Research Article

Using Activity Recognition for Building Planning Action Models

Javier Ortiz, Angel García-Olaya, and Daniel Borrajo

Departamento de Informática, Universidad Carlos III de Madrid, Avenida de la Universidad, 28911 Leganés, Madrid, Spain

Correspondence should be addressed to Javier Ortiz; jortiz@inf.uc3m.es

Received 18 February 2013; Accepted 27 April 2013

Academic Editor: Javier Bajo

Copyright © 2013 Javier Ortiz et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Automated Planning has been successfully used in many domains like robotics or transportation logistics. However, building an action model is a difficult and time-consuming task even for domain experts. This paper presents a system, asra-amla, for automatically generating planning action models from sensor readings. Activity recognition is used to extract the actions that a user performs and the states produced by those actions. Then, the sequences of actions and states are used to infer a planning action model. With this approach, the system can automatically build an action model related to human-centered activities. It allows us to automatically build an assistance system for guiding humans to complete a task using Automated Planning. To test our approach, a new dataset from a kitchen domain has been generated. The tests performed show that our system is capable of extracting actions and states correctly from sensor time series and creating a planning domain used to guide a human to complete a task correctly.

1. Introduction

Activity recognition (AR) systems have been widely used in the past to detect activities of daily living (ADL) [1], but most of the literature describes approaches that detect the whole activity like in [2–6]. In order to provide assistance for the user to complete an activity or task, it would be desirable to detect the subtasks or actions (e.g., opening/closing cabinets) that compose an activity (e.g., preparing an omelette) and the effects that these actions have (omelette cooked). So, recognizing such actions and their effects while a user tries to accomplish a task could be used to check if the user is completing the task correctly. This way, a correct sequence of actions can be generated to accomplish the task successfully. In order to generate the sequence of actions, Automated Planning (AP) tools could be used, but they require to have an action model. Such model could be generated by experts manually, but usually this is a time-consuming and error-prone process.

Hence, the goal of this work is to build a system capable of guiding users while they are completing an activity. The system will be able to recognize, from sensor time series, the actions performed by a user while completing the activity, and the states of the system produced by those actions. Using that information, the proposed system generates a user action model represented as a STRIPS (Stanford Research Institute Problem Solver) planning domain in the standard language PDDL (Planning Domain Definition Language) [7]. Then, the generated planning domain will be used by an automated planner to generate plans (sequences of actions) to accomplish a task. These plans can be used to provide assistance to people in daily activities.

This paper describes a system, called asra-amla (Action and States Recognition Algorithm-Action Model Learning Algorithm), that automatically learns an user action model to assist users while they cook a recipe. Also, this action model could be used for plan recognition, as it is shown in [8].

In order to automatically generate the planning action model, the preconditions and effects of each action have to be learned from sensor readings. For that reason, the segmentation of the time series is a key issue since the typical method used, the temporal sliding windows, may overlap several actions. Instead, this work employs a method based on events which was first used in [9]. The events produced by changes in the environment, like the actions do, are used to split the sensor time series. A segmentation method based on events may produce better results for generating planning action models than a method based on fixed-length sliding-windows as it is...
shown in [9]. Once the sensors time series were segmented, two different models to recognize actions were compared employing different features in order to obtain the best classifier. We used six different machine learning algorithms in the experiments. The best model was then used to generate the sequences of actions employed to build the action model in PDDL.

The main contribution of this paper is the creation of a system capable of the following: first, recognizing user actions; second, automatically generating user action models, represented as a STRIPS planning domain in PDDL, using the recognized actions and the states of the sensors between the actions; finally, assisting users by providing a sequence of actions to accomplish a goal. In this work, an action user model is defined as a representation of the actions that user performs. Planning action models are user action models represented as a planning domain in PDDL.

The rest of the paper is organized as follows. In Section 2 a brief introduction to Automated Planning is given and in Section 3 the ASRA-AMLA system is described. Datasets and models generated for testing are presented in Section 4, whereas Section 5 discusses related work. Finally, the findings are summarised in Section 6.

2. Automated Planning

Since the main objective of this paper is to learn a planning domain, this work focused on classical Automated Planning (AP) (more specifically in STRIPS planning). A STRIPS planning task can be defined as a tuple \( P = \{ F, A, I, G \} \), where \( F \) is a set of instantiated predicates, \( A \) is a set of instantiated actions, and \( I \subseteq F \) and \( G \subseteq F \) are the initial state and the set of goals, respectively. Each action \( a \in A \) is represented as three sets: \( \text{pre}(a) \subseteq F \) (preconditions), list of predicates that need to be true for the application of the action; \( \text{add}(a) \subseteq F \) (adds), list of predicates made true by the application of \( a \); and \( \text{del}(a) \subseteq F \) (deletes), list of predicates that are no longer true after the application of \( a \). A plan \( \pi = \{ a_1, a_2, \ldots, a_n \} \) is an ordered set of actions \( a_i \in A \) that achieves a goal state from the initial state \( I \).

On the other hand, AR systems try to infer the plan that a user performs from sensor raw data. So, this work tries to narrow the gap between AP and AR by building AP domains using real sensor data with AR methods. Our system tries to automatically learn the preconditions, adds and deletes of each action using AR in order to build a planning action model.

