Human-Assisted Rule Satisfaction in Partially Observable Environments

Viktoriya Degeler and Edward Curry
Insight Centre for Data Analytics
National University of Ireland, Galway, Ireland
Email: vdegeler@gmail.com, ed.curry@insight-centre.org

***Abstract*—Many lightweight installations of smart environment systems do not have complex and expensive sensing and actuating capabilities, leaving parts of the environment unobservable to the system. This limits reasoning and decision making complexity of such systems. A decision support system that can collaborate with human users alleviates this problem by asking users to provide missing pieces of information or to perform actuations of which the system itself is incapable. In this paper we present a smart system that uses declarative rules to describe the expected behavior of the environment. In any situation the system aims to satisfy the rules by finding the actions to transform the environment state to conform to existing restrictions. The system asks users to provide missing information that is relevant to the final decision or to perform required actions. A decision tree is constructed, which defines the actions depending on user's answers. The system constructs it in such a way to minimize the expected efforts of users. We present two ways of constructing such a decision tree. One uses backtracking for optimal results, and the other uses a heuristic approach for faster decision tree creation. We show that the relatively small drop in efficiency allows most smart environments to use the fast heuristic algorithm for decision tree construction.**

I. Introduction

Modern smart environments are characterised by high awareness and high autonomous reasoning capabilities to react to any changes and events that the environment may experience. An environment may have numerous sensors that allow it to interpret the context of its current state. This information can be used for complex analysis, or for actuations that modify the environment according to some intelligent criteria [3]. Smart environments can operate according to certain predefined behavior rules. The rules can be either manually entered in advance, or learned automatically using machine learning techniques on previously gathered data. The rules describing the smart system's behavior can be quite complex structures. Rule representation can be easily extended in case more expressiveness is needed. In practice, the complexity of a smart environment is often limited not by the rules' expessiveness, but by the amount of context information that can be obtained from ubiquitous sensors in the environment.

Wide-scale adoption of smart environment systems is often hindered by the high costs of the initial system installation. Dedicated smart environment test sites often have numerous complex and expensive sensors that are fine-tuned for their particular location and are perfectly coordinated. Such setup allows to sense the environment in a very detailed and precise manner, but comes with high costs in terms of money, initial efforts to deploy devices, initial test data collection, etc. Therefore many sites that are initially interested in adding smart elements to their environment opt for installation of a handful of simple cheap non-invasive devices, such as light, motion, or temperature sensors, on-off actuators for electronic devices, etc. While these devices allow some degree of automation, the possibilities of complex reasoning remain very limited. Moreover, many parts of such environment remain only partially observable, further decreasing the amount of decisions that can be made based on the available information. Collaborative human-computer interaction and decision support systems are very useful in closing the gaps of information gathering and decision making restrictions of purely autonomous smart systems. Collaborative smart systems assume the possibility to interact with people in the process of making decisions, asking them to provide missing information or to perform tasks that the system cannot perform due to the lack of corresponding actuators [4]. If the state of unobservable variables influences the correct set of actions of the system, the system may need to have additional input from users in order to learn the current state of an unobservable variable. The system does this by asking targeted questions, i.e. "What is the current state of the window in Room 205?". A trivial solution is to ask questions about all unobservable variables. This gives the full knowledge, and the ability to reason as if the environment was fully observable. The practical implication is that it requires a huge amount of effort from users to give all these answers, even if most of them may be not relevant to the final decision. User disturbances must be limited. When a decision is made and certain actions should be performed in the environment, users may also need to be asked to perform those actions if there is no automated actuator to do the job. As in the case of asking questions, the aim of the system is not only to choose actions that allow satisfaction of all rules, but also those that are easy for people to perform and require the least amount of effort.

In this paper, we present an approach for an automated system to find solutions that satisfy environmental rules even if only partial information about the state of the environment is available. The system deals equally well with situations where an actuation can be performed by the system itself, situations where users' help is needed, situations where enough information is already available to produce final decisions, and situations that require additional questions to gather the required information [9].

II. Partially observable environments

Our approach to increase the reasoning and decision making capabilities of a smart system in a partially observable

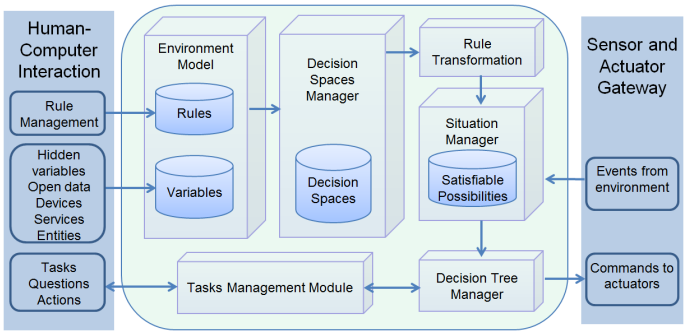


Fig. 1. Situation Awareness Decision Support Architecture

environment is to allow additional communication with human users and ask for their help in information gathering and actuations. The system has no means to automatically learn or infer the state of unobservable sensors other than explicitly asking people about their status. Yet such unobservable sensors can provide an important description of the environment and its current state, and in some cases the actual value of an unobservable variable may determine the correct set of actions to be performed. Observable status can change over time. For example, a system with a weather prediction module may report if it is raining outside. Due to most weather predictions being of probabilistic nature, the raining status variable is regarded as observable if the probability of rain is higher than 80% or lower than 20%. But the same variable can be regarded as unobservable if the probability is anywhere in between the brackets of 20%-80%. In this case the system cannot reliably know if it is raining or not, so a user enquiry is necessary. The state of an observable variable is always known, but an unobservable variable may nevertheless have its state observed, if a corresponding question was previously answered by users and the answer is still counted as valid.

