Cost-efficient Context-aware Rule Maintenance

Viktoriya Degeler and Alexander Lazovik
Distributed Systems Group, Johann Bernoulli Institute, University of Groningen
Nijenborgh 9, 9747 AG, The Netherlands
{*V.Degeler,A.Lazovik*}*@rug.nl*

***Abstract*—Energy and other costs reduction is important in the smart homes automation area. It is cumbersome and error-prone to create proper rules for saving costs manually, thus an automatic approach is desirable that continuously checks for the possibility to save costs. We propose an approach that unifies handling of user defined rules, and searches for a possibility to move each device to a more cost-efficient state when this does not violate any rules. With every event in the environment, our approach partially rechecks only those parts of the system that are affected by the change, thus saving computational resources.**

***Keywords*-Smart homes, context awareness, cost optimization, rule maintenance.**

I. Introduction

The advances in pervasive computing stimulate research in the field of smart homes automation. Many recent projects build systems that allow to combine and adapt the work of devices in order to automatically satisfy user needs and goals [1], [2]. One paradigm of home automation allows users to enter rules of building operation to the system, so that the system will continue to operate while trying to adapt its behaviour and satisfy those rules. The gathering of context information, i.e. the information about the current state of the environment, helps to infer the best way in which the rules must be satisfied at each moment of time.

The ability to reduce energy consumption and other costs as a part of home automation can help to achieve significant savings. However, if it is not specifically taken care of, managing devices to satisfy user needs does not reduce the associated costs, but may even incur large additional costs (in terms of energy or price). Let us take as example a rule $(Jack.location = Room) \Rightarrow (Room.lamp = on)$. Once Jack comes into the room, the rule is violated if the lamp is off, and to satisfy it the system turns on the lamp. However, when Jack leaves the room afterwards, the rule is no longer relevant, yet the lamp stays turned on, unnecessarily consuming energy. The traditional way of overcoming this situation is to create constraints in a form that it also contains the cost efficiency in itself. The added rule may look as $(Jack.location \neq Room) \Rightarrow (Room.lamp \neq on)$. The application of such an approach in practice can be seen in many of the modern buildings with automatic light control. One rule states that when a motion sensor detects a motion in a room, the light should be turned on. And the second rule states that when a motion sensor detects no movement for about 15 minutes, the light should be turned off. This often causes trouble for people, working with a computer or sitting still for a long period of time. Specifying such rules explicitly may not be the easiest way of handling the situation. In our example with light in Jack's room, if we want to add another rule $(Room.door = open) \Rightarrow (Room.lamp = on)$, we will have inconsistent rules when the door is open, but the room is empty. One rule states that light should be off because Jack is not in the room, and second states that light should be on, because the door is open. To overcome this inconsistency, the energy saving rule should be updated manually and it should now have the form $(Jack.location \neq Room)\&(Room.door \neq open) \Rightarrow (Room.lamp \neq on)$. The more rules we have, the harder it is to keep them all in a consistent state. Another problem arises if the second rule about the open door is deleted later. It is easy to forget to update an energy saving rule, and even though there is no reason now to keep the light on if the door is open and no one is in the room, it will still be the case, because the rule only asks to turn off the light if the door is closed.

In our approarch we unify the cost information of devices and continuously check for a possibility to move them to a less costly state without breaking any rules. Using our approach it is easier for a domain expert to maintain rules and goals, as they need only to add the user satisfaction rules, and the cost efficiency will be dealt with by the system automatically. The advantages of our approach are: (i) the decreased complexity and amount of rules that have to be defined manually; (ii) the ability of the system to reduce costs even if rules for such a saving are not defined explicitly; (iii) computational efficiency due to a fact that only affected constraints are rechecked when situation changes; (iv) the ability of a user to know, which rules are in effect that require this device to be turned on and consume energy.

In Section II we provide the architecture of our system. Section III describes the model of environment and context information. Section IV presents the algorithm for the system's response on situation change. Section V overviews related work, and Section VI concludes the paper.

