flexible and easily adjustable reasoning mechanisms are essential for rendering sensor and actuator rich indoor environments smart. Constraint-based solutions are a suitable approach for such systems. We propose an approach that allows users to specify the rules for a building's behavior, and uses context information to represent the rules and environment as a dynamic constraint satisfaction problem. The dependency graph data structure allows to find efficiently only the affected parts of the environment, thus minimizing the computational efforts after every event. We evaluate the system on a building implementation as a living lab, and with performance experiments. The testing proves the high efficiency and applicability of the approach for dynamic control of smart environments.**

***Keywords*-dynamic constraint satisfaction; rule-based systems; smart environments; predicate logic**

I. Introduction

The recent advancements in pervasive computing area allow many smart home solutions to hit the market, the goal of such systems being the ultimate increase in user comfort and satisfaction levels, as well as efficient operation of the environment, for example, from the energy and price point of view. 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 and their ability to fully control the system's decisions, adapting them to their goals and desires at any moment of time. Therefore, flexible and adaptable reasoning mechanisms are essential for environments automation.

Our approach, implemented in the GreenerBuildings [1] European FP7 project, 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 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. In particular, the model falls into the Dynamic CSP (DCSP) [2] category, 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. 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 to be solved as an optimization CSP.

Dynamic constraint satisfaction problem (DCSP) as a set of successive static CSPs with addition or removal of constraints was first formulated in [3], and subsequently targeted in many other works [4], [5], [6], [7], [2]. A comprehensive survey of the related research is presented in [8]. An alternative definition of DCSP was formulated in [9], where DCSP defines a single CSP with different additional sets of variables and constraints depending on variable values. We use the definition as stated in [3], when referring to the DCSP.

In the field of smart environments different studies propose to use the constraint satisfaction to solve reasoning problems. For example, multi-agent coordination in smart homes is modelled as distributed constraint optimization problem in [10]. Each agent manages one or more variables, and constraints model the desired minimum-cost concurrent behavior of agents. CSP-based AI planner is used in [11] to compose services for smart home scenarios. The planner allows the expression of extended goals and uses the latest advancements in the CSP field to make the search faster using enhanced inference techniques.

In this paper, we present the dynamic constraint satisfaction solution of 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, and 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 possible. Our initial ideas on context-aware rule maintenance were presented in [12]. In this paper we formally extend them 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.

II. Rule Satisfaction in Smart Environments

The GreenerBuildings project aims to increase the overall users' comfort by adapting to their needs. Usually there are several ways to satisfy them, and the additional goal is to assure the minimum energy consumption of the building, without sacrificing the user comfort. The reasoning is handled by the Rule Maintenance Engine (RME) component. Via the web interface the users are able to access the current rules, modify them, add new ones, etc. The information about the current state of the environment comes from the Context component as new sensor readings events. Informally the RME goal 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. If there are some rules which cannot be satisfied at this moment, the user must be presented with sufficient information to identify the cause of this. (2) Given the satisfaction of the rules, the energy consumption of the building should be minimal. (3) Decisions should be made in real-time and be scalable with respect to the environment size.

There are two types of rules. The RME system handles them equivalently, but for the users of the system they represent a difference between what is *necessary* and what is *desirable*. The first type is a *dependency between variables*. E.g., 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. A rule $\neg(room1.blinds1 = down \wedge room1.window1 = open)$ 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 in the room if there are some 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:

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

III. 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 external 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. On the other hand, controllable variables A can be seen as actuators that can act in the environment. 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 out of its domain to any other state out 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 states domain $d(v)$ that shows the cost of keeping the variable in this state. For the GreenerBuildings project the cost is the energy consumption of a device.

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 on practice. First of all, the CSP task should be solved with every new sensor change event. For the smart buildings with hundreds of sensors several of such events arrive every second, which makes solving the CSP a computationally heavy task. Also, every sensor change affects only a small part of the environment, therefore solving from scratch every time produces a lot of duplicate work. Using dynamic constraint satisfaction techniques would be more computationally efficient. Finally, in practice many rules (constraints) for intelligent environments are only applicable for a particular situation, which may be active only a small percentage of the time. For most of the time the constraints will not be applicable, however it will still need to be added as a part of the CSP over and over again.

The classic definition of Dynamic Constraint Satisfaction Problem [3], [4] 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. We can 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 $s = d^s_{new}$. However, while in the classic definition the domain remains of the same size, 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.

IV. 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 form. Transformations are done once at the time of addition of a new rule by users (or after a rule has been 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 want to split the rule into as many independent sub-rules as possible. E.g., 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 "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 $\bigwedge_s(P(S)) => \bigvee_a(P(A))$. 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 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 a 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 implication by taking those atomic predicates that contain only sensors, and putting them (in negated form) into the antecedent of the implication. The next step is not necessary, and is done only for convenience: 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 $desk1.pc \neq on$.