Figure 1 shows an example of one action learned in PDDL. It describes the action \( \text{pick-up}(\text{param1}, \text{param2}) \) which has two parameters. It only has one precondition, \( \text{in}(\text{param2}, \text{param1}) \). Its effects are that \( \text{in}(\text{param2}, \text{param1}) \) is no longer true and \( \text{holding}(\text{param1}) \) becomes true. Actions in the PDDL language are parametrized while most current planners instantiate the actions with problem objects to obtain the set of instantiated actions \( A \). The learning system learns the parametrized version, because it is more general and applicable to all problems in a given domain.

3. ASRA-AMLA System Description

ASRA-AMLA is composed of two modules. The first one, ASRA (Action and States Recognition Algorithm), extracts actions and states from the sensor readings creating a sequence of interleaved actions and states. The second one, AMLA (Action Model Learning Algorithm), builds an AP domain from the sequences generated by ASRA. Both modules are described in the next sections. Figure 2 shows a schema of the architecture of the whole system.

3.1. ASRA: Action and States Recognition Algorithm

The objective of ASRA-AMLA is to learn a planning action model, an AP domain in PDDL; so, in order to accomplish such a task, the ASRA module recognizes the actions the user performs and also provides enough information to learn the preconditions and effects of each action recognized by the system. It recognizes the state of the environment before and after every action. In order to recognize states, the sensors should detect the changes that the actions produce. For instance, an accelerometer in an arm can provide information to recognize certain activities [10] (e.g., peeling), but it does not provide information about the effects of those activities (e.g., potato is peeled). So, in the setup three types of sensors are used: magnetic sensors, RFIDs, and cameras. Those sensors are going to gather information while the user cooks a recipe in a kitchen where the sensors have been installed. The magnetic sensors provide information about the state of the furniture of the kitchen; this way, they capture the effects that the actions of the user have on the furniture. The RFIDs are used to detect the objects that the user is using. The cameras are used to detect the location and state of the objects and appliances. Thus, the cameras detect the effects that the actions of the user cause on the objects. For instance, when the user opens a drawer to pick a fork up and leaves it on the kitchen top, the magnetic sensors detect that the drawer has been opened and closed, RFIDs detect that the fork has been picked up and put down, and the cameras detect that the fork has been left on the kitchen top.

Once the sensors are able to detect the effects of the actions through the states, the ASRA module defines what an event is to be used for segmentation. The segmentation
method based on events presented in [9] is used instead of the mainly used method based on temporal sliding windows.

When the user performs an action, it produces some changes in the environment and those changes may generate none or several changes in the sensor readings. For example, when a user picks an object up, the object changes its location. Thus, a RFID sensor placed near the hand of the user may change its reading from not detecting anything to detecting the object. We are interested in detecting those kinds of changes in the sensor readings since they are connected with the effects of the action pick-up. An event is defined as any change in the readings of any sensor. For example, when the user opens a cabinet, a RFID detects the cabinet and generates an event. Also, the magnetic sensor changes its value from closed to open.

The events are used for the segmentation of the sensor time series to extract the actions and states. When an event is detected, the system recognizes the action to which it belongs and its effects. If the action produces more than one effect, it may also produce more than one event. Also, one event may be produced by more than one action, even for actions we are not interested in.

For magnetic sensors or RFIDs, the events are easy to define since they provide discrete values. For both sensors, all the changes in the values of the readings they report will be considered as an event. For cameras, the detection of the events is not so direct since they provide richer information, and such information may not be direct. They are used to detect the state of some appliances (on and off) and objects (e.g., cracked for an egg) and the location of some objects (e.g., fry-pan on the burner). So, cameras will report in every frame the state of the appliances that they are monitoring, the objects that they can detect, and the state of those objects. Hence, the changes in the location or state of any object detected by the cameras are considered as an event.

Once the events have been defined, the next step is to build a classifier that predicts the action the user has performed. The action recognition task is formally defined as follows. Suppose the goal is to learn a recognition function \( f \), such that being \( S \) a set of \( N \) sensors, and \( A \) the set of actions performed by an agent, \( f \) determines the action that the agent performs from the current and past sensor readings. That is, the objective is to learn the function \( f : S^m \rightarrow A \) where \( m \) is the length of the time series. The training data will be composed of a set of sensor readings over time. Time points, selected by the events defined in the previous step, are labeled with the action that the agent performed at that time point. Each sensor \( s \) generates a sequence of readings that can be represented as a vector \( X = \{x_0, \ldots, x_j, \ldots\} \), where \( x_j \) is the reading at time \( i \) of sensor \( s \). A first task consists of defining a function \( f_1 \) that takes the \( N \) sequences and produces \( Z \) vectors of features \( F_j \). Each vector is labeled with the action \( a \) that the user performed at time \( i \) when the features in \( F_j \) were extracted. The second task learns a function \( f_2 \) that takes as inputs those vectors \( F_j \) generated by \( f_1 \) and builds a classifier to infer the activities performed. So, \( f_1 \) is a function that creates training instances from sequences of raw data and \( f_2 \) is a function that builds a classifier able to predict the actions performed using those instances.