Excessive questions and tasks can decrease users' willingness to participate. If users ignore questions and tasks, additional reasoning will not be of any help to the system. Extensive studies of user engagement techniques for smart environment projects are presented in [16] and are compatible with our approach. Our system has a notion of a cost associated with every user task and representing expected user efforts. For example, if a question does not require leaving a working desk, the cost of this question can be small. Automated actuation may have a cost of zero, as it requires no user efforts at all. Actuation that requires going to another room or performing a time-consuming activity can have a considerable cost.

The system constructs a decision tree to specify user tasks and their order. Depending on the answer a user gives, different actions may be performed or more questions asked until the system has all the required information to verify full rule satisfaction. The system aims to minimize the expected cost for users. We will show two approaches to decision tree construction. One uses backtracking during search, which allows to construct optimal decision tree with minimal user efforts. The other uses a greedy heuristic for choosing user tasks, and never revisits a choice that is already made. This allows to construct the decision tree considerably faster, but can decrease its final efficiency.

III. System Architecture

The system architecture is presented in Figure 1. The system has two main communication interfaces: the user interface (HCI) to interact with human users, and the gateway to communicate with devices, both sensors and actuators.

Initial information about the environment comes from the Environment Model database. The full set of variables describes all aspects of the environment, and a valuation of these variables can unambiguously represent any environment state. Variables are of different types, they can include physical devices; virtual and physical services, an example of which can be a "mean room temperature" variable that encloses complex calculations based on several temperature sensors from different parts of the room; high-level entities, such as "a person's activity", which can be discovered by activity recognition mechanisms; open data available from web services, such as current weather, or current price of energy; etc. All these variables may also come as unobservable variables, in case there is no automated way to know their current state.

Logical rules govern the dependencies between variables and constraints over their states. Ultimately, these rules define the actions to be performed at any moment and every time a situation changes. For a usual smart system, most of the rules are entered as a predefined template to reduce manual load, but users are able to further personalize rules. The main goal is to keep all rules satisfied at all times. If at some point rules are unsatisfiable, a responsible person is informed, and the rules that form a conflict are shown. It is important to note that rules in propositional logic naturally provide declarative description of desired environment states, instead of imperative one. This means rules define the desired final state of the environment, but the exact actions to be performed are not defined, as they may vary considerably depending on the original state.

Changes to the Environment Model are propagated to the Decision Spaces Manager. This module splits variables and rules into independent subsets, named *decision spaces*. Further calculations are only performed within a single decision space at a time. Due to natural clusterization of variables and rules within smart environments (such as a cluster of interconnected variables and rules within a single room), splitting into decision spaces allows for much greater scalability of a solution. Within every decision space, rules undergo transformations as specified in the Rule Transformations module. All rules are combined and transformed into a disjunctive normal form (DNF), so that every conjunction of a DNF constitutes one possible alternative to satisfy all rules of the environment. This transformation happens only once when the environment model is changed, so there is no real-time computation cost associated with this rules transformation.

The transformed rules in a form of satisfiable alternatives are stored in the Situation Manager module. This module deals with the current situation, including the latest readings from sensors and information available from human users via questions they answer and tasks they perform. The Situation Manager uses the satisfiable alternatives provided by the Rules Transformation module and checks which of them can be potentially satisfied in the current situation, given the information about the current state of the environment. In case when available information is not enough and one or more

user tasks (questions or actions) are required from human users, the Situation Manager sends the alternatives including the potential questions to the Decision Tree Manager. The latter is responsible for creating a decision tree of questions and actions such that when the user answers them the system can be sure that all rules are definitely satisfied.

Possible tasks of the decision tree include questions to users about the current state of unobservable variables, requests to users to perform certain actions, as well as actions that can be performed automatically by the system if it has access to a corresponding actuator. If the task is an automated actuation, it is sent directly to the Sensor and Actuator Gateway, which in turn sends the required low-level commands to the corresponding devices. The confirmation of an action is sent back to the Decision Tree manager so that it can proceed. If a task requires human involvement, it is sent to the Tasks Management Module. The module distributes the tasks among human users taking into account their availability, aptitude and attitude towards this particular task, their current location, etc. The person that it deems the best for performing the task receives a notification via email, Twitter, and/or dedicated web application and has an opportunity to perform the task or ignore/reject it, in which case it will be sent to the next person. The details of the Task Management Module assignment mechanisms are fully described in [8].

A. Environment Model

An environment $E = \langle V, \gamma, R \rangle$ is defined by a set of context variables V, a known valuation of these variables γ and a set of behavior rules R.