II. System overview

The GreenerBuildings project [2] aims to develop a solution for home automation that combines user satisfaction

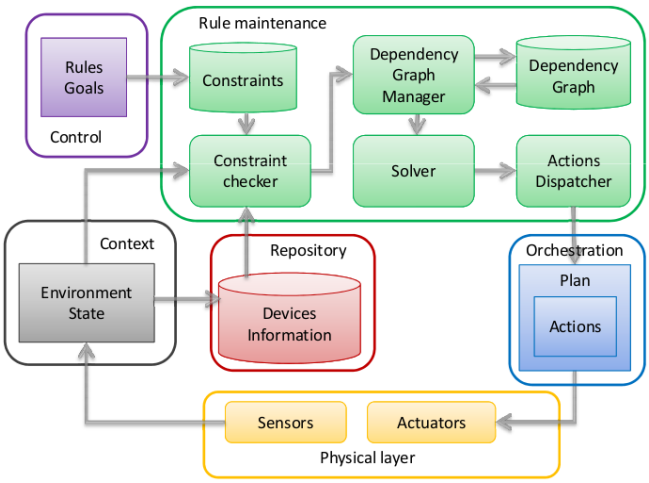


Figure 1: Smart Building system.

with energy-aware adaptation. It investigates self-powered sensors and actuators, occupant activity and behaviour inference, and an embedded software for coordinating thousands of smart objects with the goals of energy saving and user support.

Figure 1 presents a high-level overview of the system, showing main components and their inter-dependencies, as well as the internal architecture of the Rule Maintenance, which is described in details in this paper.

The Physical Layer component is responsible for gathering information from the environment sensors, and controlling the environment through the actuators. It provides an abstract unified access to the physical devices by hiding implementation details of low-level protocols, e.g., KNX, UPnP, or Bluetooth. The information gathered from sensors on a physical layer is sent to the Context component, which is responsible for creating a logical consistent view of the environment. It transforms raw data from sensors into a high-level logical representation of the environment. This data is then sent to the Repository component and the Rule Maintenance component is notified about the event. The Repository component can be seen as the environment database, which contains all the information about the domain, devices and their states, and it also stores historical data. The Control component is responsible for interaction with the users. Users enter their requirements in form of the rules of the Smart Building operations. Then, the rules are transferred from the Control component to the Rule Maintenance, and are transformed to an internal constraint view as described in Section IV. The case when all the rules can not be satisfied simultaneously, i.e. when they contradict each other, must be detected at the Control level when users enter rules to the system, so they will be asked to refine the rules in order to avoid potentially unsatisfiable situation.

When an event arrives to the Rule Maintenance, notifying about a change in the environment, the Constraint Checker component gets the updated information, finds relevant constraints and checks if all of them still hold. If there is a need to re-evaluate the current situation, the Constraint Checker updates the Dependency Graph Manager with the new information. The Dependency Graph is an internal representation of active constraints and their effects on the state of controllable devices. It is described in details in Section IV-A. The information from the Dependency Graph Manager is then used by the Solver, which finds the new optimal state of controllable devices. The Actions Dispatcher generates actions that the system has to perform in order to transform the system to a new optimal state. Those actions are then sent to the Orchestrator in a form of an execution plan. The Orchestrator in turn generates low-level commands that are sent to the Physical Layer and are performed by actuators in the environment.

A. *Cost function and off-state*

As mentioned before, one of the main tasks of the Rule Maintenance engine is to ensure that the system is always transformed to the most cost-efficient state. To be able to automate this goal the system needs to have a uniformly defined cost information for all devices. So, each state of a controllable device is associated with a certain upkeep cost. For example, a lamp can have two states, “on” and “off”, where “off” has a zero cost, and “on” has, let’s say, a cost of 0.1 kWh. Sometimes it is also possible for a lamp to have a third “dimmed” state with a cost, for example, 0.05 kWh.

If there are no user preferences or other rules associated with this device, the device should always be in the state with the minimum associated cost. For most of the devices this state is an “off-state”, when device is turned off. This leads to an observation that if the device is working and consumes energy, there always must be a rule that is violated unless the device is in this state. In our example, if the lamp is turned on, the system can always return the information to a user that it is switched on because Jack is in the room, and without turning on the lamp, the rule $(Jack.location = Room) \Rightarrow (Room.lamp = on)$ would have been unsatisfied. As soon as Jack leaves the room, this rule can be satisfied even if the lamp is turned off, so the system must recognize this situation automatically and turn off the lamp in order to reduce the upkeep costs.