The function \( f_1 \) can be formulated as follows. Given \( N \) sequences of sensor readings as above, \( X^s \), a sequence of events \( E = \{e_0, e_1, \ldots, e_j, \ldots\} \), detected at time steps \( T = \{t_0, t_1, \ldots, t_j, \ldots\} \) where \( t_j \) is the timestamp when event \( e_j \) was generated. Then, the features \( F_j \) are extracted and labeled with the action \( a_j \in A \) that produced the event \( e_j \). As features we use the values of the RFIDs and magnetic sensors, and the location and state of the objects captured with the cameras. These features are further described in Section 4.

After learning the action classifier, the states recognition is performed. When activities are recognized, sensors readings are segmented in pieces of data called temporal windows. These temporal windows do not usually match a complete activity from the beginning to the end of the activity. Instead, they normally split an activity into pieces that contain data that belongs to some parts of the activity. Actions have the same problem. Thus, in order to recognize complete actions, we assign to each temporal window an action. Next, we group consecutive temporal windows classified as the same action as an entire action. Then, the states between two different entire actions are the states extracted by the system.

Algorithm 1 shows how ASRA works. \( S \) contains the states that the module extracts and \( A \) the actions. InitialState() obtains the initial state given to the ASRA module and the sensor readings at start. EventDetector() is a function that returns the next events of the sensors time series and ActionClassifier() is the function built by \( f_2 \). Each event is classified and when it belongs to a new class the past action and state are saved in \( A \) and \( S \). Finally, \( A \) and \( S \) are merged to generate a sequence of interleaved states and actions (state-action-state).

As an example, suppose the user performs three consecutive actions, \( a, b, \) and \( c \), that generate the following sequence of events detected by the system \( e_1, e_2, e_3, e_4, e_5, e_6 \) where \( a \) generates the events \( e_1, e_2, e_3, e_4, e_5 \) and \( c \) generates \( e_6 \). If the initial state is \( s_0 \), the sequence
Algorithm 1: Action and States Recognition Algorithm (ASRA).

of actions and states detected by the action classifier is $s_1-e_1-s_2-e_2-s_3-e_3-s_4-e_4-s_5-e_5-s_6-e_6-s_f$, and the events could be classified as $s_1-a_1-s_2-a_2-s_3-a_3-s_4-a_4-s_5-a_5-s_6-a_6-s_f$; then, consecutive events classified as belonging to the same action are grouped producing the sequence $s_1-a_s-s_2-b_s-c_s-s_f$. This sequence is the input for the AMLA module.

3.2. AMLA: Action Model Learning Algorithm. This module builds an AP domain from the sequences generated by the ASRA module. These sequences are traces of an user performing an activity. AP domains generally use predicate logic for representation, so the states obtained from sensor readings have to be translated into predicates. We have created a mapping from low-level features (sensor readings) to predicates. This mapping is applied to sensor readings to automatically obtain corresponding predicate formulae. The information that sensors provide is related to the location and the state of the objects (e.g., open cupboard, fry-pan on the burner). The state of the objects is represented with a predicate with the same name as the fact in the state and one parameter, the object; for example, open(armchair), broken(egg). For the location of objects it is used the predicate in(param1,param2) where param1 is an object that is in the place given by param2. RFID tags are used to detect the objects that are lying, so they generate the predicates holding (obj,n) when a RFID detects the object obj,n.

AMLAA is given the following inputs for learning: (1) object types and generic predicates defined manually, (2) state-action sequences automatically generated by ASRA, and (3) the objects involved in each action (parameters of the actions) in the sequences defined by hand. Table 1 shows an example of some of the object types, generic predicates, and actions defined. It shows two predicates: raw that has one argument and in that has two arguments. The types of the arguments are also defined. The table shows some of the types. The predicate

| Table 1: Example of types and predicates used. |
|-----------------------------------------------|
| **Types**                                     |
| surface furniture—static object               |
| drawer1 fridge—furniture                      |
| ingredient utensil—moveable                  |
| oil water salt egg—ingredient                 |
| open close off raw cracked—state              |
| **Predicates**                                |
| raw(x—ingredient) in(x—static object, y—moveable) |
| pick-up(x—moveable, y—surface)               |
| open(x—furniture)                            |

in can be instantiated as in(fridge, egg) since fridge is of type furniture (a subtype of static object) and egg is of type ingredient (a subtype of moveable). Finally, Table 1 shows the action open, which has one parameter, and the action pick-up that has two parameters. The types of the parameters are also indicated.

Also, some standard assumptions are made. First, it is assumed that actions are deterministic. That means that it is considered that the actions are going to have always the same effects. Second, it is assumed that the preconditions are conjunctive; all the preconditions have to be true for the action to be executed. Also, a threshold (error rate) is used to eliminate the sensors’ noise. This threshold indicates a minimum percentage of times that a predicate has to appear before/after an action to be used for building the corresponding action model. The goal is to learn the preconditions and effects (adds and deletes) of the actions. This threshold allows us to deal with situations in which the action fails (i.e., the effects are not correct); or the sensors fail (i.e., the preconditions or effects are not correct).

Thus, in order to learn the preconditions of each operator, given a collection of predicates that represent the state before the action, $s_{in}$, the module removes all the predicates where the parameters of the operator are not present. It also removes all the predicates but those that appear in the previous state a percentage of times greater than or equal to the threshold. For example, suppose we want to learn the operator $a$, which has 10 instances in the input sequence, and we had set a threshold of 70%. Any predicate containing any parameter of the action, which is present at the previous state of the action $a$ at least 7 out of the 10 times, would be included as a precondition for that action. Effects are learned using the difference between the state before the action $s_{in}$ and the state after it $s_{oi}$. Thus, the adds list of the operator $a$ is $add(a) = s_{in} \setminus s_{oi}$ (set difference of state after $a$ and state before $a$) and the deletes list is $del(a) = s_{oi} \setminus s_{in}$.