Variables are split into two subsets $V = S \bigcup A$; $S \bigcap A = \emptyset$, where $S = \{s_1, s_2, ..., s_{n_s}\}$ is a set of uncontrollable variables, and $A = \{a_1, a_2, ..., a_{n_a}\}$ is a set of controllable variables. Every variable $v \in V$ has associated finite *states domain* D_v, which represents a set of values $D_v = \{d_{v1}, d_{v2}, ..., d_{vk_v}\}$ that the variable can take. Every value out of the variable's domain has a probability of being active, denoted as $pr(d_v)$. This probability can be calculated from previous sensor logs. If no prior information is available, a uniform distribution can be used, giving equal probability to every variable's value until further data collection. It is possible for a variable to have an infinite range of integer or real values, however, in this case a preprocessing step is done to split all values on sets of values that correspond to the same valuation of corresponding rules. For example, if rules contain two simple clauses for an integer variable v_i: $v_i < 0$ and $v_i > 5$, another variable v_i' will be created, that has three possible states, schematically defined as "< 0", "[0..5]", "> 5". This variable with three states will be used in further calculations instead of the original one.

Uncontrollable variables, or sensors, S change their state due to factors external to the system and cannot be directly influenced by the system. These variables do not necessarily represent a physical sensor, they can represent a function of a combination of several sensors, or a certain high-level entity, such as a person's activity type, which can be inferred from an activity recognition task. On the other hand, controllable variables A can be seen as actuators. This includes automated devices that the system can control, appropriate commands from the system, as well as devices that the system cannot directly affect, but that can be actuated by people, when they are asked to do so by 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 domain state to any other domain state.

Variables can be either directly *observable*, so that the system always knows their current status, or *unobservable*. We use the notation $K(v)$ to define the observable property of a variable v, where $K(v) = true$ means a variable is observable. We also define $k(v)$ to represent the observed status of a variable v. $k(v) = true$ if the state of v is known, and $k(v) = false$ if the state is unknown. Note that $K(v) \Rightarrow k(v)$.

At every moment of time every variable is in one of the states out of its respective domain D_v. The system always knows the state of observable variables $K(v)$. The system also knows the state of some unobservable variables, after completion of respective user task (a question about an unobservable sensor or an action with an unobservable actuator) and before the knowledge gained from this task becomes obsolete. Known values of variables for a certain situation constitute a known valuation of variables γ, where

$$\gamma(v) = \begin{cases} d(v) \in D_v & \text{if } k(v) \\ undefined & \text{if } \neg k(v) \end{cases}$$

A valuation of variables is most useful when assessing the current situation. Also, when constructing a decision tree the system models situations with more information gained from user tasks. In this case, valuations under consideration are similar to the current valuation of variables, but contain additional information gained from answers given by users.

The system decides its course of actions based on the context information about the current environment and according to certain rules of the system's behavior. These rules can be entered manually [6], or learned from previous data logs [7].

There are two different types of rules that represent the difference between what is *necessary* and what is *desirable*. The first type represents a *dependency between variables*. For example, a rule $\neg(room1.blinds1 = down \wedge room1.window1 = open)$ represents a physical constraint that blinds can only be put into the 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 predicate logic over finite domains. Every atomic predicate $P(v)$ is a function of a state of a variable that represents a certain condition over this variable with respect to a subset of its values and should result in $true$ or $false$. Examples of the most commonly used simple predicate functions are equality $room1.dimmer = 0$, inequalities $room313.dimmer1 \neq 10$ or $room313.dimmer1 \geq 10$, subset $room1.dimmer \in \{0; 10; 20\}$, etc. Atomic predicates can be combined together to form logical formulas of any additional complexity, using the standard logical operators:

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

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 Constraint Satisfaction

Problem (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 obtained by constructing the full truth table for a set of variables V_r, and retaining only those values from the table, for which the rule evaluates to $true$.

B. Decision Spaces

A decision variable may require a user action to be performed in order for the system to satisfy the environment.

Definition 1 (Decision variable): A variable $v \in V$ is a *decision variable*, defined by $B(v)$, if it is either unobservable or an actuator: $B(v) = true \Leftrightarrow \neg K(v) \vee v \in A$.

We assign a cost $c(v)$ to each decision variable, to quantify the amount of estimated user efforts to get an answer to a question or to perform an actuation.

Smart environments of even moderate sizes may contain hundreds of sensors. With unobservable variables this number grows even bigger. This makes it practically impossible to check the state of all variables after every detected event, especially considering the dynamicity of a typical environment, with possibly many events per second. When the system is making a decision on which questions to ask users, we aim to only take into consideration variables that are affected by the latest event. Also we only aim to use rules that affect these variables and that are relevant to the current situation.

In order to understand which rules are relevant to a decision to be made, all variables and rules are split in as many as possible independent *decision spaces*. Informally, a decision space is a combination of mutually dependent decision variables with rules that create their dependency. Rules that contain the same decision variable are always a part of the same decision space.

Definition 2 (Decision space): A *decision space* is a tuple $DS_i = \langle V_i, R_{oi} \rangle$ of a set of variables and a set of rules, s.t.

- Every variable is a decision variable: $\forall v \,:\, T(v)$;
- Every rule contains only decision variables from set V_i: $\forall r \in R_{oi},\ \forall P(v) \in r \,:\, v \in V_i$.
- It is impossible to split the decision space into independent sub-spaces:
  $\nexists V_1, V_2 \subset V_i,\ R_1, R_2 \subset R_{oi}$, s.t.:
  1) $V_1 \neq \emptyset,\ V_2 \neq \emptyset,\ V_1 \cap V_2 = \emptyset,\ R_1 \cap R_2 = \emptyset$;
  2) $\forall r \in R_1, \forall P(v) \in r \,:\, v \in V_1$;
     $\forall r \in R_2, \forall P(v) \in r \,:\, v \in V_2$;
  3) $\forall r \in R_{oi} \backslash R_1, \forall P(v) \in r \,:\, v \notin V_1$;
     $\forall r \in R_{oi} \backslash R_2, \forall P(v) \in r \,:\, v \notin V_2$.