The Rule Maintenance engine is designed to recognize such situations automatically. Moreover, it keeps track of dynamically changing dependencies between the controllable devices, so it will only recheck potentially affected devices with each event, decreasing required computation costs.

III. Environment and Reasoning Model

The environment $\langle V, D \rangle$ is defined by a set of context variables $V = V_u \bigcup V_c$; $V_u \bigcap V_c = \emptyset$, where $V_u = \{v_{u1}, v_{u2}, ..., v_{un}\}$ is a set of uncontrollable variables, and

$V_c = \{v_{c1}, v_{c2}, ..., v_{cn}\}$ is a set of controllable variables. Uncontrollable variables represent sensors, they provide information about the external environment, and cannot be directly influenced by the system. On the other hand, controllable variables, can be seen as actuators that act in the environment by receiving a command from the system to be set to a certain state. Every variable $v_i \in V$ varies over a *states domain* D_i, which can be either a range of integer or real values, or a set $D_i = \{d_{i1}, d_{i2}, ..., d_{im_i}\}$. Each variable v_i has a *cost function* $c(v_i)$, associated with its states domain D_i that shows the cost of keeping the variable in this state.

Let R denote a set of rules that apply to the variables in the environment. Each rule $r_i \in R$ is a formula in propositional logic, where atomic clauses $p(v_i)$ have a form $(v_i \in d_i)$ or $(v_i = d_i)$, where $d_i \subset D_i$ is a subset of the full variable domain D_i. Such atomic clause $p(v_i)$ takes $true$ if the variable v_i is in a state from the subset d_i, and $false$ otherwise.

Rules can represent either a dependency between variables, or a user preference. A *dependency between variables* represents a certain physical relationship between the affected variables. For example, the rule $\neg([Electricity = off] \wedge [Lamp = on])$ shows to the system that the lamp cannot be turned on if there is no electricity. Such rules are usually created by a domain expert. They are mostly static, and only change when a device is added or removed.

A *user preference* is entered to the system by its user and shows the desired behavior of the system. The example of such a rule is $(Jack.location = Room) \Rightarrow (Room.lamp = on)$, which shows that Jack prefers a lamp to be turned on when he is in the room.

For such model of an environment the cost-efficiency problem can be defined in the following way:

Definition 1 (Cost minimization goal). *Ensure that for a set of rules R all variables V are always in a state, where their cumulative upkeep cost is minimal, while all rules are satisfied:* $\sum_{v \in V} c(v) \rightarrow min, \quad \forall r \in R : r \equiv true$

Before going further, the original problem can be immediately simplified by taking into account the fact that the system cannot influence the state of uncontrollable variables, and can only adjust the state of controllable ones. Once the optimal cost-efficient state is found, it stays optimal until one of external uncontrollable variables changes. At that moment all the rules should be rechecked in order to ensure (i) that all of them still hold, and (ii) whether the new situation allows to put some controllable devices into a more cost-efficient state. Since the system can only set a state of controllable variables as a response to an external change in an uncontrollable one, the goal can be stated as minimization of the costs of only V_c for a given set of rules R and given values of V_u. So the above cost minimization goal can be equivalently defined as follows:

Definition 2 (Rule Maintenance goal). *Ensure that for a set of rules R and a state of a set of uncontrollable variables V_u, all controllable variables V_c are in a state, where their cumulative upkeep cost is minimal, while all rules are satisfied:* $\sum_{v \in V_c} c(v) \rightarrow min, \quad \forall r \in R : r \equiv true$

IV. Partial environment checking

A trivial approach to solving a Rule Maintenance goal in Definition 2 is to recheck all the rules each time a situation changes. However, the computation costs make this solution impossible in practice, as cost optimization is an NP-hard problem, which will make the reaction time of the system too slow for any practical environment size.

Instead of checking the whole environment, the Rule Maintenance is able to recheck only the rules (or even parts of the rules) that can be directly affected by a given change.