The threshold is also used for the effects in the same way as for the preconditions. To be part of the effects, a predicate has to appear in the effects a percentage of times greater than or equal to the threshold. So, in the previous example, a predicate has to be in the adds of $a$ in at least 7 out of the 10 instances of the action to be part of it. The same is used for the deletes.
4. Experimental Setup and Results

In order to test the system, a dataset has been generated with the task of making an omelette and the actions that compose this activity. The actions are open, close, get-out, put-away, pick-up, put-down, crack-egg, transfer, switch-on, switch-off, fry, beat, and null. Null is assigned to the events not involved in any action. Activities performed in a kitchen have been studied and included in many works in the past like in [2, 11] and some of them are in public datasets like [2]. However, none of the public datasets are suitable to test the system, since they do not provide enough information to classify the actions that compose the activities stored in the datasets. For instance, in the case of [2], the data is labeled with the name of the activities, Making Coffee or Setting the table; but not the labels of the actions that compose those activities, switch on the coffee maker or put down a plate; which is what we need.

The AMLA module uses the previous and next state to the action to build the action model through the changes detected in the sensors’ readings. For that reason we have designed a sensor network focused on getting that information. The sensor network is composed of several types of sensors. We have used magnetic sensors, RFID sensors, and cameras. Magnetic sensors have been used to monitor the state (opened or closed) of the cupboards, drawers, and the fridge of the kitchen. Two RFID receivers are used to track the objects that the user holds in both hands through two RFID gloves similar to the one used in [12]. The RFID readers employed operate at 125 kHz frequency and the antenna attached to the glove detects tags within 2–10 centimeters of its center. The glove was developed just like in [12]. 125 kHz tags are not affected by water or metals, what makes them very appropriate for the environment in which we used them. Finally, we use four cameras to detect the state and location of the objects in the kitchen. The objects used are: a bowl, forks, plates, a fry-pan, an oil bottle, and a salter. All of them have RFID tags attached to be detected. The ingredients are eggs, salt, and oil. Also, there is a cooktop as appliance, the kitchen top as working surface, and a sink. Also, we have attached RFID tags in all the cabinets, and drawers pulls to gather information about the location of the user.

Figure 3 shows three pictures. The first one in Figure 3(a) shows the kitchen where the user cooks. Figure 3(b) shows the view of one of the cameras. In this case, the camera focused on the sink and part of the kitchen top where some objects have been placed. Finally, Figure 3(c) shows the view of a camera focused on the cooktop. Also, this picture shows one of the RFID gloves used by the user and some objects with RFID tags.

We used the OpenCV library [13] to recognize the objects. First, our system removes the background using an initial photo. Then, part of the shadows of the objects are removed using the initial photo but darkened. Finally, color and shapes are used as features to recognize the objects and their states using a classifier. Each object has a different color to facilitate the recognition. Before recording the dataset, the computer vision system was trained taking photos of all the objects of the kitchen in many situations. In each photo, the background is removed and the color and shape are extracted using contours. This way, a classifier was generated. Some of algorithms provided by OpenCV were tested: Support Vector Machines (SVM) [14], k-nearest neighbor (Ibk) [15], and Random Forests (RF) [16]. The best results were achieved using SVM, which obtained a recognition rate of 98% of the objects. Inside some objects some ingredients can be found in different states: oil in the fry-pan, raw egg in the bowl, beaten egg in the bowl, beaten egg in the fry-pan, omelette in the fry-pan, and omelette on the plate. Different classifiers were developed to detect each of these situations. So, first, the system detects the object using the main classifier. If, for example, the fry-pan is detected, a second classifier is used to detect if the fry-pan is empty, with oil, with the beaten egg, or with the omelette. A third classifier was generated to detect the different states of the bowl. The recognition rates were 63% for the fry-pan classifier, and 71% for the bowl classifier. In order to avoid problems caused by occlusion, there is more than one camera focusing on the working surface. Also, no changes are reported by the cameras until detecting an object and its state. So, if the user hides an object during some seconds, the cameras will not report any information about that object, and the system will consider that the object is in the same location and state previous to be hidden. Cameras work together with the RFID, so, whenever an object is detected by a RFID, the cameras try to find that object. The system also searches for specific information in specific places (e.g., the eggs just can be fried in the fry-pan or beaten in the bowl). To determine the state of the appliances we use the area of...
the images where a display indicates the state of the appliance. Then, using the difference of the colors in that area of the images, the state is computed. Finally, the initial state of some elements is defined since the system does not have sensors to detect such information, like objects that are initially inside some cupboard or inside the fridge. In addition, we have attached some RFID tags to the cupboards, drawer, fridge and surfaces in order to have some information about the place where the user is.

We tested our system through the task of making an omelette. The task has been performed by two different users ten times each, from beginning to end. The task was performed by two users to avoid the possible bias caused by a single user. The task was performed ten times by each user in order to obtain enough instances of each action to test the models. Also, parts of the task have been recorded in order to get more instances of some of the actions that compose the task. For instance, beating the eggs and putting the mix in a fry-pan or picking an egg up from the fridge and cracking it.