Every environment has its variables and rules split into a set of decision spaces. We define this set as follows:

Definition 3 (Decision space set): The *decision space set* for an environment $E = \langle V, \gamma, R_o \rangle$ is a set of decision spaces $DSS = \{DS_i(V_i, R_{oi})\}$ such that:

- $\forall i \neq j \,:\, V_i \cap V_j = \emptyset,\ R_i \cap R_j = \emptyset$
- $\bigcup_i (V_i) = V,\ \bigcup_i (R_{oi}) = R_o$
- No rules from one decision space can contain variables that are a part of another decision space: $\nexists j \neq i \,:\, \exists r \in R_{oj}, \exists P(v) \in r \,:\, v \in V_i$.

Every decision variable of the environment corresponds to one and only one decision space. However, changes to values of observable sensor variables may affect several decision spaces at once. When some observable sensor variable changes its value, the system checks every affected decision space to establish if there is a change of status of any atomic clause that contains this variable. If at least one atomic clause has changed its status, the decision tree for this decision space is reconstructed, while keeping and reusing the available information from previous non-obsolete answers, if such exist.

C. Rules Transformation

Rules, as well as variables, can be added or removed from the system dynamically, without the need for a full restart. The initial form of rules can be very diverse, and the same rule can be expressed in different forms. Therefore the system transforms all rules into a form that is the most suitable for further reasoning. It is done once when a rule is added or removed, therefore it does not increase the computational load during the normal course of the system's operation.

All further reasoning is done for every decision space $DS = \langle V, R_o \rangle$ separately. First of all, the set of rules R_o is transformed into a disjunctive normal form (DNF). In order to perform this transformation all rules are combined into a single rule by the AND-clause: $r_{combined} = \bigwedge_i R_{oi}$. It is possible, because this does not change the final decision to be made, since all rules must be satisfied at the same time. After a single rule is created, standard transformation to DNF is performed. After the transformation, the rule is presented in a form $r_{combined} = \bigvee_i \bigwedge_j P'_{ij}(v_{ij})$. Here $P'_{ij}(v_{ij})$ is either an atomic predicate $P(v_{ij})$ or a negated atomic predicate $\neg P(v_{ij})$. The rule is also simplified by removing any conjunctions that necessarily result to $false$. If any conjunction $\bigwedge_j P'_{ij}(v_{ij})$ contains several atomic predicates with the same variable, these predicates are simplified as well and are combined by replacing them with a single predicate that only allows states that are an intersection of allowed states for all original predicates: $P'_{new}(v) = P'_1(v) \wedge P'_2(v) \wedge P'_3(v)$.

The DNF representation is the easiest form to work with due to several benefits. Among all conjunctions of a DNF rule, only a single one should be satisfied in order for a full rule to be satisfied. Therefore we can regard each conjuction of a combined rule as a separate satisfiable alternative. We denote this set of conjunctions as $R^* = \{\bigwedge_j P'_{ij}(v_{ij})\}$. Note that every conjunction represents a set of restrictions over variables, where each variable has restrictions that are independent from another variable. We denote as $\delta(v, r) \subseteq D_v$ a set of states that are allowed by the rule $r \in R^*$. We also use the notation $\delta(v, P'(v))$, where $P'(v)$ is a single atomic clause that depends on variable v and may be a part of rule r. Given the transformation as described above, the question of rule satisfaction is transformed into the question of finding a conjunction that can be satisfied, which in turn coincides with the question of finding an alternative in which all variables are in a state that is within the allowed set of states $\delta(v, r)$, not restricted by predicates of a conjunction r.

While sensor variables must be apriori in this set at the time of searching for a solution, actuator variables may be outside of the required set, as this does not prevent the alternative from being satisfiable. In this case the corresponding actions to set an actuator to a state specified by a satisfiable alternative will be created. Another benefit of the DNF representation is that it makes it easy to calculate the cost estimation heuristic for the expected cost that should be spent to make sure all rules are satisfied.

Example. Assume the system has five variables. Three of them are sensors: presence of people in a room (PR), relative humidity inside (H), and whether it is raining outside or not (R). And two variables are actuators: an air conditioner (AC), and a window (W). The system also has two behavior rules. The first one states that if humidity is high while someone is in the room, either an air conditioner should be turned on, or a window should be opened, but only if it is not raining outside: $H > 80 \wedge PR = T \Rightarrow AC = T \vee R = F \wedge W = T$. The second rule adds a constraint over the air conditioner and the window, the AC cannot be turned on if the window is open: $\neg(AC = T \wedge W = T)$. After two rules are combined with the $\wedge$-clause, transformed into DNF and simplified, the resulting six disjuncted clauses are shown in the first column in Table I.

TABLE I. EXAMPLE TRANSFORMED RULES WITH EXPECTED SATISFACTION COSTS. ORIGINAL RULES: $H > 80 \wedge PR = T \Rightarrow AC = on \vee \neg R \wedge W = open$ AND $\neg(AC = on \wedge W = open)$.