To ensure that we only recheck relevant parts, a number of preprocessing steps is performed on each rule when they are entered into the system. Thus, in the preprocessing phase we transform an original set of rules R to a set R^*. We do it in three steps:

1) Transform each rule from the set R to a form $\bigwedge_n (F_u^n(V_u) \Rightarrow F_c^n(V_c))$. As we anyway need to satisfy all the rules from the set, it is possible to split a rule in such a form to several rules in a form $F_u(V_u) \Rightarrow F_c(V_c)$ and consider them as separate rules. By transforming a rule to such a form, we separate a *controllable part* $F_c(V_c)$ of the rule from an *uncontrollable part* $F_u(V_u)$. What is even more important, we obtain an easy way to determine, whether a rule is working in a current situation, or not. If the uncontrollable part $F_u(V_u)$ is not satisfied, it means the whole rule is satisfied regardless of the controllable part.

2) Transform $F_c(V_c)$ to CNF form: $F_c(V_c) = \bigwedge_i F_{ci}^{'}(V_c)$, where $F_{ci}^{'}(V_c) = \bigvee_{v \in V_c} p(v)$

3) Each original rule is now in the form $F_u(V_u) \Rightarrow \bigwedge_i F_{ci}^{'}(V_c)$. For each part i of the conjunction we create a separate rule $F_u(V_u) \Rightarrow F_{ci}^{'}(V_c)$ and add it to R^*. So, for a single rule in the original ruleset R we may have more than one corresponding rule in the resulting ruleset R^*.

Doing this helps to decrease the coupling between controllable variables as much as possible. Since each part of the CNF form must be satisfied anyway, the checking can be done independently for each part. If a certain variable in a function $F_c^{'}(V_c)$ does not have restricted values, i.e. if an atomic clause of the function for this variable has a form $(v_{ci} \in D_i)$, we cancel out this variable from the function (as this atomic clause is always $true$), and we say that this rule does not depend on the variable v_{ci}.

A. Dependency graph

When the situation changes, we need to recheck all relevant controllable variables, to see if it is possible to transform them to a more cost-efficient state.

As example of variables relations, let us assume ruleset R^* has two rules:

$(v_{u1} = a) \Rightarrow (v_{c1} = 3) \vee (v_{c2} = 4)$
$(v_{u2} = b) \Rightarrow (v_{c2} = 4) \vee (v_{c3} = 2)$

For simplicity we assume that a value of controllable variables equals to their cost, e.g. the cost of variable v_{c1} in state '3' is also 3, though in general case the cost will be found by applying a corresponding cost function.

If both uncontrollable variables have the specified values ($v_{u1} = a,\ v_{u2} = b$), we must take into account both rules (because of connecting variable v_{c2}), and optimize them together. The optimal solution is ($v_{c1} = 0;\ v_{c2} = 4;\ v_{c3} = 0$).

But if the second uncontrollable variable has a different value ($v_{u2} \neq b$), the implied part $(v_{c2} = 4) \vee (v_{c3} = 2)$ of the second rule is in "inactive" state (which means it may as well not be satisfied). We now should only take into account the first rule for our optimization problem. In this case we can decrease the size of the optimization task by removing the variable v_{c3} from consideration. So we only optimize two variables with one constraint, instead of optimizing three variables with two constraints. The solution for this case is ($v_{c1} = 3;\ v_{c2} = 0$). As the third variable does not appear in any constraint, it is assigned the "off-state" ($v_{c3} = 0$).

The question is, how do we know, which variables are relevant to the current change, and which are not? We answer this question by creating a *dependency graph*, a data structure that stores relations between variables. Dependency graph is a bipartite graph, where one set of vertices consists of controllable variables, and the other set of vertices consists of the rules from R^* set, or more specific, from their controllable part $F_c(V_c)$.

Definition 3 (Dependency graph). *The dependency graph for the set of devices (V_u, V_c) and ruleset R^* is a bipartite graph $G = \langle V_c, R^*, E\rangle$, where V_c and R^* are two sets of vertices, and $E \subseteq V_c \times R^*$ is a set of edges, $(v_c, r) \in E$ iff the controllable part $F_c(V_c)$ of the rule r depends on v_c.*

Figure 2a shows an example of a dependency graph.