4.1. asra: Experiment Description. In order to test the asra module, we considered two different models. In the first experiment, called SingleValues, we generated a vector where all its positions except for the last one are binary. The positions represent the values of each sensor that appeared in the dataset. So, there is a position for each magnetic sensor and a position for each of the objects that the RFID readers can detect. Also, there is a position for every location of each object detected with the cameras (the hand of the user is not considered a location, because it is detected with RFIDs) and also a position for every different state in which an ingredient can be. Finally, there is a position for each RFID tag placed in a piece of furniture. A last element is included, indicating the index of the last position that changed. The generated vector contained 141 elements. It was computed as (10 places) + (10 places * 9 objects) + (2 RFID * 9 objects) + (2 eggs * 4 states) + (1 salt * 2 states) + (1 oil * 2 states) + (10 RFID places) + 1 = 141 elements. But some of the elements never change. For instance, the fry-pan or the eggs are never placed in the drawers. So, those never-change values were removed and the final vector used for the classification contained 59 positions. So, the inputs of this experiment are instances composed of 59 components: the first 57 positions contain binary values; the 58th position has a value between 1 and 57 to indicate the last position that changed; and the last position contains the class, the executed action. The outputs of the experiment are the actions executed at each event. For example, consider a magnetic sensor placed in a cabinet, one RFID reader and two objects (e.g., two eggs) that can be in two states (e.g., raw and cracked) in an environment with two locations to place the object (e.g., a cabinet and a working surface) and n cameras. Then, the vector generated for the event that was detected by a camera when the object was dropped on the working surface would be \( \langle 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 5, 7 \rangle \). In the example, the first element would represent the value of the magnetic sensor (cabinet closed), the second and third positions indicate the location of one egg, and the next two positions are the location of the other egg. The next three elements, the sixth, seventh and eighth, represent if the RFID detects the object (no object is detected) or the cabinet. In the example, both eggs are in the second location (working surface) and the cabinet is not detected. The ninth and tenth elements indicate the state of one egg and the next two elements indicate the state of the other egg. Both eggs are raw. Finally, the last position indicates that the fifth element of the vector was the last one that changed. So, the second egg was put-down on the kitchen top.

The second experiment, called ObjectInvolved, uses just the objects involved in the event that created the instance. So, whenever an event is detected, the sensors readings related to the objects involved in the event are kept and the rest are discarded, set to zero. Thus, when a specific object is picked up, a vector is generated like in the SingleValues experiment, but the rest of information in the vector about other objects is set to zero. For example, when a plate is put down on the kitchen top, an event is generated when the RFID stops detecting the plate. Then, the objects involved are the plate, which changes from being hold to be on the kitchen top, the kitchen top, and the RFID that changed from detecting the plate to detecting nothing. The kitchen top never changes, so none of the positions have specific information about it. So, the vector would contain a "1" in the position of the vector that indicates that the plate is on the kitchen top. When a RFID detects nothing, all the positions of the vector that contain information about that RFID are set to zero. Thus, the vector generated would contain just the "1" that indicates that the plate is on the kitchen top. Therefore, we focus the learning process on the objects currently used, eliminating the information about the rest of them. Similarly, planning operators have parameters where the objects affected by the operator are defined. We try to use the same information that the planning operator would contain in order to recognize the action associated to the operator. Also, we have added information about the previous event.

We have included an attribute to indicate the index of the position that was changed by the previous event. So, the vector is similar to the one generated for the SingleValues experiment but for the penultimate element, which indicates the element that was changed by the previous event. The machine learning algorithm could use the position that changed in the past event and in the current event to classify the actions. So, the inputs of the second experiment are instances composed of 60 positions: the first 57 positions contain binary values; the 58th position has a value between 1 and 57 to indicate the last position that changed; and the last position contains the class, the executed action. The outputs are the same as in the first experiment; the actions executed at each event. For example, given the previous example, the generated vector would be \( \langle 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 5, 7 \rangle \). The differences in the third and ninth positions are due to the fact that those elements do not represent information about the object involved in the current event. The last position indicates that the element that changed in the previous event was the seventh.
After generating all the instances, we used the following classifiers to learn the model: PART [17], C4.5 [18], Ibk, SVM, RF, and a Bayesian Network (BN). We have used their implementation in the Weka toolkit [19]. These classifiers have been selected because they represent different machine learning strategies and cover a wide spectrum of learning techniques.

4.2. ASRA: Experimental Results. In summary, we have performed experiments to check the performance of the ASRA-AML module. The dataset generated for testing was composed of 1720 instances. Both experiments generated the same number of instances since the segmentation method employed was the same in both cases. We have used Kappa [20] coefficient of agreement for evaluation and comparison using Dietterich's 5 × 2-Fold Cross-Validation Paired t-Test [21] for estimating the error. Kappa coefficient is a criterion classically used in classification to measure the degree of agreement [22] and takes into account the correct classification that may have been obtained "by chance" by weighting the measured accuracies.

Then, to build an AP domain we selected the best classifier according to the employed metric. Table 2 shows the kappa coefficient of every generated model and the time in seconds that took to classify all the instances of the dataset employing the models. The best model is marked in bold.