Cost:	2	1	2	7	4	**Total**
Transformed Rules	**H**	**PR**	**R**	**AC**	**W**	**Cost**
$PR \neq T \wedge W \neq T$		F			F	5
$PR \neq T \wedge AC \neq T$		F		F		8
$H \leq 80 \wedge W \neq T$	≤ 80				F	6
$H \leq 80 \wedge AC \neq T$	≤ 80			F		9
$AC = T \wedge W \neq T$				T	F	11
$R = F \wedge AC \neq T \wedge W = T$			F	F	T	13

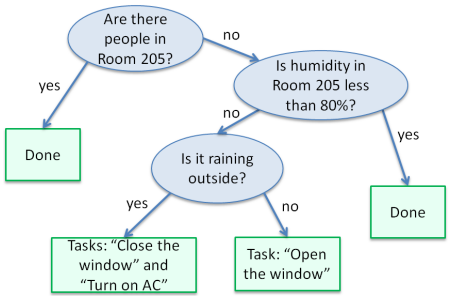


Fig. 2. Decision tree example for Table I

D. Situation Manager

After rules are transformed into a set of alternatives, each of which can potentially satisfy all original rules, the main question is: which alternative should be chosen at every moment? The main restriction is that the alternative must be satisfiable, in other words, every atomic predicate of an alternative must result to $true$. This means the current state of the predicate's variable must be in the set of states, allowed by this predicate. If the variable is unobserved, a question about its current status must be asked in order for the system to know the status of this atomic predicate. Every question about the current state of an unobservable variable may potentially rule out some alternatives (in case the current state of the variable in question does not correspond to a variable restriction specified in these alternatives), with different answers ruling out different alternatives. Therefore all questions must be chosen in such a way that there is a follow up choice of actions for every possible answer to the question, so the final actions would result in a satisfaction of at least one satisfiable alternative.

This is represented as a decision tree, where every node corresponds to a user task, a question or an action. A node branches to several children depending on the answer given, i.e. directed edges represent possible variable states. We construct a decision tree in such a way to minimize the expected user efforts, as quantified by the cost of a decision variable $c(v)$. Decision tree is represented by a tuple $DT = \langle N, E \rangle$, where N is a set of decision nodes, and E is a set of directed edges. Decision node $n \in N$ can be in one of three forms:

1) $Q(v)$ iff $v \in S \ \wedge \ \neg K(v)$ - a question. This type of node is only applicable to unobservable sensor variables and represents a question to human users about the current state of a variable. No changes to the environment are performed during the execution of this task.

2) $T(\{v_i, d_{v_i}\})$ iff $\forall v_i \in A$ - a set of actions. This type of node represents a set of actions to be performed in the environment. It is only applicable to actuators, both observable and unobservable. For automated actuators, the action can be performed by the system directly. For actuators that require a user's action, a corresponding task is created. This node requires changes to the environment to be made.

3) $Done$ is an alias for $T(\{\})$, an empty set of actions.

$Q(v)$ is a *non-leaf* node, and for any path from the root to any leaf of the tree, all nodes except the last one are questions. $T(\{v_i, d_{v_i}\})$ and $Done$ are *leaf* nodes. They are necessarily located at the bottom of the tree, and any path of the tree has one and only one action node at the end.

Decision edge $e = \langle d(v_i) \rightarrow N \rangle \in E$ represents a path to be taken after a question about the state of variable v_i is answered to be $d(v_i)$. In this case, N is the next node (question or actions) to be executed.

When constructing the decision tree we use a top-down approach. We start with available situation, and check possible questions or actions to be taken at this moment. If a set of actions is chosen, construction of this path is completed. But if a question is chosen, a question node gets expanded and the algorithm recursively constructs a tree for every possible answer, taking into account the newly obtained information.

We explore two algorithms to construct a decision tree. The first one is a heuristic algorithm that chooses a variable with the best cost estimation and immediately expands it. Following a greedy approach, the algorithm never revisits previous choices, therefore it may produce a sub-optimal decision tree. The second algorithm expands the most promising node with the best cost estimation, but after calculating the actual cost, it will try to expand the second best node, if its estimation is lower than the actual cost of the first one. The backtracking capability of the second algorithm guarantees the optimality of a solution as long as the chosen cost heuristic is proven to be admissible, but increases the computational costs considerably. The actual

performance on site allows people to choose the algorithm they want to use in their smart home system.

The transformation of rules into a set of satisfiable alternatives makes it easy to calculate the cost estimation heuristic for every possible choice of a decision variable. The cost represents the expected minimum amount of effort that should be spent in order to guarantee the satisfaction of all rules, i.e. the definite satisfaction of at least one of the alternatives specified by the DNF form. The idea behind the cost estimation calculation is the following. In order to verify that one of the alternatives is satisfied, we must know the actual values of all unobservable variables that have restrictions in this alternative, in order to ensure the restrictions hold. And we must perform actions for all actuators that are not in the state required by this alternative. Therefore, the bare minimum of effort for this alternative to be satisfied is the sum of all questions for unknown unobservable variables and all actuators that are not in the required state. This value can easily be calculated, but while it represents the minimum expected cost, during the expansion other variables may be asked about as well, therefore increasing the actual cost. For example, in an environment as described in Table I, the system may want to create a question about the presence of people (PR) in the room, hoping to satisfy the first alternative, as it is the cheapest. However, if the answer is that there are people inside, it immediately rules out the first alternative as a possible solution, therefore the system may try to satisfy the third one. The minimum cost to satisfy the third alternative is 6, which includes a question about the relative humidity (H) and an action of closing the window (W). But as soon as the question about the presence is asked, the total minimum cost of the third alternative goes up to 7, to include the cost of this question, even though it is irrelevant to the third alternative.