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: "$room1.presence > 0 \Rightarrow room1.lamp = on \vee outsidelux > 1000 \wedge room1.blinds = open$". Sensors here are $room1.presence$ and $outsidelux$. So, by putting it into CNF, splitting on two different rules, putting the sensors to the antecedent, and removing the negation from atomic predicates we obtain the following two rules: "$room1.presence > 0 \Rightarrow room1.lamp = on \vee room1.blinds = open$" and "$room1.presence > 0 \wedge outsidelux \leq 1000 \Rightarrow room1.lamp = on$".

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. On practice, since optimization CSP is used, if both choices are not restricted the system will choose to open the blinds as more energy efficient choice. But if the outside light level is sufficiently small, 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.

V. 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. The dynamic dependency mechanism, which is realised via the use of the Dependency Graph is specifically the mechanism designed to keep 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 IV, 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 of time sensor variables have a particular valuation, based on the context environment information, and represented by a set of rules R_s: $s = d_s$, $\forall s \in S$. So, 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 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. The activeness of a rule changes with time and 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 CSP task, but also many original rules are removed, leaving only those that are relevant to the current situation and state of the environment. Given the nature of smart environment rules, it may be a small subset of original rules.

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:

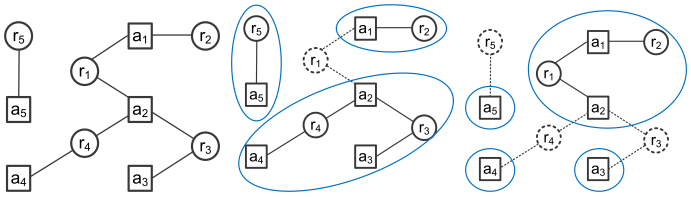
Let $X(V_x)$ represent the set of *full Cartesian product of values* for a variable set V_x: $d(v_{x1}) \times d(v_{x2}) \times \ldots$. Let $r(x)$ for $r \in R$, $x \in X(A)$ identify the result of evaluation ($true$ or $false$) of the consequent actuator part $F_a(A)$ of a rule r with values in valuation 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 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 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 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.*



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

Figure 1. Dependency graphs

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 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 4 (Active subgraph)**.** *At a certain moment of time, an* active subgraph *of the dependency graph G is a connected subgraph of G that consists only of active vertices.*

Examples of active subgraphs are shown in Figures 1b,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 that at every subsequent step it is possible to only recheck those subgraphs that changed their structure.

Lemma 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 same variables A, but only a subset of original constraints $R_i \subseteq R^*$, therefore are less restrictive. From Definition 2 and Definition 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 2 and 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 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 a lot of 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:

Theorem 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 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 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 1 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, and if it is changed, the rule is marked accordingly. While exists a *changed* rule r, the system finds a set of adjacent active subgraphs for this rule. If rule changed to inactive there may be more that 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.