As it can be seen, the differences between ObjectInvolved and SingleValues are not significant in most models. Results obtained by the ObjectInvolved experiment are more consistent, since all the results are over 0.90. So, focusing the data just on the objects involved in the last event permits the system a better recognition of actions in average, although the difference in the results achieved by most algorithms is very small. In any case, the model that obtained the best result, RF, is the one we selected to generate the action sequences used to build the planning action model. In addition, the chosen model is one of the fastest models tested, although it is not the fastest. It is important to consider the time performances because it can prevent the system to be used in real-time. In any case, the showed times belong to the classification of all the instances of the dataset. The classification time of a single instance is lower than 0.0001 s. So, all the models generated are fast enough to be used in the system.

4.3. AMLA: Experiment Description. After generating sequences of actions and states with ASRA, the AMLA module is executed to generate an AP domain using all the sequences. Then, the domain is tested to check if it is able to generate valid plans and provide assistance for users to complete a task. So, in order to test AMLA, some models were generated using different thresholds. First, a threshold of 100% was used. Then, the threshold was iteratively decreased by a 5%, until it takes the value of 0%. Thus, the system obtains the values of the threshold between which each action is learned correctly. A hand-written planning domain was created with all the actions the system has to learn automatically. To test each learned action we substitute a hand-written action by its learned equivalent. Then we check if using the new domain with the learned action we are able to solve problems. We try to solve 10 problems that require the action and consider that it is learned correctly when the 10 are solved.

We applied a twofold cross-validation by dividing the dataset into two parts of equal size. We selected one part to generate the planning domain and used the other part to test the domain. In the testing part of the domain, we randomly selected 10 sequences of actions for each planning operator where the first action that need to be executed was the action to be tested. For example, we used a planning problem to check the action pick-up where there is an object in a working surface that has to be picked up to achieve the goal, to cook an omelette. So, first, the user has to pick-up the object and then use the rest of actions to achieve the goal. This way, the problems generated are different since the initial state of each problem is different. The problems employed to check the learned actions are sequences found in the parts of the dataset used for testing.

The AP problems have been executed using a deterministic planner from the International Planning Competition (IPC).

In Section 4.4 the results are presented.

4.4. AMLA: Experimental Results. The best model for ASRA was selected and used to generate the sequences of actions and states that feed the AMLA module. Then, the AMLA module was used to generate AP domains. Table 3 shows the interval of values of the threshold in which each operator is learned correctly along with the precision obtained by each class. To detect when an operator is learned correctly we generated AP problems in which the learned action had to be applied in the first place followed by another action. These problems were extracted from the dataset. Table 3 shows the obtained results.

Table 3 shows that not all the actions were learned correctly using the same threshold. Some actions like pick-up or beat admit the lowest value of 70%. However, put-away has the upper value of 50%. So, if we want our system to learn the first two actions, pick-up and beat, put-away is not learned correctly and vice versa. With a threshold between 75% and 70% all actions but one are learned correctly. That action is put-away. This action is the one that the user takes when placing something in the cabinets or drawers. This action is not learned correctly because the action is very similar to put-down. The action put-down is taken when the user leaves something on the kitchen top. So, the readings of the sensors are very similar and some of the events produced during
the actions are identical. For that reason, the precision of the recognition of this action is one of the lowest. As we can see, transfer has even lower precision, though it can be learned correctly. This is due to the fact that the precision of transfer is affected mainly by sensor noise and sometimes it is captured correctly. However, the precision of put-away is affected by the fact that another action is very similar and both produce identical events which are classified in most of cases as put-down. So, with less precision, the threshold permits the system to absorb the sensor’s noise and learn the action transfer. But also, to learn put-away correctly the threshold has to be lowered too much to allow amla to learn other actions correctly.

The results of the learning process change as the threshold is modified like in similar approaches [23, 24]. Using a different threshold for each activity solves the problem and permits us to learn the complete planning action model correctly. However, the drawback is that it requires to provide more inputs for the system to work and that is what we want to reduce.

Figure 4 shows an example of a plan generated by a domain created by amla. As it is shown, the plan provides all the steps to cook an omelette. However, the plan has some minor problems. It closes the fridge at the end of the plan, leaving the door opened almost since the beginning. Also, it leaves the door of the cabinets opened. It can make it for the user impossible to cook.

5. Related Work

Many systems can be found in the literature using different types of sensors and detecting very different types of activities like for example [2–6]. But just a few of them deal with what we are calling actions; subtasks that compose an activity or plan. In [25] Amft et al. split activities, that they call composite activities, into actions, that they call atomic activities. First, they recognize the actions through what they call detection, into actions, that they call atomic activities. Then, they only learn the action effects, that are the transition rules among states. They build on their previous work [29] where the method was only applied to fully observable domains. The LOCM system [31] automatically induces action schema from sets of example plans. It does not have to be given any information about predicates, initial goals, or intermediate state descriptions. The example plans are a sound sequence of actions. LOCM exploited the assumption that actions change the state of objects, and require objects to be in a certain state before the actions can be executed. Planning traces are the input of LOCM, where each action is identified by its name and the objects that are affected or are necessarily present but not affected by the action execution.

| Thresholds | Open | Close | Pick-up | Put-down | Get-out | Put-away | Crack-egg | Transfer | Switch-on | Switch-off | Fry | Beat |
|------------|------|-------|---------|----------|---------|----------|-----------|----------|-----------|------------|-----|------|
| Precision  | 100% | 100%  | 97.7%   | 92.3%    | 100%    | 65%      | 100%      | 100%     | 100%      | 100%       | 75% | 66.7% |

Table 3: Precision and the values of the threshold for which the operators are learned correctly by amla.
Opmaker2 [32, 33] takes as inputs a domain ontology and a solution to a problem and automatically constructs operator schema and planning heuristics from the training session. It requires only one example of each operator schema that it learns and an ontology of objects and classes (called a partial domain model) as input. Opmaker2 is an extension of the earlier Opmaker system [34].