There is one more way to ensure the satisfaction of at least one alternative, in case it is guaranteed that original rules are necessarily satisfiable, i.e. for every possible combination of sensor values there is a combination of actuator states that satisfies all rules. If this ability is ensured, in any situation it is possible to consider only actuators, and try to find a set of actions that will satisfy the actuators part of all alternatives. No questions about unobservable sensors need to be asked, because we know that at least one of the alternatives is necessarily satisfiable, i.e. the sensor part definitely holds. If the actuator part holds for all alternatives, it also holds for the alternative with satisfied sensors, therefore we have a fully satisfied alternative. Such a solution usually requires extra actions, but avoids asking any questions about sensor variables. Therefore it may be optimal in case there are many questions, or they require more efforts than extra actions to be performed.

It can easily be shown that the cost heuristic is *admissible*: (i) the cost heuristic is always less than or equal to the actual cost of an alternative, because at least the relevant questions definitely need to be asked; and (ii) it is always monotone, because during the expansion process the cost may increase due to irrelevant questions being asked, but will not decrease. Given the admissibility of the heuristic, we can be sure that the backtracking algorithm returns the optimal solution.

We now present the algorithm to calculate the expected cost of a certain situation. Algorithm 1 uses the known valuation of the environment to retain only those rules that can still be satisfied. Algorithm 2 shows the calculation of the minimum cost for a currently available solution. The currently available solution is the one with no uncertainty, i.e. no questions need to be asked, only actuations may be performed. If there are no immediately available solutions, the algorithm will return ∞, otherwise it will return a set of actions to be performed and their cost. Algorithm 3 builds up on the previous algorithm, and calculates the expected cost heuristic for a solution even in case there is no currently available one. Finally, Algorithm 4 presents the full process of decision tree construction. Figure 2 shows a decision tree constructed for an example in Table I.

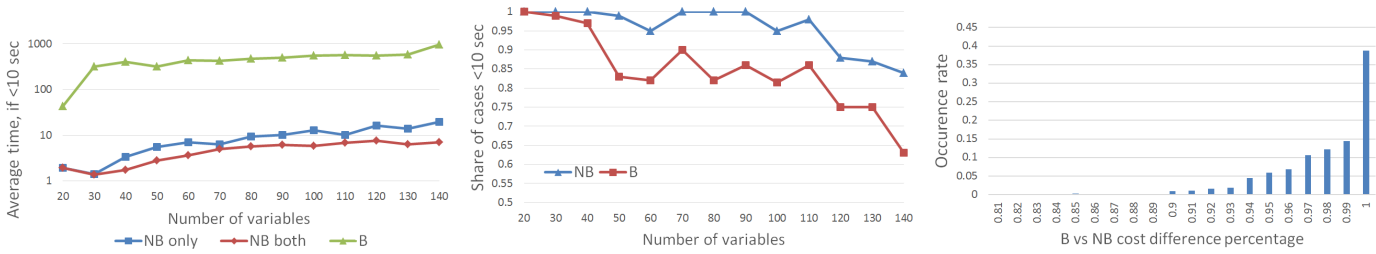
Algorithm 1 Retain only potentially satisfiable alternatives

1: **function** Rsat (E - environment)
2: $Rsat \leftarrow \{r \in E.R^*$ iff $\forall P'(v) \in r$ s.t. $v \in S \ : \neg k(v) \vee \gamma(v) \in \delta(v, r)\}$
3: **return** $Rsat$

IV. Performance Evaluation

We performed several experiments to evaluate the effectiveness of our approach. The system was written in Scala, and run on Windows 8.1 x64, Intel Core i7-4702MQ @ 2.2GHz, 8 Gb RAM machine. As a baseline we use randomly constructed environment instances. All variables are split into three equal subsets: observable sensors, unobservable sensors and actuators. Note that observability of actuators is not relevant for the purpose of decision tree construction. If the state of an actuator is unknown, it is treated in the same way as if the state is not within required bounds, so an action on this actuator is always required if there is any restriction associated with it. All variables have boolean domains, which does not lead to loss of generality, because any environment with variables of arbitrary domains can be converted into an equivalent environment that has a boolean variable for every possible domain value of original ones. Rules are created randomly and are constructed in such a way so that they are always satisfiable, i.e. for any possible combination of sensor values there is a state of actuators that satisfies all rules. For every number of variables in test, we created 10 different environments, and for every environment we created 10 random starting situations. Therefore for each number of variables the decision tree construction algorithm ran 100 times. For every situation we ran two decision tree construction algorithm variations, one with backtracking capability (B) and one without backtracking (NB). We noted the time that was needed to find a solution, and the cost of the found solution. We set the deadline of 10 seconds for the algorithm to find a solution. If the solution was not found after 10 seconds, the instance was stopped and the failure was noted.