Algorithm 1 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
```

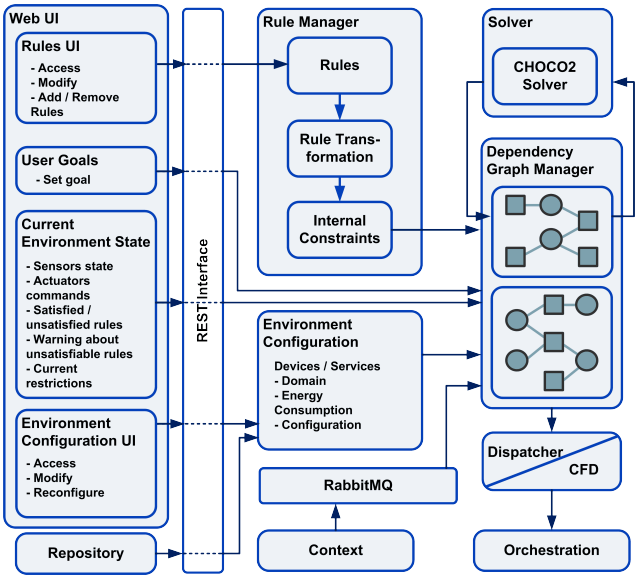


Figure 2. Rule Maintenance Engine Architecture

VI. Evaluation

A. Architecture

Figure 2 shows the Rule Maintenance Engine architecture.

The Web UI 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. The front-end consists of a client UI that runs on the Play Framework [13] and an HTML5-based interface for context information and manual control via mobile devices, such as Android-based smartphones and tablets. The REST interface can also be used by other applications to make modifications to the system programmatically.

The Repository contains information about the devices, services or virtual variables, and the latest values of the sensors. Information is loaded at startup, so it is immediately possible to make the initial check of the environment and issue any state goals. This also makes the RME 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 Web UI. Addition or modification of devices can be done dynamically, without the need to restart the system.

The Rule Manager loads rules at startup. A user can switch between sets, or modify rules via the dedicated Web UI. Rules are checked for correctness and consistency, and transformed to the internal constraint form. The transformation is done once when the rule is modified, 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. Through the associated dashboard of the Web UI the users can keep track of the current system's status.

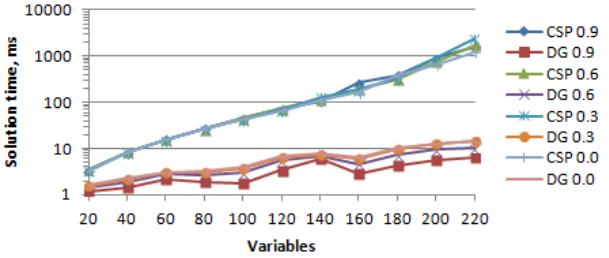
Sensors are the main source of events. The Context component collects raw sensor data, processes it, performs activity recognition [14], [15], and sends results to the RME. For the RME effectively both low-level sensors and high-level activities are represented through environment variables. Since there 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 [16] messaging framework is used. The RME subscribes to the updates it is interested in, and receives them when they are published by the Context. With every event the DG rechecks the affected parts of the environment and invokes the Solver, which finds the new optimal states of actuators. The search problem for the Solver is represented as an optimization CSP, and currently the CHOCO2 Solver [17] library is used for the task. Events are also generated by the User Control UI, where users may set their goals manually. When an actuator should change its state, the goal is generated by the DG and is sent further for execution.

B. Living Lab

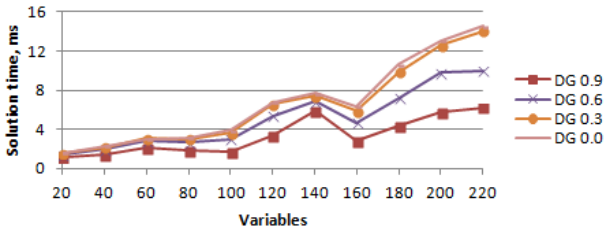
The system was evaluated in the living lab constructed on the premises of the Technical University of Eindhoven, the Netherlands. It was installed in December 2012 and has been running on a full-time basis at the time of writing this paper (June 2013). The living lab features two large spaces: a working room with four working desks, and a meeting room, with a meeting table and a presentation area. The sensors include the Plugwise power meters [18], CO_2 and humidity, passive infrared (PIR) motion, temperature, light, ultrasound (USR), acoustic, etc. 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. In total there are 135 variables. Among them are 82 raw sensors, 26 recognized activity sensors, 27 actuators.

Rules change over time, with the original preset having 39 rules that are transformed as described in Section IV into 62 internal constraints. The rules are designed for different adaptation scopes, 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.

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.



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



(b) DG-only close-up

Figure 3. Average solution times of CSP and DG representations

C. 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 runs for 100 sensor change events and 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.

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. E.g., 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.

Table I
DETAILED DATA FROM A PART OF A RANDOM INSTANCE RUN, 100 VARIABLES, 0.0 CLUSTERIZATION

Event		1	2	3	4	5	6	7	8	9	10	11	12
CSP Time		45.68	41.38	71.06	36.05	25.24	32.93	34.02	66.47	29.42	31.18	31.04	56.98
DG	Time	8.71	12.11	8.47	10.62	7.86	6.71	5.69	4.19	8.33	7.83	4.88	0.08
	Size(s)	1;21	22;1;1;8	1;1;20	26;2	3;25	25;5	1;26	1;1;3	1;1;23;1	22;1;2;1	2;20	-

Figure 3 compares solution times using a natural CSP definition (as given in Section III), 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.

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 I. 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.

VII. CONCLUSIONS

We presented a reasoning mechanism for smart environments which models the environment as a dynamic constraint satisfaction problem. We showed how the domain information can be used effectively to reduce the size of every subsequent CSP by reusing parts of the earlier solution. We presented the dependency graph, which effectively captures the dependencies between actuators, and allows to find dynamically independent subsets of variables. We proved that the partial rechecking of the environment still retains the global satisfiability and optimality of the solution. The system is fully implemented and evaluated as a part of the GreenerBuildings project solution for smart environments automation. Experiments show big performance improvement and scalability potential of the DG approach.

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.

REFERENCES

[1] "GreenerBuildings," http://www.greenerbuildings.eu, 2013.
[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] R. Dechter and A. Dechter, *Belief maintenance in dynamic constraint networks*. University of California, Computer Science Department, 1988.
[4] C. Bessiere, "Arc-consistency in dynamic constraint satisfaction problems," in *Proceedings AAAI'91*, 1991.
[5] R. Debruyne, "Arc-consistency in dynamic CSPs is no more prohibitive," in *IEEE Int. Conf. Tools with Artificial Intelligence (ICTAI)*, 1996, pp. 299–306.
[6] 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.
[7] 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.
[8] G. Verfaillie and N. Jussien, "Constraint solving in uncertain and dynamic environments: A survey," *Constraints*, vol. 10, no. 3, pp. 253–281, 2005.
[9] S. Mittal and B. Falkenhainer, "Dynamic constraint satisfaction," in *Nat. Conf. on Artificial Intelligence*, 1990, pp. 25–32.
[10] F. Pecora and A. Cesta, "Dcop for smart homes: A case study," *Computational Intelligence*, vol. 23, no. 4, pp. 395–419, 2007.
[11] 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.
[12] 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.
[13] "Play Framework," http://www.playframework.com, 2013.
[14] 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.
[15] 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.
[16] A. Videla and J. J. Williams, *RabbitMQ in action*. Manning, 2012.
[17] 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.
[18] "Plugwise," http://www.plugwise.com, 2013.