The systems presented need the sequence of actions executed and that information must be precise to work properly. In our case, that information is provided by an AR system, so it may contain errors. So, the system we propose is able to deal with the sensors’ noise like in [23, 28] but also with the errors generated by the AR system and it is capable of generating an action model.

Hoey et al. in [11] present the only work available capable of learning a human action model from sensors. They map the actions and the states directly from the sensors, except for what they called “virtual sensor.” This “virtual sensor” uses an AR method to recognize one of the actions, pouring. Once they have mapped the actions and the states, they build a Partially Observable Markov Decision Process (POMDP) to assist people. Our system is similar but we generate action models based on AP domains instead of a POMDP. A POMDP does not scale well in general and the modification of the domain is more complicated than in the AP domains. Using AP permits us to exploit the generality and power of AP algorithms. Also, the way the action model is learned is very different.

6. Conclusions

In this paper, we present a system for building AP domains from raw sensor data. The AP domains generated can be used, along with a planner, to find a sequence of actions that can be used to assist people in the environment where it was built. The system called ASRA-AMLA is composed of two modules that can recognize actions and states from raw sensor data (first module) and use the sequence of actions and states generated by the first module to build an AP domain (second module).

In the process of building the system, a working environment was created in a kitchen. Two persons performed a task, making an omelette, in the kitchen, and a sensor network recorded the data produced by the users performing the task. An AP domain was modeled to assist people in the task. However, using just one threshold does not permit to learn the entire domain. To model the AP domain successfully, more than one threshold has to be used. An alternative method would be to improve the sensor network in order to recognize better the actions. Anyway, the action that is not learned correctly does not prevent the system from guiding a person in the learned task because it is used to insert objects inside the furniture.

The sensor network used has some limitations. The user must use RFID gloves in order to recognize the object that he/she picks up. Also, different objects cannot have similar colors. Both limitations could be solved using accelerometers like in [11, 25] to detect the object that is being used. That way, the system could infer the actions performed by each object as well as its locations and state.

The system is easily extendible. In order to include more recipes or actions, the sensorial system has to be extended to recognize the new actions and to detect the effects that the new actions would produce. That way, the new actions would be learned and included in the planning action model.

As future work, it is planned to learn the entire system automatically using unsupervised methods or semisupervised methods. This would minimize the information provided to the system to generate models. Also, the system will be extended to include more tasks and to use the AP domain not just to assist people but to recognize the actions that the user performs as in [8].

Acknowledgment

This work has been partially sponsored by the Ministry of Education and Science Project nos. TIN2011-27652-C03-02, TIN2008-06701-C03-03, and TIN2012-38079-C03-02.

References

[1] M. P. Lawton and E. M. Brody, “Assessment of older people: self-maintaining and instrumental activities of daily living,” The Gerontologist, vol. 9, no. 3, pp. 179–186, 1969.
[2] D. J. Patterson, D. Fox, H. Kautz, and M. Philipose, “Fine-grained activity recognition by aggregating abstract object usage,” in Proceedings of the 9th IEEE International Symposium on Wearable Computers (ISWC ’05), pp. 44–51, October 2005.
[3] B. Logan, J. Healey, M. Philipose, E. M. Tapia, and S. Intille, “A long-term evaluation of sensing modalities for activity recognition,” in Proceedings of the 9th International Conference on Ubiquitous Computing, vol. 4717 of Lecture Notes in Computer Science, pp. 483–500, 2007.
[4] T. van Kasteren, A. Noulas, G. Englebienne, and B. Kröse, “Accurate activity recognition in a home setting,” in Proceedings of the 10th International Conference on Ubiquitous Computing (UbiComp ’08), pp. 1–9, ACM, Seoul, South Korea, September 2008.
[5] T. L. van Kasteren, G. Englebienne, and B. J. Kröse, “Hierarchical activity recognition using automatically clustered actions,” in Proceedings of the 2nd International Conference on Ambient Intelligence, pp. 82–91, Springer, 2011.
[6] S. Dernbach, B. Das, N. C. Krishnan, B. L. Thomas, and D. J. Cook, ”Simple and complex activity recognition through smart phones,” in Proceedings of the 8th International Conference on Intelligent Environments (IE’12), pp. 214–221, IEEE, 2012.
[7] D. McDermott, “The 1998 AI planning systems competition,” AI Magazine, vol. 21, no. 2, pp. 35–55, 2000.
[8] M. Ramirez and H. Geffner, “Plan recognition as planning,” in Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI ’09), pp. 1778–1783, Morgan Kaufmann, 2009.
[9] J. Ortiz, A. Garcia-Olaya, and D. Borrajo, “A dynamic sliding window approach for activity recognition,” in Proceedings of the 19th International Conference User Modeling, Adaptation and Personalization (UMAP ’11), vol. 6787, pp. 219–230, Springer, 2011.
[10] C. Pham and P. Olivier, “Slice&Dice: recognizing food preparation activities using embedded accelerometers,” in *Ambient Intelligence*, pp. 34–43, 2009.