Figure 3a shows the average time to find a solution. It includes cases that were finished before the 10 seconds deadline. The time is shown for all cases finished on time using the algorithm with backtracking (B); all cases finished on time using the algorithm without backtracking (NB only); and cases using the algorithm without backtracking, but only if the backtracking algorithm also finished on time (NB both). The last line is included for fair comparison with the backtracking algorithm, exactly the same cases are included to "NB both" and "B". As expected, there were no cases where the backtracking algorithm finished on time, but the algorithm without



(a) Average time to construct a decision tree, ms (b) Percent of cases, finished in less than 10 s (c) Occurence rate of B vs NB cost difference

Fig. 3. Running time and solution performance comparison of algorithms with backtracking (B) and without backtracking (NB)

Algorithm 2 Minimum cost of currently available solution

function costCurrent (E - environment)
if $Rsat(E) = \emptyset$ **then return** ∞ **end if**
$totalActs \leftarrow \emptyset$; $totalCost \leftarrow 0$
$singleActs \leftarrow \emptyset$; $singleCost \leftarrow \infty$
for all $r \in Rsat(E)$ **do**
  if $\nexists r.v \in S : \neg k(v)$ **then**
    $acts \leftarrow \{\forall v \; s.t. \; E.\gamma(v) \notin \delta(v,r) : \langle v, \delta(v,r)\rangle\}$
    $cost \leftarrow \sum_{act \in acts} c(act.v)$
    if $cost < singleCost$ **then**
      $singleActs \leftarrow acts$; $singleCost \leftarrow cost$
    end if
  else if $totalCost < \infty$ **then**
    for all $a \in r.v : \; v \in A$ **do**
      if $totalActs.contain(a)$ **then**
        Find $\langle a, d_{old}\rangle$ in $totalActs$
        $totalActs \leftarrow totalActs \backslash \langle a, d_{old}\rangle$
      else
        $d_{old} \leftarrow D_v$
        $totalCost \leftarrow totalCost + c(a)$
      end if
      $d_{remain} \leftarrow d_{old} \cap \delta(a,r)$
      $totalActs \leftarrow totalActs \cup \langle a, d_{remain}\rangle$
      if $d_{remain} = \emptyset$ **then** $totalCost \leftarrow \infty$ **end if**
    end for
  end if
end for
if $singleCost = \infty \wedge totalCost = \infty$ **then**
  return ∞
else if $singleCost < \infty$ **then**
  return $\langle singleActs, singleCost\rangle$
else
  return $\langle totalActs, totalCost\rangle$
end if

Algorithm 3 Calculation of the expected cost heuristic

function costHeuristic (E - environment)
$minCost \leftarrow costCurrent(E)$
for all $r \leftarrow Rsat(E)$ **do**
  $cost \leftarrow 0$
  for all $v \in E.V$ **do**
    if $v \in S \wedge \neg k(v) \vee v \in A \wedge E.\gamma(v) \notin \delta(v,r)$ **then**
      $cost \leftarrow cost + c(v)$
    end if
  end for
  $minCost \leftarrow min(cost, minCost)$
end for
return $minCost$

Algorithm 4 Decision tree construction

function decisionTree (E - environment)
$(actions, cost) \leftarrow currentCost(E)$
$queue \leftarrow \emptyset$; priority queue $\langle cost, variable\rangle$
for all $v \in S \; : \; \neg k(v)$ **do**
  $expCost \leftarrow c(v)$
  for all $d_v \leftarrow D_v$ **do**
    $E_{new} \leftarrow \langle E.V; E.\gamma + (v \rightarrow d_v); E.R\rangle$
    $cost \leftarrow costHeuristic(E_{new})$
    $expCost \leftarrow expCost + pr(d_v) * cost$
  end for
  $queue.add(\langle expCost, v\rangle)$
end for
$bestDecision \leftarrow null$; $bestCost \leftarrow \infty$
while $queue \neq \emptyset \wedge queue.head.cost < bestCost$ **do**
  $\langle expCost, v\rangle \leftarrow queue.pop$
  $actualCost \leftarrow c(v)$; $decisions \leftarrow \emptyset$
  for all $d_v \leftarrow D_v$ **do**
    $E_{new} \leftarrow \langle E.V; E.\gamma + (v \rightarrow d_v); E.R\rangle$
    $\langle cost, dec\rangle \leftarrow decisionTree(E_{new})$
    $actualCost \leftarrow actualCost + pr(d_v) * cost$
    $decisions \leftarrow decisions \cup (d_v \rightarrow dec)$
  end for
  if $actualCost < bestCost$ **then**
    $bestCost \leftarrow actualCost$
    $bestDecision \leftarrow (v \rightarrow decisions)$
  end if
  if (no backtracking) **then** Empty $queue$ **end if**
end while
return $\langle bestCost, bestDecision\rangle$

backtracking did not. Figure 3b shows the percentage of cases that finished within the 10 seconds deadline.

It can be seen that the algorithm without backtracking (NB) requires much less time on average than the one with backtracking (B). However, the NB algorithm is suboptimal, while the B algorithm gives the optimal solution. Therefore we must compare found solutions, in order to understand the trade-off in effectiveness versus time. For every case when both algorithms finished before the deadline, we recorded the cost of the found decision tree. The calculated difference between solutions found by the NB algorithm and optimal solutions is shown in Figure 3c. It can be seen that in almost 40% of cases the heuristic algorithm with no backtracking still managed to find the optimal solution. 88% of cases were above 95% percent of effectiveness, compared to the optimal solution, and 99% of cases were above 90% of effectiveness.

There were no cases where the solution found by the NB algorithm was below 81% of effectiveness compared to the optimal one. In most smart environments, the optimality of a solution is not essential, but it is the general effectiveness of solutions that is important. Therefore these results allow us to conclude that it may be beneficial to use the algorithm without backtracking due to faster solution times, compared to relatively small increase of the final cost of found solutions. Other heuristics may be investigated to push solutions of non-backtracking algorithm closer to optimal values.