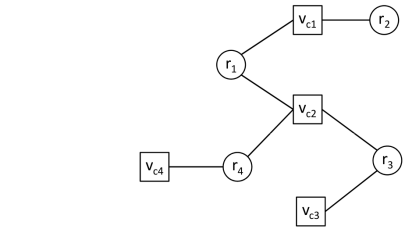
Only controllable variables act as nodes in the dependency graph, because only they may potentially require optimization. Uncontrollable variables influence the optimization task indirectly through the rule nodes. At each moment of time a rule node can be either *active* or *inactive*.

Definition 4 (Active/inactive rule vertice). *A rule node $r = (F_u(V_u) \rightarrow F_c^{'}(V_c))$ of the dependency graph is* active *in the current state of environment, if the uncontrollable part of the rule $F_u(V_u)$ is $true$, and* inactive *otherwise.*

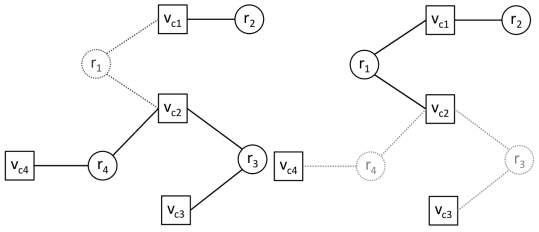
Inactive node removes the corresponding constraint for controllable variables, so the binding between them through this constraint is not active either.

Definition 5 (Active subgraph). *An active sub-graph of the dependency graph G is a connected sub-graph of G that consists only from active vertices.*

One dependency graph usually splits on several active sub-graphs. Each vertice of a dependency graph can only be a part of a single sub-graph. Examples of possible active sub-graphs are presented in Figures 2b and 2c. In Figure 2b you can see two active sub-graphs, out of nodes (v_{c1}, r_2) and $(v_{c2}, v_{c3}, v_{c4}, r_3, r_4)$. And in Figure 2c there are three active sub-graphs, one is $(v_{c1}, v_{c2}, r_1, r_2)$, and the other two have only single variable node (v_{c3}) and (v_{c4}), which means that both of those variables will be in an "off-state".



(a) All rules are active



(b) Rule r_1 is inactive (c) Rules r_3 and r_4 are inactive

Figure 2: Dependency graphs

The division on active sub-graphs allows to dynamically keep track of independent parts of the problem. Instead of trying to optimize the whole problem after each change, only the affected active sub-graphs will be optimized. Each such a sub-graph is of much smaller size than the original problem, which helps to reduce real-time computation efforts.

B. Algorithm

Algorithm 1 presents the Rule Maintenance's reaction when a change to uncontrollable variable v_u is detected. For all rules from a ruleset R^*, whose uncontrollable part F_u depends on a variable v_u, the system updates the status of the rule ($active$ vs $inactive$) based on the new information, and if the status is changed (from $active$ to $inactive$, or from $inactive$ to $active$), the rule is marked as $changed$.

While exists a $changed$, but not yet $checked$ rule r, the system finds an active sub-graph for this rule, and marks all rules in this sub-graph as $checked$. If there is at least one $changed$ rule in this active sub-graph that is $inactive$, or $active$ and $unsatisfied$, the optimization task is invoked with all controllable variables from active sub-graph added as variables to the optimization task; and all controllable parts of rules from active sub-graph added as constraints.

When the new optimal state of controllable variables from this sub-graph are found, the system checks if the new state differs from the old one and it generates an action to move to a new state. The action is then sent to the Orchestrator.

Algorithm 1 Rule Maintenance event processing

1: **function** processChange (v_u)
2: **for all** $rule \leftarrow R^*$ s.t. $rule.F_u$ depends on v_u **do**
3: Update $rule$ status
4: Mark $rule$ as $changed$ if status is changed
5: **end for**
6: **while** $\exists rule \in R^*$ s.t. $rule$ is $changed \wedge \neg checked$ **do**
7: $sg \leftarrow findActiveSubGraph(rule)$
8: Mark all $r \in sg.rules$ as $checked$
9: **if** $\exists r \in sg.rules$ s.t. r is $changed \wedge (inactive \vee false)$ **then**
10: $newstate \leftarrow Optimize(sg)$
11: **for all** $v_c \in sg.vars$ s.t. $v_c \neq newstate.v_c$ **do**
12: createAction(v_c,$newstate.v_c$)
13: **end for**
14: **end if**
15: **end while**

V. Related work

Many researchers are interested in decreasing energy consumption and other costs as a part of home automation.