[11] J. Hoey, T. Plötz, D. Jackson, A. Monk, C. Pham, and P. Olivier, “Rapid specification and automated generation of prompting systems to assist people with dementia,” *Pervasive and Mobile Computing*, vol. 7, no. 3, pp. 299–318, 2011.

[12] Y. Medynskiy, S. Gov, A. Mazalek, and D. Minnen, “Wearable RFID for play,” in *Proceedings of the Intelligent User Interfaces Tangible Play Workshop*, 2007.

[13] G. Bradski, “The openCV library,” *Dr. Dobb’s Journal of Software Tools*, vol. 25, no. 11, 2000.

[14] J. Platt, “Fast training of support vector machines using sequential minimal optimization,” in *Advances in Kernel Methods: Support Vector Learning*, B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds., chapter advances in kernel methods, pp. 185–208, MIT Press, Cambridge, MA, USA, 1999.

[15] D. W. Aha, D. Kibler, and M. K. Albert, “Instance-based learning algorithms,” *Machine Learning*, vol. 6, no. 1, pp. 37–66, 1991.

[16] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[17] E. Frank and I. Witten, “Generating accurate rule sets without global optimization,” in *Proceedings of the 5th International Conference on Machine Learning*, pp. 144–151, Citeseer, 1998.

[18] J. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, Boston, Mass, USA, 2003.

[19] I. H. Witten, E. Frank, L. Trigg, M. Hall, G. Holmes, and S. J. Cunningham, *Weka: Practical Machine Learning Tools and Techniques with Java Implementations*, 1999.

[20] J. Cohen, “Weighted kappa: nominal scale agreement provision for scaled disagreement or partial credit,” *Psychological Bulletin*, vol. 70, no. 4, pp. 213–220, 1968.

[21] T. G. Dietterich, “Approximate statistical tests for comparing supervised classification learning algorithms,” *Neural Computation*, vol. 10, no. 7, pp. 1895–1923, 1998.

[22] J. A. Richards and X. Jia, *Remote Sensing Digital Image Analysis: An Introduction*, Springer, New York, NY, USA, 1999.

[23] Q. Yang, K. Wu, and Y. Jiang, “Learning action models from plan examples with incomplete knowledge,” in *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS ’05)*, pp. 241–250, Monterey, Calif, USA, 2005.

[24] H. Zhuo, Q. Yang, D. Hu, and L. Li, “Transferring knowledge from another domain for learning action models,” in *Proceedings of the 10th Pacific Rim International Conference on Artificial Intelligence (PRICAI ’08)*, pp. 110–1115, 2008.

[25] O. Amft, C. Lombriser, T. Stiefmeier, and G. Tröster, “Recognition of user activity sequences using distributed event detection,” in *Proceedings of the 2nd European Conference on Smart Sensing and Context (EuroSSC ’07)*, vol. 4793 of Lecture Notes in Computer Science, pp. 126–141, Springer, October 2007.

[26] H. Junker, O. Amft, P. Lukowicz, and G. Tröster, “Gesture spotting with body-worn inertial sensors to detect user activities,” *Pattern Recognition*, vol. 41, no. 6, pp. 2010–2024, 2008.

[27] X. Wang, “Planning while learning operators,” in *Proceedings of the 3rd International Conference on AI Planning Systems*, p. 188, 1996.

[28] K. Mourao, R. Petrick, and M. Steedman, “Learning action effects in partially observable domains,” in *Proceedings of the ICAPS Workshop on Planning and Learning*, pp. 15–22, 2009.

[29] K. Mourao, R. P. Petrick, and M. Steedman, “Using kernel perceptrons to learn action effects for planning,” in *Proceedings of the International Conference on Cognitive Systems (CogSys ’08)*, pp. 45–50, 2008.

[30] E. Amir and A. Chang, “Learning partially observable deterministic action models,” *Journal of Artificial Intelligence Research*, vol. 33, no. 1, pp. 349–402, 2008.

[31] S. N. Cresswell, T. L. McCluskey, and M. M. West, “Acquisition of object-centred domain models from planning examples,” in *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS ’09)*, pp. 338–341, September 2009.

[32] T. McCluskey, S. Cresswell, N. Richardson, and M. M. West, “Opmaker2: efficient action schema acquisition,” in *Proceedings of the 26th Workshop of the UK PLANNING AND SCHEDULING Special Interest Group (PlanSIG ’07)*, December 2007.

[33] T. L. McCluskey, S. N. Cresswell, N. E. Richardson, and M. M. West, “Automated acquisition of action knowledge,” in *Proceedings of the 1st International Conference on Agents and Artificial Intelligence (ICAART ’09)*, pp. 93–100, January 2009.

[34] T. L. McCluskey, N. E. Richardson, and R. M. Simpson, “An interactive method for inducing operator descriptions,” in *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems*, AAAI Press, 2002.
Submit your manuscripts at http://www.hindawi.com