V. Related Work

Since the conception of smart environments research, many projects investigated different reasoning approaches to intelligent buildings automation [11]. Several studies propose constraint satisfaction techniques to solve reasoning problems. For example, multi-agent coordination in smart homes is modelled as a distributed constraint optimization problem in [12]. 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. In [2] a problem solving environment deals with complex scheduling problems, which are represented as constraints. They present O-OSCAR, a constraint based object-oriented scheduling framework. A constraint-based AI planner is used in [10] 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.

Many works deal with partial observability of environments. One approach includes modelling the events as Markov processes [1]. Such environments usually exclude the possibility to collect more information from users, therefore require the system itself to act and collect information by observing the environment transformations. A typical task for such partially observable environments is robot navigation [15].

Traditionally decision tree construction algorithms such as ID3 [13] and C4.5 [14] use machine learning approaches to find statistical correlations of numerous data points for long-term usage. In one such study [5], an automated prompting system for smart environments uses decision tree, constructed from previously gathered and annotated data, to guide participants who are asked to perform certain activities. If they perform wrong or irrelevant activities, a prompt is issued to guide them back on course. On the other hand, we investigate the possibility of real-time decision tree construction, based on existing constraints and current context information for knowledge gathering and automated decision making purposes.

VI. Conclusions

We presented a smart system that uses declarative rules to describe the desired behavior of a smart environment. The system operates equally well in fully observable environments, and in environments, where the inadequate number of sensors and actuators limits the capabilities of automated reasoning and decision making. By having an ability to communicate with users and ask them to provide additional information or to perform actions that cannot be performed automatically, the system has the ability to satisfy the expected restrictions even for complex partially observable situations. In our approach we minimize users' participation and their expected efforts to perform the tasks. The system constructs the decision tree of questions and actions to be performed, based on the given answers. We showed that it is possible to construct the optimal decision tree with minimum user efforts, or to use a heuristic which severely decreases the construction time, while keeping the efforts within 90% effectiveness close to the optimum.

Acknowledgment

This work has been supported in part by the Irish Research Council under the New Foundations Scheme and Enterprise Ireland, the National Development Plan and European Union under Grant Number IP/2012/0188.

References

[1] Anthony R Cassandra, Leslie Pack Kaelbling, and Michael L Littman. Acting optimally in partially observable stochastic domains. In *AAAI*, volume 94, pages 1023–1028, 1994.

[2] Amedeo Cesta, Gabriella Cortellessa, Angelo Oddi, Nicola Policella, and Angelo Susi. A constraint-based architecture for flexible support to activity scheduling. In *AI* IA 2001: Advances in Artificial Intelligence*, pages 369–381. Springer, 2001.

[3] Diane J Cook and Sajal K Das. How smart are our environments? an updated look at the state of the art. *Pervasive and mobile computing*, 3(2):53–73, 2007.

[4] David N Crowley, Edward Curry, and John G Breslin. Closing the loopfrom citizen sensing to citizen actuation. In *Digital Ecosystems and Technologies (DEST), 2013 7th IEEE International Conference on*, pages 108–113. IEEE, 2013.

[5] Barnan Das, Diane J Cook, Maureen Schmitter-Edgecombe, and Adriana M Seelye. Puck: an automated prompting system for smart environments: toward achieving automated prompting - challenges involved. *Personal and Ubiquitous Computing*, 16(7):859–873, 2012.

[6] Viktoriya Degeler and Alexander Lazovik. Dynamic constraint reasoning in smart environments. In *Tools with Artificial Intelligence (ICTAI)*, pages 167–174, Nov 2013.

[7] Viktoriya Degeler, Alexander Lazovik, Francesco Leotta, and Massimo Mecella. Itemset-based mining of constraints for enacting smart environments. In *Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 41–46. IEEE, 2014.

[8] Umair ul Hassan and Edward Curry. A capability requirements approach for predicting worker performance in crowdsourcing. In *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom)*, pages 429–437. IEEE, 2013.

[9] Umairul Hassan, Murilo Bassora, Ali H Vahid, Sean O'Riain, and Edward Curry. A collaborative approach for metadata management for internet of things: Linking micro tasks with physical objects. In *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom)*, pages 593–598. IEEE, 2013.

[10] Eirini Kaldeli, Ehsan Ullah Warriach, Alexander Lazovik, and Marco Aiello. Coordinating the web of services for a smart home. *ACM Transactions on the Web*, 2012.

[11] Tuan Anh Nguyen and Marco Aiello. Energy intelligent buildings based on user activity: A survey. *Energy and buildings*, 56:244–257, 2013.

[12] Federico Pecora and Amedeo Cesta. Dcop for smart homes: A case study. *Computational Intelligence*, 23(4):395–419, 2007.

[13] John Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[14] John Ross Quinlan. *C4.5: programs for machine learning*, volume 1. Morgan kaufmann, 1993.

[15] Reid Simmons and Sven Koenig. Probabilistic robot navigation in partially observable environments. In *IJCAI*, volume 95, pages 1080–1087, 1995.

[16] Ray Yun, Peter Scupelli, Azizan Aziz, and Vivian Loftness. Sustainability in the workplace: nine intervention techniques for behavior change. In *Persuasive Technology*, pages 253–265. Springer, 2013.