Some studies propose to increase awareness among occupants of a house about their energy consumption [3]. A research done by Chetty et al. [4] presents a qualitative study of 15 households, and their energy consumption practices. The study shows that householders feel a need to have better tools to monitor their day-to-day energy consumption. A hands-on study of a system with such a tool intergrated [5] shows that just adding awareness about energy consumption patterns already helps users to manually reduce the energy costs. Several industrial solutions [6], [7] are already available to help people closely monitor their energy consumption.

A study by Cheong et al. [8] goes further and introduces a system that automatically deals with energy reduction in smart homes. They do it by creating an extensive house ontology, and by adding rules for situations where energy can be saved. As an example, one of the rules states that the lighting in a room where all people are asleep should be turned off. On the other hand, we try to unify the cost and energy saving patterns so to eliminate the need to manually create rules for each possible energy saving situation.

Several studies [9]–[11] introduce prediction models that help to reduce the costs and energy by predicting further usage of devices and turning them off if the prediction tells they probably will not be used. We use stricter approach, and always try to keep devices in the least costly state, unless there is a rule that forbids us to do so.

VI. Conclusions

When creating a system for home automation, the task of energy and other costs reduction is a very important integral part, and should not be ignored. In this paper we introduce the approach based on rule maintenance using dependency graphs, which helps to dynamically keep information about what rules are responsible for keeping a device in an expensive state. The Rule Maintenance engine also issues commands to return to a more cost efficient state whenever possible without violating any of user defined rules. Among the advantages of the dependency graph is the ability to keep track of active rules that should be handled by the system, and which are already satisfied in the current situation. It also allows to recheck only parts of the system that are affected by a change in the environment, reducing the computational costs of the cost optimization task.

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.

References

[1] E. Kaldeli, E. Warriach, J. Bresser, A. Lazovik, and M. Aiello, "Interoperation, Composition and Simulation of Services at Home," in *ICSOC*, vol. 6470, 2010, pp. 167–181.

[2] "GreenerBuildings," http://www.greenerbuildings.eu/, 2011.

[3] S. Darby, "The effectiveness of feedback on energy consumption. a review for DEFRA of the literature on metering, billing, and direct displays," 2006.

[4] M. Chetty, D. Tran, and R. E. Grinter, "Getting to green: understanding resource consumption in the home," in *Proc. 10th Int. Conf. on Ubiquitous computing*, 2008, pp. 242–251.

[5] M. Jahn, M. Jentsch, C. Prause, F. Pramudianto, A. Al-Akkad, and R. Reiners, "The energy aware smart home," in *5th Int. Conf. on Future Information Technology*, 2010, pp. 1–8.

[6] "Greenbox," http://getgreenbox.com/, 2011.

[7] "Plugwise," http://www.plugwise.com/, 2011.

[8] Y.-G. Cheong, Y.-J. Kim, S. Y. Yoo, H. Lee, S. Lee, S. C. Chae, and H.-J. Choi, "An ontology-based reasoning approach towards energy-aware smart homes," in *IEEE Consumer Communications and Networking Conf.*, 2011, pp. 850–854.

[9] C. Harris and V. Cahill, "Exploiting user behaviour for context-aware power management," in *IEEE International Conference on Wireless And Mobile Computing, Networking And Communications (WiMob)*, vol. 4, 2005, pp. 122–130.

[10] A. Roy, S. Das Bhaumik, A. Bhattacharya, K. Basu, D. Cook, and S. Das, "Location aware resource management in smart homes," in *Proc. 1st IEEE Int. Conf. on Pervasive Computing and Communications (PerCom)*, 2003, pp. 481–488.

[11] M. C. Mozer, "The neural network house: An environment that adapts to its inhabitants," in *Proceedings of the American Association for Artificial Intelligence (AAAI) Spring Symposium on Intelligent Environments*, 1998, pp. 110–114.